Project 3: Fault-Tolerant Library

Final Deadline: 11:55 pm Wed 4/24

[Direct Autograder Link](https://autograder.io/)

[Starter Code](https://drive.google.com/drive/u/1/folders/1umAQjNvOOiF4J3GDdKGEldcTj6bEfc_U)

# Introduction

You will be implementing a subset of a library needed to build fault-tolerant quantum circuits based on the Steane error correction code. Specifically you will be implementing functions to:

* Correct arbitrary single qubit errors
* Generate fault-tolerant circuits for the X, Y, Z, H, S, T and CNOT gates
* Decode a measurement into a logical 0 or 1

This is not everything that is needed to fully design fault-tolerant circuits. To do so, we would also need to design fault-tolerant circuits for ancilla preparation and measurements, however we will not focus on these (although their implementation follows from material already discussed in lecture).

The content for this project was covered in lectures 15-19.

We are requiring that all functions returning circuits return them as [Instruction](https://www.youtube.com/watch?v=QkximZys08w) objects (these are like Gate objects, but allow non-unitary operations like measurement, which will be necessary for some operations). A quantum circuit can be converted to an Instruction via the to\_instruction method. E.g.:

def FT\_X() -> Instruction:

"""Returns 7 qubit circuit implementing fault tolerant X gate using Steane code"""

qc = QuantumCircuit(7)

# Add gates as appropriate

return qc.to\_instruction(label="FT\_X")

Instructions can then be added to other circuits using the [append](https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.append.html) method, just like with Gate objects.

## Preparing the Steane State and Correcting Errors

In lecture, we showed how a logical |0> can be represented across 7 qubits as an equal superposition of all even-weight Hamming codewords. For this project, the most significant bits of a statevector will be listed to the left, and so the parity bits are laid out as follows (starting at bit index 1):

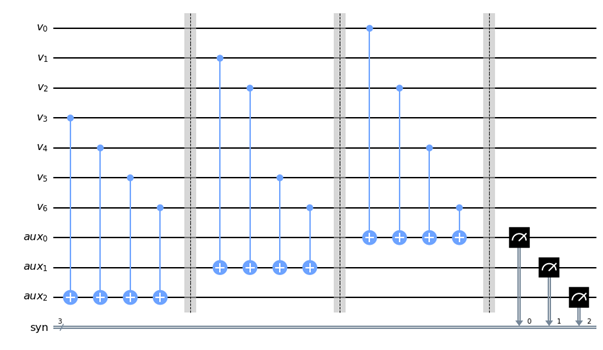
| **Bit index** | | **7** | **6** | **5** | **4** | **3** | **2** | **1** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Encoded bit | | **d\_4** | **d\_3** | **d\_2** | **p\_4** | **d\_1** | **p\_2** | **p\_1** |
| Covered by parity bit? | p\_1 | *✓* |  | *✓* |  | *✓* |  | *✓* |
| p\_2 | *✓* | *✓* |  |  | *✓* | *✓* |  |
| p\_4 | *✓* | *✓* | *✓* | *✓* |  |  |  |

The set of Hamming codewords with an even number of 1s is (again, with the most significant bits listed on the left):

|0000000>, |1010101>, |1100110>, |0110011>, |1111000>, |0101101>, |0011110>, |1001011>

A logical |1> can be represented as an equal superposition of all odd-weight Hamming codewords, which we can get as the bitwise negation of the codewords above.

As discussed in lecture, we can correct an arbitrary X rotation on a single qubit by measuring the parity bits and performing conditional X gates as necessary:



See lab 8 for guidance on how to conditionally apply gates based on a classical measurement.

Phase-flips can then be corrected by applying the same circuit, in which the code bits (listed as "v" above) are sandwiched between H gates. The sequence of these 2 circuits together will correct an arbitrary error on a signal qubit.

Since quantum circuits require exponential resources to simulate, we are requiring that you limit the number of qubits used in your circuit. In particular, you should only use 3 ancilla bits for error correction and 3 classical bits. You should reset the ancilla bits to |0> after each bit/phase-flip correction (you will need to use the [reset](https://www.youtube.com/watch?v=Mm7URLfR4Jw) instruction since computation is not possible for non-unitary transformations involving measurement):

anc = AncillaRegister(3,"anc")

qc.reset(anc) # sets each ancilla bit to |0>

You should also only use 3 classical bits for measurement, so apply any corrective gates before making another syndrome measurement and overwriting the previous results.

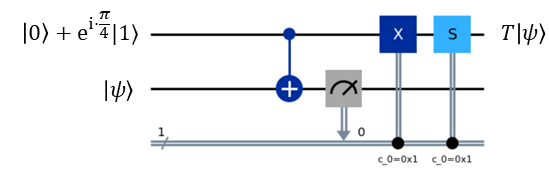
When you run tests for your designs, you will need to first initialize your circuit into a valid Steane state. There are unitary circuits to do this, but the easiest way is to just apply error correction to the default |0000000> state. This will "correct" it to the nearest valid Steane state, which happens to be |0>\_steane. You can then apply other fault-tolerant gates to map it into any desired logical state.

## Fault-tolerant Gates

As discussed in lecture 19, you will need to evaluate which gate operations must be applied transversally and which cannot. For CX, the 7 least significant qubits correspond to logical bit 0 which acts as the control, and the 7 most significant qubits correspond to logical bit 1 which acts as the target.

These gates must be designed such that an error affecting one qubit before the gate is applied can be corrected by applying the error\_correction circuit after the gate.

You will most likely find designing the encoded T Gate most challenging, since it cannot be implemented transversally. Below is the implementation of the T gate discussed in lecture.



We designed it considering one qubit each for the code and ancilla qubit. You will need to extend each logical qubit into 7 physical qubits. You can use the [initialize](https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.QuantumCircuit#initialize) method to set the auxiliary qubits to the appropriate value. This works by passing in a Python list representing the statevector of the corresponding qubits. For example:

anc = AncillaRegister(2,"anc")

qc.initialize([0,0,1,0],anc)

will set the two ancilla bits to the state |10>. The classically controlled X and S gates will need to be triggered only if the measurement across all physical qubits corresponds to a logical 1. We are restricting you to 1 classical bit for these T gates, so you must use the available auxiliary ~~gates~~ qubits to perform this decoding logic. CLARIFICATION: as with the error correction circuit, the code bits should be the least significant, and the ancilla bits should be the most significant.

You must also ensure that the final state is encoded on the original code qubits, unlike the ancilla bits shown in the circuit above.

## Decoding Measurements

At the end of your tests, you will need to perform measurements and decode the result. Since errors can generally occur during or immediately before measurement, you will need to perform error correction on the measured bit string. However, the measured result is a classical bitstring, so you will perform the error correction classically rather than with a quantum circuit. decode\_measurement will take in a string representing the measured result of 7 qubit logical |0> or |1>, perform error correction if needed, and return the int 0 or 1 as appropriate.

# Restrictions

You may use anything in the Qiskit SDK and the numpy, math, random, and unittest packages. To ensure compatibility across different Qiskit versions, you should use only use the provided execute.py to run simulations.

# Testing

You must provide a set of test functions written in tests\_p3.py to the autograder. You must use the [Unittest model](https://docs.python.org/3/library/unittest.html) discussed in lab. Your tests will be graded on whether they cause assertion failures when run on buggy solutions, but do not cause assertion failures on correct implementations.

# Submission and Grading

Submit p3.py and tests\_p3.py to the autograder using the direct link at the top of this page.

We will grade your code on functional correctness. As a reminder, you may not share any part of your solution with others. This includes both code and test cases. Doing so will result in an honor code violation. You will get feedback on your total score, but you will not have access to what the private test cases are checking for.

Efficiency is not graded, but your code must complete in a reasonable amount of time.