pyzx Documentation

Release 0.6.3

pyzx

Dec 02, 2020

CONTENTS:

CHAPTER

ONE

GETTING STARTED

PyZX can be installed as a package using pip:

```
pip install pyzx
```

If you wish to use the demo notebooks or benchmark circuits, then the repository can be cloned from Github.

The best way to get started if you have cloned the repository is to run the Getting Started notebook in Jupyter. This page contains the same general information as that notebook.

Warning: If you are using the pip installed version, please make sure it is version 0.6.0, and not 0.5.x as the api has changed considerably in between.

Warning: The newer JupyterLab as opposed to the older Jupyter Notebook uses a different framework for widgets which is currently not compatible with the widgets used in PyZX. It is therefore recommended that you use the classic notebook interface. If you are using JupyterLab you can find this interface by going to 'Help -> Launch Classic Notebook'.

Let's start by importing the library:

```
>>> import pyzx as zx
```

For all the examples in this documentation we will assume you have imported PyZX in this manner.

Quantum circuits in PyZX are represented by the *Circuit* class. File in the supported formats (QASM, QC, Quipper) can easily be imported into PyZX:

```
circuit = zx.Circuit.load("path/to/circuit.extension")
```

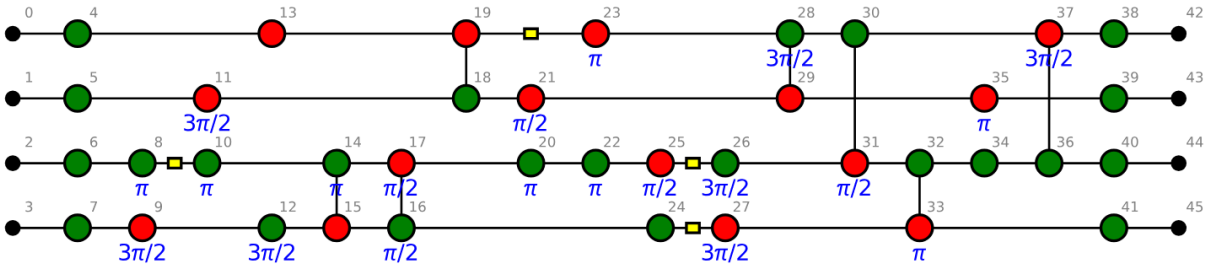
PyZX tries to automatically figure out in which format the circuit is represented. The *generate* module supplies several ways to generate random circuits:

```
>>> circuit = zx.generate.CNOT_HAD_PHASE_circuit(qubits=10,depth=20,clifford=True)
```

If you are running inside a Jupyter notebook, circuits can be easily visualized:

```
>>> zx.draw(circuit)
```

The default drawing method is to use the d3 javascript library. When not running in a Jupyter notebook `zx.draw` returns a matplotlib figure instead.



Most of the functionality of PyZX is based on the ZX-diagrams. These are represented by instances of *BaseGraph*. To convert a circuit into a ZX-diagram, simply do:

```
g = circuit.to_graph()
```

Let us use one of the built-in ZX-diagram simplification routines on this ZX-diagram:

```
>>> zx.clifford_simp(g)  # simplifies the diagram
>>> g.normalize()  # makes it more presentable
>>> zx.draw(g)
```

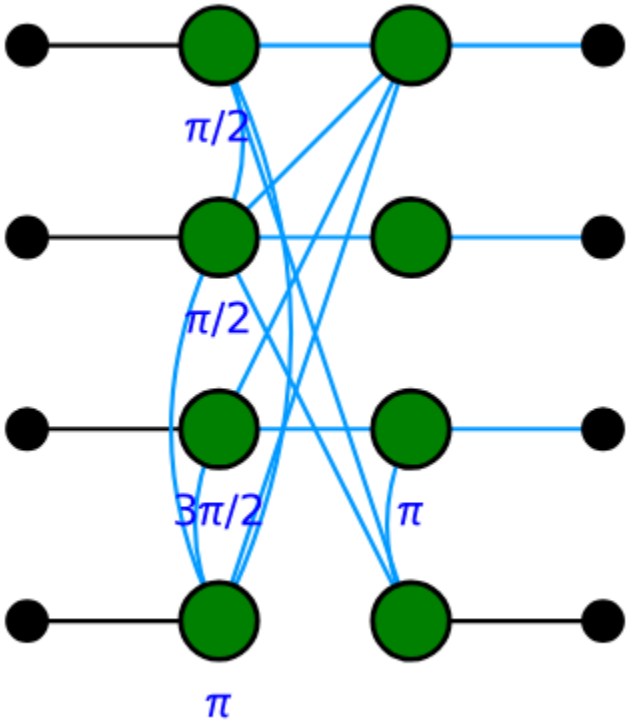


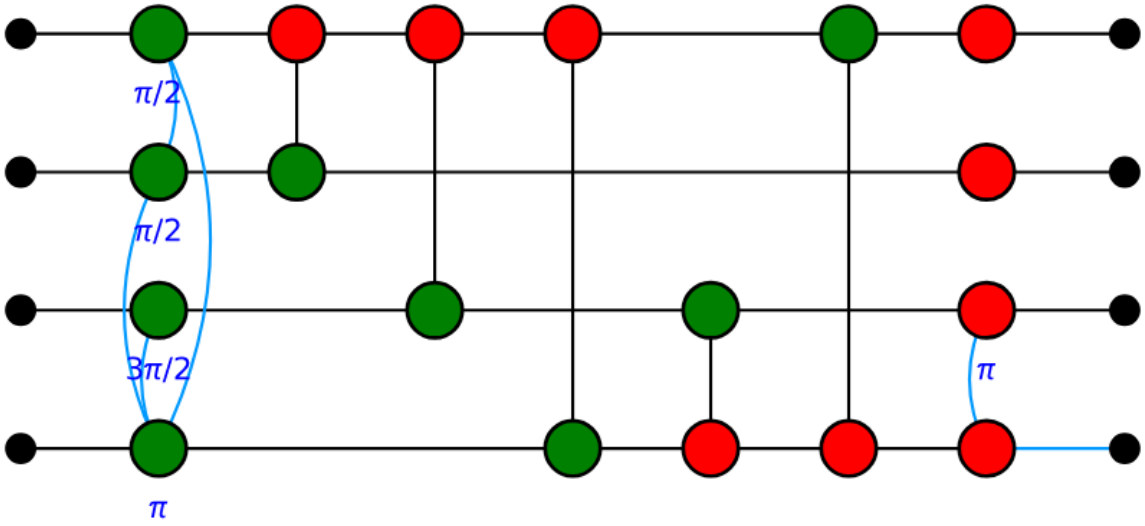
Fig. 1: The same circuit, but rewritten into a more compact form. The blue lines represent edges which have a Hadamard gate on them.

A ZX-diagram is represented internally as a graph:

```
>>> print(g)
Graph(16 vertices, 21 edges)
```

This simplified ZX-graph no longer looks like a circuit. PyZX supplies some methods for turning a ZX-graph back into a circuit:

```
>>> c = zx.extract_circuit(g.copy())
>>> zx.draw(c)
```



To verify that the simplified circuit is still equal to the original we can convert them to numpy tensors and compare equality directly:

```
>>> zx.compare_tensors(c,g,preserve_scalar=False)
        True
```

Note that a `Circuit` is not much more than just a series of gates:

```
>>> print(c.gates)
        [S(1), S*(2), Z(3), CZ(1,3), CZ(2,3), S(0), CZ(1,0), CZ(3,0), CNOT(1,0),␣
↪CNOT(2,0), CNOT(3,0), CNOT(2,3), CNOT(0,3), NOT(2), CX(2,3), HAD(3)]
```

We can convert circuits into one of several circuit description languages, such as that of QUIPPER:

```
>>> print(c.to_quipper())
Inputs: 0Qbit, 1Qbit, 2Qbit, 3Qbit
QGate["S"](1) with nocontrol
QGate["S"]*(2) with nocontrol
QGate["Z"](3) with nocontrol
QGate["Z"](3) with controls=[+1] with nocontrol
QGate["Z"](3) with controls=[+2] with nocontrol
QGate["S"](0) with nocontrol
QGate["Z"](0) with controls=[+1] with nocontrol
QGate["Z"](0) with controls=[+3] with nocontrol
QGate["not"](0) with controls=[+1] with nocontrol
QGate["not"](0) with controls=[+2] with nocontrol
QGate["not"](0) with controls=[+3] with nocontrol
```

(continues on next page)

(continued from previous page)

```
QGate["not"](3) with controls=[+2] with nocontrol
QGate["not"](3) with controls=[+0] with nocontrol
QGate["not"](2) with nocontrol
QGate["X"](3) with controls=[+2] with nocontrol
QGate["H"](3) with nocontrol
Outputs: 0Qbit, 1Qbit, 2Qbit, 3Qbit
```

Optimizing random circuits is of course not very useful, so let us do some optimization on a predefined circuit:

```
>>> c = zx.Circuit.load('circuits/Fast/mod5_4_before')  # Circuit.load auto-detects␣
↪the file format
>>> print(c.gates)  #  This circuit is built out of CCZ gates.
[NOT(4), HAD(4), CCZ(c1=0,c2=3,t=4), CCZ(c1=2,c2=3,t=4), HAD(4), CNOT(3,4), HAD(4),␣
↪CCZ(c1=1,c2=2,t=4), HAD(4), CNOT(2,4), HAD(4), CCZ(c1=0,c2=1,t=4), HAD(4), CNOT(1,
↪4), CNOT(0,4)]
>>> c = c.to_basic_gates()  #  Convert it to the Clifford+T gate set.
>>> print(c.gates)
[NOT(4), HAD(4), CNOT(3,4), T*(4), CNOT(0,4), T(4), CNOT(3,4), T*(4), CNOT(0,4), T(3),
↪ T(4), HAD(4), CNOT(0,3), T(0), T*(3), CNOT(0,3), HAD(4), CNOT(3,4), T*(4), CNOT(2,
↪4), T(4), CNOT(3,4), T*(4), CNOT(2,4), T(3), T(4), HAD(4), CNOT(2,3), T(2), T*(3),␣
↪CNOT(2,3), HAD(4), HAD(4), CNOT(3,4), HAD(4), CNOT(2,4), T*(4), CNOT(1,4), T(4),␣
↪CNOT(2,4), T*(4), CNOT(1,4), T(2), T(4), HAD(4), CNOT(1,2), T(1), T*(2), CNOT(1,2),␣
↪HAD(4), HAD(4), CNOT(2,4), HAD(4), CNOT(1,4), T*(4), CNOT(0,4), T(4), CNOT(1,4),␣
↪T*(4), CNOT(0,4), T(1), T(4), HAD(4), CNOT(0,1), T(0), T*(1), CNOT(0,1), HAD(4),␣
↪HAD(4), CNOT(1,4), CNOT(0,4)]
>>> print(c.stats())
Circuit mod5_4_before on 5 qubits with 71 gates.
        28 is the T-count
        43 Cliffords among which
        28 2-qubit gates and 14 Hadamard gates.
>>> g = c.to_graph()
>>> print(g)
Graph(109 vertices, 132 edges)
>>> zx.simplify.full_reduce(g)  # Simplify the ZX-graph
>>> print(g)
Graph(31 vertices, 38 edges)
>>> c2 = zx.extract_circuit(g).to_basic_gates()  # Turn graph back into circuit
>>> print(c2.stats())
Circuit  on 5 qubits with 42 gates.
        8 is the T-count
        34 Cliffords among which
        24 2-qubit gates and 10 Hadamard gates.
>>> c3 = zx.optimize.full_optimize(c2)  #  Do some further optimization on the circuit
>>> print(c3.stats())
Circuit  on 5 qubits with 27 gates.
        8 is the T-count
        19 Cliffords among which
        14 2-qubit gates and 2 Hadamard gates.
```

The circuit file-formats supported by `Circuit.load` are curently *qasm*, *qc* or *quipper*. PyZX can also be run from the command-line for some easy circuit-to-circuit manipulation. In order to optimize a circuit you can run the command:

```
python -m pyzx opt input_circuit.qasm
```

For more information regarding the command-line tools, run `python -m pyzx --help`.

This concludes this tutorial. For more information about the simplification procedures see *List of simplifications*. The different representations of the graphs and circuits is detailed in *Importing and exporting quantum circuits and ZX-diagrams*. How to create and modify ZX-diagrams is explained in *ZX-diagrams in PyZX and how to modify them*.

CHAPTER

TWO

OPTIMIZING AND SIMPLIFYING CIRCUITS

The main functionality of PyZX is the ability to optimize quantum circuits. The main optimization methods work by converting a circuit into a ZX-diagram, simplifying this diagram, and then converting it back into a quantum circuit. This process is explained in the next section. There are also some basic optimization methods that work directly on the quantum circuit representation. This is detailed in the section *Gate-level optimization*.

2.1 optimizing circuits using the ZX-calculus

PyZX allows the simplification of quantum circuits via a translation to the ZX-calculus. To demonstrate this functionality, let us generate a random circuit using *CNOT_HAD_PHASE_circuit*:

```
c = zx.generate.CNOT_HAD_PHASE_circuit(qubits=8, depth=100)
print(c.stats())
```

To use the ZX-diagram simplification routines, the circuit must first be converted to a ZX-diagram:

```
g = c.to_graph()
```

We can now use any of the built-in simplification strategies for ZX-diagrams. The most powerful of these is *full_reduce*:

```
zx.full_reduce(g, quiet=False) # simplifies the Graph in-place, and show the rewrite␣
↪steps taken.
g.normalize() # Makes the graph more suitable for displaying
zx.draw(g) # Display the resulting diagram
```

This rewrite strategy implements a variant of the algorithm described in this paper. The resulting diagram most-likely does not resemble the structure of a circuit. In order to extract an equivalent circuit from the diagram, we use the function *extract_circuit*.

Simply use it like so:

```
c_opt = zx.extract_circuit(g.copy())
```

For some circuits, *extract_circuit* can result in quite large circuits involving many CNOT gates. If one is only interested in optimizing the T-count of a circuit, the extraction stage can be skipped by using the *phase-teleportation* method of this paper. This applies `full_reduce` in such a way that only phases are moved around the circuit, and all other structure remains intact:

```
g = c.to_graph()
zx.teleport_reduce(g)
c_opt = zx.Circuit.from_graph(g) # This function is able to reconstruct a Circuit␣
↪from a Graph that looks sufficiently like a Circuit
```

2.2 Circuit equality verification

In order to verify that the simplified circuit is equal to the original, PyZX supplies two different methods. For circuits on a small number of qubits (generally less than 10) PyZX allows for the direct calculation of the linear maps that the circuits implement. These can then be checked for equality:

```
zx.compare_tensors(c,c_opt) # Returns True if c and c_opt implement the same circuit␣
↪(up to global phase)
```

You can also inspect the linear map itself by calling *c.to_matrix()*. For larger circuits calculating the linear map directly is not feasible. For those circuits PyZX allows you to check equality of the circuits using the built-in ZX-diagram rewrite strategy. This is done by composing one circuit with the adjoint of the other and simplifying the resulting circuit. If this is reducable to the identity, this is strong evidence that the circuits indeed implement the same unitary:

```
c.verify_equality(c_opt) # Returns True if full_reduce() is able to reduce the␣
↪composition of the circuits to the identity.
```

2.3 Gate-level optimization

Besides the advanced simplification strategies based on the ZX-calculus, PyZX also supplies some optimization methods that work directly on *Circuit*s. The most straightforward of these is *basic_optimization*.

A more advanced optimization technique involves splitting up the circuit into phase polynomial subcircuits, optimizing each of these, and then resynthesising the circuit, which can be done using *phase_block_optimize*.

The *basic_optimization* and *phase_block_optimize* functions are also combined into a single function *full_optimize*.

CHAPTER

THREE

ZX-DIAGRAMS IN PYZX AND HOW TO MODIFY THEM

ZX-diagrams are represented in PyZX by instances of the `BaseGraph` class, and are stored as simple graphs with some additional data on the vertices and edges. There are 4 different types of vertices: boundaries, Z-spiders, X-spiders and H-boxes. Boundary vertices represent an input or an output to the circuit and carry no further information. Z- and X-spider are the usual bread and butter of ZX-diagrams. H-boxes are used in ZH-diagrams as a generalisation of the Hadamard gate. Non-boundary vertices carry additional information in the form of a *phase*. This is a fraction `q` representing a phase `pi*q`.

As a simple example, we could have a graph with 3 vertices. The first being a boundary acting as input, the last being a boundary acting as output. If the middle one is a Z-vertex with phase `a` that is connected to both the input and output, then this graph represents a Z[`pi*a`]-phase gate.

Edges in a PyZX graph come in two flavors. The first is the default edge type which represents a regular connection. The second is a *Hadamard-edge*. This represents a connection between vertices with a Hadamard gate applied between them, and in the drawing functions of PyZX is represented by a blue edge.

3.1 Accessing and setting vertex and edge type

The type of a vertex `v` in a graph `g` can be retrieved by `g.type(v)`. This returns an integer representing the type. These integers are stored in `pyzx.utils.VertexType` and is one of the following:

- `VertexType.BOUNDARY`
- `VertexType.Z`
- `VertexType.X`
- `VertexType.H_BOX`

To get the type of all the vertices at once you call `g.types()`. This returns a dictionary-like object that maps vertices to their types. So for instance one can do the following:

```
ty = g.types()
if ty[vertex] == VertexType.BOUNDARY:
        #It is a boundary
```

Similarly, the type of an edge is stored as one of the integers `EdgeType.SIMPLE` or `EdgeType.HADAMARD`, where `EdgeType` can be found as `pyzx.utils.EdgeType`. The edge type of a given edge can be retrieved by `g.edge_type(edge)`.

3.2 Backends

ZX-graphs can be represented internally in different ways. The only fully functioning backend right now is *pyzx.graph.graph_s.GraphS*, which is written entirely in Python. A partial implementation using the `python-igraph` package is also available as `pyzx.graph.graph_ig.GraphIG`. A new backend can be constructed by subclassing *pyzx.graph.base.BaseGraph*.

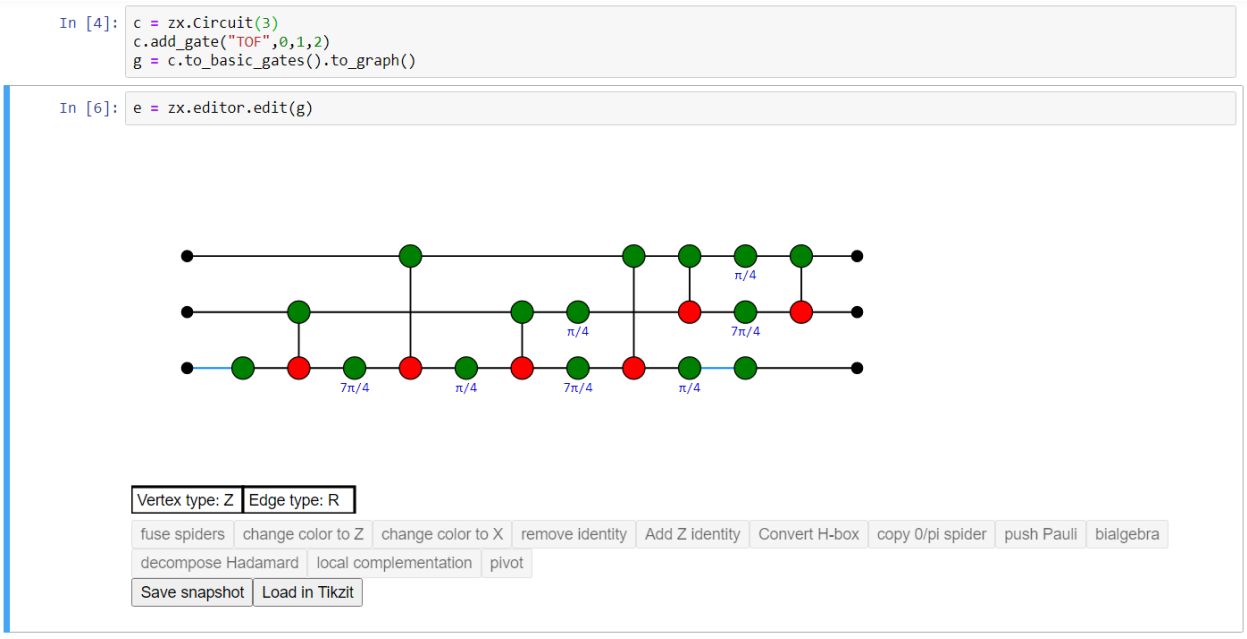
3.3 Creating and modifying ZX-diagrams

To create an empty ZX-diagram call `g=zx.Graph()`. You can then add a vertex and set its data by calling for example `v = g.add_vertex(zx.VertexType.Z, qubit=0, row=1, phase=1)`. To add an edge between vertices `v` and `w` you call `g.add_edge(g.edge(v,w),edgetype=zx.EdgeType.SIMPLE)`.

These functions are probably best used in some type of loop or function so that you don't have to set everything by hand. If you wish to create a ZX-diagram in the shape of a circuit it is probably better to use the *Circuit*, or if you don't care about the exact structure of the circuit, one of the functions in *generate*.

3.4 The ZX-diagram editor

If you are using a Jupyter notebook, probably the best way to build an arbitrarily shaped ZX-diagram is to use the built-in graphical editor. If you have a ZX-diagram `g`, call `e = zx.editor.edit(g)` to start a new editor instance. The output of the cell should be the editor, and should look something like this:



Warning: The newer JupyterLab as opposed to the older Jupyter Notebook uses a different framework for widgets which is currently not compatible with the widgets used in PyZX. For the editor to work you therefore must use the classic notebook interface. If you are using JupyterLab you can find this interface by going to 'Help -> Launch Classic Notebook'.

Ctrl-clicking (Command-clicking for Mac users) on the view of the graph will add a new vertex of the type specified by 'Vertex type' (so a Z-vertex in the above example). Click 'Vertex type' to change the type of vertex to be added, or with the editor window selected, use the hotkey 'X'.

Ctrl-drag (Command-drag) between two vertices to add a new edge of the type 'Edge type' (either Regular or Hadamard), or use the hotkey 'E' to switch. Adding an edge between vertices where there is already one present replaces the edge with the new one.

Drag a box around vertices to select them. With a set selected, drag your mouse on one of the vertices to move them. Press delete or backspace to delete your selection. You can also directly select an edge by clicking it.

Double-click a vertex to change its phase. You should enter a fraction possibly followed by pi. Example inputs: '1', '-1/2', '3/2pi'. An empty input gives the default value ('0' for Z/X spiders, '1' for H-boxes).

Any change can be undone by pressing Ctrl-Z, and redone by pressing Ctrl-Shift-Z.

Changes in the editor are automatically pushed to the underlying graph. So if we made the editor using the command `e = zx.editor.edit(g)` than any changes we make are automatically done to `g`. Alternatively, we can access the graph by `e.graph`. Outside of the editor we can also make changes to `g`. For instance, we can call `zx.spider_simp(g)` to fuse all the spiders in `g`. To view these changes in the editor, call `e.update()`.

With a set of vertices selected, you will see some of the buttons beneath the editor light up. Clicking these buttons will do the action it says on the graph. Each of these actions will preserve the semantics of your ZX-diagram (i.e. the linear map it implements).

Sometimes it is useful to see which linear map your ZX-diagram implements. If you create the editor with the command `e = zx.editor.edit(g,show_matrix=True)`, this will show a Latex-styled matrix beneath the editor with the linear map your ZX-diagram implements. This matrix is automatically updated after every change you make to the graph. Note that this only works if your ZX-diagram does not have too many inputs and outputs (at most 4). It automatically regards boundary vertices 'pointing right' as inputs, and boundary vertices 'pointing left' as outputs. You can change this manually by changing `g.inputs` and `g.outputs`.

If you click 'Save snapshot', a copy of the graph is saved in the list `e.snapshots`. When you press 'Load in Tikzit', all snapshots are loaded into a Tikz format parseable by Tikzit. In order to use this functionality you have to point `zx.settings.tikzit_location` to a valid executable.

CHAPTER

FOUR

IMPORTING AND EXPORTING QUANTUM CIRCUITS AND ZX-DIAGRAMS

There are several ways to import and export circuits and ZX-diagrams in PyZX.

4.1 Importing and exporting quantum circuits

There are a number of standards for representing quantum circuits that are supported in PyZX. To see if PyZX supports a certain file format, just call *load*:

```
circuit = zx.Circuit.load("path/to/circuit.extension")
```

The currently supported formats are

- QASM,
- the ASCII format of Quipper,
- the simple *.qc* format used for representing quantum circuits in LaTex,
- and the qsim format used by Google.

To convert a PyZX circuit to these formats, use *to_qasm*, *to_quipper*, *to_qc*.

PyZX also offers a convenience function to construct a circuit out of a string containing QASM code using either *from_qasm* or `qasm`.

To convert a Circuit into a PyZX Graph (i.e. a ZX-diagram), call the method *to_graph*.

4.2 Importing and exporting ZX-diagrams

A ZX-diagram in PyZX is represented as an instance of *Graph*. A ZX-diagram can be loaded using the `.qgraph` format that Quantomatic uses, via *from_json*. It can be converted into that format using *to_json*.

Apart from this reversible representation, there are also several one-way translations for exporting ZX-diagrams from PyZX. A graph can be exported to GraphML format using *to_graphml*. To export a ZX-diagram to tikz for easy importing to Latex, use *to_tikz*.

Additionally, PyZX diagrams can be directly exported into the applications Tikzit using the *tikzit* function or edited in Quantomatic using the function *edit_graph*.

Finally, to display a ZX-diagram in Jupyter call *draw* and to create a matplotlib picture of the ZX-diagram use *draw_matplotlib*.

Some ZX-diagrams can be converted into an equivalent circuit. For complicated ZX-diagrams, the function *extract_circuit* is supplied. For ZX-diagrams that come directly from Circuits, e.g. those produced by calling `c.to_graph` for a Circuit `c`, one can also use the static method *from_graph*, which is more lightweight.

CHAPTER

FIVE

FULL API DOCUMENTATION

Below is listed the documentation for all the supported functions, classes and methods in PyZX. Some functionality of PyZX is still experimental or not well-tested (like the ZH-diagram interface and rewrite rules), so it is not listed here.

5.1 Graph API

ZX-graphs are internally represented by instances of classes that implement the methods of *BaseGraph*. These methods are listed below. The only complete implementation currently is *GraphS*.

class GraphS
Purely Pythonic implementation of BaseGraph.

To create a graph of a specific backend a convenience method Graph is supplied:

Graph(*backend=None*)
Returns an instance of an implementation of *BaseGraph*. By default *GraphS* is used. Currently backend is allowed to be *simple* (for the default), or 'graph_tool' and 'igraph'. This method is the preferred way to instantiate a ZX-diagram in PyZX.

Example

To construct an empty ZX-diagram, just write:

```
g = zx.Graph()
```

Return type *BaseGraph*

Below you can find full documentation of all the functions supplied by a Graph in PyZX.

class BaseGraph
Base class for letting graph backends interact with PyZX. For a backend to work with PyZX, there should be a class that implements all the methods of this class. For implementations of this class see *GraphS* or GraphIG.

add_edge(*edge*, *edgetype=1*)
Adds a single edge of the given type

Return type None

add_edge_smart(*e*, *edgetype*)
Like add_edge, but does the right thing if there is an existing edge.

add_edge_table(*etab*)
Takes a dictionary mapping (source,target) –> (#edges, #h-edges) specifying that #edges regular edges must be added between source and target and $h-edges Hadamard edges. The method selectively adds or removes edges to produce that ZX diagram which would result from adding (#edges, #h-edges), and then removing all parallel edges using Hopf/spider laws.

Return type None

add_edges(*edges*, *edgetype=1*)
Adds a list of edges to the graph.

Return type None

add_to_phase(*vertex*, *phase*)
Add the given phase to the phase value of the given vertex.

Return type None

add_vertex(*ty=0*, *qubit=- 1*, *row=- 1*, *phase=None*)
Add a single vertex to the graph and return its index. The optional parameters allow you to respectively set the type, qubit index, row index and phase of the vertex.

Return type ~VT

add_vertices(*amount*)
Add the given amount of vertices, and return the indices of the new vertices added to the graph, namely: range(g.vindex() - amount, g.vindex())

Return type List[~VT]

adjoint()
Returns a new graph equal to the adjoint of this graph.

Return type *BaseGraph*

apply_effect(*effect*)
Inserts an effect into the outputs of the graph. effect should be a string with every character representing an output effect for each qubit. The possible types of effects are one of '0', '1', '+', '-' for the respective kets. If '/' is specified this output is skipped.

Return type None

apply_state(*state*)
Inserts a state into the inputs of the graph. state should be a string with every character representing an input state for each qubit. The possible types of states are on of '0', '1', '+', '-' for the respective kets. If '/' is specified this input is skipped.

Return type None

auto_detect_inputs()
DEPRECATED: alias for auto_detect_io

Return type Tuple[List[~VT], List[~VT]]

auto_detect_io()
Adds every vertex that is of boundary-type to the list of inputs or outputs. Whether it is an input or output is determined by looking whether its neighbor is further to the right or further to the left of the input. Inputs and outputs are sorted by vertical position. Raises an exception if boundary vertex does not have a unique neighbor or if this neighbor is on the same horizontal position.

Return type Tuple[List[~VT], List[~VT]]

compose(*other*)
Inserts a graph after this one. The amount of qubits of the graphs must match. Also available by the operator *graph1 + graph2*

Return type None

connected(*v1*, *v2*)
Returns whether vertices v1 and v2 share an edge.

Return type bool

copy(*adjoint=False*, *backend=None*)
Create a copy of the graph. If adjoint is set, the adjoint of the graph will be returned (inputs and outputs flipped, phases reversed). When backend is set, a copy of the graph with the given backend is produced. By default the copy will have the same backend.

Parameters

- **adjoint** (bool) – set to True to make the copy be the adjoint of the graph
- **backend** (Optional[str]) – the backend of the output graph

Return type *BaseGraph*

Returns A copy of the graph

Note: The copy will have consecutive vertex indices, even if the original graph did not.

---

depth()
Returns the value of the highest row number given to a vertex. This is -1 when no rows have been set.

Return type Union[float, int]

edge(*s*, *t*)
Returns the edge object with the given source/target.

Return type ~ET

edge_s(*edge*)
Returns the source of the given edge.

Return type ~VT

edge_set()
Returns the edges of the graph as a Python set. Should be overloaded if the backend supplies a cheaper version than this.

Return type Set[~ET]

edge_st(*edge*)
Returns a tuple of source/target of the given edge.

Return type Tuple[~VT, ~VT]

edge_t(*edge*)
Returns the target of the given edge.

Return type ~VT

edge_type(*e*)
Returns the type of the given edge: EdgeType.SIMPLE if it is regular, EdgeType.HADAMARD if it is a Hadamard edge, 0 if the edge is not in the graph.

Return type Literal[1, 2]

edges()
Iterator that returns all the edges. Output type depends on implementation in backend.

Return type Sequence[~ET]

classmethod from_json(*js*)
Converts the given .qgraph json string into a Graph. Works with the output of *to_json*.

Return type *BaseGraph*

classmethod from_tikz(*tikz*, *warn_overlap=True*, *fuse_overlap=True*, *ignore_nonzx=False*)
Converts a tikz diagram into a pyzx Graph. The tikz diagram is assumed to be one generated by Tikzit, and hence should have a nodelayer and a edgelayer..

Parameters

- **s** – a string containing a well-defined Tikz diagram.
- **warn_overlap** (bool) – If True raises a Warning if two vertices have the exact same position.
- **fuse_overlap** (bool) – If True fuses two vertices that have the exact same position. Only has effect if fuse_overlap is False.
- **ignore_nonzx** (bool) – If True suppresses most errors about unknown vertex/edge types and labels.

Warning: Vertices that might look connected in the output of the tikz are not necessarily connected at the level of tikz itself, and won't be treated as such in pyzx.

Return type *BaseGraph*

incident_edges(*vertex*)
Returns all neighboring edges of the given vertex.

Return type Sequence[~ET]

is_id()
Returns whether the graph is just a set of identity wires, i.e. a graph where all the vertices are either inputs or outputs, and they are connected to each other in a non-permuted manner.

Return type bool

neighbors(*vertex*)
Returns all neighboring vertices of the given vertex.

Return type Sequence[~VT]

normalize()
Puts every node connecting to an input/output at the correct qubit index and row.

Return type None

num_edges()
Returns the amount of edges in the graph

Return type int

num_vertices()
Returns the amount of vertices in the graph.

Return type int

pack_circuit_rows()
Compresses the rows of the graph so that every index is used.

Return type None

phase(*vertex*)
Returns the phase value of the given vertex.

Return type Union[Fraction, int]

phases()
Returns a mapping of vertices to their phase values.

Return type Mapping[~VT, Union[Fraction, int]]

qubit(*vertex*)
Returns the qubit index associated to the vertex. If no index has been set, returns -1.

Return type Union[float, int]

qubit_count()
Returns the number of inputs of the graph

Return type int

qubits()
Returns a mapping of vertices to their qubit index.

Return type Mapping[~VT, Union[float, int]]

remove_edge(*edge*)
Removes the given edge from the graph.

Return type None

remove_edges(*edges*)
Removes the list of edges from the graph.

Return type None

remove_isolated_vertices()
Deletes all vertices and vertex pairs that are not connected to any other vertex.

Return type None

remove_vertex(*vertex*)
Removes the given vertex from the graph.

Return type None

remove_vertices(*vertices*)
Removes the list of vertices from the graph.

Return type None

replace_subgraph(*left_row*, *right_row*, *replace*)
Deletes the subgraph of all nodes with rank strictly between left_row and right_row and replaces it with the graph replace. The amount of nodes on the left row should match the amount of inputs of the replacement graph and the same for the right row and the outputs. The graphs are glued together based on the qubit index of the vertices.

Return type None

row(*vertex*)
Returns the row that the vertex is positioned at. If no row has been set, returns -1.

Return type Union[float, int]

rows()
Returns a mapping of vertices to their row index.

Return type Mapping[~VT, Union[float, int]]

set_edge_type(*e*, *t*)
Sets the type of the given edge.

Return type None

set_phase(*vertex*, *phase*)
Sets the phase of the vertex to the given value.

Return type None

set_phase_master(*m*)
Points towards an instance of the class Simplifier. Used for phase teleportation.

Return type None

set_position(*vertex*, *q*, *r*)
Set both the qubit index and row index of the vertex.

set_qubit(*vertex*, *q*)
Sets the qubit index associated to the vertex.

Return type None

set_row(*vertex*, *r*)
Sets the row the vertex should be positioned at.

Return type None

set_type(*vertex*, *t*)
Sets the type of the given vertex to t.

Return type None

set_vdata(*vertex*, *key*, *val*)
Sets the vertex data associated to key to val.

Return type None

stats()

Return type str

Returns Returns a string with some information regarding the degree distribution of the graph.

tensor(*other*)
Take the tensor product of two graphs. Places the second graph below the first one. Can also be called using the operator graph1 @ graph2

Return type *BaseGraph*

to_graphml()
Returns a GraphML representation of the graph.

Return type str

to_json(*include_scalar=True*)
Returns a json representation of the graph that follows the Quantomatic .qgraph format. Convert back into a graph using *from_json*.

Return type str

to_matrix(*preserve_scalar=True*)
Returns a representation of the graph as a matrix using *tensorfy*

Return type ndarray

to_tensor(*preserve_scalar=True*)
Returns a representation of the graph as a tensor using *tensorfy*

Return type ndarray

to_tikz(*draw_scalar=False*)
Returns a Tikz representation of the graph.

Return type str

type(*vertex*)
Returns the type of the given vertex: VertexType.BOUNDARY if it is a boundary, VertexType.Z if it is a Z node, VertexType.X if it is a X node, VertexType.H_BOX if it is an H-box.

Return type Literal[0, 1, 2, 3]

types()
Returns a mapping of vertices to their types.

Return type Mapping[~VT, Literal[0, 1, 2, 3]]

update_phase_index(*old*, *new*)
When a phase is moved from a vertex to another vertex, we need to tell the phase_teleportation algorithm that this has happened. This function does that. Used in some of the rules in *simplify*.

Return type None

vdata(*vertex*, *key*, *default=0*)
Returns the data value of the given vertex associated to the key. If this key has no value associated with it, it returns the default value.

Return type Any

vdata_keys(*vertex*)
Returns an iterable of the vertex data key names. Used e.g. in making a copy of the graph in a backend-independent way.

Return type Sequence[str]

vertex_degree(*vertex*)
Returns the degree of the given vertex.

Return type int

vertex_set()
Returns the vertices of the graph as a Python set. Should be overloaded if the backend supplies a cheaper version than this.

Return type Set[~VT]

vertices()
Iterator over all the vertices.

Return type Sequence[~VT]

vindex()
The index given to the next vertex added to the graph. It should always be equal to max(g.vertices()) + 1.

Return type ~VT

5.2 Circuit API

class Circuit(*qubit_amount*, *name=''*)

Class for representing quantum circuits.

This class is mostly just a wrapper for a list of gates with methods for converting between different representations of a quantum circuit.

The methods in this class that convert a specification of a circuit into an instance of this class, generally do not check whether the specification is well-defined. If a bad input is given, the behaviour is undefined.

add_circuit(*circ*, *mask=None*)

Adds the gate of another circuit to this one. If `mask` is not given, then they must have the same amount of qubits and they are mapped one-to-one. If mask is given then it must be a list specifying to which qubits the qubits in the given circuit correspond.

Example:

```
c1 = Circuit(qubit_amount=4)
c2 = Circuit(qubit_amount=2)
c2.add_gate("CNOT",0,1)
c1.add_circuit(c2, mask=[0,3]) # Now c1 has a CNOT from the first to the last␣
↪qubit
```

If the circuits have the same amount of qubits then it can also be called as an operator:

```
c1 = Circuit(2)
c2 = Circuit(2)
c1 += c2
```

Return type `None`

add_gate(*gate*, **args*, ***kwargs*)

Adds a gate to the circuit. `gate` can either be an instance of a `Gate`, or it can be the name of a gate, in which case additional arguments should be given.

Example:

```
circuit.add_gate("CNOT", 1, 4) # adds a CNOT gate with control 1 and target 4
circuit.add_gate("ZPhase", 2, phase=Fraction(3,4)) # Adds a ZPhase gate on␣
↪qubit 2 with phase 3/4
```

Return type `None`

add_gates(*gates*, *qubit*)

Adds a series of single qubit gates on the same qubit. `gates` should be a space-separated string of gatenames.

Example:

```
circuit.add_gates("S T H T H", 1)
```

Return type `None`

static from_graph(*g*, *split_phases=True*)

Produces a *Circuit* containing the gates of the given ZX-graph. If the ZX-graph is not circuit-like then

the behaviour of this function is undefined. `split_phases` governs whether nodes with phases should be split into Z,S, and T gates or if generic ZPhase/XPhase gates should be used.

Return type *Circuit*

static from_qasm(*s*)
Produces a *Circuit* based on a QASM input string. It ignores all the non-unitary instructions like measurements in the file. It currently doesn't support custom gates that have parameters.

Return type *Circuit*

static from_qasm_file(*fname*)
Produces a *Circuit* based on a QASM description of a circuit. It ignores all the non-unitary instructions like measurements in the file. It currently doesn't support custom gates that have parameters.

Return type *Circuit*

static from_qc_file(*fname*)
Produces a *Circuit* based on a .qc description of a circuit. If a Toffoli gate with more than 2 controls is encountered, ancilla qubits are added. Currently up to 5 controls are supported.

Return type *Circuit*

static from_qsim_file(*fname*)
Produces a *Circuit* based on a .qc description of a circuit. If a Toffoli gate with more than 2 controls is encountered, ancilla qubits are added. Currently up to 5 controls are supported.

Return type *Circuit*

static from_quipper(*s*)
Produces a *Circuit* based on a Quipper ASCII description of a circuit.

Return type *Circuit*

static from_quipper_file(*fname*)
Produces a *Circuit* based on a Quipper ASCII description of a circuit.

Return type *Circuit*

static load(*circuitfile*)
Tries to detect the circuit description language from the filename and its contents, and then tries to load the file into a circuit.

Return type *Circuit*

prepend_gate(*gate*, **args*, ***kwargs*)
The same as add_gate, but adds the gate to the start of the circuit, not the end.

stats()
Returns statistics on the amount of gates in the circuit, separated into different classes (such as amount of T-gates, two-qubit gates, Hadamard gates).

Return type `str`

tcount()
Returns the amount of T-gates necessary to implement this circuit.

Return type `int`

tensor(*other*)
Takes the tensor product of two Circuits. Places the second one below the first. Can also be done as an operator: *circuit1 @ circuit2*.

Return type *Circuit*

to_basic_gates()
Returns a new circuit with every gate expanded in terms of X/Z phases, Hadamards and the 2-qubit gates CNOT, CZ, CX.

Return type *Circuit*

to_emoji()
Converts circuit into a representation that can be copy-pasted into the ZX-calculus Discord server.

Return type str

to_graph(*zh=False*, *compress_rows=True*, *backend=None*)
Turns the circuit into a ZX-Graph. If compress_rows is set, it tries to put single qubit gates on different qubits, on the same row.

Return type *BaseGraph*

to_matrix(*preserve_scalar=True*)
Returns a numpy matrix describing the circuit.

Return type ndarray

to_qasm()
Produces a QASM description of the circuit.

Return type str

to_qc()
Produces a .qc description of the circuit.

Return type str

to_quipper()
Produces a Quipper ASCII description of the circuit.

Return type str

to_tensor(*preserve_scalar=True*)
Returns a numpy tensor describing the circuit.

Return type ndarray

twoqubitcount()
Returns the amount of 2-qubit gates necessary to implement this circuit.

Return type int

verify_equality(*other*, *up_to_swaps=False*)
Composes the other circuit with the adjoint of this circuit, and tries to reduce it to the identity using simplify.full_reduce`. If successful returns True, if not returns None.

Note: A successful reduction to the identity is strong evidence that the two circuits are equal, if this function is not able to reduce the graph to the identity this does not prove anything.

Parameters

- **other** (*Circuit*) – the circuit to compare equality to.
- **up_to_swaps** (bool) – if set to True, only checks equality up to a permutation of the qubits.

Return type bool

5.3 Generating Circuits

The following are some methods to generate (random) quantum circuits.

CNOT_HAD_PHASE_circuit(*qubits*, *depth*, *p_had=0.2*, *p_t=0.2*, *clifford=False*)
Construct a Circuit consisting of CNOT, HAD and phase gates. The default phase gate is the T gate, but if `clifford=True`, then this is replaced by the S gate.

Parameters

- **qubits** (`int`) – number of qubits of the circuit
- **depth** (`int`) – number of gates in the circuit
- **p_had** (`float`) – probability that each gate is a Hadamard gate
- **p_t** (`float`) – probability that each gate is a T gate (or if `clifford` is set, S gate)
- **clifford** (`bool`) – when set to True, the phase gates are S gates instead of T gates.

Return type *Circuit*

Returns A random circuit consisting of Hadamards, CNOT gates and phase gates.

cliffordT(*qubits*, *depth*, *p_t=None*, *p_s=None*, *p_hsh=None*, *p_cnot=None*, *backend=None*)
Generates a circuit consisting of randomly placed Clifford+T gates. Optionally, take probabilities of adding T, S, HSH, and CNOT. If probabilities for only a subset of gates is given, any remaining probability will be uniformly distributed among the remaining gates.

Parameters

- **qubits** (`int`) – Amount of qubits in circuit.
- **depth** (`int`) – Depth of circuit.
- **p_t** (`Optional[float]`) – Probability that each gate is a T-gate.
- **p_s** (`Optional[float]`) – Probability that each gate is a S-gate.
- **p_hsh** (`Optional[float]`) – Probability that each gate is a HSH-gate.
- **p_cnot** (`Optional[float]`) – Probability that each gate is a CNOT-gate.
- **backend** (`Optional[str]`) – When given, should be one of the possible *Backends* backends.

Return type Instance of graph of the given backend.

cliffords(*qubits*, *depth*, *no_hadamard=False*, *t_gates=False*, *backend=None*)
Generates a circuit consisting of randomly placed Clifford gates. Uses a different approach to generating Clifford circuits then *cliffordT*.

Parameters

- **qubits** (`int`) – Amount of qubits in circuit.
- **depth** (`int`) – Depth of circuit.
- **no_hadamard** (`bool`) – Whether hadamard edges are allowed to be placed.
- **backend** (`Optional[str]`) – When given, should be one of the possible *Backends* backends.

Return type Instance of graph of the given backend.

cnots(*qubits*, *depth*, *backend=None*)
Generates a circuit consisting of randomly placed CNOT gates.

Args: qubits: Amount of qubits in circuit depth: Depth of circuit backend: When given, should be one of the possible *Backends* backends.

Return type *BaseGraph*

Returns Instance of graph of the given backend

identity(*qubits*, *depth=1*, *backend=None*)
Generates a `pyzx.graph.Graph` representing an identity circuit.

Parameters

- **qubits** (`int`) – number of qubits (i.e. parallel lines of the Graph)
- **depth** (`Union[float, int]`) – at which row the output vertices should be placed
- **backend** (`Optional[str]`) – the backend to use for the output graph

Return type *BaseGraph*

5.4 Circuit extraction and matrices over Z2

There is basically a single function that is needed for the most general extraction of a circuit from a ZX-diagram:

extract_circuit(*g*, *optimize_czs=True*, *optimize_cnots=2*, *quiet=True*)
Given a graph put into semi-normal form by *full_reduce*, it extracts its equivalent set of gates into an instance of *Circuit*. This function implements a more optimized version of the algorithm described in There and back again: A circuit extraction tale

Parameters

- **g** (*BaseGraph*[~VT, ~ET]) – The ZX-diagram graph to be extracted into a Circuit.
- **optimize_czs** (`bool`) – Whether to try to optimize the CZ-subcircuits by exploiting overlap between the CZ gates
- **optimize_cnots** (`int`) – (0,1,2,3) Level of CNOT optimization to apply.
- **quiet** (`bool`) – Whether to print detailed output of the extraction process.

Return type *Circuit*

This function uses some reasoning over matrices over the field Z2. This functionality is implemented in the following class.

class Mat2(*data*)
A matrix over Z2, with methods for multiplication, primitive row and column operations, Gaussian elimination, rank, and epi-mono factorisation.

col_add(*c0*, *c1*)
Add r0 to r1

Return type `None`

col_swap(*c0*, *c1*)
Swap the columns c0 and c1

Return type `None`

factor()
Produce a factorisation m = m0 * m1, where

m0.cols() = m1.rows() = m.rank()

Return type Tuple[*Mat2*, *Mat2*]

gauss(*full_reduce=False*, *x=None*, *y=None*, *blocksize=6*, *pivot_cols=[]*)
Compute the echelon form. Returns the number of non-zero rows in the result, i.e. the rank of the matrix.

The parameter 'full_reduce' determines whether to compute the full row-reduced form, useful e.g. for matrix inversion and CNOT circuit synthesis.

The parameter 'blocksize' gives the size of the blocks in a block matrix for performing Patel/Markov/Hayes optimization, see:

K. Patel, I. Markov, J. Hayes. Optimal Synthesis of Linear Reversible Circuits. QIC 2008

If blocksize is given as self.cols(), then this is equivalent to just eliminating duplicate rows before doing normal Gaussian elimination.

Contains two convenience parameters for saving the primitive row operations. Suppose the row-reduced form of m is computed as:

g * m = m'

Then, x –> g * x and y –> y * g^-1.

Note x and y need not be matrices. x can be any object that implements the method row_add(), and y any object that implements col_add().

Return type int

inverse()
Returns the inverse of m is invertible and None otherwise.

Return type Optional[*Mat2*]

nullspace(*should_copy=True*)
Returns a list of non-zero vectors that span the nullspace of the matrix. If the matrix has trivial kernel it returns the empty list.

Return type List[List[Literal[0, 1]]]

rank()
Returns the rank of the matrix.

Return type int

row_add(*r0*, *r1*)
Add r0 to r1

Return type None

row_swap(*r0*, *r1*)
Swap the rows r0 and r1

Return type None

solve(*b*)
Return a vector x such that M * x = b, or None if there is no solution.

Return type Optional[*Mat2*]

to_cnots(*optimize=False*)
Returns a list of CNOTs that implements the matrix as a reversible circuit of qubits.

Return type List[CNOT]

5.5 List of simplifications

Below is listed the content of simplify.py.

This module contains the ZX-diagram simplification strategies of PyZX. Each strategy is based on applying some combination of the rewrite rules in the *rules* module. The main procedures of interest are *clifford_simp* for simple reductions, *full_reduce* for the full rewriting power of PyZX, and *teleport_reduce* to use the power of *full_reduce* while not changing the structure of the graph.

simp(*g*, *name*, *match*, *rewrite*, *matchf=None*, *quiet=False*, *stats=None*)

Helper method for constructing simplification strategies based on the rules present in *rules*. It uses the match function to find matches, and then rewrites g using rewrite. If matchf is supplied, only the vertices or edges for which matchf() returns True are considered for matches.

Example

```
simp(g, 'spider_simp', rules.match_spider_parallel, rules.spider)
```

Parameters

- **g** (*BaseGraph*[~VT, ~ET]) – The graph that needs to be simplified.
- **name** (*str*) – The name to display if quiet is set to False.
- **match** (Callable[..., List[~MatchObject]]) – One of the match_* functions of *rules*.
- **rewrite** (Callable[[*BaseGraph*[~VT, ~ET], List[~MatchObject]], Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]]) – One of the rewrite functions of *rules*.
- **matchf** (Union[Callable[[~ET], bool], Callable[[~VT], bool], None]) – An optional filtering function on candidate vertices or edges, which is passed as the second argument to the match function.
- **quiet** (bool) – Suppress output on numbers of matches found during simplification.

Return type int

Returns Number of iterations of rewrite that had to be applied before no more matches were found.

bialg_simp(*g*, *quiet=False*, *stats=None*)

Return type int

clifford_simp(*g*, *quiet=True*, *stats=None*)

Keeps doing rounds of interior_clifford_simp and *pivot_boundary_simp* until they can't be applied anymore.

Return type int

full_reduce(*g*, *quiet=True*, *stats=None*)

The main simplification routine of PyZX. It uses a combination of *clifford_simp* and the gadgetization strategies *pivot_gadget_simp* and *gadget_simp*.

Return type None

gadget_simp(*g*, *quiet=False*, *stats=None*)

Return type int

id_simp(*g*, *matchf=None*, *quiet=False*, *stats=None*)

Return type int

lcomp_simp(*g*, *matchf=None*, *quiet=False*, *stats=None*)

Return type int

phase_free_simp(*g*, *quiet=False*, *stats=None*)
Performs the following set of simplifications on the graph: spider -> bialg

Return type int

pivot_boundary_simp(*g*, *matchf=None*, *quiet=False*, *stats=None*)

Return type int

pivot_gadget_simp(*g*, *matchf=None*, *quiet=False*, *stats=None*)

Return type int

pivot_simp(*g*, *matchf=None*, *quiet=False*, *stats=None*)

Return type int

reduce_scalar(*g*, *quiet=True*, *stats=None*)
Modification of full_reduce that is tailered for scalar ZX-diagrams. It skips the boundary pivots, and it additionally does supplementarity_simp.

Return type int

spider_simp(*g*, *matchf=None*, *quiet=False*, *stats=None*)

Return type int

supplementarity_simp(*g*, *quiet=False*, *stats=None*)

Return type int

tcount(*g*)
Returns the amount of nodes in g that have a non-Clifford phase.

Return type int

teleport_reduce(*g*, *quiet=True*, *stats=None*)
This simplification procedure runs *full_reduce* in a way that does not change the graph structure of the resulting diagram. The only thing that is different in the output graph are the location and value of the phases.

Return type *BaseGraph*[~VT, ~ET]

to_gh(*g*, *quiet=True*)
Turns every red node into a green node by changing regular edges into hadamard edges

Return type None

to_rg(*g*, *select=None*)
Turn green nodes into red nodes by color-changing vertices which satisfy the predicate select. By default, the predicate is set to greedily reducing the number of Hadamard-edges. :type g: *BaseGraph*[~VT, ~ET] :param g: A ZX-graph. :type select: Optional[Callable[[~VT], bool]] :param select: A function taking in vertices and returning True or False.

Return type None

5.6 List of rewrite rules

Below is listed the content of `rules.py`.

This module contains the implementation of all the rewrite rules on ZX-diagrams in PyZX.

Each rewrite rule consists of two methods: a matcher and a rewriter. The matcher finds as many non-overlapping places where the rewrite rule can be applied. The rewriter takes in a list of matches, and performs the necessary changes on the graph to implement the rewrite.

Each match function takes as input a Graph instance, and an optional "filter function" that tells the matcher to only consider the vertices or edges that the filter function accepts. It outputs a list of "match" objects. What these objects look like differs per rewrite rule.

The rewrite function takes as input a Graph instance and a list of match objects of the appropriate type. It outputs a 4-tuple (edges to add, vertices to remove, edges to remove, isolated vertices check). The first of these should be fed to *add_edge_table*, while the second and third should be fed to `remove_vertices` and *remove_edges*. The last parameter is a Boolean that when true means that the rewrite rule can introduce isolated vertices that should be removed by *remove_isolated_vertices*.

Dealing with this output is done using either *apply_rule* or *pyzx.simplify.simp*.

Warning: There is no guarantee that the matcher does not affect the graph, and currently some matchers do in fact change the graph. Similarly, the rewrite function also changes the graph other than through the output it generates (for instance by adding vertices or changes phases).

apply_copy(*g*, *matches*)

Return type `Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]`

apply_gadget_phasepoly(*g*, *matches*)
Uses the output of *match_gadgets_phasepoly* to apply a rewrite based on rule R_13 of the paper *A Finite Presentation of CNOT-Dihedral Operators*.

Return type `None`

apply_rule(*g*, *rewrite*, *m*, *check_isolated_vertices=True*)

Return type `None`

apply_supplementarity(*g*, *matches*)
Given the output of :func:`match_supplementarity`, removes non-Clifford spiders that act on the same set of targets trough supplementarity.

Return type `Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]`

bialg(*g*, *matches*)
Performs a certain type of bialgebra rewrite given matchings supplied by `match_bialg(_parallel)`.

Return type `Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]`

lcomp(*g*, *matches*)
Performs a local complementation based rewrite rule on the given graph with the given `matches` returned from `match_lcomp(_parallel)`. See *insert paper here* for more details on the rewrite

Return type `Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]`

match_bialg(*g*)
Does the same as *match_bialg_parallel* but with `num=1`.

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

match_bialg_parallel(*g*, *matchf=None*, *num=- 1*)
Finds noninteracting matchings of the bialgebra rule.

Parameters

- **g** (*BaseGraph*[~VT, ~ET]) – An instance of a ZX-graph.
- **matchf** (Optional[Callable[[~ET], bool]]) – An optional filtering function for candidate edge, should return True if a edge should considered as a match. Passing None will consider all edges.
- **num** (int) – Maximal amount of matchings to find. If -1 (the default) tries to find as many as possible.

Return type List of 4-tuples (v1, v2, neighbors_of_v1,neighbors_of_v2)

match_copy(*g*, *vertexf=None*)
Finds spiders with a 0 or pi phase that have a single neighbor, and copies them through. Assumes that all the spiders are green and maximally fused.

Return type List[Tuple[~VT, ~VT, Union[Fraction, int], Union[Fraction, int], List[~VT]]]

match_gadgets_phasepoly(*g*)
Finds groups of phase-gadgets that act on the same set of 4 vertices in order to apply a rewrite based on rule R_13 of the paper *A Finite Presentation of CNOT-Dihedral Operators*.

Return type List[Tuple[List[~VT], Dict[FrozenSet[~VT], Union[~VT, Tuple[~VT, ~VT]]]]]

match_ids(*g*)
Finds a single identity node. See *match_ids_parallel*.

Return type List[Tuple[~VT, ~VT, ~VT, Literal[1, 2]]]

match_ids_parallel(*g*, *vertexf=None*, *num=- 1*)
Finds non-interacting identity vertices.

Parameters

- **g** (*BaseGraph*[~VT, ~ET]) – An instance of a ZX-graph.
- **num** (int) – Maximal amount of matchings to find. If -1 (the default) tries to find as many as possible.
- **vertexf** (Optional[Callable[[~VT], bool]]) – An optional filtering function for candidate vertices, should return True if a vertex should be considered as a match. Passing None will consider all vertices.

Return type List of 4-tuples (identity_vertex, neighbor1, neighbor2, edge_type).

match_lcomp(*g*)
Same as *match_lcomp_parallel*, but with num=1

Return type List[Tuple[~VT, List[~VT]]]

match_lcomp_parallel(*g*, *vertexf=None*, *num=- 1*, *check_edge_types=True*)
Finds noninteracting matchings of the local complementation rule.

Parameters

- **g** (*BaseGraph*[~VT, ~ET]) – An instance of a ZX-graph.

- **num** (int) – Maximal amount of matchings to find. If -1 (the default) tries to find as many as possible.
- **check_edge_types** (bool) – Whether the method has to check if all the edges involved are of the correct type (Hadamard edges).
- **vertexf** (Optional[Callable[[~VT], bool]]) – An optional filtering function for candidate vertices, should return True if a vertex should be considered as a match. Passing None will consider all vertices.

Return type List of 2-tuples `(vertex, neighbors)`.

match_phase_gadgets(*g*)
Determines which phase gadgets act on the same vertices, so that they can be fused together.

Parameters **g** (*BaseGraph*[~VT, ~ET]) – An instance of a ZX-graph.

Return type List of 5-tuples `(axel,leaf, total combined phase, other axels with same targets, other leafs)`.

match_pivot(*g*)
Does the same as *match_pivot_parallel* but with `num=1`.

Return type List[Tuple[~VT, ~VT, List[~VT], List[~VT]]]

match_pivot_boundary(*g*, *matchf=None*, *num=- 1*)
Like *match_pivot_parallel*, but except for pairings of Pauli vertices, it looks for a pair of an interior Pauli vertex and a boundary non-Pauli vertex in order to gadgetize the non-Pauli vertex.

Return type List[Tuple[~VT, ~VT, List[~VT], List[~VT]]]

match_pivot_gadget(*g*, *matchf=None*, *num=- 1*)
Like *match_pivot_parallel*, but except for pairings of Pauli vertices, it looks for a pair of an interior Pauli vertex and an interior non-Clifford vertex in order to gadgetize the non-Clifford vertex.

Return type List[Tuple[~VT, ~VT, List[~VT], List[~VT]]]

match_pivot_parallel(*g*, *matchf=None*, *num=- 1*, *check_edge_types=True*)
Finds non-interacting matchings of the pivot rule.

Parameters

- **g** (*BaseGraph*[~VT, ~ET]) – An instance of a ZX-graph.
- **num** (int) – Maximal amount of matchings to find. If -1 (the default) tries to find as many as possible.
- **check_edge_types** (bool) – Whether the method has to check if all the edges involved are of the correct type (Hadamard edges).
- **matchf** (Optional[Callable[[~ET], bool]]) – An optional filtering function for candidate edge, should return True if a edge should considered as a match. Passing None will consider all edges.

Return type List of 4-tuples. See *pivot* for the details.

match_spider(*g*)
Does the same as *match_spider_parallel* but with `num=1`.

Return type List[Tuple[~VT, ~VT]]

match_spider_parallel(*g*, *matchf=None*, *num=- 1*)
Finds non-interacting matchings of the spider fusion rule.

Parameters

- **g** (*BaseGraph*[~VT, ~ET]) – An instance of a ZX-graph.
- **matchf** (Optional[Callable[[~ET], bool]]) – An optional filtering function for candidate edge, should return True if the edge should be considered for matchings. Passing None will consider all edges.
- **num** (int) – Maximal amount of matchings to find. If -1 (the default) tries to find as many as possible.

Return type List of 2-tuples (v1, v2)

match_supplementarity(*g*)

Finds pairs of non-Clifford spiders that are connected to exactly the same set of vertices.

Parameters **g** (*BaseGraph*[~VT, ~ET]) – An instance of a ZX-graph.

Return type List of 4-tuples (vertex1, vertex2, type of supplementarity, neighbors).

merge_phase_gadgets(*g*, *matches*)

Given the output of :func:match_phase_gadgets, removes phase gadgets that act on the same set of targets.

Return type Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]

pivot(*g*, *matches*)

Perform a pivoting rewrite, given a list of matches as returned by match_pivot(_parallel). A match is itself a list where:

m[0] : first vertex in pivot. m[1] : second vertex in pivot. m[2] : list of zero or one boundaries adjacent to m[0]. m[3] : list of zero or one boundaries adjacent to m[1].

Return type Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]

remove_ids(*g*, *matches*)

Given the output of match_ids(_parallel), returns a list of edges to add, and vertices to remove.

Return type Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]

spider(*g*, *matches*)

Performs spider fusion given a list of matchings from match_spider(_parallel)

Return type Tuple[Dict[~ET, List[int]], List[~VT], List[~ET], bool]

unspider(*g*, *m*, *qubit=- 1*, *row=- 1*)

Undoes a single spider fusion, given a match m. A match is a list with 3 elements given by:

```
m[0] : a vertex to unspider
m[1] : the neighbors of the new node, which should be a subset of the
       neighbors of m[0]
m[2] : the phase of the new node. If omitted, the new node gets all of the phase␣
↪of m[0]
```

Returns the index of the new node. Optional parameters qubit and row can be used to position the new node. If they are omitted, they are set as the same as the old node.

Return type ~VT

5.7 List of optimization functions

Below is listed the content of `optimize.py`.

This module implements several optimization methods on `Circuits`. The function *basic_optimization* runs a set of back-and-forth gate commutation and cancellation routines. *phase_block_optimize* does phase polynomial optimization using the TODD algorithm, and *full_optimize* combines these two methods.

basic_optimization(*circuit*, *do_swaps=True*, *quiet=True*)

Optimizes the circuit using a strategy that involves delayed placement of gates so that more matches for gate cancellations are found. Specifically tries to minimize the number of Hadamard gates to improve the effectiveness of phase-polynomial optimization techniques.

Parameters

- **circuit** (*Circuit*) – Circuit to be optimized.
- **do_swaps** (bool) – When set uses some rules transforming CNOT gates into SWAP gates. Generally leads to better results, but messes up architecture-aware placement of 2-qubit gates.
- **quiet** (bool) – Whether to print some progress indicators.

Return type *Circuit*

full_optimize(*circuit*, *quiet=True*)

Optimizes the circuit using first some basic commutation and cancellation rules, and then a dedicated phase polynomial optimization strategy involving the TODD algorithm.

Parameters

- **circuit** (*Circuit*) – Circuit to be optimized.
- **quiet** (bool) – Whether to print some progress indicators.

Return type *Circuit*

phase_block_optimize(*circuit*, *pre_optimize=True*, *quiet=True*)

Optimizes the given circuit, by cutting it into phase polynomial pieces, and using the TODD algorithm to optimize each of these phase polynomials. The phase-polynomial circuits are then resynthesized using the parity network algorithm.

Note: Only works with Clifford+T circuits. Will give wrong output when fed smaller rotation gates, or Toffoli-like gates. Depending on the number of qubits and T-gates this function can take a long time to run. It can be sped up somewhat by using the TOpt implementation of TODD. If this is installed, point towards it using `zx.settings.topt_command`, such as for instance `zx.settings.topt_command = ['wsl', '../TOpt']` for running it in the Windows Subsystem for Linux.

Parameters

- **circuit** (*Circuit*) – The circuit to be optimized.
- **pre_optimize** (bool) – Whether to call *basic_optimization* first.
- **quiet** (bool) – Whether to print some progress indicators. Helpful when execution time is long.

Return type *Circuit*

5.8 Functions for dealing with tensors

Below is listed the content of `tensor.py`.

This module provides methods for converting ZX-graphs into numpy tensors and using these tensors to test semantic equality of ZX-graphs. This module is not meant as an efficient quantum simulator. Due to the way the tensor is calculated it can only handle circuits of small size before running out of memory on a regular machine. Currently, it can reliably transform 9 qubit circuits into tensors. If the ZX-diagram is not circuit-like, but instead has nodes with high degree, it will run out of memory even sooner.

adjoint(*t*)

Returns the adjoint of the tensor as if it were representing a circuit:

```
t = tensorfy(circ)
tadj = tensorfy(circ.adjoint())
compare_tensors(adjoint(t),tadj) # This is True
```

Return type ndarray

compare_tensors(*t1*, *t2*, *preserve_scalar=False*)

Returns true if `t1` and `t2` represent equal tensors. When *preserve_scalar* is False (the default), equality is checked up to nonzero rescaling.

Example: To check whether two ZX-graphs *g1* and *g2* are semantically the same you would do:

```
compare_tensors(g1,g2) # True if g1 and g2 represent the same linear map up to␣
↪nonzero scalar
```

Return type bool

compose_tensors(*t1*, *t2*)

Returns a tensor that is the result of composing the tensors together as if they were representing circuits:

```
t1 = tensorfy(circ1)
t2 = tensorfy(circ2)
circ1.compose(circ2)
t3 = tensorfy(circ1)
t4 = compose_tensors(t1,t2)
compare_tensors(t3,t4) # This is True
```

Return type ndarray

find_scalar_correction(*t1*, *t2*)

Returns the complex number `z` such that `t1 = z*t2`.

> **Warning:** This function assumes that `compare_tensors(t1,t2,preserve_scalar=False)` is True, i.e. that `t1` and `t2` indeed are equal up to global scalar. If they aren't, this function returns garbage.

Return type complex

is_unitary(*g*)

Returns whether the given ZX-graph is equal to a unitary (up to a number).

Return type bool

tensor_to_matrix(*t*, *inputs*, *outputs*)
Takes a tensor generated by `tensorfy` and turns it into a matrix. The `inputs` and `outputs` arguments specify the final shape of the matrix: 2^(outputs) x 2^(inputs)

Return type `ndarray`

tensorfy(*g*, *preserve_scalar=True*)
Takes in a Graph and outputs a multidimensional numpy array representing the linear map the ZX-diagram implements. Beware that quantum circuits take exponential memory to represent.

Return type `ndarray`

5.9 Drawing

Below is listed the content of `drawing.py`.

arrange_scalar_diagram(*g*)

Return type `None`

draw(*g*, *labels=False*, ***kwargs*)
Draws the given Circuit or Graph. Depending on the value of `pyzx.settings.drawing_backend` either uses matplotlib or d3 to draw.

Return type `Any`

draw_d3(*g*, *labels=False*, *scale=None*, *auto_hbox=None*, *show_scalar=False*)

Return type `Any`

draw_matplotlib(*g*, *labels=False*, *figsize=(8, 2)*, *h_edge_draw='blue'*, *show_scalar=False*, *rows=None*)

Return type `Any`

matrix_to_latex(*m*)
Converts a matrix into latex code. Useful for pretty printing the matrix of a Circuit/Graph.

Example

Run this in a Jupyter notebook from ipywidgets import Label c = zx.Circuit(3) display(Label(matrix_to_latex(c.to_matrix())))

Return type `str`

print_matrix(*m*)
Returns a Label() Jupyter widget that displays a pretty latex representation of the given matrix. Instead of a matrix, can also give a Circuit or Graph.

Return type Label

5.10 Tikz and Quantomatic functionality

Below is listed the content of `tikz.py`.

Supplies methods to convert ZX-graphs to tikz files. These tikz files are designed to be easily readable by the program Tikzit.

tikz_to_graph(*s*, *warn_overlap=True*, *fuse_overlap=True*, *ignore_nonzx=False*, *backend=None*)

Converts a tikz diagram into a pyzx Graph. The tikz diagram is assumed to be one generated by Tikzit, and hence should have a nodelayer and a edgelayer..

Parameters

- **s** (`str`) – a string containing a well-defined Tikz diagram.
- **warn_overlap** (`bool`) – If True raises a Warning if two vertices have the exact same position.
- **fuse_overlap** (`bool`) – If True fuses two vertices that have the exact same position. Only has effect if fuse_overlap is False.
- **ignore_nonzx** (`bool`) – If True suppresses most errors about unknown vertex/edge types and labels.
- **backend** (`Optional[str]`) – Backend of the graph returned.

> **Warning:**
>
> **Vertices that might look connected in the output of the tikz are not necessarily connected** at the level of tikz itself, and won't be treated as such in pyzx.

Return type *BaseGraph*

tikzit(*g*, *draw_scalar=False*)

Opens Tikzit with the graph `g` opened as a tikz diagram. For this to work, `zx.settings.tikzit_location` must be pointed towards the Tikzit executable. Even though this function is intended to be used with Tikzit, `zx.tikz.tikzit_location` can point towards any executable that takes a tikz file as an input, such as a text processor.

Return type `None`

to_tikz(*g*, *draw_scalar=False*)

Converts a ZX-graph `g` to a string representing a tikz diagram.

Return type `str`

to_tikz_sequence(*graphs*, *draw_scalar=False*, *maxwidth=10*)

Given a list of ZX-graphs, outputs a single tikz diagram with the graphs presented in a grid. `maxwidth` is the maximum width of the diagram, before a graph is put on a new row in the tikz diagram.

Return type `str`

Below is listed the content of `quantomatic.py`.

Implements methods for interacting with Quantomatic:

```
import pyzx as zx
zx.settings.quantomatic_location = "path/to/quantomatic/jar/file.jar"
g = zx.generate.cliffordT(3,10,0.2)
```

(continues on next page)

(continued from previous page)

```
g2 = zx.quantomatic.edit_graph(g) # Opens Quantomatic with the graph g opened.␣
↪Execution is blocked until Quantomatic is closed again.
# If you have saved the qgraph file in quantomatic, then g2 should now contain your␣
↪changes.
```

edit_graph(*g*)

Opens Quantomatic with the graph `g` loaded. When you are done editing the graph, you save it in Quantomatic and close the executable. The resulting graph is returned by this function. Note that this function blocks until the Quantomatic executable is closed. For this function to work you must first set `zx.settings.quantomatic_location` to point towards the Quantomatic .jar file.

Return type *BaseGraph*

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

G

I

L

M

N

O

P

Q

R

S

T

U

V