PyMatching

Release 2.1.dev1

Oscar Higgott and Craig Gidney

Feb 06, 2024

CONTENTS:

PyMatching is a fast Python/C++ library for decoding quantum error correcting (QEC) codes using the Minimum Weight Perfect Matching (MWPM) decoder. Given the syndrome measurements from a quantum error correction circuit, the MWPM decoder finds the most probable set of errors, given the assumption that error mechanisms are *independent*, as well as *graphlike* (each error causes either one or two detection events). The MWPM decoder is the most popular decoder for decoding surface codes, and can also be used to decode various other code families, including subsystem codes, honeycomb codes and 2D hyperbolic codes.

Version 2 includes a new implementation of the blossom algorithm which is **100-1000x faster** than previous versions of PyMatching. PyMatching can be configured using arbitrary weighted graphs, with or without a boundary, and can be combined with Craig Gidney's Stim library to simulate and decode error correction circuits in the presence of circuit-level noise. The sinter package combines Stim and PyMatching to perform fast, parallelised monte-carlo sampling of quantum error correction circuits.

Documentation for PyMatching can be found at: pymatching.readthedocs.io

Our paper gives more background on the MWPM decoder and our implementation (sparse blossom) released in PyMatching v2.

To see how stim, sinter and pymatching can be used to estimate the threshold of an error correcting code with circuit-level noise, try out the stim getting started notebook.

CHAPTER

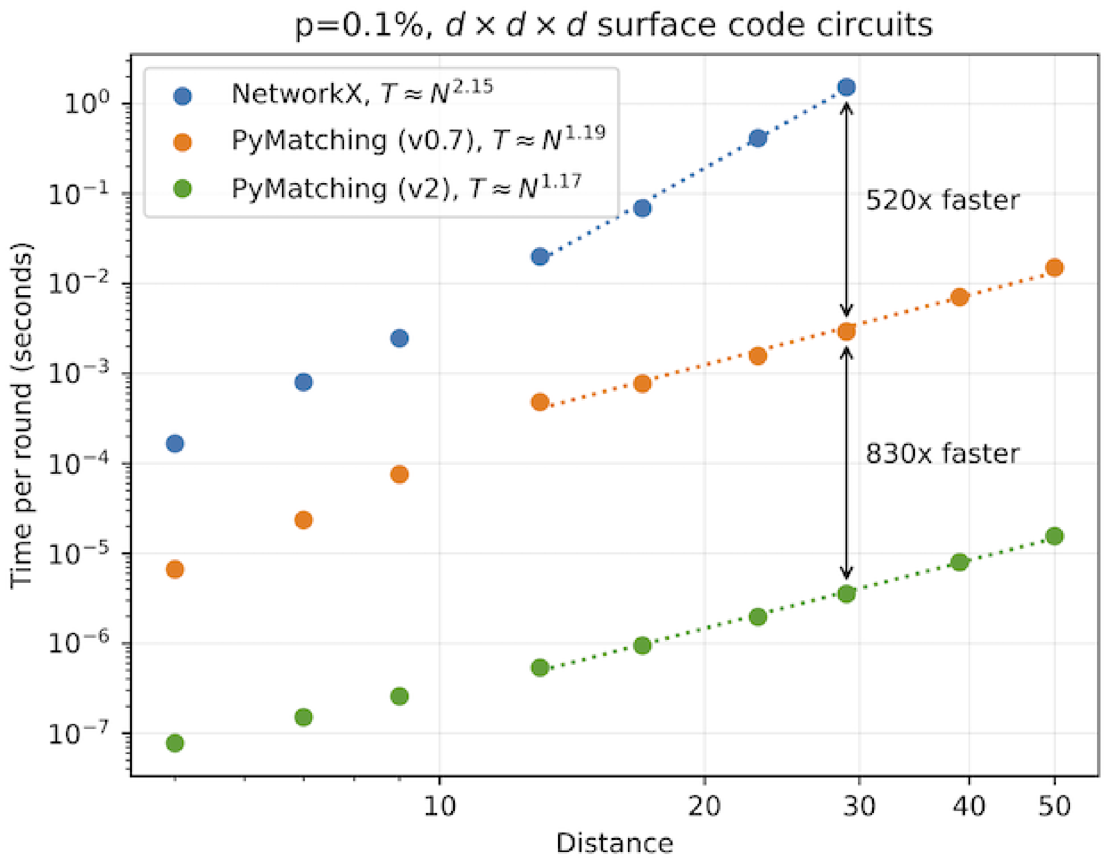
ONE

THE NEW >100X FASTER IMPLEMENTATION FOR VERSION 2

Version 2 features a new implementation of the blossom algorithm, which I wrote with Craig Gidney. Our new implementation, which we refer to as the *sparse blossom* algorithm, can be seen as a generalisation of the blossom algorithm to handle the decoding problem relevant to QEC. We solve the problem of finding minimum-weight paths between detection events in a detector graph *directly*, which avoids the need to use costly all-to-all Dijkstra searches to find a MWPM in a derived graph using the original blossom algorithm. The new version is also exact - unlike previous versions of PyMatching, no approximation is made. See our paper for more details.

Our new implementation is **over 100x faster** than previous versions of PyMatching, and is **over 100,000x faster** than NetworkX (benchmarked with surface code circuits). At 0.1% circuit-noise, PyMatching can decode both X and Z basis measurements of surface code circuits up to distance 17 in under 1 microsecond per round of syndrome extraction on a single core. Furthermore, the runtime is roughly linear in the number of nodes in the graph.

The plot below compares the performance of PyMatching v2 with the previous version (v0.7) as well as with NetworkX for decoding surface code circuits with circuit-level depolarising noise. All decoders were run on a single core of an M1 processor, processing both the X and Z basis measurements. The equations T=N^x in the legend (and plotted as dashed lines) are obtained from a fit to the same dataset for distance > 10, where N is the number of detectors (nodes) per round, and T is the decoding time per round. See the benchmarks folder in the repository for the data and stim circuits, as well as additional benchmarks.



Sparse blossom is conceptually similar to the approach described in this paper by Austin Fowler, although our approach differs in many of the details (as explained in our paper). There are even more similarities with the very nice independent work by Yue Wu, who recently released the fusion-blossom library. One of the differences with our approach is that fusion-blossom grows the exploratory regions of alternating trees in a similar way to how clusters are grown in Union-Find, whereas our approach instead progresses along a timeline, and uses a global priority queue to grow alternating trees. Yue also has a paper coming soon, so stay tuned for that as well.

CHAPTER

TWO

INSTALLATION

The latest version of PyMatching can be downloaded and installed from PyPI with the command:

```
pip install pymatching --upgrade
```

CHAPTER

THREE

USAGE

PyMatching can load matching graphs from a check matrix, a `stim.DetectorErrorModel`, a `networkx.Graph`, a `rustworkx.PyGraph` or by adding edges individually with `pymatching.Matching.add_edge` and `pymatching.Matching.add_boundary_edge`.

3.1 Decoding Stim circuits

PyMatching can be combined with Stim. Generally, the easiest and fastest way to do this is using sinter (use v1.10.0 or later), which uses PyMatching and Stim to run parallelised monte carlo simulations of quantum error correction circuits. However, in this section we will use Stim and PyMatching directly, to demonstrate how their Python APIs can be used. To install stim, run `pip install stim --upgrade`.

First, we generate a stim circuit. Here, we use a surface code circuit included with stim:

```
import numpy as np
import stim
import pymatching
circuit = stim.Circuit.generated("surface_code:rotated_memory_x",
                                 distance=5,
                                 rounds=5,
                                 after_clifford_depolarization=0.005)
```

Next, we use stim to generate a `stim.DetectorErrorModel` (DEM), which is effectively a Tanner graph describing the circuit-level noise model. By setting `decompose_errors=True`, stim decomposes all error mechanisms into *edge-like* error mechanisms (which cause either one or two detection events). This ensures that our DEM is graphlike, and can be loaded by pymatching:

```
model = circuit.detector_error_model(decompose_errors=True)
matching = pymatching.Matching.from_detector_error_model(model)
```

Next, we will sample 1000 shots from the circuit. Each shot (a row of `shots`) contains the full syndrome (detector measurements), as well as the logical observable measurements, from simulating the noisy circuit:

```
sampler = circuit.compile_detector_sampler()
syndrome, actual_observables = sampler.sample(shots=1000, separate_observables=True)
```

Now we can decode! We compare PyMatching's predictions of the logical observables with the actual observables sampled with stim, in order to count the number of mistakes and estimate the logical error rate:

```
num_errors = 0
for i in range(syndrome.shape[0]):
```

(continues on next page)

(continued from previous page)

```
    predicted_observables = matching.decode(syndrome[i, :])
    num_errors += not np.array_equal(actual_observables[i, :], predicted_observables)

print(num_errors)  # prints 8
```

As of PyMatching v2.1.0, you can use `matching.decode_batch` to decode a batch of shots instead. Since `matching.decode_batch` iterates over the shots in C++, it's faster than iterating over calls to `matching.decode` in Python. The following cell is therefore a faster equivalent to the cell above:

```
predicted_observables = matching.decode_batch(syndrome)
num_errors = np.sum(np.any(predicted_observables != actual_observables, axis=1))

print(num_errors)  # prints 8
```

3.2 Loading from a parity check matrix

We can also load a `pymatching.Matching` object from a binary parity check matrix, another representation of a Tanner graph. Each row in the parity check matrix `H` corresponds to a parity check, and each column corresponds to an error mechanism. The element `H[i,j]` of `H` is 1 if parity check `i` is flipped by error mechanism `j`, and 0 otherwise. To be used by PyMatching, the error mechanisms in `H` must be *graphlike*. This means that each column must contain either one or two 1s (if a column has a single 1, it represents a half-edge connected to the boundary).

We can give each edge in the graph a weight, by providing PyMatching with a `weights` numpy array. Element `weights[j]` of the `weights` array sets the edge weight for the edge corresponding to column `j` of `H`. If the error mechanisms are treated as independent, then we typically want to set the weight of edge `j` to the log-likelihood ratio `log((1-p_j)/p_j)`, where `p_j` is the error probability associated with edge `j`. With this setting, PyMatching will find the most probable set of error mechanisms, given the syndrome.

With PyMatching configured using `H` and `weights`, decoding a binary syndrome vector `syndrome` (a numpy array of length `H.shape[0]`) corresponds to finding a set of errors defined in a binary `predictions` vector satisfying `H@predictions % 2 == syndrome` while minimising the total solution weight `predictions@weights`.

In quantum error correction, rather than predicting which exact set of error mechanisms occurred, we typically want to predict the outcome of *logical observable* measurements, which are the parities of error mechanisms. These can be represented by a binary matrix `observables`. Similar to the check matrix, `observables[i,j]` is 1 if logical observable `i` is flipped by error mechanism `j`. For example, suppose our syndrome `syndrome`, was the result of a set of errors `noise` (a binary array of length `H.shape[1]`), such that `syndrome = H@noise % 2`. Our decoding is successful if `observables@noise % 2 == observables@predictions % 2`.

Putting this together, we can decode a distance 5 repetition code as follows:

```
import numpy as np
from scipy.sparse import csc_matrix
import pymatching
H = csc_matrix([[1, 1, 0, 0, 0],
                 [0, 1, 1, 0, 0],
                 [0, 0, 1, 1, 0],
                 [0, 0, 0, 1, 1]])
weights = np.array([4, 3, 2, 3, 4])   # Set arbitrary weights for illustration
matching = pymatching.Matching(H, weights=weights)
prediction = matching.decode(np.array([0, 1, 0, 1]))
print(prediction)  # prints: [0 0 1 1 0]
```

(continues on next page)

(continued from previous page)

```
# Optionally, we can return the weight as well:
prediction, solution_weight = matching.decode(np.array([0, 1, 0, 1]), return_weight=True)
print(prediction)  # prints: [0 0 1 1 0]
print(solution_weight)  # prints: 5.0
```

And in order to estimate the logical error rate for a physical error rate of 10%, we can sample as follows:

```
import numpy as np
from scipy.sparse import csc_matrix
import pymatching
H = csc_matrix([[1, 1, 0, 0, 0],
                [0, 1, 1, 0, 0],
                [0, 0, 1, 1, 0],
                [0, 0, 0, 1, 1]])
observables = csc_matrix([[1, 0, 0, 0, 0]])
error_probability = 0.1
weights = np.ones(H.shape[1]) * np.log((1-error_probability)/error_probability)
matching = pymatching.Matching.from_check_matrix(H, weights=weights)
num_shots = 1000
num_errors = 0
for i in range(num_shots):
    noise = (np.random.random(H.shape[1]) < error_probability).astype(np.uint8)
    syndrome = H@noise % 2
    prediction = matching.decode(syndrome)
    predicted_observables = observables@prediction % 2
    actual_observables = observables@noise % 2
    num_errors += not np.array_equal(predicted_observables, actual_observables)
print(num_errors)  # prints 4
```

Note that we can also ask PyMatching to predict the logical observables directly, by supplying them to the `faults_matrix` argument when constructing the `pymatching.Matching` object. This allows the decoder to make some additional optimisations, that speed up the decoding procedure a bit. The following example uses this approach, and is equivalent to the example above:

```
import numpy as np
from scipy.sparse import csc_matrix
import pymatching

H = csc_matrix([[1, 1, 0, 0, 0],
                [0, 1, 1, 0, 0],
                [0, 0, 1, 1, 0],
                [0, 0, 0, 1, 1]])
observables = csc_matrix([[1, 0, 0, 0, 0]])
error_probability = 0.1
weights = np.ones(H.shape[1]) * np.log((1-error_probability)/error_probability)
matching = pymatching.Matching.from_check_matrix(H, weights=weights, faults_
↪matrix=observables)
num_shots = 1000
num_errors = 0
for i in range(num_shots):
    noise = (np.random.random(H.shape[1]) < error_probability).astype(np.uint8)
    syndrome = H@noise % 2
```

(continues on next page)

(continued from previous page)

```
    predicted_observables = matching.decode(syndrome)
    actual_observables = observables@noise % 2
    num_errors += not np.array_equal(predicted_observables, actual_observables)

print(num_errors)  # prints 6
```

We'll make one more optimisation, which is to use `matching.decode_batch` to decode the batch of shots, rather than iterating over calls to `matching.decode` in Python:

```
import numpy as np
from scipy.sparse import csc_matrix
import pymatching

H = csc_matrix([[1, 1, 0, 0, 0],
                [0, 1, 1, 0, 0],
                [0, 0, 1, 1, 0],
                [0, 0, 0, 1, 1]])
observables = csc_matrix([[1, 0, 0, 0, 0]])
error_probability = 0.1
num_shots = 1000
weights = np.ones(H.shape[1]) * np.log((1-error_probability)/error_probability)
matching = pymatching.Matching.from_check_matrix(H, weights=weights, faults_
↪matrix=observables)
noise = (np.random.random((num_shots, H.shape[1])) < error_probability).astype(np.uint8)
shots = (noise @ H.T) % 2
actual_observables = (noise @ observables.T) % 2
predicted_observables = matching.decode_batch(shots)
num_errors = np.sum(np.any(predicted_observables != actual_observables, axis=1))
print(num_errors)  # prints 6
```

Instead of using a check matrix, the Matching object can also be constructed using the `` `Matching.add_edge `` <https://pymatching.readthedocs.io/en/stable/api.html#pymatching.matching.Matching.add_edge>`_ and `` `Matching.add_boundary_edge `` <https://pymatching.readthedocs.io/en/stable/api.html#pymatching.matching.Matching.add_boundary_edge>`_ methods, or by loading from a NetworkX or rustworkx graph.

For more details on how to use PyMatching, see the documentation.

CHAPTER

FOUR

ATTRIBUTION

When using PyMatching please cite our paper on the sparse blossom algorithm (implemented in version 2):

```
@article{higgott2023sparse,
  title={Sparse Blossom: correcting a million errors per core second with minimum-weight␣
↪matching},
  author={Higgott, Oscar and Gidney, Craig},
  journal={arXiv preprint arXiv:2303.15933},
  year={2023}
}
```

Note: the previous PyMatching paper descibes the implementation in version 0.7 and earlier of PyMatching (not v2).

CHAPTER

FIVE

ACKNOWLEDGEMENTS

We are grateful to the Google Quantum AI team for supporting the development of PyMatching v2. Earlier versions of PyMatching were supported by Unitary Fund and EPSRC.

5.1 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]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import hstack, kron, eye, csc_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 csc_matrix((data, (row_ind, col_ind)))


```

(continues on next page)

(continued from previous page)

```
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.
    """
    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 csc_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 = csc_matrix(([1], ([0],[0])), shape=(1,L), dtype=np.uint8)
    H0 = csc_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 csc_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_shots`. 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:

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

We use this to predict which logical X operators have been flipped:

```
predicted_logicals_flipped = logicals@prediction % 2
```

The actual logicals that were flipped are:

```
actual_logicals_flipped = logicals@noise % 2
```

Our decoder was successful if `actual_logical_observables` equals `predicted_logical_observables`.

Taken together, we obtain the following function `num_decoding_failures` that returns the number of logical errors after `num_shots` 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_via_physical_frame_changes(H, logicals, error_probability, num_
↪shots):
    matching = Matching.from_check_matrix(H, weights=np.log((1-error_probability)/error_
↪probability))
    num_errors = 0
    for i in range(num_shots):
        noise = (np.random.random(H.shape[1]) < error_probability).astype(np.uint8)
        syndrome = H@noise % 2
        prediction = matching.decode(syndrome)
        predicted_logicals_flipped = logicals@prediction % 2
        actual_logicals_flipped = logicals@noise % 2
        if not np.array_equal(predicted_logicals_flipped, actual_logicals_flipped):
            num_errors += 1
    return num_errors
```

We can speed this up slightly by telling PyMatching about the logical operators matrix when we create the `pymatching.Matching` object, using the `faults_matrix` argument. By doing this, `pymatching.Matching.decode` directly predicts which logicals have been flipped. This is a bit faster, as it allows the decoder to make some more optimisations.

[4]:
```
def num_decoding_failures(H, logicals, error_probability, num_shots):
    matching = Matching.from_check_matrix(H, weights=np.log((1-error_probability)/error_
↪probability), faults_matrix=logicals)
    num_errors = 0
    for i in range(num_shots):
        noise = (np.random.random(H.shape[1]) < error_probability).astype(np.uint8)
        syndrome = H@noise % 2
        predicted_logicals_flipped = matching.decode(syndrome)
        actual_logicals_flipped = logicals@noise % 2
```

(continues on next page)

(continued from previous page)

```
        if not np.array_equal(predicted_logicals_flipped, actual_logicals_flipped):
            num_errors += 1
    return num_errors
```

We can optimise the code a bit further by vectorising over the shots, using the `Matching.decode_batch` method. This method takes in a binary numpy array with dimensions (num_shots, syndrome_length), where syndrome_length should be large enough to include all detector nodes, and be no larger than the number of nodes (including boundary nodes). By vectorising, the iteration over shots is done in C++ rather than Python, which can be significantly faster when the decoding problem itself is easy.

```
[5]: def num_decoding_failures_vectorised(H, logicals, error_probability, num_shots):
         matching = Matching.from_check_matrix(H, weights=np.log((1-error_probability)/error_
     ↪probability), faults_matrix=logicals)
         noise = (np.random.random((num_shots, H.shape[1])) < error_probability).astype(np.
     ↪uint8)
         shots = (noise @ H.T) % 2
         actual_observables = (noise @ logicals.T) % 2
         predicted_observables = matching.decode_batch(shots)
         num_errors = np.sum(np.any(predicted_observables != actual_observables, axis=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:

```
[6]: %%time

     num_shots = 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 error_probability in ps:
             num_errors = num_decoding_failures_vectorised(Hx, logX, error_probability, num_
     ↪shots)
             log_errors.append(num_errors/num_shots)
         log_errors_all_L.append(np.array(log_errors))
```

```
Simulating L=4...
Simulating L=8...
Simulating L=12...
CPU times: user 2.01 s, sys: 11.2 ms, total: 2.03 s
Wall time: 2.02 s
```

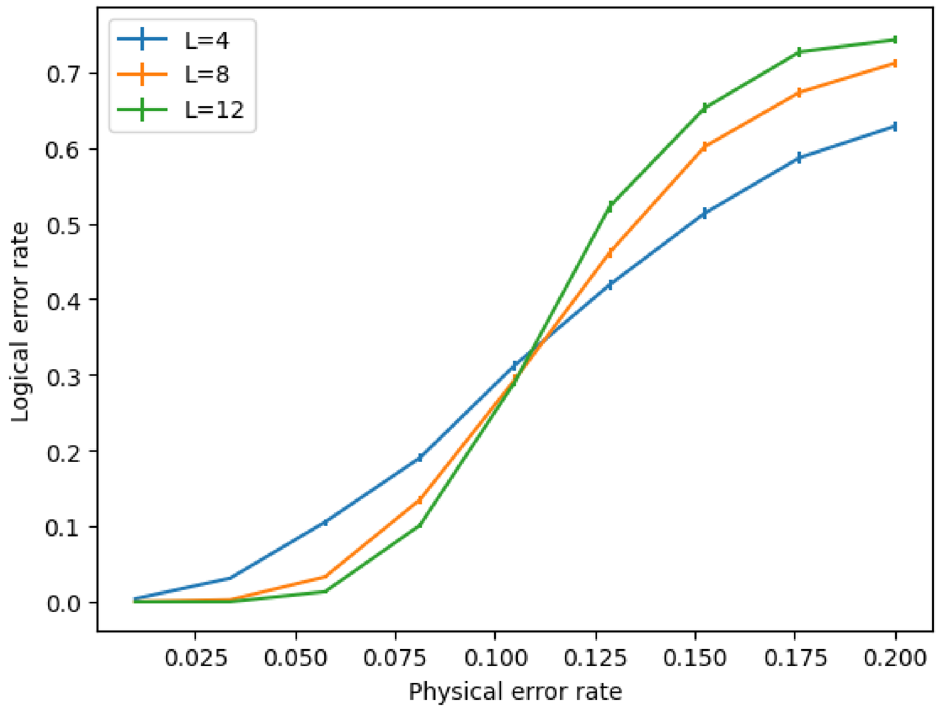
Finally, let's plot the results! We expect to see a threshold of around 10.3%, although a precise estimate requires using more trials, larger lattice sizes and scanning more values of p:

```
[7]: %matplotlib inline
```

(continues on next page)

(continued from previous page)

```
plt.figure()
for L, logical_errors in zip(Ls, log_errors_all_L):
    std_err = (logical_errors*(1-logical_errors)/num_shots)**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);
```



nbsphinx-code-borderwhite

5.1.1 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.

[8]:
```python
def num_decoding_failures_noisy_syndromes(H, logicals, p, q, num_shots, repetitions):
    matching = Matching(H, weights=np.log((1-p)/p),
                repetitions=repetitions, timelike_weights=np.log((1-q)/q), faults_
↪matrix=logicals)
    num_stabilisers, num_qubits = H.shape
    num_errors = 0
    for i in range(num_shots):
        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
        predicted_logicals_flipped = matching.decode(noisy_syndrome)
        actual_logicals_flipped = noise_total@logicals.T % 2
        if not np.array_equal(predicted_logicals_flipped, actual_logicals_flipped):
            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:

[9]:
```python
%%time

num_shots = 3000
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_shots, L)
        log_errors.append(num_errors/num_shots)
    log_errors_all_L.append(np.array(log_errors))
```

```
Simulating L=8...
Simulating L=10...
Simulating L=12...
CPU times: user 27.5 s, sys: 8.27 ms, total: 27.5 s
Wall time: 27.5 s
```

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

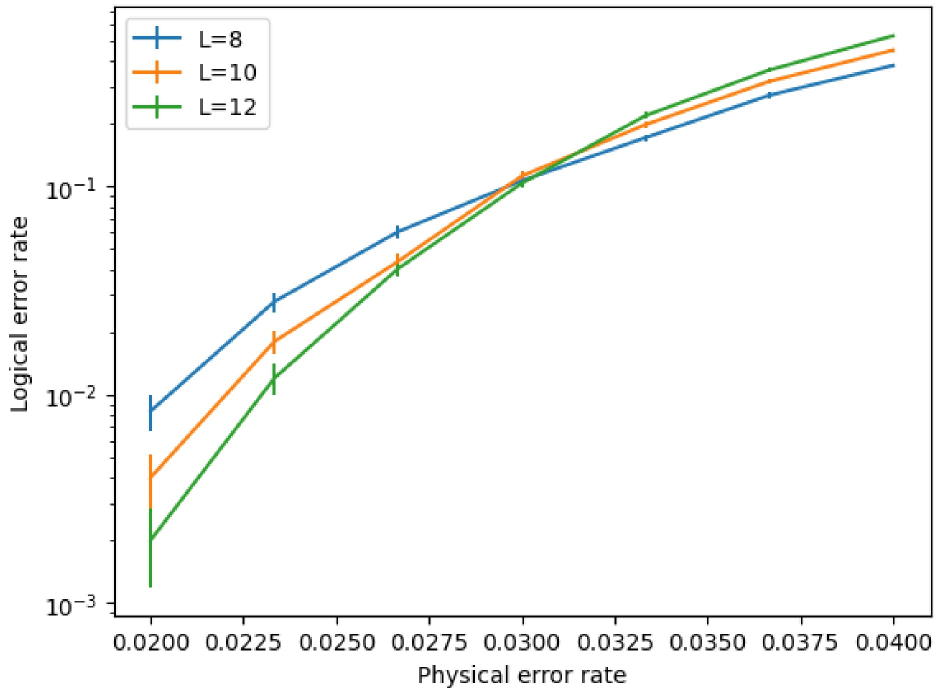
[10]:
```python
%matplotlib inline

plt.figure()
```

(continues on next page)

(continued from previous page)

```
for L, logical_errors in zip(Ls, log_errors_all_L):
    std_err = (logical_errors*(1-logical_errors)/num_shots)**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);
```



nbsphinx-code-borderwhite

5.1.2 Simulating circuit-level noise

PyMatching can be combined with Stim to decode in the presence of more realistic noise models, where errors can occur during any gate in the syndrome extraction circuit. To do this, you construct a Stim circuit for the noisy quantum error correction circuit you want to simulate (e.g. a toric code memory experiment). Stim can sample syndromes (detector measurement outcomes) from the circuit and also provides a DetectorErrorModel (essentially a generalisation of a matching graph) which PyMatching uses to construct the `Matching` object for decoding the syndrome.

Note that the sinter package combines Stim and PyMatching and uses parallelisation over shots to run error correction simulations more efficiently. It also includes other tools (such as for plotting and analysing data). However, here we will use Stim and PyMatching directly to demonstrate how the APIs can be used.

We will use the surface code here (instead of the toric code), since surface code example circuits are already included with Stim. In general you should write your own circuits tailored to the research problem you are trying to solve, however the example circuits are useful for getting started. Here we will sample shots from surface code circuits over a range of lattice sizes and circuit-level error rates:

[11]:
```
%%time

import stim
```

(continues on next page)

(continued from previous page)

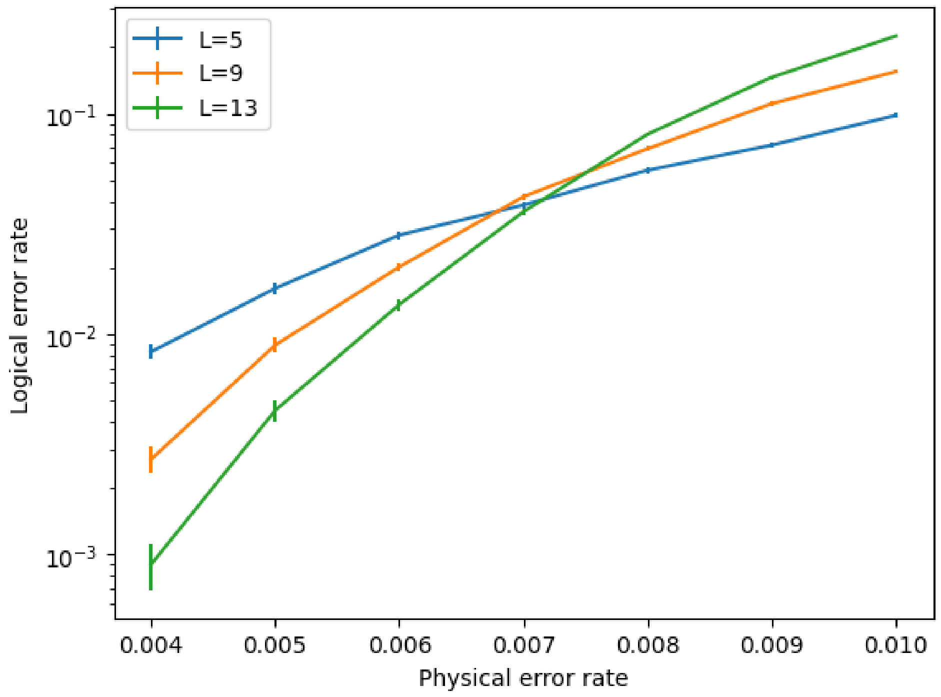
```
num_shots = 20000
Ls = range(5,14,4)
ps = np.linspace(0.004, 0.01, 7)
log_errors_all_L = []
for L in Ls:
    print("Simulating L={}...".format(L))
    log_errors = []
    for p in ps:
        circuit = stim.Circuit.generated("surface_code:rotated_memory_x",
                                        distance=L,
                                        rounds=L,
                                        after_clifford_depolarization=p,
                                        before_round_data_depolarization=p,
                                        after_reset_flip_probability=p,
                                        before_measure_flip_probability=p)
        model = circuit.detector_error_model(decompose_errors=True)
        matching = Matching.from_detector_error_model(model)
        sampler = circuit.compile_detector_sampler()
        syndrome, actual_observables = sampler.sample(shots=num_shots, separate_
↪observables=True)
        predicted_observables = matching.decode_batch(syndrome)
        num_errors = np.sum(np.any(predicted_observables != actual_observables, axis=1))
        log_errors.append(num_errors/num_shots)
    log_errors_all_L.append(np.array(log_errors))
```

```
Simulating L=5...
Simulating L=9...
Simulating L=13...
CPU times: user 32.5 s, sys: 190 ms, total: 32.7 s
Wall time: 32.7 s
```

Now let's plot the results:

[12]:
```
%matplotlib inline

plt.figure()
for L, logical_errors in zip(Ls, log_errors_all_L):
    std_err = (logical_errors*(1-logical_errors)/num_shots)**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);
```



nbsphinx-code-borderwhite

We see a threshold of around 0.7% for circuit-level depolarising noise in the surface code. For more examples of how to use Stim with PyMatching (e.g. to estimate the required size of a surface code circuit to achieve a given error rate), see the Stim documentation, including the getting started notebook.

5.2 Python API Documentation

5.2.1 Matching

class pymatching.matching.**Matching**(*graph: Union[csc_matrix, ndarray, PyGraph, Graph, List[List[int]], stim.DetectorErrorModel, spmatrix] = None, weights: Union[float, ndarray, List[float]] = None, error_probabilities: Union[float, ndarray, List[float]] = None, repetitions: int = None, timelike_weights: Union[float, ndarray, List[float]] = None, measurement_error_probabilities: Union[float, ndarray, List[float]] = None, **kwargs*)

A class for constructing matching graphs and decoding using the minimum-weight perfect matching decoder. The matching graph can be constructed using the *Matching.add_edge* and *Matching.add_boundary_edge* methods. Alternatively, it can be loaded from a parity check matrix (a *scipy.sparse* matrix or *numpy.ndarray* with one or two non-zero elements in each column), a NetworkX or rustworkx graph, or from a *stim.DetectorErrorModel*.

Attributes

boundary
Return the indices of the boundary nodes.

num_detectors
The number of detectors in the matching graph.

num_edges
The number of edges in the matching graph

num_fault_ids
The number of fault IDs defined in the matching graph

num_nodes
The number of nodes in the matching graph

Methods

`add_boundary_edge`(node[, fault_ids, weight, ...])	Add an edge connecting *node* to the boundary
`add_edge`(node1, node2[, fault_ids, weight, ...])	Add an edge to the matching graph
`add_noise`()	Add noise by flipping edges in the matching graph with a probability given by the error_probility edge attribute.
`decode`(z[, _legacy_num_neighbours, ...])	Decode the syndrome *z* using minimum-weight perfect matching
`decode_batch`(shots, *[, return_weights, ...])	Decode from a 2D *shots* array containing a batch of syndrome measurements.
`decode_to_edges_array`(syndrome)	Decode the syndrome *syndrome* using minimum-weight perfect matching, returning the edges in the solution, given as pairs of detector node indices in a numpy array.
`decode_to_matched_dets_array`(syndrome)	Decode the syndrome *syndrome* using minimum-weight perfect matching, returning the pairs of matched detection events (or detection events matched to the boundary) as a 2D numpy array.
`decode_to_matched_dets_dict`(syndrome)	Decode the syndrome *syndrome* using minimum-weight perfect matching, returning a dictionary giving the detection event that each detection event was matched to (or None if it was matched to the boundary).
`draw`()	Draw the matching graph using matplotlib Draws the matching graph as a matplotlib graph.
`edges`()	Edges of the matching graph Returns a list of edges of the matching graph.
`ensure_num_fault_ids`(min_num_fault_ids)	Set the minimum number of fault ids in the matching graph.
`from_check_matrix`(check_matrix[, weights, ...])	Load a matching graph from a check matrix
`from_detector_error_model`(model)	Constructs a *pymatching.Matching* object by loading from a *stim.DetectorErrorModel*.
`from_detector_error_model_file`(dem_path)	Construct a *pymatching.Matching* by loading from a stim DetectorErrorModel file path.
`from_networkx`(graph, *[, min_num_fault_ids])	Returns a new *pymatching.Matching* object from a NetworkX graph
`from_stim_circuit`(circuit)	Constructs a *pymatching.Matching* object by loading from a *stim.Circuit*
`from_stim_circuit_file`(stim_circuit_path)	Construct a *pymatching.Matching* by loading from a stim circuit file path.
`get_boundary_edge_data`(node)	Returns the edge data associated with the boundary edge *(node,)*.
`get_edge_data`(node1, node2)	Returns the edge data associated with the edge *(node1, node2)*.
`has_boundary_edge`(node)	Returns True if the boundary edge *(node,)* is in the graph.
`has_edge`(node1, node2)	Returns True if edge *(node1, node2)* is in the graph.
`load_from_check_matrix`([check_matrix, ...])	Load a matching graph from a check matrix
`load_from_networkx`(graph, *[, min_num_fault_ids])	Load a matching graph from a NetworkX graph into a *pymatching.Matching* object
`load_from_retworkx`(graph, *[, min_num_fault_ids])	Load a matching graph from a retworkX graph.
`load_from_rustworkx`(graph, *[, ...])	Load a matching graph from a rustworkX graph
`set_boundary_nodes`(nodes)	Set boundary nodes in the matching graph.
`to_networkx`()	Convert to NetworkX graph Returns a NetworkX graph corresponding to the matching graph.
`to_retworkx`()	Deprecated, use `pymatching.Matching.to_rustworkx` instead (since the *retworkx* package

`__init__`(*graph: Union[csc_matrix, ndarray, PyGraph, Graph, List[List[int]], stim.DetectorErrorModel, spmatrix] = None*, *weights: Union[float, ndarray, List[float]] = None*, *error_probabilities: Union[float, ndarray, List[float]] = None*, *repetitions: int = None*, *timelike_weights: Union[float, ndarray, List[float]] = None*, *measurement_error_probabilities: Union[float, ndarray, List[float]] = None*, ***kwargs*)

Constructor for the Matching class

Parameters

graph
[*scipy.spmatrix* or *numpy.ndarray* or *networkx.Graph* or *stim.DetectorErrorModel*, optional] The matching graph to be decoded with minimum-weight perfect matching, given either as a binary parity check matrix (scipy sparse matrix or numpy.ndarray), a NetworkX or rustworkx graph, or a Stim DetectorErrorModel. Each edge in the NetworkX or rustworkx 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

weights
[float or numpy.ndarray, optional] If *graph* is given as a scipy or numpy array, *weights* gives the weights of edges in the matching graph corresponding to columns of *graph*. If weights is a numpy.ndarray, it should be a 1D array with length equal to *graph.shape[1]*. If weights is a float, it is used as the weight for all edges corresponding to columns of *graph*. By default None, in which case all weights are set to 1.0 This argument was renamed from *spacelike_weights* in PyMatching v2.0, but *spacelike_weights* is still accepted in place of *weights* for backward compatibility.

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. 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 *check_matrix* is provided as a check matrix, not a NetworkX graph. By default None

timelike_weights
[float, optional] If *check_matrix* 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 *(check_matrix.shape[0],)* is given, the edge weight for each vertical timelike edge associated with the *i`th check (row) of `check_matrix* 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 *check_matrix* 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 *(check_matrix.shape[0],)* is given, the error probability for each vertical timelike edge associated with the *i`th check (row) of `check_matrix* is set to *measurement_error_probabilities[i]*. By default None

****kwargs**
The remaining keyword arguments are passed to *Matching.load_from_check_matrix* if *graph* is a check matrix.

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_boundary_edge(*node: int*, *fault_ids: Optional[Union[int, Set[int]]] = None*, *weight: float = 1.0*, *error_probability: Optional[float] = None*, ***, *merge_strategy: str = 'disallow'*, ***kwargs*) → None

Add an edge connecting *node* to the boundary

Parameters

node: int
The index of the node to be connected to the boundary with a boundary edge

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). By default None

weight: float, optional
The weight of the edge. The weight can be positive or negative, but its absolute value cannot exceed the maximum absolute edge weight of 2**24-1=16,777,215. If the absolute value of the weight exceeds this value, the edge will not be added to the graph and a warning will be raised. 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

merge_strategy: str, optional
Which strategy to use if the edge (*node1*, *node2*) is already in the graph. The available

options are "disallow", "independent", "smallest-weight", "keep-original" and "replace". "disallow" raises a *ValueError* if the edge (*node1*, *node2*) is already present. The "independent" strategy assumes that the existing edge (*node1*, *node2*) and the edge being added represent independent error mechanisms, and they are merged into a new edge with updated weights and error_probabilities accordingly (it is assumed that each weight represents the log-likelihood ratio log((1-p)/p) where p is the *error_probability* and where the natural logarithm is used. The fault_ids associated with the existing edge are kept only, since where the natural logarithm is used. The fault_ids associated with the existing edge are kept only, since the code has distance 2 if parallel edges have different fault_ids anyway). The "smallest-weight" strategy keeps only the new edge if it has a smaller weight than the existing edge, otherwise the graph is left unchanged. The "keep-original" strategy keeps only the existing edge, and ignores the edge being added. The "replace" strategy always keeps the edge being added, replacing the existing edge. By default, "disallow"

Examples

```
>>> import pymatching
>>> m = pymatching.Matching()
>>> m.add_boundary_edge(0)
>>> m.add_edge(0, 1)
>>> print(m.num_edges)
2
>>> print(m.num_nodes)
2
>>> import math
>>> m = pymatching.Matching()
>>> m.add_boundary_edge(0, fault_ids={0}, weight=math.log((1-0.05)/0.05), error_
↪probability=0.05)
>>> m.add_edge(0, 1, fault_ids={1}, weight=math.log((1-0.1)/0.1), error_
↪probability=0.1)
>>> m.add_boundary_edge(1, fault_ids={2}, weight=math.log((1-0.2)/0.2), error_
↪probability=0.2)
>>> m
<pymatching.Matching object with 2 detectors, 0 boundary nodes, and 3 edges>
>>> m = pymatching.Matching()
>>> m.add_boundary_edge(0, fault_ids=0, weight=2)
>>> m.add_boundary_edge(0, fault_ids=1, weight=1, merge_strategy="smallest-
↪weight")
>>> m.add_boundary_edge(0, fault_ids=2, weight=3, merge_strategy="smallest-
↪weight")
>>> m.edges()
[(0, None, {'fault_ids': {1}, 'weight': 1.0, 'error_probability': -1.0})]
>>> m.boundary  # Using Matching.add_boundary_edge, no boundary nodes are added␣
↪(the boundary is a virtual node)
set()
```

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

Add an edge to the matching graph

Parameters

node1: int
The index of node1 in the new edge (node1, node2)

node2: int
The index of node2 in the new edge (node1, node2)

fault_ids: set[int] or int, optional
The indices 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. The weight can be positive or negative, but its absolute value cannot exceed the maximum absolute edge weight of 2**24-1=16,777,215. If the absolute value of the weight exceeds this value, the edge will not be added to the graph and a warning will be raised. 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

merge_strategy: str, optional
Which strategy to use if the edge (*node1*, *node2*) is already in the graph. The available options are “disallow”, “independent”, “smallest-weight”, “keep-original” and “replace”. “disallow” raises a *ValueError* if the edge (*node1*, *node2*) is already present. The “independent” strategy assumes that the existing edge (*node1*, *node2*) and the edge being added represent independent error mechanisms, and they are merged into a new edge with updated weights and error_probabilities accordingly (it is assumed that each weight represents the log-likelihood ratio log((1-p)/p) where p is the *error_probability* and where the natural logarithm is used. The fault_ids associated with the existing edge are kept only, since where the natural logarithm is used. The fault_ids associated with the existing edge are kept only, since the code has distance 2 if parallel edges have different fault_ids anyway). The “smallest-weight” strategy keeps only the new edge if it has a smaller weight than the existing edge, otherwise the graph is left unchanged. The “keep-original” strategy keeps only the existing edge, and ignores the edge being added. The “replace” strategy always keeps the edge being added, replacing the existing edge. By default, “disallow”

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 math
>>> m = pymatching.Matching()
>>> m.add_edge(0, 1, fault_ids=2, weight=math.log((1-0.05)/0.05), error_
↪probability=0.05)
```

(continues on next page)

(continued from previous page)

```
>>> 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>
>>> m = pymatching.Matching()
>>> m.add_edge(0, 1, fault_ids=0, weight=2)
>>> m.add_edge(0, 1, fault_ids=1, weight=1, merge_strategy="smallest-weight")
>>> m.add_edge(0, 1, fault_ids=2, weight=3, merge_strategy="smallest-weight")
>>> m.edges()
[(0, 1, {'fault_ids': {1}, 'weight': 1.0, 'error_probability': -1.0})]
```

`add_noise`() → Optional[Tuple[ndarray, 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

set of int
The indices of the boundary nodes

`decode`(*z: Union[ndarray, List[int]]*, *_legacy_num_neighbours: Optional[int] = None*, *_legacy_return_weight: Optional[bool] = None*, ***, *return_weight: bool = False*, ***kwargs*) → Union[ndarray, Tuple[ndarray, int]]

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

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*).

_legacy_num_neighbours: int
The *num_neighbours* argument available in PyMatching versions 0.x.x is not available in PyMatching v2.0.0 or later, since it introduced an approximation that is not relevant or required in the new version 2 implementation. Providing num_neighbours as this second positional argument will raise an exception in a future version of PyMatching.

_legacy_return_weight: bool
`return_weight` used to be available as this third positional argument, but should now be set as a keyword argument. In a future version of PyMatching, it will only be possible to provide *return_weight* as a keyword argument.

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.

Raises

ValueError
If there is no error consistent with the provided syndrome. Occurs if the syndrome has odd parity in the support of a connected component without a boundary.

Examples

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

decode_batch(*shots: ndarray*, ***, *return_weights: bool = False*, *bit_packed_shots: bool = False*, *bit_packed_predictions: bool = False*) → Union[ndarray, Tuple[ndarray, ndarray]]

Decode from a 2D *shots* array containing a batch of syndrome measurements. A faster alternative to using *pymatching.Matching.decode* and iterating over the shots in Python.

Parameters

shots
[np.ndarray] A 2D numpy array of shots to decode, of *dtype=np.uint8*.

If *bit_packed_shots==False*, then *shots* should have shape *shots.shape=(num_shots, syndrome_length)*, where *num_shots* is the number of shots (samples), and *syndrome_length* is the length of the binary syndrome vector to be decoded for each shot. If *len(self.boundary)==0* (e.g. if there is no boundary, or only a virtual boundary node, the default when loading from stim) then *syndrome_length=self.num_detectors*. However, *syndrome_length* is permitted to be as high as *self.num_nodes* in case the graph contains detectors nodes with an index larger than *self.num_detectors-1* (when *len(self.boundary)>0*).

If *bit_packed_shots==True* then *shots* should have shape *shots.shape=(num_shots, math.ceil(syndrome_length / 8))*. Bit packing should be done using little endian order on the last axis (like `np.packbits(data, bitorder='little', axis=1)`), so that the bit for detection event *m* in shot *s* can be found at `(dets[s, m // 8] >> (m % 8)) & 1`.

return_weights
[bool] If True, then also return a numpy array containing the weights of the solutions for

all the shots. By default, False.

bit_packed_shots
[bool] Set to *True* to provide *shots* as a bit-packed array, such that the bit for detection event *m* in shot *s* can be found at `(dets[s, m // 8] >> (m % 8)) & 1`.

bit_packed_predictions
[bool] Set to *True* if the returned predictions should be bit-packed, with the bit for fault id *m* in shot *s* in `(obs[s, m // 8] >> (m % 8)) & 1`

Returns

predictions: np.ndarray
The batch of predictions output by the decoder, a binary numpy array of *dtype=np.uint8* and with shape *predictions.shape=(num_shots, self.num_fault_ids)*. *predictions[i, j]=1* iff the decoder predicts that fault id *j* was flipped in the shot *i*.

weights: np.ndarray
The weights of the MWPM solutions, a numpy array of *dtype=float*. *weights[i]* is the weight of the MWPM solution in shot *i*.

Examples

```
>>> import pymatching
>>> import stim
>>> circuit = stim.Circuit.generated("surface_code:rotated_memory_x",
...                                  distance=5,
...                                  rounds=5,
...                                  after_clifford_depolarization=0.005)
>>> model = circuit.detector_error_model(decompose_errors=True)
>>> matching = pymatching.Matching.from_detector_error_model(model)
>>> sampler = circuit.compile_detector_sampler()
>>> syndrome, actual_observables = sampler.sample(shots=10000, separate_
↪observables=True)
>>> syndrome.shape
(10000, 120)
>>> actual_observables.shape
(10000, 1)
>>> predicted_observables = matching.decode_batch(syndrome)
>>> predicted_observables.shape
(10000, 1)
>>> num_errors = np.sum(np.any(predicted_observables != actual_observables,␣
↪axis=1))
```

We can also decode bit-packed shots, and return bit-packed predictions: >>> import pymatching >>> import stim >>> circuit = stim.Circuit.generated(“surface_code:rotated_memory_x”, … distance=5, … rounds=5, … after_clifford_depolarization=0.005) >>> model = circuit.detector_error_model(decompose_errors=True) >>> matching = pymatching.Matching.from_detector_error_model(model) >>> sampler = circuit.compile_detector_sampler() >>> syndrome, actual_observables = sampler.sample(shots=10000, separate_observables=True, bit_packed=True) >>> syndrome.shape (10000, 15) >>> actual_observables.shape (10000, 1) >>> predicted_observables = matching.decode_batch(syndrome, bit_packed_shots=True, bit_packed_predictions=True) >>> predicted_observables.shape (10000, 1) >>> num_errors = np.sum(np.any(predicted_observables != actual_observables, axis=1)) `decode_to_edges_array`(*syndrome: Union[ndarray, List[int]]*) → ndarray

Decode the syndrome *syndrome* using minimum-weight perfect matching, returning the edges in the solution, given as pairs of detector node indices in a numpy array.

Parameters

syndrome

[numpy.ndarray] A binary syndrome vector to decode. The number of elements in *syndrome* should equal the number of nodes in the matching graph. If *syndrome* is a 1D array, then *syndrome[i]* is the syndrome at node *i* of the matching graph. If *syndrome* is 2D then *syndrome[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*).

Returns

numpy.ndarray

A 2D array *edges* giving the edges in the matching solution as pairs of detector nodes (or as a detector node and the boundary, for a boundary edge). If there are *num_predicted_edges* edges then the shape of *edges* is *edges.shape=(num_predicted_edges, 2)*, and edge *i* is between detector node *edges[i, 0]* and detector node *edges[i, 1]*. For a boundary edge *i* between a detector node *k* and the boundary (either a boundary node or the virtual boundary node), then *pairs[i,0]* is *k*, and *pairs[i,1]=-1* denotes the boundary (the boundary is always denoted by -1 and is always in the second column).

Examples

```
>>> import pymatching
>>> m = pymatching.Matching()
>>> m.add_boundary_edge(0)
>>> m.add_edge(0, 1)
>>> m.add_edge(1, 2)
>>> m.add_edge(2, 3)
>>> m.add_edge(3, 4)
>>> m.add_edge(4, 5)
>>> m.add_edge(5, 6)
>>> edges = m.decode_to_edges_array([0, 1, 0, 0, 1, 0, 1])
>>> print(edges)
[[ 0  1]
 [ 0 -1]
 [ 5  4]
 [ 5  6]]
```

`decode_to_matched_dets_array`(*syndrome: Union[ndarray, List[int]]*) → ndarray

Decode the syndrome *syndrome* using minimum-weight perfect matching, returning the pairs of matched detection events (or detection events matched to the boundary) as a 2D numpy array. Each pair of matched detection events returned by this method corresponds to a shortest path between the detection events in the solution to the problem: if you instead want the set of all edges in the solution (pairs of detector nodes), use *Matching.decode_to_edges* instead. Note that, unlike *Matching.decode*, *Matching.decode_batch* and *Matching.decode_to_edges_array*, this method currently only supports non-negative edge weights.

Parameters

syndrome

[numpy.ndarray] A binary syndrome vector to decode. The number of elements in *syndrome* should equal the number of nodes in the matching graph. If *syndrome* is a 1D array, then *syndrome[i]* is the syndrome at node *i* of the matching graph. If *syndrome* is 2D then *syndrome[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*).

Returns

numpy.ndarray
An 2D array *pairs* giving the endpoints of the paths between detection events in the solution of the matching. If there are *num_paths* paths then the shape of *pairs* is *pairs.shape=(num_paths, 2)*, and path *i* starts at detection event *pairs[i,0]* and ends at detection event *pairs[i,1]*. For a path *i* connecting a detection event to the boundary (either a boundary node or the virtual boundary node), then *pairs[i,0]* is the index of the detection event, and *pairs[i,1]=-1* denotes the boundary.

Examples

```
>>> import pymatching
>>> m = pymatching.Matching()
>>> m.add_boundary_edge(0)
>>> m.add_edge(0, 1)
>>> m.add_edge(1, 2)
>>> m.add_edge(2, 3)
>>> m.add_edge(3, 4)
>>> m.add_edge(4, 5)
>>> m.add_edge(5, 6)
>>> matched_dets = m.decode_to_matched_dets_array([0, 1, 0, 0, 1, 0, 1])
>>> print(matched_dets)
[[ 1 -1]
 [ 4  6]]
```

decode_to_matched_dets_dict(*syndrome: Union[ndarray, List[int]]*) → Union[ndarray, Tuple[ndarray, int]]

Decode the syndrome *syndrome* using minimum-weight perfect matching, returning a dictionary giving the detection event that each detection event was matched to (or None if it was matched to the boundary). Note that (unlike *Matching.decode*), this method currently only supports non-negative edge weights.

Parameters

syndrome
[numpy.ndarray] A binary syndrome vector to decode. The number of elements in *syndrome* should equal the number of nodes in the matching graph. If *syndrome* is a 1D array, then *syndrome[i]* is the syndrome at node *i* of the matching graph. If *syndrome* is 2D then *syndrome[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*).

Returns

dict
A dictionary *mate* giving the detection event that each detection event is matched to (or *None* if it is matched to the boundary). If detection event *i* is matched to detection event *j*, then *mate[i]=j*. If detection event *i* is matched to the boundary (either a boundary node or the virtual boundary node), then *mate[i]=None*.

Examples

```
>>> import pymatching
>>> m = pymatching.Matching()
>>> m.add_boundary_edge(0)
>>> m.add_edge(0, 1)
>>> m.add_edge(1, 2)
>>> m.add_edge(2, 3)
>>> m.add_edge(3, 4)
>>> d = m.decode_to_matched_dets_dict([1, 0, 0, 1, 1])
>>> d[3]
4
>>> d
{0: None, 3: 4, 4: 3}
```

`draw()` → None

Draw the matching graph using matplotlib Draws the matching graph as a matplotlib graph. Detector 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). Each node is labelled with its id/index, and each edge is labelled with its *fault_ids*. Note that you may need to call *plt.figure()* before and *plt.show()* after calling this function.

`edges()` → List[Tuple[int, Optional[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

List of (int, int, dict) tuples
A list of edges of the matching graph

`ensure_num_fault_ids`(*min_num_fault_ids: int*) → None

Set the minimum number of fault ids in the matching graph.

Let *max_id* be the maximum fault id assigned to any of the edges in a *pymatching.Matching* graph *m*. Then setting *m.ensure_num_fault_ids(n)* will ensure that *Matching.num_fault_ids=max(n, max_id)*. Note that *Matching.num_fault_ids* sets the length of the correction array output by *Matching.decode*.

Parameters

min_num_fault_ids: int
The required minimum number of fault ids in the matching graph

`static from_check_matrix`(*check_matrix: Union[csc_matrix, spmatrix, ndarray, List[List[int]]], weights: Optional[Union[float, ndarray, List[float]]] = None, error_probabilities: Optional[Union[float, ndarray, List[float]]] = None, repetitions: Optional[int] = None, timelike_weights: Optional[Union[float, ndarray, List[float]]] = None, measurement_error_probabilities: Optional[Union[float, ndarray, List[float]]] = None, *, faults_matrix: Optional[Union[csc_matrix, spmatrix, ndarray, List[List[int]]]] = None, merge_strategy: str = 'smallest-weight', use_virtual_boundary_node: bool = False, **kwargs*) → *Matching*

Load a matching graph from a check matrix

Parameters

check_matrix
[*scipy.csc_matrix* 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)

weights
[float or numpy.ndarray, optional] If *check_matrix* is given as a scipy or numpy array, *weights* gives the weights of edges in the matching graph corresponding to columns of *check_matrix*. If *weights* is a numpy.ndarray, it should be a 1D array with length equal to *check_matrix.shape[1]*. If weights is a float, it is used as the weight for all edges corresponding to columns of *check_matrix*. By default None, in which case all weights are set to 1.0 This argument was renamed from *spacelike_weights* in PyMatching v2.0, but *spacelike_weights* is still accepted in place of *weights* for backward compatibility.

error_probabilities
[float or numpy.ndarray, optional] The probabilities with which an error occurs on each edge associated with a column of check_matrix. 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 check_matrix. 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 *(check_matrix.shape[0],)* is given, the edge weight for each vertical timelike edge associated with the *i`th check (row) of `check_matrix* 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 *(check_matrix.shape[0],)* is given, the error probability for each vertical timelike edge associated with the *i`th check (row) of `check_matrix* 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

faults_matrix: `scipy.csc_matrix` or `numpy.ndarray` or List[List[int]], optional
A binary array of faults, which can be used to set the *fault_ids* for each edge in the constructed matching graph. The *fault_ids* attribute of the edge corresponding to column *j* of *check_matrix* includes fault id *i* if and only if *faults[i,j]==1*. Therefore, the number of columns in *faults* must match the number of columns in *check_matrix*. By default, *faults* is just set to the identity matrix, in which case the edge corresponding to column *j* of *check_matrix* has *fault_ids={j}*. As an example, if *check_matrix* corresponds to the X check matrix of a CSS stabiliser code, then you could set *faults* to the X logical operators: in this case the output of *Matching.decode* will be a binary array *correction* where *correction[i]==1* if the decoder predicts that the logical operator corresponding to row *i* of *faults* was flipped, given the observed syndrome.

merge_strategy: str, optional
Which strategy to use when adding an edge (*node1*, *node2*) that is already in the graph. The available options are “disallow”, “independent”, “smallest-weight”, “keep-original” and

"replace". "disallow" raises a *ValueError* if the edge (*node1*, *node2*) is already present. The "independent" strategy assumes that the existing edge (*node1*, *node2*) and the edge being added represent independent error mechanisms, and they are merged into a new edge with updated weights and error_probabilities accordingly (it is assumed that each weight represents the log-likelihood ratio log((1-p)/p) where p is the *error_probability* and where the natural logarithm is used. The fault_ids associated with the existing edge are kept only, since the code has distance 2 if parallel edges have different fault_ids anyway). The "smallest-weight" strategy keeps only the new edge if it has a smaller weight than the existing edge, otherwise the graph is left unchanged. The "keep-original" strategy keeps only the existing edge, and ignores the edge being added. The "replace" strategy always keeps the edge being added, replacing the existing edge. By default, "smallest-weight"

use_virtual_boundary_node: bool, optional
This option determines how columns are handled if they contain only a single 1 (representing a boundary edge). Consider a column contains a single 1 at row index i. If *use_virtual_boundary_node=False*, then this column will be handled by adding an edge *(i, check_matrix.shape[0])*, and marking the node *check_matrix.shape[0]* as a boundary node with *Matching.set_boundary(check_matrix.shape[0])*. The resulting graph will contain *check_matrix.shape[0]+1* nodes, the largest of which is the boundary node. If *use_virtual_boundary_node=True* then instead the boundary is a virtual node, and this column is handled with *Matching.add_boundary_edge(i, ...)*. The resulting graph will contain *check_matrix.shape[0]* nodes, and there is no boundary node. Both options are handled identically by the decoder, although *use_virtual_boundary_node=True* is recommended since it is simpler (with a one-to-one correspondence between nodes and rows of check_matrix), and is also slightly more efficient. By default, False (for backward compatibility)

Examples

```
>>> import pymatching
>>> m = pymatching.Matching.from_check_matrix([[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
>>> check_matrix = csc_matrix([[1, 1, 0], [0, 1, 1]])
>>> m = pymatching.Matching.from_check_matrix(check_matrix)
>>> m
<pymatching.Matching object with 2 detectors, 1 boundary node, and 3 edges>
```

static from_detector_error_model(*model: stim.DetectorErrorModel*) → pymatching.Matching

Constructs a *pymatching.Matching* object by loading from a *stim.DetectorErrorModel*.

A *stim.DetectorErrorModel* (DEM) describes a circuit-level noise model in a quantum error correction protocol, and is defined in the Stim documentation: https://github.com/quantumlib/Stim/blob/main/doc/file_format_dem_detector_error_model.md. When loading from a DEM, there is a one-to-one correspondence with a detector in the DEM and a node in the *pymatching.Matching* graph, and each graphlike error in the DEM becomes an edge (or merged into a parallel edge) in the *pymatching.Matching* graph. A error instruction in the DEM is graphlike if it causes either one or two detection events, and can be either its own DEM instruction, or within a suggested decomposition of a larger DEM instruction. Error instruction in the DEM that cause more than two detection events and do not have a suggested decomposition into edges are ignored. There set of *fault_ids* assigned to a *pymatching.Matching* graph edge is the set of *logical_observable* indices associated with the corresponding graphlike fault mechanism in the DEM. Parallel edges are merged, with weights chosen on the assumption that the error mechanisms associated with the parallel edges are independent. the *logical_observable* indices associated with the first added parallel edge are kept for the merged edge. If you are loading a *pymatching.Matching* graph from a DEM, you may be interested in using the sinter Python package for monte carlo sampling: https://pypi.org/project/sinter/.

Parameters

model
[stim.DetectorErrorModel] A stim DetectorErrorModel, with all error mechanisms either graphlike, or decomposed into graphlike error mechanisms

Returns

pymatching.Matching
A *pymatching.Matching* object representing the graphlike error mechanisms in *model*

Examples

```
>>> import stim
>>> import pymatching
>>> circuit = stim.Circuit.generated("surface_code:rotated_memory_x",
...                                  distance=5,
...                                  rounds=5,
...                                  after_clifford_depolarization=0.005)
>>> model = circuit.detector_error_model(decompose_errors=True)
>>> matching = pymatching.Matching.from_detector_error_model(model)
>>> matching
<pymatching.Matching object with 120 detectors, 0 boundary nodes, and 502 edges>
```

static `from_detector_error_model_file`(*dem_path: str*) → *Matching*

Construct a *pymatching.Matching* by loading from a stim DetectorErrorModel file path.

Parameters

dem_path
[str] The path of the detector error model file

Returns

pymatching.Matching
A *pymatching.Matching* object representing the graphlike error mechanisms in the stim DetectorErrorModel in the file *dem_path*

static `from_networkx`(*graph: Graph*, *, *min_num_fault_ids: Optional[int] = None*) → *Matching*

Returns a new *pymatching.Matching* object 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 determines how the solution is output via *pymatching.Matching.decode*: the binary *correction* array has length *pymatching.Matching.num_fault_ids*, and *correction[i]* is 1 if and only if an odd number of edges in the MWPM solution have *i* in their *fault_ids* attribute. 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.

min_num_fault_ids: int
Sets the minimum number of fault ids in the matching graph. Let *max_id* be the maximum fault id assigned to any of the edges in the graph. Then setting this argument will ensure that *Matching.num_fault_ids=max(min_num_fault_ids, max_id)*. Note that *Matching.num_fault_ids* sets the length of the correction array output by *Matching.decode*.

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.from_networkx(g)
>>> m
<pymatching.Matching object with 1 detector, 2 boundary nodes, and 2 edges>
```

`static from_stim_circuit`(*circuit: stim.Circuit*) → pymatching.Matching

Constructs a *pymatching.Matching* object by loading from a *stim.Circuit*

Parameters

circuit
[stim.Circuit] A stim circuit containing error mechanisms that are all either graphlike, or decomposable into graphlike error mechanisms

Returns

pymatching.Matching
A *pymatching.Matching* object representing the graphlike error mechanisms in *circuit*, with any hyperedge error mechanisms decomposed into graphlike error mechanisms. Parallel edges are merged using *merge_strategy="independent"*.

Examples

```
>>> import stim
>>> import pymatching
>>> circuit = stim.Circuit.generated("surface_code:rotated_memory_x",
...                                  distance=5,
...                                  rounds=5,
...                                  after_clifford_depolarization=0.005)
>>> matching = pymatching.Matching.from_stim_circuit(circuit)
>>> matching
<pymatching.Matching object with 120 detectors, 0 boundary nodes, and 502 edges>
```

static from_stim_circuit_file(*stim_circuit_path: str*) → *Matching*

Construct a *pymatching.Matching* by loading from a stim circuit file path.

Parameters

stim_circuit_path
[str] The path of the stim circuit file

Returns

pymatching.Matching
A *pymatching.Matching* object representing the graphlike error mechanisms in the stim circuit in the file *stim_circuit_path*, with any hyperedge error mechanisms decomposed into graphlike error mechanisms. Parallel edges are merged using *merge_strategy="independent"*.

get_boundary_edge_data(*node: int*) → Dict[str, Union[Set[int], float]]

Returns the edge data associated with the boundary edge *(node,)*.

Parameters

node: int
The index of the node

Returns

dict
A dictionary with keys *fault_ids*, *weight* and *error_probability*, and values giving the respective boundary edge attributes

get_edge_data(*node1: int*, *node2: int*) → Dict[str, Union[Set[int], float]]

Returns the edge data associated with the edge *(node1, node2)*.

Parameters

node1: int
The index of the first node

node2: int
The index of the second node

Returns

dict
A dictionary with keys *fault_ids*, *weight* and *error_probability*, and values giving the respective edge attributes

`has_boundary_edge`(*node: int*) → bool

Returns True if the boundary edge *(node,)* is in the graph. Note: this method does not check if *node* is connected to a boundary node in *Matching.boundary*; it only checks if *node* is connected to the virtual boundary node (i.e. whether there is a boundary edge *(node,)* present).

Parameters

node: int
The index of the node

Returns

bool
True if the boundary edge *(node,)* is present, otherwise False.

`has_edge`(*node1: int*, *node2: int*) → bool

Returns True if edge *(node1, node2)* is in the graph.

Parameters

node1: int
The index of the first node

node2: int
The index of the second node

Returns

bool
True if the edge *(node1, node2)* is in the graph, otherwise False.

`load_from_check_matrix`(*check_matrix: Optional[Union[csc_matrix, spmatrix, ndarray, List[List[int]]]] = None*, *weights: Optional[Union[float, ndarray, List[float]]] = None*, *error_probabilities: Optional[Union[float, ndarray, List[float]]] = None*, *repetitions: Optional[int] = None*, *timelike_weights: Optional[Union[float, ndarray, List[float]]] = None*, *measurement_error_probabilities: Optional[Union[float, ndarray, List[float]]] = None*, ***, *faults_matrix: Optional[Union[csc_matrix, spmatrix, ndarray, List[List[int]]]] = None*, *merge_strategy: str = 'smallest-weight'*, *use_virtual_boundary_node: bool = False*, ***kwargs*) → None

Load a matching graph from a check matrix

Parameters

check_matrix
[*scipy.csc_matrix* 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)

weights
[float or numpy.ndarray, optional] If *check_matrix* is given as a scipy or numpy array, *weights* gives the weights of edges in the matching graph corresponding to columns of *check_matrix*. If *weights* is a numpy.ndarray, it should be a 1D array with length equal to *check_matrix.shape[1]*. If weights is a float, it is used as the weight for all edges corresponding to columns of *check_matrix*. By default None, in which case all weights are set to 1.0 This argument was renamed from *spacelike_weights* in PyMatching v2.0, but *spacelike_weights* is still accepted in place of *weights* for backward compatibility.

error_probabilities
[float or numpy.ndarray, optional] The probabilities with which an error occurs on each edge associated with a column of check_matrix. 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 check_matrix. 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 *(check_matrix.shape[0],)* is given, the edge weight for each vertical timelike edge associated with the *i`th check (row) of `check_matrix* 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 *(check_matrix.shape[0],)* is given, the error probability for each vertical timelike edge associated with the *i`th check (row) of `check_matrix* 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

faults_matrix: `scipy.csc_matrix` or `numpy.ndarray` or List[List[int]], optional
A binary array of faults, which can be used to set the *fault_ids* for each edge in the constructed matching graph. The *fault_ids* attribute of the edge corresponding to column *j* of *check_matrix* includes fault id *i* if and only if *faults[i,j]==1*. Therefore, the number of columns in *faults* must match the number of columns in *check_matrix*. By default, *faults* is just set to the identity matrix, in which case the edge corresponding to column *j* of *check_matrix* has *fault_ids={j}*. As an example, if *check_matrix* corresponds to the X check matrix of a CSS stabiliser code, then you could set *faults* to the X logical operators: in this case the output of *Matching.decode* will be a binary array *correction* where *correction[i]==1* if the decoder predicts that the logical operator corresponding to row *i* of *faults* was flipped, given the observed syndrome.

merge_strategy: str, optional
Which strategy to use when adding an edge (*node1*, *node2*) that is already in the graph. The available options are "disallow", "independent", "smallest-weight", "keep-original" and "replace". "disallow" raises a *ValueError* if the edge (*node1*, *node2*) is already present. The "independent" strategy assumes that the existing edge (*node1*, *node2*) and the edge being added represent independent error mechanisms, and they are merged into a new edge with updated weights and error_probabilities accordingly (it is assumed that each weight represents the log-likelihood ratio log((1-p)/p) where p is the *error_probability* and where the natural logarithm is used. The fault_ids associated with the existing edge are kept only, since the code has distance 2 if parallel edges have different fault_ids anyway). The "smallest-weight" strategy keeps only the new edge if it has a smaller weight than the existing edge, otherwise the graph is left unchanged. The "keep-original" strategy keeps only the existing edge, and ignores the edge being added. The "replace" strategy always keeps the edge being added, replacing the existing edge. By default, "smallest-weight"

use_virtual_boundary_node: bool, optional
This option determines how columns are handled if they contain only a single 1 (representing a boundary edge). Consider a column contains a single 1 at row index i. If *use_virtual_boundary_node=False*, then this column will be handled by adding an edge *(i, check_matrix.shape[0])*, and marking the node *check_matrix.shape[0]* as a boundary node with *Matching.set_boundary(check_matrix.shape[0])*. The resulting graph will

contain *check_matrix.shape[0]+1* nodes, the largest of which is the boundary node. If *use_virtual_boundary_node=True* then instead the boundary is a virtual node, and this column is handled with *Matching.add_boundary_edge(i, …)*. The resulting graph will contain *check_matrix.shape[0]* nodes, and there is no boundary node. Both options are handled identically by the decoder, although *use_virtual_boundary_node=True* is recommended since it is simpler (with a one-to-one correspondence between nodes and rows of check_matrix), and is also slightly more efficient. By default, False (for backward compatibility)

Examples

```
>>> import pymatching
>>> m = pymatching.Matching()
>>> m.load_from_check_matrix([[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 >>> check_matrix = csc_matrix([[1, 1, 0], [0, 1, 1]]) >>> m = pymatching.Matching() >>> m.load_from_check_matrix(check_matrix) >>> m <pymatching.Matching object with 2 detectors, 1 boundary node, and 3 edges>

load_from_networkx(*graph: Graph*, ***, *min_num_fault_ids: Optional[int] = None*) → None

Load a matching graph from a NetworkX graph into a *pymatching.Matching* object

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 determines how the solution is output via *pymatching.Matching.decode*: the binary *correction* array has length *pymatching.Matching.num_fault_ids*, and *correction[i]* is 1 if and only if an odd number of edges in the MWPM solution have *i* in their *fault_ids* attribute. 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.

min_num_fault_ids: int

Sets the minimum number of fault ids in the matching graph. Let *max_id* be the maximum fault id assigned to any of the edges in the graph. Then setting this argument will ensure that *Matching.num_fault_ids=max(min_num_fault_ids, max_id)*. Note that *Matching.num_fault_ids* sets the length of the correction array output by *Matching.decode*.

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: PyGraph*, *, *min_num_fault_ids: Optional[int] = None*) → None

Load a matching graph from a retworkX graph. This method is deprecated since the retworkx package has been renamed to rustworkx. Please use `pymatching.Matching.load_from_rustworkx` instead.

load_from_rustworkx(*graph: PyGraph*, *, *min_num_fault_ids: Optional[int] = None*) → None

Load a matching graph from a rustworkX graph

Parameters

graph
[rustworkx.PyGraph] Each edge in the rustworkx 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.

min_num_fault_ids: int
Sets the minimum number of fault ids in the matching graph. Let *max_id* be the maximum fault id assigned to any of the edges in the graph. Then setting this argument will ensure that *Matching.num_fault_ids=max(min_num_fault_ids, max_id)*. Note that *Matching.num_fault_ids* sets the length of the correction array output by *Matching.decode*.

Examples

```
>>> import pymatching
>>> import rustworkx 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))
```

(continues on next page)

(continued from previous page)

```
>>> 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

int
The number of detectors

property num_edges: int

The number of edges in the matching graph

Returns

int
The number of edges

property num_fault_ids: int

The number of fault IDs defined in the matching graph

Returns

int
Number of fault IDs

property num_nodes: int

The number of nodes in the matching graph

Returns

int
The number of nodes

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``()` → 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
NetworkX Graph corresponding to the matching graph

`to_retworkx``()` → PyGraph

Deprecated, use `pymatching.Matching.to_rustworkx` instead (since the *retworkx* package has been renamed to *rustworkx*). This method just calls `pymatching.Matching.to_rustworkx` and returns a `rustworkx.PyGraph`, which is now just the preferred name for

`retworkx.PyGraph`. Note that in the future, only the *rustworkx* package name will be supported, see: https://pypi.org/project/retworkx/.

`to_rustworkx``()` → PyGraph

Convert to rustworkx graph Returns a rustworkx 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

rustworkx.PyGraph
rustworkx graph corresponding to the matching graph

5.2.2 Command line interface

`pymatching.`**`cli`**`()`

main(*, command_line_args: List[str]) -> int

Runs the command line tool version of pymatching with the given arguments.

5.2.3 Random number generator

pymatching.**set_seed**(*seed: int*) → None

Sets the seed of the random number generator

Parameters

seed: int
The seed for the random number generator (must be non-negative)

Examples

```
>>> import pymatching
>>> pymatching.set_seed(10)
```

pymatching.**randomize**() → None

Choose a random seed using std::random_device

Examples

```
>>> import pymatching
>>> pymatching.randomize()
```

pymatching.**rand_float**(*from: float*, *to: float*) → float

Generate a floating point number chosen uniformly at random over the interval between *from* and *to*

Parameters

from: float
Smallest float that can be drawn from the distribution

to: float
Largest float that can be drawn from the distribution

Returns

float
The random float

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX