In [3]:
```
%load_ext autoreload
%autoreload 2
```

In [115]:
```
import pandas as pd
import numpy as np
from functools import reduce
from itertools import product

from dataclasses import dataclass
```

We will try to avoid implementing thigns with circuits, for maximal transparency, at the expense of velocity.

Rerun this cell to reload latex commands.

Common concepts:

- Logical space: This is the information that you want to represent, the idealized qubit.
- Physical space: This is how qubits "actually" are. The noise affects these qubits.

We hope to establish a mapping between the physical space and the logical space so that we can operate in the logical space despite errors affecting the physical space.

A new hope: Quantum Fault-Tolerance Theorem.

In [ ]:
```
```

Operations in quantum comptuign are unitaries. These are generalized by quantum channels. In essense, a channel can be thought as a set of gates $U_1, \ldots, U_n$, and probabilities $p_1, \ldots, p_n$, such that gate U_i is applied with probability p_i.

These maps are also called, for technical reasons, Completely Positive and Trace Preserving (CPTP) maps.

The common setup in the following situations is the following. You have some quantum information, which is encoded in some way. Then something happens to the state, and you must be able to recover the initial information. The nature of the "something" depends on the state.

In stabilizer codes, "something" will mean a gate is randomly applied to the state. This includes the identity gate, which is the same as not doing anything. For the irrep and the AdS/CFT inspired code, this will mean a qubit disappearing.

In [ ]:
```
```

1 Stabilizer Codes

Let X, Y, Z the Pauli matrices and Π the subgroup generated by them, up to quarter phases. That is, $i^k, i^k X, i^k Y, i^k Z$. Note that any two of these matrices commute or anticommute. Note that the eigenvalues of X, Y, Z are ± 1.

Let $\Pi^{\otimes n}$ the group generated by tensoring Π n times. Thus

$$\Pi^{\otimes n} = \left\{ i^k A_1 \otimes A_2 \otimes \cdots \otimes A_n : k \in \{0, 1, 2, 3\}, A_i \in \{I, X, Y, Z\} \right\}$$

Note that all the matrices in $\Pi^{\otimes n}$ without a phase have eigenvalues ± 1.

Let $\mathcal{S}$ be an abelian subgroup of $\Pi^{\otimes n}$. We will call this the stabilizer subgroup.

The code space will be the shared $+1$-eigenspace. That is, it will be the subspace of $(\mathbb{C}^2)^{\otimes n}$ consisting of the vectors $|\psi\rangle$ such that $S|\psi\rangle = |\psi\rangle$ for all $S \in \mathcal{S}$. We call this shared $+1$-eigenspace $V_{\mathcal{S}}$.

Consider $E \in \Pi^{\otimes n}$, an operation (error) we want to detect. For every element $g \in \mathcal{S} \subseteq \Pi^{\otimes n}$, we have that $Eg = gE$ or $Eg = -gE$. Suppose it is the latter. Let $|\psi\rangle \in V_{\mathcal{S}}$. Definitionally $g|\psi\rangle = |\psi\rangle$. We have that $gE|\psi\rangle = -Eg|\psi\rangle = -E|\psi\rangle$. So $E|\psi\rangle \notin V_{\mathcal{S}}$! Since E kicks out elements of $V_{\mathcal{S}}$ outside, we should be able to detect it somehow.

Suppose that this never happens. This means that $gE = Eg$ for every $g \in \mathcal{S}$. Thus E is in the centralizer of $\mathcal{S}$. Note that since $\mathcal{S}$ is an abelian subgroup, this is the same as the normalizer of $\mathcal{S}$.

```
In [ ]:
```

If we have a minimal set of k generators, we have that $\mathcal{S}$ will have size 2^k, and the code space will have dimension 2^{n-k}.

To avoid dealing with floating point, we will deal with $\Pi^{\otimes n}$ in a combinatorial way.

In [153]:
```python
I = np.eye(2)
X = np.array([[0,1],[1,0]])
Y = np.array([[0,-1j],[1j,0]])
Z = np.array([[1,0],[0,-1]])

def letter_to_matrix(l):
    """Converts a letter into a Pauli matrix."""
    if l == 'I':
        return I
    elif l == 'X':
        return X
    elif l == 'Y':
        return Y
    elif l == 'Z':
        return Z
    raise ValueError('Must be I, X, Y, or Z.')

def l_otimes(*ps):
    return reduce(np.kron, ps)

def product_pauli(l, r):
    """Return the product between pauli matrices l, r, with phase."""
    if l == r:
        return 0, 'I'
    if l == 'I':
        return 0, r
    if r == 'I':
        return 0, l
    if l == 'X':
        if r == 'Y':
            return 1, 'Z'
        if r == 'Z':
            return -1, 'Y'
    if l == 'Y':
        if r == 'X':
            return -1, 'Z'
        if r == 'Z':
            return 1, 'X'
    if l == 'Z':
        if r == 'X':
            return 1, 'Y'
        if r == 'Y':
            return -1, 'X'

@dataclass
class PauliNElement:
    """Class representing an element of \Pi^{\otimes n}"""
    components: str
    phase: int = 0

    def __post_init__(self):
        self.phase = self.phase % 4

    def to_matrix(self):
        mat = l_otimes(*[letter_to_matrix(c) for c in self.components])
        if self.phase == 0:
```

```
            return mat
        if self.phase == 1:
            return 1j*mat
        if self.phase == 2:
            return -mat
        if self.phase == 3:
            return -1j*mat

    def dot(self, right):
        assert len(self.components) == len(right.components)
        phase_prod = [product_pauli(l, r) for l, r in zip(self.components,
        components = ''.join(pr for _, pr in phase_prod)
        phase = sum(ph for ph, _ in phase_prod) + self.phase + right.phase
        return PauliNElement(components, phase)

    def rotate(self, phase):
        return PauliNElement(self.components, self.phase + phase)

    def __mul__(self, other):
        return self.dot(other)

    def __neg__(self):
        return PauliNElement(self.components, 2+self.phase)
```

In [342]:
```
def is_comm(a, b):
    return a * b == b * a

def is_comm_pm(a, b):
    if is_comm(a, b):
        return 0
    else:
        return 1

def is_stabilizer_subgroup(S, tol = 1e-6):
    """Checks that all matrices in S commute with each other."""
    n = len(S)
    for i in range(n):
        for j in range(i):
            if not is_comm(S[i],S[j]):
                return False
    return True

def all_in_eigenspace_1(S, basis_espace, tol = 1e-6):
    for op in S:
        actual_op = op.to_matrix()
        for idx, vec in enumerate(basis_espace):
            if np.linalg.norm(actual_op @ vec - vec) > tol:
                print(op, idx)
                return False
    return True

def get_all_elements_generated(S):
    """Generates all elements from the generator set of the minimal set S."
    if len(S) == 1:
        yield PauliNElement('I' * len(S[0].components))
        yield S[0]
    else:
        for op in get_all_elements_generated(S[1:]):
            yield op
            yield S[0].dot(op)

def projector(S):
    if len(S) == 1:
        return S[0]
```

In [147]:
```
def encode_information(psi):
    """Encodes infrmation"""
    pass
```

In [148]:
```python
def insert_phase(it):
    for op in it:
        for i in range(4):
            yield op.rotate(i)

def PauliGroup_NoPhase(N):
    for l in product('IXYZ', repeat=N):
        yield PauliNElement(''.join(l))

PauliGroup = lambda N: insert_phase(PauliGroup_NoPhase(N))

# Generator subgroup of S
S = [
    PauliNElement('XZZXI'),
    PauliNElement('ZZXIX'),
    PauliNElement('ZXIXZ'),
    PauliNElement('XIXZZ'),
]

def syndrome(S, E):
    return [is_comm_pm(op,E) for op in S]

def group_normalizer_no_phase(S):
    N = len(S[0].components)
    for p in PauliGroup_NoPhase(N):
        comm = True
        for s in S:
            if is_comm(p, s):
                comm = False
                break
        if comm:
            yield p

group_normalizer = lambda S: insert_phase(group_normalizer_no_phase(S))
```

In [360]:
```python
def is_hermitian(m, tol = 1e-6):
    return (np.linalg.norm(np.conj(m.T)) < tol)

def measure_in_eigspace(op, psi):
    m = op.to_matrix()
#     assert is_hermitian(m)
    evals, evecs = np.linalg.eig(m)

    # indices with +1 eig
    p1_positions = evals > 0.5
    # indices with -1 eig
    n1_positions = evals < 0.5

    # psi in eigen basis

    psi_0 = np.linalg.inv(evecs) @ psi
#     breakpoint()
#     breakpoint()
    probs = np.abs(psi_0)**2

    measure_pos_prob = probs[p1_positions].sum()

    pos_vector = psi_0.copy()
    pos_vector[n1_positions] = 0
    pos_vector = evecs @ pos_vector
    if measure_pos_prob > 1e-6:
        pos_vector /= np.linalg.norm(pos_vector)
    else:
        pos_vector = None

    neg_vector = psi_0.copy()
    neg_vector[p1_positions] = 0
    neg_vector = evecs @ neg_vector
    if measure_pos_prob < 1 - 1e-6:
        neg_vector /= np.linalg.norm(neg_vector)
    else:
        neg_vector = None

    return (measure_pos_prob, pos_vector), (1-measure_pos_prob, neg_vector)
```

In [352]:
```
# For 5 qubit case
XBar = PauliNElement(components='XXXXX')

five0 = l_otimes(*[np.array([1,0]) for _ in range(5)])

zero = sum([op.to_matrix() @ five0 for op in get_all_elements_generated(S)]
zero /= np.linalg.norm(zero)

code_space = [
    zero,
    XBar.to_matrix() @ zero
]

def to_code_space(v):
    return v[0] * code_space[0] + v[1] * code_space[1]
```

In [353]:
```
# This shows that all the elements in our code space are actually eigenvect
# with eigenvalue 1
all_in_eigenspace_1(S,code_space)
```

Out[353]:
```
True
```

In [361]:
```
measure_in_eigspace(S[0], zero)
```

Out[361]:
```
((1.0000000000000002,
  array([ 0.25+0.j,  0.  +0.j,  0.  +0.j, -0.25+0.j,  0.  +0.j,  0.25+0.
j,
        -0.25+0.j,  0.  +0.j,  0.  +0.j,  0.25+0.j,  0.25+0.j,  0.  +0.
j,
        -0.25+0.j,  0.  +0.j,  0.  +0.j, -0.25+0.j,  0.  +0.j, -0.25+0.
j,
         0.25+0.j,  0.  +0.j,  0.25+0.j,  0.  +0.j,  0.  +0.j, -0.25+0.
j,
        -0.25+0.j,  0.  +0.j,  0.  +0.j, -0.25+0.j,  0.  +0.j, -0.25+0.
j,
        -0.25+0.j,  0.  +0.j])),
 (-2.220446049250313e-16, None))
```

In [365]:
```
# Error example. One bitflip in a position.
E = PauliNElement('XIIII')
print('Syndrome: ', syndrome(S,E))
```

```
Syndrome:  [0, 1, 1, 0]
```

In [367]:
```
# NoOp alternative
NoOp  = PauliNElement('IIIII')
print('Syndrome: ', syndrome(S,NoOp))
```

```
Syndrome:  [0, 0, 0, 0]
```

In [372]:
```
initial_vec = zero

error_vec = E.to_matrix() @ zero
```

In [376]:

Out[376]: 1.0000000000000002

In [379]:
```
vec = error_vec.copy()
for op in S:
    pos_case, neg_case = measure_in_eigspace(op, vec)

    # One of these cases will always be true
    # Case if negative
    if pos_case[1] is None:
        print('1 ',end='')
        vec = neg_case[1]
    else:
        print('0 ', end='')
        vec = pos_case[1]
```

```
0 1 1 0
```

```
Note this is the same as the syndrome!
```

In [380]:
```
vec = initial_vec.copy()
for op in S:
    pos_case, neg_case = measure_in_eigspace(op, vec)

    # One of these cases will always be true
    # Case if negative
    if pos_case[1] is None:
        print('1 ',end='')
        vec = neg_case[1]
    else:
        print('0 ', end='')
        vec = pos_case[1]
```

```
0 0 0 0
```

Again, same as syndrome. So we can identify that the error happened versus not happening. Now compare with the identity.

More generally, we can identify any error that has at most one non identity position and correct.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: