PECOS Documentation

Release 0.1.2

Ciarán Ryan-Anderson

Jan 30, 2019

Contents

PECOS, which stands for "Performance Estimator of Codes On Surfaces," is a Python package that provides a framework for studying, developing, and evaluating quantum error-correcting codes (QECCs).

PECOS is an attempt at balancing simplicity, usability, functionality, and extendibility as well as future-proofing. In the spirit of extendibility, PECOS is agnostic to quantum simulators, quantum operations, and QECCs. Of course, it is difficult to eloquently represent all QEC techniques. While agnostic to QECCs, the primary focus of PECOS has been the simulation and evaluation of lattice-surgery for topological stabilizer codes.

CHAPTER 1

History

The first incarnation of PECOS was created by Ciarán Ryan-Anderson in June 2014 to verify the lattice-surgery procedures in arXiv:1407.5103 *[LRA14]*.

CHAPTER 2

Make this Documentation

To build this documentation go to the `docs` folder and run:

```
>>> make clean
>>> make html
```

CHAPTER 3

What's Next?

To get started, check out the following:

3.1 Getting Started

3.1.1 Language Requirement

Python 3.5.2+ is need to run.

3.1.2 Package Requirements

Package requirements include:

- NumPy 1.15+
- SciPy 1.1+
- Matplotlib 2.2+
- NetworkX 2.1+

Optional packages include:

- Cython (for compiling C and C++ extensions)
- PyTest (for running tests)

3.1.3 Note on Python Distribution/Environment

PECOS was developed using the Anaconda Distribution of Python. If you decide to use this distribution you may want to create an environment so that PECOS's package requirements do not restrict you when working on other projects.

To create an environment for PECOS using Anaconda run:

```
>>> conda create -n pecos python=X numpy scipy matplotlib networkx
```

where *X* is whatever version of Python you wish to use with PECOS (e.g., `python=3.5.2`, `python=3.6`, `python=3.7`, etc.).

Alternatively, if you clone/download the package (see next section) and navigate to the root, you can create an environment by running:

```
>>> conda env create -f conda_environment.yml
```

This will create the environment `pecos` with the specific versions of Python and required packages that were used to develop PECOS. Note, you will still need to install PECOS using one of the methods described in the following sections.

To activate/use the environment in Windows run the command:

```
>>> activate pecos
```

In other operating systems you may need to run the following instead:

```
>>> source activate pecos
```

To deactivate/leave the PECOS environment run:

```
>>> deactivate
```

3.1.4 Installing and Uninstalling

PECOS has been developed to run on both Windows and Linux-based systems.

To install using pip run:

```
>>> pip install quantum-pecos
```

Alternatively, the plackage can be cloned or downloaded from GitHub:

https://github.com/PECOS-packages/PECOS

To clone PECOS using git run:

```
>>> git clone https://github.com/PECOS-packages/PECOS.git
```

Then, download/unzip or clone the version of PECOS you would like to use. Next, navigate to the root of the package (where setup.py is located) and run:

```
>>> pip install .
```

To install and continue to develop the version of PECOS located in the install folder, run:

```
>>> pip install -e .
```

To uninstall run:

```
>>> pip uninstall quantum-pecos
```

3.1.5 Development Branch

For the latest features, you may wish to clone/download the version of PECOS found in the development branch:

https://github.com/PECOS-packages/PECOS/tree/development

To clone using git run:

```
>>> git clone -b development https://github.com/PECOS-packages/PECOS.git
```

Be aware that as PECOS is in development in this branch, you may experience some bugs.

3.1.6 Tests

PECOS comes with tests to verify that the package is running as expected. These tests can be used in the development process to determine if any expected functionality has been broken.

To run tests, the package PyTest is require. Once installed, simply navigate to your PECOS installation directory and run:

```
>>> py.test
```

PyTest will automatically run all the PECOS's tests and inform you of any failures.

3.1.7 Importing

The standard method for importing PECOS is:

```
import pecos as pc
```

It will be assumed throughout the documentation that PECOS has been imported in this manner.

3.2 API Guide

Concepts in PECOS are organized around the following namespaces:

`circuits`	Circuits of different abstraction levels.
`qeccs`	Represent QEC protocols.
`error_gens`	Used to specify error models and generate errors.
`simulators`	Simulate states and operations.
`circuit_runners`	Coordinate gates of `circuits` and `error_gens` with a `simulator`.
`decoders`	Produce recovery operations given syndromes.
`tools`	Tools for studying and evaluating QEC protocols.
`misc`	A catch all namespace.

Classes and functions available in these namespaces are described in the following:

3.2.1 Quantum Circuits

Gate-based protocols, such as QEC procedures, are described in terms of quantum circuits. In PECOS the data structure used to represent quantum circuits is simply called `QuantumCircuit`. This class was designed with

similar methods as the commonly used data structures in Python such as `list`, `dict`, and `set`. This choice was made so that users accustomed to Python data structures would find `QuantumCircuit` familiar and, hopefully, easy to use.

The `QuantumCircuit` data-structure was particularly designed to efficiently represent the quantum circuits of QEC protocols. During each time step (tick), in QEC circuits many gates of just a few gate-types are applied to most of the qubits in the QECC. `QuantumCircuit` is a data structure that represents a sequence of ticks, where for each tick a collection keeps track of what few gate-types are being applied and, for each of these types, what qubits are being acted on. We will see examples of this in the following.

Note, in the following I will refer to "qudits" rather than "qubits" since a `QuantumCircuit` could represent a sequence of qudit operations.

Attributes

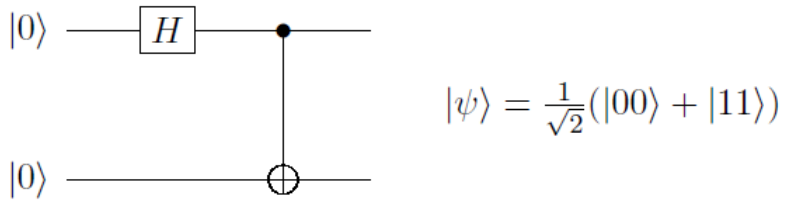
`active_qudits`	A `list` of `sets` of active qudits per tick.
`params`	A `dict` of additional information about the circuit.

Methods

`append`	Appends a collection of gates all belonging to a single tick.

An Instance

To represent a quantum circuit, such as as the preparation of the Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, seen here



we begin by creating an instance of `QuantumCircuit`:

```
>>> import pecos as pc
>>> qc = pc.circuits.QuantumCircuit()
```

A string representation of the QuantumCircuit can then be obtained:

```
>>> qc
QuantumCircuit([])
```

Here, that the object is a instance of the `QuantumCircuit` class is indicated by `QuantumCircuit()`. The brackets `[]` indicate an empty sequence.

If needed, empty ticks can be reserved when instantiating a `QuantumCircuit`:

```
>>> qc = pc.circuits.QuantumCircuit(3)
>>> qc
QuantumCircuit([{}, {}, {}])
```

Here, each tick is represented by a pair of braces `{}` and is separated by a comma. We will see later that the method `update` can used to add gates to empty ticks.

Modifying a QuantumCircuit

Next, we discuss how the methods `append`, `update`, and `discard` are used to alter a `QuantumCircuit`.

Append

We can add a tick containing some gates to the end of a `QuantumCircuit` by using the method `append`. Doing so allows us to represent the Bell-state prep circuit:

```
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append('init |0>', {0, 1})
>>> qc.append('H', {0})
>>> qc.append('CNOT', {(0,1)})
>>> qc
QuantumCircuit([{'init |0>': {0, 1}}, {'H': {0}}, {'CNOT': {(0, 1)}}])
```

Here in the final line we see a string representation of the quantum circuit in Fig 6.1. As indicated by the string, gates of the same type are grouped together. Each gatetype is indicated by a symbol (string). The standard symbols use for qubit gates in PECOS are given in appendix B. Other symbols can be used by PECOS so long as the symbols are hashable and recognized by the state-simulator used in apply the quantum circuit (see Section 6.5 for state simulators).

Paired with each gate symbol is set of gate locations, which are integers or tuples of integers. Integers are used to index qudits. Tuples are used to indicate qudits that are acted on by multi-qudit gates. The order of the qudit indices in a tuple may matter. For example, for a CNOT the first qubit is the control qubit while the second, is the target.

Listing# shows how to append a tick that consists of only one gate type. We can also append multiple gate-types per tick:

```
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append({'init |0>': {0, 1, 2, 3}})
>>> qc.append({'H': {0, 2}, 'X': {1, 3}})
>>> qc.append('CNOT', {(0,1), (2, 3)})
>>> qc
QuantumCircuit([{'init |0>': {0, 1, 2, 3}}, {'H': {0, 2}, 'X': {1, 3}}, {'CNOT': {(0,␣
↪1), (2, 3)}}])
```

Both `QuantumCircuits` and gates may have extra information that we wish to include. Such information can be added to the `QuantumCircuit` by including extra keywords as seen here:

```
>>> qc = pc.circuits.QuantumCircuit(a_var=3.0)
>>> qc.append('init |0>', {0, 1}, duration=5)
>>> qc.append({'H': {0}, 'X': {1}}, duration=1)
>>> qc
QuantumCircuit(params={'a_var': 3.0}, ticks=[{'init |0>': loc: {0, 1} - params={
↪'duration': 5}}, {'H': loc: {0} - params={'duration': 1}, 'X': loc: {1} - params={
↪'duration': 1}}])
```

As we can see in this example, extra keyword arguments are gathered into the `dicts` referred to as *params*. We will see later how the information in the params can be retrieved.

Note, the `append` method associates the extra keywords with all the gates in the tick. This limitation can be overcome by the `update`, which is discussed next.

Update

The `update` method of `QuantumCircuit` adds additional gates to a pre-existing tick. An example of using `update` is seen in the following:

```
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append({'X': {0, 1}, 'Z': {2, 3}})
>>> qc.append({'H': {0, 1}})
>>> qc.update({'CNOT': {(6, 7), (8, 9)}, 'H': {10, 11}}, tick=0)
>>> qc.update('X', {4, 5})
>>> qc
QuantumCircuit([{'X': {0, 1}, 'Z': {2, 3}, 'CNOT': {(8, 9), (6, 7)}, 'H': {10, 11}}, {
↪'H': {0, 1}, 'X': {4, 5}}])
```

By default, `update` adds gates to the current last tick of the QuantumCircuit. The `tick` keyword can be used to specify a tick. Each tick is index by an integer starting with 0.

Note, `update` will not override gate symbol-location pairs in the tick; instead, it will only add additional gate locations.

Like `append`, `update` accepts other keyword arguments and stores such information in the params `dict`:

```
>>> qc = pc.circuits.QuantumCircuit(1)
>>> qc.update('X', {0, 1}, duration=3)
>>> qc.update('H', {2, 3}, duration=2)
>>> qc
QuantumCircuit([{'X': loc: {0, 1} - params={'duration': 3}, 'H': loc: {2, 3} - params=
↪{'duration': 2}}])
```

Note, since gates in a tick should be parallel operations, if more than one gate acts on a single qudit during a gate, an `Exception` is raised.

Discard

If needed, gate locations can be removed using the method `discard`. This can be seen in the following:

```
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append('X', {0, 1, 2})
>>> qc.discard({1})
>>> qc
QuantumCircuit([{'X': {0, 2}}])
```

A `tick` keyword can be used to specify which tick the gate is discarded from. If no tick is specified, then `discard` removes gates from the last tick.

Retrieving Information

Next, how to retrieve information from a `QuantumCircuit` will be dicuss, for example, through attributes or for loops.

Number of Ticks

The number of ticks in a QuantumCircuit can be obtained using Python's code{len} function:

```
>>> qc = pc.circuits.QuantumCircuit(5)
>>> len(qc)
5
>>> qc
QuantumCircuit([{}, {}, {}, {}, {}])
```

Active Qudits

The `QuantumCircuit` data structure keeps track of which qudits have been acted on during a tick. These qudits are known as `active qudits`. The `active_qudits` attribute can be used to retrieve a list of these qudits:

```
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append({'X': {0}, 'Z': {2, 3}})
>>> qc.append({'CNOT': {(0, 2), (1, 3)}})
>>> qc.append('H', {2})
>>> qc.active_qudits
[{0, 2, 3}, {0, 1, 2, 3}, {2}]
```

This information can be useful if one wants to apply errors to inactive qudits.

For Loops

The `QuantumCircuit` class has the generator `items`, which can be used to iterate over the circuit and obtain a sequence of gate symbols, locations, and params:

```
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append({'X': {3, 5}, 'Z': {0, 1, 2}}, duration=1)
>>> qc.append({'H': {0, 1, 2, 3}})
>>> qc.append({'measure Z': {0, 3, 5}})
>>> for gate, gate_locations, params in qc.items():
...     print('%s -> %s, params: %s' % (gate, gate_locations, params))
X -> {3, 5}, params: {'duration': 1}
Z -> {0, 1, 2}, params: {'duration': 1}
H -> {0, 1, 2, 3}, params: {}
measure Z -> {0, 3, 5}, params: {}
```

One can loop over a single tick by using the keyword `tick`:

```
>>> # Following the previous example
>>> for gate, gate_locations, params in qc.items(tick=0):
...     print('%s -> %s, params: %s' % (gate, gate_locations, params))
X -> {3, 5}, params: {'duration': 1}
Z -> {0, 1, 2}, params: {'duration': 1}
```

3.2.2 QECCs

Each QECC or family of QECCs can be represented by a class. The classes available in PECOS are in the namespace `qeccs`. In this section we will discuss the methods, attributes, and structure of a `qecc`. In *Creating a QECC Class*, an example is given of how to construct a new QECC class, which can be used by PECOS.

The primary role of a `qecc` is to provide the quantum circuits of QEC protocols associated with the `qecc` such as the logical-state initialization, logical gates, and logical measurements. In the following, we will look at some examples of `qecc` classes and how they encapsulate QEC procedures.

Methods

The minimal methods expects for a `qecc`:

`gate`	Returns an instance of a requested logical gate.
`instruction`	Returns an instance of a requested logical instruction.
`plot`	Plots the physical layout of qudits.

Attributes

`name`	Name of the QECC.
`qecc_params`	Dictionary of parameters.
`distance`	Minimum number of single qudit operations that results in a logical error.
`num_logical_qudits`	Number of logical qudits.
`num_data_qudtis`	Number of data qudits.
`num_ancilla_qudits`	Number of ancillas.
`num_qudits`	Number of qudits.
`qudit_set`	Set of qudit labels used internally in the `qecc`.
`data_qudit_set`	Set of data qudit labels used internally.
`ancilla_qudit_set`	Set of ancilla qudit labels used internally.
`layout`	A `dict` of qudit label to position `tuple`.
`sides`	A `dict` describing the geometry of the `qecc`.

An Instance

Currently, the namespace `qeccs` contains classes representing the surface code on the 4.4.4.4 lattice (`Surface4444`) [Kit97b], the medial surface-code on the 4.4.4.4 lattice (`SurfaceMedial4444`) [BMD07], and the color-code on the 4.8.8 lattice (`Color488`) [BMD06].

An example instance of a `Surface4444` that represents a distance 4 surface code is given here:

```
>>> import pecos as pc
>>> surface = pc.qeccs.Surface4444(distance=3)
```

As seen in Listing 6.20, parameters are used to identify a member of the code family. For `Surface4444`, either the keyword `distance` or the keywords `height` and `width` are used to specify a member. If `distance` is used, then a representation of a square surface-code patch will be created. The `SurfaceMedial4444` class will take the same keywords as code family parameters as the `Surface4444` class. The `Color488` class only accepts `distance` as a keyword.

Logical Gate

The class `LogicalGate` represent a collection of quantum circuits that act on logical qubits. Each `LogicalGates` is identified by a symbol (string). Using this symbol, the `gate` method of a `qecc` can be used to obtain an instance of a corresponding `LogicalGate` instance:

```
>>> surface = pc.qeccs.Surface4444(distance=3)
>>> identity = surface.gate('I')
```

In the above code, the symbol `'I'` is used to retrieve a logical gate corresponding to identity (syndrome extraction).

Keyword arguments may be used to modify `LogicalGates`:

```
>>> surface = pc.qeccs.Surface4444(distance=3)
>>> # Get an identity gate with only one round of syndrome extraction.
>>> identity = surface.gate('I', num_syn_extract=1)
```

Here the keyword argument `num_syn_extract` is used to explicitly request an identity with only one round of syndrome extraction. Typically, the number of rounds of syndrome extraction for an identity gate is equal to the QECC's distance.

The main use for `LogicalGate` instances is as logical operations in the logical analogs of quantum circuits, which are described in *Logical Circuits*

Logical Instruction

A `LogicalGate` is composed of a sequence of `LogicalInstructions`. A `LogicalInstruction` represents a collection of quantum circuits. Often these collections are repeated or used in multiple `LogicalGates`. An example of a `LogicalInstruction` is one round of error correction.

Like `LogicalGates`, `LogicalInstuctions` are represented by symbols (strings). The `instr_symbols` attribute of a `LogicalGate` can be use to retrieve a list of symbols corresponding to the `LogicalInstuctions` that form the `LogicalGate`:

```
>>> surface = pc.qeccs.Surface4444(distance=3)
>>> identity = surface.gate('I')
>>> identity.instr_symbols
['instr_syn_extract', 'instr_syn_extract', 'instr_syn_extract']
>>> # Request an identity with a single round of syndrome extraction.
>>> identity = surface.gate('I', num_syn_extract=1)
>>> identity.instr_symbols
['instr_syn_extract']
```

In the following, we see how to retrieve an instance of the `'instr_syn_extract'` instruction and then see what `QuantumCircuit` it represents:

```
>>> surface = pc.qeccs.SurfaceMedial4444(distance=3)
>>> # Get the LogicalInstruction instance representing the syndrome-extraction␣
↪instruction.
>>> instr = surface.instruction('instr_syn_extract')
>>> instr.circuit
QuantumCircuit([{'init |0>': {0, 16, 4, 5, 6, 10, 11, 12}}, {'H': {0, 16, 11, 5}}, {
↪'CNOT': {(15, 12), (11, 14), (8, 6), (5, 7), (13, 10), (0, 2)}}, {'CNOT': {(9, 12),␣
↪(2, 6), (7, 10), (11, 15), (0, 3), (5, 8)}}, {'CNOT': {(7, 4), (16, 13), (14, 10),␣
↪(11, 8), (5, 1), (9, 6)}}, {'CNOT': {(3, 6), (16, 14), (11, 9), (5, 2), (8, 10), (1,
↪ 4)}}, {'H': {0, 16, 11, 5}}, {'measure Z': {0, 16, 4, 5, 6, 10, 11, 12}}])
```

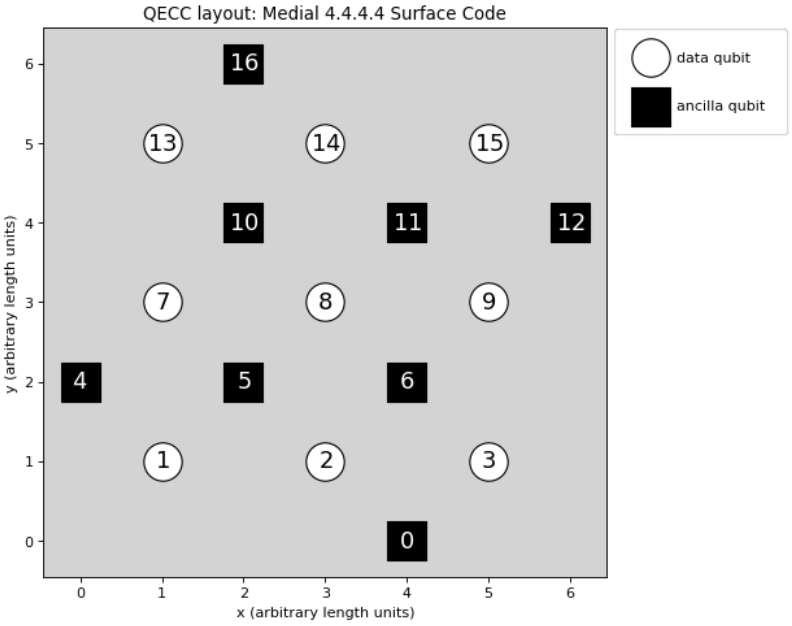
Plotting

Both `qeccs` and `LogicalInstuctions` have a method called `plot` that will generate a plot that represents the object. These plots can be useful in understanding the structure of a QECC and its logical instructions.

The following is an example of using the `plot` method for a `qecc`:

```
>>> surface = pc.qeccs.SurfaceMedial4444(distance=3)
>>> surface.plot()
```

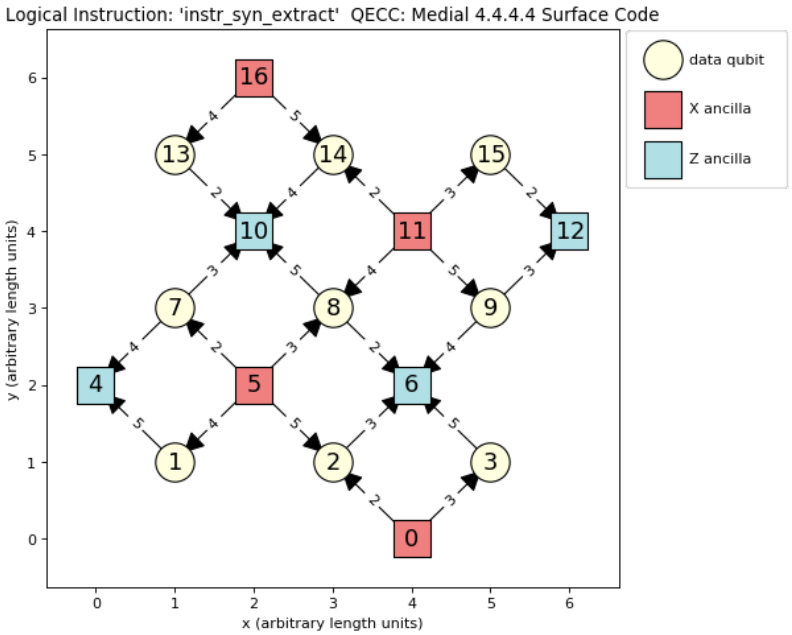
This results in the plot seen in:



The plot of `LogicalInstructions` often indicates the sequence of gate operations. An example of a plot of the syndrome extraction instruction of `surface` be obtained by the following lines:

```
>>> surface = pc.qeccs.SurfaceMedial4444(distance=3)
>>> syn_extract = surface.instruction('instr_syn_extract')
>>> syn_extract.plot()
```

The resulting plot is seen in:



The previous figure depicts the syndrome-extraction logical-instruction of a distance-three, medial surface-code. Red squares represent the ancillas involved in X checks, the blue squares represent the ancillas involved in Z checks, and the cream circles represent the data qubits. The numbers inside the squares and circles are labels of the qubits as used in `QuantumCircuits`. The edges indicate the CNOTs used in the checks. The thicker end of each edge denotes

the qubit that is the target of the CNOT. The numbers labeling the edges are ticks when the corresponding CNOT is applied.

3.2.3 Logical Circuits

The class `LogicalCircuit`, which is found in the `circuits` namespace, is a logical analog of the class `QuantumCircuit`. The `LogicalCircuit` class has the same methods and attributes as `QuantumCircuit`; however, there are a few changes in the behavior of some of the methods. As the two classes are very similar, I will give a few examples of using the `LogicalCircuit` class to illustrate their differences.

An instance of a `LogicalCircuit` can be created using the following lines:

```
>>> import pecos as pc
>>> logic = pc.circuits.LogicalCircuit()
```

Instead of gate symbols, the `append` method of the `LogicalCircuit` class accepts `LogicalGates` directly. Also, if a `LogicalCircuit` contains a single `qecc` then a gate location is not needed:

```
>>> surface = pc.qeccs.Surface4444(distance=3)
>>> logic = pc.circuits.LogicalCircuit()
>>> logic.append(surface.gate('ideal init |0>'))
>>> logic.append(surface.gate('I'))
```

3.2.4 Simulators

Quantum states and their dynamics are simulated by classes belonging to the namespace `simulators`. PECOS contains a stabilizer simulator called `StabSim`.

Expected Methods

The set of gates allowed by a simulator may differ (the standard set for PECOS is given in *Standard Gates*; however, each simulator is expected to have a set of standard methods. I will describe them in this section.

When initializing a simulator, the first argument is expected to be the number of qudits to be simulated. This reserves the size of the quantum registry:

```
>>> from pecos.simulators import StabSim
>>> state = StabSim(4)
```

Note, for all simulators, the initial state of each qudit is the state $|0\rangle$.

The only other method expected is the `run_gate` method. This method can be used to apply gates to a `simulator` instance by using the `run_gate` method:

```
>>> # Continuing from the previous Listing.
>>> state.run_gate('X', {0, 1})
{}
```

Here the first argument is a gate symbol that is recognized by the `simulator` and the second argument is a set of gate locations. Other keywords and arguments may be supplied if it is allowed by the `simulator`. Such arguments could be used to change the behavior of the gate. For example, arguments could be used to define gate rotation-angles.

If measurements are made then a dictionary indicating the measurement results is returned by `run_gate`:

```
>>> # Continuing from the previous Listing.
>>> state.run_gate('measure Z', {0, 1, 3})
{0: 1, 1: 1}
```

Here we see that the keys of the results dictionary are the qudit locations of the measurements, and the values are the corresponding measurement results except that zero results are not returned.

Classes in the `circuit_runners` namespace combine `QuantumCircuits` and simulators to apply gates to simulated quantum states. For a discussion about these classes see *Circuit Runners*.

StabSim

Methods that specific to `StabSim` will now be described.

The `print_stabs` method prints a stabilizer table corresponding to the state currently store in the simulator:

```
>>> state = StabSim(3)
>>> state.run_gate('CNOT', {(0, 1)})
{}
>>> state.run_gate('X', {0})
{}
>>> state.print_stabs()
 -ZII
 -ZZI
  IIZ
------------------------------
  XXI
  IXI
  IIX
([' -ZII', ' -ZZI', '  IIZ'], ['  XXI', '  IXI', '  IIX'])
```

Here in the print output stabilizer generators are indicated by the strings above the dashed lines, while destabilizer generators are indicated by the strings below.

The `logical_sign` method can be used to determine the sign of stabilizer generators. As the stabilizer simulators represent stabilizer states, logical basis-states are stabilized by logical operators. Therefore, this method is useful in Monte Carlo simulations to determine if logical errors have flipped the sign of logical operators.

An example of using the `logical_sign` method is seen in the following:

```
>>> # Continuing with the following example:
>>> from pecos.circuits import QuantumCircuit
>>> stab = QuantumCircuit([{'Z': {0, 1}}])
>>> state.logical_sign(stab)
1
>>> stab = QuantumCircuit([{'Z': {2}}])
>>> state.logical_sign(stab)
0
```

A 1 is returned if the phase of the stabilizer is -1, and a 0 is returned if the phase is $+1$. If the stabilizer supplied to `logical_sign` is not a stabilizer of the state, then an exception will be raised.

Extensions

Todo: Discuss the simulator extensions when finished...

3.2.5 Standard Gates

While some simulators may allow access to other gate sets, the standard gates recognized by PECOS are:

Initializations

State initializations in Pauli bases:

`'init |+>'`	(Re)initiate the state $\|+\rangle$
`'init |->'`	(Re)initiate the state $\|-\rangle$
`'init |+i>'`	(Re)initiate the state $\|+i\rangle$
`'init |-i>'`	(Re)initiate the state $\|-i\rangle$
`'init |0>'`	(Re)initiate the state $\|0\rangle$
`'init |1>'`	(Re)initiate the state $\|1\rangle$

Unitaries

Pauli operations:

`'I'`	$X \rightarrow X, Z \rightarrow Z$
`'X'`	$X \rightarrow X, Z \rightarrow -Z$
`'Y'`	$X \rightarrow -X, Z \rightarrow -Z$
`'Z'`	$X \rightarrow -X, Z \rightarrow Z$

Square-root of Pauli operations:

`'Q'`	$X \rightarrow X, Z \rightarrow -Y$
`'R'`	$X \rightarrow -Z, Z \rightarrow X$
`'S'`	$X \rightarrow Y, Z \rightarrow Z$
`'Qd'`	$X \rightarrow X, Z \rightarrow Y$
`'Rd'`	$X \rightarrow Z, Z \rightarrow -X$
`'Sd'`	$X \rightarrow -Y, Z \rightarrow Z$

Hamadard-like:

`'H'}, ``'H+z+x'}, or ``'H1'`	Hadamard: $X \leftrightarrow Z$
`'H-z-x'` or `'H2'`	$X \leftrightarrow -Z$
`'H+y-z'` or `'H3'`	$X \rightarrow Y, Z \rightarrow -Z$
`'H-y-z'` or `'H4'`	$X \rightarrow -Y, Z \rightarrow -Z$
`'H-x+y'` or `'H5'`	$X \rightarrow -X, \rightarrow Y$
`'H-x-y'` or `'H6'`	$X \rightarrow -X, Z \rightarrow -Y$

Rotations about the face of an octahedron:

'F1'	$X \to Y \to Z \to X$
'F2'	$X \to -Z, Z \to Y$
'F3'	$X \to Y, Z \to -X$
'F4'	$X \to Z, Z \to -Y$
'F1d'	$X \to Z \to Y \to X$
'F2d'	$X \to -Y, Z \to -X$
'F3d'	$X \to -Z, Z \to -Y$
'F4d'	$X \to -Y, Z \to X$

Two-qubit gates:

'CNOT'	The controlled-X gate
'CZ'	The controlled-Z gate
'SWAP'	Swap two qubits
'G'	Equivalent to: $CZ_{1,2}\ H_1 \otimes H_2\ CZ_{1,2}$

Measurements

Measurements in Pauli bases:

'measure X'	Measure in the X-basis
'measure Y'	Measure in the Y-basis
'measure Z'	Measure in the Z-basis

3.2.6 Circuit Runners

Classes belonging to the `circuit_runners` namespace apply the gates of `LogicalCircuits` and `QuantumCircuits` to states represented by simulators. `circuit_runners` are also responsible for applying error models to quantum circuit; however, we will discus this in *Error Generators*.

The main `circuit_runner` is simply called `Standard`. There is another call `TimingRunner`, which is essentially the same as `Standard` except that it is used to time how long it takes simulators to apply gates and can be used to compare the runtime of simulators. I will now discuss these two `circuit_runners`.

Standard

For convenience, the following tabels list the attributes and methods of `Standard`:

Methods

`init`	Adds a collection of gates to the end of `ticks`.
`run_circuit`	Applies gates from a `QuantumCircuit`.
`run_logic`	Applies gates from a `LogicalCircuit`.

Attributes

seed	The integer used as a seed for random number generators

Instance

To create an instance of `Standard` one can simply write:

```
>>> import pecos as pc
>>> circ_runner = pc.circuit_runners.Standard()
```

By default, a `Standard` uses the `StabSim` as a simulator. This can be changed as follows:

```
>>> from somepackage import MyCustomSim
>>> circ_runner = pc.circuit_runners.Standard(simulator=MyCustomSim)
```

The `init` method is used to (re)initialize a `simulator` instance. An example of using this method to create a four-qubit registry is seen here:

```
>>> # Following from the previous code block.
>>> circ_runner = pc.circuit_runners.Standard()
>>> state = circ_runner.init(4)
```

The `run_circuit` method is used to apply a `QuantumCircuit` to a state in the following:

```
>>> # Continuing with the previous code block.
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append('X', {0, 1})
>>> qc.append('measure Z', {0, 1, 3})
>>> circ_runner.run_circuit(state, qc)
{1: {0: 1, 1: 1}}
```

In the last line of this code block, we see the measurement record produced by the `circuit_runner`. The keys of the outer dictionary are tick indices, while for the inner dictionary the keys are the indices of qubits with non-zero measurements and the values are the measurement results.

The `run_logic` method is used to apply `LogicalCircuits`:

```
>>> surface = pc.qeccs.Surface4444(distance=3)
>>> logic = pc.circuits.LogicalCircuit()
>>> logic.append(surface.gate('ideal init |0>'))
>>> logic.append(surface.gate('I'))
>>> state = circ_runner.init(surface.num_qudits)
>>> circ_runner.run_logic(state, logic)
({}, {})
```

The final line is the output of `run_logic`. The first dictionary is a record measurement and the second is a record of the errors generated. In this example, all the measurement results are zero and we have not applied any error models. In *Error Generators*, there are examples of where this is not the case; therefore, refer to that section if you are curious about the output of `run_logic`.

TimingRunner

As mention, `TimingRunner` is essentially the same as `Standard` except the runtime for applying gates is recorded. The attribute `total_time` stores this value and is used in the following:

```
>>> circ_runner = pc.circuit_runners.TimingRunner()
>>> state = circ_runner.init(4)
>>> qc = pc.circuits.QuantumCircuit()
>>> qc.append('X', {0, 1, 2, 3})
>>> circ_runner.run_circuit(state, qc)
{}
>>> circ_runner.total_time
7.22257152574457e-06
```

`TimingRunner` times the execution of gates by using Python's `perf_counter` method. The time recorded by `total_time` continues to accumulate until it is reset by the `reset_time` method:

```
>>> # Continuing from previous Listing.
>>> circ_runner.reset_time()
>>> circ_runner.total_time
0.0
```

3.2.7 Error Generators

Error models are represented by classes called "error generators" that are in the `error_gens` namespace. They are called upon by `circuit_runners` to apply noise to ideal quantum circuits.

In this section I will discuss `GatewiseGen` and `DepolarGen` classes. Both represent `stochastic error models`. That is, error models that apply gates as noise according to classical probability distributions.

GatewiseGen

The `GatewiseGen` class allow one to define custom stochastic error-models where for each ideal gate-type the errors applied to the ideal gate and the classical probability distribution for applying errors can be specified. Since many examples of using the class are given, I have moved the discussion of the `GatewiseGen` class to *Gate-wise Error Models*.

The follow section provides examples of how `error_gens` are used in practice

DepolarGen

The `DepolarGen` class is used to represent the symmetric depolarizing channel, which is commonly studied in QEC. For single-qubit gates, this class is used to apply errors at probability p from set $\{X, Y, Z\}$. For two-qubit gates, errors also occur with probability p but errors are chosen uniformally from the set $\{I, X, Y, Z\}^{\otimes 2} \setminus I \otimes I$. Errors are always applied after ideal gates except for measurements. In which case, the errors are applied before.

An example of creating an instance of `DepolarGen` is seen here:

```
>>> import pecos as pc
>>> depolar = pc.error_gens.DepolarGen(model_level='code_capacity', has_idle_
↪errors=False, perp_errors=True)
```

The `model_level` keyword is used to specify to what set of gates the `DepolarGen` is applied to. If `model_level` is set to the value of `'code\_capacity'`, then the error model is applied before each

`LogicalInstruction` to each data qubits as if these qubits are acted on by `'I'`. The error model is not applied to any other circuit element. If `model_level` is set to the value `'phenomenological'`, then the error model applied to data qubits before each `LogicalInstruction` as well as to any measurement. If `model_level` is set to the value `'circuit'`, then the error model is applied to all the gates in the `QuantumCircuit`. The default value of `model_level` is `'circuit'`.

The `has_idle_errors` is a keyword that is only relevant when `model_level == 'circuit'`. If `has_idle_errors` is set to `True`, then the error model is applied to inactive qubits as if the qubit is acted on by `'I'`. If `has_idle_errors` is set to `False`, then this does not occur. The default value of `has_idle_errors` is `True`.

If the `perp_errors` keyword is set to `True`, then errors that are applied to Pauli-basis initializations and measurements are errors that do not include the Pauli-basis of the initializations or measurements. So, for example, Z is not applied as an error to the `'init |0>'` operation. If the `perp_errors` keyword is set to `False`, then there is no restriction to the errors. The default value of `perp_errors` is `True`.

An example of applying an error model using `DepolarGen` to a `LogicalCircuit` is seen in the following:

```
>>> depolar = pc.error_gens.DepolarGen(model_level='code_capacity')
>>> surface = pc.qeccs.Surface4444(distance=3)
>>> logic = pc.circuits.LogicalCircuit()
>>> logic.append(surface.gate('ideal init |0>'))
>>> logic.append(surface.gate('I'))
>>> circ_runner = pc.circuit_runners.Standard(seed=1)
>>> state = circ_runner.init(surface.num_qudits)
>>> meas, err = circ_runner.run_logic(state, logic, error_gen=depolar, error_params={
↪'p': 0.1})
```

Note that the keyword argument `error_params` is used to pass a dictionary that indicates the probability p of the depolarizing error model.

The values returned by the `run_logic` method is recorded in the variables `meas` and `err`. These variables are dictionaries that record the measurement output and applied errors.

An example of measurement outcomes is given here:

```
>>> # Following the previous example.
>>> meas
{(1, 0): {7: {9: 1, 11: 1}}}
```

Here, in the last line, we see the measurement outcome. The key of the outer dictionary is a tuple where the first element is the tick index of the `LogicalGate` and the second element is an index corresponding to a `LogicalInstance`. That is, the tuple records at what point in the `LogicalCircuit` was the measurement made. The value of the outer dictionary is just the measurement-outcome dictionary of a `QuantumCircuit`.

We can see the errors that were generated by the `DepolarGen` in these lines:

```
>>> # Following the previous example.
>>> err
{(1, 0): {0: {'after': QuantumCircuit([{'X': {4}, 'Z': {10}}])}}}
```

In the above Listing, we see a dictionary that stores what errors were applied to the `LogicalCircuit`. The key of the outer dictionary, once again, is a tuple indicating the tick of a `LogicalGate` and the index of a `LogicalInstance`. The key of the next inner dictionary is `QuantumCircuit` tick when the error occurred. The key `'after'` of the next inner dictionary indicates that the errors are applied after ideal gates. The key `'before'` is used when indicating that errors are applied before gates. The values of both the `'after'` and `'before'` keys are `QuantumCircuits`. These circuits are the errors that are applied.

The data structure used to describe the errors that are applied to a `LogicalCircuit` can be directly supplied to

a `run_logic` method of a `circuit_runner`. Doing so will cause the `run_logic` method to apply the given error to a `LogicalCircuit`. This can be seen in the following:

```
>>> # Continuing the previous examples.
>>> logic2 = pc.circuits.LogicalCircuit()
>>> logic2.append(surface.gate('ideal init |+>'))
>>> logic2.append(surface.gate('I'))
>>> state2 = circ_runner.init(surface.num_qudits)
>>> meas2, err2 = circ_runner.run_logic(state2, logic2, error_circuits=err)
```

One use for this is to apply the same error to a different logical basis-state. Doing so allows one to determine if a logical error occurs for the logical operations that stabilizer the basis state.

Note that the `circuit_runners` currently only apply errors to `LogicalCircuits` and not to `QuantumCircuits`.

Todo: Discuss the leakage error model when it is verified...

3.2.8 Decoders

A decoder in PECOS is simply a function or other callable that takes the measurement outcomes from error extractions (syndromes) as input and returns a `QuantumCircuit`, which is used as a recovery operation to mitigate errors. Decoder classes and functions are in the `decoders` namespace.

The `MWPM2D` class is an available decoder class, which I will discuss next.

MWPM2D

One of the standard decoders used for surface codes is the minimum-weight-perfect-matching (MWPM) decoder *[Den+02]*. The `MWPM2D` class implements the 2D version of this decoder for `Surface4444` and `SurfaceMedial4444`, that is, it decodes syndromes for a single round of error extraction:

```
>>> import pecos as pc
>>> depolar = pc.error_gens.DepolarGen(model_level='code_capacity')
>>> surface = pc.qeccs.Surface4444(distance=3)
>>> logic = pc.circuits.LogicalCircuit()
>>> logic.append(surface.gate('ideal init |0>'))
>>> logic.append(surface.gate('I', num_syn_extract=1))
>>> circ_runner = pc.circuit_runners.Standard(seed=1)
>>> state = circ_runner.init(surface.num_qudits)
>>> decode = pc.decoders.MWPM2D(surface).decode
>>> meas, err = circ_runner.run_logic(state, logic, error_gen=depolar, error_params={
↪'p': 0.1})
>>> meas
{(1, 0): {7: {3: 1, 5: 1, 9: 1, 15: 1}}}
>>> err
{(1, 0): {0: {'after': QuantumCircuit([{'Y': {4}, 'X': {10}}])}}}
>>> decode(meas)
QuantumCircuit([{'X': {10}, 'Y': {4}}])
```

3.2.9 Tools

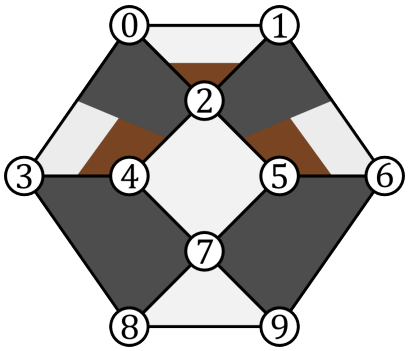
Todo: Write about the available tools.

3.3 Examples

The following are a collection of examples:

3.3.1 Verifying a Stabilizer Code

In this example we will see how `VerifyStabilizers` can be used to develop a simple, distance-three code. We begin by considering the generators in:



In this figure we see an example stabilizer code. Data qubits are indicated as white circles and are labelled by numbers. Polygons represent checks. The coloring of the polygons indicate how the associated check acts on qubits it touches. When the check is white on a qubit, the check acts as Z. When black, X. When brown, Y.

We now use `VerifyStabilizers` to represent the checks given in the previous figure:

```
>>> import pecos as pc
>>> qecc = pc.tools.VerifyStabilizers()
>>> qecc.check('X', (3, 4, 7, 8))
>>> qecc.check('X', (5, 6, 7, 9))
>>> qecc.check('Z', (2, 4, 5, 7))
>>> qecc.check('Z', (7, 8, 9))
>>> qecc.check(('Z', 'Z', 'Y'), (0, 1, 2))
>>> qecc.check(('X', 'X', 'Z', 'Y'), (0, 2, 3, 4))
>>> qecc.check(('X', 'X', 'Z', 'Y'), (1, 2, 6, 5))
```

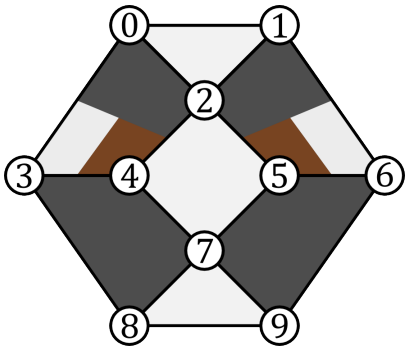
Here we see that the `check` method can be used to specify a generator. If the first argument is a string, then this indicates the Pauli-type of the check. The second argument then indicates which qubits the check acts on. If the first argument is a tuple, then the tuple is a sequence of strings which indicate how the generator acts on the corresponding qubits indicated in the tuple of the second argument.

Once one has finished specifying the generators of the code, the `compile` method should be used:

```
>>> # Continuing the last listing.
>>> qecc.compile()
Check:
check(('Z', 'Z', 'Y'), (0, 1, 2))
anticommutes with:
check(Z, (2, 4, 5, 7))
```

Once `compile` is called, `VerifyStabilizers` checks to see if all the generators anticommute. If any do `VerifyStabilizers` prints out a message indicating which checks anticommute.

Finding that our original stabilizer code design does have anticommuting generators, we can modify the QECC slightly to address the issue as seen in:



Re-specifying the generators according to the figure above, we run `compile` again and see that we have solved the commutation problem:

```
>>> qecc = pc.tools.VerifyStabilizers()
>>> qecc.check('X', (3, 4, 7, 8))
>>> qecc.check('X', (5, 6, 7, 9))
>>> qecc.check('Z', (2, 4, 5, 7))
>>> qecc.check('Z', (7, 8, 9))
>>> qecc.check(('Z', 'Z', 'Z'), (0, 1, 2))
>>> qecc.check(('X', 'X', 'Z', 'Y'), (0, 2, 3, 4))
>>> qecc.check(('X', 'X', 'Z', 'Y'), (1, 2, 6, 5))
>>> qecc.compile()
```

It is possible that we have specified a set of generators with redundant elements. That is, some of the generators can be written as products of the others. To check if this is the case, we use the method `generators`:

```
>>> # Following the last listing.
>>> qecc.generators()
Number of data qubits: 10
Number of checks: 7
Number of logical qubits: 3

Stabilizer generators:
XIXZYIIIII
IXXIIYZIII
IIZIZZIZII
ZZZIIIIIII
IIIIIIIZZZ
IIIXXIIXXI
IIIIIXXXIX

Destabilizer generators:
ZIIIIIIIII
IZIIIIIIII
ZIIIXIIIII
ZIXIXIIIII
ZIIIXIIXII
IIIIIIIIZI
IIIIIIIIIZ
```

(continues on next page)

(continued from previous page)

```
Logical operators:
. Logical Z #1:
  IIIZIIIIZI
. Logical X #1:
  ZIIXIIIIII
. Logical Z #2:
  IZIIIZIIIZ
. Logical X #2:
  ZZIIXXIIII
. Logical Z #3:
  IIIIIIZIIZ
. Logical X #3:
  IZIIIIXIII
```

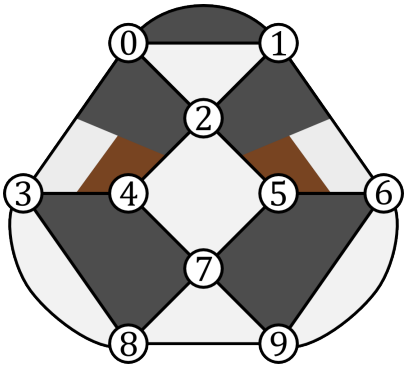
If we had redundant generators then `generators` would alert us. Luckily, we do not and `generators` has printed out some useful information including number of logical qubits, destabilizers, and a possible set of logical operators.

We can then use the `distance` method to determine the distance of the code. Note, to find the distance of a code, this method will try all combinations of possible Pauli errors. It starts with the smallest weight and evaluating larger and larger weights until a logical error is detected. Since this is a combinatorial search, the algorithm is not efficient and the runtime quickly grows with the size of the code. In practice, for smaller code of less than 20 or so qubits, the runtime is manageable.

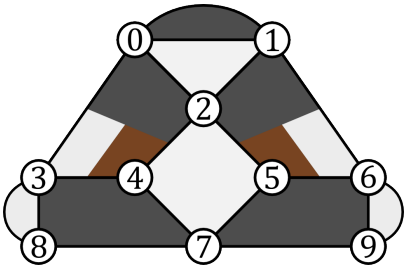
We now run the `distance` method:

```
>>> # Following the last listing.
qecc.distance()
----
Checking errors of length 1...
Checking errors of length 2...
Logical error found: Xs - {0, 1} Zs - set()
This is a [[10, 3, 2]] code.
```

The last line of the code block indicates what type of QECC we have. The notation $[[n, k, d]]$ indicates that the code encodes k qubits into n physical qubits and has a distance of d. Since the number of errors a QECC can correct is $t = \lfloor (d-1)/2 \rfloor$ and the distance of our code is two, this means our QECC can only detect but not correct errors. Because the `distance` method indicates the smallest logical error it found, we can use this information to mitigate the error by either introducing another check to detect the error or by including the logical error as a check. We do the later. Doing this, we find that we have not increased the distance of the code. If we repeat the process two more times we will end up with a code that has no logical qubits and, therefore, encodes a stabilizer state:



We seemly failed to create a higher distance code; however, we can persevere by removing a higher-weight stabilizer generator. If we remove the check that acts like Pauli Z on qubits 7, 8, and 9, we will get the stabilizer code in:



Evaluating the distance of this new version of the code:

```
>>> qecc = pc.tools.VerifyStabilizers()
>>> qecc.check('Z', (2, 4, 5, 7))
>>> # qecc.check('Z', (7, 8, 9))
>>> qecc.check('X', (3, 4, 7, 8))
>>> qecc.check('X', (5, 6, 7, 9))
>>> qecc.check(('X', 'X', 'Z', 'Y'), (0, 2, 3, 4))
>>> qecc.check(('X', 'X', 'Z', 'Y'), (1, 2, 6, 5))
>>> qecc.check('Z', (0, 1, 2))
>>> qecc.check('X', (0, 1))
>>> qecc.check('Z', (3, 8))
>>> qecc.check('Z', (6, 9))
>>> qecc.compile()
>>> qecc.generators()
Number of data qubits: 10
Number of checks: 9
Number of logical qubits: 1
----

Stabilizer generators:
  XIXZYIIIII
  IXXIIYZIII
  IIZIZZIZII
  IIIZIIIIZI
  XXIIIIIIII
  ZZZIIIIIII
  IIIIIIZIIZ
  IIIXXIIXXI
  IIIIIXXXIX

Destabilizer generators:
  IIIIZIIIZI
  ZZIIZIIIZI
  ZZIIZXIIZI
  IIIXZIIIZI
  ZIIIZIIIZI
  ZZXIZXIIZI
  ZZIIZIXIZI
  IIIIIIIIZI
  IIIIIIIIIZ

Logical operators:
. Logical Z #1:
  IIIIIIIZZZ
. Logical X #1:
  ZZIIZXIXZI
>>> qecc.distance()
```

(continues on next page)

(continued from previous page)

```
----
Checking errors of length 1...
Checking errors of length 2...
Checking errors of length 3...
Logical error found: Xs - {0, 2, 7} Zs - set()
This is a [[10, 1, 3]] code.
```

Thus, we have developed a simple distance three code.

3.3.2 Creating a QECC Class

To facilitate the evaluation of QECC protocols not included in PECOS, this appendix shows how to represent a QECC with a Python so can be used with PECOS. In particular, we look at representing the repetition code.

To begin, we create an empty Python file called `zrepetition.py` and import some useful classes:

```
"""
A representation of the Z-check repetition code.
"""
from pecos.circuits import QuantumCircuit
from pecos.qeccs import QECC, LogicalGate, LogicalInstruction
```

Subclasses of `QECC`, `LogicalGate`, and `LogicalInstruction` inherit numerous methods and attributes that simplify the creation of new `qeccs`. If some of the inherited methods and attributes are not appropriate for a QECC, one can typically override them.

The QECC class

We now create a class `ZReptition` to represent our `qecc`:

```
class ZRepetition(QECC):
    def __init__(self, **qecc_params):
        # Pass qecc_params to the parent class:
        super().__init__(**qecc_params)
        # Set variables that describe the QECC:
        self._set_qecc_description()
        # Create a lattice for placing qubits:
        self.layout = self._generate_layout()
        # Identify the sides of the QECC:
        self._determine_sides()
        # Identify symbols with gate/instruction classes:
        self._set_symbols()
```

Here, the `dict` called `qecc_params` will be used to specify parameters that identify a member of the QECC's family. We will discuss later the method calls see in the `__init__` method.

Next, we write the `_set_qecc_description`, which sets class attributes that describe the QECC:

```
def _set_qecc_description(self):
    self.name = 'Z Repetition Code'
    # Size of the repetition code:
    self.length = self.qecc_params['length']
    self.distance = 1
```

(continues on next page)

(continued from previous page)

```
    self.num_data_qudits = self.length
    self.num_logical_qudits = 1
    self.num_ancillas = self.num_data_qudits - 1
```

The `name` attribute identifies the code. The `length` attribute we will use to define how long the QECC is. We use `distance` to determine the size of the QECC. We will be describing a repetition that only has Z checks; therefore, the code will not detect any Z errors. For this reason, the distance is one no matter the length of the QECC. `num_data_qudits` is the number of data qubits. The attribute `num_logical_qudits` is the number of logical qubits we will encode with this QECC. The total number of ancillas used in all the `qecc`'s procedures is equal to `num_ancillas`. The total number of qubits is equal to the `num_qudits` attribute. This attribute is determined by the parent class `QECC`.

Next, we construct `_set_symbols`, which contain dictionaries that associate symbols to `LogicalInstructions` and `LogicalGates`. We will describe these classes later.

```
def _set_symbols(self):
    # instruction symbol => instr. class:
    self.sym2instruction_class = {
        'instr_syn_extract': InstrSynExtraction,
        'instr_init_zero': InstrInitZero, }
    # gate symbol => gate class:
    self.sym2gate_class = {
        'I': GateIdentity,
        'init |0>': GateInitZero, }
```

Now we write the method `_generate_layout`, which generates the physical layout of qubits. As we will see later, a physical layout is useful for defining the quantum circuits of the QECC protocol.

```
def _generate_layout(self):
    self.lattice_width = self.num_qudits
    data_ids = self._data_id_iter()
    ancilla_ids = self._ancilla_id_iter()
    y = 1
    for x in range(self.lattice_width):
        if x%2 == 0: # Even (ancilla qubit)
            self._add_node(x, y, data_ids)
        else: # Odd (data qubit)
            self._add_node(x, y, ancilla_ids)
    # `add_nodes` updates an attribute called `layout.`
    return self.layout
```

Finally for the `qecc`, we will add the method `_determine_sides` to create a dictionary that defines the physical boundary of the QECC. This information can be used by decoders to understand the geometry of the code.

```
def _determine_sides(self):
    self.sides = {
        'length': set(self.data_qudit_set)
        }
```

Logical Instruction Classes

Now that we have created a class to represent the QECC, we will now create classes to represent logical instructions. First create an logical instruction class, called `InstrSynExtraction`, that represents one round of syndrome extraction. Similar to the `ZRepitition` class, we will subclass our class off of the `LogicalInstruction`, which is provided by PECOS. After we do this, we will write an initialization method that receives as arguments the

qecc instance the instruction belongs to, the associated symbol, and a dictionary of logical gate parameters called gate_params. This dictionary will come from the LogicalGate that contains the LogicalInstruction and may alter the LogicalGate and the QuantumCircuit contained in the LogicalInstruction.

```
class InstrSynExtraction(LogicalInstruction):
    def __init__(self, qecc, symbol, **gate_params):
        super().__init__(qecc, symbol, **gate_params)

        # The following are convienent for plotting:
        self.ancilla_x_check = set()
        self.ancilla_z_check = qecc.ancilla_qudit_set
        self._create_checks()
        self.set_logical_ops()
        self._compile_circuit(self.abstract_circuit) # Call at end
```

We now include the _create_checks method, which we will use to define the checks of the QECC:

```
   def _create_checks(self):
       self.abstract_circuit = QuantumCircuit(**self.gate_params)
       for qid in self.qecc.ancilla_qudit_set:
           x, y = qecc.layout[qid]

           # Get the data qubits to each side.
           d1 = qecc.position2qudit[(x-1, y)]
           d2 = qecc.position2qudit[(x+1, y)]
           self.abstract_circuit.append('Z check', {qid, d1, d2}, datas=[d1, d2],␣
↪ancillas=[qid])

Here we use the physical layout of the QECC to construct checks. A ``QuantumCircuit``␣
↪called ``abstract_circuit`` is used to register each :math:`Z`-type check, the␣
↪qubits it acts on, and whether the qubits are used as data or ancilla qubits. Note,␣
↪check circuits such as the ones seen in Fig~\ref{fig:surf-checks} are used to␣
↪implement the checks. The order of the data qubits in the ``datas`` keyword␣
↪indicates the order which the data qubits are acted on by the check circuits. The␣
↪checks registered by ``abstract_circuit`` are later compiled into quantum circuits.
```

Now we will write the method set_logical_ops, which define the logical operators of the QECCs.

```
  def set_logical_ops(self):
      data_qubits = set(self.qecc.data_qudit_set)
      logical_ops = [
          {'X': QuantumCircuit([{'X': {0}}]),
           'Z': QuantumCircuit([{'Z': data_qubits}])}
           ]
      self.initial_logical_ops = logical_ops
      self.final_logical_ops = logical_ops

      # The final logical sign and stabilizer
      self.logical_stabilizers = None
      self.logical_signs = None

Here, the variables ``initial_logical_ops`` and ``final_logical_ops`` that represent␣
↪the initial and final logical operators, respectively, are set. Each of these␣
↪variables are a list where each element represents a collection of logical␣
↪operators of an encoded qudit. In particular, each element is a dictionary where␣
↪the keys are symbols identified with the logical operator and the values are␣
↪``QuantumCircuits`` representing the unitaries of logical operators.
```

If a logical operator encodes a stabilizer state then logical_stabilizers is a list of the strings represent-

ing the logical operators that stabilizer the state. If the logical operator does not specifically encode a stabilizer state, then `logical_stabilizers` is set to `None`. The variable `logical_signs` is a list of signs the corresponding logical operators in `logical_stabilizers`. If the phase of the operators is $+1$, then the element of `logical_signs` is 0. If the phase of the operators is $-$, then the element of `logical_signs` is 1. If `logical_stabilizers` is `None`, then `logical_signs` is `None`.

We now define the initialization of the logical zero-stat:

```
  class InstrInitZero(LogicalInstruction):
      def __init__(self, qecc, symbol, **gate_params):
          super().__init__(qecc, symbol, **gate_params)
          # The following are convienent for plotting:
          self.ancilla_x_check = set()
          self.ancilla_z_check = qecc.ancilla_qudit_set
          self._create_checks()
          self.set_logical_ops()
          # Must be called at the end of initiation.
          self._compile_circuit(self.abstract_circuit)

Here, the method ``_create_checks`` is used to create check by first making a shallow␣
↪copy of the ``abstract_circuit`` of the ``InstrSynExtraction`` class. After doing␣
↪this we add :math:`|0\rangle` initialization of the data qubits on the 0th tick.
```

The `_create_checks` method is as follows:

```
def _create_checks(self):
    # Get an instance of the syndrome extraction instruction
    syn_ext = qecc.instruction('instr_syn_extract', **self.gate_params)
    # Make a shallow copy of the abstract circuits.
    self.abstract_circuit = syn_ext.abstract_circuit.copy()
    # Add it the initialization of the data qubits
    data_qudits = set(qecc.data_qudit_set)
    self.abstract_circuit.append('init |0>', locations=data_qudits, tick=0)
}
```

The `set_logical_ops` method is similar to the of method of the same name in `InstrSynExtraction`. The difference for this class is that a logical zero-state is encoded by the logical operator. Because of this, `logical_stabilizers` is set to `['Z']` and `logical_signs` is set to `[0]`.

```
def set_logical_ops(self):
    data_qubits = set(self.qecc.data_qudit_set)
    self.initial_logical_ops = [
        {'X': QuantumCircuit([{'X': {0}}]),
         'Z': QuantumCircuit([{'Z': {0}}])}  ]
    self.final_logical_ops = [
        {'X': QuantumCircuit([{'X': {0}}]),
         'Z': QuantumCircuit([{'Z': data_qubits}])}  ]
    self.logical_stabilizers = ['Z']
    self.logical_signs = [0]
```

Logical Gate Classes

We now construct the `LogicalClass` classes. The construction of these classes is relatively simple compared to the create of `LogicalInstruction` classes.

To begin, we write the class representing the logical identity called `GateIdentity`:

```
  class GateIdentity(LogicalGate):
      def __init__(self, qecc, symbol, **gate_params):
          super().__init__(qecc, symbol, **gate_params)
          self.expected_params(gate_params, {'num_syn_extract', 'error_free', 'random_
↪outcome'})
          self.num_syn_extract = gate_params.get('num_syn_extract', qecc.length)
          self.instr_symbols = ['instr_syn_extract'] * self.num_syn_extract


Here, the initialization method includes the argument ``qecc`` and the argument␣
↪``symbol``. These are the ``qecc`` instance of the ``LogicalGate`` class and the␣
↪string used to represent the ``LogicalGate``, respectively. The initialization␣
↪method also accepts a keyword arguments, which are stored in the dictionary ``gate_
↪params`` and may be used to alter the ``LogicalGate`` and associated␣
↪``LogicalInstructions``.
```

The method `expected_params` determines the keyword arguments that are accepted from `gate_params`. The number of syndrome extraction rounds equal to `'num_syn_extract'`. in the `gate_params` dictionary. Finally, a list of `LogicalInstruction` symbols is stored in the variable `instr_symbols`. The `instr_symbols` indicates the order of `LogicalInstructions` that the gate represents. The correspondence between the `LogicalInstruction` classes and symbols was established by the `sym2instruction_class` method of the `ZRepetition` class.

We will also create a `LogicalGate` class the represents the initialization of logical zero:

```
  class GateInitZero(LogicalGate):
      def __init__(self, qecc, symbol, **gate_params):
          super().__init__(qecc, symbol, **gate_params)
          self.expected_params(gate_params, {'num_syn_extract', 'error_free', 'random_
↪outcome'})
          self.num_syn_extract = gate_params.get('num_syn_extract', 0)
          self.instr_symbols = ['instr_init_zero']
          syn_extract = ['instr_syn_extract'] * self.num_syn_extract
          self.instr_symbols.extend(syn_extract)

Here, all the methods function the same way as those in the ``GateIdentity`` class.
```

Example Usage

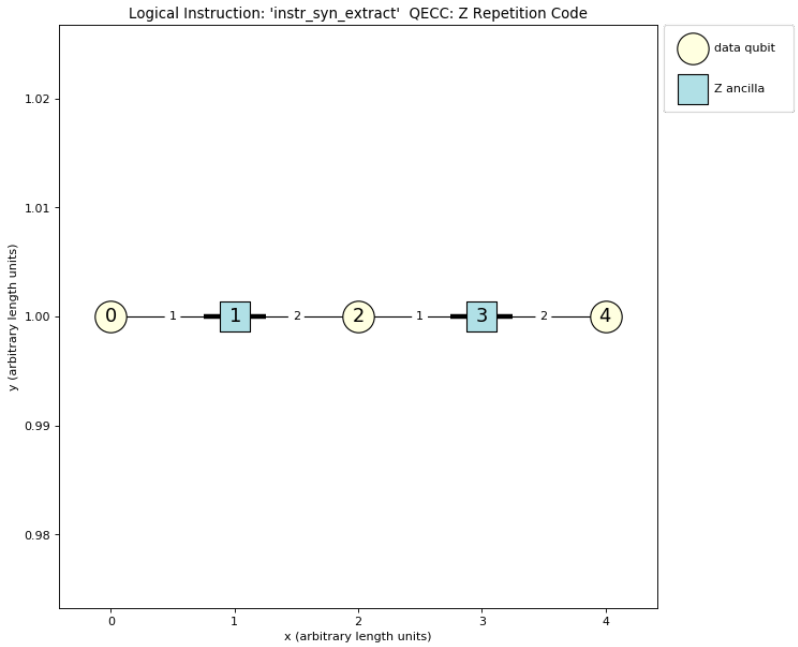
Now we will look at a small example of using the `ZRepetition` class that we created. We begin by importing the class from the `zrepetition.py` script and creating an instance of length three:

```
from zrepetition import ZRepetition
qecc = ZRepetition(length=3)
```

Now that we have created an instance, we will use the `plot` method that is inherited by the syndrome-extraction instruction:

```
qecc.instruction('instr_syn_extract').plot()
```

This code results in the plot of the length three repetition code:



The `ZRepetition` class can be used just like any other `qecc` that comes with PECOS. For example, we can run the following simulation:

```
>>> import pecos as pc
>>> depolar = pc.error_gens.DepolarGen(model_level='code_capacity')
>>> logic = pc.circuits.LogicalCircuit()
>>> logic.append(qecc.gate('ideal init |0>'))
>>> logic.append(qecc.gate('I'))
>>> circ_runner = pc.circuit_runners.Standard(seed=3)
>>> state = circ_runner.init(qecc.num_qudits)
>>> meas, err = circ_runner.run_logic(state, logic, error_gen=depolar, error_params={
↪'p': 0.1})
>>> meas
{(1, 2): {3: {3: 1}}}
>>> err
{(1, 2): {0: {'after': QuantumCircuit([{'X': {4}}])}}}
```

3.3.3 Gate-wise Error Models

The `GatewiseGen` is an `error_gen` that allows users to design error models where gates can be applied according to classical probability distributions that are specified for individual ideal gates or groups of ideal gates. To being we write the following:

```
>>> import pecos as pc
>>> myerrors = pc.error_gens.GatewiseGen()
```

To randomly add an X error after every Hadamard we write:

```
>>> # Continuing from last example.
>>> myerrors.set_gate_error('H', 'X')
```

Here, the probability of a X error occurring will, by default, equal to the value of the key `'p'` in an `error_params` dictionary that is passed to the `run_logic` method of a `circuit_runner`.

To test the error model we are creating, we can use the `get_gate_error` method to generate errors. The first argument of the method is the ideal gate-symbol. The second, is a set of qudit locations the errors may occur on. The third, is a `error_params` dictionary used to specify the probability of errors. An example of using this method is seen here:

```
>>> # Continuing from last example.
>>> myerrors.get_gate_error('H', {0, 1, 2, 3, 4}, error_params={'p':0.5})
(QuantumCircuit([{'X': {0, 1, 3}}]), QuantumCircuit([]), set())
```

Here, the method returns a tuple. The first element is the error circuit that is applied after the ideal gates. The second, before the ideal gates. The third element is the set of qudit locations corresponding to gate locations of ideal gates to be removed from the ideal quantum-circuit.

Note, by default errors specified by the `set_gate_error` method will be generated after the ideal quantum-gates. To generate errors before the gates, one can set the keyword `after` to `False` when using the `add_gate_error` method.

The probability-parameter used (default being `'p'`) can be changed by using the keyword `error_param`:

```
>>> # Continuing from last example.
>>> myerrors.set_gate_error('H', 'X', error_param='q', after=False)
>>> myerrors.get_gate_error('H', {0, 1, 2, 3, 4}, error_params={'q':0.5})
(QuantumCircuit([]), QuantumCircuit([{'X': {0, 3}}]), set())
```

Here we used the keyword‘‘error_param‘‘ to declare that `'q'` will be used to set the probability of an X error occurring. We also see an example of the keyword `after` being used to indicate that errors should be applied before the ideal gates rather than after.

Besides specifying errors of a single gate-type, we can declare a set of errors to be uniformly drawn from:

```
>>> # Continuing from last example.
>>> myerrors.set_gate_error('X', {'X', 'Y', 'S'}, error_param='r')
>>> myerrors.get_gate_error('X', {0, 1, 2, 3, 4}, error_params={'r':0.5})
(QuantumCircuit([{'S': {3, 4}}]), QuantumCircuit([]), set())
```

Such uniform error-distributions can be made for two-qubit gates as well:

```
>>> # Continuing from last example.
>>> myerrors.set_gate_error('CNOT', {('I', 'X'), ('X', 'X'), 'CNOT'}, error_param='r')
>>> myerrors.get_gate_error('CNOT', {(0,1), (2,3), (4,5), (6,7), (8,9)}, error_params=
↪{'r':0.8})
(QuantumCircuit([{'CNOT': {(0, 1), (4, 5), (8, 9)}, 'X': {3, 6, 7}, 'I': {2}}]),␣
↪QuantumCircuit([]), set())
```

Here we see that two-qubit gates or tuples of single-qubit gates can be supplied as errors.

Other distributions besides the uniform distribution can be specified by passing a callable, such as a function or a method. An example is seen in the following:

```
>>> # Continuing from last example.
>>> import random
>>> def error_func(after, before, replace, location, error_params):
...     s = error_params['s']
...     rand = random.triangular(0, 1, 0.6)
...     if rand < 0.6:
...         err = 'Q'
...     elif rand < 0.7:
...         err = 'S'
```

(continues on next page)

(continued from previous page)

```
...     else:
...         err ='R'
...     before.update(err, {location}, emptyappend=True)
```

Here, callables that are used to create unique error distributions must take the arguments `after`, `before`, `replace`, `location`, and `error_params`. The variables `after` and `before` are `QuantumCircuits` representing the errors that are applied after and before the ideals gates of a single tick, respectively. The variable `replace` is the set of qubit gate-locations of the ideals gates to be removed from the ideal quantum-circuit. These callables are called only if error occurs according to the probability of an associated error parameter, which we will see later how to set. The `location` variable is the qudit index or tuple of qudit indices where the error has occurred. The variable `error_params` is the dictionary of error parameters that are being used to determine the probability distribution of errors. In the above callable, we see a triangular distribution being used to apply quantum errors. Note that the callable is responsible for updating `QuantumCircuits` `after`, `before`, `replace` as appropriate.

To use callables to generate errors, we can call the `set_gate_error` method in the following manner:

```
>>> # Continuing from last example.
>>> myerrors.set_gate_error('Y', error_func, error_param='s')
>>> myerrors.get_gate_error('Y', {0, 1, 2, 3, 4}, error_params={'s':0.5})
(QuantumCircuit([]), QuantumCircuit([{'R': {0, 4}, 'Q': {1, 2}}]), set())
```

Here we set the probability of `error_func` being called to generate errors using the `error_params` keyword argument.

There are two special gate-symbols for which error distributions can be assigned to. These special symbols are `'data'` and `'idle'`. The error distribution associated with `'data'` is used to generate errors at the beginning of each `LogicalInstruction` for each data qudit. An error distribution associated with the `'idle'` symbol is used to generate errors whenever a qubit is not acted on by a quantum operation during a `LogicalCircuit`.

An example of setting the errors of a `'data'` and `'idle'` can see here:

```
>>> # Continuing from last example.
>>> myerrors.set_gate_error('data', 'X', error_param='q')
>>> myerrors.set_gate_error('idle', 'Y', error_param='s')
```

Besides specifying errors for individual gate-types, one can specify errors for a group of gates. To do this one may define a gate group and set the error distribution for this group:

```
>>> # Continuing from last example.
>>> myerrors.set_gate_group('measurements', {'measure X', 'measure Y', 'measure Z'})
>>> myerrors.set_group_error('measurements', {'X', 'Y', 'Z'}, error_param='m')
```

Note, `set_group_error` will override the error distribution of any gate belonging to the gate group.

The gate groups that are defined by default can be found by running:

```
>>> newerrors = pc.error_gens.GatewiseGen()
>>> newerrors.gate_groups
{'measurements': {'measure X', 'measure Y', 'measure Z'},
 'inits': {'init |+>', 'init |+i>', 'init |->', 'init |-i>', 'init |0>', 'init |1>'},
 'two_qubits': {'CNOT', 'CZ', 'G', 'SWAP'},
 'one_qubits': {'F1', 'F1d', 'F2', 'F2d', 'F3', 'F3d', 'F4', 'F4d', 'H', 'H+y-z',
↪'H+z+x', 'H-x+y', 'H-x-y', 'H-y-z', 'H-z-x', 'H1', 'H2', 'H3', 'H4', 'H5', 'H6', 'I
↪', 'Q', 'Qd', 'R', 'Rd', 'S', 'Sd', 'X', 'Y', 'Z'}}
```

Here the keys are symbols representing the gate groups and the values are the set of gate symbols belong to the corresponding gate group. These gate groups (`'measurements'`, `'inits'`, `'two_qubits'`, and `'one_qubits'`)

can be redefined by the user.

Example: The Symmetric Depolarizing-channel

As an example, the circuit-level symmetric depolarizing-channel is modeled by `DepolarGen` as discussed in *ths page*, can be represented by the `GatewiseGen` class as follows:

```
depolar_circuit = pc.error_gens.GatewiseGen()
set_gate_group('Xinit', {'init |+>', 'init |->'})
set_gate_group('Yinit', {'init |+i>', 'init |-i>'})
set_gate_group('Zinit', {'init |0>', 'init |1>'})
depolar_circuit.set_group_error('Xinit', 'Z')
depolar_circuit.set_group_error('Yinit', 'Z')
depolar_circuit.set_group_error('Zinit', 'X')
depolar_circuit.set_gate_error('measure X', 'Z', after=False)
depolar_circuit.set_gate_error('measure Y', 'Z', after=False)
depolar_circuit.set_gate_error('measure Z', 'X', after=False)
depolar_circuit.set_group_error('one_qubits', {'X', 'Y', 'Z'})
depolar_circuit.set_group_error('two_qubits', {('I', 'X'), ('I', 'Y'), ('I', 'Z'), ('X
↪', 'I'), ('X', 'X'), ('X', 'Y'), ('X', 'Z'), ('Y', 'I'), ('Y', 'X'), ('Y', 'Y'), ('Y
↪', 'Z'), ('Z', 'I'), ('Z', 'X'), ('I', 'Y'), ('Z', 'Z')})
```

Example: The Amplitude-dampening Channel

The stochastic circuit-level amplitude-dampening channel can be described as:

```
amp_damp = pc.error_gens.GatewiseGen()
amp_damp.set_group_error('inits', 'init |0>')
amp_damp.set_gate_error('measurements', 'init |0>',   after=False)
amp_damp.set_group_error('one_qubits', 'init |0>')
amp_damp.set_group_error('two_qubits', {('I', 'init |0>'), ('init |0>', 'I'), ('init␣
↪|0>', 'init |0>')})
```

3.3.4 Monte Carlo Script

In this appendix, I present how PECOS can be used to create a script to runs a Monte Carlo simulation to determine logical error-rates versus physical error-rates for a fixed distance of a medial surface-code patch. The break-even point where the physical error-rate equals the logical error-rate is known as the *pseudo-threshold*. The *threshold* is the value the pseudo-threshold converges to as the distance of the code approaches infinity.

We begin by creating a Python script `error_rates.py` and importing NumPy and PECOS:

```
import numpy as np
import pecos as pc
```

For this example, we will evaluate the identity gate of `SurfaceMedial4444` and start in the ideal logical zero-state:

```
surface = pc.qeccs.SurfaceMedial4444(distance=3)
logic = pc.circuits.LogicalCircuit(layout=surface.layout)
logic.append(surface.gate('ideal init |0>'))
logic.append(surface.gate('I', num_syn_extract=1))
circ_runner = pc.circuit_runners.Standard(seed=0)
logical_ops = surface.instruction('instr_syn_extract').final_logical_ops[0]
```

Here we also initialize the `circuit_runner` we will use and create the variable `logical_ops`, which stores the logical operations of the QECC. This can be used to determine the logical error-rate since we can track whether errors flip the signs of the logical operators.

Now we choose the depolarizing channel as our noise model (see Section *Error Generators*) and the MWPM decoder (see *Decoders*) to interpret syndromes and determine recovery operations:

```
depolar = pc.error_gens.DepolarGen(model_level='code_capacity')
decode = pc.decoders.MWPM2D(surface).decode
```

We next create the function `determine_fails` to decide if logical error occurs by examining whether, after applying a recovery operation to the state, errors have flipped logical Z. Note, since we are just protecting a logical zero-state we are only concerned with errors that flip the sign of the logical Z operator. Z errors do not affect the state.

The `determine_fails` function is:

```
def determine_fails(meas, decoder, circ_runner, state, logical_ops, fails):
    if meas:
        recovery = decoder(meas)
        circ_runner.run_circuit(state, recovery)
    sign = state.logical_sign(logical_ops['Z'], logical_ops['X'])
    fails += sign
    return fails
```

We are now almost ready to define the Monte Carlo loop. First, however, we set `runs` to represent the number of evaluations we will make per physical error-rate. Next, we add the variable `ps`, which is set to an array of 10 linearly space points between 0.1 and 0.3 to serve as the physical error-rates that we will evaluate. This array is created by NumPy's `linspace` function. Finally, we include the variable `plog`, which stores the logical error-rates we find corresponding to the physical error-rates in `ps`. All of this is done in the following lines:

```
runs = 10000
ps = np.linspace(0.1, 0.4, 10)
plog = []
```

We now create the Monte Carlo loop, which prepares a fresh initial state, applies depolarizing noise with a probability chosen by looping over `ps`, and counts the number of failures (logical flips) to determine the logical error-rate, which is stored in `plog`:

```
for p in ps:
    fails = 0
    for i in range(runs):
        state = circ_runner.init(surface.num_qudits)
        meas, _ = circ_runner.run_logic(state, logic, error_gen=depolar, error_params=
↪{'p': p})
        fails = determine_fails(meas, decoder, circ_runner, state, logical_ops, fails)
    plog.append(fails / runs)
print('ps=', list(ps))
print('plog=', plog)
```

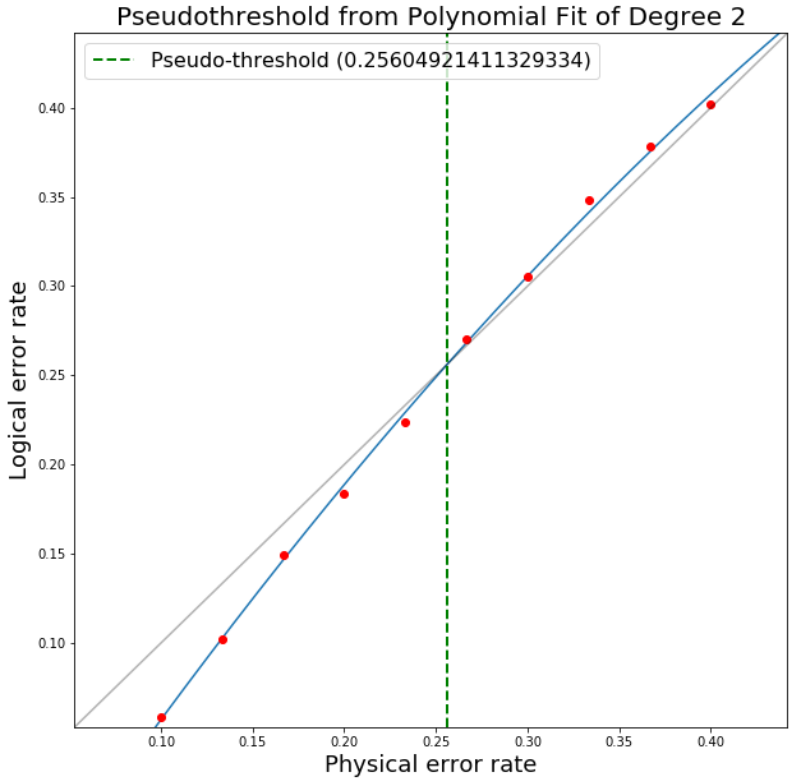
When this script is ran, an example output is:

```
ps= [0.1, 0.13333333333333336, 0.1666666666666669, 0.2, 0.23333333333333336, 0.
↪2666666666666667, 0.30000000000000004, 0.33333333333333337, 0.3666666666666667, 0.4]
plog= [0.0588, 0.102, 0.1497, 0.1835, 0.2241, 0.2702, 0.3052, 0.3485, 0.3783, 0.4017]
```

One can then use plotting packages such as Matplotlib to produce plots as appropriate for the data. PECOS provides a tool for quickly plotting and evaluating logical vs physical error-rates:

```
from pecos.tools import plot_pseudo
plot_pseudo(deg=2,plist=ps,plog=plog)
```

Running this tool results in the plot:



The script described in this appendix can be used as a basis for developing other Monte Carlo simulation scripts for evaluating pseudo-thresholds or thresholds.

3.4 Change Log

3.4.1 Planned Features

The following features are intended to be included:

- Apply error models directly to `QuantumCircuits`.
- Include simulator extensions for popular quantum software that include quantum simulators.
- Facilitate the prototyping of stabilizer `qeccs`.

3.4.2 Releases

PECOS 0.1.0

Release date: October 3, 2018

Initial release.

3.5 Bibliography

3.6 Todo List

Todo: Discuss the leakage error model when it is verified...

original entry

Todo: Discuss the simulator extensions when finished...

original entry

Todo: Write about the available tools.

original entry

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[LRA14] Andrew J. Landahl and Ciarán Ryan-Anderson. Quantum computing by color-code lattice surgery. 2014. eprint: arXiv:1407.5103

[Den+02] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. "Topological quantum memory". In: J. Math. Phys. 43.9 (Sept. 2002), pp. 4452–4505. doi: 10.1063/1.1499754. arXiv:quant-ph/0110143.