Degeneracy and the [[17,1,5]] Code

Rebecca Golovanov

July 2023

Contents

1 Group Theory and Stabilizer Formalism

1.1 Explaining Group Theory Using Modular Arithmetic

This section is a quick crash course on the group theory necessary to understand stabilizer codes. I'm going to use the integers mod 3, ($\mathbb{Z}/3\mathbb{Z}$), as my example of choice throughout, because it is probably the simplest. For reference,

$$\begin{aligned}\mathbb{Z} &\to \mathbb{Z}/3\mathbb{Z}\\ &\to \{0,1,2\}\\ x &\mapsto \text{remainder of } x/3\end{aligned}$$

Definition 1.1. $\mathbb{Z}/3\mathbb{Z}$ is a *group* under addition, meaning addition is the *group operation.* It is a group because it satisfies the following:

1. $\mathbb{Z}/3\mathbb{Z}$ has an identity, 0. For any number x, we know $0 + x = x + 0 = x$
2. If $a, b \in \mathbb{Z}/3\mathbb{Z}$, then $a + b$ in $\mathbb{Z}/3\mathbb{Z}$. We know $1, 2 \in \mathbb{Z}/3\mathbb{Z}$. Then we have $1 + 2 = 3$, which is 0 mod 3, and $0 \in \mathbb{Z}/3\mathbb{Z}$, so we're good.
3. Every element of $\mathbb{Z}/3\mathbb{Z}$ has an additive inverse. this just means we can subtract in $\mathbb{Z}/3\mathbb{Z}$.
4. Associativity: $a + (b + c) = (a + b) + c$

The Pauli group is a group under *matrix multiplication*, meaning it has all the properties above, but with matrix multiplication instead of addition. Note that commutativity is not required for the definition of a group. This is essential to how we use the Pauli group in error correction, since matrix algebra is not commutative. A group where all elements commute with each other is called an *abelian* group.

Definition 1.2. Given an additive group G and subgroup H, a *coset* of H is defined by $g + H$ for some $g \in G$. So, given the group of integers $\mathbb{Z}$ and the subgroup $3\mathbb{Z}$, the cosets of $3\mathbb{Z}$ are $1 + 3\mathbb{Z}$ and $2 + 3\mathbb{Z}$, which we represent by $1, 2$. This is because, given 5 for example, $5 = 3 + 2$ and 3 is in $3\mathbb{Z}$, so $5 + 3\mathbb{Z}$ simplifies to $2 + 3\mathbb{Z}$.

$\mathbb{Z}/3\mathbb{Z}$ is an example of what is called a *quotient group*, where we "quotient out" by $3\mathbb{Z}$. The group $3\mathbb{Z}$ is all multiples of 3, and we can think of the cosets as the remainders of the quotient group, which is why these cosets are 1 and 2.

Cosets also lend themselves to the definition of an *equivalence class.* We can also label these representatives as $[0], [1], [2]$, now referring to equivalence classes. Two numbers are in the same equivalence class if they're in the same coset. So, for example, $1, 4, 7, 16$ are all in the $[1]$ equivalence class, meaning they are all equivalent mod 3.

Definition 1.3. Let G and H be groups. A *group isomorphism* is an invertible function f from G to H, meaning there is some f^{-1} from H to G. Additionally, f preserves *group structure.* For example, let G be $\mathbb{Z}/4\mathbb{Z}$ with cosets $\{0, 1, 2, 3\}$, and let H be the group $\mathbb{Z}/5\mathbb{Z}$ under multiplication, denoted by $(\mathbb{Z}/5\mathbb{Z})^\times$ with cosets $\{1, 2, 3, 4\}$. There is no 0 because 0 doesn't have a multiplicative inverse modulo 5.

We can make a function f that will basically "match" the elements and group operation of G to that of H.

$$\begin{aligned} f : G &\to H \\ [0] &\mapsto 1 \\ [1] &\mapsto 2 \\ [2] = [1+1] &\mapsto 2 \times 2 = 4 \\ [3] = [2+1] &\mapsto 4 \times 2 \bmod 5 = 3 \\ + &\mapsto \times \end{aligned}$$

Through this, we can see for example:

$$\begin{aligned} f([1]+[2]) &= f[3] = 3 \\ f([1]) \times f([2]) &= 2 \times 4 = 3 \\ f([1]+[2]) &= f([1]) \times f([2]) \end{aligned}$$

The illustration above demonstrates the definition of preserving group structure. Given some a, b in a group $(G, +_G)$ and c, d in a group $(H, +_H)$, we must have

$$f(a +_G b) = f(a) +_H f(b)$$

This "matching" can also be reversed, meaning we can go from H to G. It is the ability to 'reverse' (invert) the function that makes f a *group isomorphism* and G and H *isomorphic*.

In group theory, if two groups are isomorphic, then they are for all intents and purposes equivalent. This is because for whatever calculations we do in group G, we can apply a function like f to 'translate' this work into the language of H. This property is used to allow us to map a harder problem onto a group which is easier to work with. We can figure out a solution in the easier space, and then invert the map to get back to the original group.

An isomorphism which is key to quantum error correction is called the Pauli to Binary Isomorphism, and it allows us to analyze Pauli errors in the language of classical error coding and parity check matrices.

Definition 1.4. The *stabilizer set* of a group G is the set of elements $s \in G$ such that $s \circ g = g \circ s = g$ for all $g \in G$. The identity (0 for addition, 1 for multiplication) of any group is an obvious stabilizer. Let's go back to $G = \mathbb{Z}/3\mathbb{Z}$. Obviously 0 is a stabilizers, but so are $3, 6, 9$, etc. The elements of $3\mathbb{Z}$ are stabilizers in G. If g is in the coset $1 + 3\mathbb{Z}$, then $g + 3, g + 6, g + 9$ are all in $1 + 3\mathbb{Z}$ as well. In this way, all multiples of 3 "act like" 0, which is how we usually think of modular arithmetic.

Definition 1.5. The *centralizer* of a group G is the set of elements c such that $cg = ga$ for all $g \in G$. Basically, any c will commute with any element of G. Since stabilizers trivially commute with all g, then the stabilizers are a subset of the centralizers.

When we encode qubits, we will see that the group of stabilizers corresponds to all possible basis states of a logical $|0\rangle$ qubit. As we saw with the $\mathbb{Z}/3\mathbb{Z}$ example, stabilizers "act like 0"/the identity, so it makes sense that the stabilizers of a code would correspond to logical 0.

If our code only encodes one qubit, then the set of centralizers which excludes the stabilizers corresponds to the basis states of a logical $|1\rangle$ qubit.

1.2 Pauli to Binary Isomorphism and Stabilizer Codes

The Pauli to Binary isomorphism, which I will denote by β, allows us to translate between Pauli error operators and discrete stabilizer codes that are more familiar to classical error correction. The Pauli group is generated by the I, X, Y, and Z operators. I is the identity, X is a bit-flip, Z is a phase flip, and Y is the combination of X and Z operators. The *N-fold Pauli group* refers to the group generated by the tensor products of Pauli operators up to degree N and is denoted by $\prod^{\otimes N}$. We denote the binary space of dimension $2N$ by $\mathbb{F}_2^{2N}$, which is basically a space of binary vectors with $2N$ entries. The isomorphism β is defined as

$$\begin{aligned} \beta: \quad \prod^{\otimes N} &\to \mathbb{F}_2^{2N} \\ I &\mapsto (0|0) \\ X &\mapsto (0|1) \\ Z &\mapsto (1|0) \\ Y &\mapsto (1|1) \end{aligned}$$

For example, take the string **ZXI**. We get

$$\mathbf{ZXI} \mapsto 100|010$$

because the beginning of the two-dimensional string is $1|0$, following by $0|1$ to get $10|01$, and finished with $0|0$ for the identity.

In order to preserve the non-abelian group structure of the Pauli group, we need to introduce an operation called the *symplectic product*.

Definition 1.6. Let g and h be binary representations of $N-$fold tensor products. The symplectic product is defined as follows:

$$g \odot h = (g_z \cdot h_x + g_x \cdot h_z) \bmod 2$$

where $\cdot$ represents the usual vector dot product.

We define this operation because it only evaluates to 0 if g and h commute, so for example all valid codewords (the centralizer group) should commute.

In a parity check matrix $\mathbf{H} = (H_z|H_x)$, the symplectic product looks like

$$H_z H_x^T + H_x H_z^T$$

and evaluates to 0 because the rows of H are the binary representation of the stabilizer generators. Given an $n-$qubit Pauli error P, we can define the syndrome

$$s := \mathbf{H} \odot P^T = (H_z P_x^T + H_x P_z^T) \bmod 2$$

If $s \neq 0$, then we have detected the error P, which is similar to classical correction.

Definition 1.7. We can define a parity check matrix $\mathbf{H}$ with the isomorphism β. *Stabilizer codes* are defined by their parity check matrix (PCM), where the rows are the binary symplectic form of the stabilizer generators.

A stabilizer code is denoted by

$$[[N, k, d]],$$

where N is the number of physical qubits, k is the number of encoded qubits, and d is the minimum distance. The distance between to encodings is the number of bit flips required to turn one into the other. As we alluded to earlier, the codespace of a stabilizer code is determined by the solutions to

$$\mathbf{H}x = 0,$$

meaning any string that commutes with the stabilizers; i.e. the centralizers. There are 2^k cosets of the stabilizer, so when we encode one qubit like in the Steane $[[7, 1, 3]]$ code, the cosets correspond to logical 0 or logical 1. With a larger k, we would have cosets corresponding to logical $010, 100, 110$, etcetera. The minimum distance is the minimum distance between cosets. For the $[[17, 1, 5]]$ code, if a is in the coset corresponding to $|0\rangle_L$ and b is in the coset corresponding to $|1\rangle_L$, then we know the minimum distance between a and b is 5. Since stabilizer codes are built from solutions to the PCM, we can decompose any error operator E into the following components

$$E = T \cdot L \cdot S$$

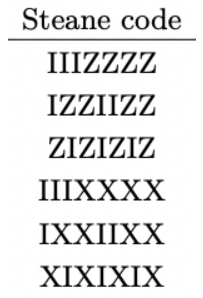
where T is the pure error, L is the logical operator, and S is the stabilizer operator.

We will now explicitly demonstrate what it means to encode a qubit using the stabilizers.

2 Encoding Qubits

We can use the explicit logical Steane encoding to understand how the [[17,1,5]] code goes about encoding information. We begin with encoding logical $|0\rangle_L$.

A. **Given the stabilizers:**



Steane code
IIIZZZZ
IZZIIZZ
ZIZIZIZ
IIIXXXX
IXXIIXX
XIXIXIX

1. We can convert the above stabilizers to binary-symplectic form to get a PCM. The Steane code is an example of a CSS code, which means the Z and X stabilizers are symmetric. This means we don't have to do out a full matrix, and we can slightly adjust our isomorphism β. We can pick just to work with Z stabilizers for example, and define them with

$$\alpha : I \mapsto 0, Z \mapsto 1$$

So, we can turn 7−qubit long operators into binary strings of length 7 to get the shorter PCM:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

2. We row reduce to get a PCM of the form $(I|A)$

$$\begin{pmatrix}1 & 0 & 0 & 1 & 0 & 1 & 1\\ 0 & 1 & 0 & 1 & 1 & 0 & 1\\ 0 & 0 & 1 & 1 & 1 & 1 & 0\end{pmatrix}$$

3. We can consider the matrix as representing the CNOT gates of the encoding circuit: I as having 3 control qubits, and the rows of A as being the respective 4 target qubits. The control qubits are prepared in the $|+\rangle$ state, and the targets in the $|0\rangle$ state. We can think of the matrix as looking like

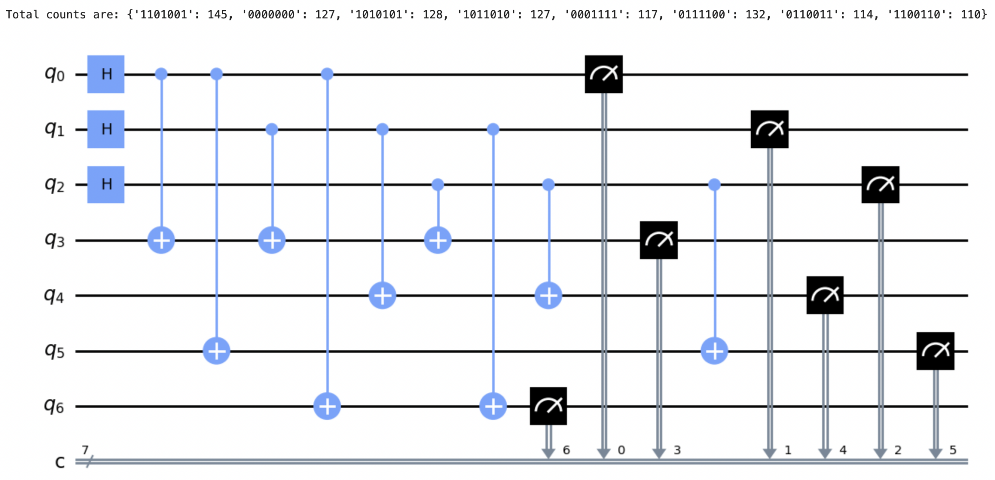
$$\begin{pmatrix}+ & & & 0 & & 0 & 0\\ & + & & 0 & 0 & & 0\\ & & + & 0 & 0 & 0 & \end{pmatrix}$$

4. Each $|+\rangle$ state can be thought of as $\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$. Since we have 3 states of superposition, we have 8 basis states which are summed together with a coefficient of $\frac{1}{\sqrt{2}^3} = \frac{1}{\sqrt{8}}$:

$$\{\psi_{in}\} = \begin{Bmatrix} |0000|000\rangle\,, \\ |0111|100\rangle\,, \\ |1011|010\rangle\,, \\ |1100|110\rangle\,, \\ |1011|001\rangle\,, \\ |1010|101\rangle\,, \\ |0110|011\rangle\,, \\ |0001|111\rangle \end{Bmatrix} \text{ (the | represents symbolic divide between control and target)}$$

$$|0_L\rangle = \frac{1}{\sqrt{8}}\left(\sum_{\psi\in\{\psi_{in}\}}\psi\right)$$

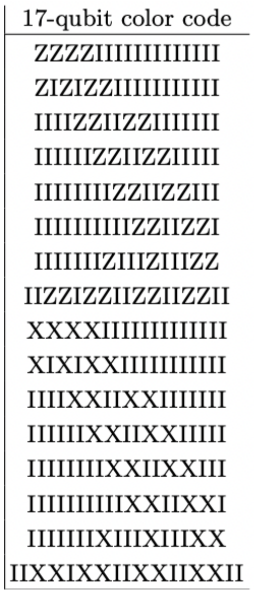
We can also visualize the matrix by doing out the encoding circuit:



```
Total counts are: {'1101001': 145, '0000000': 127, '1010101': 128, '1011010': 127, '0001111': 117, '0111100': 132, '0110011': 114, '1100110': 110}
```

There are $2^3 = 8$ basis states, and the stabilizer group is composed of these basis states exactly. Even though the Steane code technically has 6 stabilizer generators, because it is a CSS code, we can do a complete/functional error correction using only 3.

5. This means that given the stabilizers of the [[17-1-5]] code (we'll see how to derive these in the next section):



17-qubit color code
ZZZZIIIIIIIIIIIII
ZIZIZZIIIIIIIIIII
IIIIZZIIZZIIIIIII
IIIIIIZZIIZZIIIII
IIIIIIIIZZIIZZIII
IIIIIIIIIIZZIIZZI
IIIIIIIZIIIZIIIZZ
IIZZIZZIIZZIIZZII
XXXXIIIIIIIIIIIII
XIXIXXIIIIIIIIIII
IIIIXXIIXXIIIIIII
IIIIIIXXIIXXIIIII
IIIIIIIIXXIIXXIII
IIIIIIIIIIXXIIXXI
IIIIIIIXIIIXIIIXX
IIXXIXXIIXXIIXXII

We can row-reduce on the first 8 generators (since Z and X are symmetric) to get the parity check matrix which will define our encoding circuit.

$$\begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0
\end{pmatrix}$$

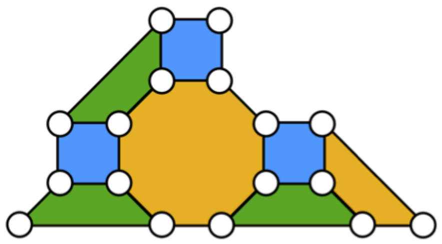
It is unnecessary to permute columns and get a matrix into standard form, $(I|A)$. Steane is a nice matrix since it row-reduced to standard form, but this one clearly did not. Fortunately, we don't need it to, and we can just take our control qubits to be the pivot points (so qubits $1, 2, 3, 5, 7, 8, 9$ and 11 in this case) for an encoding matrix that looks as follows:

$$\begin{pmatrix}
+ & & & 0 & & 0 & & & & 0 & & & 0 & & 0 & 0 & 0 \\
& + & & 0 & & & & & & & & & 0 & 0 & & & \\
& & + & 0 & & 0 & & & & 0 & & & & 0 & 0 & 0 & 0 \\
& & & & + & 0 & & & & & & & 0 & 0 & & & \\
& & & & & & + & & & & & 0 & & & 0 & & 0 \\
& & & & & & & + & & & & 0 & & & & 0 & 0 \\
& & & & & & & & + & 0 & & & 0 & 0 & & & \\
& & & & & & & & & & + & 0 & & & 0 & 0 &
\end{pmatrix}$$

6. We can use this method to encode $|0\rangle_L$ and $|1\rangle_L$ for the [[17,1,5]] code. There are $2^8 = 256$ basis states because as we saw, there are 8 encoding qubits.

3 Topology and Toric Codes

When we talk about the 17-1-5 code, we usually get this image:

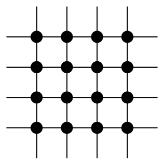


This is a topological representation of the code. Specifically, it is an example of a *toric code*. The 17 vertices correspond to the 17 physical qubits. The faces of the shapes, which are usually called *plaquettes*, correspond to the 8 Z-stabilizer generators, with a symmetric representation in the $X-$stabilizer generators. If we number the dots left to right, top to bottom, and generate strings with Z operators on the boundary of each plaquette, we get the stabilizer generators. For example, the first green trapezoid corresponds to $ZIZIZZIIIIIIIIIII$, which is one of the generators. Let's understand *why* these faces correspond to stabilizers and how to further analyze a picture like this.

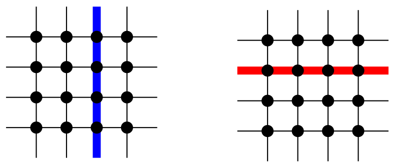
Definition 3.1. A *torus* is a surface defined by having one hole; it is most commonly associated with the donut shape. There's a joke about topologists not knowing the difference between a donut and a coffee mug: this is because the coffee mug can be deformed into the donut; it only has one hole.

A *toric code* is called that because the picture representation can be mapped onto a torus. It's hard to see with the 17-1-5, so we'll start with a much simpler lattice.

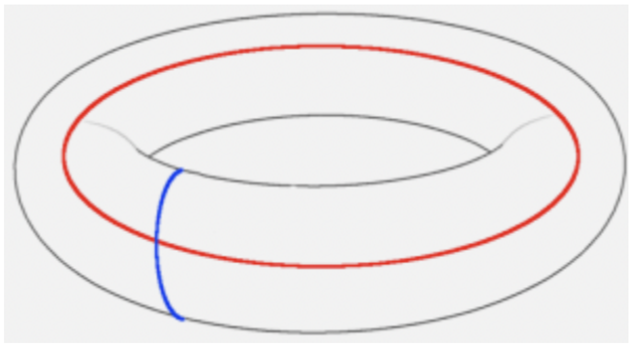
A *lattice* is like a torus laid flat. It is a grid where the top and bottom edge are 'the same', and the side edges are 'the same'. If we fold up a piece of paper according to those rules, we get a torus.
https://upload.wikimedia.org/wikipedia/commons/6/60/Torus_from_rectangle.gif
So, a toric code typically looks like a lattice with qubits at the vertices:



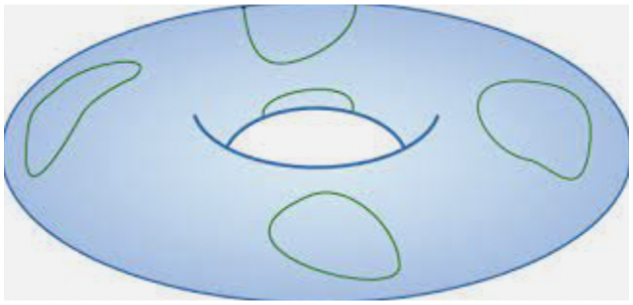
Remembering that this table depicts a 3-dimensional figure, these two lines



are actually loops when we visualize it as a torus:



These loops are called *non-trivial loops* because they can't be deformed (stretched, squished, etc) to a point and stay on the surface of the torus. Here's an example of what trivial loops would look like



[INCLUDE GIF OF SHRINKING GREEN LOOPS TO A POINT)]
The green loops are trivial because they can all be shrunk to a point while staying on the surface of the torus.

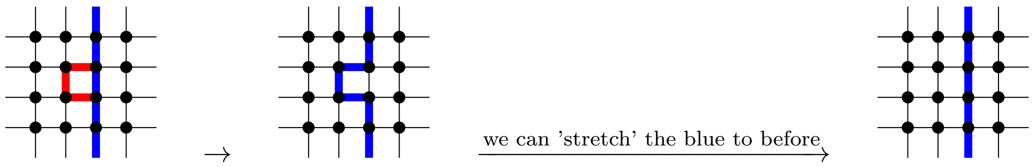
Definition 3.2. Two loops are *homotopic* if they can be continuously deformed into each-other. This is why we call the loops on the outside face (not going all the way around) trivial: because they are homotopic to a point, which is a trivial loop.

The earlier example of two non-trivial loops also can't be deformed into each other and represent the logical operators of the code.

If we map a plaquette onto the torus, we get a trivial loop. When a loop is trivial, it essentially does nothing. The algebraic definition of the stabilizers were operators that "had no effect" on group elements: $S_{\mathcal{G}} = \{s : sg = g, g \in \mathcal{G}\}$ where $\mathcal{G}$ is some group and $S_{\mathcal{G}}$ is the set of stabilizers of the group. It follows that any loop which is homotopic to a point, and therefore trivial, represents a stabilizer of the code.

Similarly to the algebraic formalism, the composition of boundaries is still a stabilizer. Merging two trivial loops makes a third trivial loop. If we take, say, one of the logical operators and compose it with a stabilizer, we get the "virtually" same effect as if the logical operator was on its own:

[WANT TO TURN BELOW INTO GIF]



If we take a logical operator L and compose it with any stabilizer, we know we can deform it back to itself. This operator L is the representative of all loops in the $L \cdot S_{\mathcal{G}}$ stabilizer coset, meaning all loops in the same stabilizer coset are homotopic to each other. If we have 8 Z-stabilizer generators, then for one logical operator, there are 2^8 homotopic loops. This corresponds to the 2^8 basis states enumerated in the encoding of $|0\rangle_L$.

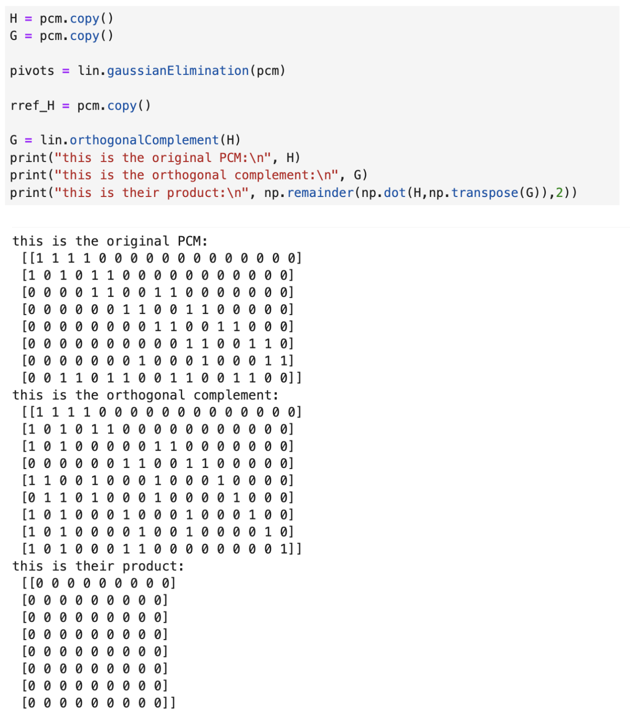
The 17-1-5 code isn't a perfect lattice, and so doesn't correspond directly to a simple torus. However, we can still make more sense of the picture. The code has distance 5. What this means is that in the diagram, a valid logical operator has minimum 5 qubits; the logical operators are the strings which take 5 qubits to go from top to bottom or side to side.

[STABILIZER VISUALIZATION HERE] using matplotlib

4 Decoding Book and Degeneracy

4.1 Creating the Codebook

There are two types of solutions to $\mathbf{H}x = 0$, where $\mathbf{H}$ is the parity check matrix. We can find the generators for all possible solutions by finding the orthogonal complement to $\mathbf{H}$ (called it G), because by definition $H \cdot G^T$ will return a zero matrix. Because the symplectic product of each row of G with the stabilizers evaluates to 0, this means each row commutes with all the stabilizers, so each row is a generator for the centralizer group.



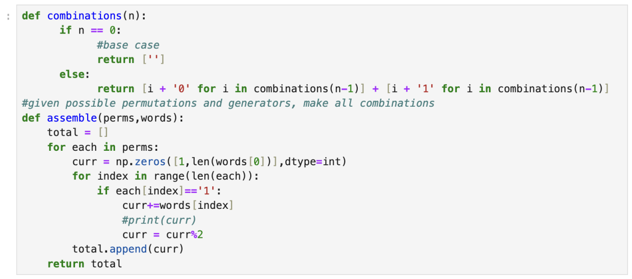
```
H = pcm.copy()
G = pcm.copy()

pivots = lin.gaussianElimination(pcm)

rref_H = pcm.copy()

G = lin.orthogonalComplement(H)
print("this is the original PCM:\n", H)
print("this is the orthogonal complement:\n", G)
print("this is their product:\n", np.remainder(np.dot(H,np.transpose(G)),2))
```

```
this is the original PCM:
 [[1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0]
 [0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1]
 [0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0]]
this is the orthogonal complement:
 [[1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0]
 [1 1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0]
 [0 1 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0]
 [1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0]
 [1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 0]
 [1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 1]]
this is their product:
 [[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
```

The codebook consists of all possible combinations of the generators. The orthogonal complement has 9 rows, so we have 9 generators for the entire group. Since we are working in a binary field, any given product can contain each generator once or zero times. Consider a smaller example:

1. Let a, b, c be stabilizer generators. Recalling binary representation, there are 8 distinct ways to combine the 3
2. For example, 110 could refer to the stabilizer $a \cdot b$. 101 corresponds to $a \cdot c$.
3. Any stabilizer generated from a, b, c has the form $a^{x_1}, b^{x_2}, c^{x_3}$, where x_1, x_2, x_3 are all either 0 or 1.

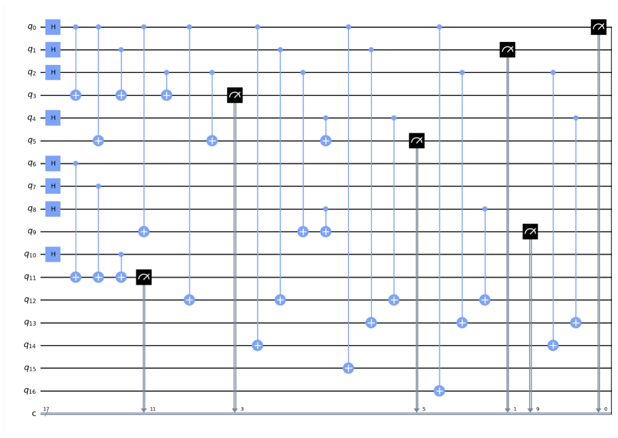
So, since we have 9 centralizer generators and 8 stabilizer generators, we can enumerate every possible combination with the following program:

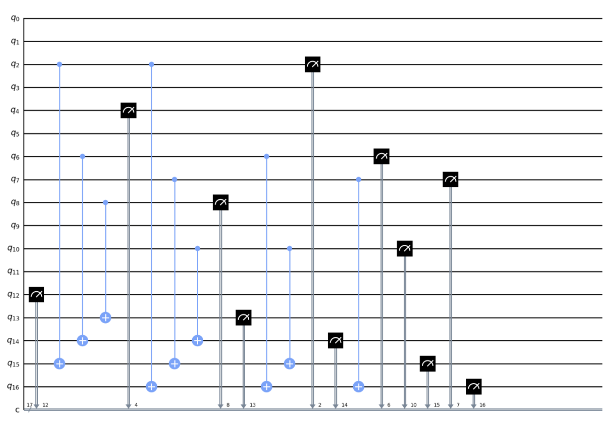


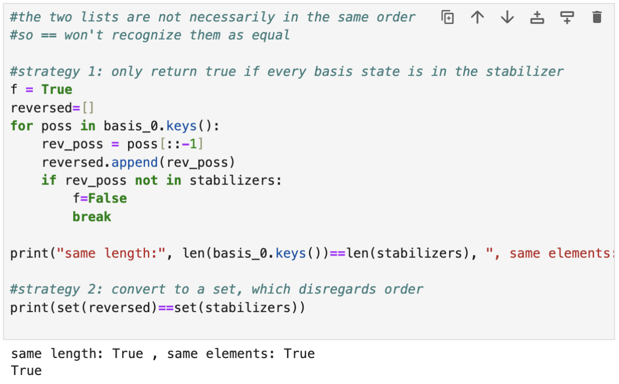
```
def combinations(n):
        if n == 0:
              #base case
              return ['']
        else:
              return [i + '0' for i in combinations(n-1)] + [i + '1' for i in combinations(n-1)]
#given possible permutations and generators, make all combinations
def assemble(perms,words):
    total = []
    for each in perms:
        curr = np.zeros([1,len(words[0])],dtype=int)
        for index in range(len(each)):
            if each[index]=='1':
                curr+=words[index]
                #print(curr)
                curr = curr%2
        total.append(curr)
    return total
```

The `combinations` function constructs all 2^x binary representations, given x number of generators. The `assemble` function does the work of evaluating each combinations given an array of generators. We can use these functions to find all combinations of the stabilizer generators, which also correspond to our $|0\rangle_L$ basis states. Given this, we can find all basis states which correspond to $|1\rangle_L$.

Using the strategy described in Encoding Qubits, I simulated the $|0\rangle_L$ and $|1\rangle_L$ basis states to confirm my results.





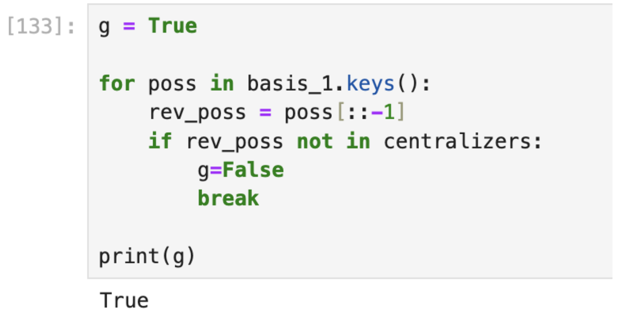


```
#the two lists are not necessarily in the same order
#so == won't recognize them as equal

#strategy 1: only return true if every basis state is in the stabilizer
f = True
reversed=[]
for poss in basis_0.keys():
    rev_poss = poss[::-1]
    reversed.append(rev_poss)
    if rev_poss not in stabilizers:
        f=False
        break

print("same length:", len(basis_0.keys())==len(stabilizers), ", same elements:

#strategy 2: convert to a set, which disregards order
print(set(reversed)==set(stabilizers))
```

```
same length: True , same elements: True
True
```

I did have to reverse each string before comparison; this is due to the basis states being written from right to left in order of qubit (rightmost qubit is the first one)

To find the $|1\rangle_L$, basis states, I added a `NOT` gate (X) to each qubit right before measurement. This is the equivalent of flipping every single bit; and I found the centralizer and $|1\rangle_L$ to be equal.



```
[133]: g = True

       for poss in basis_1.keys():
           rev_poss = poss[::-1]
           if rev_poss not in centralizers:
               g=False
               break

       print(g)
```

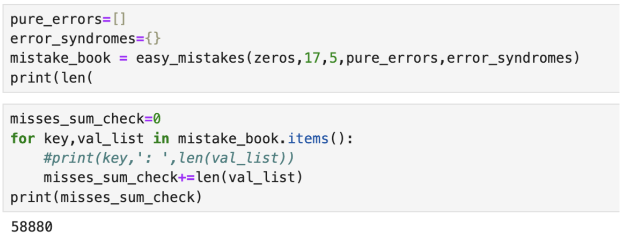
```
True
```

The codebook is comprised of all possible basis states, so it is exactly the stabilizer and centralizer lists.

4.2 Creating Fixable Errors

Since the [[17,1,5]] code has distance 5, we know we can unambiguously correct up to $5//2 = 2$ qubit errors. Since the minimum distance between logical operators is 5 qubits, there is no chance of a 2 qubit error pertaining to both $|0\rangle_L$ and $|1\rangle_L$ basis states.

So, we need to find all possible 1 and 2 qubit errors. For a given codeword, there are 17 possible 1 qubit errors, and $\binom{17}{2} = 136$ possible 2 qubit errors, for a total of 153 errors per codeword. Since we have 512 codewords, there should be $153 \times 512 = 78336$ fixable errors. However, we find this is not the case:

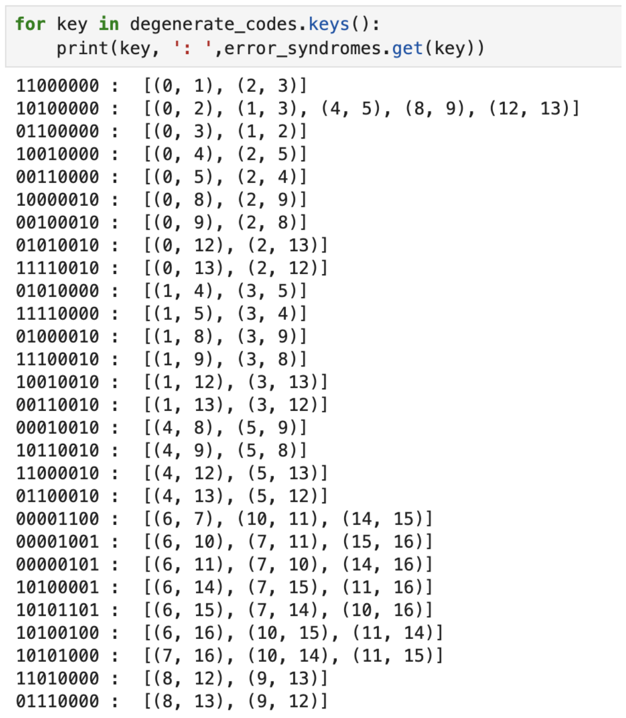


```
pure_errors=[]
error_syndromes={}
mistake_book = easy_mistakes(zeros,17,5,pure_errors,error_syndromes)
print(len(
```

```
misses_sum_check=0
for key,val_list in mistake_book.items():
    #print(key,': ',len(val_list))
    misses_sum_check+=len(val_list)
print(misses_sum_check)
```

```
58880
```

We also have only 115 syndromes instead of 153. The `error_syndromes` dictionary tracks which pure errors correspond to which syndrome. We know there must be multiple errors corresponding to the same syndrome, so we can use this dictionary to pick out exactly those special syndromes. As you can see, there are exactly $153 - 115 = 38$ errors corresponding to duplicate syndromes.



```
for key in degenerate_codes.keys():
    print(key, ': ',error_syndromes.get(key))
```

```
11000000 :  [(0, 1), (2, 3)]
10100000 :  [(0, 2), (1, 3), (4, 5), (8, 9), (12, 13)]
01100000 :  [(0, 3), (1, 2)]
10010000 :  [(0, 4), (2, 5)]
00110000 :  [(0, 5), (2, 4)]
10000010 :  [(0, 8), (2, 9)]
00100010 :  [(0, 9), (2, 8)]
01010010 :  [(0, 12), (2, 13)]
11110010 :  [(0, 13), (2, 12)]
01010000 :  [(1, 4), (3, 5)]
11110000 :  [(1, 5), (3, 4)]
01000010 :  [(1, 8), (3, 9)]
11100010 :  [(1, 9), (3, 8)]
10010010 :  [(1, 12), (3, 13)]
00110010 :  [(1, 13), (3, 12)]
00010010 :  [(4, 8), (5, 9)]
10110010 :  [(4, 9), (5, 8)]
11000010 :  [(4, 12), (5, 13)]
01100010 :  [(4, 13), (5, 12)]
00001100 :  [(6, 7), (10, 11), (14, 15)]
00001001 :  [(6, 10), (7, 11), (15, 16)]
00000101 :  [(6, 11), (7, 10), (14, 16)]
10100001 :  [(6, 14), (7, 15), (11, 16)]
10101101 :  [(6, 15), (7, 14), (10, 16)]
10100100 :  [(6, 16), (10, 15), (11, 14)]
10101000 :  [(7, 16), (10, 14), (11, 15)]
11010000 :  [(8, 12), (9, 13)]
01110000 :  [(8, 13), (9, 12)]
```

4.3 Degeneracy

We say two errors are *degenerate* if they are in the same stabilizer coset, meaning they only differ by a stabilizer. For example, suppose two operators E and E' are degenerate. Then $E = E' \cdot S_i$ for some stabilizer element S_i. Observe

$$\begin{aligned} |\overline{\psi}\rangle_\psi &= E' \, |\overline{\psi}\rangle \\ &= E \cdot S_i \, |\overline{\psi}\rangle \\ &= E \, |\overline{\psi}\rangle \\ E' \, |\overline{\psi}\rangle &= E \, |\overline{\psi}\rangle \end{aligned}$$

These two degenerate operators act on a codeword in the same way, or are indistinguishable in their effect on a quantum state. Suppose R is a recovery operator for E, meaning $R \cdot E \, |\overline{\psi}\rangle = |\overline{\psi}\rangle$. Since E and E' are degenerate, we have

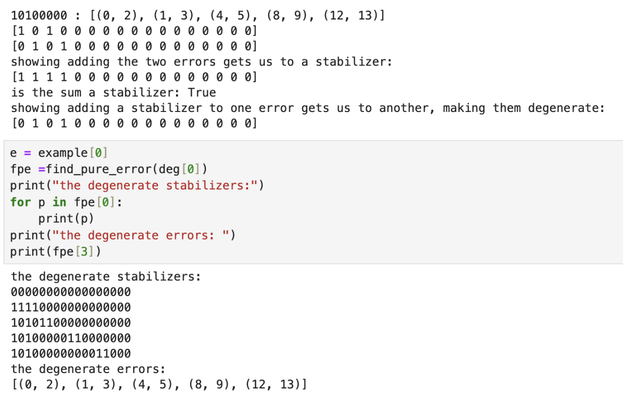
$$\begin{aligned} R \cdot E' \, |\overline{\psi}\rangle &= R \cdot E \, |\overline{\psi}\rangle \\ &= |\overline{\psi}\rangle \end{aligned}$$

which means R is a recovery operator for E' as well! The phenomenon of degeneracy massively simplifies the problem of quantum error correction, since we only need to worry about all possible $T_i \cdot L_j$ operators, and not every single element of the Pauli group.

The implication of this is that we can recover many different errors with only one recovery operator, which should improve the performance of quantum codes.

What is happening in the decoding simulation is precisely the phenomenon of degeneracy: all errors which correspond to the same syndrome are degenerate. This means that given an error operator, we may have multiple possibilities for correcting bit flips. We don't need a recovery operator to return us to the exact original word, we just need to get to a stabilizer, because these are homotopic to the identity (they can shrink to be trivial). How is this possible in a distance 5 code? That minimum distance pertains to distance between distinct logical operators ($|0\rangle_L$ versus $|1\rangle_L$), but within the stabilizer group, we have many instances of stabilizers with distance less than 5. The alternative definition to two degenerate codes is those which have a distance less than the minimum. If two stabilizers differ by 4 bits, then they can both correspond to the same error of 2 qubit flips. Given the error, there are multiple options for error qubits because we would be able to reach two different stabilizers.

Example:



```
10100000 : [(0, 2), (1, 3), (4, 5), (8, 9), (12, 13)]
[1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
showing adding the two errors gets us to a stabilizer:
[1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
is the sum a stabilizer: True
showing adding a stabilizer to one error gets us to another, making them degenerate:
[0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

```python
e = example[0]
fpe =find_pure_error(deg[0])
print("the degenerate stabilizers:")
for p in fpe[0]:
    print(p)
print("the degenerate errors: ")
print(fpe[3])
```

```
the degenerate stabilizers:
00000000000000000
11110000000000000
10101100000000000
10100000110000000
10100000000011000
the degenerate errors:
[(0, 2), (1, 3), (4, 5), (8, 9), (12, 13)]
```

It is fairly simple to decode, given a syndrome. We can look up this syndrome in our dictionary to find all potential pure errors. Assuming only unambiguous errors, we know applying any one of these flips will return an encoding in the correct set (a $|0\rangle_L$ basis state or a $|1\rangle_L$ basis state). We can check the given codeword against the earlier lists of stabilizers and centralizers to determine which logical bit this encoding corresponds to.