A Methods Focused Guide to Quantum Error Correction and Fault-Tolerant Quantum Computation

A Methods Focused Guide to Quantum Error Correction and Fault-Tolerant Quantum Computation

A Methods Focused Guide to Quantum Error Correction and Fault-Tolerant Quantum Computation

Abdullah Khalid

What will this book teach you?

This mini-book limits itself to qubit stabilizer codes, which are among the most actively researched quantum error-correction codes. I wrote this mini-book more to teach myself than anyone else. The main questions that I wanted answered in this book are, how can I

- Construct encoding, decoding and syndrome measurement circuits for stabilizer codes?
- Algorithmically generate fault-tolerant quantum circuits for stabilizer codes?
- Numerically compute the fault-tolerance thresholds of these circuits?

What are the pre-requisites to this book?

You should be familiar with the fundamentals of quantum computing model. In particular, you should be comfortable with

- The quantum circuit model and aware of the action of common gates.
- The matrix representation of quantum gates and their connection with the truth table of the gates. This includes familiarity with linear algebra.
- Any of the quantum computing sdks (qiskit, cirq, etc). These are not directly used, but experience will help with the programming in the book.

Why should you read this book?

As an introductory guide to the subject matter, this mini-book is worth paying attention to because of two reasons:

- This book focuses deeply on methods and algorithms. This is unlike many other introductions which limit themselves to abstract discussions, simple examples, and mathemtical proofs. While theorems and their proofs are an essential part of the book, the author believes that intuition and understanding cannot be built without getting one's hands dirty in algorithmic implementations and mathematical calculations.
- This book is highly interactive. Each section is accompanied by mathematical or programming tasks that enhance the user's understanding of the material. Often, essential parts of the arguments are to be filled in by reader in these tasks.

  While an online copy is viewable to entice potential readers, the source of this book - a set of Jupyter notebooks - should be downloaded and interactively read.

To download the Jupyter notebooks, please clone the repository

```
git clone https://github.com/abdullahkhalids/qecft.git
```

Or download the repository as a zip file and extract it. After this open `contents.ipynb`.

Stac, a software library for this book

Stac stands for **Sta**bilizer **c**odes, and this library is designed to work with them. Many of the algorithms that I introduce in the book are implemented within this library. It is a very simply library and easy to pick up as you read the book. However, many of the tasks and exercises in the book can be completed in pure python and your favorite quantum SDK.

Citing

To cite this book, please use the following bibtex entry

```
@book{abdullahkhalid2023,
  title = {A Methods Focused Guide to Quantum Error Correction and Fault-Tolerant Quantum
Computation},
  author = {Abdullah Khalid},
  year = 2023,
}
```

Acknowledgements

The initial drafts of this book were produced with the help of a grant by the Unitary Fund.

Stac installation, and a short guide

Stac is a python package. It has a number of dependencies, the most important of which are

- qiskit
- stim
- a working latex installation [if you want pretty latex figures]

To install stac, please run the following cell

```
!pip install git+https://github.com/abdullahkhalids/stac
```

If you don't have git installed, you can also just download the repository directly, and run `!pip install /path/to/repo/dir`.

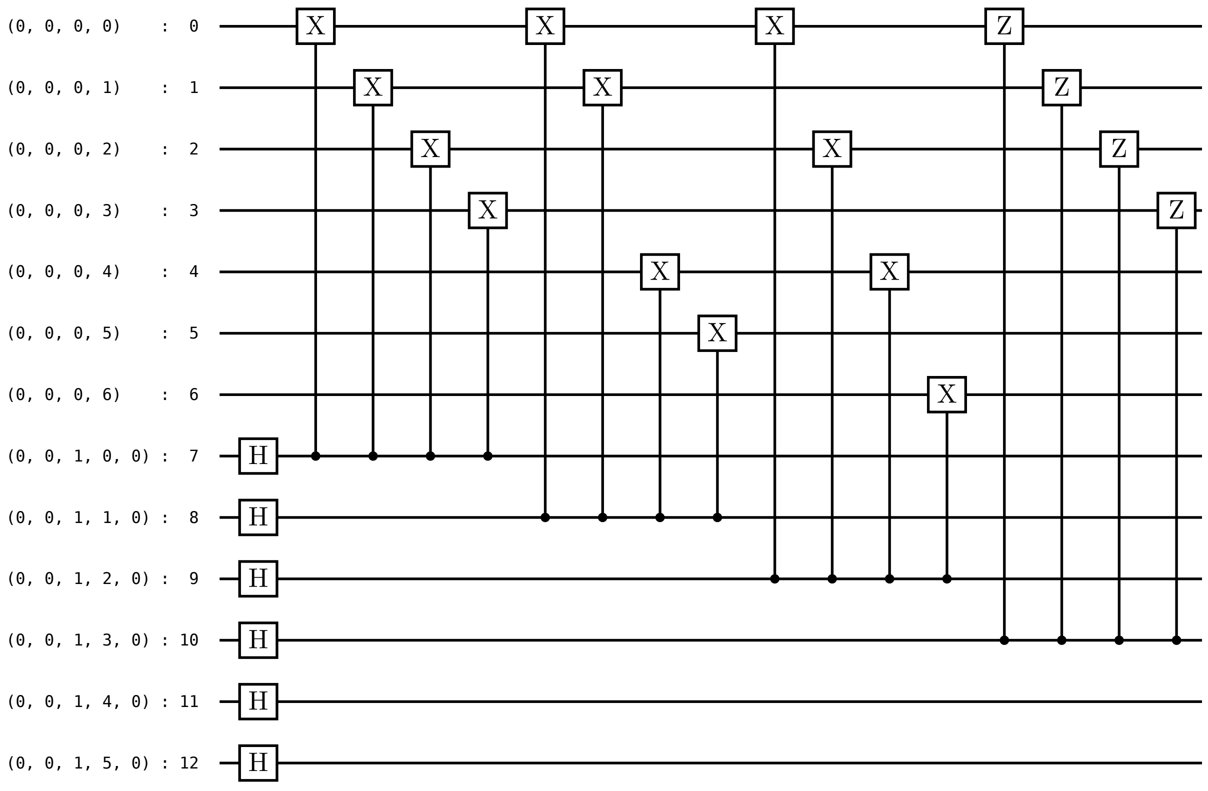
Testing the installation

The following commands show you how to create a quantum code. You will understand what they mean as you read the book. Right now, just make sure the commands work.

```
import stac
cd = stac.CommonCodes.generate_code("[[7,1,3]]")
cd.generator_matrix
```

```
array([[1, 1, 1, 1, 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, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 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, 1, 0, 1, 0, 1]])
```

```
circ = cd.construct_syndrome_circuit()
circ.draw()
```



```
# stac uses qiskit to simulate circuits
# if this fails, qiskit is not properly installed
circ.simulate()
```

```
        basis    amplitude
-------------  -----------
0000000110000        0.354
1111000110000       -0.354
1100110110000       -0.354
0011110110000        0.354
1010101110000        0.354
0101101110000       -0.354
0110011110000       -0.354
1001011110000        0.354
```

```
# it uses stim to sample from circuits
# if this fails, stim is not properly installed
circ.sample()
```

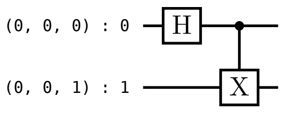
```
[1 1 1 0 0 0]
```

If the above commands work, then stac, and its dependencies, are properly installed on your machine.

A guide to stac quantum circuits

Stac allows you to create quantum circuits. Here is a short guide on the syntax stac uses.

```
import stac
# create circuit object.
# specify the number of qubits in the circuit
circ = stac.Circuit.simple(2)
# operations are added with the append method
circ.append('H', 0)
circ.append('CX', 0, 1)

# you can draw the circuit using the draw method
circ.draw()
```



```
# you can simulate a circuit using qiskit's statevector simulator like so
# this will present the output state in a nice readable form
circ.simulate()
```

```
  basis    amplitude
-------  -----------
     00        0.707
     11        0.707
```

```
# or you can use stim https://github.com/quantumlib/Stim/
# to sample from the circuit

# first add measurements to the circuit
circ.append('M', 0)
circ.append('M', 1)
# then sample 10 times
circ.sample(10)
```

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

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

Further documentation on stac

The latest documentation on stac will be available from the github repository.

Currently, the main reference documentations are

- Api reference
- Operations reference

Why error-correction and a model for it

Why do we need error-correction?

All computational and communication tasks are done with physical apparatus, and such apparatus is often noisy and unreliable. This means that there is a non-zero chance that the apparatus makes errors in the execution of its processes. One solution to this problem is to try and manufacture better devices. But this is not always possible, either because there is no known way of making better devices or because it is far too costly to make near perfect devices.

Thankfully, we have discovered procedures and algorithms that can be used to increase the probability that our noisy hardware outputs the correct results, with a modest increase in costs. The field of study that deals with such procedures and algorithms is called error-correction.

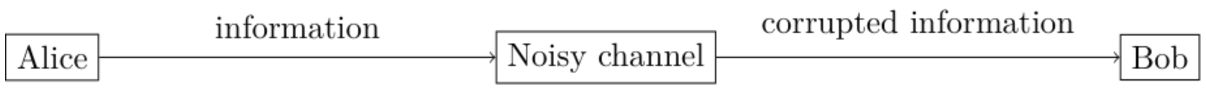
This field first emerged in tandem with the development of analog and digital classical communication and computation devices in the middle of the twentieth century. Today, classical error-correction is a mature field, whose results are applied in a wide variety of circumstances. For example, if the computer you are using to read this book is connected to the internet via Wi-Fi, then there is error-correcting algorithms being employed by your computer and the router to detect if any of the received packets were corrupted during transmission.

On the other hand, the field of quantum error-correction, which concerns error-correction techniques on quantum communication and computing devices, began in the late 1990s. Since then it has made rapid advancements, and many techniques and concrete algorithms have been discovered. This is very encouraging, because the hardware for quantum computers is so small and sensitive, it seems unlikely that we will ever be able to make physical error rates small enough to build quantum computers without the need for error-correction.

A model for error-correction problems

To understand the field, it is best to start with a simple problem that captures the essence of the field. This simple problem for error-correction, both classical and quantum, has an abstract model which we now present.

Alice wants to send information to Bob via a transmission line or channel connecting them. The information could be a string of bits in the classical case, or a string of qubits in the quantum case. Pictorially, we have the following scenario.



As you can see, we assume that the channel is noisy, so any information is subject to errors during transmission. On the other hand, we will assume, Alice and Bob have perfect apparatus, which allows them to execute error-correction algorithms with no errors.

This is our basic model for error-correction problems. It seems like it is a specialized to a communication problem, but with a little bit of imagination, we can see how it applies to other types of problems as well. For example, a computer memory can deteriorate over time, causing the stored information to be corrupted. In this scenario, Alice is the person who initially recorded the information to the memory. The noisy channel in this case is across time, rather than space as above, because the memory stays in place and over time gets corrupted. Bob is the person who attempts to read the memory at a later point in time.

Our goal

Our goal, over the next few chapters, is to design procedures for Alice and Bob to communicate reliably, under the above model. We will do so for both the classical and the quantum case.

However, as we will discuss in detail later, this model does not capture all interesting problems that are studied by the field of error-correction. The most important such problem is that of performing quantum computation with unreliable hardware. In this case, there is no part of our system that is error-free and therefore, the problem becomes much more difficult to solve. Fortunately, we will discover that there are algorithms, called fault-tolerant algorithms, that will work for such problems as well. Fault-tolerance algorithms will be the subject of latter chapters of this book.

No quantum without classical

To have a good understanding of quantum error-correction, it is essential to first start by learning the fundamentals of classical error-correction. There are a few reasons for it. The first is that classical algorithms are often conceptually simpler than quantum, and therefore easier to understand. Secondly, many quantum error-correction algorithms are directly or indirectly constructed from classical error-correction algorithms. Therefore, if one is to learn how to do this, one should be aware of the properties of the classical algorithms.

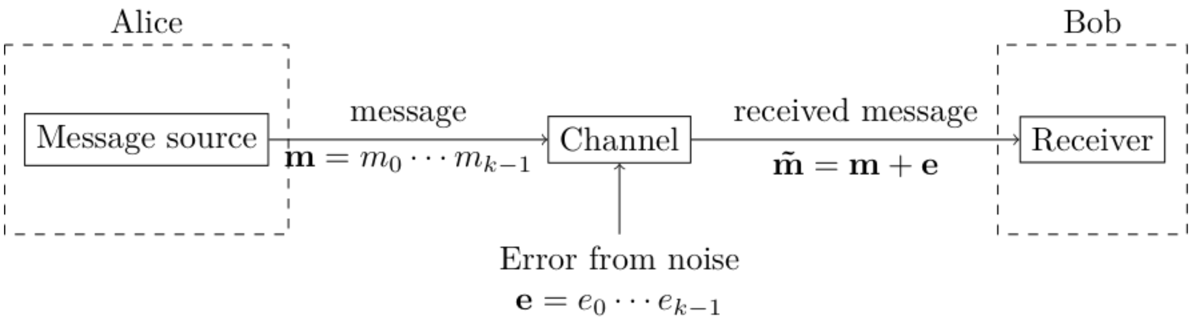
Therefore, in this book, we will first spend some time learning about classical error-correction. As we will refer to this material often later on, the reader should not skip it.

Error-correction by repetition

Classical error-correction is a rich and old field which has produced many procedures for error-correction. To begin our journey through this field, we will start with the simplest possible method of error-correction. This method is called repetition code. The term 'code' refers to a mathematical object, which we will look at in more detail later on. For now, we will concentrate on what Alice and Bob do.

When Alice and Bob don't use any error correction

Let us go back to scenario we created last time, and concretely work out what happens when Alice and Bob don't use any error correction.


Alice
Bob
Message source
message
$\mathbf{m} = m_0 \cdots m_{k-1}$
Channel
received message
$\tilde{\mathbf{m}} = \mathbf{m} + \mathbf{e}$
Receiver
Error from noise
$\mathbf{e} = e_0 \cdots e_{k-1}$

Alice wants to transmit a length-k message $\mathbf{m}$, which is a sequence of bits. When she sends it, it gets distorted by noise $\mathbf{e}$, and what Bob receives is $\tilde{\mathbf{m}} = \mathbf{m} + \mathbf{e}$.

In general $\mathbf{e}$ can be any bit string, but lets define a simple yet realistic model for the kind of transmission noise we encounter in the real world. We assume that if we send a bit m_i through the channel, with probability p the bit will be flipped, and with probability $1-p$ it is not flipped. If we send multiple bits through the channel, each one is independently flipped with probability p. This means that

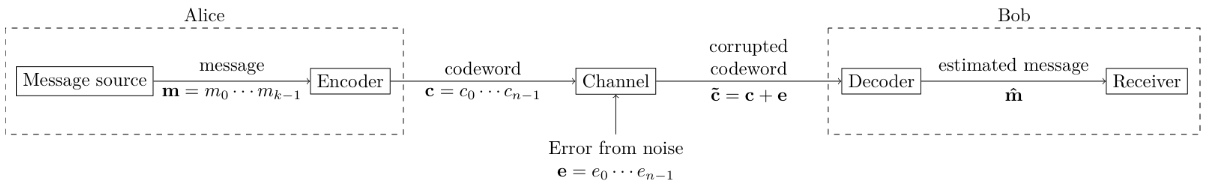
$$e_i = \begin{cases} 1 & \text{with prob } p, \\ 0 & \text{with prob } 1-p, \end{cases} \tag{1}$$

where the value of each e_i is independently set from the others.

Now, suppose that Bob receives the message $\tilde{\mathbf{m}} = 100$ from Alice. What can Bob infer from this? He doesn't know if any of the bits were flipped, so he can't say for sure that this is indeed the message that Alice wanted to send. The most he can infer is that each bit has $1-p$ chance of not getting flipped, so there is a $(1-p)^3$ that he received the right message. On the other hand, there is a $1-(1-p)^3$ chance he received the wrong message.

Can Alice and Bob do better?

To do better, Alice and Bob must do something other than just sending the message. The simplest process is one you are already familiar with, if you have ever talked to someone on the phone with a bad connection. To make sure they hear what you have to say, you repeat yourself a few times. More abstractly, you usually send a longer message from which it is easier for the person on the other end to infer what you said correctly. This abstract process is shown below.


Alice
Message source
message
$\mathbf{m} = m_0 \cdots m_{k-1}$
Encoder
codeword
$\mathbf{c} = c_0 \cdots c_{n-1}$
Channel
corrupted codeword
$\tilde{\mathbf{c}} = \mathbf{c} + \mathbf{e}$
Error from noise
$\mathbf{e} = e_0 \cdots e_{n-1}$
Bob
Decoder
estimated message
$\hat{\mathbf{m}}$
Receiver

Instead of sending $\mathbf{m}$, Alice transforms her message into a longer bit-string called the codeword $\mathbf{c}$. This process is called encoding. She then sends this through the channel. On the other end, Bob receives a corrupted codeword $\tilde{\mathbf{c}}$. He subjects it to a process called decoding. If all goes well, then with higher probability than $(1-p)^k$, he will recover the correct message, i.e. $\hat{\mathbf{m}} = \mathbf{m}$. However, note that there will always be some non-zero chance of making a mistake, $\hat{\mathbf{m}} \neq \mathbf{m}$.

Detecting and correcting by repetition

The process of detecting and correcting errors by repetition is as follows. Alice takes each bit m_i in the message one by one. For each bit, if it has value 0, Alice sends the block of bits 000, and if it has value 1, Alice sends the block 111. Therefore, the codeword $\mathbf{c}$ will have length $n = 3k$.

An alternate notation we will use throughout this book is to write messages with a bar on top, and write the codeword without any bars. For example, here we can simply write

$$\bar{0} = 000, \quad \bar{1} = 111.$$

Using the above process if the message is $\overline{1010}$ then Alice sends 111000111000.

Suppose, that Bob receives the corrupted code-stream 101100111000, in which the second and fourth bits have been flipped by the noisy channel. He, of course, doesn't know which bits have been flipped.

To correct any errors, he will apply a majority voting algorithm to each block of three bits in the corrupted codeword. The first block is 101, and since there are two 1s in it and one 0, he infers that the first sent bit is likely to be $\bar{1}$. Similarly, he infers from the second, third and fourth blocks of three, that the second, third and fourth bits are likely to be 0, 1 and 0, respectively. Hence, he correctly decodes the message to message was $\overline{1010}$.

Notice that this method only works if there is at most one bit flip in each block of three. If there are two or more flips in a block, then Bob inferred bit string will be different from the one Alice intended to send. For example, if the corrupted code-string was 101110111000, then using the same algorithm Bob infers that that message was 1110.

Question: What is the encoding of $\overline{011}$?

To make sure you understand the above process, you will now implement it as computer code.

Task 1

Complete the function `repetition_encode` that when passed a bit string, encodes it according to the repetition code.

- Parameters:
  `message`: a `str` of any length, guaranteed to only contain 0 or 1.
- Returns:
  `codeword`: a `str` that is the the encoded version of the `message`. Must have length three times the length of `message`.

```
def repetition_encode(data_stream):
    pass
```

```
# check your solution
k = 3
for i in range(2**k):
    message = bin(i)[2:].zfill(k)
    codeword = repetition_encode(message)
    print(message, codeword)
```

The process the Bob uses to check and correct for errors is called decoding.

Task 2

Complete the function `repetition_decode` that when passed a corrupted codeword, decodes it according to the repetition code. This include identifying any errors and correcting for them.

- Parameters:

  `codeword`: a `str`, guaranteed to only contain 0 or 1, and length divisible by 3.
- Returns:

  `estimated_message`: a `str` that is the the decoded version of the `codeword`

```
def repetition_decode(codeword):
    pass
```

Here is a script that generates random messages, run them though the encoder, then corrupts the result, and finally decodes them. Use this to check your answers. Remember that if there are multiple error per block then, the estimated message should not be equal the original message.

```
from random import randint, random

# these are the two parameters we need to set
message_length = 5
p = 0.1

# create a random message
message = ''.join([str(randint(0,1)) for i in range(message_length)])
print('message = ', message)

# encode the message
codeword = repetition_encode(message)
print('codeword = ', codeword)

# send it through the noisy channel
corrupted_codeword = ''
for c in codeword:
    if random() < p:
        if c == '0':
            corrupted_codeword += '1'
        else:
            corrupted_codeword += '0'
    else:
        corrupted_codeword += c
print('corrupted_codeword = ', corrupted_codeword)

# decode it
estimated_message = repetition_decode(corrupted_codeword)
print('estimated_message = ', estimated_message)
print('Is message = estimated_message?', message == estimated_message)
```

How good is the repetition based process?

Whenever we come across an error-correction process, we should evaluate how effective it is at correcting errors. The following tasks help you figure this out on your own.

Task 3 (On paper)

Using the repetition code, what is the probability that Bob correctly infers the correct bit-string? Each block of three is treated separately, so we only need to estimate the probability of correct inference from each block. Let p be the probability of error on a single bit. Let the probability of an error on each bit be independent.

What is the probability of

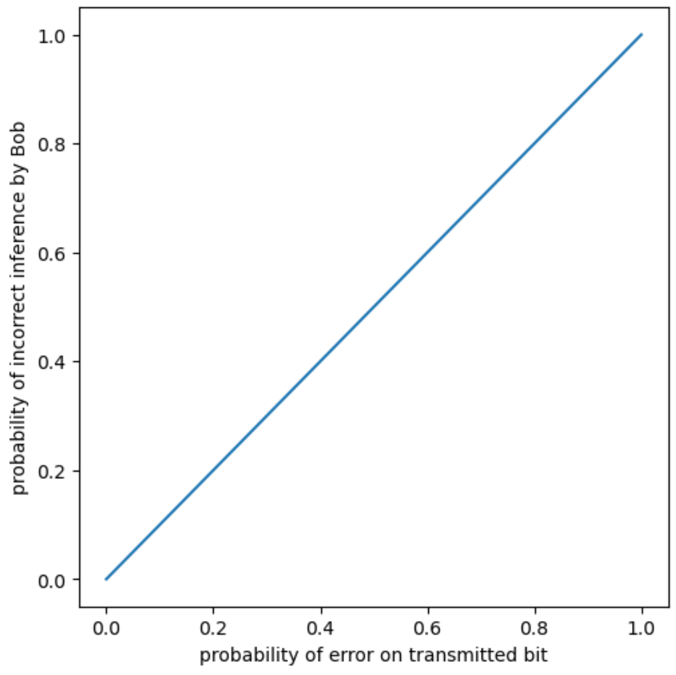
- zero errors,
- one error,
- two errors,
- three errors.

Don't forget the multiplicities.

Task 4

Consider the case of when Alice and Bob do not use any error correction process. Then if Alice sends a single bit, the probability of Bob infering the wrong bit against the physical probability of error p is as follows. Obviously, it's a straight line.

```python
import matplotlib.pyplot as plt
import numpy as np

# probability of error on transmitted bit
p = np.linspace(0, 1, 100)
# probability of incorrect inference by Bob
pr_of_incorrect_inference = p

_, ax = plt.subplots(figsize=(5, 5), layout='constrained');
ax.plot(p, pr_of_incorrect_inference);
ax.set_xlabel('probability of error on transmitted bit');
ax.set_ylabel('probability of incorrect inference by Bob');
```



When Alice and Bob do use error correction, plot the probability of incorrect inference by Bob vs the probability of error on a single bit p. Keep the line drawn above, so you can compare the two cases. For what values of p is it beneficial to use error correction?

```python
# your calculation code here

_, ax = plt.subplots(figsize=(5, 5), layout='constrained');
ax.plot(p, pr_of_incorrect_inference, label='no error correction');

# you can uncomment the line below to add another plot to the same figure
```

```
ax.plot(p, pr_of_incorrect_inference_repetition, label='with repetition code');

ax.set_xlabel('probability of error on transmitted bit');
ax.set_ylabel('probability of incorrect inference by Bob');
plt.legend();
```

What you will discover is that when the physical error rate p is low, then the repetition code helps. However if p is large then using the repetition code is actually worse than the naive process. Can you argue why?

Task 5 (On paper, optional)

This exercise will become crucial when we later discuss fault-tolerant quantum computing.

Suppose that Alice has some encoded message. She wants to do some logical operations on the encoded message without decoding it first. How can she go about it? Concretely, how can she apply

- A NOT operation, that takes $\bar{0}$ to $\bar{1}$, and $\bar{1}$ to $\bar{0}$.
- An OR operation, that has the following truth table

Input	Output
$\overline{00}$	$\bar{0}$
$\overline{01}$	$\bar{1}$
$\overline{10}$	$\bar{1}$
$\overline{11}$	$\bar{1}$

The binary field

On computer systems, information is typically represented, stored and processed in the form of 0s and 1s. Which is why, when we were studying the repetition code, we assumed that the information Alice meant to send was in the form of a string of bits.

At one point, we said that the corrupted codeword $\tilde{\mathbf{c}} = \mathbf{m} + \mathbf{e}$. What does it mean to add two strings of bits? In the following, we will discuss this in more detail, as a precursor to our study of both classical and quantum error-correcting codes. We will also figure out how to do this sort of mathematics in python.

What is the binary field?

In the binary field, there are only two numbers $\{0, 1\}$. To do anything useful with them, we need to define their addition and multiplication. The rules for addition are summarized in the following table.

+	0	1
0	0	1
1	1	0

Out of the four rules, the one in the bottom right corner is the most interesting one, $1 + 1 = 0$.

Similarly, the rules of multiplication are as follows. They are pretty straightforward and not different from the rules we use to multiply real numbers or integers.

$\times$	0	1
0	0	0

×	0	1
1	0	1

What is important about the two sets of rules is that they always ensure that the addition or multiplication of two numbers never yields anything besides $\{0, 1\}$.

> The **binary field** is labelled $\mathbb{F}_2$.

Linear algebra over $\mathbb{F}_2$

Classical error-correcting codes are defined using vector fields over the binary field. This means that any vectors and matrices we will encounter will have their elements coming from the binary field. When we do any matrix algebra, we will have to apply the addition and multiplication rules of binary field.

For example, let us take the two-dimensional vector space over the binary field. Define

$$\mathbf{v} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Then

$$\mathbf{v} + \mathbf{w} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Now, also define

$$M = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

Then,

$$M\mathbf{w} = \begin{pmatrix} 1 \times 1 + 0 \times 1 \\ 1 \times 1 + 1 \times 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Finally, we will use the following notation.

> A m-dimensional vector space over the binary field is labelled as $\mathbb{F}_2^m$.

Question: Let $\mathbf{m} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$ and $\mathbf{e} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$. What is the corrupted codeword $\tilde{\mathbf{c}}$?

Working with $\mathbb{F}_2^m$ in python

The following is some example code that will help you understand how to work with binary vectors and matrices in python. There are only two things you have to do.

- Specify `dtype=int` for any numpy arrays you use.
- For every operation, use modulus 2. This is done via `% 2` in python.

```python
import numpy as np

print('(1 + 1) % 2 = ', (1 + 1) % 2)
print('')

# declare your vectors and matrices as numpy.array
# with a data-type of int, rather than the default
# type of float
M = np.array([[1, 0], [1,1]], dtype=int)
v = np.array([1, 1], dtype=int)
print('M = \n', M)
print('')
```

```python
# if you add things as
print('M + M =\n', M+M, 'incorrect! 2 is not part of our number system.')
# you will get the wrong answer. '2' is not part of our number system.

print('')
# Instead mod everything 2. This is done with '% 2' in python
print('(M + M) % 2 =\n', (M+M) % 2, 'correct')
# The parenthesis in (M+M) % 2 are important

print('')
print('v = ', v)
# Remember that in numpy matrix multiplication is done using the @ symbol
print('M@v = ', M@v, 'incorrect')
print('M@v % 2 = ', M@v % 2, 'correct')
```

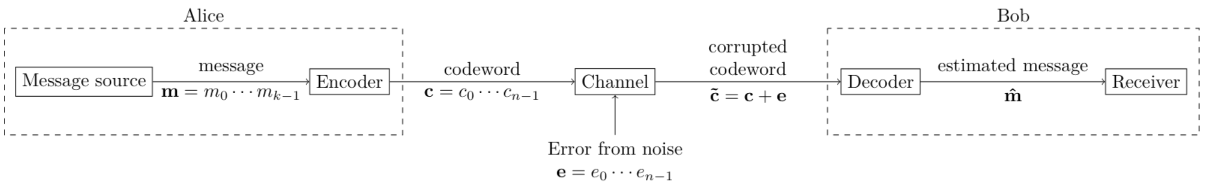
```
(1 + 1) % 2 =  0

M =
 [[1 0]
 [1 1]]

M + M =
 [[2 0]
 [2 2]] incorrect! 2 is not part of our number system.

(M + M) % 2 =
 [[0 0]
 [0 0]] correct

v =  [1 1]
M@v =  [1 2] incorrect
M@v % 2 =  [1 0] correct
```

Classical linear codes

There are many different ways of of protecting classical information from noise. The simplest and nicest such processes are built from linear codes. Quantum error-correcting codes are also often built from classical linear codes, which is a strong motivation to study such classical codes.

Treating information as vectors

The classical error correcting model we discussed earlier was as follows.

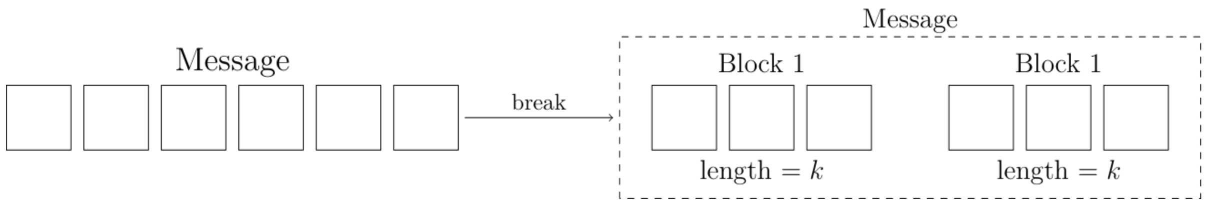


Alice
Message source
message
$\mathbf{m} = m_0 \cdots m_{k-1}$
Encoder
codeword
$\mathbf{c} = c_0 \cdots c_{n-1}$
Channel
Error from noise
$\mathbf{e} = e_0 \cdots e_{n-1}$
corrupted codeword
$\tilde{\mathbf{c}} = \mathbf{c} + \mathbf{e}$
Bob
Decoder
estimated message
$\hat{\mathbf{m}}$
Receiver

Information here is in the form of bit strings. In line with our discussion in the binary field, we will now treat these as elements of vector fields over the binary field. Let us now define each of these vector fields.

The message space

The first vector field is the one that contains the message. For the type of codes we will study, the length of the message is a fixed number, which we denote by k. If the information to be sent is longer, than we break it up into blocks of length k, and then treat each block as its own independent message. In the repetition code example, the blocks were of size 1.



Keeping this in mind, we have the following definitions.

> The information to be protected is a bit string of length k, and each such string is called a **message**.

> The **message space** is the vector space with all 2^k possible messages. This vector space is denoted as $\mathbb{F}_2^k$.

For the repetition code, $k = 1$. This means the elements of the repetition code are $\{0, 1\}$.

The codespace and the code

As in the repetition code, for linear codes each message is transformed into a longer bit-string before being transmitted. This longer bit-string is called a codeword, and the transformation from message to codeword is called encoding the message.

> A message is transformed or **encoded** to a **codeword**, which is a bit string of length n.

Codewords are of length n. There are 2^n strings of length n, but there are only 2^k valid codewords - one for each possible message. Therefore, there are two vector spaces here.

> The **codespace** is a n-dimensional vector space, labelled $\mathbb{F}_2^n$. We require that $n \geq k$.

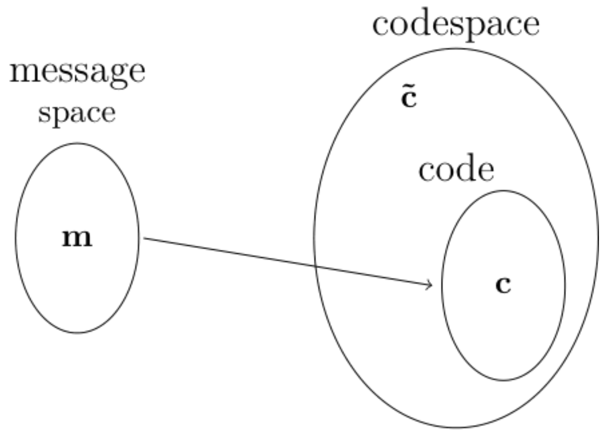
There is no typo here. Codespace is written without a space usually. For the repetition code, the codespace consists of all length-$n = 3$ bit strings,

$$\{000, 001, 010, 011, 100, 101, 110, 111\}.$$

The second vector space is the the subspace of the codespace that actually contains the valid codewords. For the repetition code, this is the two codewords $\{000, 111\}$.

> The **code** $\mathcal{C}$ is a k-dimensional subspace of $\mathbb{F}_2^n$. Compactly, we can write $\mathcal{C} \subset \mathbb{F}_2^n$.

So code is just the name for the vector space in which valid codewords live. The relationship between the three spaces is shown below.



In the repetition code, the message $\mathbf{m} = 0$ lives inside the message space. It is mapped to the codeword $\mathbf{c} = 000$, which lives inside the code $\mathcal{C}$. If $\mathbf{c}$ gets distorted by noise in the channel, then it gets transformed into the corrupted codeword $\tilde{\mathbf{c}}$ which lives inside the codespace by outside $\mathcal{C}$

Why error-correcting codes are called code?

To specify an error-correcting code, we need to define the following three things.

- The dimension of the message space, k.
- The dimension of the codespace, n.
- The code $\mathcal{C}$.

Here the code $\mathcal{C}$ is the non-trivial part, so error-correcting codes are just called code. This abuse to terminology is not too bad, as it is usually clear from context whether we are talking about $\mathcal{C}$ or the entire error-correcting code.

Error-correcting codes are usually labeled as $[n, k, d]$, where distance d is something which will be defined later. This is an underspecification as there can be more than one code with the same n, k, d but different $\mathcal{C}$. However, these parameters are usually sufficient within context to identify the code.

Question: What is the label $[n, k, d]$ for the repetition code? It has distance 3.

Encoding process

The definition of a code given above is quite abstract. To encode messages and decode corrupted codewords we need a more concrete definition that gives well-defined calculational procedures. This is done by given an explicit definition of $\mathcal{C}$.

For any code, we can identify a basis set of $\mathcal{C}$. For instance, for the repetition code, the basis set is

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

where we are now writing vectors as row matrices instead of bit string. In general, there will be k such vectors because the subspace is k-dimensional, and each vector will be of length n. We can collect these k vectors into the rows of a matrix, called the generator matrix.

> The **generator matrix**, G, is a $k \times n$ matrix whose rows are a basis of the code $\mathbf{C}$.

For the repetition code, a generator matrix is

$$G = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}. \tag{2}$$

Any message $\mathbf{m}$ can be encoded into a codeword $\mathbf{c}$ using the generator matrix. The codewords of the code are all in the row space of its generator matrix G, i.e. the set of codewords is

$$\{c = \mathbf{m}G : \mathbf{m} \in F_2^k\}. \tag{3}$$

For the repetition code, we find the codewords to be

$$\begin{pmatrix} 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}, \tag{4}$$
$$\begin{pmatrix} 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}, \tag{5}$$

which is what we obtained before. In this way, we have an explicit encoding process for any code.

> A **code** is specified entirely using a generator matrix.

Because the basis for the code C is not unique, there can be many possible generator matrices for the same code C, and as we will see, some of matrices will have more calculational utility than others. We can convert between different generator matrices of the code by doing row operations on the generator matrix, which will not change the row space. Among the forms of the generator matrices, there is a standard form, which is when $G = [I_k | A]$, where I_k is the $k \times k$ identity matrix and A is a $k \times (n-k)$ matrix. The standard form can be obtained via row operation from any other form.

Hamming codes

The repetition code is a very simple code. To understand more concepts about error-correction, we need to a more complex code. The Hamming code, invented by Richard Hamming, serves this purpose. It is a $[7, 4, 3]$ code, meaning messages of length 4 are encoded into codewords of length 7.

To define the Hamming code, we need to define the $\mathbf{C}$. First we define how to construct $\mathbf{c}$ from a $\mathbf{m}$. The first four bits of $\mathbf{c}$ are actually the same as $\mathbf{m}$, i.e.

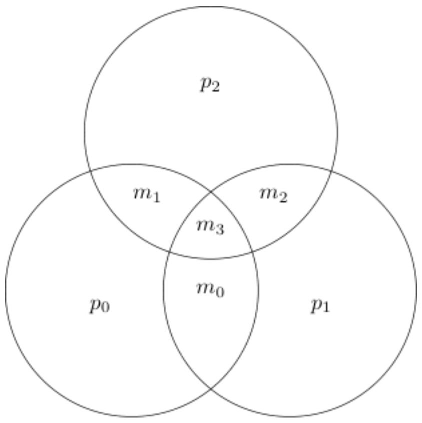
$$c_0c_1c_2c_3 = m_0m_1m_2m_3.$$

The remaining bits, $c_4c_5c_6$ are defined by using the notion of parity.

> The **parity** of a bit string is computed from the number of 1s in it. If there are even number of 1s, the parity of the string is said to be even or 0, otherwise it is odd or 1.

For example, the parity of 101 is 0 and that of 01011 is 1.

For the Hamming code, we compute the parity of various subsets of the message bits, and store them in the codeword. If the codeword is corrupted, some of the parities change, which allows us to identify the error. There are four message bits. There are $4C3 = 4$ subsets of size three. We compute the parity of three of these subsets. This is easily seen from the following diagram.



We see that

- p_0 is the parity of m_0, m_1, m_3,
- p_1 is the parity of m_0, m_2, m_3,
- p_2 is the parity of m_1, m_2, m_3.

The parity of the fourth subset (m_0, m_1, m_2) is dependent on the parity of the other three, so does not need to be computed. The full codeword is

$$\mathbf{c} = (m_0 \quad m_1 \quad m_2 \quad m_3 \quad p_0 \quad p_1 \quad p_2).$$

Before we dig into the mathematics, let us discuss how the protocol works. Alice creates the codeword according to the prescription above and sends it to Bob. When Bob receives the corrupted codeword he computes the three parities of the first four bits ($\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, \tilde{\mathbf{c}}_3$)) and compares them with the last three bits of the corrupted codeword ($\tilde{\mathbf{c}}_4, \tilde{\mathbf{c}}_5, \tilde{\mathbf{c}}_6$). If some of the parities are different, then that indicates an error has occurred.

Suppose an error happens on $\mathbf{c}_0$. We can see from the diagram that this means that p_0 and p_1 will be flipped. On the other hand, an error on $\mathbf{c}_1$ will lead to p_0 and p_2 to be flipped. Similarly, if the error happens on a parity bit, then only that parity bit will be flipped. So a single parity bit differing indicates an error on it, while two or three parity bit differences indicate an error on the "message bits".

Task 1 (On paper)

Fill out the following table.

Error location	Parity bits flipped
$\mathbf{c}_0$	
$\mathbf{c}_1$	

Error location	Parity bits flipped
$\mathbf{c}_2$	
$\mathbf{c}_3$	
$\mathbf{c}_4$	
$\mathbf{c}_5$	
$\mathbf{c}_6$	

If there are no repeats in the second column, then Bob can always correct zero or one errors. What happens if two bits experience an error? Which parity bits flip if m_0 and m_1 experience an error? Compare with the table above.

Your answer should tell you that the Hamming code can only be used to protect against one error.

Generator matrix

We now construct the generator matrix G for the Hamming code, using the description above. The first four bits have to be copied as is, so the first four columns of G must be the identity matrix. The next three columns must be the various ways of summing up the data bits. Hence, we get

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}. \tag{6}$$

This is clearly in the standard form.

Task 2

Complete the function `hamming_encode` that when passed a bit string, encodes it according to the Hamming code.

- Parameters:
  - `message` a `numpy.array`, of shape (1,4) and `dtype=int` guaranteed to only contain 0 or 1.
- Returns:
  - A `numpy.array` of shape (1,7) and `dtype=int` that is the the encoded version of the `message`

```python
# a function to create random messages
from random import randint
import numpy as np
def random_message(k):
    return np.array([randint(0,1) for _ in range(k)], dtype=int)

random_message(4)
```

```python
G = np.array([
    [ 1 , 0 , 0 , 0 , 1 , 1 , 0 ],
    [ 0 , 1 , 0 , 0 , 1 , 0 , 1 ],
    [ 0 , 0 , 1 , 0 , 0 , 1 , 1 ],
    [ 0 , 0 , 0 , 1 , 1 , 1 , 1 ]], dtype=int)

def hamming_encode(message):
    pass


m = random_message(4)
c = hamming_encode(m)
```

```
print('m = ', m)
print('c = ', c)
```

Identifying and correcting errors systematically

The previous section provided us with a way of encoding any message m into a codeword c which generalizes to any code. Now, we discuss a systematic method to identify and possibly correct errors that occur when the codeword is sent through the noisy channel.

Suppose that Bob receives the block $\tilde{\mathbf{c}}$. This block could or could not be corrupted in some way. Every code has a certain tolerance for how many errors it can correctly identify and subsequently correct. The way to see it is to recall that $\tilde{\mathbf{c}} = \mathbf{c} + \mathbf{e}$. The error vector $\mathbf{e}$ is n bits long, so there are 2^n possible errors. The codeword $\mathbf{c}$ has some information stored inside it to help identify the error. But it also has to store the message itself. So out of the n bits of information in $\mathbf{c}$, k bits of information is storing the message, while $n-k$ bits of information is available to identify errors. So, the code can at most identify 2^{n-k} different errors. For example, the repetition code can identify $2^{3-1} =$ four different errors - either no error or an error on exactly one of the three bits. Similarly, the Hamming code can identify $2^{7-4} =$ eight different errors.

What is the generic process for this? We saw in the Hamming code example that we stored some consistency checks on the message as part of our codeword. We said that the first four bits must obey some parities. Generally, in any linear code, there will be some consistency checks and will be given by a linear set of equations on the elements on the codeword. These equations are specified by a matrix multiplying the codewords, and given as follows.

> The matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ is a **parity-check matrix** for code C if $C = \{\mathbf{x} \in \mathbb{F}_2^n : H\mathbf{x}^T = 0\}$.

This definition is saying that a code consists of all the elements of the codespace that satisfy the constraint $H\mathbf{x}^T = 0$. As there are $n-k$ rows to this matrix, the matrix places $n-k$ constraints on the elements of the codespace, leaving behind a $n-(n-k) = k$-dimensional space. Note that a system of linear equations can be linear transformed into another set of equivalent equations. Therefore, the parity-charity matrix is not unique. Row operations will yield other parity-check matrices for the same code.

The constraints in the parity-check matrix are obviously not independent from the generator matrix, as the generator matrix contains all the information about the code. The relationship between the generator matrix and the parity-check matrix is given as follows.

Lemma: A generator matrix G and a parity-check matrix H of a code C are constrained by $HG^T = 0$.

Proof: Recall that any codeword $c = mG$, so $c^T = G^T m$. Now, by the definition of H, we have $Hc^T = 0$, or $HG^T m = 0$. But this last statement is true for all m, which is only possible if $HG^T = 0$.

The above theorem only tells us if a parity-check matrix is valid for a code. It does not give us an explicit procedure for constructing it. The following lemma gives a process for exactly this, using the generator matrix in standard form.

Lemma: If G is in standard form, i.e. $G = [I_k|A]$, then the parity-check matrix is $H = [A^T|I_{n-k}]$, where I_k is $k\times k$, A is $k\times(n-k)$, A^T is $(n-k)\times k$ and I_{n-k} is $(n-k)\times(n-k)$.

Proof: We need to show that $HG^T = 0$. We can compute,

$$HG^T = \begin{pmatrix} A^T & I_{n-k} \end{pmatrix} \begin{pmatrix} I_k \\ A^T \end{pmatrix} = A^T I_k + I_{n-k} A^T = A^T + A^T = 0. \tag{7}$$

Hence, proved.

For the Hamming code, the parity check matrix is

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}. \tag{8}$$

Question: Verify that $HG^T = 0$ for the Hamming code.

```
H = np.array([
    [1, 1, 0, 1, 1, 0, 0],
    [1, 0, 1, 1, 0, 1, 0],
    [0, 1, 1, 1, 0, 0, 1]], dtype=int)
```

Error syndromes

We have learned that the parity-check matrix can be used to determine if $\tilde{\mathbf{c}}$ is a codeword or not. But how can we compute which error has occurred. Imagine Alice sends the codeword $\mathbf{c}$ and it gets distorted to $\tilde{\mathbf{c}} = \mathbf{c} + \mathbf{e}$. Now, the effect of the parity-check matrix is,

$$\mathbf{s} = H\tilde{\mathbf{c}}^T = H\mathbf{c}^T + H\mathbf{e}^T = 0 + H\mathbf{e}^T = H\mathbf{e}^T. \tag{9}$$

The quantity $\mathbf{s}$ is called an error syndrome, and is a vector of length $n-k$. There are 2^{n-k} possible error syndromes. We argued above that a classical linear code can only correct up to 2^{n-k} errors. Each correctable error corresponds to a distinct error syndrome.

For example, for the Hamming code, $\mathbf{s}$ will be of length 3 and there will 8 different syndromes corresponding to the different zero bit or one bit errors. The next few exercises will help you compute them.

Task 3

Complete the function `noise_channel` that with probability p, flips each element of the input.

- Parameters:
  - c: a `numpy.array` of shape (1,n) and `dtype=int` guaranteed to only contain 0 or 1. The transmitted codeword.
  - p: `float` guaranteed to be between 0 and 1 inclusive. The probability of error.
- Returns:
  - `numpy.arrary` of shape (1,n) and `dtype=int` guaranteed to only contain 0 or 1. The corrupted codeword.

```
def noise_channel(c, p):
    pass

c = random_message(7)
corrupted_c = noise_channel(m, 0.5)
print('corrupted c = ', corrupted_c)
```

Task 4

Complete the function `hamming_syndrome` that when passed a corrupted codeword, determines the error syndrome.

- Parameters:
  - `corrupted_c`: a `numpy.array`, of shape (1,7) and `dtype=int` guaranteed to only contain 0 or 1.
- Returns:
  - Error syndrome: a `numpy.array` of shape (1,3) and `dtype=int` that is the error syndrome

```
def hamming_syndrome(corrupted_c):
    pass
```

Correcting the error

Once one knows the error $\mathbf{e}$ that has occured, to correct the codeword, one can simply note that $\tilde{\mathbf{c}} + \mathbf{e} = \mathbf{c} + \mathbf{e} + \mathbf{e} = \mathbf{c}$. So the codeword is straightforwardly obtained.

Task 5

Compute a `dict` of all possible syndromes and their corresponding one bit errors.

```
# write your code here

# key is syndrome
# value is the error
error_syndromes = dict()

# np.arrays can't be keys for dict
# So we need to convert them to tuples

# zero error case
s = np.array([0,0,0], dtype=int)
e = np.array([0,0,0,0,0,0,0], dtype=int)

error_syndromes[tuple(s)] = e

print(error_syndromes[tuple(s)])
```

Task 6

Complete the function `hamming_correct` that when passed a corrupted codeword, identifies the error syndrome, infers an error, and corrects for it.

- Parameters:

  `corrupted_c`: a `numpy.array`, of shape (1,7) and `dtype=int` guaranteed to only contain 0 or 1.
- Returns:

  The corrected codeword, a `numpy.arrary` of shape (1,7) and `dtype=int` guaranteed to only contain 0 or 1.

```
def hamming_correct(corrupted_c):
    pass
```

Distance of a code

A crucial quantity of interest when discussing codes, is their distance. The distance determines the number of errors that can be corrected. Whenever an error occurs on a codeword $\mathbf{c}$, it creates a corrupted codeword $\tilde{\mathbf{c}}$ that is at some 'distance' away from the $\mathbf{c}$. If the distance is small, then it is still possible to correct the error, but if the distance is larger than some value, than the error cannot be corrected. The threshold value is called the distance of the code. We formalize it below.

Hamming distance

The Hamming distance (not to be confused with the Hamming code; Hamming was just a very prolific scientist) is going to be used to build the notion of code distance.

> The **Hamming distance** between two bit strings $\mathbf{s}, \mathbf{t}$ is the number of places where they have different symbols. This will be denoted $d_H(s, t)$.

The Hamming distance determines the number of substitutions or errors required to transform a length-n $\mathbf{s}$ into another length-n vector $\mathbf{t}$ or vice versa. Hence, it is a very natural definition for our purposes. The Hamming distance can be calculated as

$$d_H(\mathbf{s}, \mathbf{t}) = \sum_i \mathbf{s}_i + \mathbf{t}_i, \tag{10}$$

where the addition is in the binary field, but the summation is regular integer addition.

For example, the distance between 1011 and 0111 is two. You have to flip the first two entries to go from the first to the second or vice versa.

Distance of a code

Using the above definition, we define the notion of the distance of a code.

> The distance d of a code is the minimum Hamming distance between any two codewords. Formally,
>
> $$d = \min_{\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C}} d_H(\mathbf{c}_1, \mathbf{c}_2). \tag{11}$$

This seems to be a tedious optimization problem. However, we can simplify the calculation in the following way. First, we define the weight of a bit string.

> The weight of a string $\mathbf{s}$, denoted $wt(\mathbf{s})$, is the number of ones in the string.

The weight can also be thought of as the Hamming distance of $\mathbf{s}$ from the all 0 string, i.e.

$$wt(\mathbf{s}) = d_H(\mathbf{s}, 0^n). \tag{12}$$

Next note that the zero message, $\mathbf{m} = 0^k$, maps to the zero codeword $\mathbf{c} = 0^k G = 0^n$.

Then, using the properties of the Hamming distance and the fact that the codespace is a vector space, and letting $\mathbf{c}_1 = \mathbf{x}_1 G \neq \mathbf{c}_2 = \mathbf{x}_2 G$, we have

$$\begin{aligned} d_H(\mathbf{c}_1, \mathbf{c}_2) &= d_H(\mathbf{x}_1 G, \mathbf{x}_2 G), && (13)\\ &= d_H(\mathbf{x}_1 G + \mathbf{x}_2 G, 0^k), && (14)\\ &= d_H((\mathbf{x}_1 + \mathbf{x}_2) G, 0^k), && (15)\\ &= d_H(\mathbf{c}_3, 0^k), && (16)\\ &= wt(\mathbf{c}_3), && (17) \end{aligned}$$

where in the second last step we have defined $\mathbf{c}_3 = (\mathbf{x}_1 + \mathbf{x}_2)G$, which must exist in C because $\mathbf{x}_1 + \mathbf{x}_2$ is part of the message space, and $\mathbf{c}_3 \neq 0$. So we have discovered that the distance of a code,

$$d = \min_{\mathbf{c}\in\mathcal{C}} wt(\mathbf{c}), \tag{18}$$

is just the minimum weight string.

Another theorem simplifies the calculation even more.

Theorem: The distance d of a code $\mathcal{C}$ is equal to the minimum number of linearly dependent columns of the parity check matrix H of the code.

Relationship of distance and correctable errors

The use of the distance of a code is given by the following theorem.

Theorem: A code with distance d can

- Correct at most $d-1$ erasure errors,
- Detect at most $d-1$ bit flip errors,
- Correct at most $\left\lfloor \frac{d-1}{2} \right\rfloor$ bit flip errors.

Therefore, a single quantity tells us the power of a code at detecting and correcting various types of errors.

Question: Compute the distance of the $[7,4]$ Hamming code, using both methods outlined above. Show $d=3$.

> If the distance d of a code $\mathcal{C}$ is known, then $\mathcal{C}$ is denoted as $[n,k,d]$.

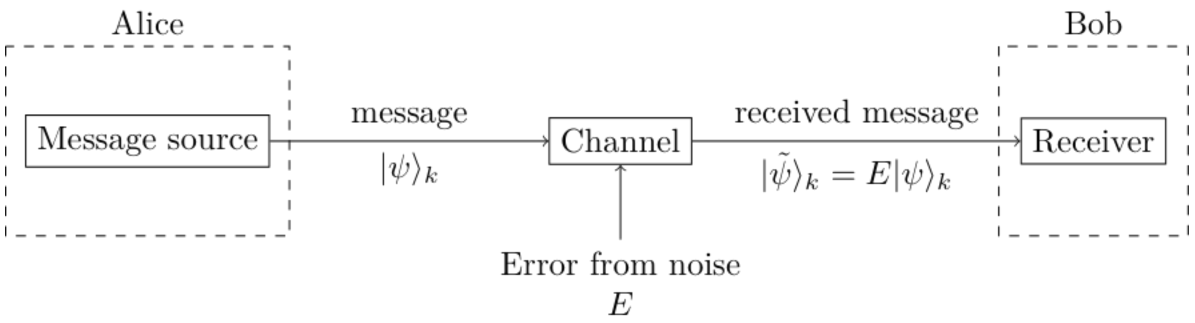
Question: Determine the distance d of the $[n,1,d]$ repetition code.

Quantum repetition code for bit-flips

We are now in a position to introduce quantum codes. Our goal is to understand how a set of qubits transmitted through a noisy channel can be protected from any errors they experience. The process is similar to classical codes in that we will take the state of k qubits and encode it into the state of n qubits. However, our task will be complicated by the fact that the operations on qubits are not just bit-flips but any possible interaction with the environment, such as phase-flips or more general rotations. Fortunately, we will discover that this possibility of infinite types of errors will not be too large a hindrance and we can construct good codes.

Setup

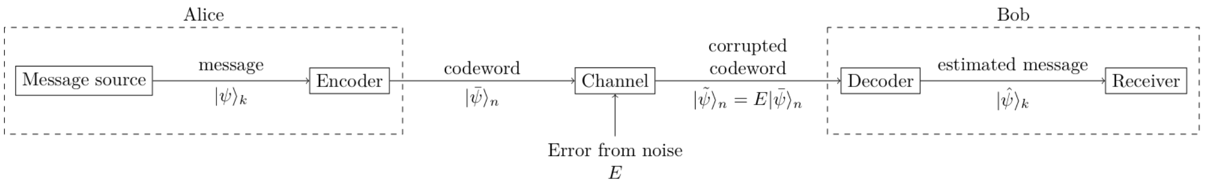
Alice has a quantum transmission channel to Bob that is noisy. In general she intends to transmit a k-qubit state to Bob. During transmission, the quantum state can experience an error E.



Let's assume for starters that the noisy channel applies, with probability p, the quantum bit-flip operator X to each qubits that pass through it. We will discuss channels with different errors later on.

Suppose for a moment that $k=1$ for simplicity. Alice wants to transmit to Bob the one-qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. If she sends the state as is, there is a probability $(1-p)$ that Bob receives the state $|\psi\rangle$ and probability p that Bob receives the state $\left|\tilde{\psi}\right\rangle = X\left|\psi\right\rangle = \beta\left|0\right\rangle + \alpha\left|1\right\rangle$.

Bob does not know whether an error occurred or not. So he is quite unclear on what state Alice meant to send. To magnify the chances of successful communication, Alice and Bob should employ a quantum error-correcting code. At the abstract level, the process will look like the following.



Let us study the different parts of this process, using the quantum repetition code as our working example.

Encoding

In the quantum version of the repetition code, Alice takes the unencoded qubit and encodes it into the state of three qubits in a repetitive manner. This is best seen by first noting the transformation on the basis states,

$$\left|0\right\rangle \rightarrow \left|\bar{0}\right\rangle = \left|000\right\rangle, \tag{19}$$

$$\left|1\right\rangle \rightarrow \left|\bar{1}\right\rangle = \left|111\right\rangle. \tag{20}$$

From this, we can determine the qubit in state $\left|\psi\right\rangle$ is encoded as

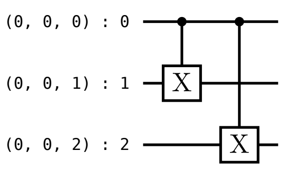
$$\left|\psi\right\rangle\left|00\right\rangle = (\alpha\left|0\right\rangle + \beta\left|1\right\rangle)\left|00\right\rangle \rightarrow \left|\bar{\psi}\right\rangle = \alpha\left|000\right\rangle + \beta\left|111\right\rangle, \tag{21}$$

where the $\left|00\right\rangle$ are ancilla qubits.

We will call the original qubit the unencoded logical qubit, the three qubits the data qubits, and their combined state $\left|\bar{\psi}\right\rangle$ the encoded logical qubit.

The encoding transformation can be done using the following quantum circuit, which takes the unencoded logical qubit and two ancillas in state $\left|\psi\right\rangle\left|00\right\rangle$, to the encoded logical qubit in state $\left|\bar{\psi}\right\rangle$.

```python
import stac
enc_circ = stac.Circuit.simple(3)
enc_circ.append('CX', 0, 1)
enc_circ.append('CX', 0, 2)
enc_circ.draw()
```



Task 1 (On paper)

Determine the encoding of the $\left|+\right\rangle$ state and $\left|-\right\rangle$ state.

Task 2 (On paper)

Determine logical gate operations $\bar{X}$ and $\bar{Z}$ for the three-qubit repetition code. These are operations that act on the logical basis $\{\left|\bar{0}\right\rangle, \left|\bar{1}\right\rangle\}$, in the normal way, i.e.

$$\bar{X}\left|\bar{0}\right\rangle = \left|\bar{1}\right\rangle, \quad \bar{X}\left|\bar{1}\right\rangle = \left|\bar{0}\right\rangle, \tag{22}$$

$$\bar{Z}\left|\bar{0}\right\rangle = \left|\bar{0}\right\rangle, \quad \bar{Z}\left|\bar{1}\right\rangle = -\left|\bar{1}\right\rangle. \tag{23}$$

They can be constructed by some combination of operations on the three data qubits. You will discover that there are possibly multiple ways of doing so.

Errors on the state

Alice sends the three data qubits through the noisy channel. Each of them will have probability p of being flipped. If the second qubit is flipped, then Bob receives the state $X_2\left|\bar{\psi}\right\rangle = \left|\tilde{\psi}\right\rangle = \alpha\left|010\right\rangle + \beta\left|101\right\rangle$. In total there, are eight possiblities for the error, with the probabilities similar to the classical case.

Error	Probability
I	$(1-p)^3$
X_0	$p(1-p)^2$
X_1	$p(1-p)^2$
X_2	$p(1-p)^2$
X_0X_1	?
X_0X_2	?
X_1X_2	?
$X_0X_1X_2$	p^3

Question: Fill the ? in the table above.

Task 3 (On paper)

Determine the impact of each error on the input basis states.

Let input state be $\left|000\right\rangle$

Error	Output state
I	$\left|000\right\rangle$
X_0	$\left|100\right\rangle$
X_1	?
X_2	?
X_0X_1	?
X_0X_2	?
X_1X_2	?
$X_0X_1X_2$	?

Let input state be $|111\rangle$

Error	Output state
I	$|111\rangle$
X_0	$|011\rangle$
X_1	$?$
X_2	$?$
X_0X_1	$?$
X_0X_2	$?$
X_1X_2	$?$
$X_0X_1X_2$	$?$

Do different errors result in the same output state?

As in the classical case, the code can only correctly recover the state if only zero or one error occurs and fails otherwise. Similarly, as in the classical case, if p is sufficiently small, then two or three errors are very unlikely to occur.

Decoding

There are three parts to decoding

1. Identifying which error has occurred.
2. Correcting the corrupted codeword.
3. Turning the corrected codeword back into message (this part is also called decoding).

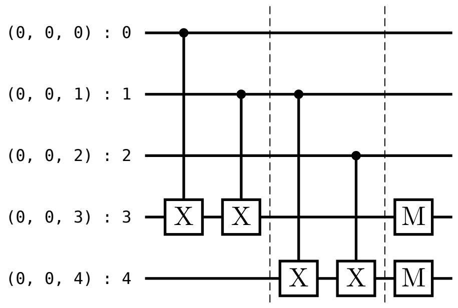
Detecting the error

Assuming that only errors from the set $\mathcal{E} = \{I, X_0, X_1, X_2\}$ occur, Bob has to (a) identify which of these errors has occurred and (b) then correct for it. His task is more difficult than in the classical case because he can't measure the received state as this would destroy the superposition, and he would gain very little information about α and β which is what Alice is really trying to transmit to Bob.

Instead the correct strategy for Bob to pursue is to do a partial measurement that only determines whether one of the qubits is different from the other two. This is the replacement of the majority voting algorithm in the classical case, but quite similar. This type of measurement does not probe the value of α or β and therefore the superposition is maintained. Specifically Bob compares the value of the first two qubits, and compares the value of the last two qubits. If there is a difference, then he will know which error occurred. The following circuit accomplishes this task, in which two ancilla qubits are required.

> The ancilla qubits used for detecting errors are called syndrome qubits.

```
synd_circ = stac.Circuit.simple(5)
synd_circ.append('CX', 0, 3)
synd_circ.append('CX', 1, 3)
synd_circ.append('TICK')
synd_circ.append('CX', 1, 4)
synd_circ.append('CX', 2, 4)
synd_circ.append('TICK')
synd_circ.append('M', 3)
synd_circ.append('M', 4)
synd_circ.draw()
```



This circuit has three parts. In the first part, the values of the first and second data qubits are added to the first syndrome qubit. This is done by CX gates, which if you recall, in the computational basis, add the value of the control qubit to the target qubit,

$$CX_{01} |a\rangle |b\rangle = |a\rangle |a \oplus b\rangle. \tag{24}$$

If the first two data qubits have the same value, then either none of the CX gates will trigger or both will, and so the value of the syndrome qubit will not change. This can be seen from the calculation,

$$\begin{aligned} CX_{03}CX_{13}(\alpha |000\rangle + \beta |111\rangle) |00\rangle &= \alpha CX_{03}CX_{13} |000\rangle |00\rangle + \beta CX_{03}CX_{13} |111\rangle |00\rangle, & (25)\\ &= \alpha |000\rangle |00\rangle + \beta |111\rangle |0+1+1,0\rangle, & (26)\\ &= \alpha |000\rangle |00\rangle + \beta |111\rangle |00\rangle. & (27) \end{aligned}$$

Here, the CX gates don't trigger for the first term in the superposition, but do trigger for the second term, which we have made explicit.

If the first two data qubits are different, then only one of the CX gates will trigger. For instance,

$$\begin{aligned} CX_{14}CX_{24}(\alpha |010\rangle + \beta |101\rangle) |00\rangle &= \alpha CX_{14}CX_{24} |010\rangle |00\rangle + \beta CX_{14}CX_{24} |101\rangle |00\rangle, & (28)\\ &= \alpha |010\rangle |0+1,0\rangle + \beta |101\rangle |0+1,0\rangle, & (29)\\ &= \alpha |010\rangle |10\rangle + \beta |101\rangle |10\rangle. & (30) \end{aligned}$$

Hence, the value of the syndrome qubit flipped.

The same story plays out for the second part of the circuit except for the second and third data qubits. Subsequently, the two syndrome qubits are measured. The measurement results are called the syndrome, and they contain information, which can be used to infer which error has occurred. This is summarized in the following table.

Syndrome	Inferred error
00	I
01	X_2
10	X_0
11	X_1

We say inferred error, because this is the error Bob assumes, but there is small chance of a two or three qubit error, so there is difference between the actual error and the inferred error.

Question: Above, we have shown how row 1 and row 3 of the table are obtained. Work out the other two rows by hand.

Test the procedure

Use the following circuit to test this process of inferring the correct error

```
circ = stac.Circuit.simple(5)

# set message to |1> by uncommenting
# circ.append('X', 0)
# encode the state
circ.append('CX', 0, 1)
circ.append('CX', 0, 2)

# apply error to 0,1 or 2
circ.append('X', 0)

# do syndrome measurements
circ.append('CX', 0, 3)
circ.append('CX', 1, 3)
circ.append('TICK')
circ.append('CX', 1, 4)
circ.append('CX', 2, 4)
circ.append('TICK')
circ.append('M', 3)
circ.append('M', 4)

# sample the measurements
# should be the syndrome
circ.sample()
```

```
[1 0]
```

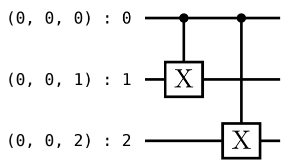
Correcting the error

Once, Bob has inferred an error, he can fix the corrupted codeword by applying the inverse of the error. In this case, the error operators are self-inverse, so he just has to apply them again to fix the corrupted codeword, eg. if he thinks error X_1 has occurred, he applied X_1 to the corrupted state to fix it.

Decoding

Finally, Bob can undo the encoding operation to recover the one-qubit state Alice meant to send. Since, the encoding operation was self-inverse, Bob just has to pass $\left|\bar{\psi}\right\rangle$ through it to recover $|\psi\rangle$.

```
dec_circ = stac.Circuit.simple(3)
dec_circ.append('CX', 0, 1)
dec_circ.append('CX', 0, 2)
dec_circ.draw()
```



A note on probability of success

We don't have to repeat our analysis of the probability of success as it is exactly the same as in the classical case. If p is less than half, then using the quantum repetition code will be beneficial.

Question: Consider the 5-qubit repetition code, in which the coding operates in a similar fashion as the 3-qubit one we have discussed, but with five qubits instead of three. Can this larger code possibly correct errors in which two qubits are flipped?

A linear algebraic analysis of the quantum repetition code

One way of understanding why the code works is by understanding the vector spaces we have dealt with above.

First, there was the 2-dimensional Hilbert space of the unencoded logical qubit. This was mapped to a 2-dimensional subspace of the 2^3-dimensional Hilbert space of the three data qubits, the latter of which is called the codespace, and the former the quantum code. Errors move the encoded state into other 2-dimensional subspaces of the codespace. For instance, the quantum code had a basis $\{|000\rangle, |111\rangle\}$. The error X_1 moved the state to a subspace with basis $\{|010\rangle, |101\rangle\}$. We can tabulate all these movements as follows.

Error	Subspace basis
I	$\{|000\rangle, |111\rangle\}$
X_0	$\{|100\rangle, |011\rangle\}$
X_1	$\{|010\rangle, |101\rangle\}$
X_2	$\{|001\rangle, |110\rangle\}$

From this analysis we can see why the code works to correct bit-flip errors. Each error doesn't fundamentally destroy the state, it only moves it to a different subspace. The process of error-detection is to determine which subspace we are in, which does not distort the actual state. The process of error-correction moves us back to the original quantum code subspace. We will use this notion of subspaces again when we formally define quantum error-correction.

Question: Recall that a vector subspace is characterized by a projector $\Pi = \sum_i |\psi_i\rangle \langle\psi_i|$, where $\{|\psi_i\rangle\}_i$ is a basis for the subspace. Show that $\Pi^2 = \Pi$.

Question: Determine the projector onto the code for the three-qubit code.

Correcting X rotation errors

The code we have created can correct errors besides simple X errors. The unitary,

$$R_x^{(i)}(\theta) = \cos\theta I - i\sin\theta X^{(i)}, \tag{31}$$

is the rotation about the X axis on the Bloch sphere for the i-th qubit. Suppose, the encoded state $\left|\bar{\psi}\right\rangle$ is effected by this unitary during its passage through the communication channel between Alice and Bob. We now show that the error detection and correction procedures we have described above deal with this error, and Bob has to do nothing extra to correct for such an error.

To see this we calculate the corrupted codeword state,

$$\left|\tilde{\psi}\right\rangle = R_x^{(i)}(\theta)\left|\bar{\psi}\right\rangle = \cos\theta\left|\bar{\psi}\right\rangle - i\sin\theta X^{(i)}\left|\bar{\psi}\right\rangle. \tag{32}$$

When, we run this state through the error-detection circuit (in which we append two ancillas to the state), we obtain before the measurement step, the state

$$(\cos\theta\left|\bar{\psi}\right\rangle - i\sin\theta X^{(i)}\left|\bar{\psi}\right\rangle)|00\rangle \rightarrow \cos\theta\left|\bar{\psi}\right\rangle|00\rangle - i\sin\theta X^{(i)}\left|\bar{\psi}\right\rangle|s\rangle, \tag{33}$$

where $|s\rangle$ is the syndrome associated with $X^{(i)}$. When we measure the ancilla, either the measurement result is 00 (with probability $|\cos\theta|^2$) and the corrupted codeword state collapses to $\left|\bar{\psi}\right\rangle$, or the measurement result is s (with probability $|\sin\theta|^2$) and the corrupted codeword state collapses to $X^{(i)}\left|\bar{\psi}\right\rangle$. In either case, the subsequent error-correction procedure will correct the state.

What we see here is that linearity of quantum mechanics and the collapse mechanism on measurement allows us to correct errors which are a linear combination of errors in E. This is an incredibly powerful result, and we will prove it more generally later on.

Phase-flip errors

Quantum bit-flip errors are only one kind of errors. Quantum systems in interacting with the environment can suffer from other types of error. One such possibility is phase-flip errors, i.e. where the qubits can be acted upon by the Z operator. This code fails to identify phase-flip errors and therefore cannot correct them either, which is why we deliberately excluded such errors from our noise channels.

To see this note that the action of, say, Z_1 on the logical state is

$$Z_1 \left|\bar{\psi}\right\rangle = Z_1(\alpha\,|000\rangle + \beta\,|111\rangle) = \alpha\,|000\rangle - \beta\,|111\rangle. \tag{34}$$

This state is identical to a different state Alice might have sent ($\alpha\,|0\rangle - \beta\,|1\rangle$), so there is no way that Bob can determine that an error has occurred. In the subspace picture discussed above single-qubit Z errors keep the state within the code. This makes it impossible, both here and generally, for the error to be detected. A well-designed code ensures that every possible error moves the state out of the code and into some other subspace of the codespace.

A classification of errors by detectability and correctability (optional)

We have discovered that not every error E that could occur is detectable or correctable. Let us break this down into a set of categories.

1. E distorts $\left|\bar{\psi}\right\rangle$ in a way that the effects of E can be reversed. For the bit-flip repetition code $E = X_0$ is an example. And generally all single-qubit bit-flip Es are corrected by the code.

2. E distorts $\left|\bar{\psi}\right\rangle$ in a way that the effects are not corrected. There are two varieties of this.

   a. If $E = X_0X_1X_2$, then $|000\rangle$ is taken to $|111\rangle$ and vice versa - i.e. the error distorts one codeword into another codeword. Another example is the single or triple qubit phase-flips. This means that the error-detection procedure thinks no error has occurred.

   b. If $E = X_0X_1$, then the error-detection process thinks that $E = X_2$ has occurred. A similar wrong inference of the error happens for any other two-qubit bit-flip error. Subsequently error-correction procedure does the wrong correction.

3. E does not distort $\left|\bar{\psi}\right\rangle$ at all, i.e. $E\left|\bar{\psi}\right\rangle = \left|\bar{\psi}\right\rangle$. For instance, $E = Z_0Z_1$ does not effect the encoded state at all. Therefore, the code is immune to such Es.

The last category is the most surprising one of all that does not occur in classical error-correction at all, because there are no phase-flips in the classical case.

Summarizing, suppose Alice sends the message $|\psi\rangle$ to Bob. Then Bob's estimated message depending on the error that occurs is given in the table below.

Error	**Estimated message** $\left|\hat{\psi}\right\rangle$
X_i	$\|\psi\rangle$
X_iX_j	$X\,\|\psi\rangle$
$X_0X_1X_2$	$X\,\|\psi\rangle$
Z_i	$Z\,\|\psi\rangle$
Z_iZ_j	$\|\psi\rangle$
$Z_0Z_1Z_2$	$Z\,\|\psi\rangle$

Quantum repetition code for phase-flips

The code presented in the previous notebook could only correct for X errors. Now, we present a code that can only correct for Z errors. In the next notebook, we will combine these codes together to correct for both X and Z errors and more.

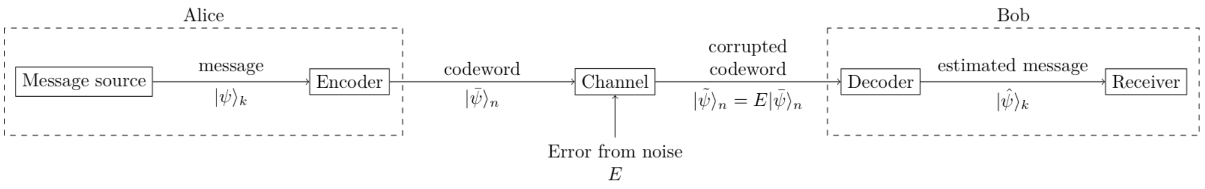
Setup

Alice wants to send a one-qubit state $|\psi\rangle$ to Bob, but via a channel that applies the Z operator to qubits with probability p. If the error occurs then the state is distorted as,

$$\left|\tilde{\psi}\right\rangle = Z|\psi\rangle = Z(\alpha|0\rangle + \beta|1\rangle) = \alpha|0\rangle - \beta|1\rangle. \tag{35}$$

Bob cannot tell if an error has occurred, so with probability p he will receive the wrong message.

Alice and Bob therefore use an error-correcting procedure.



Here we assume that on each qubit, Z_i occurs with probability p.

Encoding

To protect against such type of errors, we will employ a repetition code, but with a different basis. Now,

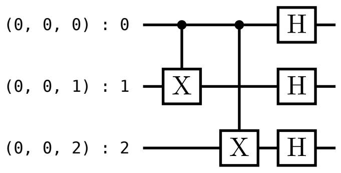
$$|0\rangle \to \left|\bar{0}\right\rangle = |+++\rangle, \tag{36}$$
$$|1\rangle \to \left|\bar{1}\right\rangle = |---\rangle. \tag{37}$$

For this code basis, a qubit in state $|\psi\rangle$ is encoded as

$$|\psi\rangle|00\rangle = (\alpha|0\rangle + \beta|1\rangle)|00\rangle \to \left|\bar{\psi}\right\rangle = \alpha|+++\rangle + \beta|---\rangle. \tag{38}$$

This is done via the following circuit.

```python
import stac
enc_circ = stac.Circuit.simple(3)
enc_circ.append('CX', 0, 1)
enc_circ.append('CX', 0, 2)
for i in range(3):
    enc_circ.append('H', i)
enc_circ.draw()
```



Task 1 (On paper)

Determine logical gate operations $\bar{X}$ and $\bar{Z}$ for the three-qubit repetition code for phase-flips. These are operations that act on the logical basis $\{\left|\bar{0}\right\rangle, \left|\bar{1}\right\rangle\}$, in the normal way, i.e.

$$\bar{X}\left|\bar{0}\right\rangle = \left|\bar{1}\right\rangle, \quad \bar{X}\left|\bar{1}\right\rangle = \left|\bar{0}\right\rangle, \tag{39}$$
$$\bar{Z}\left|\bar{0}\right\rangle = \left|\bar{0}\right\rangle, \quad \bar{Z}\left|\bar{1}\right\rangle = -\left|\bar{1}\right\rangle. \tag{40}$$

They can be constructed by some combination of operations on the three physical qubits. You will discover that there are possibly multiple ways of doing so.

Errors

The possible errors on the encoded state are given as follows.

Error	Probability
I	$(1-p)^3$
Z_0	$p(1-p)^2$
Z_1	$p(1-p)^2$
Z_2	$p(1-p)^2$
Z_0Z_1	$p^2(1-p)$
Z_0Z_2	$p^2(1-p)$
Z_1Z_2	$p^2(1-p)$
$Z_0Z_1Z_2$	p^3

Task 2 (On paper)

Determine the impact of each error on the input basis states.

Let input state be $|+++\rangle$

Error	Output state
I	$\|+++\rangle$
Z_0	$\|-++\rangle$
Z_1	$?$
Z_2	$?$
Z_0Z_1	$?$
Z_0Z_2	$?$
Z_1Z_2	$?$
$Z_0Z_1Z_2$	$?$

Let input state be $|---\rangle$

Error	Output state
I	$\|---\rangle$
Z_0	$\|+--\rangle$

Error	Output state
Z_1	?
Z_2	?
Z_0Z_1	?
Z_0Z_2	?
Z_1Z_2	?
$Z_0Z_1Z_2$	?

Do different errors result in the same output state?

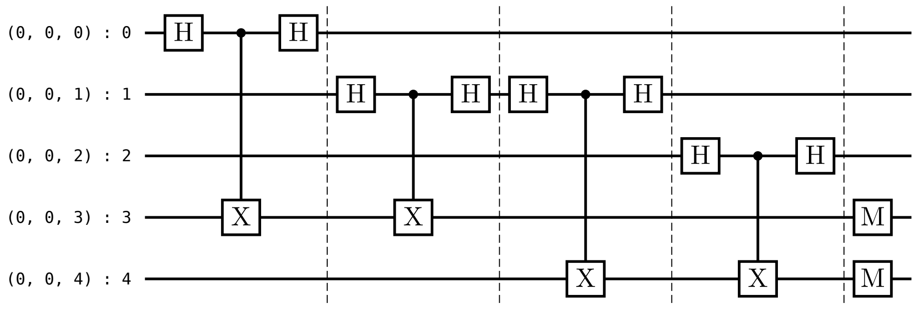
As before, we will assume that p is small, so we will only attempt to correct the errors $\{I, Z_0, Z_1, Z_2\}$. The effect of the Z errors is to flip the state between plus and minus. So, for instance,

$$Z_0 \left|\bar{\psi}\right\rangle = \alpha Z_0 \left|+++\right\rangle + \beta Z_0 \left|---\right\rangle = \alpha \left|-++\right\rangle + \beta \left|+--\right\rangle. \tag{41}$$

Decoding

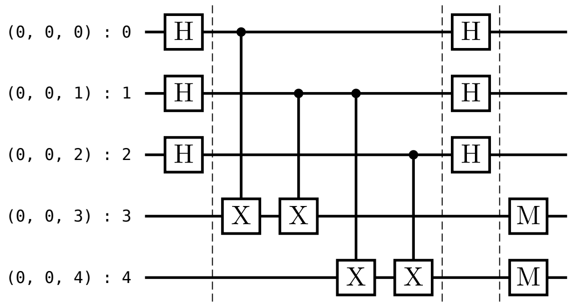
The error detection strategy is the same as before. Bob employs a circuit that compares the value of two pairs of qubits, but in the plus/minus basis. This is accomplished using the following circuit.

```
synd_circ_expanded = stac.Circuit.simple(5)
synd_circ_expanded.append('H', 0)
synd_circ_expanded.append('CX', 0, 3)
synd_circ_expanded.append('H', 0)
synd_circ_expanded.append('TICK')

synd_circ_expanded.append('H', 1)
synd_circ_expanded.append('CX', 1, 3)
synd_circ_expanded.append('H', 1)
synd_circ_expanded.append('TICK')


synd_circ_expanded.append('H', 1)
synd_circ_expanded.append('CX', 1, 4)
synd_circ_expanded.append('H', 1)
synd_circ_expanded.append('TICK')


synd_circ_expanded.append('H', 2)
synd_circ_expanded.append('CX', 2, 4)
synd_circ_expanded.append('H', 2)

synd_circ_expanded.append('TICK')
synd_circ_expanded.append('M', 3)
synd_circ_expanded.append('M', 4)

synd_circ_expanded.draw()
```



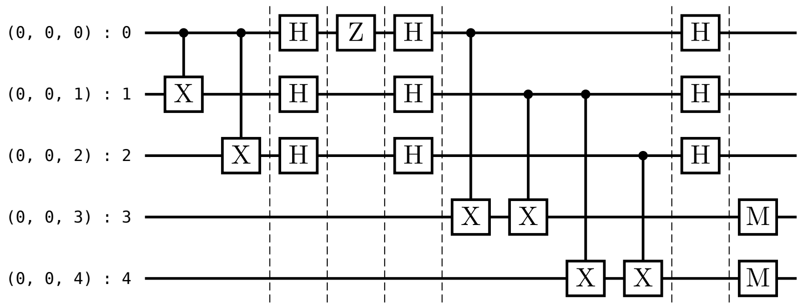
Here in each block, first we use the Hadamard gate to switch to the computational basis, then transfer the value of the qubit to the ancilla qubit using the CX gate. Finally, we convert back to the plus/minus basis using the Hadamard gate. We can simplify this circuit as follows, using the fact that $H^2 = I$.

```
synd_circ = stac.Circuit.simple(5)
for i in range(3):
    synd_circ.append('H', i)
synd_circ.append('TICK')
synd_circ.append('CX', 0, 3)
synd_circ.append('CX', 1, 3)
synd_circ.append('CX', 1, 4)
synd_circ.append('CX', 2, 4)
synd_circ.append('TICK')

for i in range(3):
    synd_circ.append('H', i)
synd_circ.append('TICK')

synd_circ.append('M', 3)
synd_circ.append('M', 4)
synd_circ.draw()
```



To determine the syndromes, we can do the following

```
circ = stac.Circuit.simple(5)
# encode the 0 state
circ += enc_circ

# # add an error
circ.append('TICK')
circ.append('Z', 0)
circ.append('TICK')

# # do a syndrome measurement
circ += synd_circ

# draw to make sure you we understand what is happening
circ.draw()

# sample the output 10 times
circ.sample(10)
```



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

Task 3

Fill the following table

Syndrome	**Inferred error**
00	I
01	
10	
11	

In this code, if Bob infers that errors Z_i occurs, than he applies Z_i to the corrupted codeword to fix it.

Bit-flip errors

We won't belabor the point that this code cannot fix bit-flip errors.

The Shor Code

It is now time to construct a code that correct more than one type of error. This will also give us the opportunity to show off a code construction technique called code concatenation.

In code concatenation, we first encode the logical qubit using a particular code into n data qubits. Then, we encode each of then n data qubits individually using the same or another code. To illustrate this, recall that, in the phase-flip code the logical basis states are encoded as

$$|0\rangle \rightarrow \left|\bar{0}\right\rangle = |+++\rangle, \tag{42}$$

$$|1\rangle \rightarrow \left|\bar{1}\right\rangle = |---\rangle. \tag{43}$$

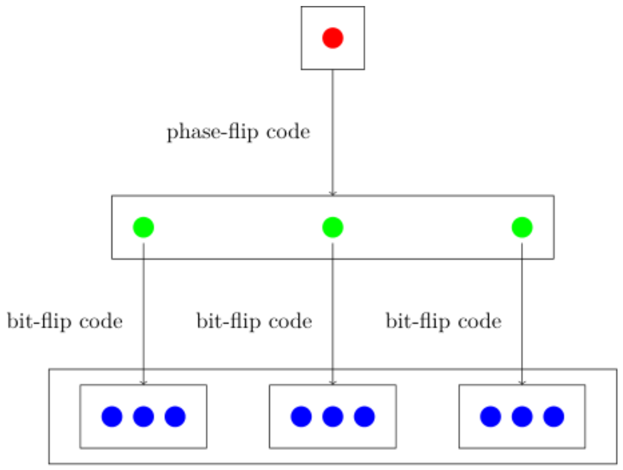
Now, we are going to take each of the three data qubits and encode them using the bit-flip repetition code. Recall that in the bit-flip repetition code, the plus and minus states encode as

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \to |\bar{+}\rangle = \frac{|000\rangle + |111\rangle}{\sqrt{2}}, \tag{44}$$
$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \to |\bar{-}\rangle = \frac{|000\rangle - |111\rangle}{\sqrt{2}}. \tag{45}$$

When we apply this encoding to each of the three qubits in the phase-flip encoding, we obtain

$$|+++\rangle \to \left|\bar{\bar{0}}\right\rangle = \frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}, \tag{46}$$
$$|---\rangle \to \left|\bar{\bar{1}}\right\rangle = \frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}. \tag{47}$$

Visually, we can draw a tree diagram to show how one qubit is first encoded into three via the phase-flip code and then each of the three qubits is encoded into three more via the bit-flip code.



Here we refer to the phase-flip code as the outer code (level 1) and the bit-flip code as the inner code (level 2).

Encoding

This combined action that maps one logical qubit to the state of nine qubits as

$$|0\rangle \to \left|\bar{\bar{0}}\right\rangle = \frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}, \tag{48}$$
$$|1\rangle \to \left|\bar{\bar{1}}\right\rangle = \frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}, \tag{49}$$

defines the encoding for the Shor code.

Task 1

Create the encoding circuit for the Shor code. You will use nine qubits.

1. Apply the phase-flip repetition code's encoding circuit to qubits 0, 3, 6.
2. Apply the bit-flip repetition code's encoding circuit to
   - qubits 0,1,2
   - qubits 3,4,5
   - qubits 6,7,8

Then use the `circ.simulate()` method to check if you get the correct output.

```
import stac
enc_circ = stac.Circuit.simple(9)
```

Question: Construct the basis states of a nine-qubit code with the bit-flip code as the outer code and the phase-flip code as the inner code.

Errors and decoding

We will now show that this code can correct every possible single-qubit error. Let's first show that it can correct bit-flip and phase-flip errors on any of the nine data qubits.

X-errors

Suppose a single-qubit X error occurs, on say the fourth qubit.

$$X_4 \left|\bar{\bar{0}}\right\rangle = \frac{(|000\rangle + |111\rangle)(|010\rangle + |101\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}, \tag{50}$$

$$X_4 \left|\bar{\bar{1}}\right\rangle = \frac{(|000\rangle - |111\rangle)(|010\rangle - |101\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}, \tag{51}$$

How can one detect that this occur occured? We will do the following syndrome measurements, in the style of the bit-flip code.

- Compare the values of qubits 0,1,2
- Compare the values of qubits 3,4,5
- Compare the values of qubits 6,7,8

In this example, comparing the value of the 3rd qubit with the 4th, and the 4th with the 5th will show that the 3rd qubit has a different value. The error can be fixed as before.

Task 2

Create the syndrome measurement circuit for detecting X errors in the Shor code.

```
# specify how many qubits you will need
sync_circ_x = stac.Circuit.simple()
```

Z-errors

The case for a single-qubit Z errors is slightly more difficult to see. Note, for instance, that if a Z error occurs on any one of the first three qubits, the sign in the first block will change. For $i = 0, 1, 2$,

$$Z_i \left|\bar{\bar{0}}\right\rangle = \frac{(|000\rangle - |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}, \tag{52}$$

$$Z_i \left|\bar{\bar{1}}\right\rangle = \frac{(|000\rangle + |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}. \tag{53}$$

A different way of seeing it is at the outer code level, where the encoding is

$$\left|\bar{\bar{0}}\right\rangle = |\bar{+}\rangle\,|\bar{+}\rangle\,|\bar{+}\rangle, \tag{54}$$

$$\left|\bar{\bar{1}}\right\rangle = |\bar{-}\rangle\,|\bar{-}\rangle\,|\bar{-}\rangle. \tag{55}$$

Then the action of Z_i for $i = 0, 1, 2$ is

$$Z_i \left|\bar{\bar{0}}\right\rangle = |\bar{-}\rangle\,|\bar{+}\rangle\,|\bar{+}\rangle, \tag{56}$$

$$Z_i \left|\bar{\bar{1}}\right\rangle = |\bar{+}\rangle\,|\bar{-}\rangle\,|\bar{-}\rangle. \tag{57}$$

At this level, it is quite easy to see the error-detecting strategy. Apply the phase-flip code's error-detection procedure to the three encoded qubits (at the outer level). We will discuss how one can do this later.

Y-errors

Right now, we want to show that the Shor code can detect and correct errors beyond just X and Z errors. One such error is the Y error, which is just $Y = iZX$, i.e. a combined bit-flip and phase-flip error. For instance, Y_4 will result in the corrupted basis states

$$Y_4\left|\bar{\bar{0}}\right\rangle = i\frac{(|000\rangle + |111\rangle)(|010\rangle - |101\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}, \tag{58}$$

$$Y_4\left|\bar{\bar{1}}\right\rangle = i\frac{(|000\rangle - |111\rangle)(|010\rangle + |101\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}, \tag{59}$$

This error is detected at both the X detection stage, and the Z detection stage, and corrected at both as well. First, we detect X errors, find that the fourth is flipped and fix it. Then we detect Z errors and determine that the middle block has experienced an error and fix it.

Hence, we have shown up till now that the Shor code can correct all errors in the set,

$$\mathcal{E} = I \,\cup\, \{X_i\}_i \,\cup\, \{Y_i\}_i \,\cup\, \{Z_i\}_i, \quad i = 0, \ldots, 8. \tag{60}$$

More general errors

We showed before that the bit-flip code could also correct for X rotation errors. By the same arguments of linearity and collapse, the Shor code (and most quantum codes we will encounter) can correct any error which is of the form,

$$E_i = e_0 I + e_1 X_i + e_2 Y_i + e_3 Z_i. \tag{61}$$

Task 3 (On paper)

Apply E_i to $\left|\bar{\psi}\right\rangle$ and then show that the error-detection circuit will collapse the state to just one of the possible errors.

Question: Suppose the state $\left|\bar{\psi}\right\rangle$ encoded by the Shor code undergoes the two-qubit error Z_1Z_2. What is the impact of this error? This is a phenomena not seen in classical codes.

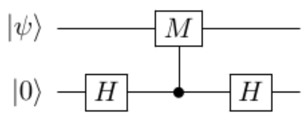
Question: The Shor can correct some (but not all) two-qubit errors as well. Characterize all these errors. Is there any three-qubit error that the Shor code can handle?

Syndrome measurements, an alternate view

In this notebook, we will further explore a technique to measure qubits, that we used before for the repetition and Shor codes. Recall, that there our goal was not to perform a destructive measurement of the entire state, but only a partial measurement that determined whether an error had occured on the state or not. Let's see if we can generalize this.

Suppose you have a qubit in some unknown state $|\psi\rangle$, and you want to measure operator M on it. We will assume for simplicity M has eigenvalues ± 1, which is true for all Pauli operators.

To make our measurement, we are going to use an ancilla to store the results of our measurement. This is achieved with the following simple circuit.



To see how this works, let's first look at an example.

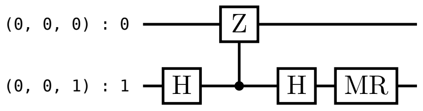
Non-destructive measurement of Z

Let $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, and let $M = Z$ be the operator we want to measure. What is the expected result?

measurement outcome	probability	post-measure state of qubit 0
0	$\lvert\alpha\rvert^2$	$\lvert 0\rangle$
1	$\lvert\beta\rvert^2$	$\lvert 1\rangle$

Now, to do this measurement, we append an ancilla to the state and execute the following circuit, where the 0th qubit is the one being measured, and the 1st qubit is the ancilla.

```python
import stac
circ = stac.Circuit.simple(2)
circ.append('H', 1)
circ.append('CZ', 1, 0)
circ.append('H', 1)
circ.append('MR', 1)
circ.draw()
```



Let's work our way through this circuit (ignoring any normalization factors). First,

$$\begin{aligned}
|\psi\rangle|0\rangle &\xrightarrow{H} |\psi\rangle|0\rangle + |\psi\rangle|1\rangle, && (62)\\
&\xrightarrow{CZ_{10}} |\psi\rangle|0\rangle + (Z|\psi\rangle)|1\rangle, && (63)\\
&\xrightarrow{H} |\psi\rangle(|0\rangle + |1\rangle) + (Z|\psi\rangle)(|0\rangle - |1\rangle), && (64)\\
&= (|\psi\rangle + Z|\psi\rangle)|0\rangle + (|\psi\rangle - Z|\psi\rangle)|1\rangle, && (65)\\
&= (\alpha|0\rangle + \beta|1\rangle + \alpha|0\rangle - \beta|1\rangle)|0\rangle + (\alpha|0\rangle + \beta|1\rangle - \alpha|0\rangle + \beta|1\rangle)|1\rangle, && (66)\\
&= \alpha|00\rangle + \beta|11\rangle && (67)
\end{aligned}$$

At this point, if we measure the ancilla, then the outcomes are

measurement outcome	probability	post-measure state of qubit 0
0	$\lvert\alpha\rvert^2$	$\lvert 0\rangle$

measurement outcome	probability	post-measure state of qubit 0
1	$\lvert\beta\rvert^2$	$\lvert 1\rangle$

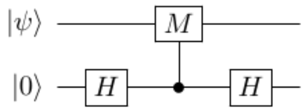
As we can see, this is exactly what we wanted.

Question: Insert the normalization factors into the circuit above and ensure the state remains normalized.

Non-destructive measurements of M

Case 1: M is a one-qubit operator

In general, how does the mathematics work out. We have the circuit



The state at the end of the circuit will be

$$(|\psi\rangle + M|\psi\rangle)|0\rangle + (|\psi\rangle - M|\psi\rangle)|1\rangle \tag{68}$$

Let M have an eigenbasis $\{|e_0\rangle, |e_1\rangle\}$, i.e.

$$M|e_0\rangle = |e_0\rangle, \tag{69}$$
$$M|e_1\rangle = -|e_1\rangle. \tag{70}$$

Then, if we expand $|\psi\rangle = \alpha|e_0\rangle + \beta|e_1\rangle$, the algebra is the same,

$$\begin{aligned}(|\psi\rangle + M|\psi\rangle)|0\rangle + (|\psi\rangle - M|\psi\rangle)|1\rangle &= (\alpha|e_0\rangle + \beta|e_1\rangle + \alpha|e_0\rangle - \beta|e_1\rangle)|0\rangle + (\alpha|e_0\rangle + \beta|e_1\rangle - \alpha|e_0\rangle + \beta|e_1\rangle)|1 \\ &= \alpha|e_0 0\rangle + \beta|e_1 1\rangle.\end{aligned}$$

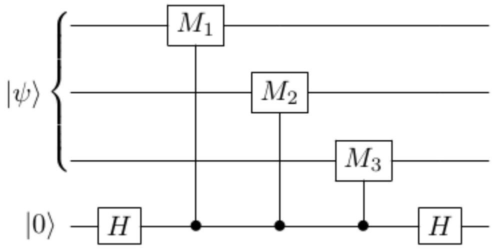
Now, if we measure the ancilla, then outcomes are

measurement outcome	probability	post-measure state of qubit 0
0	$\lvert\alpha\rvert^2$	$\lvert e_0\rangle$
1	$\lvert\beta\rvert^2$	$\lvert e_1\rangle$

In other words, measuring the ancilla in the computational basis has measured qubit 0 in the M basis.

Case 2: M is a multi-qubit operator

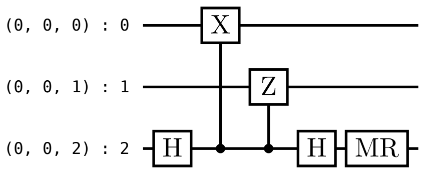
Does this process work if we want to measure a multi-qubit state, using a multi-qubit Pauli operator? As an example, let $|\psi\rangle$ be a three-qubit state, and let $M = M_1 \otimes M_2 \otimes M_3$. Again, we assume that each M_i has eigenvalues ± 1. Then, the circuit will be as follows.



The condition on the eigenvalues of M_i means that M will also only have eigenvalues ± 1. Hence, the mathematics will continue to hold (check if you are unconvinced) and measuring the syndrome qubits in the computational basis will be equivalent to measuring the data qubits in M basis.

So, to measure, say X_0Z_1, we would implement the circuit below.

```
circ = stac.Circuit.simple(3)
circ.append('H', 2)
circ.append('CX', 2, 0)
circ.append('CZ', 2, 1)
circ.append('H', 2)
circ.append('MR', 2)
circ.draw()
```



Measurement of eigenstates

One of the features of the measurement process described is that measuring an arbitrary operator on a state will destroy the superposition of the state (as governed by the rules of quantum mechanics). However, if the data qubits are in state $|\psi\rangle$. which is an eigenstate of the operator M, then there is no destruction.

Let's examine two possibilities for the state at the end of the circuit, but before measurement.

$$|\Phi\rangle = (|\psi\rangle + M|\psi\rangle)|0\rangle + (|\psi\rangle - M|\psi\rangle)|1\rangle, \tag{73}$$

where $|\psi\rangle$ is an n-qubit state and M is any n-qubit Pauli operator.

Case 1: +1 eigenstate.

If $M|\psi\rangle = |\psi\rangle$, then

$$|\Phi\rangle = |\psi\rangle|0\rangle \tag{74}$$

Then, the outcome of the measurement will be deterministically 0 and the state of the data qubits will be unchanged.

Case 1: -1 eigenstate.

If $M|\psi\rangle = -|\psi\rangle$, then

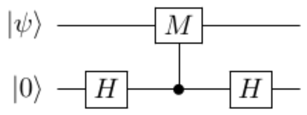
$$|\Phi\rangle = |\psi\rangle|1\rangle \tag{75}$$

Then, the outcome of the measurement will be deterministically 1 and the state of the data qubits will be unchanged.

This analysis will be useful when we construct stabilizer quantum error-correction codes because the encoded state will always be an eigenstate of certain operators.

The special case of X and Z operators

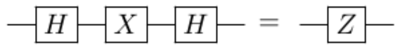
The circuit



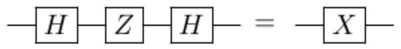
doesn't look like anything we saw before. But if M is X or Z then it can be transformed into a different circuit.

Recall the circuit identities:

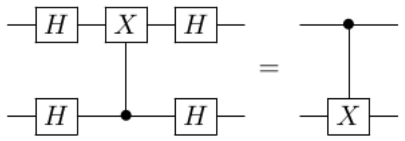
1. X conjugation by H



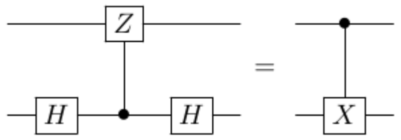
2. Z conjugation by H



3. CX conjugation by H

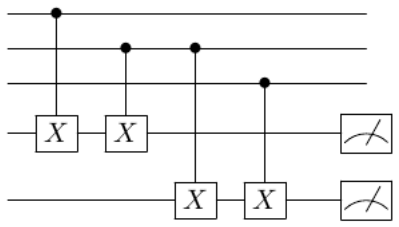


Here, if we apply identity 1 to the first qubit, we obtain

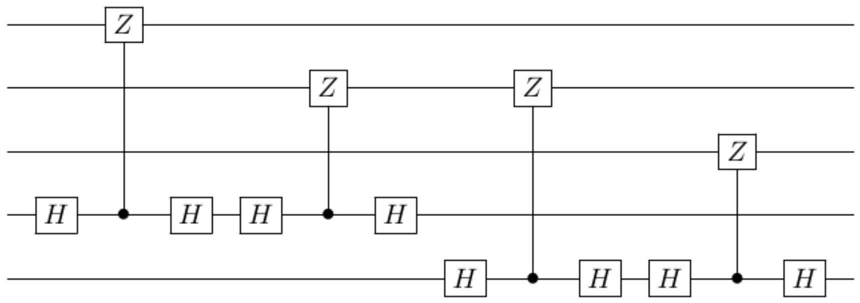


Repetition code for bit-flips

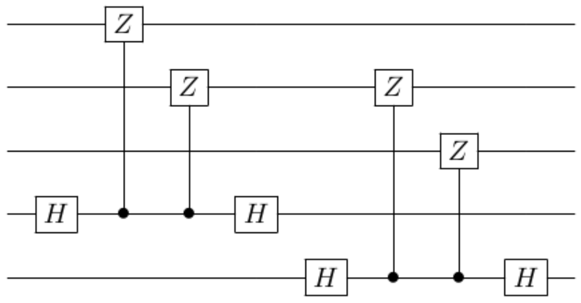
Recall that the syndrome circuit for the bit-flip repetition code was



Using the third identity above, we can see that it transforms to



Then, we use $HH = I$ to get



Hence, we can see that in the repetition code, we are actually doing measurements of $Z \otimes Z \otimes I$ and $I \otimes Z \otimes Z$.

Task 1 (On paper)

What operators are being measured in the repetition code for phase-flips?

Quantum codes

We proceed as before by defining the message space, the codespace and finally the code. The quantum message space consists of states of qubits.

> For a quantum code, the **message space** is the state space of k-qubits, i.e. the 2^k-dimensional Hilbert space $\mathcal{H}_m$. Any $|\psi\rangle \in \mathcal{H}^m$ is a valid message.

The quantum codespace is constructed from the state of n-qubits.

> For a quantum code, the **codespace** is the state space of n-qubits, i.e. the 2^n-dimensional Hilbert space $\mathcal{H}^c$.

Finally, we can define the code.

> A **qubit quantum code** $\mathcal{C}$ is a 2^k-dimensional subspace of the codespace $\mathcal{H}^c$. Such a code is labelled as $[[n, k]]$.

Elements of the code are codewords, and if we choses a basis for the code, then the basis elements are also called basis codewords.

Quantum error-correcting codes

Given a quantum code, it is not clear if it is useful. The above definition only talks about how to encode a message into the codespace. It does not speak at all about which errors can be detected or corrected with such an encoding. Why is this a problem, when this was not a significant problem for classical codes. There are three critical issues which make the construction of a quantum error-correction codes a more difficult task than their classical counterparts.

- The no-cloning theorem forbids us from copying quantum states. This limits the types of operations we can perform during the decoding process.
- Measuring quantum states in superposition destroys the superposition, so we can't simply measure the received blocks to determine what state they are in.
- Classical codes only have to deal with bit-flip errors, but quantum codes will have to deal with any sort of unitary or non-unitary noisy interactions. This includes phase errors, and it includes continuous rotations etc.

Fortunately, we will discover that all of these challenges can be surmounted. To build the theory to do so we start by explicitly defining what conditions make a quantum code an error-correcting one.

An encoding map is a transformation from the message space to the code, the subspace of the codespace, formally defined as follows.

> Given a quantum code $\mathcal{C}$, an **encoding map** or encoder is a unitary $U : \mathcal{H}^m \to \mathcal{H}^c$ that maps the message space to the code.

Given any message $|\psi\rangle \in \mathcal{H}^m$, the state $U|\psi\rangle$ is an element of the code C. The encoding map is akin to the generator matrix for classical codes.

What we require is that every code C be paired with a set of errors $\mathcal{E}$ that it can correct, and that there exist a process to actually do so.

> A **quantum error-correction code** $(C, \mathcal{E})$ is a quantum code along with a set of errors $\mathcal{E}$, such that given an encoding map U associated with the code, there exists a quantum channel $\mathcal{D}$, called the decoder, such that for all $E \in \mathcal{E}$ and for all $|\psi\rangle \in \mathcal{H}^m$,
>
> $$\mathcal{D}\left(EU|\psi\rangle\langle\psi|U^\dagger E^\dagger\right) = c(E, |\psi\rangle)\,|\psi\rangle\langle\psi|, \tag{76}$$
>
> where $c(E, |\psi\rangle)$ is a constant.

This definition postulates that the existence of a decoder is necessary for a code to be an error-correcting one. When can such a decoder exist? Whenever, there is enough information extractable from every corrupted codeword so as to reverse the effects of the error. Let's build some basic results that will lead us to some precise mathematical conditions.

- First, note that given any error E_a and any state $|\psi\rangle \in \mathcal{C}$ the corrupted codeword $\left|\tilde{\psi}\right\rangle = E_a |\psi\rangle$ must be completely outside the code. If it is not, what would happen? We could write $|\psi\rangle = |\psi_c\rangle + |\psi_\perp\rangle$ such that $E_a |\psi_\perp\rangle \in \mathcal{C}$ and $E_a |\psi_\perp\rangle \in \mathcal{C}^\perp$. Now, $|\psi\rangle$ is in the code, then so is $|\psi_c\rangle$ (up to a normalization) and it is some other valid codeword. Hence, the effect of the error E_a on $|\psi_c\rangle$ is to keep it inside the code, i.e.\ map it onto a different codeword. This is clearly a bad situation because Bob would just assume that no error has occurred and $E_a |\psi_c\rangle$ is the codeword Alice meant to send. Hence, by contradiction, we infer that our code must be designed so that all errors move every codeword out of the code.

- Next, suppose we have an orthogonal basis $\{|\psi_i\rangle\}_i$ for $\mathcal{C}$. Given any error E_a, what do we require of $E_a |\psi_i\rangle$ and $E_a |\psi_j\rangle$ for $i \neq j$, so the decoder can function. We will need these two corrupted codewords to remain orthogonal, so that the decoder has enough information to correctly decode. To see this, let's assume to the contrary that the corrupted states are not orthogonal, i.e. $(\langle\psi_i|E_a^\dagger)(E_a |\psi_j\rangle) \neq 0$. To tease out the non-orthogonal part of this inner product, let's write

$$|\psi_j\rangle = \left|\psi_\parallel\right\rangle + |\psi_\perp\rangle, \tag{77}$$

such that $E_a \left|\psi_\parallel\right\rangle$ is parallel to $E_a |\psi_i\rangle$, while $E_a |\psi_\perp\rangle$ is perpendicular to $E_a |\psi_i\rangle$. As in the argument above, $\left|\psi_\parallel\right\rangle$ must also be a valid codeword (because it is part of $|\psi_j\rangle$). And because $|\psi_i\rangle$ and $|\psi_j\rangle$ are perpendicular, $\left|\psi_\parallel\right\rangle$ and $|\psi_i\rangle$ are distinct codewords. Hence, we discover that two distinct codewords, $\left|\psi_\parallel\right\rangle$ and $|\psi_i\rangle$, when acted upon by error E_a result in the same corrupted codeword. This is again a bad situation, because the decoder will be unable to decide how to fix the error on the corrupted codeword. Hence, in order to have an error-correcting code that works, we need each error to map orthogonal basis vectors of C to orthgonal states,

$$(\langle\psi_i|E_a^\dagger)(E_a |\psi_j\rangle) = 0, \quad i \neq j. \tag{78}$$

- Let's now make an even stronger condition on our code. Consider the corrupted codewords $E_a |\psi_i\rangle$ and $E_b |\psi_j\rangle$, i.e. distinct errors on two distinct basis states. Do these need to be orthogonal? By exactly the same argument as above, yes. In words, if these two corrupted codewords are not orthogonal, then there is a part of $|\psi_j\rangle$, called $\left|\psi_\parallel\right\rangle$, that under the action of E_b becomes parallel to $E_a |\psi_i\rangle$. As above, $\left|\psi_\parallel\right\rangle$ is a valid codeword. If the Bob receives $E_a |\psi_i\rangle = E_b \left|\psi_\parallel\right\rangle$, then his decoder will be unable to tell if Alice sent $|\psi_i\rangle$ which was distorted by E_a or $\left|\psi_\parallel\right\rangle$ which was distorted by E_b. Hence, we can conclude that for a valid error-correcting code,

$$(\langle\psi_i|E_a^\dagger)(E_b |\psi_j\rangle) = 0, \quad i \neq j. \tag{79}$$

- What about the case when $i = j$ with distinct errors? Meaning, is there any requirement on $E_a |\psi_i\rangle$ and $E_b |psi_i\rangle$? First of all, let's make it clear that these don't need to be orthogonal - though they can be. We saw the example of the phase-flip code where the action of Z on different qubits yielded the same corrupted codeword. But this was not a problem, because in each case, the correction operation was the same. So up till now, our condition is

$$\langle\psi_i|E_a^\dagger E_b |\psi_j\rangle = \delta_{ij} A, \tag{80}$$

where the δ_{ij} ensures that the inner product is zero if we are dealing with two different basis states, and possibly non-zero if they are the same. This last part is determined by the unknown constant A. However, we can recognize that if $E_a |\psi_i\rangle$ and $E_b |\psi_i\rangle$ are indeed orthogonal for every pair of errors E_a and E_b, then our code will be an error-correcting one. We assume,

$$\langle\psi_i|E_a^\dagger E_b |\psi_j\rangle = \delta_{ij}\delta_{ab} A. \tag{81}$$

Now, recall that when we went from the noise channel to the set $\{E_a\}_a$ we choose from the one of the infinite such sets. Let's transform to a different set $\{F_c\}$ where

$$F_c = \sum_a \alpha_{ca} E_a. \tag{82}$$

Now, let's evaluate the inner product

$$\begin{aligned}
\langle\psi_i|F_c^\dagger F_d\,|\psi_j\rangle &= \sum_{ab}\alpha_{ca}^*\alpha_{db}\,\langle\psi_i|E_a^\dagger E_b\,|\psi_j\rangle, && (83)\\
&= \sum_{ab}\alpha_{ca}^*\alpha_{db}\delta_{ij}\delta_{ab}A, && (84)\\
&= \left(\sum_a \alpha_{ca}^*\alpha_{da}A\right)\delta_{ij}, && (85)\\
&= A'_{cd}\delta_{ij}. && (*)
\end{aligned}$$

What this calculation tells us that, unless we pick a very clever basis for the errors, the states $E_a\,|\psi_i\rangle$ and $E_b\,|\psi_i\rangle$ will not be orthogonal for every i and every pair of E_a and E_b.

In the above, we have taken care of every possible case of possible confusion of the detector, and resolved them. We can now use Eq. (*) to write down the following theorem.

Theorem: A code $\mathcal{C}$ is a quantum error-correcting code for the set of errors $\{E_a\}$ iff

$$\langle\psi_i|E_a^\dagger E_b\,|\psi_j\rangle = A_{ab}\delta_{ij}, \tag{86}$$

for the orthogonal basis $\{|\psi_i\rangle\}_i$.

Quantum operations

We will start with density matrices and CPTP maps.

> Given the Hilbert space $\mathcal{H}$, let $L(\mathcal{H})$ be the set of automorphisms on this space. Then a **density operator** is such a linear operator $\rho \in L(\mathcal{H})$, such that ρ is

- Hermitian: $\rho = \rho^\dagger$,
- Normalized: $\mathrm{Tr}(\rho) = 1$,
- Positive semidefinite $\langle\psi|\rho\,|\psi\rangle \geq 0$ (written $\rho \geq 0$)).

A density operator is the most complete description of a quantum system. The most general evolution of density operators, including unitary operations, transient interactions with other systems and measurements, is described by quantum operations, more formally referred to as Completely-Positive Trace-Preserving (CPTP) maps.

> A **quantum operation** is a CPTP map $\mathcal{S} : L(\mathcal{H}) \to L(\mathcal{H})$ that maps density operators to density operators such that $\mathcal{S}$ is

- Positive: if $\rho \geq 0$, then $\mathcal{S}(\rho) \geq 0$,
- Completely-positive: if $\sigma \in L(\mathcal{H}^{AB})$ such that $\sigma \geq 0$ and $\mathrm{Tr}_B(\sigma) = \rho$, then $(I \otimes S)(\sigma) \geq 0$.
- Trace-preserving: $\mathrm{Tr}(\mathcal{S}(\rho)) = \mathrm{Tr}(\rho)$.

Any such operation has a representation, called the Kraus representation.

> For any quantum operation $\mathcal{S}$, there exist a non-unique set of operators $\{A_k\}_k$, where $A_k \in L(\mathcal{H})$, such that
>
> $$S(\rho) = \sum_k A_k \rho A_k^\dagger, \tag{87}$$
>
> and $\sum_k A_k^\dagger A_k = I$. This is called the **Kraus representation**.

Example: The dephasing channel...

Example: The depolarizing channel...

The noise channel model

We are now ready to define the noise that any qubits sent through the channel experience. Suppose, Alice sends n-qubits to Bob via the channel. Then, the noise model is the same single-qubit channel S applied to each qubit. So, to describe the noise the n-qubit quantum operation is

$$\mathcal{N} = \mathcal{S}^{\otimes n}. \tag{88}$$

Given the description of noise model an the n-qubit operator $\mathcal{N}$, we can identify a Krauss decomposition

$$\mathcal{N}(\rho) = \sum_i A_i \rho A_i^\dagger, \tag{89}$$

where there are some A_i with weight equal to n. However, recall that in the classical case if the probability of error p is small, then large errors are unlikely. There is a similar theorem for noise in the quantum systems.

A CPTP map on n-qubits that can be written with Krauss operators each of which is a sum of operators with weight less than t is a t**-qubit error**.

Theorem: Let $\mathcal{S}$ be a single-qubit quantum channel close to the identity $||\mathcal{S} - I||_\diamond < \epsilon$ for some ϵ. Then there exists a t-qubit error channel $\tilde{\mathcal{E}}$ such that

$$||\mathcal{S}^{\otimes n} - \tilde{\mathcal{E}}||_\diamond = \mathcal{O}\left(\binom{n}{t}\epsilon^t\right). \tag{90}$$

In simple language the above theorem states that if each of the n-qubits has only a very small chance of incurring an error, then one is very likely to observe errors on not very many qubits.

Suppose now that we have a t-qubit error channel $\mathcal{E}$, with a Krauss representation given by the operators $\{E_i\}_i$. From now on, we will refer to the E_i as errors, and $\{E_i\}_i$ as the set of errors. For instance, if S is the dephasing channel, and $n = 2$, then

$$\mathcal{E} = \{I, Z_1, Z_2, Z_1 Z_2\}, \tag{91}$$

where the I is included as the no-error possibility.

Our goal is to find quantum codes that are able to correct a given set of errors. Though often the research process proceeds in the opposite direction, and first a quantum code is defined and then its set of correctable errors is determined. We are now in the position to define a quantum code.

Pauli group

We are going to study the simplest class of quantum error-correcting codes, called the stabilizer codes. To understand the stabilizer codes, we need to build up some theory.

The Pauli matrices

Recall the Pauli matrices along with the identity matrix, $\{I, X, Y, Z\}$. These matrices have the property that multiplying them together in any way, gets us back to the same set of matrices up to a factor of ± 1 or $\pm i$.

For instance,

$$XY = iZ, \quad YZ = iX, \quad ZX = iX. \tag{92}$$

Task 1 (On paper)

Complete the following multiplication table.

$\times$	I	X	Y	Z
I	$II = I$	$IX = X$	$IY = Y$	$IZ = Z$
X	$XI = X$		$XY = iZ$	
Y	$YI = Y$			
Z	$ZI = Z$			

The first row and colum have been completed for you, as well as one additional element.

Commutation relations

There are one additional property of these matrices that we will use over and over again, and that is the commutation properties.

For $P, Q \in \{I, X, Y, Z\}$, we have either

- P and Q commute, i.e. $PQ = QP$, or
- P and Q anti-commute, i.e. $PQ = -QP$.

You can verify this by comparing the (i, j) entry in the table above with the (j, i) one. Either the entries are the same (ith and jth operators commute) or they differ by a minus sign (ith and jth operators anti-commute).

The Pauli group $\mathcal{P}_1$

What we have discovered that the 16 matrices,

$$\mathcal{P}_1 = \{\pm I, \pm iI, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\}, \tag{93}$$

form a group). This means the elements of $\mathcal{P}_1$ satisfy the group axioms:

- Multiplying any two elements of $\mathcal{P}_1$ gives us another element of $\mathcal{P}_1$. You have explicitly checked this above. This property is called closure (under multiplication) - equivalently, we say the set $\mathcal{P}_1$ is closed under multiplication.
- Given three elements P, Q, R in the $\mathcal{P}_1$, we have $(PQ)R = P(QR)$. This is true because the elements of $\mathcal{P}_1$ are matrices. This property is called associativity of multiplication.
- There is an identity element, I, in $\mathcal{P}_1$.
- For every element in $\mathcal{P}_1$, its inverse is also inside $\mathcal{P}_1$. For example, the inverse of X is X itself. The inverse of iX is $-iX$.

Question: Determine the inverse of every element in $\mathcal{P}_1$.

Note that $\mathcal{P}_1$ has 16 elements here, because there are 4 Pauli matrices and 4 possible phase factors ($\pm 1, \pm i$).

The Pauli group $\mathcal{P}_2$

Now, consider the tensor product of two Pauli matrices, meaning elements such as $X \otimes Z$ or $Y \otimes Z$. You are already familiar with the multiplication of such elements. Given $P = P^{(0)} \otimes P^{(1)}$ and $Q = Q^{(0)} \otimes Q^{(1)}$ in $\mathcal{P}_2$,

$$PQ = (P^{(0)} \otimes P^{(1)})(Q^{(0)} \otimes Q^{(1)}) = P^{(0)}Q^{(0)} \otimes P^{(1)}Q^{(1)}. \tag{94}$$

Example: $(X \otimes Z)(Y \otimes Z) = (iZ) \otimes I = i(Z \otimes I)$.

Suppose, we collect all such elements into a set. How many such elements are there? In $P^{(0)} \otimes P^{(1)}$, the matrix $P^{(0)}$ can be one of four Pauli matrices, and so can $P^{(1)}$. Additionally, we have four possible phase factors. In total, we will have $4^{2+1} = 64$ elements in a set we will call $\mathcal{P}_2$.

Again, we go over the four axioms of groups, and make sure $\mathcal{P}_2$ satisfies them.

- We have confirmed that multipling these 64 elements together in any way gives us one of the 64 elements.
- Multiplication continues to be associative.
- The element $I \otimes I$ is the identity element.
- We can easily construct the inverse of any element $P^{(0)} \otimes P^{(1)}$ as ${P^{(0)}}^{-1} \otimes {P^{(1)}}^{-1}$, and it will be one among the 64 elements.

Question: Determine the inverse of $i(X \otimes Y)$.

The Pauli group $\mathcal{P}_n$

It should be clear now, how to extend the above analysis to tensor products of n Pauli matrices, $P^{(0)} \otimes P^{(1)} \otimes \cdots \otimes P^{(n-1)}$. By the same arguments as above, there will be 4^{n+1} elements in $\mathcal{P}_n$.

All our discussion of the stabilizer codes will occur in the context of the Pauli group.

Subgroups of the Pauli group

A subgroup is a subset of the elements of the group that itself form a group. Meaning the subset satisfies all four of the axioms of the group.

Example: Consider the $\{I, X\} \subset \mathcal{P}_1$. We can check all four axioms.

- $IX = X$, $I^2 = I$, $X^2 = I$ are all in the subset, so closure of multiplication is satified.
- Associativity continues to hold.
- The identity is in the subset.
- The inverse of I is I and the inverse of X is X. So all elements have their inverse in the subset as well.

Hence, this subset is a subgroup of $\mathcal{P}_1$.

Question: Does $\{I, iX\} \subset \mathcal{P}_1$ form a subgroup of $\mathcal{P}_1$?

Example: Consider $\mathcal{P}_3$, which has elements of the form $P = P^{(0)} \otimes P^{(1)} \otimes P^{(2)}$. This group has a subgroup, which is defined as follow.

Let $P^{(1)} = I$, so that $Q = P^{(0)} \otimes I \otimes P^{(2)}$, and consider all such elements in $\mathcal{P}_3$, i.e., we have the subset

$$\mathcal{H} = \{Q \in \mathcal{P}_3 \text{ such that } Q = P^{(0)} \otimes I \otimes P^{(2)}\}. \tag{95}$$

To check that $\mathcal{H}$ is indeed a subgroup, we check all four properties.

- Closure under multiplication. Its quite clear that multiplying two elements of the form Q will yield another element of the form Q, and we included all elements of this form in $\mathcal{H}$, so this property is satisfied.
- Associativity continues to hold.
- The identity $I \otimes I \otimes I$ is of the form Q, so it is in $\mathcal{H}$.
- The inverse of Q is just $P^{(0)^{-1}} \otimes I \otimes P^{(2)^{-1}}$, which is still in the form of Q. So all elements of $\mathcal{H}$ also have their inverse inside $\mathcal{H}$.

Therefore $\mathcal{H}$ is a subgroup of $\mathcal{P}_3$.

Question: How many elements are there in G?

For finite-sized groups, such as the Pauli group, we don't actually have to check all four axioms.

Theorem: Let $\mathcal{H}$ be a nonempty finite subset of a group $\mathcal{G}$. Then $\mathcal{H}$ is a subgroup of $\mathcal{G}$ if $\mathcal{H}$ is closed under multiplication.

So, we only have to check the closure axiom, and if it holds so will the other three axioms. This makes the task of checking if a subset is a subgroup much easier.

Question: Consider the set

$$G = \{I \otimes I \otimes I, X \otimes I \otimes I, I \otimes Z \otimes I, X \otimes Z \otimes I\}. \tag{96}$$

Does this form a subgroup of $\mathcal{P}_3$?

Stabilizer states

The elements of the Pauli group have another important property that relates to quantum states, which we will study now. In the next section, we will use the notions introduced here to construct stabilizer groups.

We will need to recall the eigenvectors and eigenvalues of X, Y, Z are

$$\begin{aligned} X\,|+\rangle &= |+\rangle, & X\,|-\rangle &= -\,|-\rangle, \\ Y\,|+i\rangle &= |+i\rangle & Y\,|-i\rangle &= -\,|-i\rangle, \\ Z\,|0\rangle &= |0\rangle, & Z\,|1\rangle &= -\,|1\rangle. \end{aligned}$$

Stabilizer states

In the equations above, the ones on the left are +1 eigenstates. This means acting on the state by the corresponding operator leaves the state unchanged (even accounting for a phase). We are going to give a special name to such states.

> Given $P \in \mathcal{P}_n$, a state $|\psi\rangle$ such that $P\,|\psi\rangle = |\psi\rangle$, is called a **stabilizer state** of P.

Let's look at some examples for $n > 1$.

Example: Consider, the Pauli operator $X \otimes Z$. From the equations above, it is quite easy to see that

$$(X \otimes Z)\,|+\rangle \otimes |0\rangle = |+\rangle \otimes |0\rangle. \tag{97}$$

The tensor product of the $+1$ eigenvectors of the individual operators gives a $+1$ eigenvector of the tensor-product operator. But, wait, there is more.

$$(X \otimes Z)\left|-\right\rangle \otimes \left|1\right\rangle = (-1)^2 \left|-\right\rangle \otimes \left|1\right\rangle = \left|-\right\rangle \otimes \left|1\right\rangle. \tag{98}$$

In this way, we can see that the operator $X \otimes Z$ has two $+1$ eigenvectors (it also has two -1 eigenvectors; why?).

Of course, any state that is a linear combination of $\left|+\right\rangle \otimes \left|0\right\rangle$ and $\left|-\right\rangle \otimes \left|1\right\rangle$ is also a $+1$ eigenstate of $X \otimes Z$,

$$\begin{aligned}(X \otimes Z)(\alpha \left|+\right\rangle \otimes \left|0\right\rangle + \beta \left|-\right\rangle \otimes \left|1\right\rangle) &= \alpha(X \otimes Z)\left|+\right\rangle \otimes \left|0\right\rangle + \beta(X \otimes Z)\left|-\right\rangle \otimes \left|1\right\rangle, &(99)\\ &= \alpha \left|+\right\rangle \otimes \left|0\right\rangle + \beta \left|-\right\rangle \otimes \left|1\right\rangle. &(100)\end{aligned}$$

Similarly, for any Pauli operator P in $\mathcal{P}_n$, we can construct states that are the $+1$ eigenvectors of the operator.

Example: Here is another example,

$$\begin{aligned}(X \otimes I \otimes X)\left|+\right\rangle \otimes \left|0\right\rangle \otimes \left|+\right\rangle &= \left|+\right\rangle \otimes \left|0\right\rangle \otimes \left|+\right\rangle, &(101)\\ (X \otimes I \otimes X)\left|-\right\rangle \otimes \left|0\right\rangle \otimes \left|-\right\rangle &= \left|-\right\rangle \otimes \left|0\right\rangle \otimes \left|-\right\rangle. &(102)\end{aligned}$$

I is special because every state is a $+1$ eigenvector of I. So, we could also have written any state in middle, such as

$$\begin{aligned}(X \otimes I \otimes X)\left|+\right\rangle \otimes \left|-\right\rangle \otimes \left|+\right\rangle &= \left|+\right\rangle \otimes \left|-\right\rangle \otimes \left|+\right\rangle, &(103)\\ (X \otimes I \otimes X)\left|-\right\rangle \otimes \left|-\right\rangle \otimes \left|-\right\rangle &= \left|-\right\rangle \otimes \left|-\right\rangle \otimes \left|-\right\rangle. &(104)\end{aligned}$$

etc. From this we infer that $X \otimes I \otimes X$ stabilizes all states that are a linear combination of

$$\{\left|+\right\rangle \otimes \left|0\right\rangle \otimes \left|+\right\rangle, \left|-\right\rangle \otimes \left|0\right\rangle \otimes \left|-\right\rangle, \left|+\right\rangle \otimes \left|1\right\rangle \otimes \left|+\right\rangle, \left|-\right\rangle \otimes \left|1\right\rangle \otimes \left|-\right\rangle\}. \tag{105}$$

Task 1 (On paper)

Determine the basis of the stabilizer states of the following operators.

- $X \otimes X \otimes X$
- $Z \otimes Z \otimes Z$
- $Z \otimes I \otimes X$
- $Z \otimes X \otimes X \otimes Z$

Do you see a pattern in the number of states and the size of the operator?

Stabilizer states of multiple operators

We will now show that it is possible to construct states that are the simultaneously stabilized by multiple Pauli operators. Let's start with some examples.

Example: Consider, the operators $P_1 = Z \otimes Z \otimes I$ and $P_2 = Z \otimes I \otimes Z$, both in $\mathcal{P}_3$.

- $Z \otimes Z \otimes I$ stabilizes four orthogonal states $\{\left|000\right\rangle, \left|001\right\rangle, \left|110\right\rangle, \left|111\right\rangle\}$.
- $Z \otimes I \otimes Z$ stabilizes four orthogonal states $\{\left|000\right\rangle, \left|010\right\rangle, \left|101\right\rangle, \left|111\right\rangle\}$.

The intersection of these sets is $\{\left|000\right\rangle, \left|111\right\rangle\}$. This means that any state that is a linear combination of these two states will be stabilized by both P_1 and P_2.

Remember, how $\{\left|000\right\rangle, \left|111\right\rangle\}$ were the basis states of the quantum repetition code for bit-flip codes. It turns out that code is a stabilizer code. We will soon show this, if you have some patience.

Task 2 (On paper)

Determine states that are simultaneously stabilized by $P_1 = X \otimes X \otimes I = X_0X_1$ and $P_2 = X \otimes I \otimes X = X_0X_2$. (The subscript based notation will ensure you don't injure your wrist!)

Have you seen them before?

Task 3 (On paper)

It is not necessary that two operators have simultaneously stabilized states.

- Do $P_1 = Z_0 Z_1$ and $P_2 = X_0 X_1$ have any simultaneously stabilized states?
- What about $P_1 = Z_0 Z_1$ and $P_2 = X_0 X_2$?

Stabilizer groups

Among the many subgroups of the Pauli group $\mathcal{P}_n$, there is a particular type of subgroups, called the stabilizer group S, which satisfy the following properties, in addition to the group axioms:

- All pairs of elements in S commute, i.e. for every $P, Q \in S$, we have $PQ = QP$.
- $-I = -I_0 \otimes I_1 \otimes \cdots \otimes I_{n-1}$ is not a part of S.

We will see later on why these properties are useful, but for now, let's try to create some examples of these groups so we can build some intuition.

Example: Let

$$\begin{aligned} P_0 &= I \otimes I \otimes I, && (106) \\ P_1 &= Z \otimes Z \otimes I, && (107) \\ P_2 &= Z \otimes I \otimes Z, && (108) \\ P_3 &= I \otimes Z \otimes Z, && (109) \end{aligned}$$

and consider the subset,

$$S = \{P_0, P_1, P_2, P_3\}, \tag{110}$$

of $\mathcal{P}_3$.

Task 1 (On paper)

Produce the multiplication table of the subgroup S defined in example above, to show that S is closed under multiplication and hence a subgroup of $\mathcal{P}_3$. (We will often just write I instead of $P_0 = I \otimes I \otimes I$.)

$\times$	I	P_1	P_2	P_3
I				
P_1				
P_2				
P_3				

Further, we claim that S is a stabilizer group.

- From the multiplication table you just created, you can also read off the commutation relations. You can verify that all elements of S commute.
- Moreover $-I$ is not part of S, so the second property is trivially satisfied.

Task 2 (On paper)

One way of constructing subgroups is to start with a subset of elements of the group. We multiply these elements together in every way possible. Whenever we create an element that is not part of the subset, we add it to the subset. Eventually, we will have exhausted all possible multiplications and we will have a subset closed under multiplication.

Take $P_1 = X \otimes X \otimes I$ and $P_2 = I \otimes X \otimes X$.

- Multiply them together in every combination, till you find a set of elements that is closed.

- Write down the multiplication table of this group, and verify that this is a stabilizer group.

Generators of a group

This is an opportune moment to introduce the notion of the generators of a group. The generators of a group can be used to construct all other elements of the group.

Example: In the first example above, every element in the group is some product of P_1 and P_2,

$$\begin{aligned} I \otimes I \otimes I &= P_1^2 = P_2^2, && (111)\\ Z \otimes Z \otimes I &= P_1, && (112)\\ I \otimes Z \otimes Z &= P_2, && (113)\\ Z \otimes I \otimes Z &= P_1 P_2. && (114) \end{aligned}$$

Therefore, P_1 and P_2 are called the generators of the group S.

> The **generators** of a group is a subset of elements of the group, and that can be used to construct all elements of the group. The generators are not unique.

Example: In Task 2, you created a group generated by $X \otimes X \otimes I$ and $I \otimes X \otimes X$.

Question: Show that the stabilizer group in the first example, can be generated by P_1 and P_3 as well, i.e. show how to write every element of the group in terms of these two elements.

The advantage of identifying the generators for stabilizer groups is that they reduce the storage space for specifying a group. This is because of two properties.

- Property of stabilizer groups: All elements commute.
- Property of all Pauli operators: P^2 is either I or $-I$.

Let S be a stabilizer group, with generators $\{g_i\}_{i=0}^{m-1}$. The first property means that we can write any element h in S as the product.

$$h = g_0^{j_0} g_1^{j_1} \cdots g_{m-1}^{j_{m-1}}, \tag{115}$$

where we don't care about the order of the generators in the product. The power of j_i on each generator tells us how many times it appears in the product in order to create h.

The second property means that j_i can only be 0 or 1.

This means that any element h is completely specified by the bitstring $(j_0, j_1, \ldots, j_{m-1})$. Recognizing that there are 2^m such strings, the following lemma immediately follows.

Lemma: If S has m generators, then it has 2^m elements.

To work with stabilizer codes, we will be depending highly on generators of the stabilizer groups.

Task 3 (On paper)

Determine the bitstring representation of all elements in S for

- Example 1
- Task 2

States stabilized by stabilizer groups

In stabilizer states, we saw that multiple operators can stabilize the same state.

Since, stabilizer groups are just a set of operators, a very natural question to ask is, what states are stabilized by these operators?

Example: Above, we created the stabilizer group

$$S = \{I \otimes I \otimes I, Z \otimes Z \otimes I, Z \otimes I \otimes Z, I \otimes Z \otimes Z\}. \tag{116}$$

Let's try to construct the state simultaneously stabilized by all these operators.

- The identity stabilizes all states.
- We already know that $Z \otimes Z \otimes I, I \otimes Z \otimes Z$ jointly stabilize $\{|000\rangle, |111\rangle\}$.
- It's easy to discover that $Z \otimes I \otimes Z$ has stabilizer states $\{|000\rangle, |010\rangle, |101\rangle, |111\rangle\}$.

Hence, the common intersection is $\{|000\rangle, |111\rangle\}$. Therefore,

> The group S stabilizes the states $\{|000\rangle, |111\rangle\}$, and any of their linear combinations.

We note that $\{|000\rangle, |111\rangle\}$ is the basis of the repetition code. The connection between S and repetition code will be fleshed out in the next notebook.

Generators are sufficient to determine the stabilizer states of a group

Actually, we are doing too much work in determining the stabilizer states. Each element of S is specified by a product of two generators P_1 and P_2. Note the following two self-evident properties.

- If P_1 and P_2 both stabilize a state $|\psi\rangle$, then any $h \in S$ (which is just some product of P_1 and P_2) will also stabilize $|\psi\rangle$.

- If P_1 or P_2 don't stabilize a state, then that state cannot be included in our final set of states.

Taken together this means that we only need to determine states that are stablized by the all generators to determine the states stabilized by the stabilizer group.

Lemma: The basis of states stabilized by a stabilizer group S, is the intersection of states stabilized by the generators of S.

Stabilizer codes

We now have all the ingredients to define stabilizer codes. To define a error-correcting code we need four ingredients

- Encoding: A method to encode each message into a codewords. For quantum codes, we need to map each basis element of the message space to a basis element of the codespace.
- Syndrome measurements: A method to do syndrome measurements on the corrupted codewords, to extract information about the error.
- Decoding: A method to transfrom the corrected codewords back into a message.
- Logical gates: Methods to implement logical gates directly on an encoded qubit.

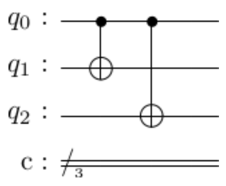
Stabilizer codes are defined with respect to a stabilizer group S. We label the stabilizer code based on S as $C(S)$. Then, the ingredients for $C(S)$ are

- Encoding: The simultaneous stabilizer states of the stabilizer group S define the basis elements of the codespace. An encoding circuit can be systematically constructed using the generators of S; a procedure we will show later.
- Syndrome measurements: All generators of S are measured one by one. The results of the measurement indicate which error has occured.
- Decoding: Decoding is also based on S.
- Logical gates: These can also be constructed from S.

Repetition code for bit-flips

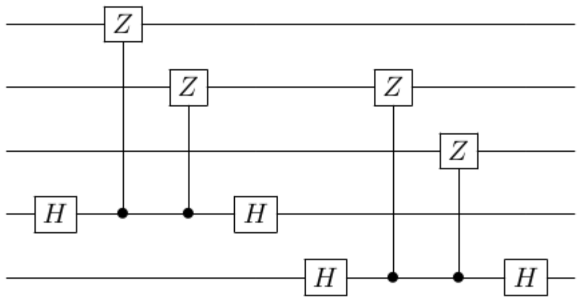
Let's see how all this applies to the repetition code. The repetition code is based on the group S generated by $P_1 = Z \otimes Z \otimes I$ and $P_2 = I \otimes Z \otimes Z$.

- Encoding: We saw in stabilizer groups that this group simultaneously stabilizes the states $\{|000\rangle, |111\rangle\}$. In quantum repetition code for bit-flips we presented an ad-hoc encoding circuit for this code as



We will see later on how to construct this from S.

- Syndrome measurements: We saw in syndrome measurements for stabilizer codes this is achieved using the following circuit.



Here we are measuring the two generators P_1 and P_2.

- Decoding: In quantum repetition code for bit-flips we noted that the encoding circuit for the repetition code is self-inverse, and can also be used for decoding. We will present a different procedure later on.

Task 1 (On paper)

Show that the repetition code for phase-flips is also a stabilizer code.

Which errors are correctable by a stabilizer code

Up till now, we have discussed three codes, and discovered that they all correct different sets of errors

- The repetition code for bit-flips corrects all single-qubit X errors.
- The repetition code for phase-flips corrects all single-qubit Z errors.
- The Shor code corrects all single-qubit errors.

Is there a rule that tells us which errors are correctable by the code? Indeed there is.

Theorem: Let S be a stabilizer group, with generators $\{g_i\}$, and let $C(S)$ be the associated stabilizer code. Then $C(S)$ detects an error $E \in \mathcal{P}_n$ if it anti-commutes with at least one generator g_k of S.

Let's explore why this theorem is correct. To detect errors, we measure all the generators of the stabilizer group. Let's concentrate on the measurement of just one generator g_k. There are three possibilities.

1. No error has occured and the data qubits are in state $\left|\bar{\psi}\right\rangle$.
2. Error E has occured, and the data qubits are in state $E\left|\bar{\psi}\right\rangle$. The two sub-possibilities are

- (a) E commutes with g_k.
- (b) E anti-commutes with g_k.

Let's analyze each possibility one by one. We will need to recall from syndrome measurements, an alternate view that if we measure g_k on some state $|\phi\rangle$, then at the end of the measurement circuit, the qubits will be in state

$$|\Phi\rangle = (|\phi\rangle + g_k|\phi\rangle)|0\rangle + (|\phi\rangle - g_k|\phi\rangle)|1\rangle, \tag{117}$$

where as per the possibilities above either $|\phi\rangle$ is the uncorrupted state $\left|\bar{\psi}\right\rangle$ or the corrupted state $E\left|\bar{\psi}\right\rangle\}$.

Possibility 1: no error.

If there is no error, then $|\phi\rangle = \left|\bar{\psi}\right\rangle$ and so $|\Phi\rangle = \left|\bar{\psi}\right\rangle|0\rangle$. Consequently measuring the ancilla will yield 0.

Possibility 2 (a) E commutes with g_k

Now, $|\phi\rangle = E\left|\bar{\psi}\right\rangle$. First note that

$$g|\phi\rangle = g(E\left|\bar{\psi}\right\rangle) = Eg\left|\bar{\psi}\right\rangle = E\left|\bar{\psi}\right\rangle = |\phi\rangle. \tag{118}$$

Hence, $|\Phi\rangle = |\phi\rangle|0\rangle$. Again, we discover that the ancilla will measure 0.

Possibility 2 (b) E anti-commutes with g_k

Now, $|\phi\rangle = E\left|\bar{\psi}\right\rangle$. First note that

$$g|\phi\rangle = g(E\left|\bar{\psi}\right\rangle) = -Eg\left|\bar{\psi}\right\rangle = -(E\left|\bar{\psi}\right\rangle) = -|\phi\rangle. \tag{119}$$

Hence, $|\Phi\rangle = |\phi\rangle|1\rangle$. This time, the ancilla will measure 1.

What we have discovered is that if and only if E and g_k anti-commute does the ancilla trigger. Therefore, it is quite easy to see that if at least one of E and $\{g_i\}$ anti-commute, the code will detect the error.

A fourth possibility

Note, that any code is also immune to errors that don't change the state. For instance, for the Shor code, we saw that errors like Z_0Z_1 leave $\left|\bar{\psi}\right\rangle$ unchanged.

Task 2 (On paper)

Determine the commutation relations of single-qubit X errors with the generators of the repetition code for bit-flips. Put 0 in the table if they commute, and 1 if they anti-commute.

Error \ Generator	$Z\otimes Z\otimes I$	$I\otimes Z\otimes Z$
$I\otimes I\otimes I$		
$X\otimes I\otimes I$		
$I\otimes X\otimes I$		
$I\otimes I\otimes X$		

Compare with the syndrome table that we presented in quantum repetition code for bit-flips.

Task 3 (On paper)

Determine the commutation relations of single-qubit Z errors with the generators of the repetition code for phase-flips. Put 0 in the table if they commute, and 1 if they anti-commute.

Error \ Generator	$X\otimes X\otimes I$	$I\otimes X\otimes X$
$I\otimes I\otimes I$		
$Z\otimes I\otimes I$		
$I\otimes Z\otimes I$		
$I\otimes I\otimes Z$		

Compare with the syndrome table that we presented in quantum repetition code for phase-flips

One important lesson you can take away from the above is that to detect X type errors, we need Z type generators. And to detect Z-type errors, we need X-type generators. This observation will be useful, when we construct more stabilizer codes.

Size of a stabilizer code

Recall the sizes of the codes we have seen

Code	No. of logical bits/qubits	No. of physical bits/qubits
classical repetition code	1	3
Hamming code	4	7
quantum repetition codes	1	3
Shor code	1	9

If we define a stabilizer code $C(S)$ using some stabilizer group S, how many logical qubits are encoded into how many physical qubits? For this the rules are as follows.

- If S is a subgroup of $\mathcal{P}_n$, then $C(S)$ has n physical qubits.
- If S has m generators, then $C(S)$ has $k = n - m$ logical qubits.

These three quantities n, m, k will be very important in our further analysis.

> A quantum stabilizer code with distance d is labeled as $[[n, k, d]]$. The double brackets remind us that it is a quantum code.

Distance of a stabilizer code

The distance of the code determines how many errors it can correct. A code with distance d can correct $\lfloor (d-1)/2 \rfloor$ errors (the $\lfloor * \rfloor$ indicates the floor function).

The simplest way to find the distance is as follows.

> The **weight** of a Pauli operator is the number of non-identity operators in it.

Example: $X \otimes I \otimes Z \otimes I$ has a weight of two.

> The **distance** of stabilizer code $C(S)$ is the minimum weight operator in $\mathcal{P}_n$ that is not in S but still commutes with all members of S.

Example: For the repetition code for bit-flips S was generated by $P_1 = Z \otimes Z \otimes I$ and $P_2 = I \otimes Z \otimes Z$. Now, the operator $Q = Z \otimes I \otimes I$ commutes with both P_1 and P_2 but is not in S. Q has weight 1, so the code has distance $d = 1$.

Since $\lfloor (1-1)/2 \rfloor = 0$, this indicates that there are one qubit errors that the repetition code cannot correct. Obviously, these are the single-qubit phase errors.

Vector form of Pauli operators and the generator matrix

It is getting quite tedious to write down multi-qubit Pauli operators like $X \otimes X \otimes I$. Here we are going to discuss a compact form for Pauli operators, that will make it very easy to do computations.

Vector form of Pauli operators

We are going to map operators in the Pauli group $\mathcal{P}_n$ to binary vectors of length $2n$.

The elements of $\mathcal{P}_1$ are

$$\begin{aligned} I &\equiv (0|0), & (120)\\ X &\equiv (1|0), & (121)\\ Y &\equiv (1|1), & (122)\\ Z &\equiv (0|1). & (123) \end{aligned}$$

For larger n we have a similar formula of operators getting mapped to the vector $v = (a|b)$ where

- a is the "X part", and is of length n, and
- b is the "Z part", and is also of length n.

Let $P = \otimes_i P_i$, where $P_i \in \{I_i, X_i, Y_i, Z_i\}$. Then,

$$\begin{aligned} a_i = 0, b_i = 0 \text{ if } P_i = I_i, && (124)\\ a_i = 1, b_i = 0 \text{ if } P_i = X_i, && (125)\\ a_i = 1, b_i = 1 \text{ if } P_i = Y_i, && (126)\\ a_i = 0, b_i = 1 \text{ if } P_i = Z_i. && (127) \end{aligned}$$

For example if $n = 4$, then $X_0 I_1 Z_2 Y_3$ is associated with the vector $v = (1001|0011)$.

Important to note in this formulation is that we will completely ignore the phase of the operators. Meaning if $P \equiv v$, then so are $-P \equiv v$ and $\pm iP \equiv v$.

Task 1 (On paper)

Determine the vector form of the following operators

- $X_0 Z_0 X_1 Z_1$
- $Y_0 Y_1 Y_2$
- $Z_0 Z_1 X_2$

Multiplying operators

One of the immediate benefits of the binary vector form is that multiplying two Paulis corresponds to adding their vectors mod 2 (up to an overall phase).

For the simplest example, note that $X = (1|0)$ and $Z = (0|1)$ and their product is $XZ = (1|0) + (0|1) = (1|1) = Y$. Quite ingenius! Please note that we will ignore the overall phase here as $XZ = -iY$, but as you delve more into error-correcting codes you will realize that such phases are not very important.

Question: Use the binary vector form to compute XX.

Commutation relations

One of the best advantages of writing Pauli operators as binary vectors is that commutation relations become very easy to compute.

First, we define the symplectic inner product between binary vectors of length $2n$

> Let $P_1 = (a|b)$ and $P_2 = (c|d)$. Then, the **symplectic inner product** between P_1 and P_2 is
>
> $$P_1 \cdot P_2 = a \cdot d + b \cdot c \mod 2, \tag{128}$$
>
> where on the right hand side, the $\cdot$ is the regular dot product between vectors. Be very careful about how we are multiplying the X-part of P_1 with the Z-part of P_2 and vice versa.

Example: The symplectic inner product of $(10|01)$ and $(01|00)$ is

$$(10|01) \cdot (01|00) = (10) \cdot (00) + (01) \cdot (01) = 0 + 1 = 1. \tag{129}$$

Task 2 (On paper)

Let $P_1 = (101|110)$ and $P_2 = (011|111)$. Determine $P_1 \cdot P_2$.

Task 3

Complete the function `symplectic_inner_product` that computes the symplectic product of two input vectors.

- Parameters:
  `P1`: a `list`, guaranteed to only contain 0 or 1, and length divisible by 2.
  `P2`: a `list`, guaranteed to only contain 0 or 1, and length divisible by 2.
- Returns:
  An `int` that is symplectic inner product of `P1` and `P2`

```python
def symplectic_inner_product(P1, P2):
    pass
```

Now we state the following theorem about the commutation relations of Pauli operators.

Lemma: Let $P_1 = (a|b)$ and $P_2 = (c|d)$ be two Pauli operators. Then,

- P_1 and P_2 commute if and only if $P_1 \cdot P_2 = 0$,
- P_1 and P_2 anti-commute if and only if $P_1 \cdot P_2 = 1$.

Question: Use the commutation relations of I_i, X_i, Y_i, Z_i to prove this theorem.

Task 4

In Task 1 in stabilizer codes we computed everything by hand. Now use the `symplectic_inner_product` function to recompute the table with code.

Generator matrix

To compactly represent the stabilizer group, and its associated code, we usually write the generators of the stabilizer code as rows of a matrix. This matrix is called sometimes called the generator matrix (because it contains the generators of the code) or the check-matrix (because it is actually analogous to the classical parity-check matrix rather than the classical generator matrix). This is unfortunate and confusing terminology.

Example: For the repetition code for phase flips, the generators are

$$X \otimes X \otimes I = (110|000), \tag{130}$$
$$I \otimes X \otimes X = (011|000). \tag{131}$$

Hence, the generator matrix is

$$G = \left(\begin{array}{ccc|ccc} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right). \tag{132}$$

Task 5 (On paper)

Write down the generator matrix for the repetition code for bit-flips.

Some common codes

We should know some more quantum error-correcting codes, in order to deepen our understanding. We will define them here, via their generator matrix.

In stac, you can create any code for which you know the generator matrix. An example for the repetition code for phase flips is given below.

```
import stac
import numpy as np

# generator matrix
gm = np.array([[1, 1, 0, 0, 0, 0],
               [0, 1, 1, 0, 0, 0]])

cd_rep = stac.Code(gm)
# n
print(f'n = {cd_rep.num_data_qubits}')
# k
print(f'k = {cd_rep.num_logical_qubits}')
# m = n-k
print(f'm = {cd_rep.num_generators}')

stac.print_paulis(cd_rep.generator_matrix)
```

```
n = 3
k = 1
m = 2
```

XXI
IXX

There are several codes whose generator matrix is stored inside stac. These are given below.

The five qubit code, $[[5, 1, 3]]$

In this code, one qubit is encoded into the state of $n = 5$ qubits. This code is unique because it is the smallest code (i.e. smallest n), which can correct any single-qubit error.

The code has generators as follows.

```
cd513 = stac.CommonCodes.generate_code('[[5,1,3]]')
stac.print_matrix(cd513.generator_matrix, augmented=True)
```

$$\left(\begin{array}{ccccc|ccccc} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right)$$

```
stac.print_paulis(cd513.generator_matrix)
```

$XZZXI$
$IXZZX$
$XIXZZ$
$ZXIXZ$

Another feature of this code is that it has generators that have both X and Z operators in them. This is unlike any of the previous codes you have seen.

```
# stac has information stored about codes
print(cd513.num_generators)
print(cd513.num_data_qubits)
print(cd513.num_logical_qubits)
```

```
4
5
1
```

The Steane code, $[[7, 1, 3]]$

The Steane code is often studied because it has a number of nice properties. It encoded one qubit into seven qubits and is able to correct any single qubit error. Interestingly, it is made out of the classical Hamming code, that we saw earlier.

You will see in the generator matrix below that the top left submatrix (which defines three X-type generators) is the matrix of the classical Hamming code. Similarly, the bottom right (which defines three Z-type generators is also the same Hamming code matrix.

```
cd713 = stac.CommonCodes.generate_code('[[7,1,3]]')
stac.print_matrix(cd713.generator_matrix, augmented=True)
```

$$\left(\begin{array}{ccccccc|ccccccc}1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1\end{array}\right)$$

```
stac.print_paulis(cd713.generator_matrix)
```

$XXXXIII$
$XXIIXXI$
$XIXIXIX$
$ZZZZIII$
$ZZIIZZI$
$ZIZIZIZ$

The $[[4, 2, 2]]$ code

Up till now, you have seen codes that encode one qubit into many. This is unlike, say the classical Hamming code, which encoded three qubits into seven. The

$$[4, 2, 2]]$$

is encodes two qubits into four qubits. However, this means that it can't correct all single-qubit errors.

```
cd422 = stac.CommonCodes.generate_code('[[4,2,2]]')
stac.print_matrix(cd422.generator_matrix, augmented=True)
```

$$\left(\begin{array}{cccc|cccc}1 & 0 & 0 & 1 & 0 & 1 & 1 & 0\\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1\end{array}\right)$$

```
stac.print_paulis(cd422.generator_matrix)
```

$XZZX$
$YXXY$

Now you are seeing a code which has Y operators within its generators.

The $[[8, 3, 3]]$ code

This code encodes three qubits into the state of eight qubits and can correct any single-qubit error.

```
cd833 = stac.CommonCodes.generate_code('[[8,3,3]]')
stac.print_matrix(cd833.generator_matrix, augmented=True)
```

$$\left(\begin{array}{cccccccc|cccccccc}1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1\\0&1&0&1&1&0&1&0&0&0&0&0&1&1&1&1\\0&1&0&1&0&1&0&1&0&0&1&1&0&0&1&1\\0&1&1&0&1&0&0&1&0&1&0&1&0&1&0&1\end{array}\right)$$

```
stac.print_paulis(cd833.generator_matrix)
```

$XXXXXXXX$
$ZZZZZZZZ$
$IXIXYZYZ$
$IXZYIXZY$
$IYXZXZIY$

Encoding circuits for stabilizer codes

Here, we present an algorithm to construct the encoding circuit for stabilizer codes. We will use the Steane code as the guiding example.

Gottesman's method to construct logical zero state of stabilizer codes

There is a very systematic method for creating the logical zero state using the stabilizer generator matrix. The method is a three part process.

1. Bring the generator matrix to standard form.
2. Construct logical operators of the code.
3. Use Gottesman's algorithm to determine the sequence of gates in the encoding circuit.

Let's load up the Steane code.

```
import stac
```

```
cd = stac.CommonCodes.generate_code('[[7,1,3]]')
stac.print_matrix(cd.generator_matrix, augmented=True)
```

$$\left(\begin{array}{ccccccc|ccccccc}1&1&1&1&0&0&0&0&0&0&0&0&0&0\\1&1&0&0&1&1&0&0&0&0&0&0&0&0\\1&0&1&0&1&0&1&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1&1&1&1&0&0&0\\0&0&0&0&0&0&0&1&1&0&0&1&1&0\\0&0&0&0&0&0&0&1&0&1&0&1&0&1\end{array}\right)$$

1. Bring the generator matrix to standard form

The matrix above is the generator matrix G of the Steane code. It has $m = 6$ generators/row, and $2n = 14$ columns, so $n = 7$ physical qubits. And it encodes $k = n - m = 1$ logical qubit.

We will now transform it to the standard form. What does this mean? Note that each row of H represents a generator of the stabilizer group associated with the Steane code. We can manipulate the generators and qubits in the following way.

- We know from group theory that we can replace one of the generators g_i with $g_i g_j$ where g_j is another generator, and we will have obtained a new generating set for the same group. Multiplying generators corresponds to adding rows of G (why?).
- We can also reindex the generators, which corresponds to permuting the rows G.
- We can also reindex the physical qubits, which corresponds to simultaneously permuting the columns on both sides of G.

All of these operations on the generators don't change the code, only create a new representation of it. The corresponding operations on G are simply those that we do when doing Gaussian reduction on a matrix. Recall that any matrix can brought into a reduced row echelon form (RREF), which is unique. We will use this fact to create the standard from of G. The goal is to simplify the generators so they have as few Paulis in them as possible.

The standard form is obtained as follows. First bring the X part of G to RREF, remembering to permute columns on both sides. This reduces G to

$$G = \left(\begin{array}{cc|cc} I & A & B & C \\ 0 & 0 & D & E \end{array}\right). \tag{133}$$

Let r be the rank of the X part of G. Then, the blocks have

- r and $m-r$ rows respectively,
- r and $n-r$ columns respectively.

If we then bring E to RREF, due to the properties of the code, G will reduce to

$$G = \left(\begin{array}{ccc|ccc} I & A_1 & A_2 & B & C_1 & C_2 \\ 0 & 0 & 0 & D & I & E_2 \end{array}\right). \tag{134}$$

Here, the colums of the blocks are of size r, $m-r$ and k respectively.

This is the standard form of G. For the Steane code, it is as follows.

```
cd.construct_standard_form()
stac.print_matrix(cd.standard_generator_matrix, augmented=True)
```

$$\left(\begin{array}{ccccccc|ccccccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}\right)$$

Task 1

Identify A_1, A_2, B, C_1, C_2, D and E_2 for the matrix above.

2. Construct logical operators of the code

Logical operators are operators that act directly on the encoded state. For instance, the logical $\bar{X}$ acts as follows on the encoded basis states $\left|\bar{0}\right\rangle$ and $\left|\bar{1}\right\rangle$.

$$\bar{X}\left|\bar{0}\right\rangle = \left|\bar{1}\right\rangle, \tag{135}$$
$$\bar{X}\left|\bar{1}\right\rangle = \left|\bar{0}\right\rangle. \tag{136}$$

Similarly, the logical $\bar{Z}$ acts as follows.

$$\bar{Z}\left|\bar{0}\right\rangle = \left|\bar{0}\right\rangle, \tag{137}$$
$$\bar{Z}\left|\bar{1}\right\rangle = -\left|\bar{1}\right\rangle. \tag{138}$$

For a stabilizer code, Gottesman's presented a method to construct them. The claim is that, the logical $\bar{X}$s are the rows of the matrix

$$\bar{X} = \begin{pmatrix} 0 & E_2^T & I & | & (E_2^T C_1^T + C_2^T) & 0 & 0 \end{pmatrix}. \tag{139}$$

Remember that some codes encode more than one logical qubit, so if a code encodes k logical qubit, we will need k logical $\bar{X}$ s.

Similarly, the logical $\bar{Z}$s are

$$\bar{Z} = \begin{pmatrix} 0 & 0 & 0 & | & A_2^T & 0 & I \end{pmatrix}. \tag{140}$$

We can read off these from H, and for the Steane code, they are as follows.

```
cd.construct_logical_operators()
print("Logical X =", cd.logical_xs)
stac.print_paulis(cd.logical_xs)

print("\nLogical Z =", cd.logical_zs)
stac.print_paulis(cd.logical_zs)
```

```
Logical X = [[0 0 0 0 1 1 1 0 0 0 0 0 0 0]]
```

$IIIIXXX$

```
Logical Z = [[0 0 0 0 0 0 0 1 1 0 0 0 0 1]]
```

$ZZIIIIZ$

Note that the logical $\bar{X}$ and $\bar{Z}$ are not unique. For instance, for the Steane code, the operators

$$\bar{X} = X_0X_1X_2X_3X_4X_5X_6, \tag{141}$$
$$\bar{Z} = Z_0Z_1Z_2Z_3Z_4Z_5Z_6, \tag{142}$$

also act just as well.

Task 1

Use `stac` to construct the logical operators for the $[[5, 1, 3]]$ code.

```
#
```

3. Use Gottesman's algorithm to determine the sequence of gates in the encoding circuit

Now we are ready to construct the encoding circuit for the zero state. It's a very simple algorithm. Let there be n qubits, and let

- qubits 0 to $n - k - 1$ be the ancilla qubits in state $|0\rangle$, and
- qubits $n - k$ to $n - 1$ qubits (k total) be in the unknown state $|\psi\rangle$, which is to be encoded.

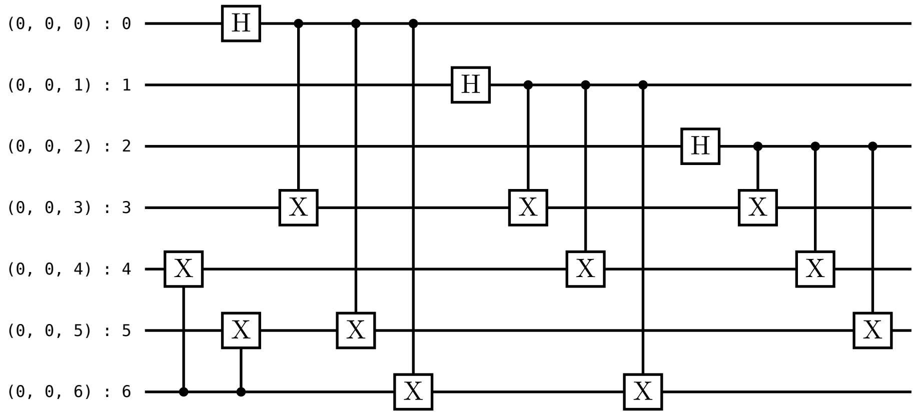
The algorithm is as follows.

```
encoding_circuit = stac.Circuit.simple(n)
for i in range(k):
    for j in range(r, n-k):
        if cd.logical_xs[i, j]:
            encoding_circuit.append("CX", n-k+i, j)

for i in range(r):
    encoding_circuit.append(["H", i])
    for j in range(n):
        if i == j:
            continue
        if cd.standard_generators_x[i, j] and cd.standard_generators_z[i, j]:
            encoding_circuit.append("CX", i, j)
            encoding_circuit.append("CZ", i, j)
        elif cd.standard_generators_x[i, j]:
            encoding_circuit.append("CX", i, j)
        elif cd.standard_generators_z[i, j]:
            encoding_circuit.append("CZ", i, j)
```

One subtlety here is that instead of using the complex Y gate, we instead let $Y = ZX$, which is a real matrix. So, the place where we have to apply CY, we apply CX followed by CZ.

Here, we will let $|\psi\rangle = |0\rangle$. Executing this algorithm, we get the circuit as follows.

```
enc_circ = cd.construct_encoding_circuit()
enc_circ.draw()
```



We should check that this produces the right state. We should produce the state

$$\left|\bar{0}\right\rangle = |0000000\rangle + |1100110\rangle + |1111000\rangle + |0011110\rangle$$
$$+ |1010101\rangle + |0110011\rangle + |0101101\rangle + |1001011\rangle.$$

```
enc_circ.simulate()
```

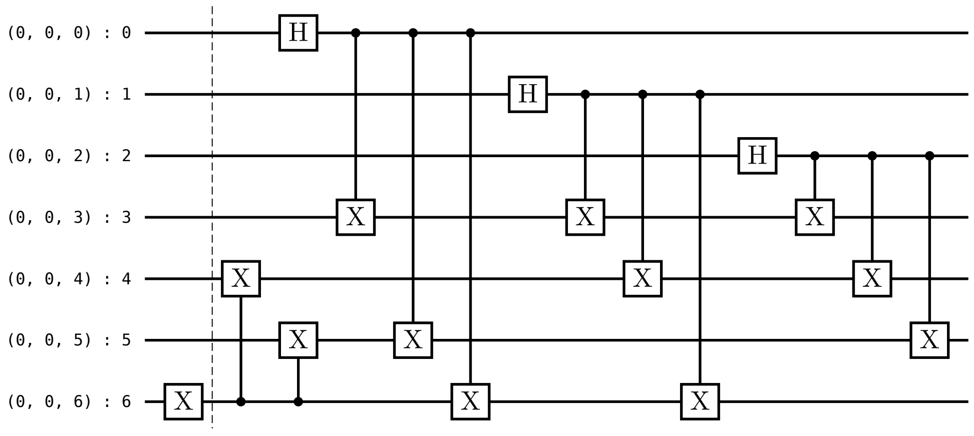
```
  basis    amplitude
-------  -----------
0000000        0.354
1111000        0.354
1100110        0.354
0011110        0.354
1010101        0.354
0101101        0.354
0110011        0.354
1001011        0.354
```

We can similarly, create the logical one state. There are two ways go about this. Either,

1. We feed in the $|1\rangle$ state into the circuit, or
2. We use the logical $\bar{X}$ to flip the logical zero state.

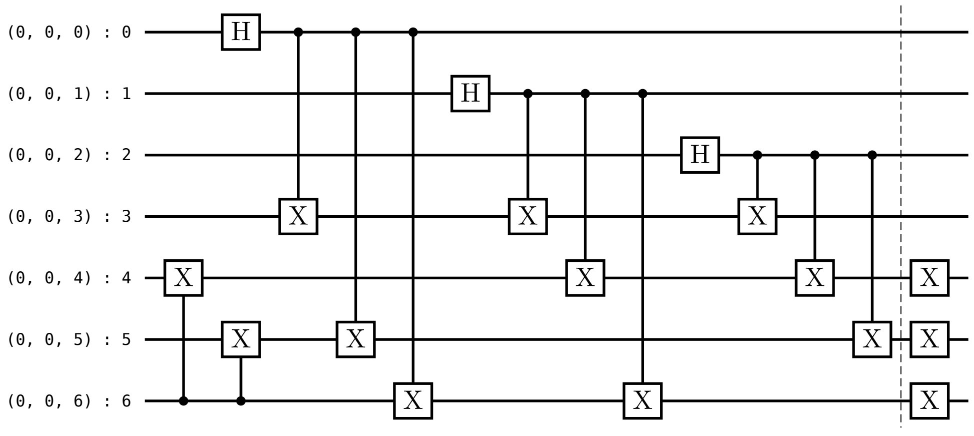
Let's do both.

```
# Option 1
enc_circ_one = stac.Circuit.simple(cd.num_data_qubits)
enc_circ_one.append('X',6)
enc_circ_one.append('TICK')
enc_circ_one += enc_circ
enc_circ_one.draw()
enc_circ_one.simulate()
# looks good
```



```
  basis    amplitude
-------  -----------
0110100        0.354
1001100        0.354
1010010        0.354
0101010        0.354
1100001        0.354
0011001        0.354
0000111        0.354
1111111        0.354
```

```
# Option 2. Use the logical X created above
enc_circ_flip = stac.Circuit.simple(cd.num_data_qubits)
enc_circ_flip += enc_circ
enc_circ_flip.append('TICK')
enc_circ_flip.append('X', 4)
enc_circ_flip.append('X', 5)
enc_circ_flip.append('X', 6)
enc_circ_flip.draw()
enc_circ_flip.simulate()
# looks good
```



```
  basis    amplitude
-------  -----------
0110100        0.354
1001100        0.354
1010010        0.354
0101010        0.354
1100001        0.354
0011001        0.354
0000111        0.354
1111111        0.354
```

Task 2

Use `stac` to create the encoding circuit for the $[[5, 1, 3]]$ code. Simulate it to determine the encoded zero and one states.

```
#
```

Decoding circuits for stabilizer codes

Decoding is not an important part of error correction when quantum error correction is employed to protect against noise during the execution of quantum algorithms. We do all operations - logical operations, error correction and measurement of the output of the computation - on the encoded states themselves. Therefore, decoding is not required at all.

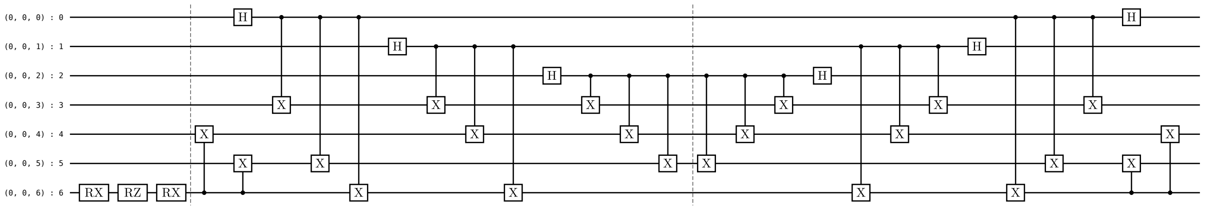
However, it's used in quantum communication protocols, so it is worth studying.

Simple way of decoding

The simple way of decoding is to just run the encoding circuit in reverse. This will work perfectly. We will first encode a random state and then decode it.

```
import stac
cd = stac.CommonCodes.generate_code('[[7,1,3]]')
cd.construct_encoding_circuit();
```

```
from random import random
from math import pi
circ = stac.Circuit.simple(7)
circ.append('rx', 6, random()*pi)
circ.append('rz', 6, random()*pi)
circ.append('rx', 6, random()*pi)

# TICK annotations will allow us to simulate
# the circuit up to every TICK
# In the table below the second column has the
# state upto this TICK
circ.append('TICK')

# then encode it.
circ += cd.encoding_circuit
# In the table below, the third column has the
# state upto this TICK
circ.append('TICK')

# then decode
rev_enc_circ = cd.encoding_circuit.reverse()
circ += rev_enc_circ
# # The fourth column shows the state at this
# # final point

# draw it
circ.draw()

# then simulate it
circ.simulate(head=['basis', 'input state','enc state', 'dec state'], incremental=True)
```



```
  basis   input state    enc state      dec state
-------  -------------  -------------  -------------
0000000  -0.401-0.798j  -0.142-0.282j  -0.401-0.798j
1111000                 -0.142-0.282j
0110100                  0.137-0.081j
1001100                  0.137-0.081j
1010010                  0.137-0.081j
0101010                  0.137-0.081j
```

```
1100110                  -0.142-0.282j
0011110                  -0.142-0.282j
0000001   0.387-0.230j                    0.387-0.230j
1100001                   0.137-0.081j
0011001                   0.137-0.081j
1010101                  -0.142-0.282j
0101101                  -0.142-0.282j
0110011                  -0.142-0.282j
1001011                  -0.142-0.282j
0000111                   0.137-0.081j
1111111                   0.137-0.081j
```

You can see that, as expected, the state decodes correctly. Run this cell a few times to convince yourself.

Gottesman's decoding algorithm

Gottesman's decoding algorithm is quite simple, and again employs the standard form of the stabilizer generator matrix that we computed last time. Instead of decoding in place, we will add k (number of logical qubits) ancilla to our circuit and put the decoded state there. The steps are

1. Compute the standard form of the generator matrix.
2. Determine the logical $\bar{X}$ and logical $\bar{Z}$ operations.
3. Use the following algorithm to determine the decoding circuit.

```
decoding_circuit = []

for i in range(len(logical_zs)):
    for j in range(n):
        if logical_zs[i, n+j]:
            decoding_circuit.append(["cx", j, n+i])

for i in range(len(logical_xs)):
    for j in range(n):
        if logical_xs[i, j] and logical_xs[i, n+j]:
            decoding_circuit.append(["cz", n+i, j])
            decoding_circuit.append(["cx", n+i, j])
        elif logical_xs[i, j]:
            decoding_circuit.append(["cx", n+i, j])
        elif logical_xs[i, n+j]:
            decoding_circuit.append(["cz", n+i, j])
```

As you can see, this algorithm just reads off the entries of the logical operators, and applies appropriate control operations to and from the ancilla qubits.

Question: Determine the number of operations required for Gottesman's encoding algorithm. What is the depth of the circuit? Determine the number of operations required for Gottesman's decoding algorithm. What is the depth of the circuit?

Now, let's try this.

```
cd.construct_standard_form()
print("G =")
stac.print_matrix(cd.standard_generator_matrix, augmented=True)
cd.construct_logical_operators()
print("Logical X =")
stac.print_matrix(cd.logical_xs, augmented=True)
print("Logical Z =")
stac.print_matrix(cd.logical_zs, augmented=True)

dec_circ = cd.construct_decoding_circuit()

dec_circ.draw()
```

```
G =
```

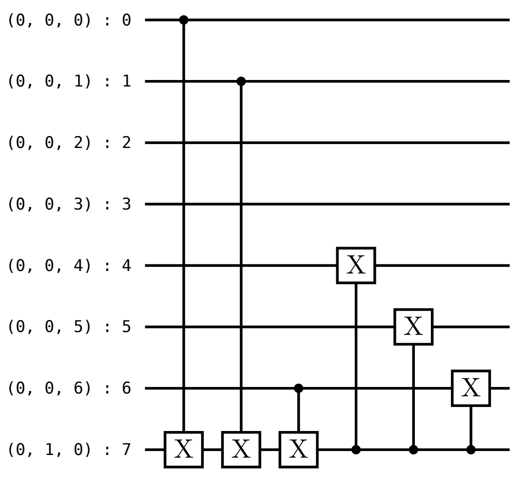
$$\left(\begin{array}{ccccccc|ccccccc}1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1\end{array}\right)$$

```
Logical X =
```

$$\left(\begin{array}{ccccccc|ccccccc}0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{array}\right)$$

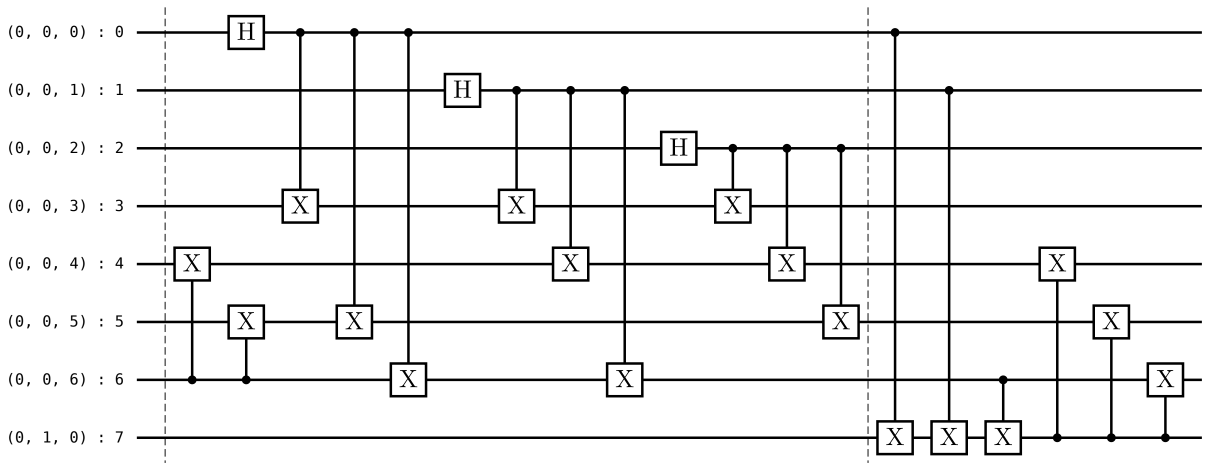
```
Logical Z =
```

$$\left(\begin{array}{ccccccc|ccccccc}0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1\end{array}\right)$$



Does this actually work? Let's try. It will be a bit difficult to see the working of this algorithm with random states, so let's work with the basis states. First, we will see if the zero state is decoded properly.

```python
cd.construct_encoding_circuit()

# first create the zero state
circ = stac.Circuit.simple(cd.num_data_qubits)
circ.append_register(stac.QubitRegister('a', 0, cd.num_logical_qubits))
circ.append('TICK')

# then encode it.
circ += cd.encoding_circuit
circ.append('TICK')

# then decode
circ += cd.decoding_circuit

circ.draw()
circ.simulate(head=['basis', 'input state','enc state', 'dec state'], incremental=True)
```

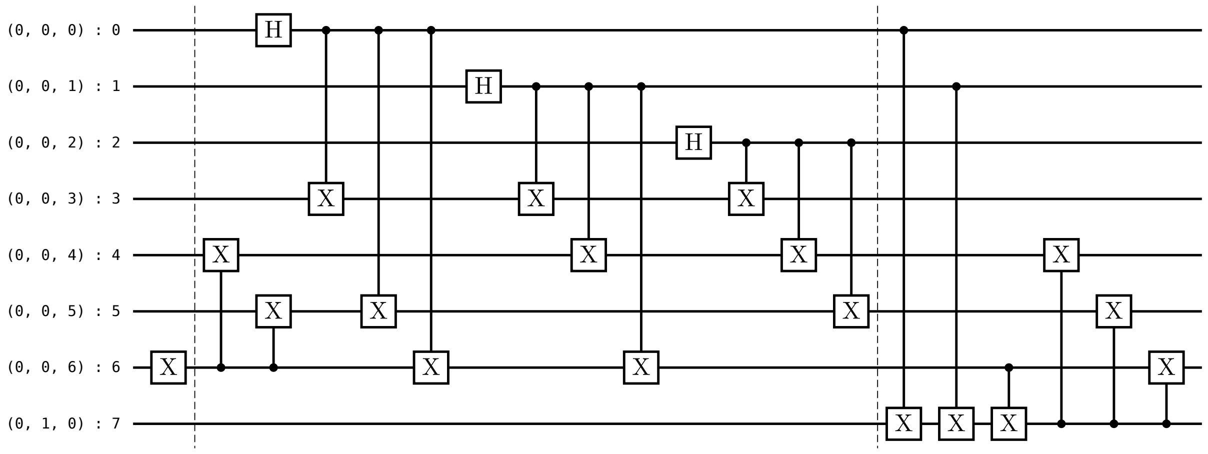


```
   basis  input state      enc state    dec state
--------  -------------  -----------  -----------
00000000   1.000               0.354        0.354
11110000                       0.354        0.354
11001100                       0.354        0.354
00111100                       0.354        0.354
10101010                       0.354        0.354
01011010                       0.354        0.354
01100110                       0.354        0.354
10010110                       0.354        0.354
```

The input state here is the 7th qubit counting from the left, out of the 8 qubits. The first 6 qubits are the encoding ancilla, and the last qubit is the decoding ancilla.

You can see that the input state is $|0000000\rangle\,|0\rangle$. Then, we apply the encoding operation to get the encoded $\left|\bar{0}\right\rangle$ state. After decoding, note that the state of the 8th qubit is still 0. This is not very interesting now. But, let's try with the one state.

```python
# first create the one state
circ = stac.Circuit.simple(cd.num_data_qubits)
circ.append_register(stac.QubitRegister('a', 0, cd.num_logical_qubits))
circ.append('X',6)
circ.append('TICK')

# then encode it.
circ += cd.encoding_circuit
circ.append('TICK')

# then decode
circ += cd.decoding_circuit

circ.draw()
circ.simulate(head=['basis', 'input state','enc state', 'dec state'], incremental=True)
```



```
   basis  input state    enc state    dec state
--------  -------------  -----------  -----------
01101000                  0.354
10011000                  0.354
10100100                  0.354
01010100                  0.354
00000010   1.000
11000010                  0.354
00110010                  0.354
00001110                  0.354
11111110                  0.354
00000001                               0.354
11110001                               0.354
11001101                               0.354
00111101                               0.354
10101011                               0.354
01011011                               0.354
01100111                               0.354
10010111                               0.354
```

And voila, you see that in the final state, the 8th qubit is in the state $|1\rangle$, hence unentangled from the remaining qubits. Also, note that the state of the first 7 qubits has changed into something outside the logical code space (none of the kets that appear in the logical zero and one states are here). This ensures that the first 7 qubits no longer contain any information about an arbitrary logical qubit, hence, respecting the no-cloning theorem.

Task 1

Decode the zero and one state for the $[[5,1,3]]$ code, and make sure you get the right answer.

Syndrome measurements for stabilizer codes

We will revisit syndrome measurements now, and demonstrate a general algorithm to construct them for stabilizer codes.

Our working example will be the Steane code.

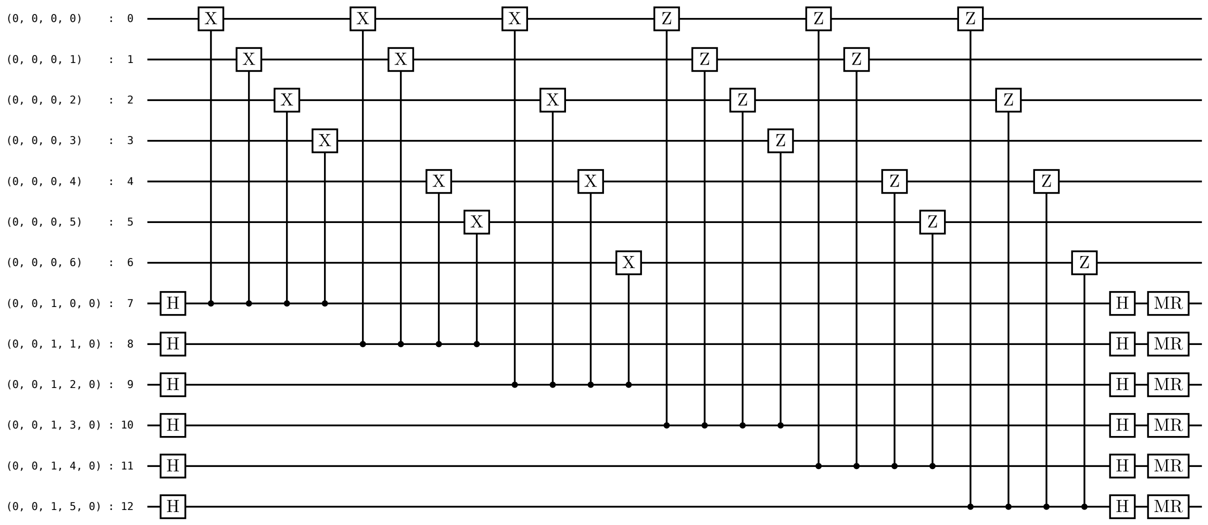
```
import stac
cd = stac.CommonCodes.generate_code("[[7,1,3]]")
stac.print_matrix(cd.generator_matrix, augmented=True)
```

$$\left(\begin{array}{ccccccc|ccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}\right)$$

As you might recall, we said that for stabilizer codes, one has to measure each generator in turn. This is done via the following algorithm, which creates a circuit with $n+m$ qubits.

```
syndrome_circuit = stac.Circuit.simple(cd.num_data_qubits + cd.num_generators)
for i in range(num_generators):
    syndrome_circuit.append("H", n+i)

for i in range(cd.num_generators):
    for j in range(cd.num_data_qubits):
        if cd.generators_x[i, j] and cd.generators_z[i, j]:
            syndrome_circuit.append("CX", n+i, j)
            syndrome_circuit.append("CZ", n+i, j)
        elif cd.generators_x[i, j]:
            syndrome_circuit.append("CX", n+i, j)
        elif cd.generators_z[i, j]:
            syndrome_circuit.append("CZ", n+i, j)

for i in range(cd.num_generators):
    syndrome_circuit.append("H", n+i)

for i in range(cd.num_generators):
        syndrome_circuit.append('MR', self.num_data_qubits+i)
```

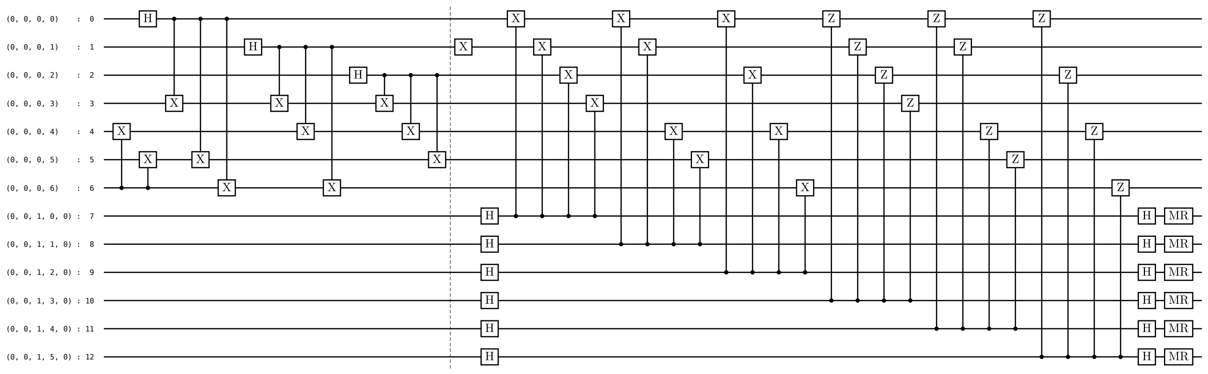
```
cd.construct_syndrome_circuit();
cd.syndrome_circuit.draw()
```



To use this in practice, we will

1. Encode the zero state
2. Introduce an error
3. Measure the stabilizers

```
# Step 1
circ = cd.construct_encoding_circuit('non_ft')
circ.append('TICK')

# Step 2
circ.append('X', 1)

# Step 3
circ += cd.syndrome_circuit

circ.draw()
```



```
# we use stim to simulate this circuit
circ.sample()
```

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

As we can see, an X error on the 2nd qubit leads to a the 3rd and 4th generators failing (counting from 0). The syndrome vector is the $n+2=9$th column (counting from 1) of the generator matrix, because as discussed previously it is the Z-type stabilizer generators that detect bit-flip errors. In this way, each of the n X-type errors and n Z-type errors correspond to one of the columns of the generator matrix.

Task 1

Determine the syndromes for each possible one qubit X error and each possible one qubit Z error.

```
#
```

Task 2

Determine the syndromes when there is a two qubit X error (errors such as X_0X_3). Compare the syndromes with the one-qubit error syndromes. What does this tell you about correcting two-qubit errors with the Steane code?

```
#
```