Quantum Computing 101: Essential Math and Concepts

Kshipra Wadikar

March 12, 2025

1 Introduction

We are all familiar with classical computing, where information is stored in bits—each bit can be either 0 or 1, but not both simultaneously. So, what makes quantum computing different?
Quantum computing leverages the principles of quantum mechanics, allowing information to be processed in a fundamentally new way. In quantum systems, we use qubits, which can exist in a superposition of both 0 and 1 at the same time.
This unique property enables quantum computers to perform complex computations exponentially faster than classical computers. Quantum computing has the potential to revolutionize artificial intelligence, cryptography, and optimization by solving problems that are infeasible for traditional computers. However, before diving into these advanced topics, it is essential to have a strong foundation in linear algebra and complex numbers, as they form the mathematical backbone of quantum mechanics.

2 Prerequisites

Before we dive into Quantum Computing, you should have a basic understanding of:

- **Matrices** – Rows, columns, and matrix multiplication
  - Introduction to Matrices (Khan Academy)
  - Matrix Operations (MIT OpenCourseWare)
- **Matrix Product** – Standard matrix multiplication rules
  - MIT OpenCourseWare - Matrix Multiplication
- **Linear Algebra** – Concept of basis vectors and transformations
  - MIT OpenCourseWare - Basis and Dimension
  - University of Oxford - Linear Algebra Notes
- **Complex Numbers** – Basics of imaginary numbers, complex conjugates, and magnitude
  - MIT OpenCourseWare - Complex Numbers (Lecture 2)
  - University of Cambridge - Complex Numbers Notes

For readers who may not be familiar with these concepts, we provide necessary background explanations and references where applicable.

3 Kronecker product rule

The Kronecker product an essential mathematical operation in quantum computing.
This operation is fundamental in constructing multi-qubit systems and understanding how quantum states scale in higher dimensions.
Let A and B be matrices as $A = \begin{bmatrix} a \\ b \end{bmatrix}$ and $B = \begin{bmatrix} c \\ d \end{bmatrix}$.
The Kronecker product is given by:

$$A \otimes B = \begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} aB \\ bB \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}$$

For example
Let A and B be matrices as $A = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$. The Kronecker product is given by:

$$A \otimes B = \begin{bmatrix} 1 \\ 2 \\ -1 \\ -2 \end{bmatrix}$$

Try It Yourself!
To deepen your understanding of the Kronecker product, consider working through the calculation manually.

- Multiply each element of matrix A by the full matrix B.
- Verify whether your result matches the expected outcome.
- If there are discrepancies, review the steps and identify patterns in the computation.

Once you are comfortable with the basics, we will extend this concept to larger matrices and real-world applications.
Our next example is as below.
Let A and B be matrices:

$$A = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad B = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

The Kronecker product is computed as:

$$A \otimes B = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1B \\ 2B \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 6 \\ 8 \end{bmatrix}$$

Now, let's apply the Kronecker product to a larger matrix.
This time, A is a 2×2 matrix, and B is a 2×1 column vector.
Our calculation follows the same rule: multiply each element of A with the full matrix B.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} c \\ d \end{bmatrix}$$

$$A \otimes B = \begin{bmatrix} 1B & 2B \\ 3B & 4B \end{bmatrix} = \begin{bmatrix} 1c & 2c \\ 1d & 2d \\ 3c & 4c \\ 3d & 4d \end{bmatrix}$$

The following **Python script** computes the Kronecker product for specific values of $c = 5$ and $d = 6$:

Listing 1: Computing the Kronecker Product in Python

```
import numpy as np

# Define values for c and d
c, d = 5, 6  # Assign actual numerical values

# Define matrices A and B
A = np.array([[1, 2], [3, 4]])  # 2x2 matrix
B = np.array([[c], [d]])        # 2x1 matrix

# Compute Kronecker product
kronecker_result = np.kron(A, B)

# Display the result

print("Kronecker Product of A and B is :\n", kronecker_result
    )
```

The output of teh above code is as below.

```
Kronecker Product of A and B is:
[[ 5 10]
 [ 6 12]
 [15 20]
 [18 24]]
```

This operation "Kronecker product" plays a fundamental role in quantum mechanics and multi-qubit systems, making it a key mathematical tool for quantum computing.

4 Understanding Qubit Basis Vectors Using the Kronecker Product

4.1 One-Qubit System

In quantum computing, qubits are represented as column vectors in a 2D complex Hilbert space $\mathcal{H}_2$:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

These are column vectors in $\mathbb{C}^2$.

In a 2D complex Hilbert space, $|0\rangle$ and $|1\rangle$ serve as basis vectors for representing a qubit.

A general qubit state is given by:
$|\psi\rangle = a_0 |0\rangle + a_1 |1\rangle$, where $a_0, a_1 \in \mathbb{C}$ and $|a_0|^2 + |a_1|^2 = 1$.

Understanding Qubit Superposition
Consider a qubit state given by: $|\psi\rangle = a_0 |0\rangle + a_1 |1\rangle$ where $a_0, a_1 \in \mathbb{C}$

and satisfy the normalization condition: $|a_0|^2 + |a_1|^2 = 1$.
a_0 and a_1 are called **probability amplitudes**.
The squares of their magnitudes, $|a_0|^2$ and $|a_1|^2$, represent the probabilities of measuring $|0\rangle$ or $|1\rangle$.
If $a_0 = 1$ and $a_1 = 0$, the qubit is in state $|0\rangle$.
If $a_0 = 0$ and $a_1 = 1$, the qubit is in state $|1\rangle$.
If both a_0 and a_1 are nonzero, the qubit is in a **superposition** of $|0\rangle$ and $|1\rangle$.

4.2 Multi-Qubit System

While a single qubit can hold two possible states $|0\rangle$ and $|1\rangle$, quantum systems become much more powerful when we have multiple qubits.
A system with two qubits is not just a combination of two single qubits—it forms a new quantum state in a larger vector space.
Let us start with a two qubit system.
A two-qubit system has four possible basis states, represented as below.

$$|00\rangle, |01\rangle, |10\rangle, |11\rangle$$

For a two-qubit system, the four basis states are formed using the Kronecker product.

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$$

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$$

$$|11\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

These four states form an **orthonormal basis** for the 2-qubit system in $\mathbb{C}^4$.
Just like a single qubit, a two-qubit system can exist in a superposition of its basis states given as below:
$|\psi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle$,
where $a_{00}, a_{01}, a_{10}, a_{11} \in \mathbb{C}$ and satisfy the normalization condition:
$|a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$.
Example (Bell state) of a two-qubit system
Consider the special case where the two-qubit system is in the following superposition:
$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$
Here, the probability amplitudes are:
$a_{00} = \frac{1}{\sqrt{2}}, \quad a_{01} = 0, \quad a_{10} = 0, \quad a_{11} = \frac{1}{\sqrt{2}}$.
Since $\left(\frac{1}{\sqrt{2}}\right)^2 + 0^2 + 0^2 + \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2} + \frac{1}{2} = 1$, the state is normalized.
Three-Qubit Systems
A three-qubit system has eight possible basis states, given by:
$|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle$.
A general three-qubit state is:

$|\psi\rangle = a_{000}|000\rangle + a_{001}|001\rangle + a_{010}|010\rangle + a_{011}|011\rangle + a_{100}|100\rangle + a_{101}|101\rangle + a_{110}|110\rangle + a_{111}|111\rangle$.
These states exist in an 8-dimensional vector space, $\mathbb{C}^8$. **A similar process extends to four-qubit systems, five-qubit systems, and so on.**

5 Introduction to Qiskit and Pennylane

Now that we have a mathematical understanding of multi-qubit systems, the next step is to explore how these concepts are implemented practically using quantum programming frameworks. **Qiskit** and **Pennylane** are two of the most popular libraries for modeling and executing quantum circuits.
Qiskit (developed by **IBM**) is an open-source framework that allows users to create, simulate, and run quantum circuits on actual quantum hardware. It provides tools to work with quantum gates, measurement, and algorithms.
Pennylane (developed by **Xanadu**) is a quantum computing library designed to integrate quantum hardware with machine learning frameworks like TensorFlow and PyTorch. It is particularly useful for quantum machine learning and variational quantum algorithms.

5.1 Installing Qiskit and Pennylane

These quantum programming libraries can be installed using Python's package manager. The following command should be executed in the terminal or command prompt:

Listing 2: install libraries

```
pip install pennylane
pip install qiskit
```

After installation, the versions of Qiskit and Pennylane can be verified using the following Python script:

Listing 3: versions of libraries

```
import qiskit
import pennylane as qml
print("Qiskit version:", qiskit.__version__)
print("PennyLane version:", qml.__version__)
```

When I ran this script on my system, I obtained the following output:

```
Qiskit version: 1.2.4
PennyLane version: 0.39.0
```

5.2 Key Qiskit Imports for Quantum Circuits

To work with quantum circuits in Qiskit, we need to import specific modules that allow us to create, manipulate, and visualize quantum states.

- `QuantumCircuit` – This class is used to create and manipulate quantum circuits. It provides methods to add quantum gates, measurements, and other operations.
- `array_to_latex` – This function is useful for displaying quantum state vectors and matrices in LaTeX format.
- `Statevector` – Represents the quantum state of a system in vector form, allowing us to simulate and analyze pure quantum states.

- `DensityMatrix` – Represents mixed quantum states, which are useful in quantum noise modeling and open quantum systems.

Below is the Python code to import these essential Qiskit components:

Listing 4: essential Qiskit components

```
from qiskit import QuantumCircuit
from qiskit.visualization import array_to_latex
from qiskit.quantum_info import Statevector, DensityMatrix
from IPython.display import display
```

These imports will help us define quantum circuits, visualize their state representations, and analyze quantum systems effectively.

5.3 Creating a Basic Quantum Circuit

First, we will create a quantum circuit with a single qubit. A single qubit can exist in a superposition of $|0\rangle$ and$|1\rangle$.
To create a quantum circuit with a single qubit, we use the 'QuantumCircuit' class from Qiskit. The following code initializes a quantum circuit with one qubit:

Listing 5: quantum circuit with one qubit

```
from qiskit import QuantumCircuit

# Create a quantum circuit with 1 qubit
qc_single = QuantumCircuit(1)

# Draw the circuit (no gates applied yet)
qc_single.draw("mpl")
```

At this stage, the qubit is in its default state, $|0\rangle$, since no quantum gates have been applied yet. The 'draw("mpl")' function visualizes the circuit, showing a single qubit with no operations performed on it.

5.4 Representing a Qubit State Using Statevector

After creating a quantum circuit, we can analyze the qubit's state using the 'Statevector' class from Qiskit. The 'Statevector' class allows us to represent the quantum state mathematically.

Listing 6: Statevector

```
from qiskit.quantum_info import Statevector

# Step 1: Create a statevector from a quantum circuit
# (qc_single should be a valid quantum circuit)
state = Statevector(qc_single)
print("State: ", state)

# Manually interpret the components
print(f"This corresponds to: {state.data[0]} * |0> + {state.data
    [1]} * |1> which is nothing but 1|0>+0|1>.")
```

The output of this code shows that the qubit is initially in the state:

```
Statevector([1.+0.j, 0.+0.j], dims=(2,))
This corresponds to: (1+0j) * |0> + 0j * |1> which is nothing but 1|0>+0|1>
```

This means that the qubit is entirely in the $|0\rangle$ state with probability 1, and there is no contribution from $|1\rangle$. This aligns with our expectation since we have not applied any quantum gates yet.

5.5 Visualizing the Statevector in LaTeX Format

Once we have obtained the statevector representation of our quantum circuit, we can convert it into LaTeX format for better visualization. This is particularly useful when working in Jupyter Notebook, as it allows us to render the quantum state in a clear and structured manner.

Listing 7: Visualizing the Statevector in LaTeX Format

```
from qiskit.visualization import array_to_latex
from IPython.display import display

# Step 2: Convert the statevector into LaTeX format
array_single = array_to_latex(state)

# Step 3: Attempting to print LaTeX object directly (won't
   display properly in Jupyter)
print("array is ", array_single)  # This will print an object
   reference, not the formatted array

# Step 4: Correct way to display the LaTeX representation in
   Jupyter Notebook
print("array is ")
display(array_single)  # This will correctly render the state as
   a formatted LaTeX expression
# Step 5: Print the raw numerical values of the statevector (
   Displays the actual array values-useful for debugging)
print("Array values:\n", state.data)
```

The output of this code is:

```
array is  <IPython.core.display.Latex object>
array is
[ 1 0]
Array values: [1.+0.j 0.+0.j]
```

This confirms that the qubit remains in the $|0\rangle$ state, represented as:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This visualization helps in understanding the quantum state representation and aligns with our mathematical expectations.

5.6 Density Matrix Representation

The density matrix is an alternative way to represent the state of a quantum system, especially useful when dealing with mixed states. For our single-qubit quantum circuit, the density matrix is given by:

Listing 8: Density Matrix Representation

```
    # Step 6: Compute the density matrix representation of the
       quantum circuit
    dm_single = DensityMatrix(qc_single)

    # Step 7: Print the density matrix in standard numerical form
    print("density matrix is \n", dm_single)  # Displays the raw
       density matrix values smoothly
```

The output of this code is:

```
    density matrix is
 DensityMatrix([[1.+0.j, 0.+0.j],
                [0.+0.j, 0.+0.j]],
               dims=(2,))
```

5.7 Creating a Two-Qubit Quantum Circuit

Now, we will create a quantum circuit with two qubits.
A two-qubit system exists in a superposition of the basis states: $|00\rangle, |01\rangle, |10\rangle, |11\rangle$.
Using Qiskit, we define a quantum circuit with two qubits as follows:

Listing 9: quantum circuit with two qubits

```
from qiskit import QuantumCircuit

# Create a quantum circuit with 2 qubits
qc_two = QuantumCircuit(2)

# Draw the circuit (no gates applied yet)
qc_two.draw("mpl")
```

At this stage, the two-qubit system is initialized to the default state $|00\rangle$. No quantum gates have been applied yet. The following Python code generates the statevector representation of our quantum circuit:

Listing 10: statevector

```
from qiskit.quantum_info import Statevector

# Step 1: Create a statevector from a two-qubit quantum circuit
state = Statevector(qc_two_system)
print("State: ", state)
```

The output of this code is:

```
Statevector([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j], dims=(2, 2))
```

This confirms that the two-qubit system is initialized in the $|00\rangle$ state, with no contribution from the other basis states. To visualize this quantum state mathematically, we can convert it into a LaTeX matrix using the '$array_to_latex$' function.

Listing 11: array to latex

```
from qiskit.visualization import array_to_latex
from IPython.display import display

# Step 2: Convert the statevector into LaTeX format
```

```
array_two = array_to_latex(state)
# Step 3: Correct way to display the LaTeX representation in
   Jupyter Notebook
print("array is ")
display(array_two)
```

The output of this code is:

```
array is
[1 0 0 0]
Array values: [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
```

This confirms that the system remains in the $|00\rangle$ state.
The density matrix is another useful representation of quantum states, especially when working with mixed states. For our two-qubit quantum circuit, the density matrix is computed as follows:

Listing 12: DensityMatrix

```
from qiskit.quantum_info import DensityMatrix
# Step 4: Compute the density matrix representation
dm_two = DensityMatrix(qc_two_system)
# Step 5: Print the density matrix in standard numerical form
print("Density matrix is \n", dm_two)
```

The output of this code is:

```
Density matrix is
DensityMatrix([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
               [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
               [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
               [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j]],
              dims=(2, 2))
```

5.8 Generalizing to an n-Qubit Quantum Circuit

The methodology we applied for single and two-qubit systems can be extended to an arbitrary number of qubits. By defining a variable n, we can create quantum circuits of any size dynamically.

Listing 13: n-Qubit Quantum Circuit

```
from qiskit import QuantumCircuit

# Define the number of qubits dynamically
n = 4  # Change this value to create circuits of different sizes

# Create an n-qubit quantum circuit
qc_n_qubits = QuantumCircuit(n)

# Draw the circuit
qc_n_qubits.draw("mpl")
```

At this stage, the quantum circuit consists of n qubits, all initialized in the $|0\rangle$ state. This allows us to **scale** our quantum circuit to accommodate larger quantum systems.

5.9 Creating a Quantum Circuit in PennyLane

Similar to Qiskit, **PennyLane** allows us to define quantum circuits and retrieve their quantum states. We will now create a simple quantum circuit in PennyLane with two qubits and check its state. To do this, we use the `qml.device()` function to define a quantum device, followed by a quantum node (`@qml.qnode`) to execute our circuit. Below is the Python code for defining an empty quantum circuit in PennyLane:

Listing 14: Creating an Empty Quantum Circuit in PennyLane

```
import pennylane as qml

# Create a PennyLane quantum device with 2 qubits
dev = qml.device("default.qubit", wires=2)

@qml.qnode(dev)
def empty_circuit():
    """A simple PennyLane circuit with 2 qubits (no gates)"""
    return qml.state()

# Get the statevector
state = empty_circuit()
print("state is ", state)
```

This process is similar to what we did earlier with Qiskit using the `Statevector` class. However, in PennyLane, we define a quantum function using the `@qml.qnode` decorator and retrieve the quantum state using `qml.state()`.