Qsurface

Mark Shui Hu

Dec 01, 2020

MAIN USAGE

1 Installation **3**
- 1.1 Requirements . . . 3

2 Usage **5**
- 2.1 Plotting . . . 5
- 2.2 Command line interface . . . 6

3 Modules **7**
- 3.1 Running simulations . . . 7
- 3.2 Running a threshold simulation . . . 11
- 3.3 Code elements . . . 14
- 3.4 Template code . . . 16
- 3.5 Code types . . . 26
- 3.6 Template error . . . 43
- 3.7 Error types . . . 46
- 3.8 Template decoder . . . 47
- 3.9 Decoders . . . 50
- 3.10 Plotting template . . . 71

4 License **79**

5 Indices and tables **81**

Bibliography **83**

Python Module Index **85**

Index **87**

Qsurface is a simulation package for the surface code, and is designed to modularize 3 aspects of a surface code simulation.

1. The surface code
2. The error model
3. The used decoder

New types of surface codes, error modules and decoders can be added to Qsurface by using the included templates for each of the three core module categories.

The current included decoders are:

- The *Mininum-Weight Perfect Matching* (`mwpm`) decoder.
- Delfosse's and Nickerson's *Union-Find* (`unionfind`) decoder, which has *almost-linear* worst-case time complexity.
- Our modification to the Union-Find decoder; the *Union-Find Node-Suspension* (`ufns`) decoder, which improves the threshold of the Union-Find decoder to near MWPM performance, while retaining quasi-linear worst-case time complexity.

The compatibility of these decoders with the included surface codes are listed below.

Decoders	`toric` code	`planar` code
`mwpm`		
`unionfind`		
`ufns`		

CHAPTER

ONE

INSTALLATION

All required packages can be installed through:

```
pip install qsurface
```

1.1 Requirements

- Python 3.7+
- Tkinter or PyQt5 for interactive plotting.
- Matplotlib 3.4+ for plotting on a 3D lattice (Refers to a future release of matplotlib, see pull request)

1.1.1 MWPM decoder

The MWPM decoder utilizes `networkx` for finding the minimal weights in a fully connected graph. This implementation is however rather slow compared to Kolmogorov's Blossom V algorithm. Blossom V has its own license and is thus not included with Qsurface. We do provided a single function to download and compile Blossom V, and to setup the integration with Qsurface automatically.

```
>>> from qsurface.decoders import mwpm
>>> mwpm.get_blossomv()
```

CHAPTER

TWO

USAGE

To simulate the toric code and simulate with bitflip error for 10 iterations and decode with the MWPM decoder:

```
>>> from qsurface.main import initialize, run
>>> code, decoder = initialize((6,6), "toric", "mwpm", enabled_errors=["pauli"])
>>> run(code, decoder, iterations=10, error_rates = {"p_bitflip": 0.1})
{'no_error': 8}
```

Benchmarking of decoders can be enabled by attaching a *benchmarker* object to the decoder. See the docs for the syntax and information to setup benchmarking.

```
>>> from qsurface.main import initialize, run
>>> benchmarker = BenchmarkDecoder({"decode":"duration"})
>>> run(code, decoder, iterations=10, error_rates = {"p_bitflip": 0.1},␣
↪benchmark=benchmarker)
{'no_error': 8,
'benchmark': {'success_rate': [10, 10],
'seed': 12447.413636559,
'durations': {'decode': {'mean': 0.00244155000000319,
'std': 0.002170364089572033}}}}
```

2.1 Plotting

The figures in Qsurface allows for step-by-step visualization of the surface code simulation (and if supported the decoding process). Each figure logs its history such that the user can move backwards in time to view past states of the surface (and decoder). Press `h` when the figure is open for more information.

```
>>> from qsurface.main import initialize, run
>>> code, decoder = initialize((6,6), "toric", "mwpm", enabled_errors=["pauli"],␣
↪plotting=True, initial_states=(0,0))
>>> run(code, decoder, error_rates = {"p_bitflip": 0.1, "p_phaseflip": 0.1}, decode_
↪initial=False)
```

Plotting will be performed on a 3D axis if faulty measurements are enabled.

```
>>> code, decoder = initialize((3,3), "toric", "mwpm", enabled_errors=["pauli"],␣
↪faulty_measurements=True, plotting=True, initial_states=(0,0))
>>> run(code, decoder, error_rates = {"p_bitflip": 0.05, "p_bitflip_plaq": 0.05},␣
↪decode_initial=False)
```

In IPython, inline images are created for each iteration of the plot, which can be tested in the example notebook.

2.2 Command line interface

Simulations can also be initiated from the command line

```
$ python -m qsurface -e pauli -D mwpm -C toric simulation --p_bitflip 0.1 -n 10
{'no_error': 8}
```

For more information on command line interface:

```
$ python -m qsurface -h
usage: qsurface
...
```

This project is proudly funded by the `Unitary Fund <https://unitary.fund/>`_.

CHAPTER

THREE

MODULES

3.1 Running simulations

Contains functions and classes to run and benchmark surface code simulations and visualizations. Use *initialize* to prepare a surface code and a decoder instance, which can be passed on to *run* and *run_multiprocess* to simulate errors and to decode them with the decoder.

qsurface.main.**initialize**(*size*, *Code*, *Decoder*, *enabled_errors=[]*, *faulty_measurements=False*, *plotting=False*, ***kwargs*)

Initializes a code and a decoder.

The function makes sure that the correct class is used to instance the surface code and decoder based on the arguments provided. A code instance must be initalized with `enabled_errors` by `initialize` after class instance to make sure that plot parameters are properly loaded before loading the plotting items included in each included error module, if `plotting` is enabled. See *plot.Template2D* and *errors._template.Plot* for more information.

Parameters

- **size** (Union[Tuple[int, int], int]) – The size of the surface in xy or (x,y).
- **Code** (Union[module, str]) – Any surface code module or module name from codes.
- **Decoder** (Union[module, str]) – Any decoder module or module name from decoders
- **enabled_errors** (List[Union[str, *Sim*]]) – List of error modules from *errors*.
- **faulty_measurements** (bool) – Enable faulty measurements (decode in a 3D lattice).
- **plotting** (bool) – Enable plotting for the surface code and/or decoder.
- **kwargs** – Keyword arguments are passed on to the chosen code, *initialize*, and the chosen decoder.

Examples

To initialize a 6x6 toric code with the MWPM decoder and Pauli errors:

```
>>> initialize((6,6), "toric", "mwpm", enabled_errors=["pauli"], check_
↪compatibility=True)
(<toric (6, 6) PerfectMeasurements>,  <Minimum-Weight Perfect Matching decoder␣
↪(Toric)>)
 This decoder is compatible with the code.
```

Keyword arguments for the code and decoder classes can be included for further customization of class initialization. Note that default errors rates for error class initialization (see *init_errors* and *errors._template.Sim*) can also be provided as keyword arguments here.

```
>>> enabled_errors = ["pauli"]
>>> code_kwargs = {
...     "initial_states": (0,0),
...     "p_bitflip": 0.1,
... }
>>> decoder_kwargs = {
...     "check_compatibility": True,
...     "weighted_union": False,
...     "weighted_growth": False,
... }
>>> initialize((6,6), "toric", "unionfind", enabled_errors=enabled_errors, **code_
↪kwargs, **decoder_kwargs)
This decoder is compatible with the code.
```

qsurface.main.**run**(*code*, *decoder*, *error_rates={}*, *iterations=1*, *decode_initial=True*, *seed=None*, *benchmark=None*, *mp_queue=None*, *mp_process=0*, ***kwargs*)

Runs surface code simulation.

Single command function to run a surface code simulation for a number of iterations.

Parameters

- **code** (*PerfectMeasurements*) – A surface code instance (see *initialize*).
- **decoder** (*Sim*) – A decoder instance (see *initialize*).
- **iterations** (int) – Number of iterations to run.
- **error_rates** (dict) – Dictionary of error rates (see errors). Errors must have been loaded during code class initialization by *initialize* or *init_errors*.
- **decode_initial** (bool) – Decode initial code configuration before applying loaded errors. If random states are used for the data-qubits of the code at class initialization (default behavior), an initial round of decoding is required and is enabled through the decode_initial flag (default is enabled).
- **seed** (Optional[float]) – Float to use as the seed for the random number generator.
- **benchmark** (Optional[*BenchmarkDecoder*]) – Benchmarks decoder performance and analytics if attached.
- **kwargs** – Keyword arguments are passed on to *decode*.

Examples

To simulate the toric code and simulate with bitflip error for 10 iterations and decode with the MWPM decoder:

```
>>> code, decoder = initialize((6,6), "toric", "mwpm", enabled_errors=["pauli"])
>>> run(code, decoder, iterations=10, error_rates = {"p_bitflip": 0.1})
{'no_error': 8}
```

Benchmarked results are updated to the returned dictionary. See *BenchmarkDecoder* for the syntax and information to setup benchmarking.

```
>>> code, decoder = initialize((6,6), "toric", "mwpm", enabled_errors=["pauli"])
>>> benchmarker = BenchmarkDecoder({"decode":"duration"})
>>> run(code, decoder, iterations=10, error_rates = {"p_bitflip": 0.1},␣
↪benchmark=benchmarker)
{'no_error': 8,
'benchmark': {'decoded': 10,
'iterations': 10,
'seed': 12447.413636559,
'durations': {'decode': {'mean': 0.00244155000000319,
'std': 0.002170364089572033}}}}
```

qsurface.main.**run_multiprocess**(*code*, *decoder*, *error_rates={}*, *iterations=1*, *decode_initial=True*, *seed=None*, *processes=1*, *benchmark=None*, ***kwargs*)

Runs surface code simulation using multiple processes.

Using the standard module `multiprocessing` and its `Process` class, several processes are created that each runs its on contained simulation using *run*. The `code` and `decoder` objects are copied such that each process has its own instance. The total number of `iterations` are divided for the number of `processes` indicated. If no `processes` parameter is supplied, the number of available threads is determined via `cpu_count` and all threads are utilized.

If a *BenchmarkDecoder* object is attached to `benchmark`, `Process` copies the object for each separate thread. Each instance of the the decoder thus have its own benchmark object. The results of the benchmark are appended to a list and addded to the output.

See *run* for examples on running a simulation.

Parameters

- **code** (*PerfectMeasurements*) – A surface code instance (see initialize).
- **decoder** (*Sim*) – A decoder instance (see initialize).
- **error_rates** (`dict`) – Dictionary for error rates (see `errors`).
- **iterations** (`int`) – Total number of iterations to run.
- **decode_initial** (`bool`) – Decode initial code configuration before applying loaded errors.
- **seed** (`Optional[float]`) – Float to use as the seed for the random number generator.
- **processes** (`int`) – Number of processes to spawn.
- **benchmark** (`Optional`[*BenchmarkDecoder*]) – Benchmarks decoder performance and analytics if attached.
- **kwargs** – Keyword arguments are passed on to every process of run.

class qsurface.main.**BenchmarkDecoder**(*methods_to_benchmark={}*, *decoder=None*, ***kwargs*)

Benchmarks a decoder during simulation.

A benchmark of a decoder can be performed by attaching the current class to a `decode`. A benchmarker will keep track of the number of simulated iterations and the number of successfull operations by the decoder in `self.data`.

Secondly, a benchmark of the decoder's class methods can be performed by the decorators supplied in the current class, which have the form def `decorator(self, func):`. The approach in the current benchmark class allows for decorating any of the decoder's class methods after it has been instanced. The benefit here is that if no benchmark class is attached, no benchmarking will be performed. The class methods to benchmark must be supplied as a dictionary, where the keys are equivalent to the class method names, and the values are the decorator names. Benchmarked values are stored as class attributes to the benchmark object.

There are two types of decorators, list decorators, which append some value to a dictionary of lists `self.lists`, and value decorators, that saves or updates some value in `self.values`.

Parameters

- **methods_to_benchmark** (dict) – Decoder class methods to benchmark.
- **decoder** (Optional[*Sim*]) – Decoder object.
- **seed** – Logged seed of the simulation.

data
Simulation data.

lists
Benchmarked data by list decorators.

values
Benchmarked data by value decorators.

Examples

To keep track of the duration of each iteration of decoding, the decoder's decode method can be decorated with the duration decorator.

```
>>> code, decoder = initialize((6,6), "toric", "mwpm", enabled_errors=["pauli"])
>>> benchmarker = BenchmarkDecoder({"decode": "duration"}, decoder=decoder)
>>> code.random_errors(p_bitflip=0.1)
>>> decoder.decode()
>>> benchmarker.lists
{'duration': {'decode': [0.0009881999976641964]}}
```

The benchmark class can also be attached to run. The mean and standard deviations of the benchmarked values are in that case updated to the output of run after running *lists_mean_var*.

```
>>> benchmarker = BenchmarkDecoder({"decode":"duration"})
>>> run(code, decoder, iterations=10, error_rates = {"p_bitflip": 0.1},␣
↪benchmark=benchmarker)
{'no_error': 8,
'benchmark': {'success_rate': [10, 10],
'seed': 12447.413636559,
'durations': {'decode': {'mean': 0.00244155000000319,
    'std': 0.002170364089572033}}}}
```

Number of calls to class methods can be counted by the count_calls decorator and stored to self.values. Values in self.values can be saved to a list to, for example, log the value per decoding iteration by the value_to_list decorator. Multiple decorators can be attached to a class method by a list of names in methods_to_benchmark. The logged data are still available in the benchmarker class itself.

```
>>> benchmarker = BenchmarkDecoder({
"decode": ["duration", "value_to_list"],
"correct_edge": "count_calls",
})
>>> run(code, decoder, iterations=10, error_rates = {"p_bitflip": 0.1},␣
↪benchmark=benchmarker)
{'no_error': 8,
```

(continues on next page)

(continued from previous page)

```
'benchmark': {'success_rate': [10, 10],
'seed': '12447.413636559',
'duration': {'decode': {'mean': 0.001886229999945499,
    'std': 0.0007808582199605158}},
'count_calls': {'correct_edge': {'mean': 6.7, 'std': 1.4177446878757827}}}}
>>> benchmarker.lists
    {'duration': {'decode': [0.0030814000019745436,
    0.0015807000017957762,
    0.0010604999988572672,
    0.0035383000031288248,
    0.0018329999984416645,
    0.001753099997586105,
    0.001290500000322936,
    0.0014110999982221983,
    0.0011783000009017996,
    0.0021353999982238747]},
    'count_calls': {'correct_edge': [10, 7, 5, 7, 6, 6, 7, 6, 5, 8]}}
```

Nested class methods can also be benchmarked, e.g. for find of Cluster, which has an alias in unionfind.sim.Toric.

```
>>> code, decoder = initialize((6,6), "toric", "unionfind", enabled_errors=["pauli
↪"])
>>> benchmarker = BenchmarkDecoder({"Cluster.find", "count_calls"})
>>> code.random_errors(p_bitflip=0.1)
>>> decoder.decode()
>>> benchmarker.values
{'count_calls': {'find': 30}}
```

lists_mean_var(*reset=True*)
Get mean and stand deviation of values in `self.lists`.

Parameters **reset** (`bool`) – Resets all in `self.lists` to empty lists.

value_to_list(*func*)
Appends all values in `self.values` to lists in `self.lists`.

duration(*func*)
Logs the duration of `func` in `self.lists`.

count_calls(*func*)
Logs the number of calls to `func` in `self.values`.

3.2 Running a threshold simulation

qsurface.threshold.**run_many**(*Code, Decoder, iterations=1, sizes=[], enabled_errors=[], error_rates=[], faulty_measurements=False, methods_to_benchmark={}, output='', mp_processes=1, recursion_limit=100000, **kwargs*)
Runs a series of simulations of varying sizes and error rates.

A series of simulations are run without plotting for all combinations of `sizes` and `error_rates`. The results are returned as a Pandas DataFrame and saved to the working directory as a csv file. If an existing csv file with the same file name is found, the existing file is loaded and new results are appended to the existing data. A *main.BenchmarkDecoder* object is attached to each simulation to log the seed and other information.

Parameters

- **Code** (Union[module, str]) – Any surface code module or module name from codes.
- **Decoder** (Union[module, str]) – Any decoder module or module name from decoders
- **iterations** (int) – Number of iterations to run per configuration.
- **sizes** (List[Union[int, Tuple[int, int]]]) – The sizes of the surface configurations.
- **enabled_errors** (List[Union[str, *Sim*]]) – List of error modules from *errors*.
- **error_rates** (List[Dict]) – List of dictionaries for error rates per configuration (see errors).
- **faulty_measurements** (bool) – Enable faulty measurements (decode in a 3D lattice).
- **methods_to_benchmark** (dict) – Decoder class methods to benchmark.
- **output** (str) – File name of outputted csv data. If set to "none", no file will be saved.
- **mp_processses** – Number of processes to spawn. For a single process, *run* is used. For multiple processes, run_multiprocess is utilized.

Examples

A series of simulations using the toric surface code and mwpm decoder can be easily setup. Benchmarking can be performed by supplying the methods_to_benchmark argument of the *BenchmarkDecoder* class. The function will initialize a benchmark object of each configuration and append all results as columns to the returned dataframe.

```
>>> data = run_many(
...     "toric",
...     "mwpm",
...     iterations = 1000,
...     sizes = [8,12,16],
...     enabled_errors = ["pauli"],
...     error_rates = [{"p_bitflip: p} for p in [0.09, 0.1, 0.11]],
... )
>>> print(data)
                   datetime  decoded   iterations  no_error  p_bitflip         ␣
↪seed  size
0   04/11/2020 14:45:36   1000.0        1000.0     820.0       0.09  13163.013981␣
↪   8.0
1   04/11/2020 14:45:45   1000.0        1000.0     743.0       0.10  13172.277886␣
↪   8.0
2   04/11/2020 14:45:54   1000.0        1000.0     673.0       0.11  13181.090130␣
↪   8.0
3   04/11/2020 14:46:36   1000.0        1000.0     812.0       0.09  13190.191461␣
↪  12.0
4   04/11/2020 14:47:18   1000.0        1000.0     768.0       0.10  13232.408302␣
↪  12.0
5   04/11/2020 14:48:16   1000.0        1000.0     629.0       0.11  13274.044268␣
↪  12.0
6   04/11/2020 14:51:47   1000.0        1000.0     855.0       0.09  13332.153639␣
↪  16.0
7   04/11/2020 14:55:15   1000.0        1000.0     754.0       0.10  13542.533067␣
↪  16.0
8   04/11/2020 14:59:14   1000.0        1000.0     621.0       0.11  13751.082511␣
↪  16.0
```

(continues on next page)

(continued from previous page)

Return type `Optional[DataFrame]`

`qsurface.threshold.`**`read_csv`**(*file*)

Reads a CSV file parses it as a Pandas DataFrame.

Return type `DataFrame`

`class` `qsurface.threshold.`**`ThresholdFit`**(*modified_ansatz=False*, *p=0.09, 0.1, 0.11*, *A=- inf, 0, inf*, *B=- inf, 0, inf*, *C=- inf, 0, inf*, *D=- 2, 1.6, 2*, *nu=0.8, 0.95, 1.2*, *mu=0, 0.7, 3*)

Fitter for code threshold with data obtained by `~.threshold.run`.

Threshold fitting is performed using the equations described in [wang2003confinement]. The threshold is computing the ground state of the Hamiltonian that described the phase transition or the Nishimori line in the Random Bond Ising Model. The source provides two functions which are included in this fitting class, where the `modified_ansatz` includes a nonuniversion additive correction to correct for finite size effects.

`fit_data`(*data*, *column*, ***kwargs*)

Fits for the code threshold.

Parameters

- **`data`** (`DataFrame`) – Data obtained via `run`.
- **`column`** (`str`) – The column of the DataFrame to fit for.
- **`kwargs`** – Keyword arguments are passed on the `scipy.curve_fit`.

`plot_data`(*data*, *column*, *figure=None*, *rescaled=False*, *scatter_kwargs={'s': 10}*, *line_kwargs={'alpha': 0.5, 'ls': 'dashed', 'lw': 1.5}*, *axis_attributes={'title': 'Threshold', 'xlabel': 'Physical error rate', 'ylabel': 'Logical error rate'}*, *num_x_fit=1000*, ***kwargs*)

Plots the inputted data and the fit for the code threshold.

Parameters

- **`data`** (`DataFrame`) – Data obtained via `run`.
- **`column`** (`str`) – The column of the DataFrame to fit for.
- **`figure`** (`Optional[Figure]`) – If a figure is attached, `show` is not called. Instead, the figure is returned for futher manipulation.
- **`rescaled`** (`bool`) – Plots the data on a rescaled x axis where the fit is a single line.
- **`scatter_kwargs`** (`dict`) – Keyword arguments to pass on to the `scatter` for the markers.
- **`line_kwargs`** (`dict`) – Keyword arguments to pass on to the ~matplotlib.pyplot.plot for the line plot.
- **`axis_attributes`** (`dict`) – Attributes to set of the axis via `axis.set_{attribute}(value)`.
- **`num_x_fit`** (`int`) – Numpy of points to plot for the fit.

3.3 Code elements

class qsurface.codes.elements.**Qubit**(*loc*, *z=0*, **args*, ***kwargs*)
General type qubit object.

This class mainly serves as a superclass or template to other more useful qubit types, which have the apprioate subclass attributes and subclass methods. For other types to to the 'See Also' section.

Parameters

- **loc** (Tuple[float, float]) – Location of the qubit in coordinates.
- **z** (float) – Layer position of qubit. Different layers correspond to time instances of a surface for faulty measurement simulations.

class qsurface.codes.elements.**DataQubit**(**args*, ***kwargs*)
Data type qubit object.

The state of a data-qubit is determined by two *Edge* objects stored in the `self.edges` dictionary. Each of the edges are part of a separate graph on the surface lattice.

edges
Dictionary of edges with the error type as key (e.g. "x" or "z").

self.edges = {"x": Edge_x, "z", Edge_z}

Type dict of *Edge*

state
A class property that calls to each of the edges stored at the `self.edges` attribute and returns all edge states as a dictionary.

Type dict of bool

reinitialized
Indicator for a reinitialized (replaced) data qubit.

Type bool

class qsurface.codes.elements.**AncillaQubit**(**args*, *state_type='default'*, ***kwargs*)
General type qubit object.

An ancilla-qubit is entangled to one or more `DataQubit` objects. The `self.state_type` attribute determines the state on which the measurement is applied. A single measurement is applied when the class property `self.state` is called. The state of the last measurement is stored in `self.measured_state` for state access without prompting a new measurement.

Parameters **state_type** (*str, {"x", "z"}*) – Type of 'codes.elements.Edge' objects belonging to the *DataQubit* objects entangled to the current ancilla-qubit for stabilizer measurements.

parity_qubits
All qubits in this dictionary are entangled to the current ancilla for stabilizer measurements.

Type dict of *DataQubit*

z_neighbors
Neighbor ancilla in the z direction that is an instance of the same qubit at a different time, required for faulty measurements.

Type {*codes.elements.AncillaQubit*: *PseudoEdge*}

state
Property that measures the parity of the qubits in `self.parity_qubits`.

Type bool

measured_state
The result of the last parity measurement.

Type bool

syndrome
Whether the current ancilla is a syndrome.

Type bool

measurement_error
Whether an error occurred during the last measurement.

Type bool

Examples

The state of the entangled `DataQubit` is located at:

```
>>> AncillaQubit.parity_qubits[key].edges[AncillaQubit.state_type]
True
```

measure (*p_bitflip_plaq=0*, *p_bitflip_star=0*, ***kwargs*)
Applies a parity measurement on the ancilla.

The functions loops over all the data qubits in `self.parity_qubits`. For every edge associated with the entangled state on the data qubit, the value of a `parity` boolean is flipped.

Parameters

- **p_bitflip_plaq** (*float*) – Bitflip rate for plaquette (XXXX) operators.
- **p_bitflip_star** (*float*) – Bitflip rate for star (ZZZZ) operators.

Return type `bool`

class `qsurface.codes.elements.`**Edge** (*qubit*, *state_type=''*, *initial_state=None*, ***kwargs*)
A state object belonging to a `DataQubit` object.

An edge cannot have open vertices and must be spanned by two nodes. In this case, the two nodes must be `AncillaQubit` objects, and are stored in `self.nodes`.

Parameters

- **qubit** (`DataQubit`) – Parent qubit object.
- **state_type** (`str`) – Error type associated with the current edge.
- **initial_state** (`Optional[bool]`) – State of the object after initialization.

nodes
The vertices that spans the edge.

Type list of two ~.codes.elements.AncillaQubit` objects

state
The current quantum state on the edge object.

Type bool

add_node (*node*, ***kwargs*)
Adds a node to the edge's `self.nodes` attribute.

class `qsurface.codes.elements.`**PseudoQubit** (**args*, *state_type='default'*, ***kwargs*)
Boundary element, imitates *`codes.elements.AncillaQubit`*.

Edges needs to be spanned by two nodes. For data qubits on the boundary, one of its edges additionally requires an ancilla qubit like node, which is the pseudo-qubit.

measure (*p_bitflip_plaq=0*, *p_bitflip_star=0*, ***kwargs*)
Applies a parity measurement on the ancilla.

The functions loops over all the data qubits in `self.parity_qubits`. For every edge associated with the entangled state on the data qubit, the value of a `parity` boolean is flipped.

Parameters

- **p_bitflip_plaq** (*float*) – Bitflip rate for plaquette (XXXX) operators.
- **p_bitflip_star** (*float*) – Bitflip rate for star (ZZZZ) operators.

Return type `bool`

class `qsurface.codes.elements.`**PseudoEdge** (*qubit*, *state_type=''*, *initial_state=None*, ***kwargs*)
Vertical edge connecting time instances of ancilla-qubits, imitates *`codes.elements.Edge`*.

add_node (*node*, ***kwargs*)
Adds a node to the edge's `self.nodes` attribute.

3.4 Template code

3.4.1 Simulation

class `qsurface.codes._template.sim.`**PerfectMeasurements** (*size*, ***kwargs*)
Simulation code class for perfect measurements.

The qubits of the code class are stored in a double dictionary, with the keys in the outer dictionary corresponding to the qubit layer. For perfect measurements, there is a single layer. For faulty measurements, there are multiple layers (and defaults to `self.size`). In the nested dictionaries each qubit is stored by `qubit.loc` as key. A qubit can thus be accessed by `self.qubits[layer][(x,y)]`.

The qubit and edge classes from *Code elements* can be replaced with inherited classes to store decoder dependent attributes.

Parameters **size** (*int or tuple*) – Size of the surface code in single dimension or two dimensions `(x,y)`.

ancilla_qubits
Nested dictionary of *`AncillaQubit`* objects.

Type dict of dict

data_qubits
Nested dictionary of *`DataQubit`* objects.

Type dict of dict

pseudo_qubits
Nested dictionary of *`PseudoQubit`* objects.

Type dict of dict

errors
Dictionary of error modules with the module name as key. All error modules from *Error types* loaded in `self.errors` will be applied during a simulation by `random_errors()`.

Type dict

logical_operators
Dictionary with lists of `Edge` objects that from a trivial loop over the surface and correspond to a logical operator. The logical state of each operator can be obtained by the state of each Edge in the list.

Type dict of list

logical_state
Dictionary with the states corresponding to the logical operators in `self.logical_operators`.

Type dict of bool

no_error
Property for whether there is a logical error in the last iteration. The value for `self.no_error` is updated after a call to `self.logical_state`.

Type bool

trivial_ancillas
Property for whether all ancillas are trivial. Usefull for checking if decoding has been successfull.

Type bool

instance
Time stamp that is renewed every time `random_errors` is called. Helps with identifying a 'round' of simulation when using class attributes.

Type float

initialize(**args*, ***kwargs*)
Initializes all data objects of the code.

Builds the surface with `init_surface`, adds the logical operators with `init_logical_operator`, and loads error modules with `init_errors`. All keyword arguments from these methods can be used for `initialize`.

abstract init_surface()
Initiates the surface code.

abstract init_logical_operator()
Initiates the logical operators.

init_errors(**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (`Union`[`str`, `Sim`]) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (`dict`) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to `0.1` and erasure to `0.03`.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to `0.05`.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

add_data_qubit (*loc, z=0, initial_states=None, None, **kwargs*)
Initializes a `DataQubit` and saved to `self.data_qubits[z][loc]`.

Parameters **initial_states** (`Tuple[float, float]`) – Initial state for the data-qubit.

Return type `DataQubit`

add_ancilla_qubit (*loc, z=0, state_type='x', **kwargs*)
Initializes a `AncillaQubit` and saved to `self.ancilla_qubits[z][loc]`.

Return type `AncillaQubit`

add_pseudo_qubit (*loc, z=0, state_type='x', **kwargs*)
Initializes a `PseudoQubit` and saved to `self.pseudo_qubits[z][loc]`.

Return type `PseudoQubit`

static entangle_pair (*data_qubit, ancilla_qubit, key, edge=None, **kwargs*)
Entangles one `DataQubit` to a `AncillaQubit` for parity measurement.

Parameters

- **data_qubit** (`DataQubit`) – Control qubit.
- **ancilla_qubit** (`AncillaQubit`) – Controlled qubit.
- **key** (`Any`) – The entanglement is saved by adding the `DataQubit` to `AncillaQubit.parity_qubits[key]`
- **edge** (`Optional[Edge]`) – The edge of the data-qubit to entangle to.

random_errors (*apply_order=None, measure=True, **kwargs*)
Applies all errors loaded in `self.errors` attribute to layer `z`.

The random error is applied for each loaded error module by calling `random_error`. If `apply_order` is specified, the error modules are applied in order of the error names in the list. If no order is specified, the errors are applied in a random order. Addionally, any error rate can set by supplying the rate as a keyword argument e.g. `p_bitflip = 0.1`.

Parameters

- **apply_order** (`Optional[List[str]]`) – The order in which the error modules are applied. Items in the list must equal keys in `self.errors` or the names of the loaded error modules.
- **measure** (`bool`) – Measure ancilla qubits after errors have been simulated.

class qsurface.codes._template.sim.**FaultyMeasurements**(*size*, **args*, *layers=None*, *p_bitflip_plaq=0*, *p_bitflip_star=0*, ***kwargs*)

Simulation code class for faulty measurements.

A 3D graph is initiated with `layers` amount of 2D surfaces from `PerfectMeasurement` stacked on top of each other. The structure of the `self.data_qubits`, `self.ancilla_qubits` and `self.pseudo_qubits` dictionary attributes allows for the storage for various time instances of the same qubits in the first nested layer. E.g. `self.data_qubits[0][(0,0)]` and `self.data_qubits[1][(0,0)]` store the data-qubit at (0,0) at time instances 0 and 1, respectively. Consecutive instances of `AncillaQubit` objects and `PseudoQubit` objects are connected in the 3D graph by `PseudoEdge` objects.

Parameters

- **layers** (`Optional[int]`) – Number of layers in 3D graph for faulty measurements.
- **p_bitflip_plaq** (`float`) – Default bitflip rate during measurements on plaquette operators (XXXX).
- **p_bitflip_star** (`float`) – Default bitflip rate during measurements on star operators (ZZZZ).

simulate(***kwargs*)

Simulate an iteration or errors and measurement.

On all but the final layer, the default or overriding error rates (via keyworded arguments) are applied. On the final layer, perfect measurements are applied by setting `p_bitflip_plaq=0` and `p_bitflip_star=0`.

init_surface(***kwargs*)

Initiates the surface code.

The 3D lattice is initialized by first building the ground layer. After that each consecutive layer is built and pseudo-edges are added to connect the ancilla qubits of each layer.

add_vertical_edge(*lower_ancilla*, *upper_ancilla*, ***kwargs*)

Adds a `PseudoEdge` to connect two instances of an ancilla-qubit in time.

A surface code with faulty measurements must be decoded in 3D. Instances of the same ancilla qubits in time must be connected with an edge. Here, `lower_ancilla` is an older instance of layer 'z', and `upper_ancilla` is a newer instance of layer 'z+1'.

Parameters

- **lower_ancilla** (`AncillaQubit`) – Older instance of ancilla-qubit.
- **upper_ancilla** (`AncillaQubit`) – Newer instance of ancilla-qubit.

random_errors(*p_bitflip_plaq=None*, *p_bitflip_star=None*, ***kwargs*)

Performs a round of parity measurements on layer `z` with faulty measurements.

Parameters

- **p_bitflip_plaq** (*int or float, optional*) – Probability of a bitflip during a parity check measurement on plaquette operators (XXXX).
- **p_bitflip_star** (*int or float, optional*) – Probability of a bitflip during a parity check measurement on star operators (ZZZZ).

random_errors_layer(***kwargs*)

Applies a layer of random errors loaded in `self.errors`.

Parameters **kwargs** – Keyword arguments are passed on to `random_errors`.

add_ancilla_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *AncillaQubit* and saved to `self.ancilla_qubits[z][loc]`.

Return type *AncillaQubit*

add_data_qubit (*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)
Initializes a *DataQubit* and saved to `self.data_qubits[z][loc]`.

Parameters **initial_states** (`Tuple[float, float]`) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *PseudoQubit* and saved to `self.pseudo_qubits[z][loc]`.

Return type *PseudoQubit*

static entangle_pair (*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)
Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.
- **key** (`Any`) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* `.parity_qubits[key]`
- **edge** (`Optional[Edge]`) – The edge of the data-qubit to entangle to.

init_errors (**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (`Union[str, Sim]`) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (`dict`) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to `0.1` and erasure to `0.03`.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to `0.05`.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

abstract init_logical_operator()
Initiates the logical operators.

initialize(**args*, ***kwargs*)
Initializes all data objects of the code.

Builds the surface with *init_surface*, adds the logical operators with *init_logical_operator*, and loads error modules with *init_errors*. All keyword arguments from these methods can be used for *initialize*.

random_measure_layer(***kwargs*)
Measures a layer of ancillas.

If the measured state of the current ancilla is not equal to the measured state of the previous instance, the current ancilla is a syndrome.

Parameters **kwargs** – Keyword arguments are passed on to `get_state`.

3.4.2 Plotting

class qsurface.codes._template.plot.**PerfectMeasurements**(**args*, ***kwargs*)
Plotting template code class for perfect measurements.

figure
Figure object of the current code.

Type *Figure*

initialize(**args*, ***kwargs*)
Initializes the code with a figure. Also takes keyword arguments for *init_plot*.

Since each error object delivers extra plot properties to the figure, which are dependent on the `self.params` values in the figure itself, we must initialize in the following sequence.

- First load figure to load `self.params` instance of the *PlotParams* dataclass.
- Initialize lattice, error initialization must have figure properties
- Draw figure with plot elements from errors

random_errors(**args*, ***kwargs*)
Applies all errors loaded in `self.errors` attribute to layer `z`.

The random error is applied for each loaded error module by calling *random_error*. If `apply_order` is specified, the error modules are applied in order of the error names in the list. If no order is specified, the errors are applied in a random order. Addionally, any error rate can set by supplying the rate as a keyword argument e.g. `p_bitflip = 0.1`.

Parameters

- **apply_order** – The order in which the error modules are applied. Items in the list must equal keys in `self.errors` or the names of the loaded error modules.
- **measure** – Measure ancilla qubits after errors have been simulated.

show_corrected(***kwargs*)
Redraws the qubits and ancillas to show their states after decoding.

plot_data(*iter_name=None*, ***kwargs*)
Update plots of all data-qubits. A plot iteration is added if a `iter_name` is supplied. See *draw_figure*.

plot_ancilla(*iter_name=None*, ***kwargs*)
Update plots of all ancilla-qubits. A plot iteration is added if a `iter_name` is supplied. See *draw_figure*.

class Figure(*code*, **args*, ***kwargs*)
Surface code plot for perfect measurements.

The inner figure class that plots the surface code based on the `Qubit.loc` and `Qubit.z` values on the set of `code.data_qubits`, `code.ancilla_qubits` and `code.pseudo_qubits`. This allows for a high amount of code inheritance.

An additional `matplotlib.widgets.RadioButtons` object is added to the figure which allows for the user to choose one of the loaded errors and apply the error directly to a qubit via `_pick_handler`.

Parameters

- **code** (*PerfectMeasurements*) – Surface code instance.
- **kwargs** – Keyword arguments are passed on to *plot.Template2D*.

error_methods
A dictionary of the various error methods loaded in the outer class.
Type dict

code_params
Additional plotting parameters loaded to the *plot.PlotParams* instance at `self.params`.

init_plot(***kwargs*)
Plots all elements of the surface code onto the figure. Also takes keyword arguments for *init_legend*.

An additional `matplotlib.widgets.RadioButtons` object is added to the figure which allows for the user to choose one of the loaded errors and apply the error directly to a qubit via `_pick_handler`. This object is added via the *init_plot* method to make sure that the errors are already loaded in `self.code.errors`. The method for each loaded error is saved to `self.error_methods`. See *errors._template.Plot* for more information.

init_legend(*legend_items=[]*, ***kwargs*)
Initializes the legend of the main axis of the figure. Also takes keyword arguments for `legend`.

The legend of the main axis `self.main_ax` consists of a series of `Line2D` objects. The qubit, vertex and stars are always in the legend for a surface code plot. Any error from *Error types* loaded in the code at `code.errors` in de outer class will add an extra element to the legend for differentiation if an error occurs. The `Line2D` attributes are stored at `error.Plot.legend_params` of the error module (see *errors._template.Plot*).
Parameters legend_items (list of `Line2D`, optional) – Additional elements to the legend.

static change_properties(*artist*, *prop_dict*)
Changes the plot properties and draw the plot object or artist.

close()
Closes the figure.

draw_figure(*new_iter_name=None*, *output=True*, *carriage_return=False*, ***kwargs*)
Draws the canvas and blocks code execution.

Draws the queued plot changes onto the canvas and calls for *focus()* which blocks the code execution and catches user input for history navigation.

If a new iteration is called by supplying a `new_iter_name`, we additionally check for future property changes in the `self.future_dict`, and add these changes to the queue. Finally, all queued property changes for the next iteration are applied by `change_properties`.

Parameters

- **new_iter_name** (Optional[str]) – Name of the new iteration. If no name is supplied, no new iteration is called.
- **output** (bool) – Prints information to the console.
- **carriage_return** (bool) – Applies carriage return to remove last line printed.

See also:

`focus()`, `change_properties()`

focus()

Enables the blocking object, catches input for history navigation.

The BlockingKeyInput object is called which blocks the execution of the code. During this block, the user input is received by the blocking object and return to the current method. From here, we can manipulate the plot or move through the plot history and call `focus()` again when all changes in the history have been drawn and blit.

key	function
h	show help
i	show all iterations
d	redraw current iteration
enter or right	go to next iteration, enter iteration number
backspace or left	go to previous iteration
n	go to newest iteration
0-9	input iteration number

When the method is active, the focus is on the figure. This will be indicated by a green circle in the bottom right of the figure. When the focus is lost, the code execution is continued and the icon is red. The change is icon color is performed by `_set_figure_state()`, which also hides the interactive elements when the focus is lost.

property history_at_newest

load_interactive_backend()

Configures the plotting backend.

If the Tkinter backend is enabled or can be enabled, the function returns True. For other backends False is returned.

Return type bool

new_artist(*artist*, *axis=None*)

Adds a new artist to the `axis`.

Newly added artists must be hidden in the previous iteration. To make sure the history is properly logged, the visibility of the `artist` is set to `False`, and a new property of shown visibility is added to the queue of the next iteration.

Parameters

- **artist** (Artist) – New plot artist to add to the `axis`.
- **axis** (Optional[Axes]) – Axis to add the figure to.

Return type None

new_properties(*artist*, *properties*, *saved_properties={}*, ***kwargs*)

Parses a dictionary of property changes of a *matplotlib* artist.

New properties are supplied via `properties`. If any of the new properties is different from its current value, this is seen as a property change. The old property value is stored in `self.history_dict[self.history_iteration]`, and the new property value is stored at `self.`

`history_dict[self.history_iteration+1]`. These new properties are *queued* for the next interation. The queue is emptied by applying all changes when *draw_figure* is called. If the same property changes 2+ times within the same iteration, the previous property change is removed with `next_prop.pop(key, None)`.

The `saved_properties` parameter is used when temporary property changes have been applied by `temporary_changes`, in which the original properties are saved to `self.temporary_saved` as the saved properties. Before a new iteration is drawn, the temporary changes, which can be overwritten, are compared with the saved changes and the differences in properties are saved to `[self.history_dict[self.history_iter-1]]` and `self.history_dict[self.history_iteration]`.

Some color values from different *matplotlib* objects are nested, some are list or tuple, and others may be a `numpy.ndarray`. The nested methods `get_nested()` and `get_nested_property()` make sure that the return type is always a list.

Parameters

- **artist** (Artist) – Plot object whose properties are changed.
- **properties** (dict) – Plot properties to change.
- **saved_properties** (dict) – Override current properties and parse previous and current history.

temporary_properties (*artist*, *properties*, ***kwargs*)

Applies temporary property changes to a *matplotlib* artist.

Only available on the newest iteration, as we cannot change what is already in the past. All values in `properties` are immediately applied to `artist`. Since temporary changes can be overwritten within the same iteration, the first time a temporary property change is requested, the previous value is saved to `self.temporary_saved`. When the iteration changes, the property differences of the previous and current iteration are recomputed and saved to `self.history_dict` in `_draw_from_history()`.

Parameters

- **artist** (Artist) – Plot object whose properties are changed.
- **properties** (dict) – Plot properties to change.

add_ancilla_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)

Initializes a *AncillaQubit* and saved to `self.ancilla_qubits[z][loc]`.

Return type *AncillaQubit*

add_data_qubit (*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)

Initializes a *DataQubit* and saved to `self.data_qubits[z][loc]`.

Parameters initial_states (Tuple[float, float]) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)

Initializes a *PseudoQubit* and saved to `self.pseudo_qubits[z][loc]`.

Return type *PseudoQubit*

static entangle_pair (*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)

Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.
- **key** (Any) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* `.parity_qubits[key]`

- **edge** (Optional[*Edge*]) – The edge of the data-qubit to entangle to.

init_errors (**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (Union[str, *Sim*]) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (dict) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to `0.1` and erasure to `0.03`.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to `0.05`.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

abstract init_logical_operator()
Initiates the logical operators.

abstract init_surface()
Initiates the surface code.

class qsurface.codes._template.plot.**FaultyMeasurements**(**args*, *figure3d=True*, ***kwargs*)
Plotting template code class for faulty measurements.

Inherits from *codes._template.sim.FaultyMeasurements* and *codes._template.plot.PerfectMeasurements*. See documentation for these classes for more.

Dependent on the `figure3d` argument, either a 3D figure object is created that inherits from *Template3D* and *codes._template.plot.PerfectMeasurements.Figure*, or the 2D *codes._template.plot.PerfectMeasurements.Figure* is used.

Parameters

- **args** – Positional arguments are passed on to *codes._template.sim.FaultyMeasurements*.
- **figure3d** (bool) – Enables plotting on a 3D lattice. Disable to plot layer-by-layer on a 2D lattice, which increases responsiveness.
- **kwargs** – Keyword arguments are passed on to *codes._template.sim.FaultyMeasurements* and the figure object.

3.5 Code types

All surface code modules in this section inherit from the template surface code module, see *Template code*.

3.5.1 Toric code

Simulation

class qsurface.codes.toric.sim.**PerfectMeasurements**(*size*, ***kwargs*)

init_surface(*z=0*, ***kwargs*)
Initializes the toric surface code on layer `z`.

Parameters z (*int or float, optional*) – Layer of qubits, `z=0` for perfect measurements.

init_parity_check(*ancilla_qubit*, ***kwargs*)
Initiates a parity check measurement.

For every ancilla qubit on `(x,y)`, four neighboring data qubits are entangled for parity check measurements. They are stored via the wind-directional keys.

Parameters ancilla_qubit (*AncillaQubit*) – Ancilla-qubit to initialize.

init_logical_operator(***kwargs*)
Initiates the logical operators [x1, x2, z1, z2] of the toric code.

add_ancilla_qubit(*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *AncillaQubit* and saved to `self.ancilla_qubits[z][loc]`.

Return type *AncillaQubit*

add_data_qubit(*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)
Initializes a *DataQubit* and saved to `self.data_qubits[z][loc]`.

Parameters initial_states (`Tuple[float, float]`) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit(*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *PseudoQubit* and saved to `self.pseudo_qubits[z][loc]`.

Return type *PseudoQubit*

static entangle_pair(*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)
Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.
- **key** (`Any`) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* `.parity_qubits[key]`
- **edge** (`Optional[Edge]`) – The edge of the data-qubit to entangle to.

init_errors(**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (`Union[str, Sim]`) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (`dict`) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to `0.1` and erasure to `0.03`.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to `0.05`.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

initialize (**args*, ***kwargs*)
Initializes all data objects of the code.

Builds the surface with `init_surface`, adds the logical operators with `init_logical_operator`, and loads error modules with `init_errors`. All keyword arguments from these methods can be used for `initialize`.

random_errors (*apply_order=None*, *measure=True*, ***kwargs*)
Applies all errors loaded in `self.errors` attribute to layer `z`.

The random error is applied for each loaded error module by calling `random_error`. If `apply_order` is specified, the error modules are applied in order of the error names in the list. If no order is specified, the errors are applied in a random order. Addionally, any error rate can set by supplying the rate as a keyword argument e.g. `p_bitflip = 0.1`.

Parameters

- **apply_order** (`Optional[List[str]]`) – The order in which the error modules are applied. Items in the list must equal keys in `self.errors` or the names of the loaded error modules.
- **measure** (`bool`) – Measure ancilla qubits after errors have been simulated.

class `qsurface.codes.toric.sim.`**FaultyMeasurements** (*size*, **args*, *layers=None*, *p_bitflip_plaq=0*, *p_bitflip_star=0*, ***kwargs*)

add_ancilla_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a `AncillaQubit` and saved to `self.ancilla_qubits[z][loc]`.

Return type `AncillaQubit`

add_data_qubit (*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)
Initializes a *DataQubit* and saved to `self.data_qubits[z][loc]`.

Parameters **initial_states** (Tuple[float, float]) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *PseudoQubit* and saved to `self.pseudo_qubits[z][loc]`.

Return type *PseudoQubit*

add_vertical_edge (*lower_ancilla*, *upper_ancilla*, ***kwargs*)
Adds a *PseudoEdge* to connect two instances of an ancilla-qubit in time.

A surface code with faulty measurements must be decoded in 3D. Instances of the same ancilla qubits in time must be connected with an edge. Here, `lower_ancilla` is an older instance of layer 'z', and `upper_ancilla` is a newer instance of layer 'z+1'.

Parameters

- **lower_ancilla** (*AncillaQubit*) – Older instance of ancilla-qubit.
- **upper_ancilla** (*AncillaQubit*) – Newer instance of ancilla-qubit.

static entangle_pair (*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)
Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.
- **key** (Any) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* `.parity_qubits[key]`
- **edge** (Optional[*Edge*]) – The edge of the data-qubit to entangle to.

init_errors (**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (Union[str, *Sim*]) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (dict) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to `0.1` and erasure to `0.03`.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to `0.05`.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

init_logical_operator(***kwargs*)
Initiates the logical operators [x1, x2, z1, z2] of the toric code.

init_parity_check(*ancilla_qubit*, ***kwargs*)
Initiates a parity check measurement.

For every ancilla qubit on `(x,y)`, four neighboring data qubits are entangled for parity check measurements. They are stored via the wind-directional keys.

Parameters **ancilla_qubit** (*AncillaQubit*) – Ancilla-qubit to initialize.

init_surface(***kwargs*)
Initiates the surface code.

The 3D lattice is initialized by first building the ground layer. After that each consecutive layer is built and pseudo-edges are added to connect the ancilla qubits of each layer.

initialize(**args*, ***kwargs*)
Initializes all data objects of the code.

Builds the surface with *init_surface*, adds the logical operators with *init_logical_operator*, and loads error modules with *init_errors*. All keyword arguments from these methods can be used for *initialize*.

random_errors(*p_bitflip_plaq=None*, *p_bitflip_star=None*, ***kwargs*)
Performs a round of parity measurements on layer `z` with faulty measurements.

Parameters

- **p_bitflip_plaq** (*int or float, optional*) – Probability of a bitflip during a parity check measurement on plaquette operators (XXXX).
- **p_bitflip_star** (*int or float, optional*) – Probability of a bitflip during a parity check measurement on star operators (ZZZZ).

random_errors_layer(***kwargs*)
Applies a layer of random errors loaded in `self.errors`.

Parameters **kwargs** – Keyword arguments are passed on to *random_errors*.

random_measure_layer(***kwargs*)
Measures a layer of ancillas.

If the measured state of the current ancilla is not equal to the measured state of the previous instance, the current ancilla is a syndrome.

Parameters **kwargs** – Keyword arguments are passed on to `get_state`.

simulate(***kwargs*)
Simulate an iteration or errors and measurement.

On all but the final layer, the default or overriding error rates (via keyworded arguments) are applied. On the final layer, perfect measurements are applied by setting `p_bitflip_plaq=0` and `p_bitflip_star=0`.

Plotting

class qsurface.codes.toric.plot.**PerfectMeasurements**(**args*, ***kwargs*)

class Figure(*code*, **args*, ***kwargs*)

static change_properties(*artist*, *prop_dict*)
Changes the plot properties and draw the plot object or artist.

close()
Closes the figure.

draw_figure(*new_iter_name=None*, *output=True*, *carriage_return=False*, ***kwargs*)
Draws the canvas and blocks code execution.

Draws the queued plot changes onto the canvas and calls for *focus()* which blocks the code execution and catches user input for history navigation.

If a new iteration is called by supplying a `new_iter_name`, we additionally check for future property changes in the `self.future_dict`, and add these changes to the queue. Finally, all queued property changes for the next iteration are applied by *change_properties*.

Parameters
- **new_iter_name** (`Optional[str]`) – Name of the new iteration. If no name is supplied, no new iteration is called.
- **output** (`bool`) – Prints information to the console.
- **carriage_return** (`bool`) – Applies carriage return to remove last line printed.

See also:

focus(), *change_properties()*

focus()
Enables the blocking object, catches input for history navigation.

The BlockingKeyInput object is called which blocks the execution of the code. During this block, the user input is received by the blocking object and return to the current method. From here, we can manipulate the plot or move through the plot history and call *focus()* again when all changes in the history have been drawn and blit.

key	function
h	show help
i	show all iterations
d	redraw current iteration
enter or right	go to next iteration, enter iteration number
backspace or left	go to previous iteration
n	go to newest iteration
0-9	input iteration number

When the method is active, the focus is on the figure. This will be indicated by a green circle in the bottom right of the figure. When the focus is lost, the code execution is continued and the icon is red. The change is icon color is performed by `_set_figure_state()`, which also hides the interactive elements when the focus is lost.

property history_at_newest

init_legend(*legend_items=[]*, ***kwargs*)

Initializes the legend of the main axis of the figure. Also takes keyword arguments for `legend`.

The legend of the main axis `self.main_ax` consists of a series of `Line2D` objects. The qubit, vertex and stars are always in the legend for a surface code plot. Any error from *Error types* loaded in the code at `code.errors` in de outer class will add an extra element to the legend for differentiation if an error occurs. The `Line2D` attributes are stored at `error.Plot.legend_params` of the error module (see *errors._template.Plot*).

Parameters legend_items (list of `Line2D`, optional) – Additional elements to the legend.

init_plot(***kwargs*)

Plots all elements of the surface code onto the figure. Also takes keyword arguments for *init_legend*.

An additional `matplotlib.widgets.RadioButtons` object is added to the figure which allows for the user to choose one of the loaded errors and apply the error directly to a qubit via `_pick_handler`. This object is added via the *init_plot* method to make sure that the errors are already loaded in `self.code.errors`. The method for each loaded error is saved to `self.error_methods`. See *errors._template.Plot* for more information.

load_interactive_backend()

Configures the plotting backend.

If the Tkinter backend is enabled or can be enabled, the function returns True. For other backends False is returned.

Return type `bool`

new_artist(*artist*, *axis=None*)

Adds a new artist to the `axis`.

Newly added artists must be hidden in the previous iteration. To make sure the history is properly logged, the visibility of the `artist` is set to `False`, and a new property of shown visibility is added to the queue of the next iteration.

Parameters

- **artist** (`Artist`) – New plot artist to add to the `axis`.
- **axis** (`Optional`[`Axes`]) – Axis to add the figure to.

Return type `None`

new_properties(*artist*, *properties*, *saved_properties={}*, ***kwargs*)

Parses a dictionary of property changes of a *matplotlib* artist.

New properties are supplied via `properties`. If any of the new properties is different from its current value, this is seen as a property change. The old property value is stored in `self.history_dict[self.history_iteration]`, and the new property value is stored at `self.history_dict[self.history_iteration+1]`. These new properties are *queued* for the next interation. The queue is emptied by applying all changes when *draw_figure* is called. If the same property changes 2+ times within the same iteration, the previous property change is removed with `next_prop.pop(key, None)`.

The `saved_properties` parameter is used when temporary property changes have been applied by `temporary_changes`, in which the original properties are saved to `self.temporary_saved` as the saved properties. Before a new iteration is drawn, the temporary changes, which can be overwritten, are compared with the saved changes and the differences in properties are saved to `[self.history_dict[self.history_iter-1]]` and `self.history_dict[self.history_iteration]`.

Some color values from different *matplotlib* objects are nested, some are list or tuple, and others may be a `numpy.ndarray`. The nested methods `get_nested()` and `get_nested_property()` make sure that the return type is always a list.

Parameters

- **artist** (`Artist`) – Plot object whose properties are changed.
- **properties** (`dict`) – Plot properties to change.
- **saved_properties** (`dict`) – Override current properties and parse previous and current history.

temporary_properties(*artist*, *properties*, ***kwargs*)

Applies temporary property changes to a *matplotlib* artist.

Only available on the newest iteration, as we cannot change what is already in the past. All values in `properties` are immediately applied to `artist`. Since temporary changes can be overwritten within the same iteration, the first time a temporary property change is requested, the previous value is saved to `self.temporary_saved`. When the iteration changes, the property differences of the previous and current iteration are recomputed and saved to `self.history_dict` in `_draw_from_history()`.

Parameters

- **artist** (`Artist`) – Plot object whose properties are changed.
- **properties** (`dict`) – Plot properties to change.

add_ancilla_qubit(*loc*, *z=0*, *state_type='x'*, ***kwargs*)

Initializes a *AncillaQubit* and saved to `self.ancilla_qubits[z][loc]`.

Return type *AncillaQubit*

add_data_qubit(*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)

Initializes a *DataQubit* and saved to `self.data_qubits[z][loc]`.

Parameters **initial_states** (`Tuple[float, float]`) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit(*loc*, *z=0*, *state_type='x'*, ***kwargs*)

Initializes a *PseudoQubit* and saved to `self.pseudo_qubits[z][loc]`.

Return type *PseudoQubit*

static entangle_pair(*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)

Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.
- **key** (`Any`) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* `.parity_qubits[key]`
- **edge** (`Optional[Edge]`) – The edge of the data-qubit to entangle to.

init_errors(**error_modules*, *error_rates={}*, ***kwargs*)

Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (Union[str, *Sim*]) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (dict) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to 0.1 and erasure to 0.03.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to 0.05.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

init_logical_operator (***kwargs*)
Initiates the logical operators [x1, x2, z1, z2] of the toric code.

init_parity_check (*ancilla_qubit*, ***kwargs*)
Initiates a parity check measurement.

For every ancilla qubit on (x,y), four neighboring data qubits are entangled for parity check measurements. They are stored via the wind-directional keys.

Parameters **ancilla_qubit** (*AncillaQubit*) – Ancilla-qubit to initialize.

init_surface (*z=0*, ***kwargs*)
Initializes the toric surface code on layer z.

Parameters **z** (*int or float, optional*) – Layer of qubits, z=0 for perfect measurements.

initialize (**args*, ***kwargs*)
Initializes the code with a figure. Also takes keyword arguments for *init_plot*.

Since each error object delivers extra plot properties to the figure, which are dependent on the self.params values in the figure itself, we must initialize in the following sequence.

- First load figure to load self.params instance of the *PlotParams* dataclass.
- Initialize lattice, error initialization must have figure properties
- Draw figure with plot elements from errors

plot_ancilla (*iter_name=None*, ***kwargs*)
Update plots of all ancilla-qubits. A plot iteration is added if a iter_name is supplied. See *draw_figure*.

plot_data (*iter_name=None*, ***kwargs*)
Update plots of all data-qubits. A plot iteration is added if a iter_name is supplied. See *draw_figure*.

random_errors(**args*, ***kwargs*)
Applies all errors loaded in `self.errors` attribute to layer `z`.

The random error is applied for each loaded error module by calling *random_error*. If `apply_order` is specified, the error modules are applied in order of the error names in the list. If no order is specified, the errors are applied in a random order. Addionally, any error rate can set by supplying the rate as a keyword argument e.g. `p_bitflip = 0.1`.

Parameters

- **apply_order** – The order in which the error modules are applied. Items in the list must equal keys in `self.errors` or the names of the loaded error modules.
- **measure** – Measure ancilla qubits after errors have been simulated.

show_corrected(***kwargs*)
Redraws the qubits and ancillas to show their states after decoding.

class qsurface.codes.toric.plot.**FaultyMeasurements**(**args*, *figure3d=True*, ***kwargs*)
Plotting code class for faulty measurements.

Inherits from *codes.toric.sim.FaultyMeasurements* and *codes.toric.plot.PerfectMeasurements*. See documentation for these classes for more.

Dependent on the `figure3d` argument, either a 3D figure object is created that inherits from *Template3D* and *codes.toric.plot.PerfectMeasurements.Figure*, or the 2D *codes.toric.plot.PerfectMeasurements.Figure* is used.

Parameters

- **args** – Positional arguments are passed on to *codes.toric.sim.FaultyMeasurements*.
- **figure3d** (`bool`) – Enables plotting on a 3D lattice. Disable to plot layer-by-layer on a 2D lattice, which increases responsiveness.
- **kwargs** – Keyword arguments are passed on to *codes.toric.sim.FaultyMeasurements* and the figure object.

3.5.2 Planar code

Simulation

class qsurface.codes.planar.sim.**PerfectMeasurements**(*size*, ***kwargs*)

init_surface(*z=0*, ***kwargs*)
Initializes the planar surface code on layer `z`.

Parameters **z** (*int or float, optional*) – Layer of qubits, `z=0` for perfect measurements.

init_parity_check(*ancilla_qubit*, ***kwargs*)
Initiates a parity check measurement.

For every ancilla qubit on `(x,y)`, four neighboring data qubits are entangled for parity check measurements.

Parameters **ancilla_qubit** (*AncillaQubit*) – Ancilla qubit to initialize.

init_logical_operator(***kwargs*)
Initiates the logical operators [x,z] of the planar code.

add_ancilla_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *AncillaQubit* and saved to `self.ancilla_qubits[z][loc]`.

Return type *AncillaQubit*

add_data_qubit (*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)
Initializes a *DataQubit* and saved to `self.data_qubits[z][loc]`.

Parameters **initial_states** (Tuple[float, float]) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *PseudoQubit* and saved to `self.pseudo_qubits[z][loc]`.

Return type *PseudoQubit*

static entangle_pair (*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)
Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.
- **key** (Any) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* `.parity_qubits[key]`
- **edge** (Optional[*Edge*]) – The edge of the data-qubit to entangle to.

init_errors (**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (Union[str, *Sim*]) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (dict) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to `0.1` and erasure to `0.03`.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to `0.05`.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

initialize(**args*, ***kwargs*)
Initializes all data objects of the code.

Builds the surface with *init_surface*, adds the logical operators with *init_logical_operator*, and loads error modules with *init_errors*. All keyword arguments from these methods can be used for *initialize*.

random_errors(*apply_order=None*, *measure=True*, ***kwargs*)
Applies all errors loaded in `self.errors` attribute to layer `z`.

The random error is applied for each loaded error module by calling *random_error*. If `apply_order` is specified, the error modules are applied in order of the error names in the list. If no order is specified, the errors are applied in a random order. Addionally, any error rate can set by supplying the rate as a keyword argument e.g. `p_bitflip = 0.1`.

Parameters

- **apply_order** (`Optional[List[str]]`) – The order in which the error modules are applied. Items in the list must equal keys in `self.errors` or the names of the loaded error modules.
- **measure** (`bool`) – Measure ancilla qubits after errors have been simulated.

class `qsurface.codes.planar.sim.`**FaultyMeasurements**(*size*, **args*, *layers=None*, *p_bitflip_plaq=0*, *p_bitflip_star=0*, ***kwargs*)

add_ancilla_qubit(*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *AncillaQubit* and saved to `self.ancilla_qubits[z][loc]`.

Return type *AncillaQubit*

add_data_qubit(*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)
Initializes a *DataQubit* and saved to `self.data_qubits[z][loc]`.

Parameters **initial_states** (`Tuple[float, float]`) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit(*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *PseudoQubit* and saved to `self.pseudo_qubits[z][loc]`.

Return type *PseudoQubit*

add_vertical_edge(*lower_ancilla*, *upper_ancilla*, ***kwargs*)
Adds a *PseudoEdge* to connect two instances of an ancilla-qubit in time.

A surface code with faulty measurements must be decoded in 3D. Instances of the same ancilla qubits in time must be connected with an edge. Here, `lower_ancilla` is an older instance of layer 'z', and `upper_ancilla` is a newer instance of layer 'z+1'.

Parameters

- **lower_ancilla** (*AncillaQubit*) – Older instance of ancilla-qubit.
- **upper_ancilla** (*AncillaQubit*) – Newer instance of ancilla-qubit.

static entangle_pair(*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)
Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.

- **key** (Any) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* .parity_qubits[key]
- **edge** (Optional[*Edge*]) – The edge of the data-qubit to entangle to.

init_errors (**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (Union[str, *Sim*]) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (dict) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to 0.1 and erasure to 0.03.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to 0.05.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

init_logical_operator (***kwargs*)
Initiates the logical operators [x,z] of the planar code.

init_parity_check (*ancilla_qubit*, ***kwargs*)
Initiates a parity check measurement.

For every ancilla qubit on (x,y), four neighboring data qubits are entangled for parity check measurements.

Parameters **ancilla_qubit** (*AncillaQubit*) – Ancilla qubit to initialize.

init_surface (***kwargs*)
Initiates the surface code.

The 3D lattice is initialized by first building the ground layer. After that each consecutive layer is built and pseudo-edges are added to connect the ancilla qubits of each layer.

initialize (**args*, ***kwargs*)
Initializes all data objects of the code.

Builds the surface with *init_surface*, adds the logical operators with *init_logical_operator*, and loads error modules with *init_errors*. All keyword arguments from these methods can be used for *initialize*.

random_errors (*p_bitflip_plaq=None*, *p_bitflip_star=None*, ***kwargs*)

Performs a round of parity measurements on layer `z` with faulty measurements.

Parameters

- **p_bitflip_plaq** (*int or float, optional*) – Probability of a bitflip during a parity check measurement on plaquette operators (XXXX).
- **p_bitflip_star** (*int or float, optional*) – Probability of a bitflip during a parity check measurement on star operators (ZZZZ).

random_errors_layer (***kwargs*)

Applies a layer of random errors loaded in `self.errors`.

Parameters **kwargs** – Keyword arguments are passed on to *random_errors*.

random_measure_layer (***kwargs*)

Measures a layer of ancillas.

If the measured state of the current ancilla is not equal to the measured state of the previous instance, the current ancilla is a syndrome.

Parameters **kwargs** – Keyword arguments are passed on to `get_state`.

simulate (***kwargs*)

Simulate an iteration or errors and measurement.

On all but the final layer, the default or overriding error rates (via keyworded arguments) are applied. On the final layer, perfect measurements are applied by setting `p_bitflip_plaq=0` and `p_bitflip_star=0`.

Plotting

class qsurface.codes.planar.plot.**PerfectMeasurements** (**args*, ***kwargs*)

class Figure (*code*, **args*, ***kwargs*)

static change_properties (*artist*, *prop_dict*)

Changes the plot properties and draw the plot object or artist.

close ()

Closes the figure.

draw_figure (*new_iter_name=None*, *output=True*, *carriage_return=False*, ***kwargs*)

Draws the canvas and blocks code execution.

Draws the queued plot changes onto the canvas and calls for *focus()* which blocks the code execution and catches user input for history navigation.

If a new iteration is called by supplying a `new_iter_name`, we additionally check for future property changes in the `self.future_dict`, and add these changes to the queue. Finally, all queued property changes for the next iteration are applied by *change_properties*.

Parameters

- **new_iter_name** (`Optional[str]`) – Name of the new iteration. If no name is supplied, no new iteration is called.
- **output** (`bool`) – Prints information to the console.
- **carriage_return** (`bool`) – Applies carriage return to remove last line printed.

See also:

focus(), *change_properties()*

focus()

Enables the blocking object, catches input for history navigation.

The BlockingKeyInput object is called which blocks the execution of the code. During this block, the user input is received by the blocking object and return to the current method. From here, we can manipulate the plot or move through the plot history and call *focus()* again when all changes in the history have been drawn and blit.

key	function
h	show help
i	show all iterations
d	redraw current iteration
enter or right	go to next iteration, enter iteration number
backspace or left	go to previous iteration
n	go to newest iteration
0-9	input iteration number

When the method is active, the focus is on the figure. This will be indicated by a green circle in the bottom right of the figure. When the focus is lost, the code execution is continued and the icon is red. The change is icon color is performed by `_set_figure_state()`, which also hides the interactive elements when the focus is lost.

property history_at_newest

init_legend(*legend_items=[]*, ***kwargs*)

Initializes the legend of the main axis of the figure. Also takes keyword arguments for `legend`.

The legend of the main axis `self.main_ax` consists of a series of `Line2D` objects. The qubit, vertex and stars are always in the legend for a surface code plot. Any error from *Error types* loaded in the code at `code.errors` in de outer class will add an extra element to the legend for differentiation if an error occurs. The `Line2D` attributes are stored at `error.Plot.legend_params` of the error module (see *errors._template.Plot*).

Parameters **legend_items** (list of `Line2D`, optional) – Additional elements to the legend.

init_plot(***kwargs*)

Plots all elements of the surface code onto the figure. Also takes keyword arguments for *init_legend*.

An additional `matplotlib.widgets.RadioButtons` object is added to the figure which allows for the user to choose one of the loaded errors and apply the error directly to a qubit via `_pick_handler`. This object is added via the *init_plot* method to make sure that the errors are already loaded in `self.code.errors`. The method for each loaded error is saved to `self.error_methods`. See *errors._template.Plot* for more information.

load_interactive_backend()

Configures the plotting backend.

If the Tkinter backend is enabled or can be enabled, the function returns True. For other backends False is returned.

Return type `bool`

new_artist(*artist*, *axis=None*)

Adds a new artist to the `axis`.

Newly added artists must be hidden in the previous iteration. To make sure the history is properly logged, the visibility of the `artist` is set to `False`, and a new property of shown visibility is added to the queue of the next iteration.

Parameters
- **artist** (Artist) – New plot artist to add to the axis.
- **axis** (Optional[Axes]) – Axis to add the figure to.

Return type None

new_properties (*artist*, *properties*, *saved_properties={}*, ***kwargs*)
Parses a dictionary of property changes of a *matplotlib* artist.

New properties are supplied via properties. If any of the new properties is different from its current value, this is seen as a property change. The old property value is stored in self.history_dict[self.history_iteration], and the new property value is stored at self.history_dict[self.history_iteration+1]. These new properties are *queued* for the next interation. The queue is emptied by applying all changes when *draw_figure* is called. If the same property changes 2+ times within the same iteration, the previous property change is removed with next_prop.pop(key, None).

The saved_properties parameter is used when temporary property changes have been applied by temporary_changes, in which the original properties are saved to self.temporary_saved as the saved properties. Before a new iteration is drawn, the temporary changes, which can be overwritten, are compared with the saved changes and the differences in properties are saved to [self.history_dict[self.history_iter-1]] and self.history_dict[self.history_iteration].

Some color values from different *matplotlib* objects are nested, some are list or tuple, and others may be a numpy.ndarray. The nested methods get_nested() and get_nested_property() make sure that the return type is always a list.

Parameters
- **artist** (Artist) – Plot object whose properties are changed.
- **properties** (dict) – Plot properties to change.
- **saved_properties** (dict) – Override current properties and parse previous and current history.

temporary_properties (*artist*, *properties*, ***kwargs*)
Applies temporary property changes to a *matplotlib* artist.

Only available on the newest iteration, as we cannot change what is already in the past. All values in properties are immediately applied to artist. Since temporary changes can be overwritten within the same iteration, the first time a temporary property change is requested, the previous value is saved to self.temporary_saved. When the iteration changes, the property differences of the previous and current iteration are recomputed and saved to self.history_dict in _draw_from_history().

Parameters
- **artist** (Artist) – Plot object whose properties are changed.
- **properties** (dict) – Plot properties to change.

add_ancilla_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *AncillaQubit* and saved to self.ancilla_qubits[z][loc].

Return type *AncillaQubit*

add_data_qubit (*loc*, *z=0*, *initial_states=None, None*, ***kwargs*)
Initializes a *DataQubit* and saved to self.data_qubits[z][loc].

Parameters initial_states (Tuple[float, float]) – Initial state for the data-qubit.

Return type *DataQubit*

add_pseudo_qubit (*loc*, *z=0*, *state_type='x'*, ***kwargs*)
Initializes a *PseudoQubit* and saved to self.pseudo_qubits[z][loc].

Return type *PseudoQubit*

static entangle_pair(*data_qubit*, *ancilla_qubit*, *key*, *edge=None*, ***kwargs*)
Entangles one *DataQubit* to a *AncillaQubit* for parity measurement.

Parameters

- **data_qubit** (*DataQubit*) – Control qubit.
- **ancilla_qubit** (*AncillaQubit*) – Controlled qubit.
- **key** (Any) – The entanglement is saved by adding the *DataQubit* to *AncillaQubit* `.parity_qubits[key]`
- **edge** (Optional[*Edge*]) – The edge of the data-qubit to entangle to.

init_errors(**error_modules*, *error_rates={}*, ***kwargs*)
Initializes error modules.

Any error module from *Error types* can loaded as either a string equivalent to the module file name or as the module itself. The default error rates for all loaded error modules can be supplied as a dictionary with keywords corresponding to the default error rates of the associated error modules.

Parameters

- **error_modules** (Union[str, *Sim*]) – The error modules to load. May be a string or an error module from *Error types*.
- **error_rates** (dict) – The default error rates for the loaded modules. Must be a dictionary with probabilities with keywords corresponding to the default or overriding error rates of the associated error modules.

Examples

Load *Pauli error* and *Erasure error* modules via string names. Set default bitflip rate to `0.1` and erasure to `0.03`.

```
>>> code.init_errors(
...     "pauli",
...     "erasure",
...     error_rates={"p_bitflip": 0.1, "p_erasure": 0.03}
... )
```

Load Pauli error module via module. Set default phaseflip rate to `0.05`.

```
>>> import .errors.pauli as pauli
>>> code.init_errors(pauli, error_rates={"p_phaseflip": 0.05})
```

init_logical_operator(***kwargs*)
Initiates the logical operators [x,z] of the planar code.

init_parity_check(*ancilla_qubit*, ***kwargs*)
Initiates a parity check measurement.

For every ancilla qubit on `(x,y)`, four neighboring data qubits are entangled for parity check measurements.

Parameters **ancilla_qubit** (*AncillaQubit*) – Ancilla qubit to initialize.

init_surface(*z=0*, ***kwargs*)
Initializes the planar surface code on layer `z`.

Parameters **z** (*int or float, optional*) – Layer of qubits, `z=0` for perfect measurements.

initialize(**args*, ***kwargs*)

Initializes the code with a figure. Also takes keyword arguments for *init_plot*.

Since each error object delivers extra plot properties to the figure, which are dependent on the `self.params` values in the figure itself, we must initialize in the following sequence.

- First load figure to load `self.params` instance of the *PlotParams* dataclass.
- Initialize lattice, error initialization must have figure properties
- Draw figure with plot elements from errors

plot_ancilla(*iter_name=None*, ***kwargs*)

Update plots of all ancilla-qubits. A plot iteration is added if a `iter_name` is supplied. See *draw_figure*.

plot_data(*iter_name=None*, ***kwargs*)

Update plots of all data-qubits. A plot iteration is added if a `iter_name` is supplied. See *draw_figure*.

random_errors(**args*, ***kwargs*)

Applies all errors loaded in `self.errors` attribute to layer `z`.

The random error is applied for each loaded error module by calling *random_error*. If `apply_order` is specified, the error modules are applied in order of the error names in the list. If no order is specified, the errors are applied in a random order. Addionally, any error rate can set by supplying the rate as a keyword argument e.g. `p_bitflip = 0.1`.

Parameters

- **apply_order** – The order in which the error modules are applied. Items in the list must equal keys in `self.errors` or the names of the loaded error modules.
- **measure** – Measure ancilla qubits after errors have been simulated.

show_corrected(***kwargs*)

Redraws the qubits and ancillas to show their states after decoding.

class qsurface.codes.planar.plot.**FaultyMeasurements**(**args*, *figure3d=True*, ***kwargs*)

Plotting code class for faulty measurements.

Inherits from *codes.planar.sim.FaultyMeasurements* and *codes.planar.plot.PerfectMeasurements*. See documentation for these classes for more.

Dependent on the `figure3d` argument, either a 3D figure object is created that inherits from *Template3D* and *codes.planar.plot.PerfectMeasurements.Figure*, or the 2D *codes.planar.plot.PerfectMeasurements.Figure* is used.

Parameters

- **args** – Positional arguments are passed on to *codes.planar.sim.FaultyMeasurements*.
- **figure3d** (*bool*) – Enables plotting on a 3D lattice. Disable to plot layer-by-layer on a 2D lattice, which increases responsiveness.
- **kwargs** – Keyword arguments are passed on to *codes.planar.sim.FaultyMeasurements* and the figure object.

3.6 Template error

class qsurface.errors._template.**Sim**(*code=None*, ***kwargs*)
Template simulation class for errors.

The template simulation error class can be used as a parent class for error modules for surface code classes that inherit from *codes._template.sim.PerfectMeasurements* or *codes._template.sim.FaultyMeasurements*. The error of the module must be applied to each qubit separately using the abstract method *random_error*.

Parameters **code** (*codes._template.sim.PerfectMeasurements*) – Simulation surface code class.

default_error_rates
The error rates that are applied at default.

Type dict of float

abstract random_error(*qubit*, ***kwargs*)
Applies the current error type to the qubit.

Parameters **qubit** (DataQubit) – Qubit on which the error is (conditionally) applied.

Return type None

class qsurface.errors._template.**Plot**(**args*, ***kwargs*)
Template plot class for errors.

The template plotting error class can be used as a parent class for error modules for surface code classes that inherit from *codes._template.plot.PerfectMeasurements* or *codes._template.plot.FaultyMeasurements*, which have a figure object attribute at code.figure. The error of the module must be applied to each qubit separately using the abstract method random_error.

To change properties of the qubit (a matplotlib.patches.Circle object) if an error has been appied to visualize the error. The template plot error class features an easy way to define the plot properties of an error. First of all, each error must be defined in an *error method* that applies the error to the qubit. The template can contain multiple *error methods*, all of which must be called by random_error. For all errors that we wish to plot, we must add the names of the methods to self.error_methods. The plot properties are stored under the same name in self.plot_params.

```
class CustomPlotError(Plot):

    error_methods = ["example_method"]
    plot_params = {
        "example_method": {"edgecolor": "color_edge", "facecolor": (0,0,0,0)}
    }

    def random_error(self, qubit):
        if random.random < 0.5:
            self.error_method(qubit)

    def example_method(self, qubit):
        # apply error
        pass
```

Note that the properties can either be literal or refer to some attribute of the *PlotParams* object stored at self.code.figure.params (see *load_params*). Thus the name for the error methods **must be unique** to any attribute in PlotParams.

Similarly, additional legend items can be added to the surface code plot `self.code.figure`. Each legend item is a `matplotlib.lines.line2D`. The properties for each additional item in the legend is stored at `self.legend_params`, and must also be unique to any *PlotParams* attribute. The legend titles for each item is stored in `self.legend_titles` at the same keys. The additional legend items are added in `init_legend`.

```
class CustomPlotError(Plot):

    error_methods = ["example_method"]
    plot_params = {
        "example_method": {"edgecolor": "color_edge", "facecolor": (0,0,0,0)}
    }
    legend_params = {
        "example_item": {
            "marker": "o",
            "color": "color_edge",
            "mfc": (1, 0, 0),
            "mec": "g",
        },
    }
    legend_titles = {
        "example_item": "Example error"
    }

    def random_error(self, qubit):
        if random.random < 0.5:
            self.error_method(qubit)

    def example_method(self, qubit):
        # apply error
        pass
```

Finally, error methods can be also be added to the GUI of the surface code plot. For this, each error method must a *static method* that is not dependant on the error class. Each error method to be added in the GUI must be included in `self.gui_methods`. The GUI elements are included in `init_plot`.

```
class CustomPlotError(Plot):

    error_methods = ["example_method"]
    gui_methods = ["example_method"]
    plot_params = {
        "example_method": {"edgecolor": "color_edge", "facecolor": (0,0,0,0)}
    }
    legend_params = {
        "example_item": {
            "marker": "o",
            "color": "color_edge",
            "mfc": (1, 0, 0),
            "mec": "g",
        },
    }
    legend_titles = {
        "example_item": "Example error"
    }

    def random_error(self, qubit):
        if random.random < 0.5:
```

(continues on next page)

(continued from previous page)

```
            self.error_method(qubit)

    @staticmethod
    def example_method(qubit):
        # apply error
        pass
```

Parameters **code** (*PerfectMeasurements*) – Plotting surface code class.

error_methods
List of names of the error methods that changes the qubit surface code plot according to properties defined in `self.plot_params`.

Type list

plot_params
Qubit plot properties to apply for each of the error methods in `self.error_methods`. Properties are loaded to the *PlotParams* object stored at the `self.code.figure.params` attribute of the surface code plot (see *load_params*).

Type {method_name: properties}

legend_params {method_name
Legend items to add to the surface code plot. Properties are loaded to the *PlotParams* object stored at the `self.code.figure.params` attribute of the surface code plot (see *load_params*), and used to initialize a `Line2D` legend item.

Type Line2D properties}

legend_titles
Titles to display for the legend items in `self.legend_params`.

Type {method_name: legend_title}

gui_permanent
If enabled, the application of an error method on a qubit cannot be reversed within the same simulation instance.

Type bool

gui_methods
List of names of the static error methods include in the surface plot GUI.

Type list

plot_error(*error_name*)
Decorates the error method with plotting features.

The method `error_name` is decorated with plot property changes defined in `self.plot_params`. For each of the properties to change, the original property value of the artist is stored and requested as a change at the end of the simulation instance.

See also:

`None()`, `None()`

3.7 Error types

All error modules in this section inherit from the template error module, see *Template error*.

3.7.1 Pauli error

class qsurface.errors.pauli.**Plot**(**args*, ***kwargs*)

Plot Pauli error class.

class qsurface.errors.pauli.**Sim**(**args*, *p_bitflip=0*, *p_phaseflip=0*, ***kwargs*)

Simulation Pauli error class.

Parameters

- **p_bitflip** (*float or int, optional*) – Default probability of X-errors or bitflip errors.
- **p_phaseflip** (*float or int, optional*) – Default probability of Z-errors or phaseflip errors.

static bitflip(*qubit*, ***kwargs*)

Applies a bitflip or Pauli X on `qubit`.

static bitphaseflip(*qubit*, ***kwargs*)

Applies a bitflip and phaseflip or ZX on `qubit`.

static phaseflip(*qubit*, ***kwargs*)

Applies a phaseflip or Pauli Z on `qubit`.

random_error(*qubit*, *p_bitflip=0*, *p_phaseflip=0*, ***kwargs*)

Applies a Pauli error, bitflip and/or phaseflip.

Parameters

- **qubit** (*Qubit*) – Qubit on which the error is (conditionally) applied.
- **p_bitflip** (`float`) – Overriding probability of X-errors or bitflip errors.
- **p_phaseflip** (`float`) – Overriding probability of Z-errors or phaseflip errors.

3.7.2 Erasure error

class qsurface.errors.erasure.**Plot**(**args*, ***kwargs*)

Plot erasure error class.

class qsurface.errors.erasure.**Sim**(**args*, *p_erasure=0*, *initial_states=0, 0*, ***kwargs*)

Simulation erasure error class.

Parameters

- **p_erasure** (`float`) – Default probability of erasure errors.
- **initial_states** (`Tuple[float, float]`) – Default state of the qubit after re-initialization.

static erasure(*qubit*, *instance=0*, *initial_states=0, 0*, ***kwargs*)

Erases the `qubit` by resetting its attributes.

Parameters

- **qubit** (*DataQubit*) – Qubit to erase.

- **instance** (float) – Current simulation instance.
- **initial_states** (Tuple[float, float]) – State of the qubit after re-initialization.

random_error (*qubit*, *p_erasure=0*, *initial_states=None*, ***kwargs*)

Applies an erasure error.

Parameters

- **qubit** – Qubit on which the error is (conditionally) applied.
- **p_erasure** (float) – Overriding probability of erasure errors.
- **initial_states** (Optional[Tuple[float, float]]) – Overriding state of the qubit after re-initialization.

3.8 Template decoder

qsurface.decoders._template.**write_config** (*config_dict*, *path*)

Writes a configuration file to the path.

Parameters

- **config_dict** (*dict*) – Dictionary of configuration parameters. Can be nested.
- **path** (*str*) – Path to the file. Must include the desired extension.

qsurface.decoders._template.**read_config** (*path*, *config_dict=None*)

Reads an INI formatted configuration file and parses it to a nested dict

Each category in the INI file will be parsed as a separate nested dictionary. A default `config_dict` can be provided with default values for the parameters. Parameters under the "main" section will be parsed in the main dictionary. All data types will be converted by `ast.literal_eval()`.

Parameters

- **path** (*str*) – Path to the file. Must include the desired extension.
- **config_dict** (*dict, optional*) – Nested dictionary of default parameters

Returns Parsed dictionary.

Return type dict

Examples

Let us look at the following example INI file.

```
[main]
param1 = hello

[section]
param2 = world
param3 = 0.1
```

This file will be parsed as follows

```
>>> read_config("config.ini")
{
    "param1": "hello",
    "section": {
        "param2": "world",
        "param3": 0.1
    }
}
```

qsurface.decoders._template.**init_config**(*ini_file*, *write=False*, ***kwargs*)
Reads the default and the user defined INI file.

First, the INI file stored in file directory is read and parsed. If there exists another INI file in the working directory, the attributes defined there are read, parsed and overwrites and default values.

Parameters **write** (bool) – Writes the default configuration to the working direction of the user.

See also:

write_config(), *read_config()*

class qsurface.decoders._template.**Sim**(*code*, *check_compatibility=False*, ***kwargs*)
Decoder simulation class template.

Parameters

- **code** (*PerfectMeasurements*) – A PerfectMeasurements or FaultyMeasurements class from the sim module of *Code types*.
- **check_compatibility** (bool) – Checks compatibility of the decoder with the code class and loaded errors by *check_compatibility*.

compatibility_measurements
Compatibility with perfect or faulty measurements.

Type dict

compatibility_errors
Compatibility with the various error modules in *Error types*.

Type dict

check_compatibility()
Checks compatibility of the decoder with the code class and loaded errors.

static get_neighbor(*ancilla_qubit*, *key*)
Returns the neighboring ancilla-qubit of ancilla_qubit in the direction of key.

Return type Tuple[*AncillaQubit*, *Edge*]

get_neighbors(*ancilla_qubit*, *loop=False*, ***kwargs*)
Returns all neighboring ancillas, including other time instances.

Parameters **loop** (bool) – Include neighbors in time that are not chronologically next to each other during decoding within the same instance.

correct_edge(*ancilla_qubit*, *key*, ***kwargs*)
Applies a correction.

The correction is applied to the data-qubit located at ancilla_qubit.parity_qubits[key]. More specifically, the correction is applied to the *Edge* object corresponding to the state_type of ancilla_qubit.

Return type *AncillaQubit*

get_syndrome (*find_pseudo=False*)
Finds the syndrome of the code.

Parameters **find_pseudo** (*bool, optional*) – If enabled, the lists of syndromes returned are not only *AncillaQubit* objects, but tuples of (ancilla, pseudo), where pseudo is the closest *PseudoQubit* in the boundary of the code.

Return type Union[Tuple[List[*AncillaQubit*], List[*AncillaQubit*]], Tuple[List[Tuple[*AncillaQubit*, *PseudoQubit*]], List[Tuple[*AncillaQubit*, *PseudoQubit*]]]]

Returns

- *list* – Plaquette operator syndromes.
- *list* – Star operator syndromes.

abstract decode (**args*, ***kwargs*)
Decodes the surface loaded at self.code after all ancilla-qubits have been measured.

class qsurface.decoders._template.**Plot** (**args*, ***kwargs*)
Decoder plotting class template.

The plotting decoder class requires a surface code object that inherits from *codes._template.plot.PerfectMeasurements*. The template decoder provides the *plot_matching_edge* method that is called by *correct_edge* to visualize the matched edges on the lattice.

Parameters

- **args** – Positional and keyword arguments are passed on to *Sim*.
- **kwargs** – Positional and keyword arguments are passed on to *Sim*.

line_color_match
Plot properties for matched edges.

Type dict

line_color_normal
Plot properties for normal edges.

Type dict

matching_lines
Dictionary of edges that have been added to the matching.

Type defaultdict(bool)

decode (**args*, ***kwargs*)
Decodes the surface loaded at self.code after all ancilla-qubits have been measured.

correct_edge (*qubit*, *key*, ***kwargs*)
Applies a correction.

The correction is applied to the data-qubit located at ancilla_qubit.parity_qubits[key]. More specifically, the correction is applied to the *Edge* object corresponding to the state_type of ancilla_qubit.

plot_matching_edge (*line=None*)
Plots the matching edge.

Based on the colors defined in self.line_color_match, if a Line2D object is supplied, the color of the edge is changed. A future change back to its original color is immediately saved in figure.future_dict.

check_compatibility()
Checks compatibility of the decoder with the code class and loaded errors.

static **get_neighbor**(*ancilla_qubit*, *key*)
Returns the neighboring ancilla-qubit of `ancilla_qubit` in the direction of `key`.

Return type `Tuple`[*`AncillaQubit`*, *`Edge`*]

get_neighbors(*ancilla_qubit*, *loop=False*, ***kwargs*)
Returns all neighboring ancillas, including other time instances.

Parameters **loop** (`bool`) – Include neighbors in time that are not chronologically next to each other during decoding within the same instance.

get_syndrome(*find_pseudo=False*)
Finds the syndrome of the code.

Parameters **find_pseudo** (*`bool`, `optional`*) – If enabled, the lists of syndromes returned are not only *`AncillaQubit`* objects, but tuples of `(ancilla, pseudo)`, where `pseudo` is the closest *`PseudoQubit`* in the boundary of the code.

Return type `Union`[`Tuple`[`List`[*`AncillaQubit`*], `List`[*`AncillaQubit`*]], `Tuple`[`List`[`Tuple`[*`AncillaQubit`*, *`PseudoQubit`*]], `List`[`Tuple`[*`AncillaQubit`*, *`PseudoQubit`*]]]]

Returns

- *list* – Plaquette operator syndromes.
- *list* – Star operator syndromes.

3.9 Decoders

All decoder modules in this section inherit from the template decoder module, see *Template decoder*.

3.9.1 mwpm

The Minimum-Weight Perfect Matching decoder.

Information

The most popular decoder for surface codes is the Minimum-Weight Perfect Matching (MWPM) decoder. It performs near-optimal for a pauli noise model [dennis2002] on a standard toric code with a threshold of $p_{\text{th}} = 10.3\%$, and for a phenomenological noise model (including faulty measurements) [wang2003], which includes faulty measurements, with $p_{\text{th}} = 2.9\%$. The main idea is to approximate the error with the minimum-weight error configuration compatible with the syndrome. The minimum-weight configuration is found by constructing a fully connected graph between the nodes of the syndrome, which leads to a cubic worst-case time complexity [kolmogorov2009].

The decoder defaults to using a Python implementation of MWPM by `networkx.algorithms.matching.max_weight_matching`. This implementation is however quite slow. Optionally, Blossom V [kolmogorov2009], a C++ algorithm, can be used to increase the speed of the decoder. Since this software has its own license, it is not bundeled with qsurface. A script is provided to download and compile the latest release of BlossomV in *`get_blossomv`*. The interface of the C++ code and Python is taken from Fault Tolerant Simulations.

`qsurface.decoders.mwpm.`**get_blossomv**(*accept=False*)
Downloads and compiles the BlossomV algorithm, which is distributed under the following license:

License:

Simulation

class `qsurface.decoders.mwpm.sim.`**`Toric`**(*code*, *check_compatibility=False*, ***kwargs*)

Minimum-Weight Perfect Matching decoder for the toric lattice.

Parameters

- **`args`** – Positional and keyword arguments are passed on to *`decoders._template.Sim`*.
- **`kwargs`** – Positional and keyword arguments are passed on to *`decoders._template.Sim`*.

`decode`(***kwargs*)

Decodes the surface loaded at `self.code` after all ancilla-qubits have been measured.

`match_syndromes`(*syndromes*, *use_blossomv=False*, ***kwargs*)

Decodes a list of syndromes of the same type.

A graph is constructed with the syndromes in `syndromes` as nodes and the distances between each of the syndromes as the edges. The distances are dependent on the boundary conditions of the code and is calculated by *`get_qubit_distances`*. A minimum-weight matching is then found by either *`match_networkx`* or *`match_blossomv`*.

Parameters

- **`syndromes`** (`List`[*`AncillaQubit`*]) – Syndromes of the code.

- **use_blossomv** (bool) – Use external C++ Blossom V library for minimum-weight matching. Needs to be downloaded and compiled by calling *get_blossomv*.

Returns Minimum-weight matched ancilla-qubits.

Return type list of *AncillaQubit*

correct_matching (*syndromes*, *matching*, ***kwargs*)
Applies the matchings as a correction to the code.

static match_networkx (*edges*, *maxcardinality*, ***kwargs*)
Finds the minimum-weight matching of a list of `edges` using `networkx.algorithms.matching.max_weight_matching`.

Parameters

- **edges** (*[[nodeA, nodeB, distance(nodeA,nodeB)],..]*) – A graph defined by a list of edges.
- **maxcardinality** (float) – See `networkx.algorithms.matching.max_weight_matching`.

Returns Minimum weight matching in the form of [[nodeA, nodeB],..].

Return type list

static match_blossomv (*edges*, *num_nodes=0*, ***kwargs*)
Finds the minimum-weight matching of a list of `edges` using Blossom V.

Parameters edges (*[[nodeA, nodeB, distance(nodeA,nodeB)],..]*) – A graph defined by a list of edges.

Returns Minimum weight matching in the form of [[nodeA, nodeB],..].

Return type list

static get_qubit_distances (*qubits*, *size*)
Computes the distance between a list of qubits.

On a toric lattice, the shortest distance between two qubits may be one in four directions due to the periodic boundary conditions. The `size` parameters indicates the length in both x and y directions to find the shortest distance in all directions.

check_compatibility ()
Checks compatibility of the decoder with the code class and loaded errors.

correct_edge (*ancilla_qubit*, *key*, ***kwargs*)
Applies a correction.

The correction is applied to the data-qubit located at `ancilla_qubit.parity_qubits[key]`. More specifically, the correction is applied to the *Edge* object corresponding to the `state_type` of `ancilla_qubit`.

Return type *AncillaQubit*

static get_neighbor (*ancilla_qubit*, *key*)
Returns the neighboring ancilla-qubit of `ancilla_qubit` in the direction of `key`.

Return type Tuple[*AncillaQubit*, *Edge*]

get_neighbors (*ancilla_qubit*, *loop=False*, ***kwargs*)
Returns all neighboring ancillas, including other time instances.

Parameters loop (bool) – Include neighbors in time that are not chronologically next to each other during decoding within the same instance.

get_syndrome (*find_pseudo=False*)
Finds the syndrome of the code.

Parameters find_pseudo (`bool, optional`) – If enabled, the lists of syndromes returned are not only `AncillaQubit` objects, but tuples of `(ancilla, pseudo)`, where `pseudo` is the closest `PseudoQubit` in the boundary of the code.

Return type `Union`[`Tuple`[`List`[`AncillaQubit`], `List`[`AncillaQubit`]], `Tuple`[`List`[`Tuple`[`AncillaQubit`, `PseudoQubit`]], `List`[`Tuple`[`AncillaQubit`, `PseudoQubit`]]]]

Returns

- *list* – Plaquette operator syndromes.
- *list* – Star operator syndromes.

class `qsurface.decoders.mwpm.sim.`**Planar** (*code*, *check_compatibility=False*, ***kwargs*)
Minimum-Weight Perfect Matching decoder for the planar lattice.

Additionally to all edges, virtual qubits are added to the boundary, which connect to their main qubits.Edges between all virtual qubits are added with weight zero.

decode (***kwargs*)
Decodes the surface loaded at `self.code` after all ancilla-qubits have been measured.

correct_matching (*syndromes*, *matching*)
Applies the matchings as a correction to the code.

static get_qubit_distances (*qubits*, **args*)
Computes the distance between a list of qubits.

On a planar lattice, any qubit can be paired with the boundary, which is inhabited by `PseudoQubit` objects. The graph of syndromes that supports minimum-weight matching algorithms must be fully connected, with each syndrome connecting additionally to its boundary pseudo-qubit, and a fully connected graph between all pseudo-qubits with weight 0.

check_compatibility ()
Checks compatibility of the decoder with the code class and loaded errors.

correct_edge (*ancilla_qubit*, *key*, ***kwargs*)
Applies a correction.

The correction is applied to the data-qubit located at `ancilla_qubit.parity_qubits[key]`. More specifically, the correction is applied to the `Edge` object corresponding to the `state_type` of `ancilla_qubit`.

Return type `AncillaQubit`

static get_neighbor (*ancilla_qubit*, *key*)
Returns the neighboring ancilla-qubit of `ancilla_qubit` in the direction of `key`.

Return type `Tuple`[`AncillaQubit`, `Edge`]

get_neighbors (*ancilla_qubit*, *loop=False*, ***kwargs*)
Returns all neighboring ancillas, including other time instances.

Parameters loop (`bool`) – Include neighbors in time that are not chronologically next to each other during decoding within the same instance.

get_syndrome (*find_pseudo=False*)
Finds the syndrome of the code.

Parameters **find_pseudo** (*bool, optional*) – If enabled, the lists of syndromes returned are not only *AncillaQubit* objects, but tuples of (ancilla, pseudo), where pseudo is the closest *PseudoQubit* in the boundary of the code.

Return type Union[Tuple[List[*AncillaQubit*], List[*AncillaQubit*]], Tuple[List[Tuple[*AncillaQubit*, *PseudoQubit*]], List[Tuple[*AncillaQubit*, *PseudoQubit*]]]]

Returns

- *list* – Plaquette operator syndromes.
- *list* – Star operator syndromes.

static match_blossomv(*edges*, *num_nodes=0*, ***kwargs*)

Finds the minimum-weight matching of a list of edges using Blossom V.

Parameters **edges** (*[[nodeA, nodeB, distance(nodeA,nodeB)],..]*) – A graph defined by a list of edges.

Returns Minimum weight matching in the form of [[nodeA, nodeB],..].

Return type list

static match_networkx(*edges*, *maxcardinality*, ***kwargs*)

Finds the minimum-weight matching of a list of edges using networkx.algorithms.matching.max_weight_matching.

Parameters

- **edges** (*[[nodeA, nodeB, distance(nodeA,nodeB)],..]*) – A graph defined by a list of edges.
- **maxcardinality** (float) – See networkx.algorithms.matching.max_weight_matching.

Returns Minimum weight matching in the form of [[nodeA, nodeB],..].

Return type list

match_syndromes(*syndromes*, *use_blossomv=False*, ***kwargs*)

Decodes a list of syndromes of the same type.

A graph is constructed with the syndromes in syndromes as nodes and the distances between each of the syndromes as the edges. The distances are dependent on the boundary conditions of the code and is calculated by *get_qubit_distances*. A minimum-weight matching is then found by either *match_networkx* or *match_blossomv*.

Parameters

- **syndromes** (List[*AncillaQubit*]) – Syndromes of the code.
- **use_blossomv** (bool) – Use external C++ Blossom V library for minimum-weight matching. Needs to be downloaded and compiled by calling *get_blossomv*.

Returns Minimum-weight matched ancilla-qubits.

Return type list of *AncillaQubit*

Plotting

class qsurface.decoders.mwpm.plot.**Toric**(**args*, ***kwargs*)

Plot MWPM decoder for the toric code.

Parameters

- **args** – Positional and keyword arguments are passed on to *decoders._template.Plot* and *decoders.mwpm.sim.Toric*.
- **kwargs** – Positional and keyword arguments are passed on to *decoders._template.Plot* and *decoders.mwpm.sim.Toric*.

class qsurface.decoders.mwpm.plot.**Planar**(**args*, ***kwargs*)

Plot MWPM decoder for the planar code.

Parameters

- **args** – Positional and keyword arguments are passed on to *Toric* and *decoders.mwpm.sim.Planar*.
- **kwargs** – Positional and keyword arguments are passed on to *Toric* and *decoders.mwpm.sim.Planar*.

3.9.2 unionfind

The Union-Find decoder.

Information

The Union-Find decoder [delfosse2017almost] maps each element of the syndrome σ to an ancilla v in a non-connected graph defined on the code lattice. From this starting point, it grows clusters around these ancillas by repeatedly adding a layer of edges and ancillas to existing clusters, until all clusters have an even number of non-trivial syndrome ancillas. Then, it selects a spanning tree F for each cluster.

The leaves of each spanning tree are conditionally peeled in a tail-recursive breadth-first search until all non-trivial syndrome ancillas are paired and linked by a path within F, which is the correcting operator $\mathcal{C}$ [delfosse2017linear]. The strategy for constructing the clusters turns out to have a strong effect on performance. For instance, the threshold for bitflip noise of a decoder that grows the clusters following a random order is 9.2% [delfosse2017almost], while if the clusters are grown in order of cluster size, which we call **Weighted Growth**, the threshold increases to 9.9% [delfosse2017almost].

The complexity of the Union-Find decoder is driven by the merging of the clusters. For this, the algorithm uses the Union-Find or disjoint-set data structure [tarjan1975efficiency]. This data structure contains a set of elements, in this case ancillas on the lattice. The set of elements is represented by a two-level tree. At the root of the tree sits one element chosen arbitrarily; the rest of the elements are linked to the root element. The structure admits two functions: $Find$ and $Union$. Given v an element from the structure, the function $Find(v)$ returns the root element of the tree. This is is used to identify the cluster to which v belongs. The second function is $Union(u, v)$, this function merges the sets associated with elements u and v. This requires pointing all the elements of one of the sets to the root of the other. In order to minimize the number of operations the root of the set with the larger number of elements is chosen as root for the merged set, this is called **Weighted Union**. In this context, $Union$ is used when the growth of a cluster requires adding a vertex that belongs to another.

class qsurface.decoders.unionfind.elements.**Cluster**(*index*, *instance*, ***kwargs*)

CLuster of *AncillaQubit* objects.

A disjoint set, or cluster, of ancilla-qubits. The size of the cluster is equal to the number of qubits in the cluster. The parity of the cluster is equal to the number of non-trivial ancilla-qubits in the cluster.

A cluster can be joined with another by *union*. Joined clusters are stored in the union-find data structure [tarjan1975efficiency]. The representative element or root cluster is returned by *find*.

Parameters

- **index** (int) – Indicator index number.
- **instance** (float) – The epoch timestamp of the simulation.

size
Size of this cluster based on the number contained ancillas.

Type int

support
Growth state of the cluster.

Type int

parity
Parity of this cluster based on the number non-trivial ancilla-qubits.

Type int

parent
The parent cluster of the current cluster.

Type *Cluster*

bound, new_bound
The current and next boundary of the current cluster.

Type list, `[[inner_ancilla, edge, outer_ancilla],...]`

bucket
The bucket number the current ancilla belongs to.

Type int

on_bound
Whether this cluster is connected to the boundary.

Type bool

add_ancilla(*ancilla*)
Adds an ancilla to a cluster.

find(***kwargs*)
Finds the representative root cluster.

The function is applied recursively until the root element of the union-find tree is encountered. The representative root element is returned. Path compression is applied to reduce the depth of the tree.

Examples

For joined clusters in the union-find data structure:

```
    cl0
    / \
  cl1 cl2
  /
cl2
```

the representative element can be found by

```
>>> cl2.find()
cl0
```

Return type *Cluster*

union(*cluster*, ***kwargs*)
Merges two clusters.

The `cluster` is made a child of the current cluster. The joined size and parity attributes are updated.

Parameters **cluster** (*Cluster*) – The cluster to merge with `self`.

Examples

For two clusters `cl0` and `cl1`, `cl0.union(cl1)` results in the following tree:

```
  cl0
  /
cl1
```

Simulation

The following description also applies to *unionfind.sim.Planar*.

class qsurface.decoders.unionfind.sim.**Toric**(**args*, ***kwargs*)
Union-Find decoder for the toric lattice.

In this implementation, cluster properties are not stored at the root of the tree. Instead, ancillas are collected within *Cluster* objects, which contain the *union* and *find* methods.

Default values for the following parameters can be supplied via a *decoders.ini* file under the section of `[unionfind]`.

The `cluster` and `peeled` attributes are monkey patched to the *AncillaQubit* object to assist the identification of its parent cluster and to assist peeling. The `forest` attribute is monkey-patched to `AncillaQubit` and `Edge` if a dynamic forest is not maintained to assist with the construction of the acyclic forest after cluster growth.

Parameters

- **weighted_growth** (*bool, optional*) – Enables weighted growth via bucket growth. Default is true. See *grow_clusters*.
- **weighted_union** (*bool, optional*) – Enables weighted union, Default is true. See *union_bucket*.
- **dynamic_forest** (*bool, optional*) – Enables dynamically mainted forests. Default is true.
- **print_steps** (*bool, optional*) – Prints additional decoding information. Default is false.
- **kwargs** – Keyword arguments are forwarded to *Sim*.

support
Dictionary of growth states of all edges in the code.

value	state
2	fully grown
1	half grown
0	none
-1	removed by cycle or peel
-2	added to matching

Type dict

buckets
Ordered dictionary (by index) for bucket growth (implementation of weighted growth). See *grow_clusters*.

Type defaultdict

bucket_max_filled
The hightest occupied bucket. Allows for break from bucket loop.

Type int

clusters
List of all clusters at initialization.

Type list

cluster_index
Index value for cluster differentiation.

Type int

decode (***kwargs*)
Decodes the code using the Union-Find algorithm.

Decoding process can be subdivided into 3 sections:

1. Finding the initial clusters.
2. Growing and merging these clusters.
3. Peeling the clusters using the Peeling algorithm.

Parameters kwargs – Keyword arguments are passed on to *find_clusters*, *grow_clusters* and *peel_clusters*.

get_cluster (*ancilla*)
Returns the cluster to which `ancilla` belongs to.

If `ancilla` has no cluster or the cluster is not from the current simulation, none is returned. Otherwise, the root element of the cluster-tree is found, updated to `ancilla.cluster` and returned.

Parameters ancilla (*AncillaQubit*) – The ancilla for which the cluster is to be found.

Return type Optional[*Cluster*]

cluster_add_ancilla (*cluster, ancilla, parent=None, **kwargs*)
Recursively adds erased edges to `cluster` and finds the new boundary.

For a given `ancilla`, this function finds the neighboring edges and ancillas that are in the the currunt cluster. If the newly found edge is erased, the edge and the corresponding ancilla will be added to the cluster, and the function applied recursively on the new ancilla. Otherwise, the neighbor is added to the new boundary `self.new_bound`.

Parameters

- **cluster** (*Cluster*) – Current active cluster
- **ancilla** (*AncillaQubit*) – Ancilla from which the connected erased edges or boundary are searched.

find_clusters (***kwargs*)

Initializes the clusters on the lattice.

For every non-trivial ancilla on the lattice, a *Cluster* is initiated. If any set of ancillas are connected by some set of erased qubits, all connected ancillas are found by *cluster_add_ancilla* and a single cluster is initiated for the set.

The cluster is then placed into a bucket based on its size and parity by *place_bucket*. See *grow_clusters* for more information on buckets.

grow_clusters (***kwargs*)

Grows odd-parity clusters outward for union with others until all clusters are even.

Lists of odd-parity clusters are maintained at `self.buckets`. Starting from bucket 0, odd-parity clusters are popped from the bucket by 'grow_bucket and grown at the boundary by *grow_boundary* by adding 1 for every boundary edge in `cluster.bound` in `self.support`. Grown clusters are then placed in a new bucket by *place_bucket* based on its size if it has odd parity.

Edges are fully added to the cluster per two growth iterations. Since a cluster with half-grown edges at the boundary has the same size (number of ancillas) as before growth, but is non-arguably *bigger*, the degeneracy in cluster size is differentiated by `cluster.support`. When an union occurs between two clusters during growth, if the merged cluster is odd, it is placed in a new bucket. Thus the real bucket number is saved at the cluster locally as `cluster.bucket`. These two checks are performed before a cluster is grown in *grow_bucket*.

The chronology of events per bucket must be the following:

1. Grow all clusters in the bucket if checks passed.
   - Add all odd-parity clusters after growth to `place_list`.
   - Add all merging clusters to `union_list`.
2. Merge all clusters in `union_list`
   - Add odd-parity clusters after union to `place_list`.
3. Place all clusters in `place_list` in new bucket if parity is odd.

For clusters with `cluster.support==1` or with half-grown edges at the boundary, the new boundary at `clusters.new_bound` consists of the same half-grown edges. For clusters with `cluster.support==0`, the new boundary is found by `cluster_add_ancilla`.

If *weighted_growth* is disabled, odd-parity clusters are always placed in `self.buckets[0]`. The same checks for `cluster.bucket` and `cluster.support` are applied to ensure clusters growth is valid.

grow_bucket (*bucket, bucket_i, **kwargs*)

Grows the clusters which are contained in the current bucket.

See *grow_clusters* for more information.

Parameters

- **bucket** (`List`[*Cluster*]) – List of clusters to be grown.
- **bucket_i** (`int`) – Current bucket number.

Return type `Tuple`[`List`, `List`]

Returns

- *list* – List of potential mergers between two cluster-distinct ancillas.
- *list* – List of odd-parity clusters to be placed in new buckets.

grow_boundary (*cluster*, *union_list*, ***kwargs*)

Grows the boundary of the `cluster`.

See *grow_clusters* for more information.

Parameters

- **cluster** (*Cluster*) – The cluster to be grown.
- **union_list** (List[Tuple[*AncillaQubit*, *Edge*, *AncillaQubit*]]) – List of potential mergers between two cluster-distinct ancillas.

union_bucket (*union_list*, ***kwargs*)

Merges clusters in `union_list` if checks are passed.

Items in `union_list` consists of `[ancillaA, edge, ancillaB]` of two ancillas that, at the time added to the list, were not part of the same cluster. The cluster of an ancilla is stored at `ancilla.cluster`, but due to cluster mergers the cluster at `ancilla_cluster` may not be the root element in the cluster-tree, and thus the cluster must be requested by `ancilla.cluster.` *find*. Since the clusters of `ancillaA` and `ancillaB` may have already merged, checks are performed in *union_check* after which the clusters are conditionally merged on `edge` by `union_edge`.

If `weighted_union` is enabled, the smaller cluster is always made a child of the bigger cluster in the cluster-tree. This ensures the that the depth of the tree is minimized and the future calls to *find* is reduced.

If `dynamic_forest` is disabled, cycles within clusters are not immediately removed. The acyclic forest is then later constructed before peeling in *peel_leaf*.

Parameters **union_list** (List[Tuple[*AncillaQubit*, *Edge*, *AncillaQubit*]]) – List of potential mergers between two cluster-distinct ancillas.

union_check (*edge*, *ancilla*, *new_ancilla*, *cluster*, *new_cluster*)

Checks whether `cluster` and `new_cluster` can be joined on `edge`.

See *union_bucket* for more information.

Return type bool

place_bucket (*clusters*, *bucket_i*)

Places all clusters in `clusters` in a bucket if parity is odd.

If `weighted_growth` is enabled. the cluster is placed in a new bucket based on its size, otherwise it is placed in `self.buckets[0]`

Parameters

- **clusters** (List[*Cluster*]) – Clusters to place in buckets.
- **bucket_i** (int) – Current bucket number.

peel_clusters (***kwargs*)

Loops over all clusters to find pendant ancillas to peel.

To make sure that all cluster-trees are fully peeled, all ancillas are considered in the loop. If the ancilla has not been peeled before and belongs to a cluster of the current simulation, the ancilla is considered for peeling by *peel_leaf*.

peel_leaf (*cluster*, *ancilla*)

Recursive function which peels a branch of the tree if the input ancilla is a pendant ancilla

If there is only one neighbor of the input ancilla that is in the same cluster, this ancilla is a pendant ancilla and can be peeled. The function calls itself on the other ancilla of the edge leaf.

If ["dynamic_forest"] is disabled, once a pendant leaf is found, the acyclic forest is constructed by `static_forest`.

Parameters

- **`cluster`** – Current cluster being peeled.
- **`ancilla`** – Pendant ancilla of the edge to be peeled.

`flip_edge`(*ancilla*, *edge*, *new_ancilla*, ***kwargs*)
Flips the values of the ancillas connected to `edge`.

`static_forest`(*ancilla*)
Constructs an acyclic forest in the cluster of `ancilla`.

Applies recursively to all neighbors of `ancilla`. If a cycle is detected, edges are removed from the cluster.

Parameters **`ancilla`** (`AncillaQubit`) –

`check_compatibility`()
Checks compatibility of the decoder with the code class and loaded errors.

`correct_edge`(*ancilla_qubit*, *key*, ***kwargs*)
Applies a correction.

The correction is applied to the data-qubit located at `ancilla_qubit.parity_qubits[key]`. More specifically, the correction is applied to the `Edge` object corresponding to the `state_type` of `ancilla_qubit`.

Return type `AncillaQubit`

`static get_neighbor`(*ancilla_qubit*, *key*)
Returns the neighboring ancilla-qubit of `ancilla_qubit` in the direction of `key`.

Return type `Tuple`[`AncillaQubit`, `Edge`]

`get_neighbors`(*ancilla_qubit*, *loop=False*, ***kwargs*)
Returns all neighboring ancillas, including other time instances.

Parameters **`loop`** (`bool`) – Include neighbors in time that are not chronologically next to each other during decoding within the same instance.

`get_syndrome`(*find_pseudo=False*)
Finds the syndrome of the code.

Parameters **`find_pseudo`** (`bool, optional`) – If enabled, the lists of syndromes returned are not only `AncillaQubit` objects, but tuples of `(ancilla, pseudo)`, where `pseudo` is the closest `PseudoQubit` in the boundary of the code.

Return type `Union`[`Tuple`[`List`[`AncillaQubit`], `List`[`AncillaQubit`]], `Tuple`[`List`[`Tuple`[`AncillaQubit`, `PseudoQubit`]], `List`[`Tuple`[`AncillaQubit`, `PseudoQubit`]]]]

Returns

- *list* – Plaquette operator syndromes.
- *list* – Star operator syndromes.

`class` `qsurface.decoders.unionfind.sim.`**`Planar`**(**args*, ***kwargs*)
Union-Find decoder for the planar lattice.

See the description of `unionfind.sim.Toric`.

Plotting

class qsurface.decoders.unionfind.plot.**Toric**(**args*, ***kwargs*)

Union-Find decoder for the toric lattice with union-find plot.

Has all class attributes and methods from `unionfind.sim.Toric`, with additional parameters below. Default values for these parameters can be supplied via a *decoders.ini* file under the section of `[unionfind]` (see `decoders._template.read_config`).

The plotting class initiates a `qsurface.plot` object. For its usage, see *Usage*.

Parameters

- **step_bucket** (*bool, optional*) – Waits for user after every occupied bucket. Default is false.
- **step_cluster** (*bool, optional*) – Waits for user after growth of every cluster. Default is false.
- **step_cycle** (*bool, optional*) – Waits for user after every edge removed due to cycle detection. Default is false.
- **step_peel** (*bool, optional*) – Waits for user after every edge removed during peeling. Default is false.

class Figure2D(*decoder*, *name*, **args*, ***kwargs*)

Visualizer for the Union-Find decoder and Union-Find based decoders with perfect measurements.

Parameters

- **args** – Positional and keyword arguments are forwarded to `plot.Template2D`.
- **kwargs** – Positional and keyword arguments are forwarded to `plot.Template2D`.

class Figure3D(**args*, ***kwargs*)

Visualizer for the Union-Find decoder and Union-Find based decoders with faulty measurements.

Parameters

- **args** – Positional and keyword arguments are forwarded to `Figure2D` and `plot.Template3D`.
- **kwargs** – Positional and keyword arguments are forwarded to `Figure2D` and `plot.Template3D`.

class qsurface.decoders.unionfind.plot.**Planar**(**args*, ***kwargs*)

Union-Find decoder for the planar lattice with union-find plot.

Has all class attributes and methods from `unionfind.sim.Planar`, with additional parameters below. Default values for these parameters can be supplied via a *decoders.ini* file under the section of `[unionfind]` (see `decoders._template.read_config`).

The plotting class initiates a `qsurface.plot` object. For its usage, see *Usage*.

Parameters

- **step_bucket** (*bool, optional*) – Waits for user after every occupied bucket. Default is false.
- **step_cluster** (*bool, optional*) – Waits for user after growth of every cluster. Default is false.

- **step_cycle** (*bool, optional*) – Waits for user after every edge removed due to cycle detection. Default is false.
- **step_peel** (*bool, optional*) – Waits for user after every edge removed during peeling. Default is false.
- **kwargs** – Keyword arguments are passed on to *unionfind.sim.Planar*.

class Figure2D (*decoder, name, *args, **kwargs*)
Visualizer for the Union-Find decoder and Union-Find based decoders with perfect measurements.

Parameters

- **args** – Positional and keyword arguments are forwarded to *unionfind.plot.Toric.Figure2D*.
- **kwargs** – Positional and keyword arguments are forwarded to *unionfind.plot.Toric.Figure2D*.

class Figure3D (**args, **kwargs*)
Visualizer for the Union-Find decoder and Union-Find based decoders with faulty measurements.

Parameters

- **args** – Positional and keyword arguments are forwarded to *Figure2D* and *plot.Template3D*.
- **kwargs** – Positional and keyword arguments are forwarded to *Figure2D* and *plot.Template3D*.

3.9.3 ufns

The Union-Find Node-Suspension decoder.

Information

The Union-Find Node-Suspension decoder [hu2020thesis] uses the potential matching weight as a heuristic to prioritize growth in specific partitions – the nodes – of the Union-Find cluster (see *Information*). The potential matching weight is approximated by levering a node-tree in the Node-Suspension Data-structure. The elements of the node-tree are descendent objects of *Node*.

The complexity of the algorithm is determined by the calculation of the *node parity* in *ns_parity*, the *node delay* in *ns_delay*, and the growth of the cluster, which is now applied as a recursive function that inspects all nodes in the node tree (*ufns.sim.Toric.grow_node*). During cluster mergers, additional to *union*, node-trees are joined by `join_node_trees`.

Todo: Proper calculation of delay for erasures/empty nodes in the graph

class qsurface.decoders.ufns.elements.**Node** (*primer*)
Element in the node-tree.

A subgraph $\mathcal{V} \subseteq C$ is a spanning-tree of a cluster C if it is a connected acyclic subgraph that includes all vertices of C and a minimum number of edges. We call the spanning-tree of a cluster its ancilla-tree. An acyclic connected spanning-forest is required for the Union-Find Decoder.

A node-tree $\mathcal{N}$ is a partition of a ancilla-tree $\mathcal{V}$, such that each element of the partition – which we call a *node* n – represents a set of adjacent vertices that lie at the same distance – the *node radius :math:`r` – from the *primer* *ancilla*, which initializes the node and lies at its center. The node-tree is a directed acyclic graph, and its edges $\mathcal{E}_i$ have lengths equal to the distance between the primer vertices of neighboring nodes.

Parameters **primer** (*AncillaQubit*) – Primer ancilla-qubit.

short
Short name of the node.

Type str

old_bound
Current boundary edges.

Type list

new_bound
Next boundary edges.

Type list

neighbors
Neighboring nodes in the node-tree.

Type list

root_list
List of even subroots of merged node-trees.

Type list

radius
Node radius size.

Type int

parity
Node parity.

Type {0,1}

delay
Number of iterations to wait.

Type int

waited
Number of iterations waited.

Type int

abstract ns_parity()
Calculates and returns the parity of the current node.

ns_delay(*parent=None*, *min_delay=None*)
Calculates the node delay.

Head recursive function that calculates the delays of the current node and all its descendent nodes.

$$n_d = m_d + \lfloor n_r - m_r \rfloor - (-1)^{n_p} |(n, m)|$$

The minimal delay `min_delay` in the tree is maintained as the actual delay is relative to the minimal delay value within the entire node-tree.

Parameters

- **parent** (Optional[Tuple[*Node*, int]]) – The parent node and the distance to the parent node.
- **min_delay** (Optional[int]) – Minimal delay value encountered during the current calculation.

Return type int

class qsurface.decoders.ufns.elements.**Syndrome**(*primer*)

ns_parity(*parent_node=None*)
Calculates the node parity.

Tail recursive function that calculates the parities of the current node and all its descendent nodes.

$$s_p = \Big(\sum_{n \in \text{ children of } s} (1 + s_p) \Big) \bmod 2$$

Parameters **parent_node** (Optional[*Node*]) – Parent node in node-tree to indicate direction.

Return type int

class qsurface.decoders.ufns.elements.**Junction**(*primer*)

ns_parity(*parent_node=None*)
Calculates the node parity.

Tail recursive function that calculates the parities of the current node and all its children.

$$j_p = 1 - \Big(\sum_{n \in \text{ children of } j} (1 + n_p) \Big) \bmod 2.$$

Parameters **parent_node** (Optional[*Node*]) – Parent node in node-tree to indicate direction.

Return type int

class qsurface.decoders.ufns.elements.**OddNode**(**args*, ***kwargs*)

ns_parity(**args*, ***kwargs*)
Calculates and returns the parity of the current node.

Return type int

qsurface.decoders.ufns.elements.**print_tree**(*current_node*, *parent_node=None*)
Prints the node-tree of `current_node` and its descendents.

Utilizes pptree to print a tree of nodes, which requires a list of children elements per node. Since the node-tree is semi-directional (the root can be any element in the tree), we need to traverse the node-tree from `current_node` in all directions except for the `parent_node` to find the children attributes for the current direction.

Parameters

- **current_node** (*Node*) – Current root of the node-tree to print.
- **parent_node** (Optional[*Node*]) – Parent node which will not be printed. s

Simulation

The following description also applies to `ufns.sim.Planar`.

class qsurface.decoders.ufns.sim.**Toric**(**args*, ***kwargs*)
Union-Find Node-Suspension decoder for the toric lattice.

Within the combined Union-Find and Node-Suspension data structure, every `Cluster` is partitioned into one or more `Node` objectss. The `node` attribute is monkey-patched to the `AncillaQubit` object to assist the identification of its parent `Node`.

The boundary of every cluster is not stored at the cluster object, but divided under its partitioned nodes. Cluster growth is initiated from the root of the node-tree. The attributes `root_node` and `min_delay` are monkey-patched to the `Cluster` object to assist with cluster growth in the Node-Suspension data structure. See `grow_node` for more.

The current class inherits from `unionfind.sim.Toric` for its application the Union-Find data structure for cluster growth and mergers. To maintain low operating complexity in UFNS, the following parameters are set of the Union-Find parent class.

parameter	value
weighted_growth	True
weighted_union	True
dynamic_forest	True

new_boundary
List of newly found cluster boundary elements.

Type list

cluster_add_ancilla(*cluster*, *ancilla*, *parent=None*, ***kwargs*)
Recursively adds erased edges to `cluster` and finds the new boundary.

For a given `ancilla`, this function finds the neighboring edges and ancillas that are in the the currunt cluster. If the newly found edge is erased, the edge and the corresponding ancilla will be added to the cluster, and the function applied recursively on the new ancilla. Otherwise, the neighbor is added to the new boundary `self.new_boundary`.

Parameters

- **cluster** (`Cluster`) – Current active cluster
- **ancilla** (`AncillaQubit`) – Ancilla from which the connected erased edges or boundary are searched.

bound_ancilla_to_node()
Saves the new boundary to their respective nodes.

Saves all the new boundaries found by `cluster_add_ancilla`, which are of the form `[inner_ancilla, edge, outer_ancilla]`, to the node at `inner_ancilla.node`. This method is called after cluster union in `union_bucket`, which also joins the node-trees, such that the new boundary is saved to the updated nodes.

find_clusters(***kwargs*)
Initializes the clusters on the lattice.

For every non-trivial ancilla on the lattice, a `Cluster` is initiated. If any set of ancillas are connected by some set of erased qubits, all connected ancillas are found by `cluster_add_ancilla` and a single cluster is initiated for the set.

Additionally, a syndrome-node is initiated on the non-trivial ancilla – a syndrome – with the ancilla as primer. New boundaries are saved to the nodes by bound_ancilla_to_node.

The cluster is then placed into a bucket based on its size and parity by *place_bucket*. See *grow_clusters* for more information on buckets.

grow_clusters (***kwargs*)
Grows odd-parity clusters outward for union with others until all clusters are even.

Lists of odd-parity clusters are maintained at self.buckets. Starting from bucket 0, odd-parity clusters are popped from the bucket by 'grow_bucket and grown at the boundary by *grow_boundary* by adding 1 for every boundary edge in cluster.bound in self.support. Grown clusters are then placed in a new bucket by *place_bucket* based on its size if it has odd parity.

Edges are fully added to the cluster per two growth iterations. Since a cluster with half-grown edges at the boundary has the same size (number of ancillas) as before growth, but is non-arguably *bigger*, the degeneracy in cluster size is differentiated by cluster.support. When an union occurs between two clusters during growth, if the merged cluster is odd, it is placed in a new bucket. Thus the real bucket number is saved at the cluster locally as cluster.bucket. These two checks are performed before a cluster is grown in *grow_bucket*.

The chronology of events per bucket must be the following:

1. Grow all clusters in the bucket if checks passed.
   - Add all odd-parity clusters after growth to place_list.
   - Add all merging clusters to union_list.
2. Merge all clusters in union_list
   - Add odd-parity clusters after union to place_list.
3. Place all clusters in place_list in new bucket if parity is odd.

For clusters with cluster.support==1 or with half-grown edges at the boundary, the new boundary at clusters.new_bound consists of the same half-grown edges. For clusters with cluster.support==0, the new boundary is found by cluster_add_ancilla.

The current implementation of *grow_clusters* for the ufns decoder currently includes a work-around for a non-frequently occuring bug. Since the grown of a cluster is separated into nodes, and nodes may be *buried* by surrounding cluster trees such that it is an interior element and has no boundaries, it may be possible that when an odd cluster is grown no edges are actually added to the cluster. In this case, due to cluster parity duality the odd cluster will be placed in the same bucket after two rounds of growth. The work-around is to always check if the previous bucket is empty before moving on to the next one.

grow_boundary (*cluster*, *union_list*, ***kwargs*)
Grows the boundary of the cluster.

See *grow_clusters* for more information. Each element in the root_list of the root node of the cluster is a subroot of an even subtree in the node-tree. From each of these subroots, the node parity and delays are calculated by ns_parity and ns_delay. The node-tree is then recursively grown by *grow_node*.

Parameters

- **cluster** (*Cluster*) – The cluster to be grown.
- **union_list** (List[Tuple[*AncillaQubit*, *Edge*, *AncillaQubit*]]) – List of odd-parity clusters to be placed in new buckets.

grow_node (*cluster*, *node*, *union_list*, *parent_node=None*)
Recursive function that grows a node and its descendents.

Grows the boundary list that is stored at the current node if there the current node is not suspended. The condition required is the following:

where $\mathcal{N}$ is the node-tree. The minimal delay value in the node-tree here stored as `cluster.min_delay`. Fully grown edges are added to `union_list` to be later considered by *union_bucket*.

Parameters

- **cluster** (*Cluster*) – Parent cluster object of `node`.
- **node** (*Node*) – Node to consider for growth.
- **union_list** (List[Tuple[*AncillaQubit*, *Edge*, *AncillaQubit*]]) – List of potential mergers between two cluster-distinct ancillas.
- **parent_node** (Optional[*Node*]) – Parent node in the node-tree to indicate recursive direction.

grow_node_boundary (*node*, *union_list*)
Grows the boundary of a `node`.

union_bucket (*union_list*, ***kwargs*)
Potentially merges two neighboring ancillas.

If the check by *union_check* is passed, the clusters of `ancilla` and `new_ancilla` are merged. additionally, the node-trees either directly joined, or by the creation of a new *junction-node* which as `new_ancilla` as its primer. Weighted union is applied to ensure low operating complexity.

check_compatibility ()
Checks compatibility of the decoder with the code class and loaded errors.

correct_edge (*ancilla_qubit*, *key*, ***kwargs*)
Applies a correction.

The correction is applied to the data-qubit located at `ancilla_qubit.parity_qubits[key]`. More specifically, the correction is applied to the *Edge* object corresponding to the `state_type` of `ancilla_qubit`.

Return type *AncillaQubit*

decode (***kwargs*)
Decodes the code using the Union-Find algorithm.

Decoding process can be subdivided into 3 sections:

1. Finding the initial clusters.
2. Growing and merging these clusters.
3. Peeling the clusters using the Peeling algorithm.

Parameters kwargs – Keyword arguments are passed on to *find_clusters*, *grow_clusters* and *peel_clusters*.

flip_edge (*ancilla*, *edge*, *new_ancilla*, ***kwargs*)
Flips the values of the ancillas connected to `edge`.

get_cluster (*ancilla*)
Returns the cluster to which `ancilla` belongs to.

If `ancilla` has no cluster or the cluster is not from the current simulation, none is returned. Otherwise, the root element of the cluster-tree is found, updated to `ancilla.cluster` and returned.

Parameters ancilla (*AncillaQubit*) – The ancilla for which the cluster is to be found.

Return type Optional[*Cluster*]

static get_neighbor (*ancilla_qubit*, *key*)
Returns the neighboring ancilla-qubit of `ancilla_qubit` in the direction of `key`.

Return type Tuple[*AncillaQubit*, *Edge*]

get_neighbors (*ancilla_qubit*, *loop=False*, ***kwargs*)
Returns all neighboring ancillas, including other time instances.

Parameters **loop** (bool) – Include neighbors in time that are not chronologically next to each other during decoding within the same instance.

get_syndrome (*find_pseudo=False*)
Finds the syndrome of the code.

Parameters **find_pseudo** (*bool, optional*) – If enabled, the lists of syndromes returned are not only *AncillaQubit* objects, but tuples of `(ancilla, pseudo)`, where `pseudo` is the closest *PseudoQubit* in the boundary of the code.

Return type Union[Tuple[List[*AncillaQubit*], List[*AncillaQubit*]], Tuple[List[Tuple[*AncillaQubit*, *PseudoQubit*]], List[Tuple[*AncillaQubit*, *PseudoQubit*]]]]

Returns

- *list* – Plaquette operator syndromes.
- *list* – Star operator syndromes.

grow_bucket (*bucket*, *bucket_i*, ***kwargs*)
Grows the clusters which are contained in the current bucket.

See *grow_clusters* for more information.

Parameters

- **bucket** (List[*Cluster*]) – List of clusters to be grown.
- **bucket_i** (int) – Current bucket number.

Return type Tuple[List, List]

Returns

- *list* – List of potential mergers between two cluster-distinct ancillas.
- *list* – List of odd-parity clusters to be placed in new buckets.

peel_clusters (***kwargs*)
Loops over all clusters to find pendant ancillas to peel.

To make sure that all cluster-trees are fully peeled, all ancillas are considered in the loop. If the ancilla has not been peeled before and belongs to a cluster of the current simulation, the ancilla is considered for peeling by *peel_leaf*.

peel_leaf (*cluster*, *ancilla*)
Recursive function which peels a branch of the tree if the input ancilla is a pendant ancilla

If there is only one neighbor of the input ancilla that is in the same cluster, this ancilla is a pendant ancilla and can be peeled. The function calls itself on the other ancilla of the edge leaf.

If ["dynamic_forest"] is disabled, once a pendant leaf is found, the acyclic forest is constructed by *static_forest*.

Parameters

- **cluster** – Current cluster being peeled.
- **ancilla** – Pendant ancilla of the edge to be peeled.

place_bucket (*clusters*, *bucket_i*)
Places all clusters in `clusters` in a bucket if parity is odd.

If `weighted_growth` is enabled. the cluster is placed in a new bucket based on its size, otherwise it is placed in `self.buckets[0]`

Parameters

- **clusters** (`List`[*`Cluster`*]) – Clusters to place in buckets.
- **bucket_i** (`int`) – Current bucket number.

static_forest (*ancilla*)
Constructs an acyclic forest in the cluster of `ancilla`.

Applies recursively to all neighbors of `ancilla`. If a cycle is detected, edges are removed from the cluster.

Parameters **ancilla** (*`AncillaQubit`*) –

union_check (*edge*, *ancilla*, *new_ancilla*, *cluster*, *new_cluster*)
Checks whether `cluster` and `new_cluster` can be joined on `edge`.

See *`union_bucket`* for more information.

Return type `bool`

class qsurface.decoders.ufns.sim.**Planar** (**args*, ***kwargs*)
Union-Find Node-Suspension decoder for the planar lattice.

See the description of *`ufns.sim.Toric`*.

Plotting

class qsurface.decoders.ufns.plot.**Toric** (**args*, ***kwargs*)
Union-Find Node-Suspension decoder for the toric lattice with union-find plot.

Has all class attributes, methods, and nested figure classes from *`ufns.sim.Toric`*, with additional parameters below. Default values for these parameters can be supplied via a *decoders.ini* file under the section of `[ufns]` (see *`decoders._template.read_config`*).

The plotting class initiates a *`qsurface.plot`* object. For its usage, see *Usage*.

Parameters

- **step_bucket** (*`bool, optional`*) – Waits for user after every occupied bucket. Default is false.
- **step_cluster** (*`bool, optional`*) – Waits for user after growth of every cluster. Default is false.
- **step_node** (*`bool, optional`*) – Waits for user after growth of every node. Default is false.
- **step_cycle** (*`bool, optional`*) – Waits for user after every edge removed due to cycle detection. Default is false.
- **step_peel** (*`bool, optional`*) – Waits for user after every edge removed during peeling. Default is false.

class qsurface.decoders.ufns.plot.**Planar**(**args*, ***kwargs*)

Union-Find Node-Suspension decoder for the planar lattice with union-find plot.

Has all class attributes, methods, and nested figure classes from `ufns.sim.Planar`, with additional parameters below. Default values for these parameters can be supplied via a *decoders.ini* file under the section of `[ufns]` (see `decoders._template.read_config`).

The plotting class initiates a `qsurface.plot` object. For its usage, see *Usage*.

Parameters

- **step_bucket** (*bool, optional*) – Waits for user after every occupied bucket. Default is false.
- **step_cluster** (*bool, optional*) – Waits for user after growth of every cluster. Default is false.
- **step_node** (*bool, optional*) – Waits for user after growth of every node. Default is false.
- **step_cycle** (*bool, optional*) – Waits for user after every edge removed due to cycle detection. Default is false.
- **step_peel** (*bool, optional*) – Waits for user after every edge removed during peeling. Default is false.

3.10 Plotting template

3.10.1 Usage

Plot objects that inherit from the template plot classes have the following properties.

- Fast plotting by use of `matplotlib.canvas.blit`.
- Redrawing past iterations of the figure by storing all changes in history.
- Keyboard navigation for iteration selection.
- Plot object information by picking.

When the focus is on the figure, indicated by a green circle in the bottom right corner, the user can navigate through the history of the figure by the commands below.

key	function
h	show help
i	show all iterations
enter or right	go to next iteration, enter iteration number
backspace or left	go to previous iteration
n	go to newest iteration
0-9	input iteration number

If the focus is lost, it can be regained by calling `template.focus` on the plot object.

Default values for plot properties such as colors and linewidths are saved in a *plot.ini* file. Any plot property can be overwritten by supplying the override value as a keyword argument during object initialization or a custom *plot.ini* file in the working directory.

3.10.2 Development

class qsurface.plot.**PlotParams**(*blocking_wait=-1, blocking_pick_radius=10, scale_figure_length=10, scale_figure_height=10, scale_font_primary=12, scale_font_secondary=10, scale_3d_layer=8, color_background=1, 1, 1, 0, color_edge=0.8, 0.8, 0.8, 1, color_qubit_edge=0.7, 0.7, 0.7, 1, color_qubit_face=0.95, 0.95, 0.95, 1, color_x_primary=0.9, 0.3, 0.3, 1, color_z_primary=0.5, 0.5, 0.9, 1, color_y_primary=0.9, 0.9, 0.5, 1, color_x_secondary=0.9, 0.7, 0.3, 1, color_z_secondary=0.3, 0.9, 0.3, 1, color_y_secondary=0.9, 0.9, 0.5, 1, color_x_tertiary=0.5, 0.1, 0.1, 1, color_z_tertiary=0.1, 0.1, 0.5, 1, color_y_tertiary=0.9, 0.9, 0.5, 1, alpha_primary=0.35, alpha_secondary=0.5, line_width_primary=1.5, line_width_secondary=3, line_style_primary='solid', line_style_secondary='dashed', line_style_tertiary='dotted', patch_circle_2d=0.1, patch_rectangle_2d=0.1, patch_circle_3d=30, patch_rectangle_3d=30, axis_main=0.075, 0.1, 0.7, 0.85, axis_main_non_interact=0.0, 0.05, 0.8, 0.9, axis_block=0.96, 0.01, 0.03, 0.03, axis_nextbutton=0.85, 0.05, 0.125, 0.05, axis_prevbutton=0.85, 0.12, 0.125, 0.05, axis_legend=0.85, 0.5, 0.125, 0.3, axis_text=0.05, 0.025, 0.7, 0.05, axis_radio=0.85, 0.19, 0.125, 0.125, font_default_size=12, font_title_size=16, font_button_size=12, axis3d_pane_color=1, 1, 1, 0, axis3d_line_color=0, 0, 0, 0.1, axis3d_grid_line_style='dotted', axis3d_grid_line_alpha=0.2*)

Parameters for the plotting template classes.

Contains all parameters used in inherited objects of `Template2D` and `Template3D`. The dataclass is initialized with many default values for an optimal plotting experience. But if any parameters should be changed, the user can call the class to create its own instance of plotting paramters, where the altered paramters are supplied as keyword arguments. The instance can be supplied to the plotting class via the `plot_params` keyword argument.

Examples

See the below example where the background color of the figure is changed to black. Note that we have to inherit from the `Template2D` class.

```
>>> class Plotting(Template2D):
...     pass
>>> custom_params = PlotParams(color_background = (0,0,0,1))
>>> plot_with_custom_params = Plotting(plot_params=custom_params)
```

load_params(*param_dict*)

Loads extra plotting parameters.

Additional parameters can be loaded to the dataclass via this method. The additional parameters must be a dictionary where values are stored to the dataclass with the key as attribute name. If the value is a string that equals to any already defined dataclass attribute, the value at the existing attribute is used for the new parameter. See examples.

Parameters **params_dict** – Dictionary or dictionary of dictionaries of additional parameters.

Examples

New parameters can be added to the dataclass. Values of dataclass attributes are used if present.

```
>>> params = PlotParams()
>>> params.alpha_primary
0.35
>>> params.load_params({
...     "new_attr" : "some_value",
...     "use_existing" : "alpha_primary",
... })
>>> params.new_attr
some_value
>>> params.use_existing
0.35
```

Nested dictionaries will also load existing attribute values.

```
>>> params.load_params({
...     "category": {
...         "new_attr" : "some_value",
...         "use_existing" : "alpha_primary",
...     }
... })
>>> params.category
{"new_attr" : "some_value", "use_existing" : 0.35}
```

class qsurface.plot.**BlockingKeyInput**(**args*, ***kwargs*)

Blocking class to receive key presses.

See also:

None Inherited blocking class.

class qsurface.plot.**Template2D**(*plot_params=None*, *projection=None*, ***kwargs*)

Template 2D plot object with history navigation.

This template plot object which can either be an interactive figure using the Tkinter backend, or shows each plotting iteration as a separate figure for the IPython `inline` backend. The interactive figure has the following features.

- Fast plotting by use of "blitting".
- Redrawing past iterations of the figure by storing all changes in history.
- Keyboard navigation for iteration selection.
- Plot object information by picking.

To instance this class, one must inherit the current class. The existing objects can then be altered by updating their plot properties by *`new_properties()`*, where the changed properties must be a dictionary with keywords and values corresponding tho the respective matplotlib object. Every change in plot property is stored in `self.history_dict`. This allows to undo or redo changes by simply applying the saved changed properties in the dictionary. Fast plotting is enabled by not drawing the figure after every queued change. Instead, each object is draw in the canvas individually after a property change and a series of changes is drawn to the figure when a new plot iteration is requested via `new_iter()`. This is performed by *blitting* the canvas.

Keyboard navigation and picking is enabled by blocking the code via a custom *`BlockingKeyInput`* class. While the code is blocked, inputs are caught by the blocking class and processed for history navigation or picking navigation. Moving the iteration past the available history allows for the code to continue. The keyboard input is parsed by *focus()*.

Default values for plot properties such as colors and linewidths loaded from *PlotParams*. A custom parameter dataclass can be supplied via the `plot_params` keyword argument.

Parameters **plot_params** (Optional[*PlotParams*]) – Plotting parameters dataclass containing colors, styles and others.

figure
Main figure.

Type matplotlib.figure.Figure

interactive
Enables GUI elements and interactive plotting.

Type bool

main_ax
Main axis of the figure.

Type matplotlib.axes.Axes

history_dict
For each iteration, for every plot object with changed properties, the properties are stored as a nested dictionary. See the example below.

```
>>> history_dict = {
        0: {
            "<Line2D object>": {
                "color": "k",
            },
            "<Circle object>": {
                "linestyle": "-",
            }
        }
        1: {
            "<Line2D object>": {
                "color": "r",
            },
            "<Circle object>": {
                "linestyle": ":",
            }
        }
    }
```

Type collections.defaultdict

history_iters
Total number of iterations in history.

Type int

history_iter
The current plot iteration.

Type int

history_iter_names
List of length `history_iters` containing a title for each iteration.

Type list of str

history_at_newest
Whether the current plot iteration is the latest or newest.

Type bool

history_event_iter
String catching the keyboard input for the wanted plot iteration.

Type str

future_dict
Same as `history_dict` but for changes for future iterations.

Type `collections.defaultdict`

temporary_changes
Temporary changes for plot properties, requested by *`temporary_properties()`*, which are immediately drawn to the figure. These properties can be overwritten or undone before a new iteration is requested via `new_iter()`. When a new iteration is requested, we need to find the difference in properties of the queued changes with the current iteration and save all differences to `self.history_dict`.

Type `collections.defaultdict`

temporary_saved
Temporary changes are saved to the current iteration `iter`. Thus when a new iteration `iter + 1` is requested, we need to recalculate the differences of the properties in `iter-1` and the current iteration with the temporary changes. The previous property values when temporary changes are requested by *`temporary_properties()`* are saved to `self.temporary_saved` and used as the property changes for `iter-1`.

Type `collections.defaultdict`

interact_axes
All iteractive elements should have their own axis saved in `self.interact_axes`. The `axis.active` attribute must be added to define when the axis is shown. If the focus on the figure is lost, all axes in `self.interact_axes` are hidden by setting `axis.active=False`.

Type dict of `matplotlib.axes.Axes`

interact_bodies
All interactive elements such as buttons, radiobuttons, sliders, should be saved to this dictionary with the same key as their axes in `self.interact_axes`.

Type dict

Notes

Note all backends support blitting. It does not work with the OSX backend (but does work with other GUI backends on mac).

Examples

A matplotlib.lines.Line2D object is initiated with `color="k"` and `ls="-"`. We request that the color of the object is red in a new plot iteration.

```
>>> import matplotlib.pyplot as plt
... class Example(Template2D):
...     def __init__(self, *args, **kwargs):
...         super().__init__(*args, **kwargs)
...         self.line = plt.plot(0, 0, color="k", ls="-")[0]     # Line located at␣
↪[0] after plot
>>> fig = Example()
>>> fig.new_properties(fig.line, {"color": "r})
>>> fig.new_iter()
>>> fig.history_dict
{
    0: {"<Line2D>": {"color": "k"}},
    1: {"<Line2D>": {"color": "r"}},
}
```

The attribute `self.history_dict` thus only contain changes to plot properties. If we request another iteration but change the linestyle to ":", the initial linestyle will be saved to iteration 1.

```
>>> fig.new_properties(fig.line, {"ls": ":"})
>>> fig.new_iter()
>>> fig.history_dict
{
    0: {"<Line2D>": {"color": "k"}},
    1: {"<Line2D>": {"color": "r", "ls: "-"}},
    2: {"<Line2D>": {ls: ":"}},
}
```

We temporarily alter the linewidth to 2, and then to 1.5. After we are satisfied with the temporary changes. we request a new iteration with the final change of color to green.

```
>>> fig.temporary_properties(fig.line, {"lw": 2})
>>> fig.temporary_properties(fig.line, {"lw": 1.5})
>>> fig.temporary_changes
{"<Line2D>": {"lw": 1.5}}
>>> fig.temporary_saved
{"<Line2D>": {"lw": 1}}       # default value
>>> fig.new_properties(fig.line, {"color": "g"})
>>> fig.new_iter()
>>> fig.history_dict
{
    0: {"<Line2D>": {"color": "k"}},
    1: {"<Line2D>": {"color": "r", "ls: "-", "lw": 1}},
    2: {"<Line2D>": {"lw": 1.5, color": "r"},
    3: {"<Line2D>": {"color": "g"}},
}
```

Properties in `self.temporary_saved` are saved to `self.history_dict` in the previous iteration, properties in `self.temporary_changes` are saved to the current iteration, and new properties are saved to the new iteration.

The `history_dict` for a plot with a Line2D object and a Circle object. In the second iteration, the color of the Line2D object is updated from black to red, and the linestyle of the Circle object is changed from "-" to ":".

load_interactive_backend()

Configures the plotting backend.

If the Tkinter backend is enabled or can be enabled, the function returns True. For other backends False is returned.

Return type `bool`

close()

Closes the figure.

focus()

Enables the blocking object, catches input for history navigation.

The BlockingKeyInput object is called which blocks the execution of the code. During this block, the user input is received by the blocking object and return to the current method. From here, we can manipulate the plot or move through the plot history and call *focus()* again when all changes in the history have been drawn and blit.

key	function
h	show help
i	show all iterations
d	redraw current iteration
enter or right	go to next iteration, enter iteration number
backspace or left	go to previous iteration
n	go to newest iteration
0-9	input iteration number

When the method is active, the focus is on the figure. This will be indicated by a green circle in the bottom right of the figure. When the focus is lost, the code execution is continued and the icon is red. The change is icon color is performed by `_set_figure_state()`, which also hides the interactive elements when the focus is lost.

draw_figure(*new_iter_name=None*, *output=True*, *carriage_return=False*, ***kwargs*)

Draws the canvas and blocks code execution.

Draws the queued plot changes onto the canvas and calls for *focus()* which blocks the code execution and catches user input for history navigation.

If a new iteration is called by supplying a `new_iter_name`, we additionally check for future property changes in the `self.future_dict`, and add these changes to the queue. Finally, all queued property changes for the next iteration are applied by *change_properties*.

Parameters

- **new_iter_name** (`Optional[str]`) – Name of the new iteration. If no name is supplied, no new iteration is called.
- **output** (`bool`) – Prints information to the console.
- **carriage_return** (`bool`) – Applies carriage return to remove last line printed.

See also:

focus(), *change_properties()*

new_artist(*artist*, *axis=None*)

Adds a new artist to the `axis`.

Newly added artists must be hidden in the previous iteration. To make sure the history is properly logged, the visibility of the `artist` is set to `False`, and a new property of shown visibility is added to the queue of the next iteration.

Parameters

- **`artist`** (`Artist`) – New plot artist to add to the `axis`.
- **`axis`** (`Optional[Axes]`) – Axis to add the figure to.

Return type `None`

`static change_properties`(*artist*, *prop_dict*)

Changes the plot properties and draw the plot object or artist.

`new_properties`(*artist*, *properties*, *saved_properties={}*, ***kwargs*)

Parses a dictionary of property changes of a *matplotlib* artist.

New properties are supplied via `properties`. If any of the new properties is different from its current value, this is seen as a property change. The old property value is stored in `self.history_dict[self.history_iteration]`, and the new property value is stored at `self.history_dict[self.history_iteration+1]`. These new properties are *queued* for the next interation. The queue is emptied by applying all changes when *`draw_figure`* is called. If the same property changes 2+ times within the same iteration, the previous property change is removed with `next_prop.pop(key, None)`.

The `saved_properties` parameter is used when temporary property changes have been applied by *`temporary_changes`*, in which the original properties are saved to `self.temporary_saved` as the saved properties. Before a new iteration is drawn, the temporary changes, which can be overwritten, are compared with the saved changes and the differences in properties are saved to `[self.history_dict[self.history_iter-1]]` and `self.history_dict[self.history_iteration]`.

Some color values from different *matplotlib* objects are nested, some are list or tuple, and others may be a `numpy.ndarray`. The nested methods `get_nested()` and `get_nested_property()` make sure that the return type is always a list.

Parameters

- **`artist`** (`Artist`) – Plot object whose properties are changed.
- **`properties`** (`dict`) – Plot properties to change.
- **`saved_properties`** (`dict`) – Override current properties and parse previous and current history.

`temporary_properties`(*artist*, *properties*, ***kwargs*)

Applies temporary property changes to a *matplotlib* artist.

Only available on the newest iteration, as we cannot change what is already in the past. All values in `properties` are immediately applied to `artist`. Since temporary changes can be overwritten within the same iteration, the first time a temporary property change is requested, the previous value is saved to `self.temporary_saved`. When the iteration changes, the property differences of the previous and current iteration are recomputed and saved to `self.history_dict` in `_draw_from_history()`.

Parameters

- **`artist`** (`Artist`) – Plot object whose properties are changed.
- **`properties`** (`dict`) – Plot properties to change.

`class` `qsurface.plot.`**`Template3D`**(**args*, ***kwargs*)

Template 3D plot object with history navigation.

CHAPTER

FOUR

LICENSE

BSD 3-Clause License

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[wang2003confinement] Chenyang Wang, Jim Harrington and John Preskill, *Confinement-Higgs transition in a disordered gauge theory and the accuracy threshold for quantum memory*, Annals of Physics, 1:31-58, 2003.

[dennis2002] Dennis, Eric and Kitaev, Alexei and Landahl, Andrew and Preskill, John, *Topological quantum memory*, in **Journal of Mathematical Physics**, 2002, 43(9)4452-4505.

[wang2003] Wang, Chenyang and Harrington, Jim and Preskill, John, *Confinement-Higgs transition in a disordered gauge theory and the accuracy threshold for quantum memory*, in **Annals of Physics**, 2003, 303(1)31-58.

[kolmogorov2009] Kolmogorov, Vladimir, *Blossom V: A new implementation of a minimum cost perfect matching algorithm* in **Mathematical Programming Computation (MPC)**, July 2009, 1(1):43-67.

[delfosse2017almost] Delfosse, Nicolas and Nickerson, Naomi H., *Almost-linear time decoding algorithm for topological codes*, arXiv preprint arXiv:1709.06218, 2017.

[delfosse2017linear] Delfosse, Nicolas and Zemor, Gilles, *Linear-time maximum likelihood decoding of surface codes over the quantum erasure channel*, arXiv preprint arXiv:1703.01517, 2017.

[tarjan1975efficiency] Tarjan, Robert, *Efficiency of a good but not linear set union algorithm*, Journal of the ACM, 22(2)215-225, 1975.

[hu2020thesis] Hu, Mark Shui, *Quasilinear Time Decoding Algorithm for Topological Codes with High Error Threshold*, DOI: 10.13140/RG.2.2.13495.96162, 2020.

PYTHON MODULE INDEX

q

qsurface.codes.elements, 14
qsurface.decoders._template, 47
qsurface.decoders.mwpm, 50
qsurface.decoders.mwpm.plot, 55
qsurface.decoders.ufns, 63
qsurface.decoders.ufns.elements, 63
qsurface.decoders.unionfind, 55
qsurface.decoders.unionfind.elements, 55
qsurface.errors._template, 43
qsurface.errors.erasure, 46
qsurface.errors.pauli, 46
qsurface.main, 7
qsurface.plot, 72
qsurface.threshold, 11

INDEX

A

add_ancilla() (qsurface.decoders.unionfind.elements.Cluster method), 56
add_ancilla_qubit() (qsurface.codes._template.plot.PerfectMeasurements method), 24
add_ancilla_qubit() (qsurface.codes._template.sim.FaultyMeasurements method), 19
add_ancilla_qubit() (qsurface.codes._template.sim.PerfectMeasurements method), 18
add_ancilla_qubit() (qsurface.codes.planar.plot.PerfectMeasurements method), 40
add_ancilla_qubit() (qsurface.codes.planar.sim.FaultyMeasurements method), 36
add_ancilla_qubit() (qsurface.codes.planar.sim.PerfectMeasurements method), 34
add_ancilla_qubit() (qsurface.codes.toric.plot.PerfectMeasurements method), 32
add_ancilla_qubit() (qsurface.codes.toric.sim.FaultyMeasurements method), 27
add_ancilla_qubit() (qsurface.codes.toric.sim.PerfectMeasurements method), 26
add_data_qubit() (qsurface.codes._template.plot.PerfectMeasurements method), 24
add_data_qubit() (qsurface.codes._template.sim.FaultyMeasurements method), 20
add_data_qubit() (qsurface.codes._template.sim.PerfectMeasurements method), 18
add_data_qubit() (qsurface.codes.planar.plot.PerfectMeasurements method), 40
add_data_qubit() (qsurface.codes.planar.sim.FaultyMeasurements method), 36
add_data_qubit() (qsurface.codes.planar.sim.PerfectMeasurements method), 35
add_data_qubit() (qsurface.codes.toric.plot.PerfectMeasurements method), 32
add_data_qubit() (qsurface.codes.toric.sim.FaultyMeasurements method), 27
add_data_qubit() (qsurface.codes.toric.sim.PerfectMeasurements method), 26
add_node() (qsurface.codes.elements.Edge method), 15
add_node() (qsurface.codes.elements.PseudoEdge method), 16
add_pseudo_qubit() (qsurface.codes._template.plot.PerfectMeasurements method), 24
add_pseudo_qubit() (qsurface.codes._template.sim.FaultyMeasurements method), 20
add_pseudo_qubit() (qsurface.codes._template.sim.PerfectMeasurements method), 18
add_pseudo_qubit() (qsurface.codes.planar.plot.PerfectMeasurements method), 40
add_pseudo_qubit() (qsurface.codes.planar.sim.FaultyMeasurements method), 36
add_pseudo_qubit() (qsurface.codes.planar.sim.PerfectMeasurements method), 35
add_pseudo_qubit() (qsurface.codes.toric.plot.PerfectMeasurements method), 32
add_pseudo_qubit() (qsur-

face.codes.toric.sim.FaultyMeasurements method), 28
add_pseudo_qubit() (qsurface.codes.toric.sim.PerfectMeasurements method), 26
add_vertical_edge() (qsurface.codes._template.sim.FaultyMeasurements method), 19
add_vertical_edge() (qsurface.codes.planar.sim.FaultyMeasurements method), 36
add_vertical_edge() (qsurface.codes.toric.sim.FaultyMeasurements method), 28
ancilla_qubits (qsurface.codes._template.sim.PerfectMeasurements attribute), 16
AncillaQubit (class in qsurface.codes.elements), 14

B

BenchmarkDecoder (class in qsurface.main), 9
bitflip() (qsurface.errors.pauli.Sim static method), 46
bitphaseflip() (qsurface.errors.pauli.Sim static method), 46
BlockingKeyInput (class in qsurface.plot), 73
bound_ancilla_to_node() (qsurface.decoders.ufns.sim.Toric method), 66
bucket (qsurface.decoders.unionfind.elements.Cluster attribute), 56
bucket_max_filled (qsurface.decoders.unionfind.sim.Toric attribute), 58
buckets (qsurface.decoders.unionfind.sim.Toric attribute), 58

C

change_properties() (qsurface.codes._template.plot.PerfectMeasurements.Figure static method), 22
change_properties() (qsurface.codes.planar.plot.PerfectMeasurements.Figure static method), 38
change_properties() (qsurface.codes.toric.plot.PerfectMeasurements.Figure static method), 30
change_properties() (qsurface.plot.Template2D static method), 78
check_compatibility() (qsurface.decoders._template.Plot method), 49
check_compatibility() (qsurface.decoders._template.Sim method), 48
check_compatibility() (qsurface.decoders.mwpm.sim.Planar method), 53
check_compatibility() (qsurface.decoders.mwpm.sim.Toric method), 52
check_compatibility() (qsurface.decoders.ufns.sim.Toric method), 68
check_compatibility() (qsurface.decoders.unionfind.sim.Toric method), 61
close() (qsurface.codes._template.plot.PerfectMeasurements.Figure method), 22
close() (qsurface.codes.planar.plot.PerfectMeasurements.Figure method), 38
close() (qsurface.codes.toric.plot.PerfectMeasurements.Figure method), 30
close() (qsurface.plot.Template2D method), 77
Cluster (class in qsurface.decoders.unionfind.elements), 55
cluster_add_ancilla() (qsurface.decoders.ufns.sim.Toric method), 66
cluster_add_ancilla() (qsurface.decoders.unionfind.sim.Toric method), 58
cluster_index (qsurface.decoders.unionfind.sim.Toric attribute), 58
clusters (qsurface.decoders.unionfind.sim.Toric attribute), 58
code_params (qsurface.codes._template.plot.PerfectMeasurements.Figure attribute), 22
compatibility_errors (qsurface.decoders._template.Sim attribute), 48
compatibility_measurements (qsurface.decoders._template.Sim attribute), 48
correct_edge() (qsurface.decoders._template.Plot method), 49
correct_edge() (qsurface.decoders._template.Sim method), 48
correct_edge() (qsurface.decoders.mwpm.sim.Planar method), 53
correct_edge() (qsurface.decoders.mwpm.sim.Toric method), 52
correct_edge() (qsurface.decoders.ufns.sim.Toric method), 68
correct_edge() (qsurface.decoders.unionfind.sim.Toric method), 61
correct_matching() (qsurface.decoders.mwpm.sim.Planar method), 53
correct_matching() (qsurface.decoders.mwpm.sim.Toric method),

52
count_calls() (qsurface.main.BenchmarkDecoder method), 11

D

data (qsurface.main.BenchmarkDecoder attribute), 10
data_qubits (qsurface.codes._template.sim.PerfectMeasurements attribute), 16
DataQubit (class in qsurface.codes.elements), 14
decode() (qsurface.decoders._template.Plot method), 49
decode() (qsurface.decoders._template.Sim method), 49
decode() (qsurface.decoders.mwpm.sim.Planar method), 53
decode() (qsurface.decoders.mwpm.sim.Toric method), 51
decode() (qsurface.decoders.ufns.sim.Toric method), 68
decode() (qsurface.decoders.unionfind.sim.Toric method), 58
default_error_rates (qsurface.errors._template.Sim attribute), 43
delay (qsurface.decoders.ufns.elements.Node attribute), 64
draw_figure() (qsurface.codes._template.plot.PerfectMeasurements.Figure method), 22
draw_figure() (qsurface.codes.planar.plot.PerfectMeasurements.Figure method), 38
draw_figure() (qsurface.codes.toric.plot.PerfectMeasurements.Figure method), 30
draw_figure() (qsurface.plot.Template2D method), 77
duration() (qsurface.main.BenchmarkDecoder method), 11

E

Edge (class in qsurface.codes.elements), 15
edges (qsurface.codes.elements.DataQubit attribute), 14
entangle_pair() (qsurface.codes._template.plot.PerfectMeasurements static method), 24
entangle_pair() (qsurface.codes._template.sim.FaultyMeasurements static method), 20
entangle_pair() (qsurface.codes._template.sim.PerfectMeasurements static method), 18
entangle_pair() (qsurface.codes.planar.plot.PerfectMeasurements static method), 41
entangle_pair() (qsurface.codes.planar.sim.FaultyMeasurements static method), 36
entangle_pair() (qsurface.codes.planar.sim.PerfectMeasurements static method), 35
entangle_pair() (qsurface.codes.toric.plot.PerfectMeasurements static method), 32
entangle_pair() (qsurface.codes.toric.sim.FaultyMeasurements static method), 28
entangle_pair() (qsurface.codes.toric.sim.PerfectMeasurements static method), 26
erasure() (qsurface.errors.erasure.Sim static method), 46
error_methods (qsurface.codes._template.plot.PerfectMeasurements.Figure attribute), 22
error_methods (qsurface.errors._template.Plot attribute), 45
errors (qsurface.codes._template.sim.PerfectMeasurements attribute), 17

F

FaultyMeasurements (class in qsurface.codes._template.plot), 25
FaultyMeasurements (class in qsurface.codes._template.sim), 18
FaultyMeasurements (class in qsurface.codes.planar.plot), 42
FaultyMeasurements (class in qsurface.codes.planar.sim), 36
FaultyMeasurements (class in qsurface.codes.toric.plot), 34
FaultyMeasurements (class in qsurface.codes.toric.sim), 27
figure (qsurface.codes._template.plot.PerfectMeasurements attribute), 21
figure (qsurface.plot.Template2D attribute), 74
find() (qsurface.decoders.unionfind.elements.Cluster method), 56
find_clusters() (qsurface.decoders.ufns.sim.Toric method), 66
find_clusters() (qsurface.decoders.unionfind.sim.Toric method), 59
fit_data() (qsurface.threshold.ThresholdFit method), 13
flip_edge() (qsurface.decoders.ufns.sim.Toric method), 68

flip_edge() (*qsurface.decoders.unionfind.sim.Toric method*), 61
focus() (*qsurface.codes._template.plot.PerfectMeasurements.Figure method*), 23
focus() (*qsurface.codes.planar.plot.PerfectMeasurements.Figure method*), 38
focus() (*qsurface.codes.toric.plot.PerfectMeasurements.Figure method*), 30
focus() (*qsurface.plot.Template2D method*), 77
future_dict (*qsurface.plot.Template2D attribute*), 75

G

get_blossomv() (*in module qsurface.decoders.mwpm*), 50
get_cluster() (*qsurface.decoders.ufns.sim.Toric method*), 68
get_cluster() (*qsurface.decoders.unionfind.sim.Toric method*), 58
get_neighbor() (*qsurface.decoders._template.Plot static method*), 50
get_neighbor() (*qsurface.decoders._template.Sim static method*), 48
get_neighbor() (*qsurface.decoders.mwpm.sim.Planar static method*), 53
get_neighbor() (*qsurface.decoders.mwpm.sim.Toric static method*), 52
get_neighbor() (*qsurface.decoders.ufns.sim.Toric static method*), 69
get_neighbor() (*qsurface.decoders.unionfind.sim.Toric static method*), 61
get_neighbors() (*qsurface.decoders._template.Plot method*), 50
get_neighbors() (*qsurface.decoders._template.Sim method*), 48
get_neighbors() (*qsurface.decoders.mwpm.sim.Planar method*), 53
get_neighbors() (*qsurface.decoders.mwpm.sim.Toric method*), 52
get_neighbors() (*qsurface.decoders.ufns.sim.Toric method*), 69
get_neighbors() (*qsurface.decoders.unionfind.sim.Toric method*), 61
get_qubit_distances() (*qsurface.decoders.mwpm.sim.Planar static method*), 53
get_qubit_distances() (*qsurface.decoders.mwpm.sim.Toric static method*), 52
get_syndrome() (*qsurface.decoders._template.Plot method*), 50
get_syndrome() (*qsurface.decoders._template.Sim method*), 48
get_syndrome() (*qsurface.decoders.mwpm.sim.Planar method*), 53
get_syndrome() (*qsurface.decoders.mwpm.sim.Toric method*), 52
get_syndrome() (*qsurface.decoders.ufns.sim.Toric method*), 69
get_syndrome() (*qsurface.decoders.unionfind.sim.Toric method*), 61
grow_boundary() (*qsurface.decoders.ufns.sim.Toric method*), 67
grow_boundary() (*qsurface.decoders.unionfind.sim.Toric method*), 60
grow_bucket() (*qsurface.decoders.ufns.sim.Toric method*), 69
grow_bucket() (*qsurface.decoders.unionfind.sim.Toric method*), 59
grow_clusters() (*qsurface.decoders.ufns.sim.Toric method*), 67
grow_clusters() (*qsurface.decoders.unionfind.sim.Toric method*), 59
grow_node() (*qsurface.decoders.ufns.sim.Toric method*), 67
grow_node_boundary() (*qsurface.decoders.ufns.sim.Toric method*), 68
gui_methods (*qsurface.errors._template.Plot attribute*), 45
gui_permanent (*qsurface.errors._template.Plot attribute*), 45

H

history_at_newest (*qsurface.plot.Template2D attribute*), 75
history_at_newest() (*qsurface.codes._template.plot.PerfectMeasurements.Figure property*), 23
history_at_newest() (*qsurface.codes.planar.plot.PerfectMeasurements.Figure property*), 39
history_at_newest() (*qsurface.codes.toric.plot.PerfectMeasurements.Figure property*), 31
history_dict (*qsurface.plot.Template2D attribute*), 74

history_event_iter (*qsurface.plot.Template2D attribute*), 75
history_iter (*qsurface.plot.Template2D attribute*), 74
history_iter_names (*qsurface.plot.Template2D attribute*), 74
history_iters (*qsurface.plot.Template2D attribute*), 74

I

init_config() (*in module qsurface.decoders._template*), 48
init_errors() (*qsurface.codes._template.plot.PerfectMeasurements method*), 25
init_errors() (*qsurface.codes._template.sim.FaultyMeasurements method*), 20
init_errors() (*qsurface.codes._template.sim.PerfectMeasurements method*), 17
init_errors() (*qsurface.codes.planar.plot.PerfectMeasurements method*), 41
init_errors() (*qsurface.codes.planar.sim.FaultyMeasurements method*), 37
init_errors() (*qsurface.codes.planar.sim.PerfectMeasurements method*), 35
init_errors() (*qsurface.codes.toric.plot.PerfectMeasurements method*), 32
init_errors() (*qsurface.codes.toric.sim.FaultyMeasurements method*), 28
init_errors() (*qsurface.codes.toric.sim.PerfectMeasurements method*), 26
init_legend() (*qsurface.codes._template.plot.PerfectMeasurements.Figure method*), 22
init_legend() (*qsurface.codes.planar.plot.PerfectMeasurements.Figure method*), 39
init_legend() (*qsurface.codes.toric.plot.PerfectMeasurements.Figure method*), 31
init_logical_operator() (*qsurface.codes._template.plot.PerfectMeasurements method*), 25
init_logical_operator() (*qsurface.codes._template.sim.FaultyMeasurements method*), 20
init_logical_operator() (*qsurface.codes._template.sim.PerfectMeasurements method*), 17
init_logical_operator() (*qsurface.codes.planar.plot.PerfectMeasurements method*), 41
init_logical_operator() (*qsurface.codes.planar.sim.FaultyMeasurements method*), 37
init_logical_operator() (*qsurface.codes.planar.sim.PerfectMeasurements method*), 34
init_logical_operator() (*qsurface.codes.toric.plot.PerfectMeasurements method*), 33
init_logical_operator() (*qsurface.codes.toric.sim.FaultyMeasurements method*), 29
init_logical_operator() (*qsurface.codes.toric.sim.PerfectMeasurements method*), 26
init_parity_check() (*qsurface.codes.planar.plot.PerfectMeasurements method*), 41
init_parity_check() (*qsurface.codes.planar.sim.FaultyMeasurements method*), 37
init_parity_check() (*qsurface.codes.planar.sim.PerfectMeasurements method*), 34
init_parity_check() (*qsurface.codes.toric.plot.PerfectMeasurements method*), 33
init_parity_check() (*qsurface.codes.toric.sim.FaultyMeasurements method*), 29
init_parity_check() (*qsurface.codes.toric.sim.PerfectMeasurements method*), 26
init_plot() (*qsurface.codes._template.plot.PerfectMeasurements.Figure method*), 22
init_plot() (*qsurface.codes.planar.plot.PerfectMeasurements.Figure method*), 39
init_plot() (*qsurface.codes.toric.plot.PerfectMeasurements.Figure method*), 31
init_surface() (*qsurface.codes._template.plot.PerfectMeasurements method*), 25
init_surface() (*qsurface.codes._template.sim.FaultyMeasurements method*), 19
init_surface() (*qsurface.codes._template.sim.PerfectMeasurements method*), 17

init_surface() (qsurface.codes.planar.plot.PerfectMeasurements method), 41
init_surface() (qsurface.codes.planar.sim.FaultyMeasurements method), 37
init_surface() (qsurface.codes.planar.sim.PerfectMeasurements method), 34
init_surface() (qsurface.codes.toric.plot.PerfectMeasurements method), 33
init_surface() (qsurface.codes.toric.sim.FaultyMeasurements method), 29
init_surface() (qsurface.codes.toric.sim.PerfectMeasurements method), 26
initialize() (in module qsurface.main), 7
initialize() (qsurface.codes._template.plot.PerfectMeasurements method), 21
initialize() (qsurface.codes._template.sim.FaultyMeasurements method), 21
initialize() (qsurface.codes._template.sim.PerfectMeasurements method), 17
initialize() (qsurface.codes.planar.plot.PerfectMeasurements method), 42
initialize() (qsurface.codes.planar.sim.FaultyMeasurements method), 37
initialize() (qsurface.codes.planar.sim.PerfectMeasurements method), 35
initialize() (qsurface.codes.toric.plot.PerfectMeasurements method), 33
initialize() (qsurface.codes.toric.sim.FaultyMeasurements method), 29
initialize() (qsurface.codes.toric.sim.PerfectMeasurements method), 27
instance (qsurface.codes._template.sim.PerfectMeasurements attribute), 17
interact_axes (qsurface.plot.Template2D attribute), 75
interact_bodies (qsurface.plot.Template2D attribute), 75
interactive (qsurface.plot.Template2D attribute), 74

J

Junction (class in qsurface.decoders.ufns.elements), 65

L

legend_titles (qsurface.errors._template.Plot attribute), 45
line_color_match (qsurface.decoders._template.Plot attribute), 49
line_color_normal (qsurface.decoders._template.Plot attribute), 49
lists (qsurface.main.BenchmarkDecoder attribute), 10
lists_mean_var() (qsurface.main.BenchmarkDecoder method), 11
load_interactive_backend() (qsurface.codes._template.plot.PerfectMeasurements.Figure method), 23
load_interactive_backend() (qsurface.codes.planar.plot.PerfectMeasurements.Figure method), 39
load_interactive_backend() (qsurface.codes.toric.plot.PerfectMeasurements.Figure method), 31
load_interactive_backend() (qsurface.plot.Template2D method), 76
load_params() (qsurface.plot.PlotParams method), 72
logical_operators (qsurface.codes._template.sim.PerfectMeasurements attribute), 17
logical_state (qsurface.codes._template.sim.PerfectMeasurements attribute), 17

M

main_ax (qsurface.plot.Template2D attribute), 74
match_blossomv() (qsurface.decoders.mwpm.sim.Planar static method), 54
match_blossomv() (qsurface.decoders.mwpm.sim.Toric static method), 52
match_networkx() (qsurface.decoders.mwpm.sim.Planar static method), 54
match_networkx() (qsurface.decoders.mwpm.sim.Toric static method), 52
match_syndromes() (qsurface.decoders.mwpm.sim.Planar method), 54

match_syndromes() (qsurface.decoders.mwpm.sim.Toric method), 51
matching_lines (qsurface.decoders._template.Plot attribute), 49
measure() (qsurface.codes.elements.AncillaQubit method), 15
measure() (qsurface.codes.elements.PseudoQubit method), 16
measured_state (qsurface.codes.elements.AncillaQubit attribute), 15
measurement_error (qsurface.codes.elements.AncillaQubit attribute), 15
module
    qsurface.codes.elements, 14
    qsurface.decoders._template, 47
    qsurface.decoders.mwpm, 50
    qsurface.decoders.mwpm.plot, 55
    qsurface.decoders.ufns, 63
    qsurface.decoders.ufns.elements, 63
    qsurface.decoders.unionfind, 55
    qsurface.decoders.unionfind.elements, 55
    qsurface.errors._template, 43
    qsurface.errors.erasure, 46
    qsurface.errors.pauli, 46
    qsurface.main, 7
    qsurface.plot, 72
    qsurface.threshold, 11

N

neighbors (qsurface.decoders.ufns.elements.Node attribute), 64
new_artist() (qsurface.codes._template.plot.PerfectMeasurements.Figure method), 23
new_artist() (qsurface.codes.planar.plot.PerfectMeasurements.Figure method), 39
new_artist() (qsurface.codes.toric.plot.PerfectMeasurements.Figure method), 31
new_artist() (qsurface.plot.Template2D method), 77
new_bound (qsurface.decoders.ufns.elements.Node attribute), 64
new_boundary (qsurface.decoders.ufns.sim.Toric attribute), 66
new_properties() (qsurface.codes._template.plot.PerfectMeasurements.Figure method), 23
new_properties() (qsurface.codes.planar.plot.PerfectMeasurements.Figure method), 40
new_properties() (qsurface.codes.toric.plot.PerfectMeasurements.Figure method), 31
new_properties() (qsurface.plot.Template2D method), 78
no_error (qsurface.codes._template.sim.PerfectMeasurements attribute), 17
Node (class in qsurface.decoders.ufns.elements), 63
nodes (qsurface.codes.elements.Edge attribute), 15
ns_delay() (qsurface.decoders.ufns.elements.Node method), 64
ns_parity() (qsurface.decoders.ufns.elements.Junction method), 65
ns_parity() (qsurface.decoders.ufns.elements.Node method), 64
ns_parity() (qsurface.decoders.ufns.elements.OddNode method), 65
ns_parity() (qsurface.decoders.ufns.elements.Syndrome method), 65

O

OddNode (class in qsurface.decoders.ufns.elements), 65
old_bound (qsurface.decoders.ufns.elements.Node attribute), 64
on_bound (qsurface.decoders.unionfind.elements.Cluster attribute), 56

P

parent (qsurface.decoders.unionfind.elements.Cluster attribute), 56
parity (qsurface.decoders.ufns.elements.Node attribute), 64
parity (qsurface.decoders.unionfind.elements.Cluster attribute), 56
parity_qubits (qsurface.codes.elements.AncillaQubit attribute), 14
peel_clusters() (qsurface.decoders.ufns.sim.Toric method), 69
peel_clusters() (qsurface.decoders.unionfind.sim.Toric method), 60
peel_leaf() (qsurface.decoders.ufns.sim.Toric method), 69
peel_leaf() (qsurface.decoders.unionfind.sim.Toric method), 60
PerfectMeasurements (class in qsurface.codes._template.plot), 21
PerfectMeasurements (class in qsurface.codes._template.sim), 16

PerfectMeasurements (*class in qsurface.codes.planar.plot*), 38
PerfectMeasurements (*class in qsurface.codes.planar.sim*), 34
PerfectMeasurements (*class in qsurface.codes.toric.plot*), 30
PerfectMeasurements (*class in qsurface.codes.toric.sim*), 26
PerfectMeasurements.Figure (*class in qsurface.codes._template.plot*), 22
PerfectMeasurements.Figure (*class in qsurface.codes.planar.plot*), 38
PerfectMeasurements.Figure (*class in qsurface.codes.toric.plot*), 30
phaseflip() (*qsurface.errors.pauli.Sim static method*), 46
place_bucket() (*qsurface.decoders.ufns.sim.Toric method*), 70
place_bucket() (*qsurface.decoders.unionfind.sim.Toric method*), 60
Planar (*class in qsurface.decoders.mwpm.plot*), 55
Planar (*class in qsurface.decoders.mwpm.sim*), 53
Planar (*class in qsurface.decoders.ufns.plot*), 70
Planar (*class in qsurface.decoders.ufns.sim*), 70
Planar (*class in qsurface.decoders.unionfind.plot*), 62
Planar (*class in qsurface.decoders.unionfind.sim*), 61
Planar.Figure2D (*class in qsurface.decoders.unionfind.plot*), 63
Planar.Figure3D (*class in qsurface.decoders.unionfind.plot*), 63
Plot (*class in qsurface.decoders._template*), 49
Plot (*class in qsurface.errors._template*), 43
Plot (*class in qsurface.errors.erasure*), 46
Plot (*class in qsurface.errors.pauli*), 46
plot_ancilla() (*qsurface.codes._template.plot.PerfectMeasurements method*), 21
plot_ancilla() (*qsurface.codes.planar.plot.PerfectMeasurements method*), 42
plot_ancilla() (*qsurface.codes.toric.plot.PerfectMeasurements method*), 33
plot_data() (*qsurface.codes._template.plot.PerfectMeasurements method*), 21
plot_data() (*qsurface.codes.planar.plot.PerfectMeasurements method*), 42
plot_data() (*qsurface.codes.toric.plot.PerfectMeasurements method*), 33
plot_data() (*qsurface.threshold.ThresholdFit method*), 13
plot_error() (*qsurface.errors._template.Plot method*), 45
plot_matching_edge() (*qsurface.decoders._template.Plot method*), 49
plot_params (*qsurface.errors._template.Plot attribute*), 45
PlotParams (*class in qsurface.plot*), 72
print_tree() (*in module qsurface.decoders.ufns.elements*), 65
pseudo_qubits (*qsurface.codes._template.sim.PerfectMeasurements attribute*), 16
PseudoEdge (*class in qsurface.codes.elements*), 16
PseudoQubit (*class in qsurface.codes.elements*), 16

Q

qsurface.codes.elements
module, 14
qsurface.decoders._template
module, 47
qsurface.decoders.mwpm
module, 50
qsurface.decoders.mwpm.plot
module, 55
qsurface.decoders.ufns
module, 63
qsurface.decoders.ufns.elements
module, 63
qsurface.decoders.unionfind
module, 55
qsurface.decoders.unionfind.elements
module, 55
qsurface.errors._template
module, 43
qsurface.errors.erasure
module, 46
qsurface.errors.pauli
module, 46
qsurface.main
module, 7
qsurface.plot
module, 72
qsurface.threshold
module, 11
Qubit (*class in qsurface.codes.elements*), 14

R

radius (*qsurface.decoders.ufns.elements.Node attribute*), 64
random_error() (*qsurface.errors._template.Sim method*), 43
random_error() (*qsurface.errors.erasure.Sim method*), 47
random_error() (*qsurface.errors.pauli.Sim method*), 46

random_errors() (*qsurface.codes._template.plot.PerfectMeasurements method*), 21
random_errors() (*qsurface.codes._template.sim.FaultyMeasurements method*), 19
random_errors() (*qsurface.codes._template.sim.PerfectMeasurements method*), 18
random_errors() (*qsurface.codes.planar.plot.PerfectMeasurements method*), 42
random_errors() (*qsurface.codes.planar.sim.FaultyMeasurements method*), 37
random_errors() (*qsurface.codes.planar.sim.PerfectMeasurements method*), 36
random_errors() (*qsurface.codes.toric.plot.PerfectMeasurements method*), 33
random_errors() (*qsurface.codes.toric.sim.FaultyMeasurements method*), 29
random_errors() (*qsurface.codes.toric.sim.PerfectMeasurements method*), 27
random_errors_layer() (*qsurface.codes._template.sim.FaultyMeasurements method*), 19
random_errors_layer() (*qsurface.codes.planar.sim.FaultyMeasurements method*), 38
random_errors_layer() (*qsurface.codes.toric.sim.FaultyMeasurements method*), 29
random_measure_layer() (*qsurface.codes._template.sim.FaultyMeasurements method*), 21
random_measure_layer() (*qsurface.codes.planar.sim.FaultyMeasurements method*), 38
random_measure_layer() (*qsurface.codes.toric.sim.FaultyMeasurements method*), 29
read_config() (*in module qsurface.decoders._template*), 47
read_csv() (*in module qsurface.threshold*), 13
reinitialized (*qsurface.codes.elements.DataQubit attribute*), 14
root_list (*qsurface.decoders.ufns.elements.Node attribute*), 64
run() (*in module qsurface.main*), 8
run_many() (*in module qsurface.threshold*), 11
run_multiprocess() (*in module qsurface.main*), 9

S

short (*qsurface.decoders.ufns.elements.Node attribute*), 64
show_corrected() (*qsurface.codes._template.plot.PerfectMeasurements method*), 21
show_corrected() (*qsurface.codes.planar.plot.PerfectMeasurements method*), 42
show_corrected() (*qsurface.codes.toric.plot.PerfectMeasurements method*), 34
Sim (*class in qsurface.decoders._template*), 48
Sim (*class in qsurface.errors._template*), 43
Sim (*class in qsurface.errors.erasure*), 46
Sim (*class in qsurface.errors.pauli*), 46
simulate() (*qsurface.codes._template.sim.FaultyMeasurements method*), 19
simulate() (*qsurface.codes.planar.sim.FaultyMeasurements method*), 38
simulate() (*qsurface.codes.toric.sim.FaultyMeasurements method*), 29
size (*qsurface.decoders.unionfind.elements.Cluster attribute*), 56
state (*qsurface.codes.elements.AncillaQubit attribute*), 14
state (*qsurface.codes.elements.DataQubit attribute*), 14
state (*qsurface.codes.elements.Edge attribute*), 15
static_forest() (*qsurface.decoders.ufns.sim.Toric method*), 70
static_forest() (*qsurface.decoders.unionfind.sim.Toric method*), 61
support (*qsurface.decoders.unionfind.elements.Cluster attribute*), 56
support (*qsurface.decoders.unionfind.sim.Toric attribute*), 57
Syndrome (*class in qsurface.decoders.ufns.elements*), 65
syndrome (*qsurface.codes.elements.AncillaQubit attribute*), 15

T

Template2D (*class in qsurface.plot*), 73
Template3D (*class in qsurface.plot*), 78
temporary_changes (*qsurface.plot.Template2D attribute*), 75
temporary_properties() (*qsurface.codes._template.plot.PerfectMeasurements.Figure method*), 24

temporary_properties() (*qsurface.codes.planar.plot.PerfectMeasurements.Figure method*), 40
temporary_properties() (*qsurface.codes.toric.plot.PerfectMeasurements.Figure method*), 32
temporary_properties() (*qsurface.plot.Template2D method*), 78
temporary_saved (*qsurface.plot.Template2D attribute*), 75
ThresholdFit (*class in qsurface.threshold*), 13
Toric (*class in qsurface.decoders.mwpm.plot*), 55
Toric (*class in qsurface.decoders.mwpm.sim*), 51
Toric (*class in qsurface.decoders.ufns.plot*), 70
Toric (*class in qsurface.decoders.ufns.sim*), 66
Toric (*class in qsurface.decoders.unionfind.plot*), 62
Toric (*class in qsurface.decoders.unionfind.sim*), 57
Toric.Figure2D (*class in qsurface.decoders.unionfind.plot*), 62
Toric.Figure3D (*class in qsurface.decoders.unionfind.plot*), 62
trivial_ancillas (*qsurface.codes._template.sim.PerfectMeasurements attribute*), 17

U

union() (*qsurface.decoders.unionfind.elements.Cluster method*), 57
union_bucket() (*qsurface.decoders.ufns.sim.Toric method*), 68
union_bucket() (*qsurface.decoders.unionfind.sim.Toric method*), 60
union_check() (*qsurface.decoders.ufns.sim.Toric method*), 70
union_check() (*qsurface.decoders.unionfind.sim.Toric method*), 60

V

value_to_list() (*qsurface.main.BenchmarkDecoder method*), 11
values (*qsurface.main.BenchmarkDecoder attribute*), 10

W

waited (*qsurface.decoders.ufns.elements.Node attribute*), 64
write_config() (*in module qsurface.decoders._template*), 47

Z

z_neighbors (*qsurface.codes.elements.AncillaQubit attribute*), 14