PyZX: Large Scale Automated Diagrammatic Reasoning

Aleks Kissinger
University of Oxford
aleks.kissinger@cs.ox.ac.uk

John van de Wetering
Radboud University
john@vdwetering.name

The ZX-calculus is a graphical language for reasoning about ZX-diagrams, a type of tensor networks that can represent arbitrary linear maps between qubits. Using the ZX-calculus, we can intuitively reason about quantum theory, and optimise and validate quantum circuits. In this paper we introduce PyZX, an open source library for automated reasoning with large ZX-diagrams. We give a brief introduction to the ZX-calculus, then show how PyZX implements methods for circuit optimisation, equality validation, and visualisation and how it can be used in tandem with other software. We end with a set of challenges that when solved would enhance the utility of automated diagrammatic reasoning.

1 Introduction

The ZX-calculus was introduced in 2008 by Coecke and Duncan [8, 9]. Since then, much progress has been made on understanding this language: rule-sets have been found that make the language complete [19, 17, 33]; variants of the language have been developed that allow easy reasoning about different domains, e.g. ZW for quantum entanglement [10, 16] as well as Fermionic quantum computing [31] and ZH for Toffoli-Hadamard circuits [5]; and a diverse set of applications have been found including measurement-based quantum computing [13, 26], surface code lattice surgery [6] and circuit optimisation [14, 25]. However, in order to bring these applications into the practical domain, we need tools that allow us to reason with ZX-diagrams on a large scale.

PyZX (pronounced like 'physics' without the H) is a Python-based library designed for reasoning with large quantum circuits and ZX-diagrams. PyZX is Free and Open Source Software, licensed under GPLv3. The project is hosted on GitHub and available at:

https://github.com/Quantomatic/pyzx

It allows users to efficiently rewrite ZX-diagrams using built-in simplification strategies. These strategies are powerful enough to reduce Clifford diagrams to a pseudo-normal form [14], perform highly effective T-count optimisation [25], and to verify correctness of optimised circuits. Besides the powerful set of rewrite strategies, PyZX offers many ways to present and visualise circuits and ZX-diagrams. It can input and output circuits in the QASM, QC, and Quipper formats, and it can convert ZX-diagrams into TikZ diagrams or the Quantomatic qGraph format.

PyZX is best within Jupyter Notebooks [20], a popular way to run Python interactively with rich input and output features. In this setting ZX-diagrams can be visualised on the fly. PyZX can also be called as a command-line tool for circuit-to-circuit optimisation, e.g. for optimising many circuits or combining with other procedures. See Figure 1 for an overview.

As much of the theory behind PyZX has already been detailed elsewhere [14, 25], we will focus in this paper on the PyZX tool itself. We describe the capabilities of PyZX and several ways in which PyZX can be used, and we give some of the implementation details. We end the paper by offering several
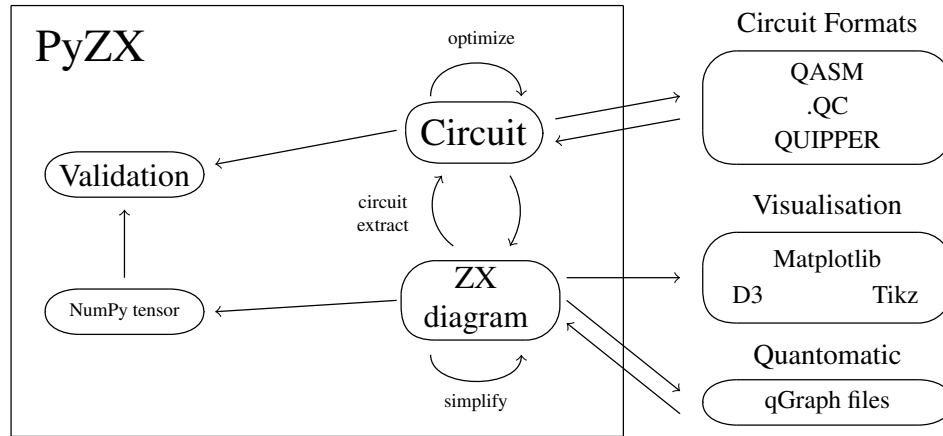
Figure 1: An overview of the functionality of PyZX.

challenges to the community, each of which when completed would expand the power of automated diagrammatic reasoning, or would help in understanding its limitations.

We start by giving a brief overview of the ZX-calculus in Section 2. Then, in Section 3 we describe the core functionality of PyZX and give some details about how it is implemented. In Section 4 we detail how PyZX interacts with TikZ and Quantomatic. We discuss the performance of PyZX in Section 5, and finally, in Section 6 we present a series of challenges for ourselves and other potential contributors/collaborators.

2   The ZX-calculus

This section provides a brief overview of the ZX-calculus. For an in-depth reference see Ref. [11].

The ZX-calculus is a diagrammatic language similar to the familiar quantum circuit notation. A *ZX-diagram* (or simply *diagram*) consists of *wires* and *spiders*. Wires entering the diagram from the left are *inputs*; wires exiting to the right are *outputs*. Given two diagrams we can compose them by joining the outputs of the first to the inputs of the second, or form their tensor product by simply stacking the two diagrams.

Spiders are linear operations which can have any number of input or output wires. There are two varieties: *Z* spiders depicted as green dots and *X* spiders depicted as red dots:[1]

$$\vcenter{\hbox{\includegraphics{z-spider}}} := |0...0\rangle\langle0...0| + e^{i\alpha}|1...1\rangle\langle1...1| \qquad \vcenter{\hbox{\includegraphics{x-spider}}} := |+...+\rangle\langle+...+| + e^{i\alpha}|-...-\rangle\langle-...-|$$

The diagram as a whole corresponds to a linear map built from the spiders (and SWAPs represented by wire crossings) by the usual composition and tensor product of linear maps. As a special case, diagrams with no inputs represent (unnormalised) state preparations.

It will be convenient to introduce a symbol – a yellow square – for the Hadamard gate. This is defined by the equation:

$$-\blacksquare- = -\left(\tfrac{\pi}{2}\right)\!\!\left(\tfrac{\pi}{2}\right)\!\!\left(\tfrac{\pi}{2}\right)- \tag{1}$$

---

[1]If you are reading this document in monochrome or otherwise have difficulty distinguishing green and red, *Z* spiders will appear lightly-shaded and *X* darkly-shaded.

We will not view this Hadamard gate as an additional generator. Instead, we consider a Hadamard box between two spiders as a different type of edge. We illustrate this by writing a Hadamard between two spiders as a blue dashed edge:

$$\tag{2}$$

Both the blue edge notation and the Hadamard box can always be translated back into spiders when necessary. We will refer to the blue edge as a *Hadamard edge*.

Two diagrams are considered *equal* when one can be deformed to the other by moving the vertices around in the plane, bending, unbending, crossing, and uncrossing wires, so long as the connectivity and the order of the inputs and outputs is maintained.

Additionally, there is a set of equations that we call the *rules* of the ZX-calculus that allow us to equate additional diagrams; these are shown in Figure 2. Equal diagrams correspond to equal linear maps. In this paper we will ignore global scalar factors, and hence we will consider linear maps A and B to be "equal" if they satisfy $A = zB$ for some non-zero $z \in \mathbb{C}$. The rules of the calculus can be made exact with respect to scalar factors [4], but we do not consider this possibility in this paper.

Quantum circuits can be translated into ZX-diagrams in a straightforward manner. We will take as our starting point circuits constructed from the following universal set of gates:

$$\text{CNOT} := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad Z_\alpha := \begin{pmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{pmatrix} \qquad H := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

This gate set allows a convenient representation in terms of spiders:

$$\text{CNOT} = \qquad Z_\alpha = \qquad H = \tag{3}$$

Note that the green spider is the first (i.e. control) qubit of the CNOT and the red spider is the second (i.e. target) qubit. Other common gates can easily be expressed in terms of these gates. In particular, $S := Z_{\frac{\pi}{2}}$, $T := Z_{\frac{\pi}{4}}$ and:
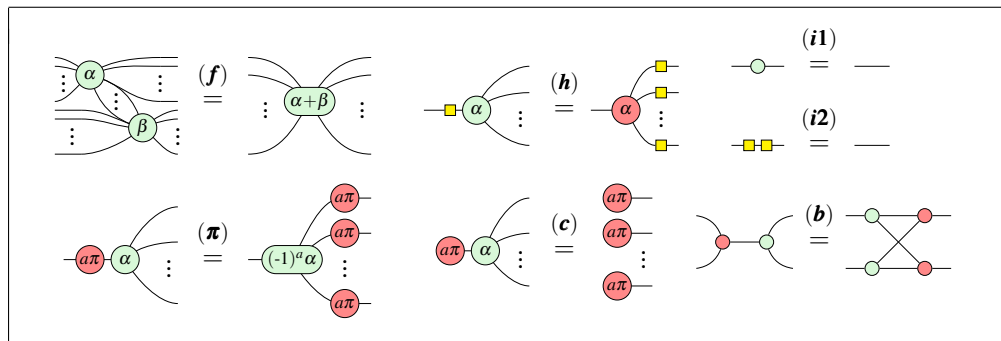
$$X_\alpha = \qquad = \qquad \text{CZ} = \qquad = \qquad = \tag{4}$$

Figure 2: A convenient presentation for the ZX-calculus. These rules hold for all $\alpha, \beta \in [0, 2\pi)$, and due to (**h**) and (**i2**) all rules also hold with the colours interchanged.

When we want to write a quantum circuit as a ZX-diagram, we simply convert each of the gates of the circuit into their equivalent ZX-diagram, and compose these per the structure of the circuit. For circuits using gates not described above, such as the Toffoli gate, we first convert the gate into a Clifford+T representation, and then we write it as a ZX-diagram.

3   Core functionality

In this section we detail the main features of PyZX and how they are implemented. Our intention is to give a brief tour of the capabilities of PyZX, rather than to give a detailed tutorial. Full documentation is available at `pyzx.readthedocs.io`.

PyZX is written in pure Python with the only dependencies being NumPy [3] and Matplotlib [2]. A relatively easy way to get these dependencies (and Jupyter) is to use Anaconda [1].

3.1   ZX-diagrams and circuits

There are two main data-structures present in PyZX: `Circuits` and `Graphs`. A `Circuit` is basically a wrapper around a list of gates, while a `Graph` represents a ZX-diagram.

The `Circuit` class is the entry-point for importing and exporting circuits to and from PyZX. It also provides methods to do gate-level operations, such as converting a Toffoli-circuit into a Clifford+T circuit or taking the adjoint of all its gates. There is also a variety of circuit optimisation schemes that act directly on the `Circuit` class which can be found in the `optimize` sub-module.

A `Circuit` consists of a list of `Gates`, which in turn are small classes containing some information about the gate and how to convert it into various representations, such as ZX-diagrams or the QASM [12] format.

The `Graph` class is more interesting. The graphs in PyZX are simple, undirected graphs with typed vertices and edges. Vertices come in three types: boundaries, Z-spiders and X-spiders. Each vertex can be labelled by a phase that is stored as a fraction representing a rational multiple of π. For instance, a label of $\frac{1}{2}$ corresponds to a phase of $\frac{\pi}{2}$.

The edges come in two types, which correspond respectively to a regular connection between spiders, and a Hadamard-edge as shown in Eq. (2). As ZX-diagrams allow parallel edges between spiders and self-loops, we need a way to deal with these in PyZX graphs. Adding an edge where there is already one present (via the `add_edge` method) will simply replace it. Often, it is more convenient to use the rules of the ZX-calculus to resolve parallel edges and self-loops whenever a new edge is added. How this should be done depends on the types of edges and vertices involved:
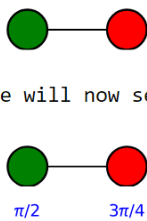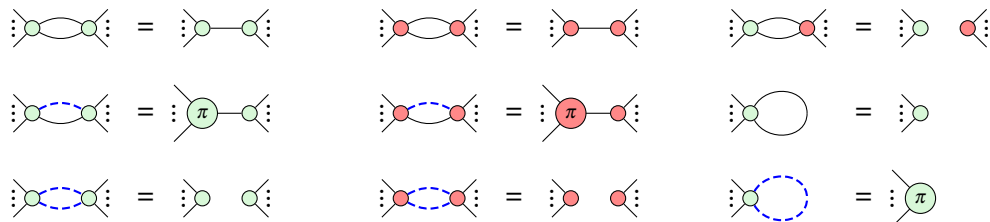

In order to use this built-in functionality to resolve the edges, one can use the `Graph` method `add_edge_table`, which takes in a list of edges and edge-types, and resolves what the ZX-diagram would look like when all the double edges and self-loops have been resolved.

The `Graph` class abstracts away some of the details of the underlying representation of the graph. As a result, a different implementation of a graph can simply subclass the `BaseGraph` class in order to work with all the other functionality of PyZX.

The default, pure Python, implementation of a graph in PyZX, `GraphS`, stores its connectivity as a dictionary of dictionaries, where the first level has vertices as keys (identified by an integer), with the values being another dictionary containing all the neighbours of this vertex. The values of these dictionaries in turn specify by which type of vertex the vertices are connected. Phases are stored in another dictionary. We have experimented with other graph back-ends, but we found that this simple implementation is fast enough to handle and simplify diagrams with hundreds of thousands of vertices in a reasonable time-frame. See Section 5 for more details.

Figure 3 gives a small example of creating a ZX-diagram in PyZX and visualising it. This example takes place inside a Jupyter Notebook [20]. A Jupyter Notebook is an interactive Python shell that runs in a web-browser and allows one to easily rerun parts of your code, and to add interactive HTML and Javascript widgets. We use Jupyter as a GUI to PyZX, using its integration with Matplotlib to visualise ZX-diagrams.

```
In [113]: import pyzx as zx  #  This is the recommended way to import PyZX
          g = zx.Graph() # We construct an empty ZX-diagram
          v = g.add_vertex(1,0,0)  #  We add a single Z-spider at position (0,0)
          w = g.add_vertex(2,0,1)  #  We add a single X-spider at position (0,1)
          g.add_edge((v,w))        #  We connect the two spiders together
          display(zx.draw(g,figsize=(1,0.3))) #  Draw the ZX-diagram onto the screen
          print("We will now set the phase of the spiders to a different value:")
          from fractions import Fraction    #   For this we need the Fraction class
          g.set_phase(v,Fraction(1,2))      # The phase is set as a multiple of pi
          g.set_phase(w,Fraction(3,4))
          display(zx.draw(g,figsize=(1,0.3)))
```

We will now set the phase of the spiders to a different value:

$\pi/2$ $3\pi/4$

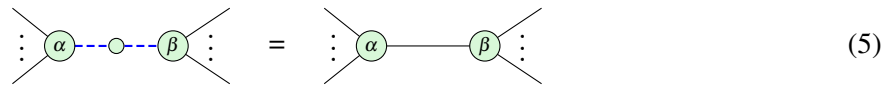Figure 3: Simple example of construction a ZX-diagram from scratch and visualising it.

Usually you will not want to manually create a ZX-diagram. There are various ways to import ZX-diagrams into PyZX. Directly importing ZX-diagrams is possible using the Quantomatic format (see Section 4), but PyZX can also read files describing quantum circuits in a variety of languages. It currently supports QASM [12], the Quipper ASCII format [32] and the QC/TFC format used by the Reversible Circuit Benchmarks page [29]. A circuit can be imported by calling for instance `circ = zx.Circuit.load("circuit.extension")`. The file-format is automatically inferred. As the circuit might contain gates that have no straightforward translation to ZX-diagrams, we do not automatically convert the imported circuit into a ZX-diagram. The method `to_graph()` transforms a `Circuit` into a `Graph` by converting the circuit into a Clifford+phases representation that is easily translated to a ZX-diagram.

3.2   Simplification and Verification

The simplification strategies for ZX-diagrams in PyZX are built in a hierarchical manner.

At the bottom level there are *rules* that consist of a *matcher* and a *rewriter*. A matcher loops over all the vertices or edges of a graph (which one depends on the rule) and tries to find as many non-overlapping sections of the graph that are suitable for the rule to be applied. Once it finishes, it returns a list of matches (what such a match object looks like differs per rule). The rewriter takes this list of matches and figures out which changes to the graph need to be made. Since all the matches are non-overlapping, this can be done for each match separately. If these changes involve adding vertices or changing phases, then this is done immediately, but in order to reduce overhead, other changes (adding edges or removing edges and vertices) are collected until the rewriter has processed all matches, and then implemented simultaneously.

For a simple example of a rule, let us consider *identity removal*:

$$\vcenter{\hbox{:}}\,\fbox{α}\text{---}\circ\text{---}\fbox{β}\,\vcenter{\hbox{:}} \quad = \quad \vcenter{\hbox{:}}\,\fbox{α}\text{------}\fbox{β}\,\vcenter{\hbox{:}} \tag{5}$$

The matcher loops over vertices and tries to find those vertices that have a phase of zero, and exactly two neighbours. When it has found such a vertex, it removes its neighbours from the list of candidates (in order to prevent overlapping applications), and it makes it into a match that contains the vertex, its neighbours, and the type of the edge that should be made between these neighbours in the rewritten graph (in the case above, since both edges to the middle vertex were Hadamard edges, the resulting edge is a regular edge, cf. (**i2**)). The rewriter takes this list of matches, builds a list of vertices to be removed, and edges to be added, and applies this all at once. As the neighbours of the removed vertex could already have been connected in the original graph, this could lead to a self-loop that is handled in the way described in Section 3.1.

The next stage in the hierarchy of simplification strategies are the *basic simplifiers*. These are built on a single rule. A simplifier keeps applying the matcher and rewriter of a single rule, until the matcher finds no more matches. Because a matcher only finds non-overlapping rules, and thereby might miss possible applications for the rule the first time around, and because the rewritten diagram might produce new sections of the graph that are suitable for application of the rule, the simplifier might need to do the process of matching and rewriting many times before no new matches are found. In order for the simplifier to not get stuck in an infinite loop, it is important that the rule actually simplifies the diagram in some manner, i.e. that some kind of metric on the graph is reduced. In the case of the basic simplifiers used in PyZX, this metric is usually that the amount of vertices in the graph is reduced, but it could also be something more complex such as trading one type of vertex for another one.

Let us consider the identity removal simplifier (named `id_simp` in PyZX). Its simplifier removes all the identity spiders in the diagram as in (5). The rule never generates new arity 2 zero phase spiders, hence the only way in which the simplifier has to run multiple times is when there were multiple identities in a row, i.e. when the rule applications had overlap. Since every application of the rule removes a vertex, this simplifier indeed terminates.

At the top level of the hierarchy are the *compound simplifiers*. These simply combine other simplifiers into a more complicated simplification strategy, by applying other simplifiers in some particular order, potentially looping over this combination until none of the simplifiers finds any more reductions.

For instance, PyZX also implements a basic simplifier that fuses spiders together (cf. (***f***)), called `spider_simp`. We could combine `id_simp` and `spider_simp` together in a compound simplifier as follows:

```
def fuse_simp(g):
    i = 0
    while True:
        i1 = id_simp(g)      # simplifiers return the amount of iterations
        i2 = spider_simp(g)
        if i1 == 0 and i2 == 0: break  # No more matches were found
        i += 1
    return i
```

This simplifier simply keeps removing identities and fusing spiders until this is no longer possible. Note that removing an identity might make more possibilities for fusing spiders, and fusing spiders might make a zero phase arity two spider. Hence, we indeed need the loop in this function to make full use of these simplifiers.

The rewrite strategies of PyZX are included in the `simplify` sub-package. It includes the previously described simplification strategies, but also more complex rewrite strategies based on the *local complementation* and *pivoting* procedures of Ref. [14]. These rewrite strategies are combined into two top level compound rewrite strategies: `zx.simplify.clifford_simp` implements the rewrite strategy of Ref. [14] that is able to reduce Clifford circuits to GS-LC (graph-state with local Cliffords) pseudo-normal form, while `zx.simplify.full_reduce` builds on that by implementing the strategy of Ref. [25] that is able to do significant T-count optimisation. For a demonstration of the usage of the rewrite system see Figure 4.
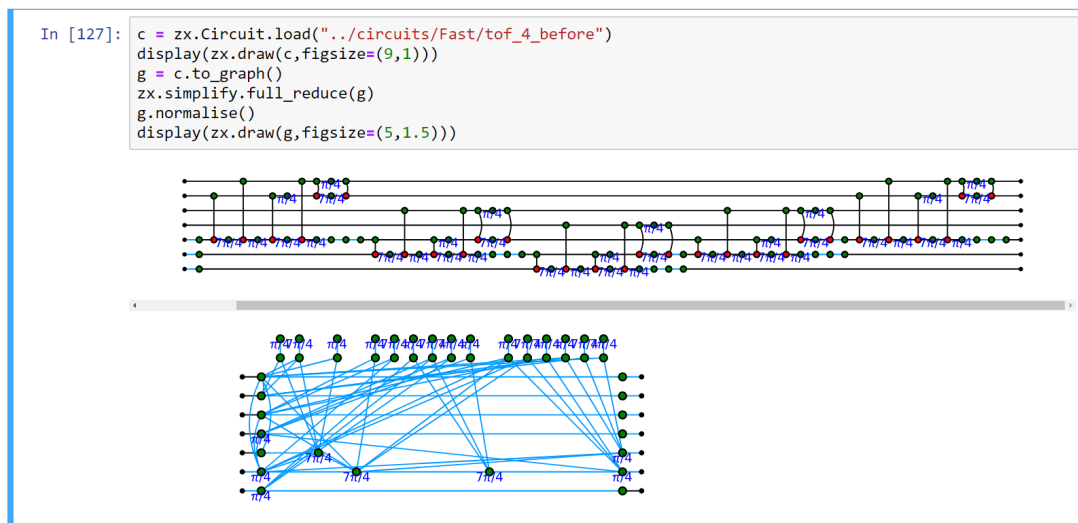

Figure 4: Example of simplifying a circuit with PyZX. The circuit `tof_4_before` is from Ref. [30]. The function `g.normalise()` does not change the ZX-diagram, but changes the coordinates of the vertices in order to make it more suitable for drawing. Note that the blue wires of the bottom diagram correspond to the Hadamard edges of (2).

It is of course essential to know whether the rewrite rules are implemented correctly and actually preserve the semantics of the diagram. PyZX offers a straightforward way to do this by allowing ZX-diagrams to be converted into their underlying linear maps using NumPy [3]. To test equality of diagrams

we can then simply test whether all the elements in the two tensors are equal up to a global non-zero scalar:

```
t1 = g1.to_tensor()
t2 = g2.to_tensor()
zx.compare_tensors(t1,t2)
```

This method uses an exponential amount of memory in terms of the qubits, or in the case of general ZX-diagrams: some function of the amount of inputs and the maximal arity of the spiders involved. In practice it can validate circuits up to 10 qubits on a 32-bit version of Python.

Interestingly, we can also use the rewriting engine itself to validate equality of diagrams or circuits. Given circuits (or just any ZX-diagrams) c1 and c2 we construct the circuit c as:

```
c = c1.adjoint()
c.add_circuit(c2)
```

If c1 and c2 indeed implement the same unitary, then c should implement the identity. To test the equality of c to the identity, we try to simplify it as much as possible. If we can successfully reduce it to the identity circuit, then we know (at least with very high likelihood) that c is indeed the identity. The function `c1.verify_equality(c2)`implements this idea using the strategy of Ref. [25] (i.e. `zx.simplify.full_reduce`). This method is fast (for circuits with dozens of qubits and thousands of gates, it finishes in seconds), and if equality is verified, it could in principle even give a certificate of this equality in terms of a list of rewrite rules. But if this method fails to reduce the circuit to the identity, then the answer is inconclusive, so that the utility of this method is determined by the variety of circuits it can verify.

We have run this validation method on the original and optimised circuits of Ref. [30] and managed to validate equality of all pairs of circuits except for one, where it turned out that the optimised circuit contained an error. Based on the ability of `full_reduce` to validate correctness of all these circuits, we speculate that whenever two ZX-diagrams can be transformed into one another by a combination of the rules of Figure 2, that `full_reduce` should be able to verify equality of these circuits.

3.3 Circuit-to-circuit optimisation

As can be seen in Figure 4, when a circuit-like ZX-diagram is fed into the simplification procedure, the simplified diagram might no longer resemble the structure of a circuit. If one is interested in using these simplifications for circuit optimisation, the resulting diagram should then be converted back into a circuit somehow. This is known as the *circuit extraction problem* [14]. To convert a ZX-diagram into a circuit in PyZX one can use the function `zx.extract.streaming_extract`. This method uses the proven results regarding circuit extraction from Ref. [14], but builds on that with a series of additional heuristics to extract a circuit from a ZX-diagram. There is currently no results known about when these heuristics suffice to extract a circuit, but when the diagram is produced from a circuit by `zx.simplify.full_reduce` then the extraction always seems to succeed[2]. Alternatively, PyZX also implements the *phase teleportation* routine from Ref. [25] that allows one to optimise circuits 'in place', side-stepping the issue of circuit extraction. This method is implemented in `zx.simplify.teleport_reduce`.

---

[2]We have tested the extraction on thousands of randomly generated circuits of varying sizes, next to dozens of standard benchmark circuits. In every case, our extraction heuristic successfully terminates.

Next to these optimisation methods that use the ZX-calculus, PyZX also implements some straight circuit-to-circuit optimisation methods. In particular, it contains an implementation of the TODD phase-polynomial optimiser of Ref. [18].

Instead of doing these operations manually inside a Python shell, PyZX also offers a command-line interface, e.g. by calling `python -m pyzx opt input_circuit`.

4   TikZiT and Quantomatic integration

Besides the core functionality based on the rewrite engine, PyZX offers a couple of convenience features for interacting with other software. Any ZX-diagram in PyZX can be exported as a TikZ figure for easy importing into a Latex document. This is how many of the bigger diagrams in Ref. [14] were produced. The TikZ output is compatible with the TikZ figure editing software TikZiT [21] and in fact, PyZX allows direct export into TikZiT:

```
zx.tikz.tikzit_location = 'path/to/tikzit.executable'
zx.tikz.tikzit(g)  # This opens Tikzit with the ZX-diagram g loaded
```

Alternatively, a circuit can be converted to a TikZ ZX-diagram by a command-line call: `python -m pyzx tikz input_circuit`.

PyZX also integrates with the graphical theorem prover Quantomatic [27] in various ways. The file-format that Quantomatic uses to store diagrams, *qGraph*, is supported by PyZX, allowing non-circuit ZX-diagrams to be stored and read. Furthermore, when Quantomatic is installed, it can be used as a GUI for editing ZX-diagrams:

```
zx.quantomatic.quantomatic_location = '/path/to/quantomatic.jar'
g2 = zx.quantomatic.edit_graph(g)  # Open Quantomatic with diagram g loaded
# The above function blocks until Quantomatic is closed
# Now g2 contains whatever changes were made to g in Quantomatic
```

The rewrite strategies of PyZX are also available from inside Quantomatic, by importing PyZX as a Python module. For more details, we refer to the documentation [24].

5   Performance

PyZX is able to handle and simplify ZX-diagrams containing tens of thousands of vertices in seconds; See Figure 5. As can be seen, reducing a 15 qubit circuit with 2500 gates takes less then 5 seconds. Interestingly, the time it takes to bring a Clifford circuit to normal form seems to follow a power-law, although the constant of the power-law depends on the amount of qubits in the circuit.

Let us compare this to a, on the surface, similar piece of software: Quantomatic. Like PyZX, Quantomatic supports automated reasoning with ZX-diagrams [27]. There are however two important differences.

The first is that Quantomatic is designed to be agnostic concerning the particular graphical language that is being used. As a result, particular behaviour that is 'hard-coded' into PyZX, such as the removal of parallel edges between vertices, needs to be implemented as a separate rule in Quantomatic, slowing down the way in which graphs can be simplified.

The second difference is that the rewrite engine of Quantomatic is based on *bang-boxes* [23]. Whenever a rule allows a variable amount of edges or vertices to be involved, it must be implemented in

```
In [2]: x,y = [],[]
        for i in range(1,251):
            gate_count = 30+i*10; x.append(gate_count)
            g = zx.generate.CNOT_HAD_PHASE_circuit(qubits=15, gates=gate_count, p_had=0.1, p_t=0.15,clifford=True).to_graph()
            t = time.time(); zx.simplify.clifford_simp(g,quiet=True)
            y.append(time.time()-t)
            if i%10 == 0: print(i,end='. ')

        10. 20. 30. 40. 50. 60. 70. 80. 90. 100. 110. 120. 130. 140. 150. 160. 170. 180. 190. 200. 210. 220. 230. 240. 250.
```

```
In [3]: plt.style.use('seaborn-whitegrid')
        x = np.array(x); y = np.array(y)
        plt.scatter(x, y,color='grey')  # Plot all the datapoints
        c = np.polyfit(np.log(x),np.log(y),1, w=np.sqrt(y))  # Generate power law fit
        a = np.linspace(0,max(x),100); b = (math.e**c[1])*a**c[0]
        plt.plot(a,b,'-',color='red',label="$y = a\cdot x^{{{:.2f}}}$".format(c[0]))  # Plot the fit
        plt.ylim(ymin=0); plt.xlim(xmin=0,xmax=max(x)+50);
        plt.xlabel("Gate-count"); plt.ylabel("Time [seconds]"); plt.legend(loc='upper left');
```
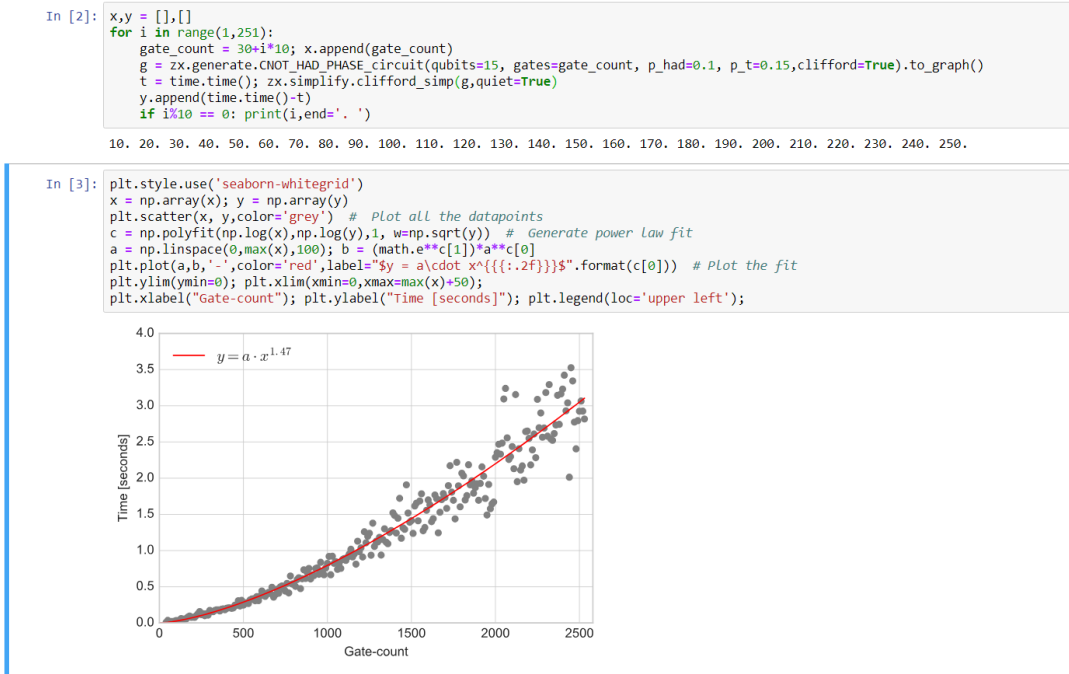

Figure 5: Benchmark of the Clifford simplification procedure that reduces circuits to normal form, as implemented in a Jupyter Notebook. The red-line is the least squares best-fit for a power-law. This was run on a consumer laptop.

Quantomatic using bang-boxes. While a powerful tool, finding matches involving bang-boxes can be slow. As every rule in PyZX is a piece of hand-written Python code, it does not have to rely on the strict logic of bang-boxes. While this necessitates more validation to make sure the implementation is correct, the result is that PyZX can be much faster.

In Ref. [15] Quantomatic was used to optimise Clifford circuits. Optimising a 5 qubit circuit with less than 100 gates took upwards of half an hour. Contrast this to Figure 5 that shows PyZX doing the similar task of bringing Clifford circuits with tens of qubits and thousands of gates to normal-form in seconds.

Of course, simplifying a Clifford circuit to normal-form is not a standard task for most purposes, but even for the more common use of optimising the T-count of a circuit, PyZX is fast. For instance, the implementation of the phase teleportation routine of Ref. [25] can optimise most of the circuits used in Ref. [30] in seconds. For example, the 9 qubit circuit `nth_prime6.tfc` from the Reversible Circuit Benchmark page [29] contains 1241 gates, 567 of which are T gates. PyZX optimises this to a circuit with 279 T gates in less than a second.

6   Challenges

While PyZX is already a capable, general-purpose circuit optimiser [25], there are still many ways in which PyZX can be improved. We list some of these as a series of challenges.

Challenge 1: Use ancillae in optimising ZX-diagrams. While progress has been made on optimising

circuits using the ZX-calculus [14, 25], these results only concern ancilla-free optimisation. It is currently still unknown how the ZX-calculus might be used to create optimisations that take advantage of the presence of ancillae.

Challenge 2: Find the limitations of circuit extraction. Understanding of the circuit extraction problem is still limited [14]. Even though we have a heuristic that always seems to work in the diagrams we produce by simplification, better understanding of the extraction problem might lead to ways to extract smaller circuits, which would be beneficial to optimisation. We believe the general problem of circuit extraction to be hard (in the complexity-theoretic manner), so finding specific circumstances in which we can successfully extract a circuit is of both theoretical and practical importance.

Challenge 3: Combine circuit routing with the ZX-calculus. Circuits on physical quantum computers (and even logical level descriptions of circuits such as the surface code) do not allow two-qubit interactions to take place between any pair of qubits. If a two-qubit gate needs to take place between qubits that are not adjacent, then some form of *routing* needs to happen. When extracting a circuit from a ZX-diagram, there is a natural place for circuit routing to happen, as noted in Ref. [22]. It would be interesting to see whether this leads to improvements over other methods.

Challenge 4: Use the ZH-calculus for circuit optimisation. The ZX-calculus is very well-suited for describing circuits with Clifford and phase gates. It can however only indirectly reason about Toffoli and CCZ gates. A more natural candidate for describing those gates is the *ZH-calculus* [5]. It stands to reason then that the optimisation of these circuits can be improved by using the ZH-calculus.

Challenge 5: Use graphical calculi for circuit simulation. An intriguing possibility for the usage of ZX-diagrams, is in *strong simulation* of quantum circuits, i.e. determining specific output probabilities given a specific input state. Such a problem can be described by a tensor network [28], or as a scalar ZX-diagram. Using the simplification engine of PyZX, simplifications to the diagram can potentially make the calculation of the scalar more efficient then in other approaches (a similar approach to simplifying tensor networks has already shown that this can make the calculation significantly more efficient [7]).

Challenge 6: Use the ZX-calculus for lattice surgery compilation. The ZX-calculus is a natural choice for describing lattice surgery on a surface code [6]. As a result, one can imagine there being some kind of automated way to convert a quantum circuit into a sequence of lattice surgery operations that can be performed on a surface code.

References

[1] *Anaconda: Python for Data Science*. https://www.anaconda.com/.

[2] *Matplotlib*. https://matplotlib.org/.

[3] *NumPy*. https://www.numpy.org/.

[4] Miriam Backens (2015): *Making the stabilizer ZX-calculus complete for scalars*. In Chris Heunen, Peter Selinger & Jamie Vicary, editors: *Proceedings of the 12th International Workshop on Quantum Physics and Logic (QPL 2015)*, Electronic Proceedings in Theoretical Computer Science 195, pp. 17–32, doi:10.4204/EPTCS.195.2.

[5] Miriam Backens & Aleks Kissinger (2018): *ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity*. arXiv preprint arXiv:1805.02175, doi:10.4204/EPTCS.287.2.

[6] Niel de Beaudrap & Dominic Horsman (2017): *The ZX calculus is a language for surface code lattice surgery*.

[7] Jianxin Chen, Fang Zhang, Mingcheng Chen, Cupjin Huang, Michael Newman & Yaoyun Shi (2018): *Classical simulation of intermediate-size quantum circuits*. arXiv preprint arXiv:1805.01450.

[8] B. Coecke & R. Duncan (2008): *Interacting quantum observables*. In: *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, doi:10.1007/978-3-540-70583-3_25.

[9] B. Coecke & R. Duncan (2011): *Interacting quantum observables: categorical algebra and diagrammatics*. *New Journal of Physics* 13, p. 043016, doi:10.1088/1367-2630/13/4/043016. arXiv:quant-ph/09064725.

[10] B. Coecke & A. Kissinger (2010): *The compositional structure of multipartite quantum entanglement*. In: *Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer, pp. 297–308, doi:10.1007/978-3-642-14162-1_25. Extended version: arXiv:1002.2540.

[11] B. Coecke & A. Kissinger (2014): *Picturing Quantum Processes*. Cambridge University Press, doi:10.1007/978-3-319-91376-6_6.

[12] Andrew W Cross, Lev S Bishop, John A Smolin & Jay M Gambetta (2017): *Open quantum assembly language*. arXiv preprint arXiv:1707.03429.

[13] R. Duncan & S. Perdrix (2010): *Rewriting measurement-based quantum computations with generalised flow*. In: *Proceedings of ICALP*, Lecture Notes in Computer Science, Springer, pp. 285–296, doi:10.1007/978-3-642-14162-1_24.

[14] Ross Duncan, Aleks Kissinger, Simon Pedrix & John van de Wetering (2019): *Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus*. arXiv preprint arXiv:1902.03178.

[15] Andrew Fagan & Ross Duncan (2019): *Optimising Clifford Circuits with Quantomatic*. In Peter Selinger & Giulio Chiribella, editors: Proceedings of the 15th International Conference on *Quantum Physics and Logic (QPL)*, *Electronic Proceedings in Theoretical Computer Science* 287, Open Publishing Association, pp. 85–105, doi:10.4204/EPTCS.287.5.

[16] Amar Hadzihasanovic (2017): *The algebra of entanglement and the geometry of composition*. Ph.D. thesis, University of Oxford.

[17] Amar Hadzihasanovic, Kang Feng Ng & Quanlong Wang (2018): *Two Complete Axiomatisations of Pure-state Qubit Quantum Computing*. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, ACM, New York, NY, USA, pp. 502–511, doi:10.1145/3209108.3209128.

[18] Luke Ellis Heyfron & Earl Campbell (2018): *An efficient quantum compiler that reduces T count*. *Quantum Science and Technology*, doi:10.1088/2058-9565/aad604.

[19] Emmanuel Jeandel, Simon Perdrix & Renaud Vilmart (2018): *A Complete Axiomatisation of the ZX-Calculus for Clifford+T Quantum Mechanics*. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, ACM, New York, NY, USA, pp. 559–568, doi:10.1145/3209108.3209131.

[20] Project Jupyter: *Project Jupyter Homepage*. https://jupyter.org/index.html.

[21] Aleks Kissinger: *Tikzit Homepage*. https://tikzit.github.io/.

[22] Aleks Kissinger & Arianne Meijer-van de Griend (2019): *CNOT circuit extraction for topologically-constrained quantum memories*. arXiv preprint arXiv:1904.00633.

[23] Aleks Kissinger, Alex Merry & Matvey Soloviev (2014): *Pattern graph rewrite systems*. In Benedikt Löwe & Glynn Winskel, editors: Proceedings 8th International Workshop on *Developments in Computational Models*, Cambridge, United Kingdom, 17 June 2012, *Electronic Proceedings in Theoretical Computer Science* 143, Open Publishing Association, pp. 54–66, doi:10.4204/EPTCS.143.5.

[24] Aleks Kissinger & John van de Wetering: *PyZX Documentation*. https://pyzx.readthedocs.io.

[25] Aleks Kissinger & John van de Wetering (2019): *Reducing T-count with the ZX-calculus*. arXiv preprint arXiv:1903.10477.

[26] Aleks Kissinger & John van de Wetering (2019): *Universal MBQC with generalised parity-phase interactions and Pauli measurements*. Quantum 3, p. 134, doi:10.22331/q-2019-04-26-134.

[27] Aleks Kissinger & Vladimir Zamdzhiev (2015): *Quantomatic: A proof assistant for diagrammatic reasoning*. In: *International Conference on Automated Deduction*, pp. 326–336, doi:10.1007/978-3-319-21401-6_22.

[28] Igor L Markov & Yaoyun Shi (2008): *Simulating quantum computation by contracting tensor networks*. SIAM Journal on Computing 38(3), pp. 963–981, doi:10.1137/050644756.

[29] Dmitri Maslov: *Reversible Logic Synthesis Benchmarks Page*. http://webhome.cs.uvic.ca/~dmaslov/.

[30] Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs & Dmitri Maslov (2018): *Automated optimization of large quantum circuits with continuous parameters*. npj Quantum Information 4(1), p. 23, doi:10.1038/s41534-018-0072-4.

[31] Kang Feng Ng, Amar Hadzihasanovic & Giovanni de Felice (2019): *A diagrammatic calculus of fermionic quantum circuits*. 15, Episciences.org, doi:10.4230/LIPIcs.FSCD.2018.17.

[32] Peter Selinger: *Quipper Language Homepage*. https://www.mathstat.dal.ca/~selinger/quipper/.

[33] Renaud Vilmart (2018): *A Near-Optimal Axiomatisation of ZX-Calculus for Pure Qubit Quantum Mechanics*. Available at https://arxiv.org/abs/1812.09114.