Stabilizers

February 16, 2022

1 Homework 3

1.1 ECE 621 (Spring 2022)

1.1.1 Adam Carriker

1.2 Problem 1

1.3 1a.

Show the state |0>|+>|+i> is stabilized by the group generated by <ZII,IXI, IIY>

[1]:
```
import numpy as np
import cmath

ZERO = np.array([[0,0],[0,0]], dtype=complex)
I=np.array([[1,0],[0,1]], dtype=complex)
X=np.array([[0,1], [1,0]], dtype = complex)
Y=np.array([[0,-1j], [1j,0]], dtype = complex)
Z=np.array([[1,0], [0,-1]], dtype = complex)
iI=np.dot(1j,I)
iX=np.dot(1j,X)
iY=np.dot(1j,Y)
iZ=np.dot(1j,Z)

k_0 = np.array([[1],[0]])
k_1 = np.array([[0],[1]])
k_p = 1/cmath.sqrt(2)*(k_0 + k_1)
k_m = 1/cmath.sqrt(2)*(k_0 - k_1)
k_pi = 1/cmath.sqrt(2)*(k_0 + 1j*k_1)
k_mi = 1/cmath.sqrt(2)*(k_0 - 1j*k_1)

init_state = np.kron(k_0, np.kron(k_p, k_pi))
print("Initial state |> = |0>|+>|+i> = \n", init_state)

group = {
    "ZII" : np.kron(Z,np.kron(I,I)),
    "IXI" : np.kron(I,np.kron(X,I)),
```

```
    "IIY" : np.kron(I,np.kron(I,Y))
}
group_items = list(group.items())

print("group <ZII, IXI, IIY> :")
for (key,val) in group_items:
    print(key, "=", val)

for (key,val) in group_items:
    op=np.matmul(val,init_state)
    print(f"{key}|> = ", op)
    print("= |> " if np.array_equal(op,init_state) else " |> ")
```

```
Initial state |> = |0>|+>|+i> =
 [[0.5+0.j ]
 [0. +0.5j]
 [0.5+0.j ]
 [0. +0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
group <ZII, IXI, IIY> :
ZII = [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -1.+0.j -0.+0.j -0.+0.j -0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -0.+0.j -1.+0.j -0.+0.j -0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -0.+0.j -0.+0.j -1.+0.j -0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -0.+0.j -0.+0.j -0.+0.j -1.+0.j]]
IXI = [[0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
IIY = [[0.+0.j 0.-1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.-1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.-1.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+1.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.-1.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+1.j 0.+0.j]]
ZII|> =  [[0.5+0.j ]
```

```
 [0. +0.5j]
 [0.5+0.j ]
 [0. +0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
= |>
IXI|> =  [[0.5+0.j ]
 [0. +0.5j]
 [0.5+0.j ]
 [0. +0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
= |>
IIY|> =  [[0.5+0.j ]
 [0. +0.5j]
 [0.5+0.j ]
 [0. +0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
= |>
```

Since each generator operator has |> as an eigenstate with eigenvalue +1, |> is stabilized by this group.

1.4 1b.

Write down an alternative set of generators. Idea: try every permutation of Pauli gates {I,X,Y,Z} in a 3-deep gate

Better idea: use the fact that Clifford operators map Pauli operators to Pauli operators, which means they also must map stabilizer states to stabilizer states.

[2]:
```
def aboutequal(a,b,rt=1e-15, at=1e-15):
    return True if np.array_equal(np.allclose(a,b,rtol=rt, atol=at),np.
 ↪array([[True,True],[True,True]]))\
    else False
```

[3]:
```
# proof below:

H = cmath.sqrt(1/2)*(X+Z)
S = cmath.exp(1j*cmath.pi/4)*cmath.sqrt(1/2)*(I-iZ)
CNOT = 1/2*(np.kron(I,I)+np.kron(Z,I)+np.kron(I,X)-np.kron(Z,X))
```

```
computedZ = np.matmul(H,np.matmul(X,H))
computedX = np.matmul(H,np.matmul(Z,H))
computedI = np.matmul(H,np.matmul(I,H))
computedY = np.matmul(H,np.matmul(Y,H))

print("HXH = Z? :", aboutequal(Z,computedZ))
print("HXH = X? :", aboutequal(X,computedX))
print("HIH = I? :", aboutequal(I,computedI))
print("computedY = ", computedY)
print("HYH = Y? :", aboutequal(Y,computedY))
print("HYH = -Y? :", aboutequal(-Y,computedY))
```

```
HXH = Z? : False
HXH = X? : False
HIH = I? : False
computedY =  [[0.-4.26642159e-17j 0.+1.00000000e+00j]
 [0.-1.00000000e+00j 0.+4.26642159e-17j]]
HYH = Y? : False
HYH = -Y? : False
```

So, now we can try applying the H operator before and after each of the gates in the previous group to get the new group. For example:

```
ZII -> (HZH)(HIH)(HIH) = (X)(I)(I) = XII
IXI -> (HIH)(HXH)(HIH) = (I)(Z)(I) = IZI
IIY -> (HIH)(HIH)(HYH) = (I)(I)(-Y) = II(-Y)
```

[4]:
```
group2 = {
    "ZII" : np.kron(X,np.kron(I,I)),
    "IXI" : np.kron(I,np.kron(Z,I)),
    "IInY" : np.kron(I,np.kron(I,-Y))
}
group2_items = list(group.items())

print("alternate group <XII, IZI, II(-Y)> :")
for (key,val) in group2_items:
    print(key, "=", val)

for (key,val) in group2_items:
    op=np.matmul(val,init_state)
    print(f"{key}|> = ", op)
    print("= |> " if np.array_equal(op,init_state) else " |> ")
```

```
alternate group <XII, IZI, II(-Y)> :
ZII = [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -1.+0.j -0.+0.j -0.+0.j -0.+0.j]
```

```
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -0.+0.j -1.+0.j -0.+0.j -0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -0.+0.j -0.+0.j -1.+0.j -0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j -0.+0.j -0.+0.j -0.+0.j -1.+0.j]]
IXI = [[0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
IIY = [[0.+0.j 0.-1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.-1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+1.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.-1.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+1.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.-1.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+1.j 0.+0.j]]
ZII|> =  [[0.5+0.j ]
 [0. +0.5j]
 [0.5+0.j ]
 [0. +0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
= |>
IXI|> =  [[0.5+0.j ]
 [0. +0.5j]
 [0.5+0.j ]
 [0. +0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
= |>
IIY|> =  [[0.5+0.j ]
 [0. +0.5j]
 [0.5+0.j ]
 [0. +0.5j]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]
 [0. +0.j ]]
= |>
```

Therefore, the group <XII,IZI,II(-Y)> is also a generator for the state |0>|+>|+i>.

1.5 1c)

Find a gate sequence composed of CNOT, H, and S=|0><0|+i|1><1|, (you can write the circuit, if you wish), that transforms the state above into a state stabilized by <XXX,ZZI,IZZ>.

[5]:
```
cnot12 = np.kron(CNOT, I)
print(cnot12)
cnot23 = np.kron( I, CNOT)
print(cnot23)

state1 = np.kron(k_p,np.kron(k_p,k_0))
print(state1)

print(np.matmul(cnot12,state1))
```

```
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]]
[[0.5+0.j]
 [0. +0.j]
 [0.5+0.j]
 [0. +0.j]
 [0.5+0.j]
 [0. +0.j]
 [0.5+0.j]
 [0. +0.j]]
[[0.5+0.j]
 [0. +0.j]
 [0.5+0.j]
 [0. +0.j]
 [0.5+0.j]
 [0. +0.j]
 [0.5+0.j]
 [0. +0.j]]
```

1. Starting with: generators <ZII,IXI,IIY> and state |0>|+>|+i>.
2. Apply H_1: generators H_1<ZII,IXI,IIY> -> <XII,IXI,IIY>; state H_1|0>|+>|+i> ->

|+>|+>|+i>.

3. Apply H_2: generators H_2<XII,IXI,IIY> -> <XII,IZI,IIY>; state H_2|+>|+>|+i> -> |+>|0>|+i>.
4. Apply S_3: generators S_3<XII,IZI,IIY> -> <XII,IZI,II(-X)>; state S_3|+>|0>|+i> -> |+>|0>|->.
5. Apply H_3: generators H_3<XII,IZI,II(-X)> -> <XII,IZI,II(-Z)>; state H_3|+>|0>|-> -> |+>|+>|1>.
6. Apply X_3: generators X_3<XII,IZI,II(-Z)> -> <XII,IZI,IIZ>; state X_3|+>|+>|1> -> |+>|+>|0>.
7. Apply CNOT_{1,2}: generators CNOT_{1,2}<XII,IZI,IIZ> -> <XXI,ZZI,IIZ>; state CNOT_{1,2}|+>|+>|0> -> |+>|+>|0>.
8. Apply CNOT_{2,3}: generators CNOT_{2,3}<XXI,ZZI,IIZ> -> <XXX,ZZI,IZZ>; state CNOT_{2,3}|+>|+>|0> -> 1/Sqrt[2]*(|000>+|110>).

The final gate sequence to obtain generators <XXX,ZZI,IZZ> is: CNOT_{2,3} CNOT_{1,2} X_3 H_3 S_3 H_2 H_1 |0>|+>|+i>

1.6 1d) Write the wavefunction of the state above.

The wavefunction of the state above becomes 1/Sqrt[2]*(|000>+|110>).

1.7 Problem 2

Your friend Andrew Cross tells you that he has a code on 7 qubits that is stabilized by the following generators

<Z1Z5, Z2Z5, Z3Z6, Z4Z7, X3X4Y6Y7, X1X2X3Z4X5X6>

or

<ZIIIZII, IZIIZII, IIZIIZI, IIIZIIZ, IIXXIYY, XXXZXXI>

1.8 1) Show that this code can correct any single qubit error.

Considering only one single qubit error, we can write the quantum error correction condition as:

<j|_L E^{†}$|k > \_l$

Considering that E^{†}$canbeanyPauli, wantwewantisfortheerrorconditiontogoto0whenweapplyastabilizergenerator$

<ϕ■|_LE■{†}$S\_n|\phi >\ = 0$

This will be true if a stabilizer generator has an anti-commuting gate with respect to the error on the same qubit.

[6]:
```
# Find which Paulis anti-commute


def find_anticommutator(a,b):
    return np.matmul(a,b)+np.matmul(b,a)

pauli_group = {
```

```
    "I": I,
    "X": X,
    "Y": Y,
    "Z": Z
}

for (kop1, op1) in pauli_group.items():
    for (kop2, op2) in pauli_group.items():
        anticommutator = find_anticommutator(op1,op2)
        if np.array_equal(anticommutator,ZERO) is True:
            print(f"Pauli operators: {kop1}, {kop2} anti-commute.")
```

```
Pauli operators: X, Y anti-commute.
Pauli operators: X, Z anti-commute.
Pauli operators: Y, X anti-commute.
Pauli operators: Y, Z anti-commute.
Pauli operators: Z, X anti-commute.
Pauli operators: Z, Y anti-commute.
```

Above shows that Pauli X anti-commutes with Y and Z, Y anti-commutes with X and Z, and Z anti-commutes with X and Y. Therefore, to show that any 1-qubit error can be corrected by our stabilizer code, we just need to show that we can arbitrarily apply an X or Z operator at any qubit position by selecting the correct stabilizer row S_n. The first four rows of the code, {ZIIIZII, IZIIZII, IIZIIZI, IIIZIIZ}, ensure that a Z gate can be applied at any qubit position. The last two rows, {IIXXIYY, XXXZXXI} allow an X gate to be applied to any qubit position. This allows us to correct any single qubit error.

1.9 2) Determine logical Z and X operators for this code.

[27]:
```
def find_commutator(a,b):
    return np.matmul(a,b)-np.matmul(b,a)

def find_anticommutator(a,b):
    return np.matmul(a,b)+np.matmul(b,a)

def create_operator_from_string(op_string):
    op = I
    for c in op_string:
        op = np.matmul(op, pauli_group[c])
    return op

def create_operator_matrix(operator_string_dict):
    generated_op = {}
    for (k,v) in operator_string_dict.items():
        op = create_operator_from_string(v)
        generated_op[k] = op
    return generated_op
```

[30]:
```
# Based off of code from: https://www.geeksforgeeks.org/
↪print-all-combinations-of-given-length/
# !!!! This runs in s^k time, very inefficient !!!!
def allKLength(s, k):
    n = len(s)
    return(allKLengthRec(s, "", n, k, ret=[]))

def allKLengthRec(s, prefix, n, k, ret):
    if (k == 0) :
        ret.append(prefix)
        return ret
    for i in range(n):
        newPrefix = prefix + s[i]
        ret=allKLengthRec(s, newPrefix, n, k - 1, ret)
    return ret
```

[34]:
```
# brute force guess logical X and Z operators that work for code
# !!! This is incredibly inefficient, i'm so sorry if you actually run this !!!
def find_logical_ops_handler(generators_strs):
    (lX, lZ) = find_logical_ops(generators_strs)
    generators_strs.update({
        "lX": lX,
        "lZ": lZ
    })
    new_code_generated = create_operator_matrix(generators_strs)

    for (kop1, op1) in new_code_generated.items():
        if(kop1 in ("lZ", "lX")):
            for (kop2, op2) in new_code_generated.items():
                commutator = find_commutator(op1,op2)
                anticommutator = find_anticommutator(op1,op2)
                if np.array_equal(commutator,ZERO) is True:
                    print(f"Pauli operators: {kop1}, {kop2} commute.")
                if np.array_equal(anticommutator,ZERO) is True:
                    print(f"Pauli operators: {kop1}, {kop2} anti-commute.")

def find_logical_ops(generators_strs):
    n=len(generators_strs["g1"])
    generators = create_operator_matrix(generators_strs)
    possible_operators = allKLength(list(pauli_group.keys()), n)
    for lZ in possible_operators:
        for lX in possible_operators:
            lX_op = create_operator_from_string(lX)
            lZ_op = create_operator_from_string(lZ)
            if(np.array_equal(find_anticommutator(lX_op,lZ_op),ZERO)):
                commute_flag = True
                for (l_kop, l_op) in ((lX, lX_op),(lZ, lZ_op)):
```

```
                    for (kop, op) in generators.items():
                        commutator = find_commutator(l_op,op)
                        anticommutator = find_anticommutator(l_op,op)
                        if np.array_equal(commutator,ZERO) is False:
                            commute_flag = False
                            break
                    if commute_flag == False:
                        break
                if commute_flag is True:
                    print(f"logical operators X_L={lX} and Z_L={lZ} work.")
                    return (lX,lZ)


# Test find_logical_ops_handler
five_qubit_gens_strs = {
    "g1": "XZZXI",
    "g2": "IXZZX",
    "g3": "XIXZZ",
    "g4": "ZXIXZ",
}

find_logical_ops_handler(five_qubit_gens_strs)
```

```
logical operators X_L=IIIIY and Z_L=IIIIX work.
Pauli operators: lX, g1 commute.
Pauli operators: lX, g2 commute.
Pauli operators: lX, g3 commute.
Pauli operators: lX, g4 commute.
Pauli operators: lX, lX commute.
Pauli operators: lX, lZ anti-commute.
Pauli operators: lZ, g1 commute.
Pauli operators: lZ, g2 commute.
Pauli operators: lZ, g3 commute.
Pauli operators: lZ, g4 commute.
Pauli operators: lZ, lX anti-commute.
Pauli operators: lZ, lZ commute.
```

[33]:
```
# Test problem 2 code generators and logical ops
# !!! This is incredibly inefficient, i'm so sorry if you actually run this !!!
prob2_code_strs = {
    "g1": "ZIIIZII",
    "g2": "IZIIZII",
    "g3": "IIZIIZI",
    "g4": "IIIZIIZ",
    "g5": "IIXXIYY",
    "g6": "XXXZXXI",
}
```

```
find_logical_ops(prob2_code_strs)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/var/folders/k8/wy1014qx29x7j42dzjtj5zbh0000gn/T/ipykernel_69985/1320410860.py in
 ↪<module>
     10 }
     11
---> 12 find_logical_ops(prob2_code_strs)

/var/folders/k8/wy1014qx29x7j42dzjtj5zbh0000gn/T/ipykernel_69985/487974432.py in
 ↪find_logical_ops(generators_strs)
     25     for lZ in possible_operators:
     26         for lX in possible_operators:
---> 27             lX_op = create_operator_from_string(lX)
     28             lZ_op = create_operator_from_string(lZ)
     29             if(np.array_equal(find_anticommutator(lX_op,lZ_op),ZERO)):

/var/folders/k8/wy1014qx29x7j42dzjtj5zbh0000gn/T/ipykernel_69985/2932398810.py in
 ↪create_operator_from_string(op_string)
      8     op = I
      9     for c in op_string:
---> 10         op = np.matmul(op, pauli_group[c])
     11     return op
     12

KeyboardInterrupt:
```

I was not able to get this function to get the logical X and Z operators in a reasonable amount of time.

1.10 3) Show that transforming the stabilizer generators by applying a single qubit clifford gate to any qubit still yields a code that corrects all single qubit errors.

[22]:
```
# Apply a H gate to each Pauli and add new operators to Pauli group
H= cmath.sqrt(1/2)*(X+Z)
pauli_group.update({
    "i": np.matmul(H,I),
    "x": np.matmul(H,X),
    "y": np.matmul(H,Y),
    "z": np.matmul(H,Z)
})

# Modify generators to have H applied to first qubit
prob2_code_strs = {
```

```python
    "g1": "zIIIZII",
    "g2": "iZIIZII",
    "g3": "iIZIIZI",
    "g4": "iIIZIIZ",
    "g5": "iIXXIYY",
    "g6": "xXXZXXI",
}

prob2_code_generated = create_operator_matrix(prob2_code_strs)

for (kop1, op1) in prob2_code_generated.items():
    for (kop2, op2) in prob2_code_generated.items():
        commutator = find_commutator(op1,op2)
        anticommutator = find_anticommutator(op1,op2)
        if np.array_equal(commutator,ZERO) is True:
            print(f"Pauli operators: {kop1}, {kop2} commute.")
        if np.array_equal(anticommutator,ZERO) is True:
            print(f"Pauli operators: {kop1}, {kop2} anti-commute.")
```

```
Pauli operators: g1, g1 commute.
Pauli operators: g1, g2 commute.
Pauli operators: g1, g3 commute.
Pauli operators: g1, g4 commute.
Pauli operators: g1, g5 commute.
Pauli operators: g2, g1 commute.
Pauli operators: g2, g2 commute.
Pauli operators: g2, g3 commute.
Pauli operators: g2, g4 commute.
Pauli operators: g2, g5 commute.
Pauli operators: g3, g1 commute.
Pauli operators: g3, g2 commute.
Pauli operators: g3, g3 commute.
Pauli operators: g3, g4 commute.
Pauli operators: g3, g5 commute.
Pauli operators: g4, g1 commute.
Pauli operators: g4, g2 commute.
Pauli operators: g4, g3 commute.
Pauli operators: g4, g4 commute.
Pauli operators: g4, g5 commute.
Pauli operators: g5, g1 commute.
Pauli operators: g5, g2 commute.
Pauli operators: g5, g3 commute.
Pauli operators: g5, g4 commute.
Pauli operators: g5, g5 commute.
Pauli operators: g6, g6 commute.
```

Since each generator still commutes with each other generator with an H gate applied to the first qubit, we know that we can still generate any correction that we were able to before with the regular generators. Therefore, we are still able to correct any single qubit error.