Quantum Error Correction

Contents

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.

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

2 Literature overview

- *(2014, Tomita and Svore) "Low-distance surface codes under realistic quantum noise"* [3]. References:
  - [2] *(2020, Brun) "Quantum Error Correction"* [1]
  - [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"* [2]

3 Some theory

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.

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

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

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

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.

3.1 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 *
```

3.2 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` introduces $FTCircuit \to FTCircuit$ mappings which implement various Quantum error correction schemes.

3.3 Base Types

3.3.1 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]
```

```python
# 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
```

3.3.2 Quantum circuits

```python
# 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]
```

3.3.3 Mapping form circuit to circuit

The map operation is defined on $FTCircuit_q$ is

$$\forall a, b \;.\; map\_circuit : Map_{a,b} \rightarrow FTCircuit_a \rightarrow FTCircuit_b$$

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

```python
@dataclass
class Map[Q1,Q2]:
  """ Base class for stateful circuit mapping algorithms. """
  def map_op(self, op:FTOp[Q1]) -> FTCircuit[Q2]:
    """ Maps an operation of a source circuit into a destination circuit. """
    raise NotImplementedError
```

```
def map_circuit[Q1,Q2](c:FTCircuit[Q1], m:Map[Q1,Q2]) -> FTCircuit[Q2]:
  """ Maps the circuit `c` by mapping each its operation and taking a compostion """
  def _traverse_op(op:FTOp[Q1], acc) -> None:
    return FTComp(acc, m.map_op(op))
  return traverse_circuit(c, _traverse_op, FTOps([]))
```

3.3.4 Example: Quantum teleportation

```
circuit_ft = FTComp(
  FTOps([
    FTInit(qubit=0, alpha=1.0, beta=0.0),          # Initialize the qubit in a known state
    FTPrim(OpName.H, [1]),                         # Apply Hadamard to qubit 1
    FTCtrl(control=1, op=FTPrim(OpName.X, [2])),   # Entangle qubits 1 and 2
    FTCtrl(control=0, op=FTPrim(OpName.X, [1])),   # Bell state preparation
    FTMeasure(qubit=0, label="m0"),
    FTMeasure(qubit=1, label="m1")
  ]),
  FTComp(
    FTOps([
      FTCond(lambda m: m["m0"] == 1, FTPrim(OpName.X, [2]))  # Conditional X based on m0
    ]),
    FTOps([
      FTCond(lambda m: m["m1"] == 1, FTPrim(OpName.Z, [2]))  # Conditional Z based on m1
    ])
  )
)
cPL = to_pennylane_mcm(circuit_ft)
qml.draw_mpl(cPL)()
plt.savefig('../img/teleport.png')
```

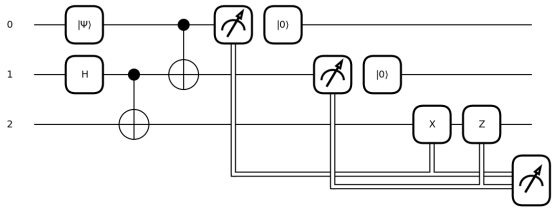

Figure 1: Quantum teleportation circuit.

3.3.5 Example: 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
  def _error_correction_cycle(self, q:Q1) -> FTCircuit[Q2]
  def map_op(self, op:FTOp[Q1]) -> FTCircuit[Q2]
```

3.4 Quantum Error Correction with Surface25u

3.4.1 Overview

We implement the Surface25 (unified) quantum error correction code following [3]. We consider syndrome qubits ideal so we unify them as a single model qubit to speed up the computations.

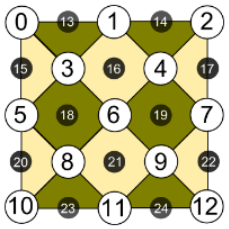

Figure 2: Surface 25 quantum error correction code.

3.4.2 Hadamard tests

The backbone of the suface quantum error correction codes are Hadamard tests projecting the quantum state to one of stabilizer eigenstates and revealing its signs. We define these tests as follows:

```
def stabilizer_test_X[Q](tile:FTPrim[Q], syndrome:Q, ml:MeasureLabel[Q]) -> FTCircuit[Q]:
  """ Define a stabilizer X-test circuit. """
  assert tile.name == OpName.X, tile
  return FTOps([
    FTPrim(OpName.H, [syndrome]),
    *[FTCtrl(control=syndrome, op=FTPrim(OpName.X, [qubit_label]))
      for qubit_label in tile.qubits
    ],
    FTPrim(OpName.H, [syndrome]),
    FTMeasure(qubit=syndrome, label=ml)
  ])

def stabilizer_test_Z[Q](tile:FTPrim[Q], syndrome:Q, ml:MeasureLabel[Q]) -> FTCircuit[Q]:
  """ Define a stabilizer Z-test circuit. """
  assert tile.name == OpName.Z, tile
  return FTOps([
    *[FTCtrl(control=qubit_label, op=FTPrim(OpName.X, [syndrome]))
      for qubit_label in tile.qubits
    ],
    FTMeasure(qubit=syndrome, label=ml)
  ])
```

3.4.3 Error correction code primitives

We split the implementation in several functions:

- The Surface25 stabilizer layout is defined as a constant:

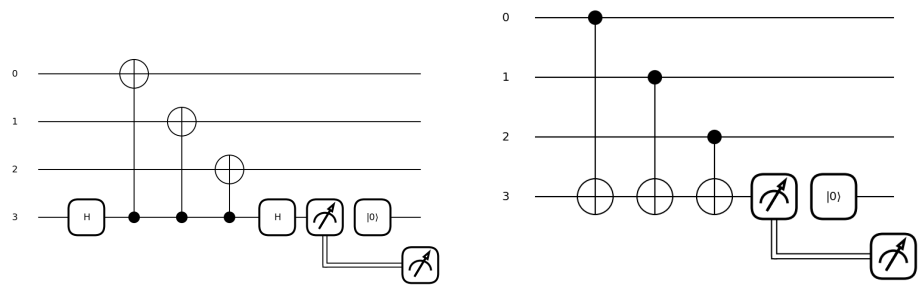

Figure 3: Stabilizer X-test and stabilizer Z-test.

```
def surface25_stabilizers() -> list[FTPrim[int]]:
  """ Surface25 stabilizers. Qubit labels may be interpreted as indices. """
  def X(data):
    return FTPrim(OpName.X, data)
  def Z(data):
    return FTPrim(OpName.Z, data)
  return [
            X([0,1,3]), X([1,2,4]),
    Z([0,3,5]), Z([1,3,4,6]), Z([2,4,7]),
           X([3,5,6,8]), X([4,6,7,9]),
    Z([5,8,10]), Z([6,8,9,11]), Z([7,9,12]),
           X([8,10,11]), X([9,11,12]),
  ]
```

- The detector function converts the stabilizer layout into a sequence of Hadamard tests and returns syndrome measurement labels along with the resulting circuit.

```
def surface25u_detect[Q](
  data: list[Q], syndromes: list[Q], layer:int=0
) -> tuple[FTCircuit[Q],list[MeasureLabel]]:
  """ Build the surface25u error detection circuit. Return the circuit alongside with a list of
  mid-circuit measurement labels. """
  assert len(data) == 13, f"Expected 13 data qubit labels, got {data}"
  assert len(syndromes) == 1, f"Expected 1 syndrome qubit label, got {syndrome}"
  syndrome = syndromes[0]
  labels = []

  def _to_hadamard_test(op):
    qubits = [data[q] for q in op.qubits]
    labels.append((layer,op.name,tuple(qubits)))
    if op.name == OpName.X:
      return stabilizer_test_X(FTPrim(OpName.X, qubits), syndrome, labels[-1])
    elif op.name == OpName.Z:
      return stabilizer_test_Z(FTPrim(OpName.Z, qubits), syndrome, labels[-1])
    else:
      raise ValueError(f"Unrecognized op {op}")

  return reduce(FTComp, map(_to_hadamard_test, surface25_stabilizers())), labels
```

- The correction implements a simple decoding protocol to fix any single-qubit errors.

```
def surface25u_correct[Q](data:list[Q], layer0:int, layer:int) -> FTCircuit[Q]:
  """ Build the surface25u error correction circuit assuming `layer` measurememnts are
  ↪  available.
  Use `layer0` measurements as a reference. """
  def _corrector(op, opc, d):
    def _cond(msms):
      neigb = [l[2] for l in list(msms.keys()) if l[0]==layer and l[1]==op and (d in l[2])]
      others = [l[2] for l in list(msms.keys()) if l[0]==layer and l[1]==op and (d not in
      ↪  l[2])]
      return reduce(
        lambda a,b: a & b, [
          *[(msms[(layer0,op,n)] != msms[(layer,op,n)]) for n in neigb],
          *[(msms[(layer0,op,n)] == msms[(layer,op,n)]) for n in others],
        ]
      )
    return FTCond(_cond, FTPrim(opc,[d]))
  return FTComp(
    FTOps([_corrector(OpName.X, OpName.Z, d) for d in data]),
    FTOps([_corrector(OpName.Z, OpName.X, d) for d in data])
  )
```

- In addition, we define an utility syndrome printing function

```
def surface25u_print2(msms:dict[MeasureLabel,int], flt:list[MeasureLabel], ref_layer:int=0)
```

3.4.4 Example: full error correction cycle

We demonstrate the usage with the following code. First, we initialize the $|0_L\rangle$ state by passing the $|0\rangle$ through the detection circuit and measuring stabilizer signs. Next, we introduce an error on the sixth qubit. Finally, we apply the error correction cycle where we use the stabilizer signs obtained initially.

```
data, syndrome = list(range(13)), [13]
layer0, layer1, layer2 = 0, 1, 2
init, ml1 = surface25u_detect(data, syndrome, layer0)
err = FTOps([FTPrim(OpName.H,[6])])
detect, ml2 = surface25u_detect(data, syndrome, layer1)
correct = surface25u_correct(data, layer0, layer1)
cPL = to_pennylane_mcm(reduce(FTComp, [init, err, detect, correct]))
msms = cPL()
print(surface25u_print2(msms, ml2))
```

We visuzlize the error syndrome before the correction.

Result

```
o   o   o
  o   o
o X o X o
  o   o
o   o   o
```

We make sure the error is gone by applying the final detection and re-running the simulation.

```
check, ml3 = surface25u_detect(data, syndrome, layer2)
cPL = to_pennylane_mcm(reduce(FTComp, [init, err, detect, correct, check]))
msms = cPL()
print(surface25u_print2(msms, ml3))
```

```
Result


o   o   o
  o   o
o   o   o
  o   o
o   o   o
```

3.4.5 High-level interface: circuit mapper

Finally, we wrap these functions into the `Surface25u` circuit mapper interface.

```
@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
  def _error_correction_cycle(self, q:Q1, layer0:int) -> FTCircuit[Q2]
  def map_op(self, op: FTOp[Q1]) -> FTCircuit[Q2]
```

4 Questions

4.1 Logical state initialization

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.

4.2 Access to mid-circuit measurements in PennyLane

Question 2: Is it possible to get a direct access to mid-circuit measurements in PennyLane during the simulation?

As of the time of writing, it appears that PennyLane cannot provide immediate measurement results prior to the conclusion of a simulation. Instead, it offers a `MeasurementResult` promise, which only supports basic operations. See Documentation link.

References

[1] Todd A. Brun. *Quantum Error Correction*. Feb. 2020. DOI: 10.1093/acrefore/9780190871994.013.35. URL: http://dx.doi.org/10.1093/acrefore/9780190871994.013.35.

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

[3] Yu Tomita and Krysta M. Svore. "Low-distance surface codes under realistic quantum noise". In: *Physical Review A* 90.6 (Dec. 2014). ISSN: 1094-1622. DOI: 10.1103/physreva.90.062320. URL: http://dx.doi.org/10.1103/PhysRevA.90.062320.