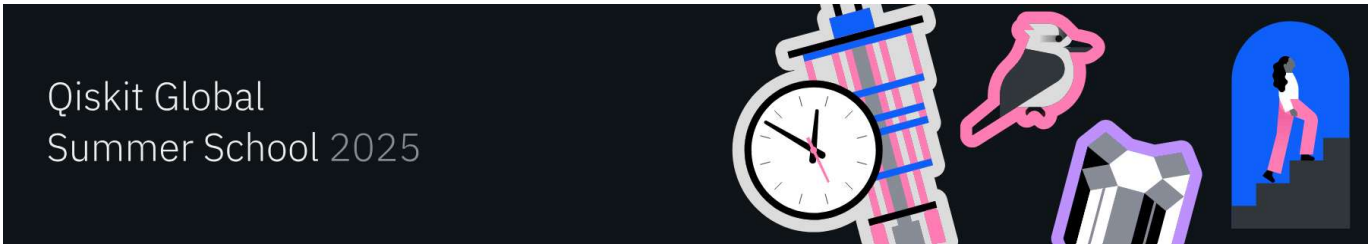


Lab 4: Quantum Error Correction: From Core Concepts to the Road to Fault-Tolerant Quantum Computing

Welcome to the forth coding challenge of the Qiskit Global Summer School. This lab explores error-correcting codes, beginning with foundational classical error-correcting codes and key concepts. It then transitions to Quantum Error Correction (QEC) — crucial for fault-tolerant quantum computing — and covers the stabilizer formalism along with key examples. Subsequently, the notebook introduces advanced QEC architectures, including the QLDPC, Toric, and gross codes, with exercises.

Table of Contents

Requirements

Before starting this tutorial, please make sure that you have the following installed:

- Qiskit SDK v2.0 or later with visualization support (`pip install 'qiskit[visualization]'`)
- Qiskit Runtime v0.40 or later (`pip install qiskit-ibm-runtime`)

```
#%pip install "qc-grader[qiskit,jupyter] @ git+https://github.com/qiskit-community/Quantu
```

```
import qiskit
import qc_grader
print(f"Qiskit version: {qiskit.__version__}")
print(f"Grader version: {qc_grader.__version__}")
```

You should have Qiskit version `>=2.0.0` and Grader `>=0.22.12`. If you see a lower version, you need to reinstall the grader and restart the kernel. Also make sure you have set up everything according to lab 0 and test it with the cell below.

```
# Check that the account has been saved properly
from qiskit_ibm_runtime import QiskitRuntimeService

service = QiskitRuntimeService(name="qgss-2025")
service.saved_accounts()
```

Imports

```
# Import common packages first
import numpy as np

# Import qiskit classes
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
from qiskit.visualization import plot_histogram

# Import utils and cosystems
```

```
from lab4_util import hamming_distance, minimum_distance, bring_states, matrixRank
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime import SamplerV2  as Sampler

# Import grader
from qc_grader.challenges.qgss_2025 import (
    grade_lab4_ex1,
    grade_lab4_ex2,
    grade_lab4_ex3,
    grade_lab4_ex4,
    grade_lab4_ex5,
    grade_lab4_ex6
)
```

Chapter 1: Classical error-correcting code revisit

1.1 Classical [n, k, d] Codes

In coding theory, a classical binary linear block code is often denoted as an $[n, k, d]$ code for messages composed of binary (bit) strings, that is, a series of 0s and 1s. These parameters represent:

- n: The **codeword length**. Each codeword in the code is a binary string of length n.
- k: The **message length**. It represents the number of information bits encoded into a codeword. There are 2^k distinct messages and thus 2^k unique codewords in the code.
- d: The **minimum Hamming distance** of the code. This is the smallest Hamming distance between any pair of *distinct* codewords in the code. The minimum distance d determines the code's error detection and correction capabilities. Only a subset of the 2^n possible binary strings represent valid codewords that each carry k bits of information.

1.2 Hamming Distance

The **Hamming distance** between two strings (or vectors) of equal length is the number of positions at which the corresponding symbols are different. For binary strings (composed of 0s and 1s), it's simply the count of positions where one string has a '0' and the other has a '1'.

Mathematically, for two binary vectors $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$, the Hamming distance $d_H(x, y)$ is:

$$d_H(x, y) = \sum_{i=1}^{n} (x_i \oplus y_i)$$

where $\oplus$ denotes the XOR operation (addition modulo 2), which is equivalent to counting the number of '1's in the result of the bitwise XOR operation between x and y. This count is also known as the **Hamming weight** of the XOR result.

Important

The Hamming distance is crucial for understanding error correction codes:

1. **Error Detection**: A code with minimum distance d can detect any error pattern affecting up to $d-1$ bits. If fewer than d bits are flipped during the transmission of a codeword, the resulting string cannot be another valid codeword, so the error is detected.
2. **Error Correction**: A code with minimum distance d can correct any error pattern affecting up to t bits in a codeword, where $t = \lfloor \frac{d-1}{2} \rfloor$. This is because if at most t errors occur, the received word is still closer (in Hamming distance) to the original codeword than to any other valid codeword.

Here's a Python function to calculate the Hamming distance between two binary strings:

```
# Example usage:
str1 = "10110"
str2 = "11100"
dist = hamming_distance(str1, str2)
print(f"Hamming distance between '{str1}' and '{str2}' is: {dist}") # Output: 2

vec1 = [1, 0, 0, 1]
vec2 = [0, 0, 1, 1]
dist_vec = hamming_distance(vec1, vec2)
print(f"Hamming distance between {vec1} and {vec2} is: {dist_vec}") # Output: 2
```

Minimum Distance of a Code (d)

The minimum Hamming distance d of a code C is the smallest Hamming distance found between any pair of distinct codewords in $C \subset \{0,1\}^n$.

$$d = \min d_H(c_1, c_2) \mid c_1, c_2 \in C, c_1 \neq c_2 .$$

For linear codes, where $c_1 \in C$ and $c_2 \in C$ implies $(c_1 + c_2) \in C$, the minimum distance is also equal to the minimum Hamming weight of all non-zero codewords. The Hamming weight of a codeword is its distance from the all-zero codeword.

```
# --- Example: A Simple [4, 3, 2] Parity Check Code ---
# This code takes 3 message bits (k=3) and adds an even parity bit
# to make the total codeword length n=4.
# Messages: 000, 001, 010, 011, 100, 101, 110, 111
# Codewords (adding even parity bit):
parity_code_4_3 = [
    "0000", # 000 + 0 (even parity)
    "0011", # 001 + 1
    "0101", # 010 + 1
    "0110", # 011 + 0
    "1001", # 100 + 1
    "1010", # 101 + 0
    "1100", # 110 + 0
    "1111"  # 111 + 1
]
```

```
# Calculate the minimum distance d
d_parity = minimum_distance(parity_code_4_3)
print(f"Codewords: {parity_code_4_3}")
print(f"Calculated minimum distance d = {d_parity}") # Output: 2
```

This is therefore a [4, 3, 2] code.

For our example code parity_code_4_3, we found $d = 2$.

- Error Detection: $t_{detect} = d - 1 = 2 - 1 = 1$. This code can detect any single-bit error.
- Error Correction: $t_{correct} = \lfloor \frac{d-1}{2} \rfloor = \lfloor \frac{2-1}{2} \rfloor = \lfloor 0.5 \rfloor = 0$.

In particular, because any single error will result in a string that is not in the codespace, we know an error occurred and, as such, is detected!

While this code can identify errors, it lacks the capability to correct them in all cases. Consider an error where the codeword `0000` is altered to `0001`. The error is detectable as `0001` is not a valid codeword. However, correction is not possible because `0001` has an identical Hamming distance to four different codewords (`0000`, `0011`, `0101`, and `1001`).

Hence, in a [4, 3, 2] code, a single bit-flip error can yield a binary string that is not a codeword - but one cannot reliably correct such an error, as there is no unique closest codeword for correction - thus, it is impossible to know which of the original codewords was sent.

In general, the reason a code can detect up to $d - 1$ errors is that if any such set of bit-flips were to occur and corrupt the sent message, then one could identify that the transformed message is no longer a codeword and, as such, detect that some set of errors occurred. While correcting such errors is more involved, any error of weight at most $\lfloor \frac{d-1}{2} \rfloor$ can be corrected, as one can consider the corrupted codeword and look at all possible codewords within Hamming distance $\lfloor \frac{d-1}{2} \rfloor$ - there will only be one such possibility. Doing this task efficiently, also known as decoding, is in general a non-trivial task that is an active area of research, depending on the code.

To investigate this further, let's recalculate for a different code, e.g., the [3, 1, 3] repetition code $C = 000, 111$.

1.3 The [3, 1, 3] Repetition Code

The simplest error-correcting code is the repetition code. To send a single bit (k=1), we repeat it `n` times. For the `[3, 1, 3]` code, we repeat the bit 3 times.

- Message `0` is encoded as `000`.
- Message `1` is encoded as `111`.

Here, n=3 and k=1. The codewords are {000, 111}. The Hamming distance between 000 and 111 is 3. So, $d = 3$. This code can detect up to $d - 1 = 2$ errors. It can correct up to $\lfloor ((d - 1)/2) \rfloor = \lfloor ((3 - 1)/2) \rfloor$ = 1 error.

Encoding:

- If the message bit is `m`, codeword `c = mmm`.

Error & Decoding (Majority Vote): Suppose a codeword `c` is transmitted, and `c'` is received. If at most one bit is flipped, the original message can be recovered by majority voting.

- Receive `000` -> Decode `0`
- Receive `001` -> Decode `0` (1 error)
- Receive `010` -> Decode `0` (1 error)
- Receive `100` -> Decode `0` (1 error)
- Receive `111` -> Decode `1`
- Receive `110` -> Decode `1` (1 error)
- Receive `101` -> Decode `1` (1 error)
- Receive `011` -> Decode `1` (1 error)

If two bits are flipped (e.g., `000` -> `011`), majority voting decodes to the wrong message (`1` in this case). The code can detect this as a 2-bit error but cannot correct it.

Let's first check the minimum distance of this code with both error detection and error correction capabilities.

```
# --- Example: [3, 1, 3] Repetition Code ---
repetition_code_3_1 = ["000", "111"]
d_repetition = minimum_distance(repetition_code_3_1)
print(f"Calculated minimum distance d = {d_repetition}") # Output: 3

# Capabilities for d=3:
t_detect = d_repetition - 1
t_correct = int((d_repetition - 1) / 2) // 1
print(f"Error Detection Capability (t_detect = d-1): {t_detect}") # Output: 2
print(f"Error Correction Capability (t_correct = floor((d-1)/2)): {t_correct}") # Output:
```

Practice: Lookup table-based Decoder ofthe [3, 1, 3] Code

Since this code can correct errors, we can now create a lookup table-based decoder in the form of a dictionary to perform error correction. A lookup table-based decoder maps every possible received 3-bit string to the most-likely transmitted 1-bit message, with regard to the corrected message. Here we will use a helper function `hamming_distance` to see how the Hamming distance affects correcting errors. First let's do a simple test with one single case.

```
test_str = "010"

print("Hamming distance between 010 and 000 is", hamming_distance(test_str, "000"))
print("Hamming distance between 010 and 111 is", hamming_distance(test_str, "111"))
```

"000" is a logical `0` and "111" is a logical `1`, so this `test_str` will be corrected as `0`. Try to complete the dictionary below to complete the whole lookup table-based decoder of the [3, 1, 3] classical error correcting code, by mapping the final corrected string to the received string.

Before checking the solution table below, it's helpful to understand its columns: 'Received text' is the 3-bit string that was observed after potential errors; 'Hamming distance with "000"' and 'Hamming distance with "111"' shows how many bits differ from the two valid logical codewords (000 and 111, respectively); and 'Corrected str' is the single bit (0 or 1) that the decoder determines was the original message.

```
hardcode_decoder_3_1_3 ={
    '000': '0',
    '001': '',
    '010': '',
    '011': '',
    '100': '',
    '101': '',
    '110': '',
    '111': '1'}
```

▶ Check your decoder with this!

Chapter 2. Quantum Error Correcting Codes [[n, k, d]]

Quantum information is much more fragile than classical information. Qubits are susceptible to various types of errors, not just bit-flips (X errors) but also phase-flips (Z errors) and combinations of both (Y errors). Quantum error correction aims to protect quantum states from these errors.

A quantum error correcting code is often denoted by `[[n, k, d]]`, where:

- `n`: The number of **data qubits** used to construct the code.
- `k`: The number of **logical qubits** encoded by the code. This means the code protects a 2^k-dimensional logical state space (codespace).
- `d`: The **minimum distance** of the code. This is the minimum number of qubits an operator needs to act on to perform a non-identity operation on a logical qubit. In the case of stabilizer codes, it is the minimum number of qubits a Pauli operator must act on in order to apply a logical Pauli operator.

Note on resource overhead:

It's important to note that the `n` in the `[[n, k, d]]` notation typically refers to the data qubits involved in the code block. This count usually *omits* any ancilla or syndrome qubits required for tasks like error detection and correction. Therefore, the total resource overhead is not simply `n - k` data qubits, but rather `(n - k)` (the number of redundant data qubits) *plus* the number of

ancilla qubits needed for syndrome measurements. In many stabilizer codes, the number of independent stabilizers (and thus often the number of ancilla qubits) is `n - k`, but this can vary, and in some schemes, the ancilla count could be significant, potentially as large as `n`.

Similar to classical codes, the minimum distance d determines the code's error correction capability. An `[[n, k, d]]` code can detect up to `d-1` errors and can potentially correct up to `floor((d-1)/2)` arbitrary single-qubit errors.

2.1 Stabilizer Formalism

The stabilizer formalism is a powerful framework for constructing and understanding many QEC codes.

- The **Pauli group** P_n on n qubits consists of all n-fold tensor products of Pauli matrices $\{I, X, Y, Z\}$, multiplied by overall factors $\{\pm 1, \pm i\}$.
- A **stabilizer code** is defined by its **stabilizer group S**, which is an abelian subgroup of P_n such that $-I \notin S$.
- The **codespace C(S)** is the subspace of $(\mathbb{C}^2)^{\otimes n}$ simultaneously stabilized by all elements of S. That is, for any state $|\psi\rangle \in C(S)$ and any $g \in S$, $g|\psi\rangle = |\psi\rangle$. So, states in the codespace are +1 eigenstates of all stabilizer operators.
- Typically, S is specified by a set of $m = n - k$ independent and commuting generators $S = \langle g_1, g_2, \ldots, g_m \rangle$.

A comprehensive resource is the IBM Quantum Learning course by John Watrous: Foundations of Quantum Error Correction. If you wish to delve deeper into the mathematical details and further examples, we recommend exploring this resource. Click `Stabilizer Formalism Summary` to see the summary, or you can skip this.

▶ **Stabilizer Formalism Summary**

Stabilizer formalism provides the foundation for understanding how codes like the `[[7, 1, 3]]` example (below) are constructed and how they detect errors of any type.

2.2 3-qubit bit-flip Code Practice

(Note: Throughout this explanation, we use Qiskit's little-endian convention, where the rightmost character in a Pauli string corresponds to qubit 0, e.g., `IZZ` acts as $I_2 \otimes Z_1 \otimes Z_0$.)

This code is the quantum analog of the classical [3, 1, 3] repetition code. It can correct a **single bit-flip (X) error** on any of the three physical qubits - however, it will not be able to correct for Z phase errors.

- n=3 physical qubits, k=1 logical qubit. The goal of the code is to preserve the state of the logical qubit, despite errors on the physical qubits.
- Logical states (unnormalized):

- $|0_L\rangle \equiv |000\rangle$
- $|1_L\rangle \equiv |111\rangle$
- An arbitrary logical state is $\alpha|0_L\rangle + \beta|1_L\rangle = \alpha|000\rangle + \beta|111\rangle$.

Stabilizer Generators: This code is stabilized by:

- $S_0 = Z_2 Z_1 I_0$ (or simply $Z_1 Z_2$)
- $S_1 = I_2 Z_1 Z_0$ (or simply $Z_0 Z_1$) These are Z-type stabilizers, which are used to detect X-errors (bit-flips).

Logical Operators:

- Logical Z: $\bar{Z} = Z_0$ (or Z_1 or Z_2, as $Z_0|0_L\rangle = |0_L\rangle, Z_0|1_L\rangle = -|1_L\rangle$). A common choice is $Z_2 Z_1 Z_0$.
- Logical X: $\bar{X} = X_2 X_1 X_0$. ($\bar{X}|0_L\rangle = |1_L\rangle, \bar{X}|1_L\rangle = |0_L\rangle$).

Error Detection (Syndrome Extraction): Let s_0, s_1 be the eigenvalues (-1 if flipped, +1 if not) for S_0, S_1 respectively. The syndrome is (s_0, s_1).

- No error: $(+1, +1)$
- X_0 error (bit-flip on qubit 0): S_0 commutes, S_1 anti-commutes ($Z_0 Z_1 X_0 = -X_0 Z_0 Z_1$). Syndrome: $(s_1, s_0) = (-1, +1)$. Correction: Apply X_0.
- X_1 error (bit-flip on qubit 1): S_0 anti-commutes, S_1 anti-commutes. Syndrome: $(-1, -1)$. Correction: Apply X_1.
- X_2 error (bit-flip on qubit 2): S_0 anti-commutes, S_1 commutes. Syndrome: $(+1, -1)$. Correction: Apply X_2.

Vulnerability to Z-errors

Keep in mind that this code is primarily designed to protect against bit-flip (X-type) errors. It's crucial to note this code does undergo any Z-type (phase-flip) errors. A single Z-type error on any of the physical qubits would commute with the stabilizers that detect X-errors (e.g., $Z_i Z_j$) and would manifest as an uncorrectable logical Z error on the encoded qubit, altering its phase information without being detected.

Exercise 1: Lookup Table-based Decoder of 3-bit code

As we introduced, the quantum bit-flip code, defined by two stabilizer generators $S_0 = ZZI$ ($Z_2 \otimes Z_1 \otimes I_0$) and $S_1 = IZZ$ ($I_2 \otimes Z_1 \otimes Z_0$) and be measured into two classical bit. So the syndrome outcome will be a 2-bit classical bit string. This bit strings serve as the input of a hardcord decoder, which is a lookup table that maps the syndrome to the specific correction operation needed.

Your task is to complete the dictionary that maps these measured syndrome bits to the corresponding error code (e.g., 'X0' for an X error on qubit 0, 'X1' for an X error on qubit 1, 'X2' for an X error on qubit 2, or 'I' for no error) for a single (Weight-1) bit-flip error.

```
hardcode_decoder_bit_flip_syndrome_map = {
    # ---- TODO : Task 1 ---
    # Fill in the other entries of the decoder (leave the "" in place).
    #{"s1s0": "Error Code"}
    '00': 'I',
    '01': '',
    '10': '',
    '11': ''
    # --- End of TODO ---
}



# Submit your answer using following code
grade_lab4_ex1(hardcode_decoder_bit_flip_syndrome_map )
```

2.3 CSS Codes (Calderbank-Shor-Steane)

CSS codes, named after Calderbank, Shor, and Steane, are a powerful family of quantum codes constructed from classical linear codes. Here we provide a high-level introduction, and for a more detailed explanation of CSS codes, refer to the IBM Quantum Learning course material on Quantum Code Constructions (CSS Codes).

Construction: A CSS code is defined using two classical binary linear codes, $C_1 = [n, k_1, d_1]$ and $C_2 = [n, k_2, d_2]$, satisfying the condition $C_2^\perp \subseteq C_1$. Here, $C_2^\perp$ is the dual code of C_2.

- **Stabilizers:** The stabilizer group S for the CSS code $[[n, k, d]]$ (where $k = k_1 + k_2 - n$) is generated by:
    - **X-type stabilizers:** Derived from the parity check matrix H_1 of C_1. For each row $h \in H_1$, create a stabilizer $\bigotimes_{i:h_i=1} X_i$. These stabilizers consist only of Pauli X and I operators.
    - **Z-type stabilizers:** Derived from the parity check matrix $H_2^\perp$ of $C_2^\perp$. For each row $h' \in H_2^\perp$, create a stabilizer $\bigotimes_{i:h'_i=1} Z_i$. These stabilizers consist only of Pauli Z and I operators.
    - (Note: Conventions might swap the roles of C_1 and $C_2^\perp$ for X and Z stabilizers).*

Key Property and Decoding: The crucial property of CSS codes is the separation of stabilizer types:

- **X-stabilizers** commute with X errors but can anti-commute with Z errors on the qubits they act upon. They are used to detect **Z-type errors** (and the Z component of Y errors).
- **Z-stabilizers** commute with Z errors but can anti-commute with X errors on the qubits they act upon. They are used to detect **X-type errors** (and the X component of Y errors).

This separation simplifies decoding:

1. Measure all stabilizers to get the full syndrome.

2. Extract the syndrome bits corresponding to the **Z-stabilizers**. Use these bits and a classical decoding algorithm associated with the code C_1 (whose checks define the X stabilizers) to identify and correct likely **X or Y errors**.
3. Extract the syndrome bits corresponding to the **X-stabilizers**. Use these bits and a classical decoding algorithm associated with the code $C_2^\perp$ (whose checks define the Z stabilizers) to identify and correct likely **Z or Y errors**.

This powerful CSS construction, which distinctly handles X and Z errors using classical code components, is exemplified by the renowned [[7, 1, 3]] Steane code. In the next section, we'll explore the specific stabilizers and practical aspects of this important code.

2.4 [[7,1,3]] Steane Code Practice

The **[[7, 1, 3]] Steane code** is a quintessential CSS code. It encodes $k = 1$ logical qubit into $n = 7$ physical qubits and can correct any arbitrary single-qubit error (since $d = 3$). It is constructed using the classical [7, 4, 3] Hamming code for both C_1 and C_2 (i.e., $H_1 = H_2 = H_{\text{Hamming}}$).

Parity Check Matrix Primer

A **parity check matrix** (H) for a classical linear code is a fundamental tool. If c is a codeword, then the product $Hc^T = \mathbf{0}$ (the zero vector) must hold. Each row of H defines a parity check, meaning it specifies a subset of codeword bits that must sum to 0 (modulo 2). If $Hc^T \neq \mathbf{0}$, the result is called the **syndrome**, which indicates an error has occurred and can be used to identify it. For CSS codes, the rows of classical parity check matrices are used to define the X-type and Z-type stabilizer generators of the quantum code.

Stabilizer Generators from the Parity Check Matrix

The parity check matrix H whose rows define stabilizer structures (for qubits $q_0, \ldots, q_6$) of the Steane code is:

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

Each row $(h_0, h_1, h_2, h_3, h_4, h_5, h_6)$ of this matrix H corresponds to a stabilizer generator. For an X-stabilizer, an X_i operator is present if $h_i = 1$. For a Z-stabilizer, a Z_i operator is present if $h_i = 1$.

So, the stabilizer generators are:

- **X-type stabilizers** (these detect Z-errors), derived from the rows of H:
  - $S_{X0} = X_0X_1X_2X_3 = IIIXXXX$
  - $S_{X1} = X_0X_1X_4X_5 = IXXIIXX$
  - $S_{X2} = X_0X_2X_4X_6 = XIXIXIX$

- **Z-type stabilizers** (these detect X-errors), also derived from the rows of H:
  - $S_{Z0} = Z_0 Z_1 Z_2 Z_3 = IIIZZZZ$
  - $S_{Z1} = Z_0 Z_1 Z_4 Z_5 = IZZIIZZ$
  - $S_{Z2} = Z_0 Z_2 Z_4 Z_6 = ZIZIZIZ$

These $m = n - k = 7 - 1 = 6$ operators generate the stabilizer group S. The X-stabilizers commute with all Z-stabilizers. This is because for any row h_a from H defining an X-stabilizer and any row h_b from H defining a Z-stabilizer, the number of positions where both rows have a '1' (i.e., shared qubits) is even. This ensures that S_{Xa} and S_{Zb} commute.

Error Detection and Correction: A single Z-error on qubit j, Z_j, will anti-commute with a specific subset of $\{S_{X0}, S_{X1}, S_{X2}\}$. The 3-bit syndrome derived from measuring these X-stabilizers will uniquely identify j. For example, a Z_0 error anti-commutes with S_{X0}, S_{X1}, S_{X2}, giving an X-syndrome (eigenvalue flips) of $(1, 1, 1)$ (if 1 means flip). A Z_3 error anti-commutes only with S_{X0}, giving syndrome $(1, 0, 0)$. Each column of H represents the syndrome pattern for an error on the corresponding qubit. Similarly, a single X-error on qubit j, X_j, will anti-commute with a specific subset of $\{S_{Z0}, S_{Z1}, S_{Z2}\}$, and the 3-bit syndrome from these Z-stabilizers will identify j. A Y_j error will be caught by both sets of syndromes.

Logical Operators: A common choice for logical operators, which must commute with all stabilizers but are not themselves stabilizers, is:

- $\bar{X} = X_0 X_1 X_2 X_3 X_4 X_5 X_6$ (product of X on all 7 qubits)
- $\bar{Z} = Z_0 Z_1 Z_2 Z_3 Z_4 Z_5 Z_6$ (product of Z on all 7 qubits)

Minimum distance and error detection of the Steane code

Let's see why d for the $[[7, 1, 3]]$ bit-flip code is 3. You can skip this if you already familiar with it.

▶ Click to check

Exercise 2: Lookup table of [[7, 1, 3]] Steane Code

The goal is to create a Python dictionary that maps each possible 6-bit syndrome to a specific single-qubit error code (e.g., X0 is X error injected on the 0th qubit). If we assume syndrome measure results of each syndrome as `s0` ... `s5` = $S_{X0}, \ldots, S_{Z2}$, the final error if the syndrome is (s5, s4, s3, s2, s1, s0), it indicates an X error on qubit 0, and the corresponding dictionary entry should be "111000": "X0". Each value in the dictionary follows the format "Pq", where P is the Pauli operator (X, Y, or Z) and q is the qubit index from 0 to 6. If there is no error, the syndrome will be all zeros, and the correction should be "I" to indicate the identity (i.e., do nothing).

You are asked to complete this mapping for all possible single-qubit Pauli errors to determine the syndrome produced by X, Y, and Z errors, and assign the correct label in the dictionary. When you're done, your decoder will be able to take any 6-bit syndrome from a single error and output the appropriate corrective action.

(**Tip**) Try to fill out the commute/anti-commute table below for the X errors! Mark 0 for commute and 1 for anti-commute.

Error Code	Error Pauli String	S5(ZIZIZIZ)	S4(IZZIIZZ)	S3 (IIIZZZZ)	S2 (XIXIXIX)	S1 (IXXIIXX)	S0 (IIIX)
X_0	IIIIIIX	1(anti-commute)	1(anti-commute)	1(anti-commute)	0(commute)	0(commute)	0(comm
X_1	IIIIIXI						
X_2	IIIIXII						
X_3	IIIXIII						
X_4	IIXIIII						
X_5	IXIIIII						
X_6	XIIIIII						

```
# ---- TODO : Task 2 ---
# Fill in the other error codes
steane_decoder_syndrome_map = {
    '111000': 'X0',
    '': 'X1',
    '': 'X2',
    '': 'X3',
    '': 'X4',
    '': 'X5',
    '': 'X6',
    '': 'Y0',
    '': 'Y1',
    '': 'Y2',
    '': 'Y3',
    '': 'Y4',
    '': 'Y5',
    '': 'Y6',
    '': 'Z0',
    '': 'Z1',
    '': 'Z2',
    '': 'Z3',
    '': 'Z4',
    '': 'Z5',
    '': 'Z6',
    '000000': 'I'}
# --- End of TODO ---


# Submit your answer using following code
grade_lab4_ex2(steane_decoder_syndrome_map)
```

Exercise 3: Detect Error with a Lookup Table

Now, let's use the lookup table you've created to find errors.

Run the cell below to download a quantum state from the grader. This quantum state is a logical zero state of the Steane code, into which a specific error has been injected.

Your task is to:

1. Construct a quantum circuit.
2. Initialize the `qr_data` qubits to the provided quantum state.
3. Complete `measure_steane_syndrome` function. We left S0 as an example.
4. Compare the measurement results (the syndrome) with your lookup table to identify which qubit experienced what type of error.
5. Submit your findings to the grader for verification.

First, complete `measure_steane_syndrome` by preparing all the stabilizers of the Steane code.

```
def measure_steane_syndrome(qc, q_data, q_anc, c_reg):

    # Measure X-type Stabilizers (S0, S1, S2 -> q_anc[0], q_anc[1], q_anc[2])
    # Apply H to data qubits -> CNOT -> Apply H to data qubits -> Measure ancilla
    qc.h(q_data) # H on data qubits

    #S0: IIIXXXX (X_3 X_2 X_1 X_0)
    qc.cx(q_data[0], q_anc[0])
    qc.cx(q_data[1], q_anc[0])
    qc.cx(q_data[2], q_anc[0])
    qc.cx(q_data[3], q_anc[0])

    # ---- TODO : Task 3 ---
    # Fill in the required code for S1 and S2
    #S1

    #S2
    # --- End of TODO ---

    qc.h(q_data) # Restore H on data qubits
    qc.measure(q_anc[0:3], c_reg[0:3]) # Measure X syndrome (s1, s2, s3)
    qc.barrier()


    # ---- TODO : Task 3 ---
    # Measure Z-type Stabilizers (S3, S4, S5 -> q_anc[3], q_anc[4], q_anc[5])
    # CNOT -> Measure ancilla
    # Fill in the required code for S3, S4 and s5

    #S3

    #S4

    #S5
    # --- End of TODO ---

    qc.measure(q_anc[3:6], c_reg[3:6]) # Measure Z syndrome (s3, s4, s5)
    qc.barrier()
```

Let's initialize `qr_data` with the provided statevector (`State`) by using initialize of `QuantumCircuit`.

```
state = bring_states()

# Logical qubit (7 data qubits)
qr_data = QuantumRegister(7, name='q')
# Ancilla qubits for syndrome measurement (6)
qr_anc = QuantumRegister(6, name='anc')
# Classical registers for syndrome (initial & verify)
cr_initial_syn = ClassicalRegister(6, name='c_initial_syn')
cr_final_syn = ClassicalRegister(6, name='c_final_syn')

# Total circuit (13 qubits, 12 classical bits)
qc = QuantumCircuit(qr_data, qr_anc, cr_initial_syn, cr_final_syn)


# ---- TODO : Task 3 ---
#initialize qc with the correctState

# --- End of TODO ---
```

Next, add the syndrome measurement by executing the cell below and check your quantum circuit by drawing it.

```
# --- AddSyndrome Measurement ---

measure_steane_syndrome(qc, qr_data, qr_anc, cr_initial_syn)

qc.draw('mpl', fold=-1)
```

Now, run this quantum circuit to find the error injected into the logical $|0\rangle$ state. To complete this task, you'll need the key (the measured bitstring) from the `counts` dictionary. If the stabilizer measurements are implemented correctly, you will observe a single state with 100% probability in the simulation results.

To get a correct answer

If you run the circuit below on a real backend, you might not be able to obtain the error codes accurately due to various errors injected into the qubits. Therefore, please execute the following code using a simulator backend.

```
# --- Run the Simulation using AerSimulator
backend = AerSimulator()

#make quantum circuit compatible to the backend
pm = generate_preset_pass_manager( backend = backend, optimization_level=3)
qc_isa = pm.run(qc)

#run and get counts
sampler = Sampler(mode=backend)
counts = sampler.run([qc_isa], shots = 10000).result()[0].data.c_initial_syn.get_counts()
```

```
# ---- TODO : Task 3 ---
#get key of simulation result and find the error code, ex: X1

error_code =
# --- End of TODO ---


plot_histogram(counts)
```

Submit your error code string to the grader to see if you detected correctly.

```
# Submit your answer using following code
grade_lab4_ex3(error_code)
```

Congratulations! You've successfully implemented the stabilizer measurement circuit for the Steane code and detected the error.

Since Pauli errors are unitary operations and self-inverse, applying the same Pauli gate a second time acts as the identity operation ($P \cdot P = I$), effectively canceling the error. For this optional exercise (non-graded), use this property to correct the error you detected. Apply the gate corresponding to your detected error code to the erroneous quantum state and verify that the error is corrected.

No Grader Extra Exercise: Correct Error

Now let's correct this error with a proper gate operation.

Complete the cell below to add proper correction, and then do a stabilizer measurement one more time to get the syndrome measurement result: 000000, which means no error detected. Start by initializing `qr_data` with State again and apply a correction gate, then do a stabilizer measurement. You can use the `measure_steane_syndrome` function you already made.

```
qc = QuantumCircuit(qr_data, qr_anc, cr_initial_syn, cr_final_syn)


# ---- TODO : ungraded task ---
#initialize qr_data with State and correct error by applying proper gate



# --- End of TODO ---

measure_steane_syndrome(qc, qr_data, qr_anc, cr_final_syn)

#qc.draw('mpl', fold=-1)
```

To get a correct answer

If you run the circuit below on a real backend, you might not be able to obtain the error codes accurately due to various errors injected into the qubits. Therefore, please execute the following code using a simulator backend.

```
qc_isa = pm.run(qc)
counts = sampler.run([qc_isa], shots = 10000).result()[0].data.c_final_syn.get_counts()

plot_histogram(counts)
```

Did you obtain `000000` with 100% probability in your syndrome measurements? Congratulations 🎉! You now understand the Steane code and how to detect and correct errors.

So far, you've explored the fundamental concepts of both classical and quantum error correction. You've delved into the operational mechanics of two foundational quantum error correction codes, constructed lookup tables for error detection and correction, and practically applied these principles by detecting errors in the Steane code.

Chapter 3: Exploring Advanced Quantum Error Correction Codes & Their Efficiency

Having explored basic quantum error correction codes like the Steane code, this chapter delves into more advanced topics. Our primary goal is to understand different strategies for achieving robust fault tolerance and to investigate what improvements advanced QLDPC constructions (such as the gross code and related families) offer over other well-known codes like the surface and toric codes, particularly in terms of error correction capabilities and efficiency.

We will explore codes with particular topologies, where the geometric layout and connectivity of qubits influence the code's properties. We will analyze instances of the toric code and compare them with the gross code. For these, our focus will be on aspects like their parity check matrices and the number of logical qubits each can encode relative to physical qubits and distance. This exploration will highlight different strategies for achieving fault tolerance and shed light on the diverse landscape of contemporary QEC research.

Note on Code Representations and a Pedagogical Approach

It's essential to reiterate that the primary objective here is to understand comparative principles. The specific examples or graphical representations used to illustrate differences between code families (e.g., surface/toric vs. gross-like codes) should be understood as **illustrative toy models**. They are designed to capture essential characteristics and distinguishing features that lead to different performance potentials, rather than being the exact process of building codes.

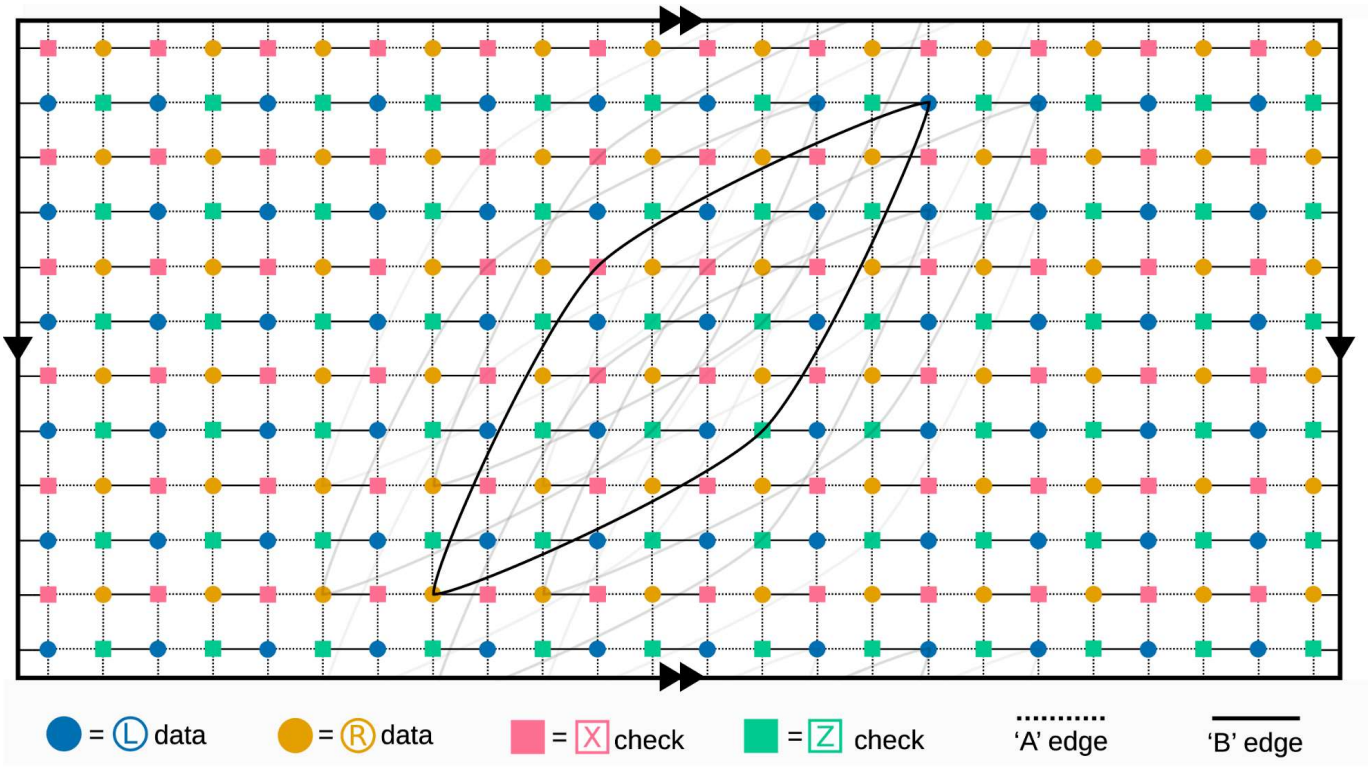
3.1 Foundational Concepts and Key QLDPC Architectures for Comparison

Before directly introducing the toric and gross codes, this section will give you a high-level introduction to the *Quantum Low-Density Parity-Check (QLDPC) codes* and introduce key QLDPC architectures.

Low-Density Parity-Check (LDPC) codes are classical error-correcting codes known for having a sparse parity check matrix, meaning the matrix contains very few non-zero entries relative to its size. This sparsity is crucial because it allows for highly efficient decoding algorithms, making LDPC codes powerful tools for classical communication and data storage. Quantum Low-Density Parity-Check (QLDPC) codes are the quantum counterparts to classical LDPC codes. In QLDP codes, the **stabilizer generators are sparse**; each stabilizer generator acts non-trivially on only a small number of qubits, and each qubit is only a part of a small number of these generators. The aim is to design codes that not only feature this beneficial sparsity but also achieve good scaling of code distance d (a measure of error correction capability) with the number of physical qubits n, and ideally offer high **encoding rates** (k/n, where k is the number of logical qubits).

Now that we've covered these basics, we'll look at and compare the two main types of QLDP codes:

- **Surface/Toric Codes**: These are well-studied QLDP codes defined on a 2D lattice where stabilizers act geometrically **locally** (typically on nearest-neighbor qubits). While known for relatively high error thresholds and compatibility with 2D hardware, their code distance d typically scales as $O(\sqrt{n})$ for n physical qubits, which can be a limitation for achieving very large distances efficiently - additionally, for each patch of surface (or toric) code, there is only 1 logical qubit (2 in the case of the toric code).
- **Bivariate Bicycle Codes (and related QLDPC families)**: This broader class of QLDPC constructions aims for excellent asymptotic scaling of parameters k, d, n. Their construction does not necessarily restrict them to simple 2D geometric locality. The Tanner graphs of these codes might exhibit more complex or "**long-range**" (in a 2D sense) connections, while still maintaining overall sparsity. This different structural connectivity is believed to underpin their potential for superior performance, such as:
    - Better distance scaling: Achieving a code distance d that grows more favorably with n.
    - Higher encoding rates (k/n): Encoding more logical qubits for a similar number of physical qubits and distance.



(Image taken from https://arxiv.org/pdf/2308.07915: Tanner graph representation for the gross code, illustrating data qubits (circles) and stabilizers/checks (squares) on a toroidal grid.)

3.2 Qubit Layout and Conventions for Exercises

To concretely compare how different code structures impact performance with simplified models, we will construct codes on the same physical arrangement of qubits by using above image. We will build upon concepts discussed in the lecture "Toward Fault-tolerant Quantum Computing with IBM QLDPC codes" by Patrick Rall.

Our focus will be on constructing the parity check matrices for two illustrative code instances on a two-dimensional grid with periodic boundary conditions (a torus), as visualized in the provided Tanner graph:

1. One representing a **Toric code** with local connections.
2. Another representing a **gross code** that incorporates additional long-range connections.

Key Elements in Our Model:

- **Data Qubits (Circles)**: Store the quantum information. In the provided diagram, these are the blue and orange circles. There are $12 \times 6 = 72$ blue qubits and $12 \times 6 = 72$ orange qubits, totaling $n = 144$ data qubits.
- **Stabilizers/Checks (Squares)**: Define the code's stabilizer operators.
  - **X-stabilizers** (products of Pauli-X) are associated with the **red** squares.
  - **Z-stabilizers** (products of Pauli-Z) are associated with the **green** squares.
- **CSS Code Structure**: We are working within the Calderbank-Shor-Steane (CSS) framework.

A clear **qubit labeling** convention is crucial for translating the Tanner graph's connectivity into parity check matrices. The rows of these matrices will correspond to stabilizers, and columns to data qubits.

Qubit Labeling Convention (144 total data qubits):

- **Blue** Data Qubits (Indices 0-71): Labeled column by column (left to right), then bottom-up within each column (0-indexed).
    - The bottom-leftmost blue qubit is `0`; the one above it is `1`, up to `5`.
    - The next column's blue qubits are `6` through `11`, and so on.
- **Orange** Data Qubits (Indices 72-143): Follow the same labeling scheme, starting with the bottom-leftmost orange qubit as `72`.

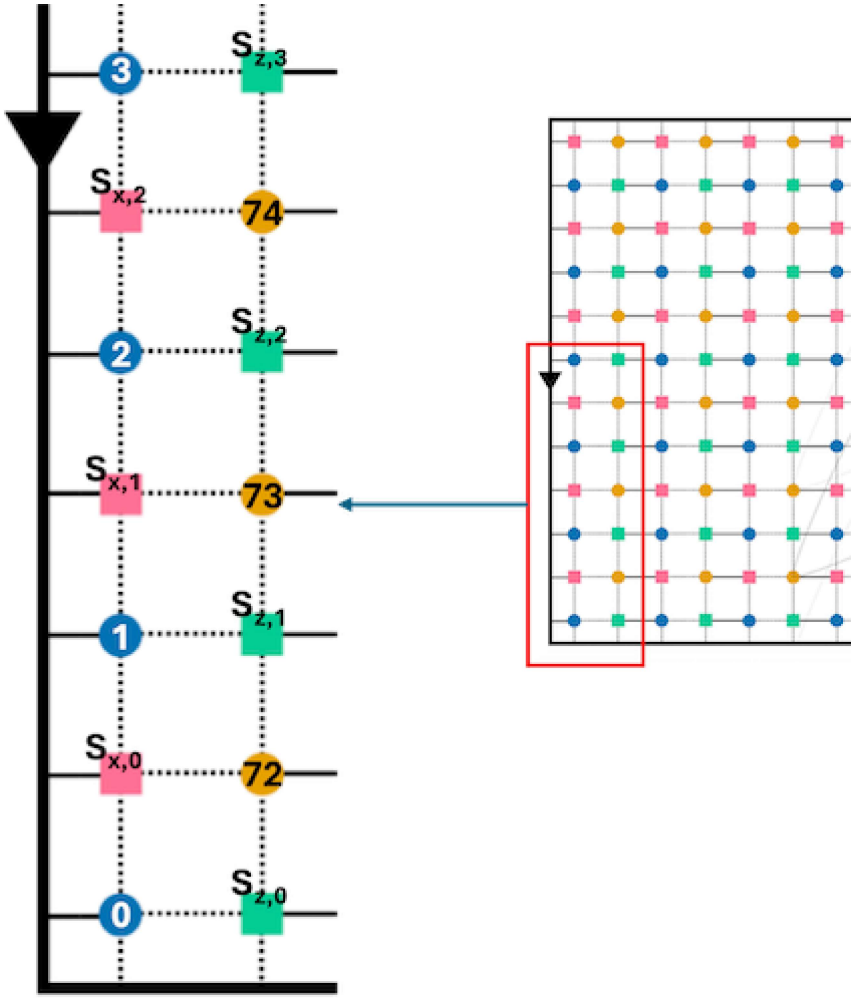
Parity Check Matrix Convention (for this lab):

We will define two binary parity check matrices:

- H_X **(for X-stabilizers)**:
    - Each row represents an X-stabilizer (derived from a **red square**).
    - This matrix H_X is used to detect **Z-type errors**. (Syndrome: s_x)
- H_Z **(for Z-stabilizers)**:
    - Each row represents a Z-stabilizer (derived from a **green square**).
    - This matrix H_Z is used to detect **X-type errors**. (Syndrome: s_z)

Using the layout and conventions above, we will now define stabilizers based on the connectivity depicted in the provided Tanner graph for two distinct code versions on the *same* physical qubit layout as the exercises.

1. **The toric code:** This version will be constructed using only local, nearest-neighbor connections between the stabilizers (squares) and the data qubits (circles) they act upon, following the standard toric code model on a grid with periodic boundaries. Note: the toric code is very similar to the surface code; however, it is embedded on a torus with periodic boundaries in both the x and y directions. In particular, it can encode one additional qubit when compared to the surface code without the periodic boundaries.
2. **The gross code:** This version will augment the local connections typical of the toric code by including specific "long-range" connections, the leaf-shaped connections in the above graph. These additional connections modify the definitions of the stabilizer generators, leading to a code with different parameters and properties compared to the standard toric code on the same qubit layout.



3.3 Toric Code Exercise

HX

We will now establish the binary representation for the **X-stabilizers** (associated with red squares) for the toric code, which considers only nearest-neighbor connections on the torus. These will form the rows of the matrix you'll be asked to construct (referred to as H_X in the exercise prompt's output variable).

Let's consider the bottom-leftmost red square in the diagram. This X-stabilizer acts on four neighboring data qubits:

1. The blue data qubit below it: This is blue qubit `0` (column 0, row 0).
2. The blue data qubit above it: This is blue qubit `1` (column 0, row 1).
3. The orange data qubit to its right: This is the bottom-leftmost orange qubit, which is orange qubit `72` (orange column 0, row 0).
4. The orange data qubit to its left (across the periodic boundary): This is the bottom-rightmost orange qubit. Orange qubits are indexed 72-143. There are 12 columns of orange qubits, 6 qubits per column. The last orange column (column 11) contains qubits $72 + 11 \times 6 +$ row_idx. The bottom-rightmost is $72 + 11 \times 6 + 0 = 72 + 66 = 138$. So, this is orange qubit `138`.

Therefore, the first X-stabilizer acts on data qubits `0`, `1`, `72`, and `138`. Its corresponding row in the binary matrix (which will be a row in your H_X for the exercise, or H_Z in standard notation) will have 1s at columns 0, 1, 72, and 138, and 0s elsewhere. This 144-element row vector is:

```
[1 1 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
 1 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 1 0 0 0 0 0 0]
```

The example given in the original problem for the first 6 rows of "parity check matrix H_X" (derived from the first 6 red squares) uses this logic:

```
[[1 1 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
  1 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 1 0 0 0 0 0 0],

 [0 1 1 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 1 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 1 0 0 0 0 0 0],

 [0 0 1 1 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 1 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 1 0 0 0 0 0 0],

 [0 0 0 1 1 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 1 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 1 0 0 0 0 0 0],

 [0 0 0 0 1 1 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 1 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 1 0 0 0 0 0 0],

 [1 0 0 0 0 1 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 1 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 1 0 0 0 0 0 1]]
```

HZ

HZ follows the same basic rule as HX for connecting to its nearest neighbor. The only difference is that HZ is marked with a green square, and it starts at the very bottom of the second column from the left of the entire grid.

Let's consider the bottom-leftmost green square in the diagram. This Z-stabilizer acts on four neighboring data qubits:

1. The blue data qubit to its left: This is blue qubit `0` (blue column 0, row 0).
2. The blue data qubit to its right: This is blue qubit `6` (blue column 1, row 0).
3. The orange data qubit above: This is the bottom-leftmost orange qubit, which is orange qubit `72` (orange column 0, row 0).
4. The orange data qubit below (across the periodic boundary): This is the top-leftmost orange qubit. So, this is orange qubit `77`, which is 5 rows above the orange qubit `72` (orange column 0, row 5)

Therefore, the first Z-stabilizer acts on data qubits `0, 6, 72`, and `77`. Its corresponding row in the binary matrix. This 144-element row vector is:

```
[1 0 0 0 0 0 1 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
 1 0 0 0 0 1 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]
```

Exercise 4: Find the parity check matrices of the toric code

Following the above convention for qubit labeling and the definition of stabilizers from the diagram, find the parity check matrices for the **toric code** (i.e., the code containing only nearest-neighbor connections).

Output the result for the **X-stabilizers (red squares)** in a NumPy array labeled H_X and similarly for the **Z-stabilizers (green squares)** in a NumPy array labeled H_Z.(As per our clarification note, the matrix H_X you generate from red squares will be used to detect Z-errors, and H_Z from green squares will detect X-errors).

Please ensure that the NumPy arrays H_X and H_Z strictly adhere to the expected dimensions and structure (`np.zeros((72,144),dtype=int)`). Modifying the size or structure of these matrices from what is anticipated by the exercise might lead to a failure during grading.

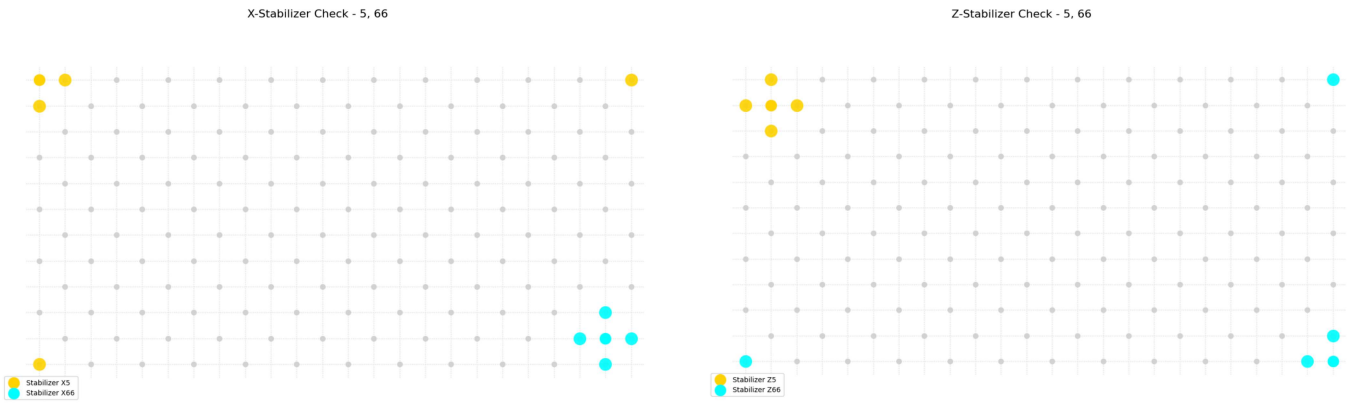
Hint: Notice the periodic structure in the examples. Can this be used to your advantage to write an efficient piece of code to generate these matrices?

Check the connectivity using a helper function!

Once the parity check matrices are finalized, execute the provided code to verify the connectivity of the 5th stabilizer of HXtc and the 66th stabilizer of `HXtc` and `HZtc`, respectively. Please focus on carefully checking the boundary conditions. Success is indicated by generating a graph that matches the following example.

```
from lab4_util import generate_stabilizer_plots
generate_stabilizer_plots(HXtc, HZtc)
```

Acknowledgement: Prof. Daniel Sierra-Sosa, Qiskit Advocate and a Mentor of QGSS 2025), for the discussion and his contribution on making a helper function.




```
# Some helpful code to start Exercise 4.


# We will define the parity check matrices for the toric code
HXtc = np.zeros((72, 144),dtype=int) # initializing the matrices
HZtc = np.zeros((72, 144),dtype=int)

# We will ask you to modify the matrices by adding 1s in appropriate places
# As an example, we will show how to do so for the first few rows of the toric code


# ---- TODO : Task 4 ---
# Write code to calculate HXtc and HZtc

# 0-th stabilizer
HXtc[0][0] = 1
HXtc[0][1] = 1
HXtc[0][72] = 1
HXtc[0][138] = 1

# 1-st stabilizer
HXtc[1][1] = 1
HXtc[1][2] = 1
HXtc[1][73] = 1
HXtc[1][139] = 1

# 2-nd stabilizer
HXtc[2][2] = 1
HXtc[2][3] = 1
HXtc[2][74] = 1
HXtc[2][140] = 1
```

```
# 3-rd stabilizer
HXtc[3][3] = 1
HXtc[3][4] = 1
HXtc[3][75] = 1
HXtc[3][141] = 1

# 4-th stabilizer
HXtc[4][4] = 1
HXtc[4][5] = 1
HXtc[4][76] = 1
HXtc[4][142] = 1

# 5-th stabilizer
HXtc[5][5] = 1
HXtc[5][np.mod(6,6)] = 1
HXtc[5][72 + 5] = 1
HXtc[5][72 + 6*np.mod(-1,12) + 5] = 1

# The last definition suggested a general rule for finding the appropriate locations to p
# HXtc[j][j] = 1
# HXtc[j][6*np.floor(j/6) + np.mod(j+1,6)]
# HXtc[j][j+72] = 1
# HXtc[j][72 + 6*np.mod(np.floor(j/6)-1,12) + np.mod(j,6)]

# Write a loop over j from 0 to 143 to set all of the rows.

# Inspired from this, write a similar loop for the Z parity check matrices and complete H

HZtc[0][0] = 1
HZtc[0][6] = 1
HZtc[0][72] = 1
HZtc[0][77] = 1




# --- End of TODO ---


# Check the connectivity of HXtc and HZtc

from lab4_util import generate_stabilizer_plots
generate_stabilizer_plots(HXtc, HZtc)


# Submit your answer using following code
grade_lab4_ex4(HXtc, HZtc)
```

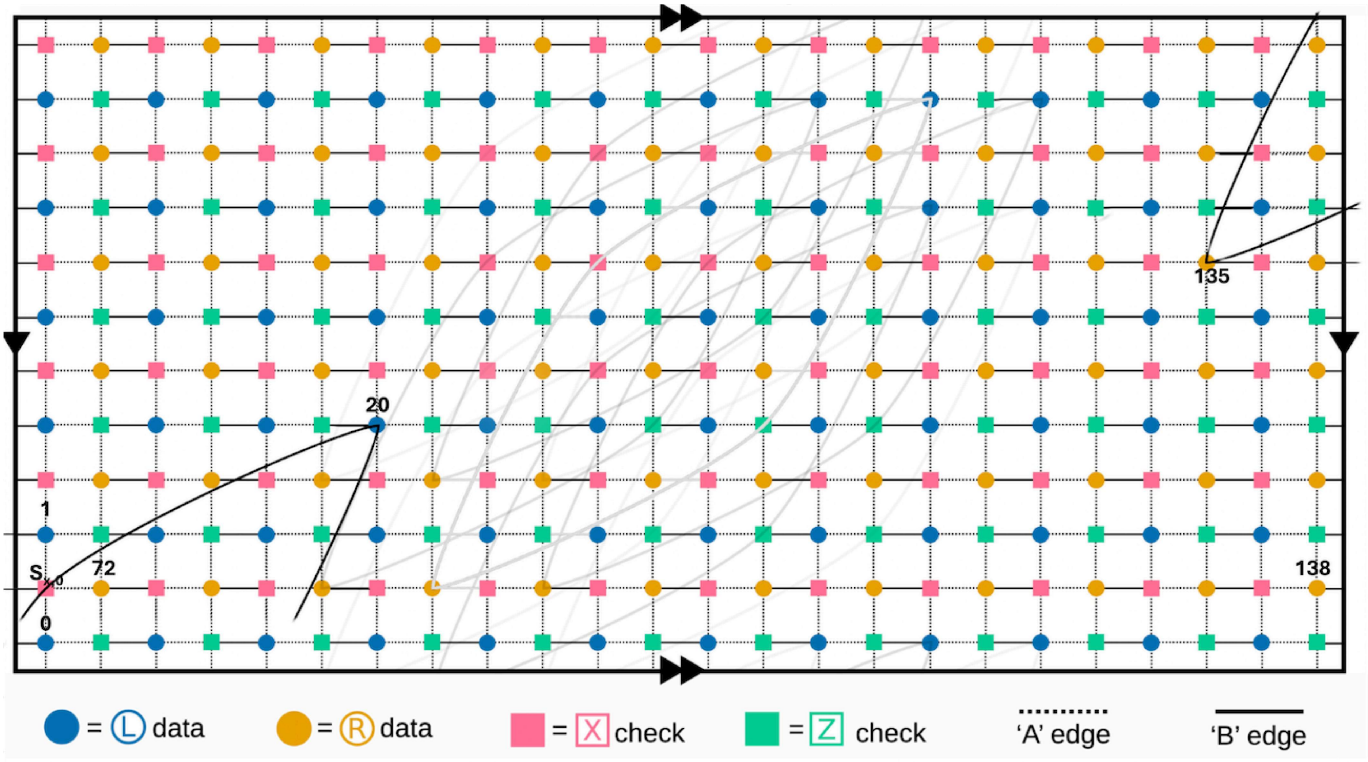
3.4 Gross Code Exercise

Now we consider the parity check matrices for the gross(-like) code. This code is generated by including the long-range connections shown abstractly in the figure (e.g., the leaf-like diagonal connections, though the exact rules are specific to the code's definition). The fundamental qubit layout and labeling convention remain the same as for the toric code.

The key difference for the gross code is that each stabilizer (square check) will now have two additional long-range connections on top of its four nearest-neighbor connections. This means each stabilizer in the gross code will act on a total of $4 + 2 = 6$ data qubits. Consequently, each row in the parity check matrices (H_X and H_Z) for the gross code will have six '1's.

A practical way to construct these matrices is to start with the toric code's connectivity (nearest-neighbor connections) and then add the two specific long-range connections for each stabilizer.

Let's illustrate this for the first row of H_X, which corresponds to the X-stabilizer at the bottom-leftmost red square (conceptually at $c_s = 0, r_s = 0$).



HX

1. Nearest-Neighbor Connections (from Toric Code): As established in Section 3.3, this bottom-leftmost red square has nearest-neighbor connections to:

- Blue data qubit at index $\mathbf{0}$ (visualized as "below" or part of the square, blue column 0, row 0).
- Blue data qubit at index $\mathbf{1}$ (visualized as "above" or part of the square, blue column 0, row 1).

- Orange data qubit at index $\mathbf{72}$ (visualized as "to the right" or part of the square, orange column 0, row 0).
- Orange data qubit at index $\mathbf{138}$ (visualized as "to the left" across periodic boundary, orange column 11, row 0).

2. Additional Long-Range Connections (Specific to this Gross Code instance): The problem states that for this particular gross code construction, the bottom-leftmost red square ($c_s = 0, r_s = 0$) will have two additional long-range connections:

- First long-range connection: To the blue data qubit 20.
    - The text describes this as "the blue qubit three rows above and six columns to the right." The specific rules of the gross code define how these relative positions map to a precise qubit index on the torus. For this stabilizer, this rule results in a connection to blue data qubit with global index $\mathbf{20}$.
- Second long-range connection: To the orange data qubit 135.
    - The text describes this as "the orange qubit three columns to the left and six rows down (recall the periodic boundary conditions)." Again, these relative descriptions are specific to the gross code's connection rules for this stabilizer, resulting in a connection to orange data qubit with global index $\mathbf{135}$.

3. Resulting First Row of H_X for the Gross Code: Combining both nearest-neighbor and the two specified long-range connections, the bottom-leftmost X-stabilizer now acts on data qubits with indices: `0, 1, 20, 72, 135, 138`.

Therefore, the first row of the H_X matrix for this gross code (a 144-element binary vector) will have '1's at these six positions and '0's elsewhere. Let's verify this against the provided snippet:

Snippet:

```
[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 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
 1 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 1 0 0 1 0 0 0 0 0]
```

This confirms that the snippet for the first row of H_X for the gross code has exactly six '1's at the global data qubit indices 0, 1, 20, 72, 135, and 138, matching the combination of nearest-neighbor and the specified long-range connections.

HZ

To find the H_Z, please revisit the Tanner graph, and take a close look at the `leaf` shape of the long-range connection.

1. Nearest-Neighbor Connections (from Toric Code): As established in Section 3.3, this bottom-leftmost red square has nearest-neighbor connections to:

- Orange data qubit at index 77 (visualized as "to the below" across periodic boundary, orange column 0, row 5).
- Orange data qubit at index 72 (visualized as "above" or part of the square, orange column 0, row 0).
- Blue data qubit at index 6 (visualized as "to the right" or part of the square, blue column 1, row 0).
- Blue data qubit at index 0 (visualized as "to the left" or part of the square, blue column 0, row 0).

2. Additional Long-Range Connections (Specific to this Gross Code instance):

- First long-range connection: To the blue data qubit 15.
    - The text describes this as "the blue qubit six rows above and three columns to the right." The specific rules of the gross code define how these relative positions map to a precise qubit index on the torus. For this stabilizer, this rule results in a connection to blue data qubit with global index 15.
- Second long-range connection: To the orange data qubit 130.
    - The text describes this as "the orange qubit six columns to the left and three rows down (recall the periodic boundary conditions)." Again, these relative descriptions are specific to the gross code's connection rules for this stabilizer, resulting in a connection to orange data qubit with global index 130.

3. Resulting First Row of H_Z for the Gross Code: Combining both nearest-neighbor and the two specified long-range connections, the bottom-leftmost X-stabilizer now acts on data qubits with indices: `0, 6, 15, 72, 77, 130`.

Snippet:

```
[1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 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
 1 0 0 0 0 1 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 1 0 0 0 0 0 0 0 0 0 0]
```

This confirms that the snippet for the first row of H_Z for the gross code has exactly six '1's at the global data qubit indices 0, 6, 15, 72, 77, and 130, matching the combination of nearest-neighbor and the specified long-range connections.

This process of adding two specific long-range connections would be repeated for *every* stabilizer (both X-type from red squares and Z-type from green squares) to transform the toric code matrices into the gross code matrices. The exact rules for determining the long-range partners for other stabilizers would follow a consistent pattern defined by the gross code's construction on this lattice.

Exercise 5: Find the parity check matrices of the gross code

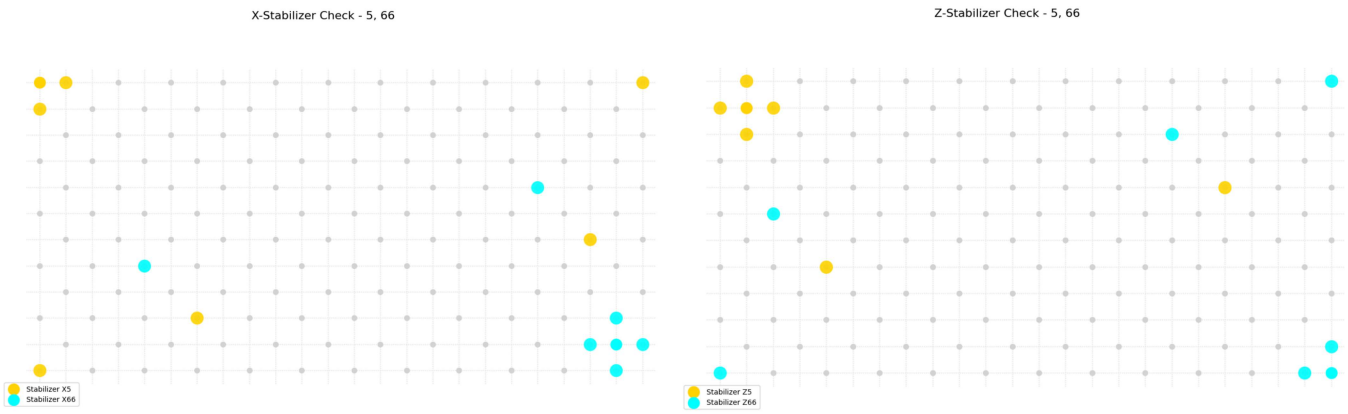
Following the above labeling convention, construct the parity check matrix for the gross code, that is, including the long-range connections. Output the result for the X stabilizers in numpy array labeled H_X and similarly for the Z stabilizers in an array labeled H_Z.

Hint: Notice the periodic structure of the above examples. Can this be used to your advantage to make an efficient piece of code to generate the matrices? Each row should be weight-6 - that is, contain six 1s.

Check the connectivity using a helper function!

Like exerciese 4 after finish compose `HXgc` and `HZgc`, execute the provided code to verify the connectivity of the 5th stabilizer of HXtc and the 66th stabilizer. Please focus on carefully checking the boundary conditions. Success is indicated by generating a graph that matches the following example.

```
from lab4_util import generate_stabilizer_plots
generate_stabilizer_plots(HXgc, HZgc)
```




```
# We will define the parity check matrices for the gross code
HXgc = np.zeros((72,144),dtype=int) # initializing the matrices
HZgc = np.zeros((72,144),dtype=int)
# ---- TODO : Task 5 ---
# Write code to calculate HXgc and HZgc

# 0-th X stabilizer
HXgc[0][0] = 1
HXgc[0][1] = 1
HXgc[0][20] = 1
HXgc[0][72] = 1
HXgc[0][135] = 1
HXgc[0][138] = 1


# 0-th Z stabilizer
HZgc[0][0] = 1
HZgc[0][6] = 1
```

```
HZgc[0][15] = 1
HZgc[0][72] = 1
HZgc[0][77] = 1
HZgc[0][130] = 1




# --- End of TODO ---


#Check stabilizer

generate_stabilizer_plots(HXgc, HZgc)


# Submit your answer using following code
grade_lab4_ex5(HXgc, HZgc)
```

3.5 Counting the Number of Logical Qubits

The parity check matrices you've been working with, H_X and H_Z, are constructed from the stabilizers of the code. Each row in these matrices represents a generator of the stabilizer group. However, it's possible that some of these chosen generators are not independent; one stabilizer generator might be a product of others. Redundant generators don't add new constraints to define the codespace.

To find the actual number of independent stabilizer generators, we calculate the **rank** of these parity check matrices. Since we are dealing with qubits and Pauli operators (which square to identity and whose effects are often considered over GF(2) in the context of their binary representation), all matrix operations for rank calculation are performed modulo 2. The rank tells us the true number of unique conditions imposed by the stabilizers.

In general, for a quantum stabilizer code: If n is the number of physical data qubits used in the code, and r is the total number of independent stabilizer generators, then the number of logical qubits (k) encoded by the code is given by:

$$k = n - r$$

Each independent stabilizer generator effectively halves the dimension of the Hilbert space, "fixing" one degree of freedom. Thus, r independent stabilizers reduce the 2^n-dimensional space of n physical qubits to a 2^{n-r}-dimensional codespace, which can then encode $k = n - r$ logical qubits.

For CSS codes, like the toric and gross codes we are considering, the X-stabilizers and Z-stabilizers form two distinct, commuting groups. This allows us to count their independent generators separately. If r_X is the number of independent X-stabilizers (obtained from the rank of the matrix whose rows are the X-stabilizer generators, which you've labeled H_X from red

squares) and r_Z is the number of independent Z-stabilizers (obtained from the rank of the matrix whose rows are the Z-stabilizer generators, labeled H_Z from green squares), then the number of logical qubits in the CSS code is:

$$k = n - r_X - r_Z$$

This formula holds because the X-stabilizers and Z-stabilizers are chosen such that they all commute with each other, and the total number of independent conditions imposed on the codespace is the sum of the independent X-type and Z-type conditions.

Exercise 6: Count the number of logicals for the toric and gross codes

You will now use the `matrixRank` function to determine the number of logical qubits for both the toric code and the gross code you constructed in Exercises 4 and 5.

1. Import the function: Start by importing the `matrixRank` function using the command: `from lab4_util import matrixRank` This function calculates the rank of a binary matrix (here `HXtc`, `HZtc`, `HXgc` and `HZgc`), which is exactly what we need for our parity check matrices.
2. Calculate Ranks:
   - For the toric code, use `matrixRank` to find the rank of H_X (the matrix of X-stabilizers you generated from red squares in Exercise 4) – this will give you $r_{X,\text{toric}}$.
   - Similarly, find the rank of H_Z (the matrix of Z-stabilizers from green squares in Exercise 4) – this will give you $r_{Z,\text{toric}}$.
   - Repeat this process for the gross code matrices (H_X and H_Z) you generated in Exercise 5 to find $r_{X,\text{gross}}$ and $r_{Z,\text{gross}}$.
3. Calculate Logical Qubits (k):
   - Using the formula $k = n - r_X - r_Z$ (where $n = 144$ is the total number of data qubits), calculate the number of logical qubits for the toric code (k_{toric}).
   - Perform the same calculation for the gross code (k_{gross}).

Grading: Submit the calculated number of logical qubits (k) for both the toric code and the gross code (`k_toric` and `k_gross`). You can verify your toric code result against the known fact that it encodes $k = 2$ logical qubits.

```
# ---- TODO : Task 6 ---
# Write code to calculate k_toric and k_gross
# Hint: You can use the matrixRank imported from lab4_util

#toric code
rx_toric =
rz_toric=
k_toric =

# gross code
rx_gross=
rz_gross=
```

```
k_gross =

# Submit your answer using following code
grade_lab4_ex6(k_toric, k_gross)
```

3.6 Concluding remarks: The power of the connectivity

These exercises demonstrate a key principle in Quantum Error Correction: strategic modifications to a code's underlying structure, such as introducing long-range connections, can significantly enhance its performance.

The construction of parity check matrices for both the toric-like and gross-like codes on the identical 144-data-qubit framework has been completed. Your calculations in Exercise 6 for the number of logical qubits (k) illustrate how these differing connectivities influence k_{gross} relative to the standard k_{toric}.

However, a more pronounced advantage of the gross code variant is evident in its **code distance** (d), a critical parameter dictating its error correction strength. While detailed distance calculations are beyond this lab's immediate scope, but for the above code instances the toric code would be of distance 6 (as the periodicity along one of the directions is 6 lattice spacing in terms of the number of data qubits) - however, the gross code would have distance 12!

This substantial improvement in error correction capability, achieved by adding specific long-range connections, underscores the impact of such structural modifications on code performance, particularly in enhancing protection against errors. This highlights a valuable strategy in the design of advanced quantum codes.

```
# Check your submission status with the code below
from qc_grader.grader.grade import check_lab_completion_status

check_lab_completion_status("qgss_2025")
```

Additional information