Chapter 4

PyZX: ZX-Calculus Using Python

Now that we have a basic understanding about ZX-Calculus, we can now take our knowledge a step further into a more computational approach. Since we have already installed PyZX as a package in the previous chapter, we will dive more into quantum circuits and how this library works.

PyZX is a Python-based library designed with the purpose of reasoning with large quantum circuits and ZX-diagrams. Luckily for us, it is an open-source resource – and completely free! With PyZX, we can efficiently rewrite ZX-diagrams using some simplification strategies. And don't think these "strategies" are not that important – they are so powerful that they are able to perform highly effective T-count optimization and even to verify correctness of optimised circuits.

If you think that's not enough, PyZX also offers diverse ways to visualise circuits and ZX-diagrams. With this tool, we will be able to present circuits in QC, QASM and Quipper formats, and it can also convert ZX-diagrams into TikZ diagrams. In short, PyZX is a really powerful tool for circuit optimization; for example, to optimize multiple circuits or combining these with other external procedures.

4.1 Overview

We will now specify the main features of PyZX and how to implement them. The objective of this section is not to give a detailed tutorial, but rather a brief tour of what PyZX can do.

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

4.1.1 `Circuit` class

To be more precise, the `Circuit` class is the entry-point for importing and exporting circuits to and from PyZX. There are also multiple ways to implement optimization schemes that directly act onto the `Circuit` (however, we will explain that in the "Circuit Optimisation" chapter). So, to wrap the definition up, a `Circuit` is a list of `Gates` that can be converted

into various representations, like ZX-diagrams or the QASM format. The way to import a circuit into PyZX is the following:

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

With this, PyZX will automatically try to figure out in which format the circuit is represented! There is also a way to generate random circuits:

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

Also, if you use `zx.draw`, a matplotlib figure would be returned instead.

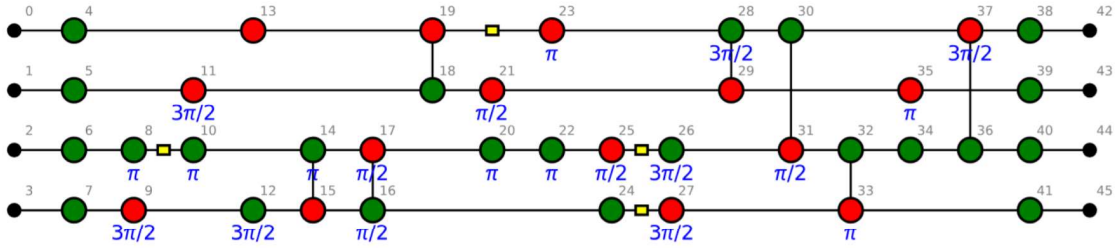


Figure 4.1: Example of the default drawing method of quantum circuits.

You can also convert a circuit into a ZX-diagram:

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

To explain in further depth the simplification routines of ZX-diagrams, we will use a built-in example:

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

Internally, a ZX-diagram is represented as a **graph**.

4.1.2 `Graph` class

The `Graph` class is much more interesting. As you could see in the previous chapter, the graphs in PyZX are pretty simple graphs with typed vertices and edges. There are three types of vertices: boundaries, Z-spiders, and X-spiders. Every single vertex can be labelled by a phase that is stored as a fraction representing a rational multiple of π. These edges can also be divided in two types: one with a regular connection between spiders and a Hadamard-edge.

One of the most common things to deal with PyZX graphs in ZX-Calculus is to add an edge. This can be done with the *add_edge* method. However, if we add an edge where there is already one present will simply replace it. Because of this, it is sometimes more convenient to use the rules of ZX-Calculus to resolve parallel edges and self-loops whenever a new edge

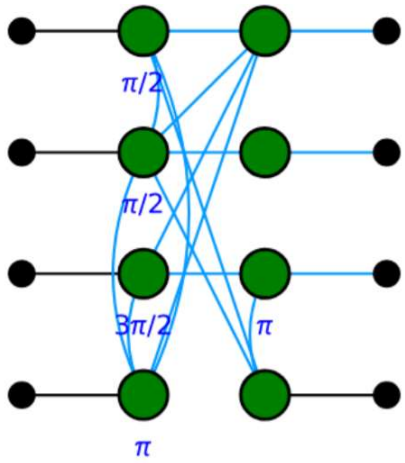


Figure 4.2: A new more compact form of the example above. The blue lines represent edges which have a Hadamard gate on them.

is added. And to use this functionality, we can use the `Graph` method *add_edge_table*, which will take in a list of edges and edge-types to then resolve how the ZX-diagram would look like when every single self-loop or double edges have been resolved.

4.2 TikZiT

LaTeX is an operating system for preparing well-formatted documents. It is in great use in academia where it is commonly used to prepare academic documents like research papers, books, and research articles. As a matter of fact, this book has been written with LaTeXtoo!

TikZ is a programming language in LaTeXthat uses code to make vector-graphics images.

TikZiT is a TikZ-based GUI editor for converting circuit and graphs in PyZX to vector images. TikZiT can be used as a standalone application software – for which you can refer to TikZiT homepage – or you can employ the built-in TikZiT files for buidling ZX-diagrams in LaTeXfrom scratch.

4.2.1 Setting up the Environment in LaTeX

In the following code boxes, you will find the codes of the files necessary to use TikZiT with LaTeX. They can also be found at the official TikZiT Github repository.

```
% sample.tikzdefs

\usepackage{bm}
\newcommand{\param}[1]{\ensuremath{\vec{\bm{#1}}}}
```

Code Listing 4.1: sample.tikzstyles file

```
% TiKZ style file generated by TikZiT. You may edit this file manually,
% but some things (e.g. comments) may be overwritten. To be readable in
```

```
% TikZiT, the only non-comment lines must be of the form:
% \tikzstyle{NAME}=[PROPERTY LIST]

% Node styles
% \tikzstyle{none}=[fill=black, tikzit category=nodes, shape=dot, draw=black
   ]
\tikzstyle{blue node}=[fill=blue, shape=circle, draw=black, tikzit category=
   nodes]
\tikzstyle{X dot}=[fill=red, tikzit category=nodes, shape=circle, draw=black
   ]
\tikzstyle{blue node}=[fill=blue, shape=circle, draw=black, tikzit category=
   nodes]
\tikzstyle{Z dot}=[tikzit fill=green, fill=green, shape=circle, draw=black,
   tikzit category=nodes]
\tikzstyle{H}=[draw=black, fill=yellow, shape=rectangle]
\tikzstyle{blue node 2}=[fill={rgb,255: red,128; green,0; blue,128}, draw=
   black, shape=circle, tikzit fill=blue]
\tikzstyle{D}=[draw=black, fill=black, shape=diamond]

% Edge styles
\tikzstyle{dashed edge}=[<->, dashed]
\tikzstyle{blue pointer}=[->, draw=blue]
```

Code Listing 4.2: sample.tikzdefs file

```
\usepackage{tikz}
\usetikzlibrary{backgrounds}
\usetikzlibrary{arrows}
\usetikzlibrary{shapes,shapes.geometric,shapes.misc}

% this style is applied by default to any tikzpicture included via \tikzfig
\tikzstyle{tikzfig}=[baseline=-0.25em,scale=0.5]

% these are dummy properties used by TikZiT, but ignored by LaTex
\pgfkeys{/tikz/tikzit fill/.initial=0}
\pgfkeys{/tikz/tikzit draw/.initial=0}
\pgfkeys{/tikz/tikzit shape/.initial=0}
\pgfkeys{/tikz/tikzit category/.initial=0}

% standard layers used in .tikz files
\pgfdeclarelayer{edgelayer}
\pgfdeclarelayer{nodelayer}
\pgfsetlayers{background,edgelayer,nodelayer,main}

% style for blank nodes
\tikzstyle{none}=[inner sep=0mm]
```

```
% include a .tikz file
\newcommand{\tikzfig}[1]{%
{\tikzstyle{every picture}=[tikzfig]
\IfFileExists{#1.tikz}
  {\input{#1.tikz}}
  {%
    \IfFileExists{./figures/#1.tikz}
      {\input{./figures/#1.tikz}}
      {\tikz[baseline=-0.5em]{\node[draw=red,font=\color{red},fill=red!10!
         white] {\textit{#1}};}}%
  }}%
}

% the same as \tikzfig, but in a {center} environment
\newcommand{\ctikzfig}[1]{%
\begin{center}\rm
  \tikzfig{#1}
\end{center}}

% fix strange self-loops, which are PGF/TikZ default
\tikzstyle{every loop}=[]
```

Code Listing 4.3: tikzit.sty package file

Save these files in your project folder and include them in your project by using

```
\usepackage{tikzit}
\input{sample.tikzdefs}
\input{sample.tikzstyles}
```

Now, if you compile and build your project, your LaTeXenvironment will install the necessary packages and your project will be capable of supporting TikZiT.

If you don't understand the code language, do not be concerned as this is not necessary for you overall understanding of PyZX functionality.

4.3 Examples

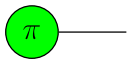
While the TikZiT package allows us to build ZX-diagrams from scratch in LaTeX, its real power is manifested when used with PyZX. Let's explore this with the help of examples.

```
\begin{equation*} // We do not intend to label the `diagram equation'
    \begin{tikzpicture}
        \begin{pgfonlayer}{nodelayer}
                \node [style=Z dot] (0) at (0, 0) {$\pi$};
                \node [] (1) at (1.25, 0) {};
        \end{pgfonlayer}
```

```
        \begin{pgfonlayer}{edgelayer}
                \draw (0) to (1.center);
        \end{pgfonlayer}
    \end{tikzpicture}
\end{equation*}
```

Code Listing 4.4: An example to add TikZiT code in LaTeX

Here is the output:



Note that it is important to encapsulate the tikzfigure in a math environment in LaTeXas tikzfigures are math objects (figures, on the other hand, are float objects).

Writing such code can make your project file tedious. Thus, to save you from this, you can save the code as a text file with .tikz extension. You can then include it in your LaTeXproject by using

```
\begin{equation*}
    \tikzfig{path/sample.tikz}
\end{equation*}
```

Code Listing 4.5: An example to add TikZiT file in LaTeX

To centre the figure, use

```
\begin{equation*}
    \ctikzfig{path/sample.tikz}
\end{equation*}
```

Code Listing 4.6: An example to add a centred TikZiT file in LaTeX

Even now, it sounds like a daunting task to prepare ZX-diagrams in TikZiT, right? Don't worry, because you can conveniently use PyZX for this task. PyZX library has a package `tikz` to provide TikZiT functionality in PyZX. This can be best explained through the following example:

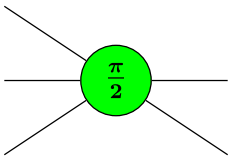
```
# Instantiate an empty graph
g =zx.Graph()
# Add vertices using the following command:
# add_vertex(vertex_type, qubit_num, row_num, phase)
a1 =g.add_vertex(0, 0, 0)
a2 =g.add_vertex(0, 1, 0)
a3 =g.add_vertex(0, 2, 0)
i =g.add_vertex(1, 1, 1.5, np.pi/2)
b1 =g.add_vertex(0, 1, 3)
b2 =g.add_vertex(0, 2, 3)

g.add_edges([(a1, i), (a2, i), (a3, i), (b1, i), (b2, i)])
```

```
# Then obtain the tikzit code
print(zx.tikz.to_tikz(g))
```

This is the output:

```
\begin{tikzpicture}
    \begin{pgfonlayer}{nodelayer}
        \node [style=none] (0) at (0.00, 0.00) {};
        \node [style=none] (1) at (0.00, -1.00) {};
        \node [style=none] (2) at (0.00, -2.00) {};
        \node [style=Z dot] (3) at (1.50, -1.00) {\param{\frac{\pi}{2}}};
        \node [style=none] (4) at (3.00, -1.00) {};
        \node [style=none] (5) at (3.00, -2.00) {};
    \end{pgfonlayer}
    \begin{pgfonlayer}{edgelayer}
        \draw (0) to (3);
        \draw (1) to (3);
        \draw (2) to (3);
        \draw (3) to (4);
        \draw (3) to (5);
    \end{pgfonlayer}
\end{tikzpicture}
```

Now, if you execute this code in LaTeX, the obtained output will be



As a matter of fact, this is how the $Z_m^n(\alpha)$ example was included in this book!

Now that you have had the basic exposure to ZX-Calculus and PyZX (and know some inside hacks of this book), we believe that you are equipped enough to explore the application of these tools. This is what the rest of the book will be focusing on.