Quantum Error Correction

Contents

1 The problem statement **1**

2 Literature overview **1**

3 Initial thoughts **2**

4 Implementation notes **3**

4.1 Setup 4
4.2 Overview 4
4.3 Base Types 4
4.3.1 Quantum Operations 4
4.3.2 Quantum circuits 5
4.3.3 Mapping circuits 5
4.4 Quantum error correction mappings 6
4.4.1 Bitflip code 6
4.4.2 Surface25u surface code 7

1 The problem statement

This document contains the solution for the surface code quantum error correction problem. Sergei's questions are rendered in dark green color. The project repository is hosted on Github[1].

Your task is to **set up a distance 3 surface code using PennyLane**. If you are not familiar with Surface Codes, this may be a very useful resource: `https://arxiv.org/pdf/1404.3747`. Try to simulate **at least one cycle of the quantum error correction scheme** and give a quick interpretation of the results. Try **adding some noise to the circuit** (for example, add a random bit-flip in the circuit), and see what happens to the measurement. You should **describe in words how the decoding would happen**. If you have the extra time, feel free to also implement some **simple decoding protocols**. The expected output of this exercise is a Jupyter Notebook. But feel free to deliver your answer in whichever medium you see fit. Most importantly, try to learn about quantum error correction and give us an insight into the way you tackle difficult, unseen problems.

2 Literature overview

- Low-distance Surface Codes under Realistic Quantum Noise by Yu Tomita and Krysta M. Svore (2014). References:
  - [2] *(***Brune2020***,* **Brune2020***)* *"***Brune2020***"* [**Brune2020**]

[1]Github repo: `https://github.com/sergei-mironov/qecsurface`

  - `[6]` Surface codes: Towards practical large-scale quantum computation by Austin G. Fowler (2012)

- Wiki about the Shor code
- IQC 2024 Lecture 29 Quantum Error Correction: Surface Codes
- Coursera course Hands-on quantum error correction
  - Topological Code Autotune by Austin G. Fowler (2012)
- Decoding algorithms for surface codes by lOlius (2024)
  - `50` Realizing Repeated Quantum Error Correction in a Distance-Three Surface Code
  - `51` Suppressing quantum errors by scaling a surface code logical qubit
- *(2011, Nielsen and Chuang) "Quantum Computation and Quantum Information: 10th Anniversary Edition"* [1]

3 Initial thoughts

In this section, we summarize our current understanding of the problem domain.

Quantum error correction codes

Quantum error correction codes (QECC) are techniques used to protect quantum information from errors due to decoherence, noise, and other quantum imperfections. They work by encoding logical quantum bits (qubits) into a larger number of physical qubits, allowing for the detection and correction of errors that can occur during quantum computation or storage.

Common error correction codes

1. **Bit-flip code:** One of the simplest quantum error correction codes, the bit-flip code protects against bit-flip errors by encoding each logical qubit using three physical qubits.
2. **Phase-flip code:** Similar to the bit-flip code, the phase-flip code protects against phase-flip errors by encoding each logical qubit using three physical qubits.
3. **Shor code:** The Shor code is a 9-qubit code that can protect against both bit-flip and phase-flip errors, essentially combining the bit-flip and phase-flip codes.
4. **Steane code:** The Steane code is a 7-qubit code that can correct arbitrary single-qubit errors. It is based on classical error-correcting codes and is more efficient than the Shor code.
5. **Five-qubit code:** Also known as the perfect code, this is the smallest possible code that can correct arbitrary single-qubit errors, using just five physical qubits to encode one logical qubit.
6. **Surface codes:** Surface codes are defined on a lattice and are known for their high threshold for error tolerance and efficient implementation.

Question 1: One thing that is not very clear to me is: what overall usage scheme should I aim for? I saw two candidates.

- Receive a 1-qubit quantum state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ as input. Encode it using QECC into the superposition of logical states $|\psi_L\rangle = \alpha|0_L\rangle + \beta|1_L\rangle$. Pass it through a noisy quantum channel and decode it back into the original $|\psi\rangle$.
- Initialize the logical qubit with the logical zero state $|0_L\rangle$. Apply logical operations such as X_L, Z_L, or others (TODO: specify which ones exactly) to perform fault-tolerant computations, e.g., obtain a desired $|\psi_L\rangle$. Measure it and interpret the results in an algorithm-specific way.

Code space

Code space refers to the subspace of a quantum Hilbert space in which the logical qubits are encoded. We assume that every practical QECC has a code space large enough to encode at least one qubit, that is, the $\mathbb{H}^{\otimes 1}$.

For example, the code space of the Stean code is spanned across the following vectors

$$|0_L\rangle = \frac{1}{\sqrt{8}}(|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle + |0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle)$$

$$|1_L\rangle = \frac{1}{\sqrt{8}}(|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle + |1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle)$$

Stabilizer codes

The stabilizer formalism is a framework used in quantum error correction to describe quantum states and codes. It is based on the concept of stabilizer groups, which are sets of commuting operators from the Pauli group that stabilize (or leave unchanged) a particular subspace of a quantum system's Hilbert space, known as the code space.

In the context of a quantum error-correcting code (QECC), stabilizer formalism describes logical states such as $|0_L\rangle$ and $|1_L\rangle$ of a QECC by specifying the stabilizers that leave these states unchanged. These logical states are defined as the common $+1$ eigenstates of the stabilizer generators.

The **Stabilizer Theorem** provides an important property that helps define valid stabilizer codes: A valid stabilizer code must have a stabilizer group composed of operators that do not include the negative identity operator $-I$ as an element.

- Every element of the stabilizer group must be a Hermitian operator with eigenvalues ± 1. The presence of $-I$ would imply that the eigenvalue -1 is always included, which conflicts with the requirement that the identity operator I (with eigenvalue $+1$) must be an element of the group.
- The requirement to exclude $-I$ ensures the stabilizer group forms a valid subgroup of the Pauli group, allowing the stabilizer code to properly define the code space and detect/correct errors without inherent contradictions.

4 Implementation notes

- ~~Unfortunately, it seems that Pennylane doest not have mid-circuit qubit resets yet.~~ V 0.40.40 says `qml.measure` has a `reset=True` parameter (link).

4.1 Setup

The project could be cloned from the GitHub repository https://github.com/sergei-mironov/qecsurface. The repository contains sourceable file `env.sh` which adjusts `PATH` and `PYTHONPATH` environment variables to make project specific shell-scripts and Python modules available.
Overall, the typical initizlization command sequence is follows:

```
Shell

$ git clone https://github.com/sergei-mironov/qecsurface
$ cd qecsurface
$ . env.sh
$ ipython # Enter IPython interactive shell
>>> from quecsurface import *
>>> from quecsurface_tests import *
```

4.2 Overview

The qecsurface project defines the following main Python modules:

- `qecsurface.type` defines a minimalistic domain-specific language for quantum circuits modeling. The main data structure for circuits is named *FTCircuit* where FT stands for fault-tolerant.
- `qecsurface.pennylane` defines routines which lower *FTCircuits* to PennyLane circuits.
- Finally, `qecsurface.qeccs` defines mappings from *FTCircuit* → *FTCircuit* which implements various specific Quantum error correction schemes.

4.3 Base Types

4.3.1 Quantum Operations

```
class OpName(Enum):
  """ Quantum operation labels """
  I = 0
  X = 1
  Z = 2
  H = 3

def opname2str(n:OpName)->str:
  return {OpName.I:'I', OpName.H:'H', OpName.Z:'Z', OpName.X:'X'}[n]

# Common type alias for quantum operations
type FTOp[Q] = Union["FTInit[Q]", "FTPrim[Q]", "FTCond[Q]", "FTCtrl[Q]", "FTMeasure[Q]",
                     "FTErr[Q]"]

@dataclass
class FTInit[Q]:
  """ Primitive quantum operation acting on one qubit. """
  qubit:Q
  alpha:complex
  beta:complex

@dataclass
class FTPrim[Q]:
  """ Primitive quantum operation acting on one qubit. """
  name:OpName
  qubits:list[Q]

@dataclass
class FTCtrl[Q]:
  """ Quantum control operation acting on two qubits. """
  control:Q
  op:FTOp[Q]

# Syndrome test measurement label encodes a layer, the syndrome type and the qubit labels
type MeasureLabel[Q] = tuple[int,OpName,tuple[Q,...]]

@dataclass
class FTMeasure[Q]:
  """ Quantum measure oprtation which acts on a `qubit`. Measurement result is assiciated with a
  `label`. """
  qubit:Q
  label:MeasureLabel[Q]

@dataclass
class FTCond[Q]:
  """ A quantum operation applied if a classical condition is met. """
  cond:Callable[[dict[MeasureLabel[Q],int]],bool]
  op:FTOp[Q]

@dataclass
class FTErr[Q]:
  """ Apply an error to `phys` physical qubit constituting the logical qubit Q. """
  qubit:Q
  phys:int
  name:OpName
```

4.3.2 Quantum circuits

```
# Common type alias for quantum circuits, where Q is type of qubit label.
type FTCircuit[Q] = Union["FTOps[Q]", "FTComp[Q]"]

@dataclass
class FTOps[Q]:
  """ A primitive circuit consisting of a tape of operations. """
  ops:list[FTOp[Q]]

@dataclass
class FTComp[Q]:
  """ Composition of circuits, also known as circuit tensor product. """
  a: FTCircuit[Q]
  b: FTCircuit[Q]
```

4.3.3 Mapping circuits

The main operation defined on $FTCircuit$ is

$$map\_circuit : Map_{a,b} \to FTCircuit_a \to FTCircuit_b$$

The subsequent quantum error correction algorithms are defined as a derived classes of the $Map_{a,b}$ base class.

```
@dataclass
class Map[Q1,Q2]:
  """ Map circuit handler base class. """
  def map_op(self, op:FTOp[Q1]) -> FTCircuit[Q2]:
    """ Maps an operation of the source circuit into the destination circuit """
    raise NotImplementedError


def map_circuit[Q1,Q2](c:FTCircuit[Q1], m:Map[Q1,Q2]) -> FTCircuit[Q2]:
  def _traverse_op(op:FTOp[Q1], acc) -> None:
    return FTComp(acc, m.map_op(op))
  return traverse_circuit(c, _traverse_op, FTOps([]))
```

4.4 Quantum error correction mappings

4.4.1 Bitflip code

```
@dataclass
class Bitflip[Q1,Q2](Map[Q1,Q2]):
  """ Maps quantum circuit into a quantum circuit with Bitflip quantum error correction. """
  qmap:dict[Q1,tuple[list[Q2],list[Q2]]]
  layer:int = 0

  def _next_layer(self) -> int:
    l = self.layer
    self.layer = self.layer + 1
    return l

  def _error_correction_cycle(self, q:Q1) -> FTCircuit[Q2]:
    qubits, syndromes = self.qmap[q]
    layer = self._next_layer()
    det = bitflip_detect(qubits, syndromes, layer)
    corr = bitflip_correct(qubits, layer)
    return FTComp(det,corr)

  def map_op(self, op:FTOp[Q1]) -> FTCircuit[Q2]:
    qmap = self.qmap
    acc = []
    if isinstance(op, FTInit):
      q = op.qubit
      acc.append(FTOps([FTInit(qmap[q][0][0], op.alpha, op.beta)]))
      acc.append(bitflip_encode(qmap[q][0][0], qmap[q][0]))
    elif isinstance(op, FTErr):
      q = op.qubit
      qubits = qmap[q][0]
      equbit = qubits[op.phys % len(qubits)]
      acc.append(FTOps([FTPrim(op.name, [equbit])]))
      acc.append(self._error_correction_cycle(q))
    elif isinstance(op, FTPrim):
      for q in op.qubits:
        qubits = qmap[q][0]
        if op.name == OpName.I:
          pass
        elif op.name == OpName.X:
          acc.append(FTOps([FTPrim(OpName.X, qubits)]))
        elif op.name == OpName.Z:
          acc.append(FTOps([FTPrim(OpName.Z, qubits)]))
        else:
          raise ValueError(f"Bitflip qecc: Unsupported primitive operation: {op}")
        acc.append(self._error_correction_cycle(q))
    else:
      raise ValueError(f"Bitflip qecc: Unsupported operation: {op}")
    return reduce(FTComp, acc)
```

4.4.2 Surface25u surface code

```
@dataclass
class Surface25u[Q1, Q2](Map[Q1, Q2]):
  qmap: dict[Q1, tuple[list[Q2], Q2]]
  layer: int = 0
  layers0: dict[Q1, int] = field(default_factory=dict)
  mls: dict[Q1, list[MeasureLabel]] = field(default_factory=lambda: defaultdict(list))

  def _next_layer(self) -> int:
    l = self.layer
    self.layer += 1
    return l

  def _error_correction_cycle(self, q:Q1, layer0:int) -> FTCircuit[Q2]:
    qubits, syndrome = self.qmap[q]
    layer = self._next_layer()
    det, ml = surface25u_detect(qubits, [syndrome], layer)
    corr = surface25u_correct(qubits, layer0, layer)
    self.mls[q].append(ml)
    return FTComp(det,corr)

  def map_op(self, op: FTOp[Q1]) -> FTCircuit[Q2]:
    qmap = self.qmap
    layers0 = self.layers0
    acc = []
    if isinstance(op, FTInit):
      if (op.alpha, op.beta) != (1.0, 0.0):
        raise ValueError(f"Surface25u could only be initialized with |0> (got {op})")
      q = op.qubit
      qubits, syndrome = qmap[q]
      layers0[q] = self._next_layer()
      c, ml = surface25u_detect(qubits, [syndrome], layers0[q])
      self.mls[q].append(ml)
      acc.append(c)
    elif isinstance(op, FTErr):
      q = op.qubit
      if layers0.get(q) is None:
        raise ValueError(f"Surface25u: qubit {q} was not initialized")
      qubits,_ = qmap[q]
      equbit = qubits[op.phys % len(qubits)]
      acc.append(FTOps([FTPrim(op.name, [equbit])]))
      acc.append(self._error_correction_cycle(q, layers0[q]))
    elif isinstance(op, FTPrim):
      for q in op.qubits:
        if layers0.get(q) is None:
          raise ValueError(f"Surface25u: qubit {q} was not initialized")
        qubits,_ = qmap[q]
        if op.name == OpName.I:
          pass
        elif op.name == OpName.X:
          acc.append(FTOps([FTPrim(op.name, [qubits[1], qubits[6], qubits[11]])]))
        elif op.name == OpName.Z:
          acc.append(FTOps([FTPrim(op.name, [qubits[5], qubits[6], qubits[7]])]))
        else:
          raise ValueError(f"Surface25u: Unsupported logical operation: {op}")
        acc.append(self._error_correction_cycle(q, layers0[q]))
    else:
      raise ValueError(f"Surface25u: Unsupported operation: {op}")
```

References

[1] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011. ISBN: 1107002176.