Assignment 5

***Due**: Wednesday, Nov 29th, 2023 @ 5:00PM*

CPSC 447/547 Introduction to Quantum Computing (Fall 2023)

1 Introduction

Welcome to Assignment 5 for CPSC 447/547 (Introduction to Quantum Computing). As usual, collaboration is encouraged; if you discussed with anyone besides the course staff about the assignment, *please list their names* in your submission.

Getting Started.

This assignment has two parts, a *written portion* and a *programming portion*. The tasks that are marked by "($\star$ pts)" are optional. Typesetting your solutions to the written portion is not mandatory but highly encouraged. See the instructor's note on Ed for details about Latex for quantum computing. Some basic familiarity with Python and object-oriented programming is required to complete the programming portion of this assignment. No Python packages, except for `math` and `numpy`, are allowed. To start,

- Create a folder for Assignment 5, e.g., `A5/`
- Download the starter files for this assignment to that folder from the course website:
  - `written.tex`
  - `A5.py`
  - `requirement_A5.py` (Do not modify)
- Write your solutions in `A5.py` (for programming tasks) and in `written.tex` or on paper (for written tasks).
- Debug and test your solution locally by running '`python3 A5.py`' on command line. This will check for any violation of the requirements and run correctness tests. Feel free to add more tests in `A5.py`. Do not hardcode your solutions for each public test cases.

Submission. Once you have completed and are ready to submit, upload two files to Gradescope (accessed through Canvas): `written.pdf` and `A5.py`. Gradescope will immediately show the results from running the requirement test and public test cases. If your file fails the tests in `requirement_A5.py`, a **0** score will be assigned.

After the deadline, your written solution will be graded manually by our course staff; your programming solution will be graded using our auto-grading script that contains private test cases. Late submissions (for up to two days) will receive a 50% penalty.

WRITTEN PORTION
This portion of the assignment has a total of 80 *points.*

In this assignment, we will explore the basics of quantum error correction. In the first part of the assignment, we will use the operator sum representation to understand some important single-qubit quantum errors. Then in the second part, we introduce the stabilizer language to describe quantum error correction codes.

2 Phase Flip Channel

Task 2.1 ($\star$ pts)

Consider a single-qubit error model where the qubit experiences a phase flip, Z, with probability p, and stays the same otherwise. Specifically, if the input quantum state is ρ_{in}, then the output quantum state is:

$$\rho_{out} = pZ\rho_{in}Z^\dagger + (1-p)\rho_{in} \tag{1}$$

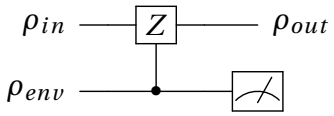
(a) Suppose $\rho_{in} = |\psi\rangle\langle\psi|$ where $\left|\psi\right\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. What is ρ_{out}? Write your answer as a 2×2 density matrix, in terms of parameter p.

(b) The effect of an error on a quantum system can be mathematically represented by a quantum channel, $\mathscr{E}(\rho)$, defined as

$$\mathscr{E}(\rho) = \sum_k E_k \rho E_k^\dagger$$

where the E_k are called the Kraus operators satisfying the condition $\sum_k E_k^\dagger E_k = I$. Give the set of two Kraus operators $\{E_0, E_1\}$ describing the phase flip channel.

(c) Recall from lecture that one can model such error as an interaction of the quantum system with an environment. The Kraus operators will then arise from tracing out the environment. To see this, we construct a quantum circuit model for the phase flip channel as follows:



Here, the top qubit is the input quantum system, and the bottom qubit is the environment. Give the environment state ρ_{env} such that the circuit is equivalent to the phase flip channel on the top qubit ρ_{in}. Write ρ_{env} as a density matrix in terms of the parameter p. (Hint: we want to initialize ρ_{env} such that a Z gate is performed on ρ_{in} with probability p.)

3 Dephasing Channel

Task 3.1 (30 pts)

Dephasing channel is an important physical process in quantum systems; it does not make any transitions in the $\{|0\rangle, |1\rangle\}$ basis, but instead changes the relative phase between $|0\rangle$ and $|1\rangle$.

In the operator sum representation, we consider for a single-qubit input state ρ, a dephasing channel is defined as

$$\mathscr{E}(\rho) = E_0 \rho E_0^\dagger + E_1 \rho E_1^\dagger$$

where $E_0 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-\lambda} \end{bmatrix}, E_1 = \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{\lambda} \end{bmatrix}$.

(a) Consider the input quantum state $\rho_{in} = |+\rangle\langle+|$. Give the output quantum state $\rho_{out} = \mathscr{E}(\rho_{in})$. Write your answer as a 2×2 matrix in terms of λ.

(b) It turns out we can write the two Kraus operators as a linear combination of the Pauli operators, I and Z. In particular, find the coefficients, a, b, c, d, such that $E_0 = aI + bZ$ and $E_1 = cI + dZ$. Write your answers in terms of λ.

(c) Use your solution to part (b) to rewrite the dephasing channel. Show that you can write $\mathscr{E}(\rho) = E_0 \rho E_0^\dagger + E_1 \rho E_1^\dagger = pZ\rho Z^\dagger + (1-p)\rho$ for some p. Write p in terms of λ.

(d) (**Optional**) In real quantum systems, dephasing occurs continuously in time. Imagine r is the rate of dephasing, so the dephasing parameter $\lambda = r\Delta T \ll 1$ for some small interval of time ΔT. At time $T = n\Delta T$, we can model the resulting quantum state as applying the dephasing channel n times: $\mathscr{E}^n(\rho)$. Consider the input quantum state $\rho_{in} = |+\rangle\langle+|$. To model a continuous dephasing process up to some constant time T, we divide T into $n \to \infty$ segments, each applied with $\mathscr{E}$ for infinitesimal duration ΔT. Give the output quantum state at some fixed time T: $\rho_{out}(T) = \mathscr{E}^n(\rho_{in})$, for $n \to \infty$. Write your answer as a 2×2 matrix for ρ_{out} in terms of r and T.

4 7-Bit Hamming Code

Task 4.1 ($\star$ pts)

Recall from lecture, we define a classical linear code using a $k \times n$ generator matrix G and a $(n-k) \times n$ parity check matrix H to encode k bits of information using n bits. The codewords are defined as the column vectors $\vec{x}_v = G^T \vec{v}$, where $\vec{v}$ is the column vector representing a k-bit integer, ranging from 0 to $2^k - 1$. Note that all arithmetic operations are done with modulo 2. Errors can be detected by performing parity checks specified by the rows of a $(n-k) \times n$ matrix H. A good parity check matrix satisfies the condition that $H\vec{x} = 0$ for all codewords $\vec{x}$.

For example, the Hamming code encodes $k = 4$ bits using $n = 7$ bits. The generator matrix G is defined as

$$G = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

The parity check matrix H is defined as

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

(a) Give the column vector for the codeword $\vec{x}_0 = G^T\vec{0}$. Here $\vec{0} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^T$.

(b) Give the column vector for the codeword $\vec{x}_3 = G^T\vec{3}$. Here $\vec{3} = \begin{bmatrix}0 & 0 & 1 & 1\end{bmatrix}^T$.

(c) Give the column vector for the codeword $\vec{x}_{15} = G^T\vec{15}$. Here $\vec{15} = \begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix}^T$.

(d) Verify that $H\vec{x}_{15} = 0$.

(e) Consider an error $\vec{e}$ which occurs to one of the codewords, resulting in

$$\vec{y} = \vec{x} + \vec{e} = \begin{bmatrix}0 & 1 & 1 & 0 & 0 & 1 & 0\end{bmatrix}^T.$$

What are the results of the parity checks: $H\vec{y}$?

(f) Consider an error $\vec{e}$ which occurs to one of the codewords, resulting in $\vec{y} = \vec{x} + \vec{e}$. Suppose the results of the parity checks: $H\vec{y} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^T$. If we assume that $\vec{e}$ is a single bit-flip, can we locate the bit flip? If so, give $\vec{e}$. If not, explain why.

5 7-Qubit Steane Code

Task 5.1 (50 pts)

The 7-qubit code (also known as Steane code) is constructed from the classical 7-bit Hamming code C. The Steane code encodes 1 qubit of information using 7 qubits. In this question, we will explore this construction and its properties.

(a) For each codeword x of C, we define a 7-qubit quantum state $|x\rangle$. For example, if $x = \begin{bmatrix}1 & 1 & 0 & 0 & 1 & 1 & 0\end{bmatrix}^T$, then $|x\rangle = |1100110\rangle$. For each row h of the parity check matrix H, we define a Z-type stabilizer operator S_h. For example, if $H_h = \begin{bmatrix}1 & 1 & 1 & 1 & 0 & 0 & 0\end{bmatrix}$, then $S_h = Z\otimes Z\otimes Z\otimes Z\otimes I\otimes I\otimes I$. Show that $|1100110\rangle$ is a $(+1)$-eigenstate of all three stabilizers. That is, $S_h|1100110\rangle = |1100110\rangle$ for each S_h.

(b) A single-qubit bit-flip error can be diagnosed by the above stabilizers. Consider a non-codeword $\big|y\big\rangle = |0110010\rangle$. Show that $\big|y\big\rangle$ is a (-1)-eigenstate of at least one of the stabilizers.

(c) In order to also diagnose phase flip errors, we group the codewords into two subsets: $E = \{|x\rangle : x \in C \text{ and } x \text{ has even number of } 1\}$ and $O = \{|x\rangle : x \in C \text{ and } x \text{ has odd number of } 1\}$. List the elements in E and O respectively.

(d) We define the two codewords of the 7-qubit Steane code as follows:

$$|0\rangle_L = \frac{1}{\sqrt{8}}\sum_{x\in E}|x\rangle, \qquad |1\rangle_L = \frac{1}{\sqrt{8}}\sum_{x\in O}|x\rangle$$

We also define an additional set of stabilizers: For each row h of the parity check matrix H, we define an X-type stabilizer operator S'_h. For example, if $H_h = \begin{bmatrix}1 & 1 & 1 & 1 & 0 & 0 & 0\end{bmatrix}$, then $S'_h = X\otimes X\otimes X\otimes X\otimes I\otimes I\otimes I$. Show that $S'_h|0\rangle_L = |0\rangle_L$ and $S'_h|1\rangle_L = |1\rangle_L$ for all three S'_h.

(e) Consider an initial quantum state $\big|\psi\big\rangle_L = \alpha|0\rangle_L + \beta|1\rangle_L$. Due to some single-qubit bit-flip or phase flip error, the quantum state results in $\big|\psi'\big\rangle$. Suppose the stabilizer syndrome measurements have the following outcome:

$$S_0\big|\psi'\big\rangle = \big|\psi'\big\rangle, S_1\big|\psi'\big\rangle = \big|\psi'\big\rangle, S_2\big|\psi'\big\rangle = \big|\psi'\big\rangle, S'_0\big|\psi'\big\rangle = \big|\psi'\big\rangle, S'_1\big|\psi'\big\rangle = -\big|\psi'\big\rangle, S'_2\big|\psi'\big\rangle = \big|\psi'\big\rangle.$$

What was the error?

PROGRAMMING PORTION
This portion of the assignment has a total of 20 *points.*
Important: *This portion is optional for CPSC 547 students.*

In lectures, we discussed quantum error correction and how to correct general quantum errors by modeling them with Pauli errors. In this programming task, you'll first write a simulator (Task 6) that computes the measurement results of a quantum circuit with only Clifford gates, Pauli $\hat{X}$ and CNOT, together with initialization and measurement in the $\hat{Z}$ basis. Given some (random) Pauli $\hat{X}$ errors at specific positions throughout the circuit, your simulator will generate the measurement results stored as classical bits. Next, you'll implement a QEC decoder (Task 7) that takes the measurement results as input and compute a most likely error pattern that generated this measurement result. This most likely error pattern can therefore be used to correct the physical qubit errors and protects the state of the logical qubit.

6 Clifford Gate Simulation

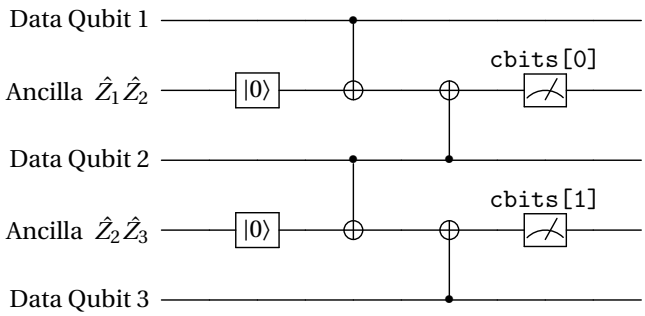


Figure 1: Quantum Repetition Code (single round of measurement, code distance $d = 3$).

According to the Gottesman–Knill theorem, Clifford gates can be efficiently simulated in polynomial time. This kind of simulation avoids keeping track of the exponential-sized quantum state vector and thus can simulate many more qubits than what you implemented in Assignment 1. Such simulation is especially helpful for analyzing quantum error correction codes.

We take the quantum repetition code as an example – it's implemented with CNOT gates, $|0\rangle$ initialization gates and $\hat{Z}$ measurement gates. A single measurement round of the quantum repetition code is shown in Figure 1. It consists of 4 steps:

- Step 0: initialize ancilla qubits to $|0\rangle$ states
- Step 1: simultaneous CNOT gates
- Step 2: simultaneous CNOT gates
- Step 3: measure ancilla qubits in $\hat{Z}$ basis (= measure joint $\hat{Z}_1\hat{Z}_2$ of adjacent data qubits)

We assume the circuit is initialized into a joint +1 eigenstate of all stabilizer (measurement) operators. Recall from lecture, this is called a logical codeword. Note that $|0_L\rangle$ and $|1_L\rangle$ form an orthonormal basis, i.e., $\langle 0_L|1_L\rangle = 0$. Any superposition of these two logical basis states is a logical state as well. Since quantum error correction does not obtain any information of the logical qubit, we cannot make any assumption of the logical state. Fortunately, knowing that the initial state is a joint +1 eigenstate is already enough for simulation.

To see how this is possible, we first look at two equivalent circuits in Figure 2. Only Pauli $\hat{X}$ error on the control qubit propagates to the target qubit. Thus, assuming there are some kind of errors before the CNOT gate, we can easily calculate the errors after the CNOT gate by commuting them through the CNOT gate.

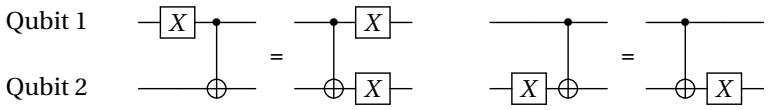


Figure 2: Equivalent Circuits.

Task 6.1 (10 pts)

Implement the function that computes the errors after gates:

- `x_error`: a Pauli X error. E.g. `before = [False]` then `after = [True]`, vice versa.
- `cx`: CNOT gate. E.g. `before = [True, False]` then `after = [True, True]`.
- `initialize`: initialize multiple qubits into their $|0\rangle$ states. `after = [False, ...]`.

```
class Gate(object):
    ...
    def commute_through(self, pauli_x_errors_before):
        # YOUR IMPLEMENTATION HERE
```

Task 6.2 (10 pts)

Implement the function that simulates one step. If this step is a normal gate, it updates the Pauli $\hat{X}$ errors `self.qubits.pauli_x_errors` by invoking the `commute_through` function. If this step is a measurement gate, it updates the corresponding classical bits `self.cbits.state` with new measurement: if it's +1 measurement then `False`, if it's -1 measurement then `True`. A -1 measurement implies some Pauli X errors in the circuit, which needs to be corrected.

```
class QuantumCircuit(object):
    ...
    def evolveOneStep(self):
        # YOUR IMPLEMENTATION HERE
```

You can check your implementation with public tests. One example test is shown in Figure 3:

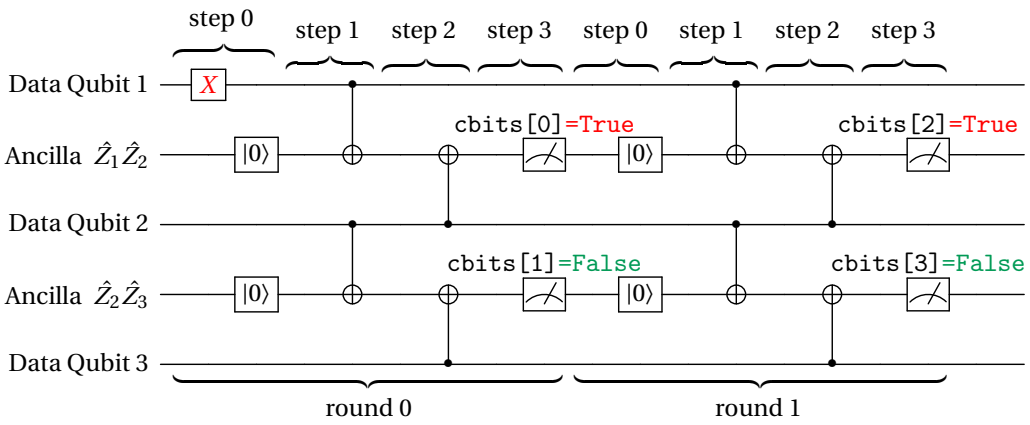


Figure 3: Basic example of simulation (2 rounds of measurements, each round has 4 steps).

7 Quantum Error Correction (QEC) Decoder

Given the simulator you've implemented in Task 6, you can simulate the measurement results given any random Pauli errors. In Task 7, you'll implement a decoder that takes the measurement results as input, and try to identify and correct the errors in the circuit.

Your decoder should be aware of the noise model, which defines in which position errors can happen, and at what probability. For simplicity, we consider three kinds of noise models.

- *Code-Capacity Noise Model:* each data qubit has $\hat{X}$ error in step 0 with probability p.
- *Phenomenological Noise Model:* multiple rounds of measurement, each applying the code-capacity noise model. Additionally, except for the last round, each stabilizer is subject to measurement error with probability p, i.e. the measurement result is flipped. Note that `noisy_measurements = 0` is equivalent to the code-capacity noise model.
- *Circuit-Level Noise Model:* errors can happen anywhere (not limited to data qubits) with probability p, except for the last round of measurement with `step > 0`.

Note that all three kinds of noise model assume a noiseless final measurement round. It's practical because the last measurement result usually comes from measuring all the data qubits and compute (classically) the stabilizer results directly from the data qubit measurements.

You'll implement a Most-Likely Error (MLE) decoder for each of these noise models. A MLE decoder computes the most likely error pattern that causes this syndrome (all the stabilizer measurement results). Assuming each error is independent, the probability of an error pattern $E \subseteq \{e\}$ is the product of each occurring error p_e where $e \in E$ and each non-occurring error $1-p_e$ where $e \notin E$.

$$P(E) = \prod_{e\in E} p_e \prod_{e\notin E} (1-p_e) = \prod_{e\in E} \frac{p_e}{1-p_e} \prod_e (1-p_e) \propto \prod_{e\in E} \frac{p_e}{1-p_e}$$

Thus, maximizing $P(E)$ is equivalent to maximizing $\prod_{e\in E} \frac{p_e}{1-p_e}$, which is also equivalent to minimizing $\sum_{e\in E} \log \frac{1-p_e}{p_e} = \sum_{e\in E} w_e$ where the weight is defined as $w_e = \log \frac{1-p_e}{p_e}$. Since our noise model only has single probability p_e, you can safely use $w_e = 1$. In conclusion, the decoding

problem can be formalized as: given any syndrome S, compute error pattern E

$$E = \arg\min_{E} \sum_{e \in E} w_e = \arg\min_{E} \sum_{e \in E} \ln \frac{1 - p_e}{p_e} \text{ , subject to } S(E) = S$$

Note that you don't have to implement an algorithm that solves the above mathematical problem. You only need to understand the concepts and convert the problem properly to implement the decoder. Install a library by "`pip3 install fusion-blossom==0.2.1`".

Task 7.1 (⋆ pts)

Implement a Most-Likely Error (MLE) decoder for code-capacity noise model.

```
class QuantumRepetitionCode(object):
    ...
    def decode_code_capacity_noise(self, p, measurements, visualize=False):
        # YOUR IMPLEMENTATION HERE
```

All the possible errors in the code-capacity noise model are shown in Figure 4.

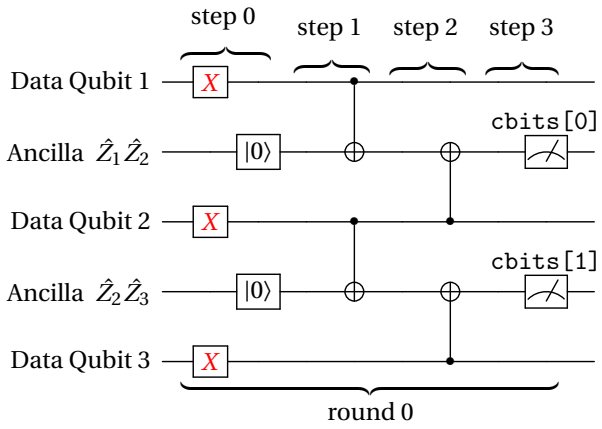


Figure 4: Code-Capacity Noise Model.

The measurement result is the parity check of the two adjacent data qubit errors. In order to solve the most likely error pattern that causes the measurement result, you can use the `fusion-blossom` library (https://github.com/yuewuo/fusion-blossom) to solve a Minimum-Weight Parity Subgraph (MWPS). Given any (non-negative weighted) graph, it solves a subgraph E (a subset of edges) that has minimum total weight ($\sum_{e \in E} w_e$) while satisfying the parity check:

- normal vertex: number of edges incident to this vertex is even. This is useful when describing `cbit = False`.
- defect vertex: number of edges incident to this vertex is odd. This is useful when describing `cbit = True`: odd number of independent errors causing this -1 stabilizer measurement.
- virtual vertex: number of edges incident to this vertex is arbitrary. This is useful when an independent error only causes a single defect vertex, in which case you can connect the other end to a virtual vertex.

We have created the vertices for you in the starter code. Here is the layout of vertices. You can modify `testBasicDecode()` function by changing `visualize=False` to `visualize=True` to open the visualization tool for ease of debugging[1]. The middle two white vertices corresponds to the measurement results (normal vertex or defect vertex), while the yellow ones are virtual vertices. You should first complete the decoding graph by adding edges to it. Each edge corresponds to an independent error (the three $\hat{X}$ errors in Figure 4). Each edge connects the 2 vertices that can detect the error. Specifically, you should append edges to `weighted_edges` list, with a format `(vertex_index_1, vertex_index_2, weight)`.

Figure 5: Decoding Graph of Code-Capacity Noise Model (vertex indices [0, 1, 2, 3] from left to right)

Then, try to convert the measurement results to defect vertices in their indices. For example, a measurement results of `[False, False]` corresponds to defect vertices `[]`. Similarly, `[True, False] -> [1]`, `[False, True] -> [2]`, `[True, True] -> [1,2]`. Defect vertices are marked as red circle in the visualization tool.

Finally, given the Minimum-Weight Parity Subgraph (MWPS) result `subgraph`, you'll calculate the correction on the data qubits. Specifically, `correction` is a list of bool value with length equal to the number of data qubits (or code distance `d` in quantum repetition code). A `True` means the corresponding data qubit should be corrected by $\hat{X}$. Given the indices of edges in `subgraph` (depending on how you add the edges), you can flip the value in the `correction`.

Once finished, you should pass the `testBasicDecode` test case.

Task 7.2 ($\star$ pts)

Implement a Most-Likely Error (MLE) decoder for phenomenological noise model.

```
class QuantumRepetitionCode(object):
    ...
    def decode_phenomenological_noise(self, p, measurements, visualize=False):
        # YOUR IMPLEMENTATION HERE
```

All the possible errors in the phenomenological noise model are shown in Figure 6.

[1] You need a browser to open the visualization tool. Although it's not necessary to finish the assignment, it helps in debugging.

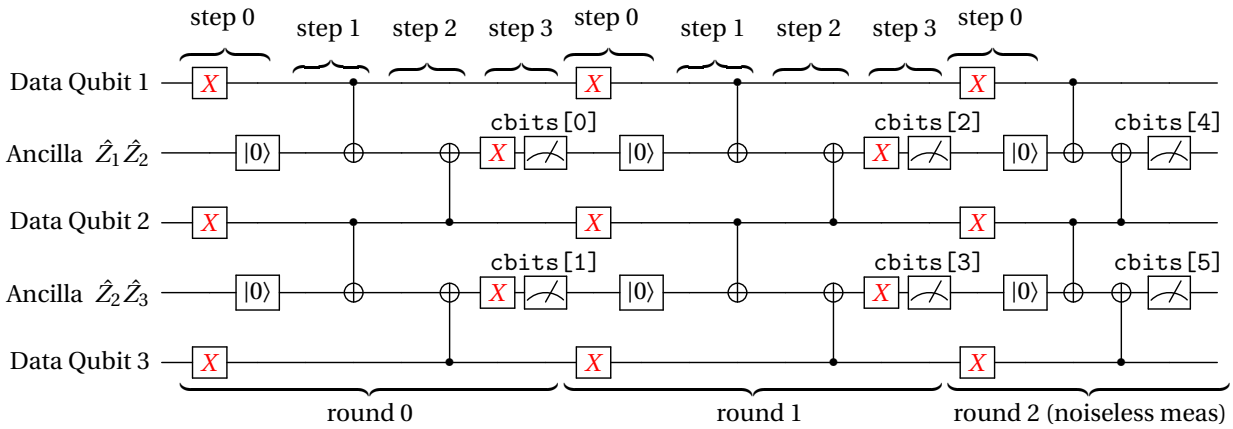


Figure 6: Phenomenological Noise Model (`noisy_measurement = 2`).

The phenomenological noise model adds pure measurement errors on top of the code-capacity noise model. In this case, the defect vertex does not necessarily corresponds to -1 stabilizer measurement (or `cbit = True`) because a single error will change all the measurement value afterwards. Thus, here we use defect vertex to represent measurement results that are *different* from the previous round. A pure measurement error appears as a pair of defect vertices because the measurement result will change twice.

We have built the vertices in the starter code, but feel free to modify it as you like. Similar to the previous task, you'll first complete the decoding graph as shown in Figure 7. Then create the defect vertices given the measurement results. Finally build the correction given the MWPS. Note that not all edges in the MWPS change the correction pattern, e.g. pure measurement errors does not affect data qubits.

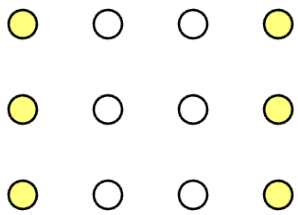


Figure 7: Decoding Graph of Phenomenological Noise Model (vertex indices [0, 1, 2, 3] at the bottom)

Once finished, you should pass the `testDecodePhenomenological` test case.

Task 7.3 (⋆ pts)

Implement a Most-Likely Error (MLE) decoder for circuit-level noise model.

```
class QuantumRepetitionCode(object):
    ...
    def decode_circuit_level_noise(self, p, measurements, visualize=False):
        # YOUR IMPLEMENTATION HERE
```

All the possible errors in the circuit-level noise model are shown in Figure 8.

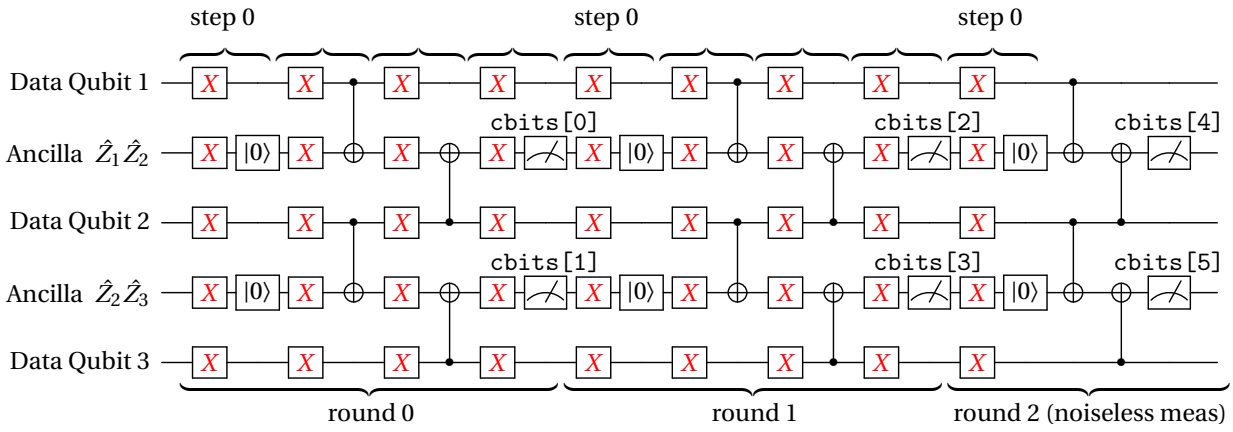


Figure 8: Circuit-Level Noise Model (`noisy_measurement = 2`).

Note that there are some errors that correspond to diagonal edges in the decoding graph. For example an error at round 0, step 2 on data qubit 2 generates defect vertices in round 0 at $\hat{Z}_1\hat{Z}_2$ stabilizer and round 1 at $\hat{Z}_2\hat{Z}_3$ stabilizer.

Task 7.4 (⋆ pts)

Evaluate decoder thresholds. In this task, you do not need to submit any results. You may need to modify your code to support different `noisy_measurements` and `d` (code distance) values. You'll get full score by achieving a reasonable logical error rate as shown in the following figures.

For the code-capacity noise model, it has an optimal threshold of $p_{\text{th}} = 0.5$. That is, given any physical error rate below the threshold, one can always achieve exponentially lower logical error rate by increasing the code distance `d`. Use the `evaluateDecoderThreshold` function to evaluate your decoder. An example is in Figure 9.

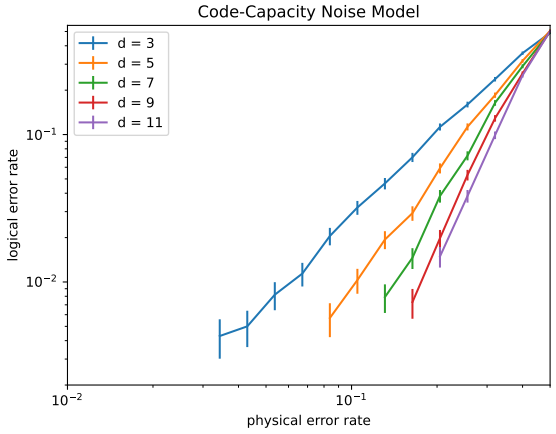
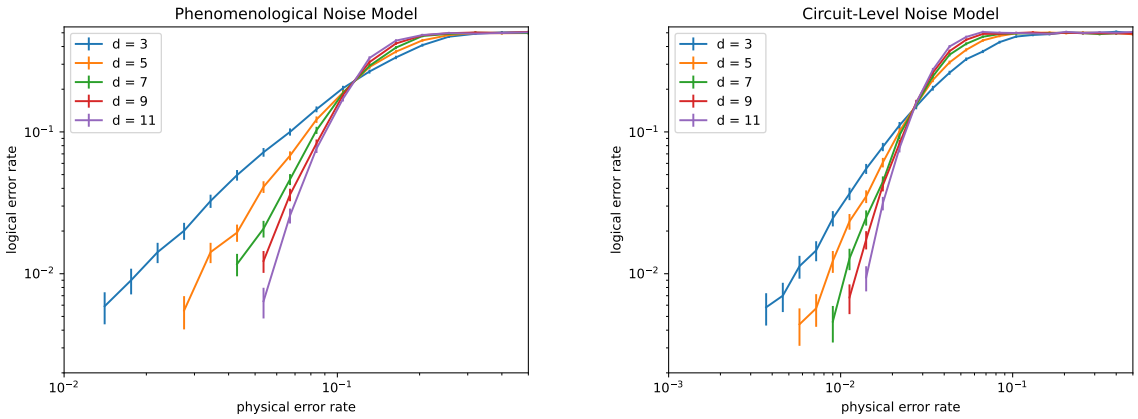


Figure 9: Code Capacity Noise Decoder

For the phenomenological noise model and circuit-level noise model, it has lower threshold. Use the `evaluateDecoderThresholdPhenomenological/CircuitLevel` functions to evaluate your decoder. Examples are shown in Figure 10a and Figure 10b.



(a) Phenomenological Noise Decoder

(b) Circuit-Level Noise Decoder

Task 7.5 (⋆ pts)

Understand logical error rate. In the previous evaluation, we use `noisy_measurements = d` to simulate a minimum fault-tolerant logical state preparation. In reality, the logical qubit should last for much longer than the lifetime of a physical qubit. When the physical error rate is below the threshold, we can increase the code distance to achieve exponentially longer logical qubit lifetime. Use the `evaluateLogicalErrorAccumulation` function to understand how logical qubit's error rate changes with time (rounds of noisy measurements). We also plot the physical qubit's error accumulation labeled in `d = 1`. An example is in Figure 11.

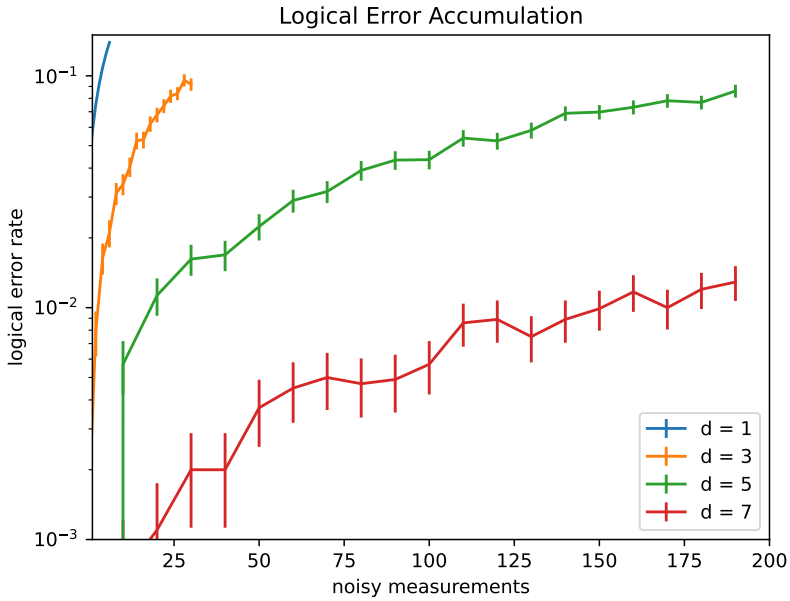


Figure 11: Logical Error Accumulation in Phenomenological Noise Model with $p = 0.02$