PyMatching

Release 0.7.0

Oscar Higgott

May 23, 2022

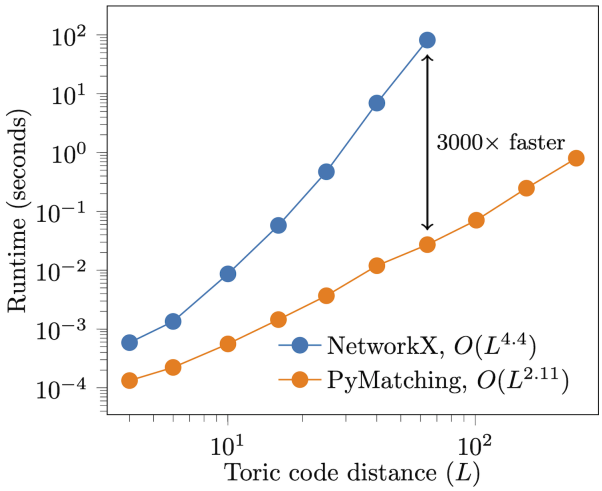
CONTENTS:

PyMatching is a Python package for decoding quantum codes with the minimum-weight perfect matching (MWPM) decoder, and is designed to be fast and easy to use.

While a Python package such as NetworkX can also be used to implement MWPM, it is far too slow to be used for large fault-tolerance simulations, which often require matching graphs with many thousands of nodes. On the other hand, the widely used C++ BlossomV library is fast, but using it to decode quantum codes also requires path-finding algorithms, which must also be implemented in C++ for a fast implementation. Furthermore, attempting to solve the full matching problem even with BlossomV can become prohibitively expensive for matching graphs with more than a few thousand nodes, since the complexity is worse than quadratic in the number of nodes. BlossomV is also not open-source since it does not have a permissive license.

PyMatching is typically faster than a BlossomV/C++ implementation of the full matching problem, while being easy to use in conjunction with numpy, scipy and NetworkX using the Python bindings. The core algorithms and data structures are implemented in C++ for good performance (with the help of the open-source LEMON and Boost Graph libraries), using a local variant of the matching decoder given in the Appendix of https://arxiv.org/abs/2010.09626, which empirically has an average runtime roughly linear in the number of nodes and gives the same output as full matching in practice. Since PyMatching uses the open-source LEMON C++ library for the Blossom algorithm, which has similar performance to Kolmogorov's BlossomV library, both PyMatching and its dependencies have permissive licenses. PyMatching can be applied to any quantum code for which defects come in pairs (or in isolation at a boundary), and it does not require knowledge of the specific geometry used.

Compared to a pure Python NetworkX implementation of MWPM, PyMatching can be orders of magnitude faster, as shown here for the toric code under an independent noise model at $p = 0.05$:



For more information, please also see the PyMatching paper. To make a feature request or report a bug, please visit the PyMatching GitHub repository.

CHAPTER

ONE

INSTALLATION

PyMatching can be downloaded and installed from PyPI with the command:

```
pip install pymatching
```

This is the recommended way to install PyMatching since pip will fetch the pre-compiled binaries, rather than building the C++ extension from source on your machine. Note that PyMatching requires Python 3.

If instead you would like to install PyMatching from source, clone the repository (using the *–recursive* flag to include the lib/pybind11 submodule) and then use *pip* to install:

```
git clone --recursive https://github.com/oscarhiggott/PyMatching.git
pip install -e ./PyMatching
```

The installation may take a few minutes since the C++ extension has to be compiled. If you'd also like to run the tests, first install pytest, and then run:

```
pytest ./PyMatching/tests
```

CHAPTER

TWO

USAGE

2.1 Getting Started

Most of the functionality of PyMatching is available through the `pymatching.matching.Matching` class, which can be imported in Python with:

[1]:
```
from pymatching import Matching
```

The `Matching` class is used to represent the X-type or Z-type matching graph of a CSS quantum code for which syndrome defects come in pairs (or in isolation at a boundary). Each edge in the matching graph corresponds to a single error, and each node corresponds to a stabiliser measurement (or a boundary). The simplest way to construct a `Matching` object is from the X or Z check matrix of the code, which can be given as a numpy or a scipy array. For example, we can construct the Z-type matching graph for a five-qubit quantum bit-flip repetition code (which has Z stabilisers $ZZIII$, $IZZII$, $IIZZI$ and $IIIZZ$) from the Z check matrix using:

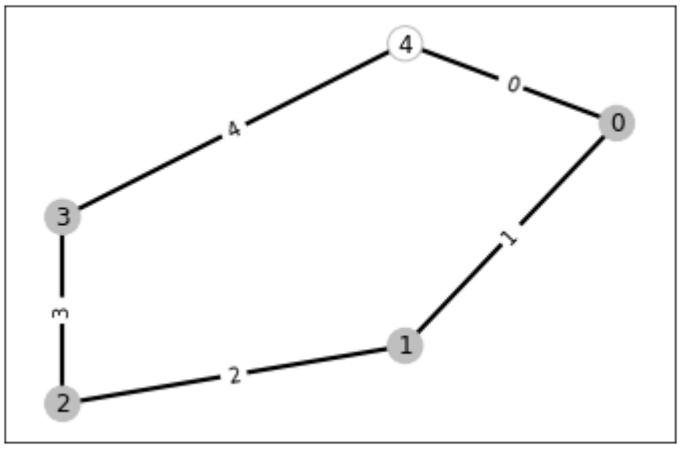
[2]:
```
import numpy as np

"""
Each column of Hz corresponds to an X error on a qubit, and each
row corresponds to a Z stabiliser.

Hz[i,j]==1 if Z stabiliser i acts non-trivially
on qubit j, and is 0 otherwise.
"""
Hz = np.array([
    [1,1,0,0,0],
    [0,1,1,0,0],
    [0,0,1,1,0],
    [0,0,0,1,1]
])

m = Matching(Hz)
m
```

[2]:
```
<pymatching.Matching object with 4 detectors, 1 boundary node, and 5 edges>
```

Note that, since two qubits (0 and 4) are incident to only a single stabiliser, a boundary node has automatically been created in the matching graph, and is connected to the stabilisers acting non-trivially on qubits 0 and 4. The weights of all edges in the matching graph default to 1.0, unless they are specified using the `spacelike_weights` parameter.

We can visualise the matching graph using the `Matching.draw()` method:

[3]:
```
%matplotlib inline
m.draw()
```



[ ]:
```
```

Note that the stabiliser nodes are shown as filled circles, and the boundary node (labelled 4) is shown as a hollow circle. Each edge is labelled with its `fault_ids` attribute, which gives the id (or id's) of any self-inverse faults (such as frame changes) which are flipped when the edge is flipped. When a `pymatching.Matching` object is constructed from a check matrix `H` as done here, each edge is given a `fault_ids` attribute equal to the index of its column in `H`. Since here we chose to define `H` from the Z stabilisers of the code, each column corresponds to a single physical Pauli X error on a physical qubit (so there is a one-to-one correspondence between each self-inverse fault and each qubit). Note that in earlier versions of PyMatching, `fault_ids` was instead named `qubit_id`, and as a result `qubit_id` is still accepted instead of `fault_ids` as an argument when constructing `Matching` objects in order to maintain backward compatibility.

If X errors occur on the third and fourth qubits we have a binary noise vector:

[4]:
```
noise = np.array([0,0,1,1,0])
```

and the resulting syndrome vector is:

[5]:
```
z = Hz@noise % 2
print(z)
```

```
[0 1 0 1]
```

This syndrome vector `z` can then be decoded simply using:

[6]:
```
c = m.decode(z)
print("c: {}, of type {}".format(c, type(c)))
```

```
c: [0 0 1 1 0], of type <class 'numpy.ndarray'>
```

where `c` is the X correction operator (i.e. $IIXXI$).

Note that for larger check matrices you may instead prefer to use a scipy sparse matrix to represent the check matrix:

[7]:
```
import scipy

Hz = scipy.sparse.csr_matrix(Hz)
```

(continues on next page)

(continued from previous page)

```
m = Matching(Hz)
m
```

[7]:
```
<pymatching.Matching object with 4 detectors, 1 boundary node, and 5 edges>
```

2.2 Noisy Syndromes

2.2.1 Spacetime matching graph

If stabiliser measurements are instead noisy, then each stabiliser measurement must be repeated, with each defect in the matching graph corresponding to a change in the syndrome (see IV B of this paper). We will repeat each stabiliser measurement 5 times, with each qubit suffering an X error with probability `p`, and each stabiliser will be measured incorrectly with probability `q`. Spacelike edges will be weighted with $\log((1-p)/p)$ and timelike edges will be weighted with $\log((1-q)/q)$. The Matching object representing this spacetime matching graph can be constructed using:

[8]:
```
repetitions=5
p = 0.05
q = 0.05
m2d = Matching(Hz,
               spacelike_weights=np.log((1-p)/p),
               repetitions=repetitions,
               timelike_weights=np.log((1-q)/q)
              )
```

2.2.2 Simulate noisy syndromes

Now if each qubit suffers an X error with probability `p` in each round of stabiliser measurements, the errors on the data qubits can be given as a 2D numpy array:

[9]:
```
num_stabilisers, num_qubits = Hz.shape
np.random.seed(1) # Keep RNG deterministic
noise = (np.random.rand(num_qubits, repetitions) < p).astype(np.uint8)
noise # New errors in each time step
```

[9]:
```
array([[0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

[10]:
```
noise_cumulative = (np.cumsum(noise, 1) % 2).astype(np.uint8)
noise_total = noise_cumulative[:,-1] # Total cumulative noise at the last round
noise_cumulative # Cumulative errors in each time step
```

[10]:
```
array([[0, 0, 1, 1, 1],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

The corresponding noiseless syndrome would be:

[11]:
```
noiseless_syndrome = Hz@noise_cumulative % 2
noiseless_syndrome # Noiseless syndrome
```

[11]:
```
array([[0, 0, 1, 1, 1],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0]])
```

We assume each syndrome measurement is incorrect with probability `q`, but that the last round of measurements is perfect to ensure an even number of defects (a simple approximation - the overlapping recovery method could be used in practice):

[12]:
```
syndrome_error = (np.random.rand(num_stabilisers, repetitions) < q).astype(np.uint8)
syndrome_error[:,-1] = 0
syndrome_error # Syndrome errors
```

[12]:
```
array([[0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

[13]:
```
noisy_syndrome = (noiseless_syndrome + syndrome_error) % 2
noisy_syndrome # Noisy syndromes
```

[13]:
```
array([[0, 0, 0, 1, 1],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 1, 1],
       [0, 0, 0, 0, 0]])
```

[14]:
```
noisy_syndrome[:,1:] = (noisy_syndrome[:,1:] - noisy_syndrome[:,0:-1]) % 2
noisy_syndrome # Convert to difference syndrome
```

[14]:
```
array([[0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0]])
```

2.2.3 Decode

Decoding can now be done just by inputting this 2D syndrome vector to the `Matching.decode` method:

[15]:
```
correction = m2d.decode(noisy_syndrome)
correction
```

[15]:
```
array([1, 0, 1, 0, 0], dtype=uint8)
```

And we see that this correction operator successfully corrects the cumulative total noise:

[16]:
```
(noise_total + correction) % 2
```

[16]:
```
array([0, 0, 0, 0, 0], dtype=uint8)
```

2.3 Loading from NetworkX graphs

While it can be convenient to decode directly from the check matrices, especially when simulating under a standard independent or phenomenological noise model, it is sometimes necessary to construct the matching graph nodes, edges, weights and boundaries explicitly. This is useful for decoding under more complicated (e.g. circuit-level) noise models, for which matching graph edges can be between nodes separated in both space and time ("diagonal edges"). There can also be so called "hook errors", which are single faults (matching graph edges) corresponding to errors on two or more qubits. Furthermore, the stabilisers themselves can change as a function of time when using schedule-induced gauge fixing of a subsystem code (see this paper).

To provide the functionality to handle these use cases, PyMatching allows Matching objects to be constructed explicitly from NetworkX graphs.

Each node in the matching graph with n nodes, represented by the `pymatching.Matching` object, should be uniquely identified by an integer between 0 and $n-1$ (inclusive). Edges are then added between these integer nodes, with optional attributes `weight`, `fault_ids` and `error_probability`.

We will again use the five qubit quantum repetition code as an example. This time, nodes 1, 2, 3 and 4 will correspond to stabiliser measurements (detectors), and nodes 0 and 5 will be boundary nodes. We'll start by creating the following NetworkX graph:

[17]:
```
import networkx as nx

p = 0.2
g = nx.Graph()
g.add_edge(0, 1, fault_ids=0, weight=np.log((1-p)/p), error_probability=p)
g.add_edge(1, 2, fault_ids=1, weight=np.log((1-p)/p), error_probability=p)
g.add_edge(2, 3, fault_ids=2, weight=np.log((1-p)/p), error_probability=p)
g.add_edge(3, 4, fault_ids=3, weight=np.log((1-p)/p), error_probability=p)
g.add_edge(4, 5, fault_ids=4, weight=np.log((1-p)/p), error_probability=p)
```

Here each "fault_ids" attribute is used to store the id of the qubit which is acted on by an X error (we assume each stabiliser is a Z-type operator).

Recall that nodes 0 and 5 should be boundary nodes, since they do not correspond to stabilizers/detectors. E.g. the boundary node 0 allows us to associate an edge (0, 1) with a fault mechanism that only flips detector 1. We can specify that nodes 0 and 5 are boundary nodes by setting their optional `is_boundary` attribute to `True`:

[18]:
```
g.nodes[0]['is_boundary'] = True
g.nodes[5]['is_boundary'] = True
```

We now connect these boundary nodes with an edge of `weight` zero, and with `fault_ids` either unspecified or set to `set()` or `-1` (since edges between boundaries do not correspond to Pauli errors):

[19]:
```
g.add_edge(0, 5, weight=0.0, fault_ids=-1, error_probability=0.0)
```

Here we have used two boundary nodes to demonstrate that multiple boundary nodes can be added. However, usually only one boundary node needs to be added. For example, we could have connected a single boundary node to nodes 1 and 4 instead here.

Just for the purpose of demonstration, we'll assume that there is also an error process that gives a single hook error on qubits 2 and 3, corresponding to a single edge between node 2 and node 4. This error will occur with probability `p2`. This can be added using:

[20]:
```
p2 = 0.12
g.add_edge(2, 4, fault_ids={2, 3}, weight=np.log((1-p2)/p2), error_probability=p2)
```

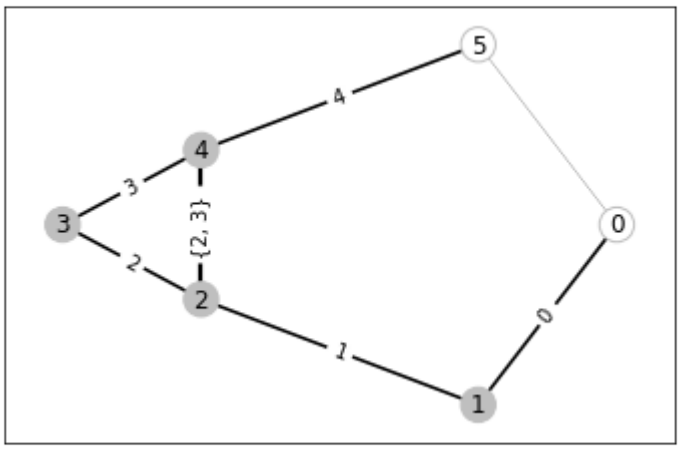
Finally, we can now use this NetworkX graph to construct the `Matching` object:

[21]:
```
m = Matching(g)
m
```

[21]:
```
<pymatching.Matching object with 4 detectors, 2 boundary nodes, and 7 edges>
```

We can also use the `Matching.draw()` method to visualise our matching graph as before:

[22]:
```
%matplotlib inline
m.draw()
```



While the noise and syndrome can be generated separately without PyMatching, if the optional `error_probability` attribute is given to every edge, then the edges can be flipped independently with the `error_probability` assigned to them using the `add_noise` method:

[23]:
```
from pymatching import set_seed
set_seed(1) # Keep RNG deterministic

noise, syndrome = m.add_noise()
print(noise)
print(syndrome)
```

```
[0 1 0 0 0]
[0 1 1 0 0 0]
```

We can now decode as before using the `decode` method:

[24]:
```
correction = m.decode(syndrome)
print((correction+noise)%2)
```

```
[0 0 0 0 0]
```

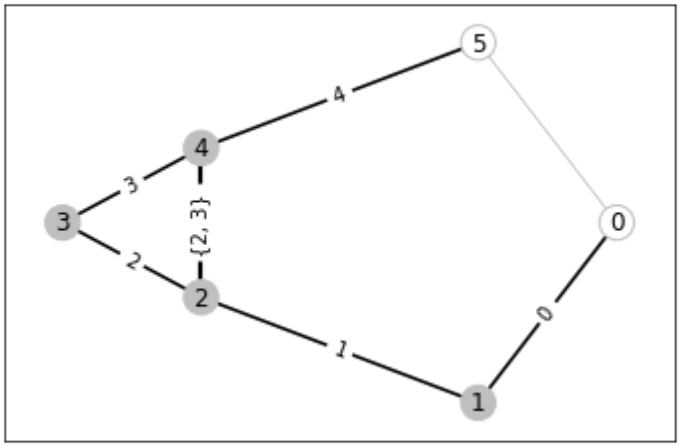
2.4 Constructing a matching graph by adding edges directly to the Matching object

The most direct way to construct a matching graph is to add edges explicitly to the `pymatching.Matching` object. This approach is just as flexible as constructing the graph via NetworkX. For example, the example used in the previous section with NetworkX can instead be constructed as follows:

[25]:
```
p = 0.2
m = Matching()
m.add_edge(0, 1, fault_ids=0, weight=np.log((1-p)/p), error_probability=p)
m.add_edge(1, 2, fault_ids=1, weight=np.log((1-p)/p), error_probability=p)
m.add_edge(2, 3, fault_ids=2, weight=np.log((1-p)/p), error_probability=p)
m.add_edge(3, 4, fault_ids=3, weight=np.log((1-p)/p), error_probability=p)
m.add_edge(4, 5, fault_ids=4, weight=np.log((1-p)/p), error_probability=p)
m.add_edge(0, 5, weight=0.0, fault_ids=set(), error_probability=1.0)
p2 = 0.12
m.add_edge(2, 4, fault_ids={2, 3}, weight=np.log((1-p2)/p2), error_probability=p2)
m.set_boundary_nodes({0, 5})
m
```

[25]:
```
<pymatching.Matching object with 4 detectors, 2 boundary nodes, and 7 edges>
```

[26]:
```
%matplotlib inline
m.draw()
```



2.5 Using Stim to construct a PyMatching matching graph

For simulations of quantum error correcting codes using more realistic circuit-level noise, manually constructing the matching graph can be challenging and time-consuming. Fortunately, these matching graphs can be constructed automatically using Stim for Clifford stabiliser measurement circuits and Pauli noise models. Using Stim, you need only define an annotated stabiliser measurement circuit, from which the matching graph is automatically generated (via a Detector Error Model). Stim can also sample directly from the stabiliser measurement circuit. For more information on combining Stim and PyMatching, see the Stim "getting started" notebook.

CHAPTER

THREE

TORIC CODE EXAMPLE

In this example, we'll use PyMatching to estimate the threshold of the toric code under an independent noise model with perfect syndrome measurements. The decoding problem for the toric code is identical for X-type and Z-type errors, so we will only simulate decoding Z-type errors using X-type stabilisers in this example.

First, we will construct a check matrix H_X corresponding to the X-type stabilisers. Each element $H_X[i,j]$ will be 1 if the ith X stabiliser acts non-trivially on the jth qubit, and is 0 otherwise.

We will construct H_X by taking the hypergraph product of two repetition codes. The hypergraph product code construction $HGP(H_1, H_2)$ takes as input the parity check matrices of two linear codes $C_1 := \ker H_1$ and $C_2 := \ker H_2$. The code $HGP(H_1, H_2)$ is a CSS code with the check matrix for the X stabilisers given by

$$H_X = [H_1 \otimes I_{n_2}, I_{r_1} \otimes H_2^T]$$

and with the check matrix for the Z stabilisers given by

$$H_Z = [I_{n_1} \otimes H_2, H_1^T \otimes I_{r_2}]$$

where H_1 has dimensions $r_1 \times n_1$, H_2 has dimensions $r_2 \times n_2$ and I_l denotes the $l \times l$ identity matrix.

Since we only need the X stabilisers of the toric code with lattice size L, we only need to construct H_X, using the check matrix of a repetition code with length L for both H_1 and H_2:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import hstack, kron, eye, csr_matrix, block_diag


def repetition_code(n):
    """
    Parity check matrix of a repetition code with length n.
    """
    row_ind, col_ind = zip(*((i, j) for i in range(n) for j in (i, (i+1)%n)))
    data = np.ones(2*n, dtype=np.uint8)
    return csr_matrix((data, (row_ind, col_ind)))


def toric_code_x_stabilisers(L):
    """
    Sparse check matrix for the X stabilisers of a toric code with
    lattice size L, constructed as the hypergraph product of
    two repetition codes.
    """
```

(continues on next page)

(continued from previous page)

```
    Hr = repetition_code(L)
    H = hstack(
            [kron(Hr, eye(Hr.shape[1])), kron(eye(Hr.shape[0]), Hr.T)],
            dtype=np.uint8
        )
    H.data = H.data % 2
    H.eliminate_zeros()
    return csr_matrix(H)
```

From the Künneth theorem, the X logical operators of the toric code are given by

$$L_X = \begin{pmatrix} \mathcal{H}^1 \otimes \mathcal{H}^0 & 0 \\ 0 & \mathcal{H}^0 \otimes \mathcal{H}^1 \end{pmatrix}$$

where $\mathcal{H}^0$ and $\mathcal{H}^1$ are the zeroth and first cohomology groups of the length-one chain complex that has the repetition code parity check matrix as its boundary operator. We can construct this matrix with the following function:

[2]:
```
def toric_code_x_logicals(L):
    """
    Sparse binary matrix with each row corresponding to an X logical operator
    of a toric code with lattice size L. Constructed from the
    homology groups of the repetition codes using the Kunneth
    theorem.
    """
    H1 = csr_matrix(([1], ([0],[0])), shape=(1,L), dtype=np.uint8)
    H0 = csr_matrix(np.ones((1, L), dtype=np.uint8))
    x_logicals = block_diag([kron(H1, H0), kron(H0, H1)])
    x_logicals.data = x_logicals.data % 2
    x_logicals.eliminate_zeros()
    return csr_matrix(x_logicals)
```

Now that we have the X check matrix and X logicals of the toric code, we can use PyMatching to simulate its performance using the minimum-weight perfect matching decoder and an error model of our choice.

To do so, we first import the Matching class from PyMatching, and use it to construct a Matching object from the check matrix of the stabilisers:

```
from pymatching import Matching
matching=Matching(H)
```

Constructing the Matching object, while efficient, is often slower than the decoding step itself. As a result, it's best to construct the Matching object only at the beginning of the experiment, and not before every use of the decoder, in order to obtain the best performance.

We also choose a number of trials, `num_trials`. For each trial, we simulate a Z error under an independent noise model, in which each qubit independently suffers a Z error with probability p:

```
noise = np.random.binomial(1, p, H.shape[1])
```

Here, `noise` is a binary vector and `noise[i]` is 1 if qubit i suffers a Z error, and 0 otherwise.

The syndrome of the X stabilisers is then calculated from the dot product (modulo 2) with the X check matrix H:

```
syndrome = H@noise % 2
```

We can now use PyMatching to infer the most probable individual error given the syndrome:

```
correction = matching.decode(syndrome)
```

The total error is now given by the sum (modulo 2) of the noise and the correction:

```
error = (noise + correction) % 2
```

PyMatching is guaranteed to give a correction that returns us to the code space, so a logical Z error will anti-commute with at least one of the X logicals. Therefore a logical error has occurred if the condition

```
np.any(error@logicals.T % 2)
```

is `True`, where `logicals` is the binary matrix L_X with each row corresponding to an X logical.

Taken together, we obtain the following function `num_decoding_failures` that returns the number of logical errors after `num_trials` Monte Carlo trials, simulating an independent error model with error probability `p`, with the X stabiliser check matrix `H` and X logical matrix `logicals`:

[3]:
```
from pymatching import Matching

def num_decoding_failures(H, logicals, p, num_trials):
    matching = Matching(H, spacelike_weights=np.log((1-p)/p))
    num_errors = 0
    for i in range(num_trials):
        noise = np.random.binomial(1, p, H.shape[1])
        syndrome = H@noise % 2
        correction = matching.decode(syndrome)
        error = (noise + correction) % 2
        if np.any(error@logicals.T % 2):
            num_errors += 1
    return num_errors
```

Using this function, we can now estimate the threshold of the toric code by varying the error rate p, for a range of lattice sizes L. Running this next cell may take a couple of minutes:

[4]:
```
%%time

num_trials = 5000
Ls = range(4,14,4)
ps = np.linspace(0.01, 0.2, 9)
np.random.seed(2)
log_errors_all_L = []
for L in Ls:
    print("Simulating L={}...".format(L))
    Hx = toric_code_x_stabilisers(L)
    logX = toric_code_x_logicals(L)
    log_errors = []
    for p in ps:
        num_errors = num_decoding_failures(Hx, logX, p, num_trials)
        log_errors.append(num_errors/num_trials)
    log_errors_all_L.append(np.array(log_errors))
```

```
Simulating L=4...
Simulating L=8...
Simulating L=12...
```

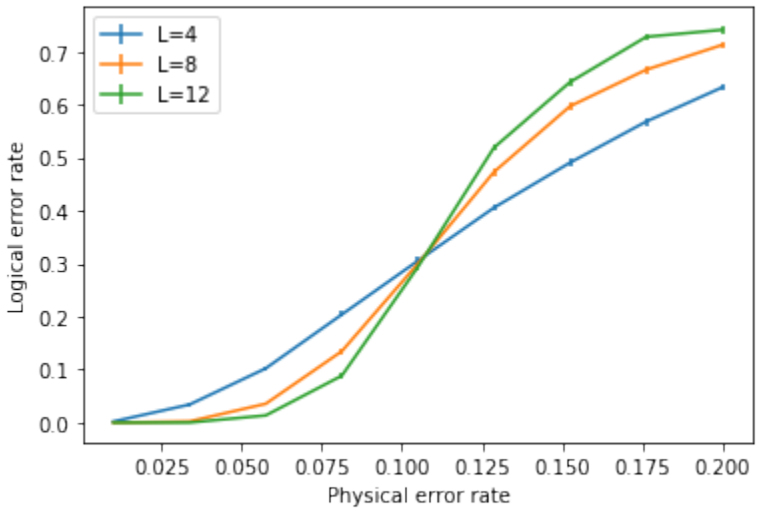
(continues on next page)

(continued from previous page)

```
CPU times: user 3min 20s, sys: 267 ms, total: 3min 20s
Wall time: 3min 20s
```

Finally, let's plot the results! We expect to see a threshold of around 10.3%, although a precise estimate requires using more trials, larger lattice sizes and scanning more values of p:

[5]:
```
%matplotlib inline

plt.figure()
for L, logical_errors in zip(Ls, log_errors_all_L):
    std_err = (logical_errors*(1-logical_errors)/num_trials)**0.5
    plt.errorbar(ps, logical_errors, yerr=std_err, label="L={}".format(L))
plt.xlabel("Physical error rate")
plt.ylabel("Logical error rate")
plt.legend(loc=0);
```



3.1 Noisy syndromes

In the presence of measurement errors, each syndrome measurement is repeated $O(L)$ times, and decoding instead takes place over a 3D matching graph with an additional time dimension (see Section IV B of this paper). The time dimension can be added to the matching graph by specifying the number of repetitions when constructing the matching object:

```
matching = Matching(H, repetitions=T)
```

where here T is the number of repetitions. For decoding, the difference syndrome should be supplied as an $r \times T$ binary numpy matrix, where r is the number of checks (rows in H). The difference syndrome in time step t is the difference (modulo 2) between the syndrome measurement in time step t and $t-1$, and ensures that any single measurement error results in two syndrome defects (at the endpoints of a timelike edge in the matching graph). The last round of syndrome measurements should be free of measurement errors to ensure that the overall syndrome has even parity: when qubits are measured individually at the end of a computation then the final round of syndrome measurement is indeed error-free (stabilisers can be determined exactly in post-processing), however the overlapping recovery method should be implemented when decoding must be completed before all qubits are measured.

The following example demonstrates decoding in the presence of measurement errors using a phenomenological error model. In this error model, in each round of measurements each qubit suffers an error with probability p, and each syndrome is measured incorrectly with probability q.

[6]:
```
def num_decoding_failures_noisy_syndromes(H, logicals, p, q, num_trials, repetitions):
    matching = Matching(H, spacelike_weights=np.log((1-p)/p),
                repetitions=repetitions, timelike_weights=np.log((1-q)/q))
    num_stabilisers, num_qubits = H.shape
    num_errors = 0
    for i in range(num_trials):
        noise_new = (np.random.rand(num_qubits, repetitions) < p).astype(np.uint8)
        noise_cumulative = (np.cumsum(noise_new, 1) % 2).astype(np.uint8)
        noise_total = noise_cumulative[:,-1]
        syndrome = H@noise_cumulative % 2
        syndrome_error = (np.random.rand(num_stabilisers, repetitions) < q).astype(np.
↪uint8)
        syndrome_error[:,-1] = 0 # Perfect measurements in last round to ensure even␣
↪parity
        noisy_syndrome = (syndrome + syndrome_error) % 2
        # Convert to difference syndrome
        noisy_syndrome[:,1:] = (noisy_syndrome[:,1:] - noisy_syndrome[:,0:-1]) % 2
        correction = matching.decode(noisy_syndrome)
        error = (noise_total + correction) % 2
        assert not np.any(H@error % 2)
        if np.any(error@logicals.T % 2):
            num_errors += 1
    return num_errors
```

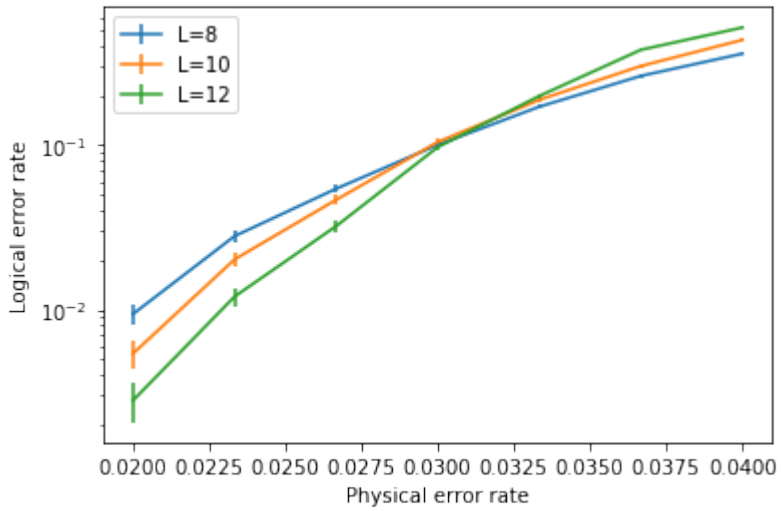
We'll now simulate the performance of the decoder for a range of lattice sizes L and physical error rate p (taking $q = p$) and estimate the threshold. This next cell takes around 20 minutes to execute:

[7]:
```
%%time

num_trials = 5000
Ls = range(8,13,2)
ps = np.linspace(0.02, 0.04, 7)
log_errors_all_L = []
for L in Ls:
    print("Simulating L={}...".format(L))
    Hx = toric_code_x_stabilisers(L)
    logX = toric_code_x_logicals(L)
    log_errors = []
    for p in ps:
        num_errors = num_decoding_failures_noisy_syndromes(Hx, logX, p, p, num_trials, L)
        log_errors.append(num_errors/num_trials)
    log_errors_all_L.append(np.array(log_errors))
```

```
Simulating L=8...
Simulating L=10...
Simulating L=12...
CPU times: user 23min 10s, sys: 1.21 s, total: 23min 11s
Wall time: 23min 11s
```

Plotting the results, we find a threshold of around 3%, consistent with the threshold of 2.9% found in this paper:

[8]:
```
%matplotlib inline

plt.figure()
for L, logical_errors in zip(Ls, log_errors_all_L):
    std_err = (logical_errors*(1-logical_errors)/num_trials)**0.5
    plt.errorbar(ps, logical_errors, yerr=std_err, label="L={}".format(L))
plt.yscale("log")
plt.xlabel("Physical error rate")
plt.ylabel("Logical error rate")
plt.legend(loc=0);
```



CHAPTER

FOUR

CODE DOCUMENTATION

4.1 Matching

class pymatching.matching.**Matching**(*H: Optional[Union[scipy.sparse.base.spmatrix, numpy.ndarray, retworkx.PyGraph, networkx.classes.graph.Graph, List[List[int]]]] = None*, *spacelike_weights: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *error_probabilities: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *repetitions: Optional[int] = None*, *timelike_weights: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *measurement_error_probabilities: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *precompute_shortest_paths: bool = False*, ***kwargs*)

A class for constructing matching graphs and decoding using the minimum-weight perfect matching decoder

The Matching class provides most of the core functionality of PyMatching. A PyMatching object can be constructed from a check matrix with one or two non-zero elements in each column (e.g. the Z or X check matrix of some classes of CSS quantum code), given as a *scipy.sparse* matrix or *numpy.ndarray*, along with additional argument specifying the edge weights, error probabilities and number of repetitions. Alternatively, a Matching object can be constructed from a NetworkX graph, with node and edge attributes used to specify edge weights, fault ids, boundaries and error probabilities.

__init__(*H: Optional[Union[scipy.sparse.base.spmatrix, numpy.ndarray, retworkx.PyGraph, networkx.classes.graph.Graph, List[List[int]]]] = None*, *spacelike_weights: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *error_probabilities: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *repetitions: Optional[int] = None*, *timelike_weights: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *measurement_error_probabilities: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *precompute_shortest_paths: bool = False*, ***kwargs*)

Constructor for the Matching class

Parameters

- **H** (*scipy.spmatrix* or *numpy.ndarray* or *networkx.Graph* object, optional) – The quantum code to be decoded with minimum-weight perfect matching, given either as a binary check matrix (scipy sparse matrix or numpy.ndarray), or as a matching graph (NetworkX graph). Each edge in the NetworkX graph can have optional attributes `fault_ids`, `weight` and `error_probability`. `fault_ids` should be an int or a set of ints. Each fault id corresponds to a self-inverse fault that is flipped when the corresponding edge is flipped. These self-inverse faults could correspond to physical Pauli errors (physical frame changes) or to the logical observables that are flipped by the fault (a logical frame change, equivalent to an obersvable ID in an error instruction in a Stim detector error model). The *fault_ids* attribute was previously named *qubit_id* in an earlier version of PyMatching, and *qubit_id* is still accepted instead of *fault_ids* in order to maintain backward compatibility. Each `weight`

attribute should be a non-negative float. If every edge is assigned an error_probability between zero and one, then the `add_noise` method can be used to simulate noise and flip edges independently in the graph. By default, None

- **spacelike_weights** (*float or numpy.ndarray, optional*) – If *H* is given as a scipy or numpy array, *spacelike_weights* gives the weights of edges in the matching graph corresponding to columns of *H*. If spacelike_weights is a numpy.ndarray, it should be a 1D array with length equal to *H.shape[1]*. If spacelike_weights is a float, it is used as the weight for all edges corresponding to columns of *H*. By default None, in which case all weights are set to 1.0
- **error_probabilities** (*float or numpy.ndarray, optional*) – The probabilities with which an error occurs on each edge corresponding to a column of the check matrix. If a single float is given, the same error probability is used for each edge. If a numpy.ndarray of floats is given, it must have a length equal to the number of columns in the check matrix H. This parameter is only needed for the Matching.add_noise method, and not for decoding. By default None
- **repetitions** (*int, optional*) – The number of times the stabiliser measurements are repeated, if the measurements are noisy. This option is only used if *H* is provided as a check matrix, not a NetworkX graph. By default None
- **timelike_weights** (*float, optional*) – If *H* is given as a scipy or numpy array and *repetitions>1*, *timelike_weights* gives the weight of timelike edges. If a float is given, all timelike edges weights are set to the same value. If a numpy array of size *(H.shape[0],)* is given, the edge weight for each vertical timelike edge associated with the *i`th check (row) of `H* is set to *timelike_weights[i]*. By default None, in which case all timelike weights are set to 1.0
- **measurement_error_probabilities** (*float, optional*) – If *H* is given as a scipy or numpy array and *repetitions>1*, gives the probability of a measurement error to be used for the add_noise method. If a float is given, all measurement errors are set to the same value. If a numpy array of size *(H.shape[0],)* is given, the error probability for each vertical timelike edge associated with the *i`th check (row) of `H* is set to *measurement_error_probabilities[i]*. By default None
- **precompute_shortest_paths** (*bool, optional*) – It is almost always recommended to leave this as False. If the exact matching is used for decoding (setting *num_neighbours=None* in *decode*), then setting this option to True will precompute the all-pairs shortest paths. By default False

Examples

```
>>> import pymatching
>>> import math
>>> m = pymatching.Matching()
>>> m.add_edge(0, 1, fault_ids={0}, weight=0.1)
>>> m.add_edge(1, 2, fault_ids={1}, weight=0.15)
>>> m.add_edge(2, 3, fault_ids={2, 3}, weight=0.2)
>>> m.add_edge(0, 3, fault_ids={4}, weight=0.1)
>>> m.set_boundary_nodes({3})
>>> m
<pymatching.Matching object with 3 detectors, 1 boundary node, and 4 edges>
```

Matching objects can also be created from a check matrix (provided as a scipy.sparse matrix, dense numpy array, or list of lists): >>> import pymatching >>> m = pymatching.Matching([[1, 1, 0, 0], [0, 1, 1, 0], [0,

0, 1, 1]]) >>> m <pymatching.Matching object with 3 detectors, 1 boundary node, and 4 edges>

add_edge(*node1: int*, *node2: int*, *fault_ids: Optional[Union[int, Set[int]]] = None*, *weight: float = 1.0*, *error_probability: Optional[float] = None*, ***kwargs*) → None

Add an edge to the matching graph

Parameters

- **node1** (*int*) – The ID of node1 in the new edge (node1, node2)
- **node2** (*int*) – The ID of node2 in the new edge (node1, node2)
- **fault_ids** (*set[int] or int, optional*) – The IDs of any self-inverse faults which are flipped when the edge is flipped, and which should be tracked. This could correspond to the IDs of physical Pauli errors that occur when this edge flips (physical frame changes). Alternatively, this attribute can be used to store the IDs of any logical observables that are flipped when an error occurs on an edge (logical frame changes). In earlier versions of PyMatching, this attribute was instead named *qubit_id* (since for CSS codes and physical frame changes, there can be a one-to-one correspondence between each fault ID and physical qubit ID). For backward compatibility, *qubit_id* can still be used instead of *fault_ids* as a keyword argument. By default None
- **weight** (*float, optional*) – The weight of the edge, which must be non-negative, by default 1.0
- **error_probability** (*float, optional*) – The probability that the edge is flipped. This is used by the *add_noise()* method to sample from the distribution defined by the matching graph (in which each edge is flipped independently with the corresponding *error_probability*). By default None

Examples

```
>>> import pymatching
>>> m = pymatching.Matching()
>>> m.add_edge(0, 1)
>>> m.add_edge(1, 2)
>>> print(m.num_edges)
2
>>> print(m.num_nodes)
3
```

```
>>> import pymatching
>>> import math
>>> m = pymatching.Matching()
>>> m.add_edge(0, 1, fault_ids=2, weight=math.log((1-0.05)/0.05), error_
↪probability=0.05)
>>> m.add_edge(1, 2, fault_ids=0, weight=math.log((1-0.1)/0.1), error_
↪probability=0.1)
>>> m.add_edge(2, 0, fault_ids={1, 2}, weight=math.log((1-0.2)/0.2), error_
↪probability=0.2)
>>> m
<pymatching.Matching object with 3 detectors, 0 boundary nodes, and 3 edges>
```

add_noise() → Optional[Tuple[numpy.ndarray, numpy.ndarray]]

Add noise by flipping edges in the matching graph with a probability given by the error_probility edge attribute. The `error_probability` must be set for all edges for this method to run, otherwise it returns

None. All boundary nodes are always given a 0 syndrome.

Returns

- *numpy.ndarray of dtype int* – Noise vector (binary numpy int array of length self.num_fault_ids)
- *numpy.ndarray of dtype int* – Syndrome vector (binary numpy int array of length self.num_detectors if there is no boundary, or self.num_detectors+len(self.boundary) if there are boundary nodes)

property boundary: Set[int]

Return the indices of the boundary nodes.

Note that this property is a copy of the set of boundary nodes. In-place modification of the set Matching.boundary will not change the boundary nodes of the matching graph - boundary nodes should instead be set or updated using the *Matching.set_boundary_nodes* method.

Returns The indices of the boundary nodes

Return type set of int

decode(*z: Union[numpy.ndarray, List[int]]*, *num_neighbours: int = 30*, *return_weight: bool = False*) → Union[numpy.ndarray, Tuple[numpy.ndarray, int]]

Decode the syndrome *z* using minimum-weight perfect matching

If the parity of the weight of *z* is odd and the matching graph has one connected component, then an arbitrarily chosen boundary node in `self.boundary` is flipped, and all other stabiliser and boundary nodes are left unchanged. If the matching graph has multiple connected components, then the parity of the syndrome weight within each connected component is checked separately, and if a connected component has odd parity then an arbitrarily chosen boundary node in the same connected component is highlighted. If the parity of the syndrome weight in a connected component is odd, and the same connected component does not have a boundary node, then a *ValueError* is raised.

Parameters

- **z** (*numpy.ndarray*) – A binary syndrome vector to decode. The number of elements in *z* should equal the number of nodes in the matching graph. If *z* is a 1D array, then *z[i]* is the syndrome at node *i* of the matching graph. If *z* is 2D then *z[i,j]* is the difference (modulo 2) between the (noisy) measurement of stabiliser *i* in time step *j+1* and time step *j* (for the case where the matching graph is constructed from a check matrix with *repetitions>1*).
- **num_neighbours** (*int, optional*) – Number of closest neighbours (with non-trivial syndrome) of each matching graph node to consider when decoding. If *num_neighbours* is set (as it is by default), then the local matching decoder in https://arxiv.org/abs/2105.13082 is used, and *num_neighbours* corresponds to the parameter *m* in the paper. It is recommended to leave *num_neighbours* set to at least 20. If *num_neighbours is None*, then instead full matching is performed, with the all-pairs shortest paths precomputed and cached the first time it is used. Since full matching is more memory intensive, it is not recommended to be used for matching graphs with more than around 10,000 nodes, and is only faster than local matching for matching graphs with less than around 1,000 nodes. By default 30
- **return_weight** (*bool, optional*) – If *return_weight==True*, the sum of the weights of the edges in the minimum weight perfect matching is also returned. By default False

Returns

- **correction** (*numpy.ndarray or list[int]*) – A 1D numpy array of ints giving the minimum-weight correction operator as a binary vector. The number of elements in *correction* is one greater than the largest fault ID. The ith element of *correction* is 1 if the minimum-weight

perfect matching (MWPM) found by PyMatching contains an odd number of edges that have i as one of the *fault_ids*, and is 0 otherwise. If each edge in the matching graph is assigned a unique integer in its *fault_ids* attribute, then the locations of nonzero entries in *correction* correspond to the edges in the MWPM. However, *fault_ids* can instead be used, for example, to store IDs of the physical or logical frame changes that occur when an edge flips (see the documentation for `Matching.add_edge` for more information).

- **weight** (*float*) – Present only if *return_weight==True*. The sum of the weights of the edges in the minimum-weight perfect matching.

Examples

```
>>> import pymatching
>>> import numpy as np
>>> H = np.array([[1, 1, 0, 0],
...               [0, 1, 1, 0],
...               [0, 0, 1, 1]])
>>> m = pymatching.Matching(H)
>>> z = np.array([0, 1, 0])
>>> m.decode(z)
array([1, 1, 0, 0], dtype=uint8)
```

Each bit in the correction provided by Matching.decode corresponds to a fault_ids. The index of a bit in a correction corresponds to its fault_ids. For example, here an error on edge (0, 1) flips fault_ids 2 and 3, as inferred by the minimum-weight correction: >>> import pymatching >>> m = pymatching.Matching() >>> m.add_edge(0, 1, fault_ids={2, 3}) >>> m.add_edge(1, 2, fault_ids=1) >>> m.add_edge(2, 0, fault_ids=0) >>> m.decode([1, 1, 0]) array([0, 0, 1, 1], dtype=uint8)

To decode with a phenomenological noise model (qubits and measurements both suffering bit-flip errors), you can provide a check matrix and number of syndrome repetitions to construct a matching graph with a time dimension (where nodes in consecutive time steps are connected by an edge), and then decode with a 2D syndrome (dimension 0 is space, dimension 1 is time): >>> import pymatching >>> import numpy as np >>> np.random.seed(0) >>> H = np.array([[1, 1, 0, 0], … [0, 1, 1, 0], … [0, 0, 1, 1]]) >>> m = pymatching.Matching(H, repetitions=5) >>> data_qubit_noise = (np.random.rand(4, 5) < 0.1).astype(np.uint8) >>> print(data_qubit_noise) [[0 0 0 0 0]

[0 0 0 0 0] [0 0 0 0 1] [1 1 0 0 0]]

```
>>> cumulative_noise = (np.cumsum(data_qubit_noise, 1) % 2).astype(np.uint8)
>>> syndrome = H@cumulative_noise % 2
>>> print(syndrome)
[[0 0 0 0 0]
 [0 0 0 0 1]
 [1 0 0 0 1]]
>>> syndrome[:,:-1] ^= (np.random.rand(3, 4) < 0.1).astype(np.uint8)
>>> # Take the parity of consecutive timesteps to construct a difference␣
↪syndrome:
>>> syndrome[:,1:] = syndrome[:,:-1] ^ syndrome[:,1:]
>>> m.decode(syndrome)
array([0, 0, 1, 0], dtype=uint8)
```

draw() → None

Draw the matching graph using matplotlib

Draws the matching graph as a matplotlib graph. Stabiliser nodes are filled grey and boundary nodes are

filled white. The line thickness of each edge is determined from its weight (with min and max thicknesses of 0.2 pts and 2 pts respectively). Note that you may need to call *plt.figure()* before and *plt.show()* after calling this function.

edges() → List[Tuple[int, int, Dict]]
Edges of the matching graph

Returns a list of edges of the matching graph. Each edge is a tuple *(source, target, attr)* where *source* and *target* are ints corresponding to the indices of the source and target nodes, and *attr* is a dictionary containing the attributes of the edge. The dictionary *attr* has keys *fault_ids* (a set of ints), *weight* (the weight of the edge, set to 1.0 if not specified), and *error_probability* (the error probability of the edge, set to -1 if not specified).

Returns A list of edges of the matching graph

Return type List of (int, int, dict) tuples

load_from_check_matrix(*H: Union[scipy.sparse.base.spmatrix, numpy.ndarray, List[List[int]]]*, *spacelike_weights: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *error_probabilities: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *repetitions: Optional[int] = None*, *timelike_weights: Optional[Union[float, numpy.ndarray, List[float]]] = None*, *measurement_error_probabilities: Optional[Union[float, numpy.ndarray, List[float]]] = None*, ***kwargs*) → None
Load a matching graph from a check matrix

Parameters

- **H** (*scipy.spmatrix* or *numpy.ndarray* or List[List[int]]) – The quantum code to be decoded with minimum-weight perfect matching, given as a binary check matrix (scipy sparse matrix or numpy.ndarray)
- **spacelike_weights** (*float or numpy.ndarray, optional*) – If *H* is given as a scipy or numpy array, *spacelike_weights* gives the weights of edges in the matching graph corresponding to columns of *H*. If spacelike_weights is a numpy.ndarray, it should be a 1D array with length equal to *H.shape[1]*. If spacelike_weights is a float, it is used as the weight for all edges corresponding to columns of *H*. By default None, in which case all weights are set to 1.0
- **error_probabilities** (*float or numpy.ndarray, optional*) – The probabilities with which an error occurs on each edge associated with a column of H. If a single float is given, the same error probability is used for each column. If a numpy.ndarray of floats is given, it must have a length equal to the number of columns in H. This parameter is only needed for the Matching.add_noise method, and not for decoding. By default None
- **repetitions** (*int, optional*) – The number of times the stabiliser measurements are repeated, if the measurements are noisy. By default None
- **timelike_weights** (*float or numpy.ndarray, optional*) – If *repetitions>1*, *timelike_weights* gives the weight of timelike edges. If a float is given, all timelike edges weights are set to the same value. If a numpy array of size *(H.shape[0],)* is given, the edge weight for each vertical timelike edge associated with the *i`th check (row) of `H* is set to *timelike_weights[i]*. By default None, in which case all timelike weights are set to 1.0
- **measurement_error_probabilities** (*float or numpy.ndarray, optional*) – If *repetitions>1*, gives the probability of a measurement error to be used for the add_noise method. If a float is given, all measurement errors are set to the same value. If a numpy array of size *(H.shape[0],)* is given, the error probability for each vertical timelike edge associated with the *i`th check (row) of `H* is set to *measurement_error_probabilities[i]*. This argument can also be given using the keyword argument *measurement_error_probability* to maintain backward compatibility with previous versions of Pymatching. By default None

Examples

```
>>> import pymatching
>>> m = pymatching.Matching([[1, 1, 0, 0], [0, 1, 1, 0], [0, 0, 1, 1]])
>>> m
<pymatching.Matching object with 3 detectors, 1 boundary node, and 4 edges>
```

Matching objects can also be initialised from a sparse scipy matrix: >>> import pymatching >>> from scipy.sparse import csc_matrix >>> H = csc_matrix([[1, 1, 0], [0, 1, 1]]) >>> m = pymatching.Matching(H) >>> m <pymatching.Matching object with 2 detectors, 1 boundary node, and 3 edges>

load_from_networkx(*graph: networkx.classes.graph.Graph*) → None

Load a matching graph from a NetworkX graph

Parameters **graph** (*networkx.Graph*) – Each edge in the NetworkX graph can have optional attributes `fault_ids`, `weight` and `error_probability`. `fault_ids` should be an int or a set of ints. Each fault id corresponds to a self-inverse fault that is flipped when the corresponding edge is flipped. These self-inverse faults could correspond to physical Pauli errors (physical frame changes) or to the logical observables that are flipped by the fault (a logical frame change, equivalent to an obersvable ID in an error instruction in a Stim detector error model). The *fault_ids* attribute was previously named *qubit_id* in an earlier version of PyMatching, and *qubit_id* is still accepted instead of *fault_ids* in order to maintain backward compatibility. Each `weight` attribute should be a non-negative float. If every edge is assigned an error_probability between zero and one, then the `add_noise` method can be used to simulate noise and flip edges independently in the graph.

Examples

```
>>> import pymatching
>>> import networkx as nx
>>> import math
>>> g = nx.Graph()
>>> g.add_edge(0, 1, fault_ids=0, weight=math.log((1-0.1)/0.1), error_
↪probability=0.1)
>>> g.add_edge(1, 2, fault_ids=1, weight=math.log((1-0.15)/0.15), error_
↪probability=0.15)
>>> g.nodes[0]['is_boundary'] = True
>>> g.nodes[2]['is_boundary'] = True
>>> m = pymatching.Matching(g)
>>> m
<pymatching.Matching object with 1 detector, 2 boundary nodes, and 2 edges>
```

load_from_retworkx(*graph: retworkx.PyGraph*) → None

Load a matching graph from a retworkX graph

Parameters **graph** (*retworkx.PyGraph*) – Each edge in the retworkx graph can have dictionary payload with keys `fault_ids`, `weight` and `error_probability`. `fault_ids` should be an int or a set of ints. Each fault id corresponds to a self-inverse fault that is flipped when the corresponding edge is flipped. These self-inverse faults could correspond to physical Pauli errors (physical frame changes) or to the logical observables that are flipped by the fault (a logical frame change, equivalent to an obersvable ID in an error instruction in a Stim detector error model). The *fault_ids* attribute was previously named *qubit_id* in an earlier version of PyMatching, and *qubit_id* is still accepted instead of *fault_ids* in order to maintain backward

compatibility. Each `weight` attribute should be a non-negative float. If every edge is assigned an error_probability between zero and one, then the `add_noise` method can be used to simulate noise and flip edges independently in the graph.

Examples

```
>>> import pymatching
>>> import retworkx as rx
>>> import math
>>> g = rx.PyGraph()
>>> matching = g.add_nodes_from([{} for _ in range(3)])
>>> edge_a =g.add_edge(0, 1, dict(fault_ids=0, weight=math.log((1-0.1)/0.1),␣
↪error_probability=0.1))
>>> edge_b = g.add_edge(1, 2, dict(fault_ids=1, weight=math.log((1-0.15)/0.15),␣
↪error_probability=0.15))
>>> g[0]['is_boundary'] = True
>>> g[2]['is_boundary'] = True
>>> m = pymatching.Matching(g)
>>> m
<pymatching.Matching object with 1 detector, 2 boundary nodes, and 2 edges>
```

property num_detectors: int
The number of detectors in the matching graph. A detector is a node that can have a non-trivial syndrome (i.e. it is a node that is not a boundary node).

Returns The number of detectors

Return type int

property num_edges: int
The number of edges in the matching graph

Returns The number of edges

Return type int

property num_fault_ids: int
The number of fault IDs defined in the matching graph

Returns Number of fault IDs

Return type int

property num_nodes: int
The number of nodes in the matching graph

Returns The number of nodes

Return type int

set_boundary_nodes(*nodes: Set[int]*) → None
Set boundary nodes in the matching graph. This defines the nodes in *nodes* to be boundary nodes.

Parameters **nodes** (*set[int]*) – The IDs of the nodes to be set as boundary nodes

Examples

```
>>> import pymatching
>>> m = pymatching.Matching()
>>> m.add_edge(0, 1)
>>> m.add_edge(1, 2)
>>> m.set_boundary_nodes({0, 2})
>>> m.boundary
{0, 2}
>>> m
<pymatching.Matching object with 1 detector, 2 boundary nodes, and 2 edges>
```

to_networkx() → networkx.classes.graph.Graph
Convert to NetworkX graph

Returns a NetworkX graph corresponding to the matching graph. Each edge has attributes *fault_ids*, *weight* and *error_probability* and each node has the attribute *is_boundary*.

Returns NetworkX Graph corresponding to the matching graph

Return type NetworkX.Graph

to_retworkx() → retworkx.PyGraph
Convert to retworkx graph

Returns a retworkx graph object corresponding to the matching graph. Each edge payload is a `dict` with keys *fault_ids*, *weight* and *error_probability* and each node has a `dict` payload with the key `is_boundary` and the value is a boolean.

Returns retworkx graph corresponding to the matching graph

Return type retworkx.PyGraph

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

D

E

L

M

N

P

S

T