Chapter 3

ZX-Calculus: First Contact

To study quantum computing, we have a few special-purpose notations:

- **Quantum Circuits:** This notation studies quantum computing on an implementation level. It explores the components required to build quantum circuits and their *tuning* for optimal performance. This involves both designing and hardware implementation of quantum circuits.
- **Measurement Calculus:** This notation entails a mathematical study of quantum circuits. Making use of mathematical tools, it studies the *manipulation* of quantum data at different stages of quantum circuits.

ZX-calculus falls in the second category of notations. It makes use of the complementarity of Z-basis and X-basis, and defines the quantum circuits in a graphical language so that they are more mathematically elaborate. The advantage is the understanding of quantum algorithms beyond gate-level implementation – among many.

This chapter is dedicated as an introduction to the notation of ZX-Calculus and its elements. It will make use of PyZX, a Python package for ZX-Calculus, to provide an early acquaintance to this tool. PyZX itself will be explored in more detail in the next chapter.

3.1 PyZX: Python Library for ZX-Calculus

As we mentioned in chapter 2, programming is a tool used by engineers, scientists, and pretty much everyone for assistance in computation. We also introduced you to libraries with the promise that we will be using them often. In fact, we will be actively working with coding throughout this book.

PyZX (pronounced as 'pixie') is a python library for assisting with the visualisation of ZX-calculus diagrammatics and for computationally optimising the ZX-graphs.

For installing PyZX, run the following command using either the terminal or your Jupyter notebook:

```
pip install pyzx
```

Once installed, you can use PyZX in your Python program by importing it. For that, use:

```
import pyzx
```

We will not dive into PyZX any deeper in this chapter and will come back to it in chapter 4 when we are in a better position of understanding ZX-Calculus and appreciating PyZX. For now, this much detail will suffice for our working.

3.2 Building Blocks

The ZX-Calculus language is composed of several building blocks which connect together to form the diagrams. As the ZX-Calculus is a graphical diagramamtic language, it chiefly consists of nodes and edges. We will explain them one by one in this section:

3.2.1 Wires

You may have known about electric wires from the circuits in your home and from your science books; they are used to connect the components of an electrical circuit together. Very much like that, we have wires in ZX-Calculus language to connect different components together. However, do not think them of a carrier of electrons like in a classical electrical circuit. Instead, these are like the *edges* of a graph, connecting different nodes together.

Diagrammatically, wires are simply black lines. Let's use the following code to see what wires look like in PyZX:

```
# Instantiate an empty graph
g =pyzx.Graph()
# Add vertices using the following command:
i =g.add_vertex(0, 0, 0)
o =g.add_vertex(0, 0, 3)
g.add_edges([(i,o)])
# Then draw it
display(pyzx.draw(g))
```

Here's the output obtained by running the code:

This is the usual output obtained by using PyZX. However, the lines can be curved as well. One may as well draw the lines freehand as long as they follow the expected path.

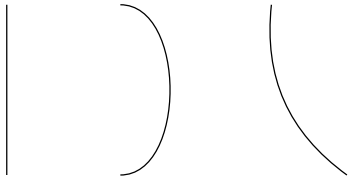


Figure 3.1: All the wires in this figure have the same meaning diagrammatically. Although they have different shapes, they all depict a simple circuit in which the input and output are directly connected.

3.2.2 Z(α) Vertices

Z_m^n are green-coloured vertices which have m inputs and n outputs. m and n can take non-zero integral values, even 0.

Let's make a Z_0^0 vertex using PyZX. For this, run the following code in Python.

```
# Instantiate an empty graph
g =pyzx.Graph()
# Add vertices using the following command:
# add_vertex(vertex_type, qubit_num, row_num, phase)
i =g.add_vertex(1, 0, 0)
# Then draw it
display(pyzx.draw(g))
```

This is the output of the code.

Z vertices also take a phase α as a parameter. The default value of the phase is 0. The significance of this phase will be explained in the next section.

Let's make a $Z_0^0(\pi/2)$ vertex using PyZX. As Python itself does not have have the value of pi stored, we will have to use the numpy library. For that, install numpy by running

```
pip install numpy
```

and then import it using

```
import numpy as np
```

Now, we can add the vertex by using the following code:

```
# add_vertex(vertex_type, qubit_num, row_num, phase)
g.add_vertex(1, 0, 0, np.pi/2)
```

Note:
When running the code on Python, you will not see the value as exactly $\pi/2$. Instead, it will appear as a fractional approximation. This is completely expected and should not put you off when you run the code.

We will explain the Z vertex further in the coming sections, so bear with us as we first introduce you to other vertices in the ZX-language.

3.2.3 X(α) Vertices

X_m^n is a red-coloured vertex having m inputs and n outputs. Just like Z_m^n-vertices, m and n can take non-zero integral values, even 0.

To make a X_0^0 vertex using PyZX, run the following code in Python:

```
# Instantiate an empty graph
g =pyzx.Graph()
# Add vertices using the following command:
# add_vertex(vertex_type, qubit_num, row_num, phase)
i =g.add_vertex(2, 0, 0)
# Then draw it
display(pyzx.draw(g))
```

Similarly, an $X_0^0(\alpha)$ vertex can be generated by using:

```
g.add_vertex(2, 0, 0, np.pi)
```

3.2.4 H Vertices

H vertices are yellow square-shaped vertices which have only one input and one output. They signify the Hadamard transform and can be written in matrix notation as

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

To make an H-vertex using Python, run the follwoing code:

```
# Instantiate an empty graph
g =pyzx.Graph()
# Add vertices using the following command:
i =g.add_vertex(3, 0, 0)
# Then draw it
display(pyzx.draw(g))
```

3.2.5 $\sqrt{D}$ Vertices

A $\sqrt{D}$ vertex is a black diamond-shaped vertex. It has no input or output.

Note:
All these building blocks of ZX-Calculus – the wires and the vertices – are called *generators* and are used to contruct ZX-diagrams, as we will see in the next section.

3.3 Circuits Notation

In ZX-calculus, the diagrams are enclosed in a box with a network of wires and vertices in it. This box is called the *interface* through which the input and output pass through the circuit and through which we can connect other diagrams to the existing circuit. Thus, the simplest diagram would be the input and output connected together, as shown in fig. 3.2 i.e. the qubit goes through the circuit unprocessed. You can think of it in matrix notation as the identity matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

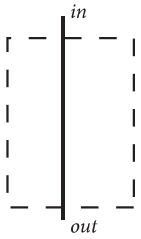


Figure 3.2: A simple ZX-diagram consisting of a single wire connecting input and output

A 2-qubit version would be having two wires passing through the circuit as shown in fig. 3.3. This can be represented by an I_4 matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

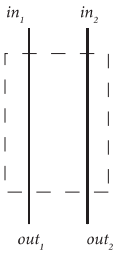


Figure 3.3: A ZX-circuit having two input/output connections.

Remember that we mentioned earlier that the shape of the wires do not matter. It is also worth noting that the overlapping paths of the wires do not matter either, hence the circuit in fig. 3.4 is the same as the one in fig. 3.3.

Notice the size of the matrices; the circuit with 1 wire from input to output has a matrix representation if size 2-by2 while the one with two wires is 4-by-4. In fact,

The 2-by-2 matrix represents the transformation of a 2-by-1 statevector to another 2-by-1 vector such that

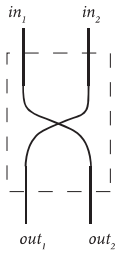


Figure 3.4: In ZX-calculus notation, the wires can overlap without being "shorted". Hence, this figure shows an I_4 circuit.

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Thus, the circuit in fig. 3.3 is a unitary transformation from a single-qubit statevector $(\alpha |0\rangle + \beta |1\rangle)$ to another single-qubit statevector. This can be written mathematically as $|\psi\rangle \to |\psi\rangle$.

You can verify this result for the 4-by-4 matrix by taking the statevector as

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix}.$$

3.3.1 Vertices

But working with wires alone is no fun, right? Thus, to "play" with qubits a bit, we can introduce vertices. These vertices allow us to perform certain operations on qubits. Of what kind? Let's explore!

Let's begin with the H-vertex. It is simply the Hadamard transform as discussed in the previous section with a matrix representation

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Thus, this vertex represents a unitary transformation.

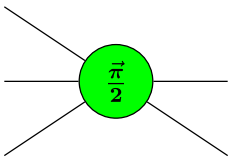
Z_1^1-vertex represents the $R_z(\alpha)$ transform having matrix representation

$$\begin{pmatrix} 1 & 0 \\ 0 & e^{\iota\alpha} \end{pmatrix}$$

This means that the state $|0\rangle$ will remain unaffected after passing through the Z-vertex while the state $|1\rangle$ will gain a local phase of α. Here, we have ignored the local phase.

$$\mathrm{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}; \qquad -\boxed{\mathrm{Z}}- \quad = \quad -(\pi)-$$

Thus, a $Z_m^n(\alpha)$-vertex can be seen a transformation of the state $\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)^{\otimes m}$ to $\left(\frac{|0\rangle + e^{\iota\alpha}|1\rangle}{\sqrt{2}}\right)^{\otimes n}$.



A last interesting case for the Z-vertex will be the Z_0^1 and Z_1^0 vertices. $Z_0^1(\alpha)$ is a transformation $1 \to |0\rangle$ and also $1 \to e^{\iota\alpha}|1\rangle$ (as the wire signifies a superposition state $|+\rangle$). Thus, this vertex is represented as $\frac{1}{\sqrt{2}}(|0\rangle + e^{\iota\alpha}|1\rangle)$. In the same way, $Z_1^0(\alpha)$ vertex is a transformation $\frac{1}{\sqrt{2}}(|0\rangle + e^{\iota\alpha}|1\rangle) \to 1$ and hence can be represented as $\frac{1}{\sqrt{2}}(\langle 0| + e^{\iota\alpha}\langle 1|)$. Notice that it is simply the transpose of Z_0^1 vertex.

$$|+\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}; \qquad |+\rangle \text{ —} \quad = \text{●—}$$

$$|-\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}; \qquad |-\rangle \text{ —} \quad = \text{(π)—}$$

$X_1^1(\alpha)$-vertex represents the $R_x(\alpha)$ transform having matrix representation

$$\begin{pmatrix}\cos\frac{\alpha}{2} & -\iota\sin\frac{\alpha}{2}\\ -\iota\sin\frac{\alpha}{2} & \cos\frac{\alpha}{2}\end{pmatrix}$$

This means that the state $|+\rangle$ will become $|-\rangle$ after passing through the X-vertex while the state $|-\rangle$ will become $|+\rangle$.

$$\mathrm{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}; \qquad \text{—[X]—} \quad = \text{—(π)—}$$

Thus, an $X_m^n(\pi)$-vertex can be seen a transformation of the state $\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)^{\otimes m}$ to $\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)^{\otimes n}$.

A last interesting case for the X-vertex will be the X_0^1 and X_1^0 vertices. $X_0^1(\alpha)$ is a transformation $1 \to (\cos(\frac{\alpha}{2}) - \iota\sin(\frac{\alpha}{2}))|0\rangle$ and also $1 \to (\cos(\frac{\alpha}{2}) - \iota\sin(\frac{\alpha}{2}))|1\rangle$ (as the wire signifies a superposition state $|+\rangle$). Thus, this vertex is represented as

$$\frac{(\cos(\frac{\alpha}{2}) - \iota\sin(\frac{\alpha}{2}))}{\sqrt{2}}(|0\rangle + |1\rangle)$$

. In the same way, $X_1^0(\alpha)$ vertex is a transformation $\frac{(\cos(\frac{\alpha}{2}) - \iota \sin(\frac{\alpha}{2}))}{\sqrt{2}}(|0\rangle + |1\rangle) \to 1$ and hence can be represented as $\frac{(\cos(\frac{\alpha}{2}) - \iota \sin(\frac{\alpha}{2}))}{\sqrt{2}}(\langle 0| + \langle 1|)$. Notice that it is simply the transpose of X_0^1 vertex.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \qquad |0\rangle \text{ —} \quad = \text{ —●—}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}; \qquad |1\rangle \text{ —} \quad = \text{ —●}_{\pi}\text{—}$$