Qiskit Intro Course: A Beginner's Guide to Quantum Computing

Dr. Nenad Balaneskovic

June 8, 2025

1 Introduction to Quantum Computing

Quantum computing is a revolutionary field that leverages **quantum mechanics** to process information in a fundamentally different way compared to classical computing.

1.1 Why Quantum Computing?

Quantum computing offers key advantages over classical computing due to principles such as:

- **Superposition**: Qubits can exist in multiple states simultaneously.
- **Entanglement**: Qubits can be correlated across distances.
- **Quantum Speedup**: Certain problems (factoring, search, optimization) can be solved exponentially faster.

1.2 Classical vs. Quantum Computation

The table below outlines the differences between classical and quantum computing:

Feature	Classical Computing	Quantum Computing
Bits Used	0 or 1 (Binary)	Qubits (Superposition)
Processing Type	Deterministic	Probabilistic
Parallelism	Limited	Exponential Scaling

1.3 Real-World Applications of Quantum Computing

Quantum computing has the potential to revolutionize various industries:

- **Finance**: Risk modeling, portfolio optimization.
- **Cryptography**: Breaking RSA encryption, post-quantum security.

- **Artificial Intelligence**: Machine learning speedups.
- **Drug Discovery**: Simulating molecular interactions at atomic precision.
- **Logistics**: Optimizing supply chain management.

Now that we understand the fundamentals of quantum computing, let's move on to **Chapter 2: Getting Started with Qiskit!**

2 Getting Started with Qiskit

To begin programming quantum circuits, we use **Qiskit**, an open-source quantum computing framework developed by IBM.

2.1 Installing Qiskit

You can install Qiskit using Python's package manager:

```
!pip install qiskit qiskit-aer numpy matplotlib scipy
```

Verify the installation by checking your Qiskit version:

```
import qiskit
print("Qiskit Version:", qiskit.__version__)
```

2.2 Importing Essential Libraries

Once installed, import the necessary libraries:

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
import numpy as np
import matplotlib.pyplot as plt
```

2.3 Checking Available Backends

Qiskit allows users to run circuits on quantum simulators or real quantum hardware. You can list available simulators:

```
from qiskit_aer import AerSimulator

# Initialize the simulator
simulator = AerSimulator()

print("Available Simulators:", simulator)
```

2.4 Creating a Simple Quantum Circuit

Here's a basic quantum circuit using Qiskit:

```
# Create a quantum circuit with 1 qubit
qc = QuantumCircuit(1, 1)

# Apply a Hadamard gate to create superposition
qc.h(0)

# Measure the qubit
qc.measure(0, 0)

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

2.5 Executing the Circuit on a Simulator

After defining a quantum circuit, we can execute it on a simulator:

```
# Transpile the circuit for the simulator
compiled_circuit = transpile(qc, simulator)

# Execute the circuit
job = simulator.run(compiled_circuit, shots=1024)

# Get results
result = job.result()
counts = result.get_counts()

print("Measurement Outcomes:", counts)
```

Next up is **Chapter 3: Fundamental Quantum Concepts**

3 Fundamental Quantum Concepts

Quantum computing relies on key principles of **quantum mechanics**, such as **superposition, entanglement, interference, and measurement**. Understanding these fundamental concepts is crucial for building and executing quantum circuits.

3.1 Qubits and Superposition

Unlike classical bits, which are strictly 0 or 1, a qubit can exist in a quantum superposition of both states:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex probability amplitudes satisfying:

$$|\alpha|^2 + |\beta|^2 = 1$$

This property enables **parallel computation**, allowing quantum computers to explore multiple possibilities simultaneously.

3.2 Quantum Entanglement

Entanglement is a phenomenon where two or more qubits become correlated regardless of distance. This means measuring one entangled qubit **instantaneously influences** the state of the other qubits.

A maximally entangled state, known as a **Bell state**, is represented as:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Entanglement is fundamental to **quantum cryptography, teleportation, and superdense coding**.

3.3 Quantum Interference

Quantum algorithms leverage **interference** to amplify correct solutions and suppress incorrect ones. Constructive interference **boosts** desired outcomes, while destructive interference **eliminates incorrect states**.

3.4 Quantum Measurement

When a qubit is measured, its quantum state **collapses** to either $|0\rangle$ or $|1\rangle$, losing its probabilistic nature. Measurement probabilities are determined by the probability amplitudes α and β.

The **No-Cloning Theorem** states that quantum states **cannot be copied exactly**, making quantum cryptography highly secure.

3.5 Quantum Gates and Unitary Operations

Quantum gates manipulate qubits using **unitary operations**, which preserve quantum information. A matrix U is unitary if:

$$U^\dagger U = UU^\dagger = I$$

where $U^\dagger$ is the **Hermitian conjugate** (complex conjugate and transpose).

Common quantum gates:

- **Hadamard Gate (H)**: Creates superposition.

- **Pauli-X Gate**: Flips the qubit (like a classical NOT gate).
- **Pauli-Z Gate**: Applies a phase flip.
- **CNOT Gate**: Entangles two qubits.

3.6 Visualizing Qubits with the Bloch Sphere

Qubits are often represented on the **Bloch sphere**, a geometric visualization of quantum states.

A general qubit state is:

$$|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\phi}\sin(\theta/2)|1\rangle$$

where θ and ϕ define the qubit's position on the sphere.

—

Next up is **Chapter 4: Hands-On Quantum Circuits!**

4 Hands-On: Building Quantum Circuits

Now that we understand the basics of **qubits, gates, and measurements**, it's time to **build and execute quantum circuits** using Qiskit.

4.1 Creating a Bell State Circuit

The **Bell state** is a fundamental quantum entangled state that connects two qubits, making their measurements correlated.

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{1}$$

A Bell state circuit consists of:

- **Hadamard Gate (H)**: Creates superposition on the first qubit.
- **CNOT Gate**: Entangles the second qubit with the first.

4.2 Building the Circuit in Qiskit

Here's how you can create a Bell state circuit:

```
from qiskit import QuantumCircuit

# Create a 2-qubit quantum circuit
qc = QuantumCircuit(2, 2)

# Apply Hadamard to q0
qc.h(0)
```

```
# Apply CNOT to entangle q1
qc.cx(0, 1)

# Measure both qubits
qc.measure([0, 1], [0, 1])

qc.draw("mpl")  # Visualize the circuit
```

4.3 Executing the Circuit on a Simulator

Once the circuit is created, we can execute it on a quantum simulator:

```
from qiskit import transpile
from qiskit_aer import AerSimulator

# Select the quantum simulator
simulator = AerSimulator()

# Transpile the circuit for the simulator
compiled_circuit = transpile(qc, simulator)

# Execute the circuit
job = simulator.run(compiled_circuit, shots=1024)

# Retrieve results
result = job.result()
counts = result.get_counts()

print("Measurement Outcomes:", counts)  # Expected ~50% '00' and ~50% '11'
```

4.4 Visualizing Quantum Circuits

Qiskit allows us to **draw quantum circuits** for better understanding. You can visualize a circuit using:

```
qc.draw("mpl")  # Standard circuit visualization

# If using Jupyter Notebook, try rendering the circuit in text format:
qc.draw("text")  # ASCII-style circuit representation
```

4.5 Simulating a Measurement

We can simulate a measurement outcome using the following code:

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
```

```
from qiskit.visualization import plot_histogram

# Create a 1-qubit quantum circuit
qc = QuantumCircuit(1, 1)

# Apply Hadamard gate to create superposition
qc.h(0)

# Measure the qubit
qc.measure(0, 0)

# Select the quantum simulator
simulator = AerSimulator()

# Transpile the circuit for the simulator
compiled_circuit = transpile(qc, simulator)

# Run the circuit
job = simulator.run(compiled_circuit, shots=1024)
result = job.result()
counts = result.get_counts()

# Display measurement outcomes
print("Measurement Results:", counts)
plot_histogram(counts)
```

—

Next up is **Chapter 5: Quantum Algorithms Applications!**

5 Quantum Algorithms & Applications

Quantum algorithms leverage **superposition, entanglement, and interference** to solve problems faster than classical approaches. This chapter explores three groundbreaking quantum algorithms: **Deutsch-Jozsa, Grover's Search, and Shor's Factorization**.

5.1 Deutsch-Jozsa Algorithm: Quantum Speedup

The **Deutsch-Jozsa algorithm** demonstrates quantum speedup by determining whether a **black-box function** is **constant** or **balanced** using just one quantum query.

5.1.1 Key Concepts

- Uses **Hadamard gates** to place qubits in superposition.

- Evaluates the function **in parallel**.
- Provides a definite answer in **one quantum computation**, unlike classical methods.

5.1.2 Quantum Circuit Implementation

```
from qiskit import QuantumCircuit

# Create a 3-qubit quantum circuit
qc = QuantumCircuit(3, 3)

# Apply Hadamard gates to input qubits
qc.h([0, 1])

# Oracle (example balanced function)
qc.cx(0, 2)
qc.cx(1, 2)

# Apply Hadamard gates again
qc.h([0, 1])

# Measure the qubits
qc.measure([0, 1], [0, 1])

qc.draw("mpl")
```

—

5.2 Grover's Algorithm: Quantum Search

Grover's algorithm provides a **quadratic speedup** for searching an unsorted database, reducing search complexity from **O(N) $\rightarrow$ O($\sqrt{N}$) * *.

5.2.1 Key Concepts

- Uses **quantum oracle** gates to mark solutions.
- Applies an **amplitude amplification technique** to boost probability.
- Finds the correct solution in $\mathbf{O(}\sqrt{N})queriesinsteadofO(N)$.

5.2.2 Quantum Circuit Implementation

```
from qiskit import QuantumCircuit

# Create a Grover search circuit
qc = QuantumCircuit(2, 2)
```

```
# Apply Hadamard gates (superposition)
qc.h([0, 1])

# Oracle marking the solution (example)
qc.cz(0, 1)

# Grover diffusion operator
qc.h([0, 1])
qc.z([0, 1])
qc.h([0, 1])

# Measure the qubits
qc.measure([0, 1], [0, 1])

qc.draw("mpl")
```

—

5.3 Shor's Algorithm: Integer Factorization

Shor's algorithm efficiently **factorizes large numbers** using quantum techniques, posing a threat to classical cryptographic security (RSA encryption).

5.3.1 Key Concepts

- Uses **Quantum Fourier Transform (QFT)** to extract periodicity.
- Finds the **prime factors** of large numbers exponentially faster.
- Demonstrates how quantum computing can **break classical encryption**.

5.3.2 Quantum Circuit Implementation for Factoring 15

```
import numpy as np
from math import gcd
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from qiskit.circuit.library import QFT

# Step 1: Choose N and a random 'a' such that 1 < a < N and gcd(a, N) = 1
N = 15
a = 11

# Step 2: Construct Quantum Circuit for Period Finding
num_main_qubits = 4  # Qubits for exponent register
```

```
num_auxiliary_qubits = 4  # Auxiliary qubits for modular exponentiation
qc = QuantumCircuit(num_main_qubits + num_auxiliary_qubits, num_main_qubits)

# Apply Hadamard gates for superposition
qc.h(range(num_main_qubits))

# Initialize auxiliary register to |1⟩ for modular exponentiation
qc.x(num_main_qubits)

# Implement Modular Exponentiation: Controlled multiplications for f(x) = a^x mod N
for i in range(num_main_qubits):
    qc.cx(i, num_main_qubits)

# Apply Quantum Fourier Transform (QFT)
qc.append(QFT(num_main_qubits, do_swaps=True), range(num_main_qubits))
qc.measure(range(num_main_qubits), range(num_main_qubits))

# Simulate the Circuit
simulator = AerSimulator()
compiled_circuit = transpile(qc, simulator)
job = simulator.run(compiled_circuit, shots=32768)
result = job.result()
counts = result.get_counts()

# Extract period and compute factors
measured_values = [int(key, 2) for key in counts.keys() if int(key, 2) > 0]
possible_periods = sorted(measured_values)
factors = set()
if possible_periods:
    for period in possible_periods:
        factor1 = gcd(a ** (period // 2) - 1, N)
        factor2 = gcd(a ** (period // 2) + 1, N)
        if factor1 > 1 and factor1 < N:
            factors.add(factor1)
        if factor2 > 1 and factor2 < N:
            factors.add(factor2)

# Output Results
print("Measured Period Values:", possible_periods)
print(f"Computed Factors of {N}:", list(factors))
plot_histogram(counts)
```

—

Next up is **Chapter 6: Running on Real Quantum Hardware!**

6 Running on Real Quantum Hardware

Quantum computing isn't just theoretical—you can **run real quantum circuits on actual hardware!** IBM Quantum provides **cloud-accessible quantum processors**, allowing users to submit their circuits for execution.

6.1 Connecting to IBM Quantum

To access IBM's quantum computers, **create an account** at IBM Quantum and retrieve your API token.

6.1.1 Authenticating with IBM Quantum

```
from qiskit import IBMQ

# Load IBMQ account (replace 'MY_API_TOKEN' with your actual token)
IBMQ.save_account('MY_API_TOKEN')
IBMQ.load_account()
```

After running this, you can **query available backends** to see which quantum processors are accessible.

6.2 Checking Available Quantum Devices

```
provider = IBMQ.get_provider('ibm-q')
print("Available Quantum Computers:", provider.backends())
```

6.3 Submitting Circuits to Actual Quantum Processors

Once connected, you can **execute circuits** on a real quantum computer.

6.3.1 Example: Running a Quantum Circuit on an IBMQ Device

```
from qiskit.providers.ibmq import least_busy
from qiskit import Aer, execute

# Select the least busy quantum backend
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= 5
                                       and not x.configuration().simulator))

print("Using Quantum Computer:", backend.name())

# Create a simple quantum circuit
from qiskit import QuantumCircuit

qc = QuantumCircuit(2, 2)
qc.h(0)
```

```
qc.cx(0, 1)
qc.measure([0, 1], [0, 1])

# Execute on the selected backend
job = execute(qc, backend, shots=1024)
print("Job submitted! ID:", job.job_id())
```

6.4 Analyzing Results

After execution, retrieve and analyze your quantum results.

6.4.1 Fetching & Visualizing Results

```
from qiskit.visualization import plot_histogram

# Get job result
result = job.result()
counts = result.get_counts(qc)

# Plot measurement results
plot_histogram(counts)
```

—

Next up is **Chapter 7: Challenges & Next Steps!**

7 Challenges & Next Steps

Now that you've built and executed quantum circuits, it's time to **apply your knowledge** with a mini-project, explore further learning resources, and refine your Kaggle notebook.

7.1 Mini-Project: Create Your Own Quantum Circuit

Challenge yourself to **design a quantum circuit** using the concepts you've learned.

7.1.1 Suggested Goals

- Create a **custom quantum algorithm**.
- Explore **multi-qubit entanglement** (Bell pairs, GHZ states).
- Implement **Grover's search** for a custom problem.
- Submit your circuit to **IBM Quantum hardware**.

7.1.2 Example Starter Code

```
from qiskit import QuantumCircuit

# Create a 3-qubit quantum circuit
qc = QuantumCircuit(3, 3)

# Apply Hadamard gates to all qubits
qc.h([0, 1, 2])

# Apply CNOT to entangle qubits
qc.cx(0, 1)
qc.cx(1, 2)

# Measure all qubits
qc.measure([0, 1, 2], [0, 1, 2])

qc.draw("mpl")  # Visualize the circuit
```

Modify this **base circuit** to create your own **quantum experiment**!

—

7.2 Further Learning Resources

7.2.1 Quantum Computing Papers & Tutorials

- Qiskit Documentation
- IBM Quantum Experience
- Quantum Algorithm Zoo
- MIT Quantum Computing Course

7.2.2 Recommended Books & Learning Platforms

- **Quantum Computing for Everyone** — Chris Bernhardt
- **Quantum Computation and Quantum Information** — Nielsen & Chuang
- Online courses on Coursera, Udacity, & edX

—

Next up is **Chapter 8: Exercises!**

8 Exercises

This chapter presents hands-on exercises to solidify your understanding of quantum circuits, superposition, entanglement, and quantum algorithms.

8.1 Exercise 1: Creating a Bell State

8.1.1 Task

Create a quantum circuit that **entangles two qubits** using a **Bell state**.

8.1.2 Hint

- Apply a **Hadamard gate** to the first qubit to create **superposition**.
- Use a **CNOT gate** to entangle the second qubit.
- Measure both qubits to observe correlated outcomes.

8.1.3 Solution Code

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram

# Create a 2-qubit circuit
qc = QuantumCircuit(2, 2)

# Apply Hadamard to q0
qc.h(0)

# Apply CNOT to entangle q1
qc.cx(0, 1)

# Measure both qubits
qc.measure([0, 1], [0, 1])

# Initialize the simulator
simulator = AerSimulator()

# Transpile the circuit for execution
compiled_circuit = transpile(qc, simulator)

# Run the circuit
job = simulator.run(compiled_circuit, shots=1024)

# Get results
result = job.result()
```

```
counts = result.get_counts()

# Visualize results
plot_histogram(counts)
qc.draw("mpl")
```

Expected Measurement Outcomes: **50

—

8.2 Exercise 2: Quantum Superposition & Measurement

8.2.1 Task

Apply a **Hadamard gate** to a single qubit, then measure the outcome.

8.2.2 Hint

- The Hadamard gate puts the qubit into an **equal superposition** of $|0\rangle$ and $|1\rangle$.
- After measurement, the qubit **collapses** to either state randomly.

8.2.3 Solution Code

```
# Create a 1-qubit circuit
qc = QuantumCircuit(1, 1)

# Apply Hadamard gate
qc.h(0)

# Measure the qubit
qc.measure(0, 0)

# Simulate the circuit
simulator = AerSimulator()
compiled_circuit = transpile(qc, simulator)
job = simulator.run(compiled_circuit, shots=1024)
result = job.result()
counts = result.get_counts()

# Display results
print("Measurement Outcomes:", counts)
plot_histogram(counts)
```

Expected Results: **50

—

8.3 Exercise 3: Implementing Grover's Algorithm

8.3.1 Task

Use Grover's Algorithm to **search for a marked item** in a quantum system.

8.3.2 Hint

- Create **equal superposition** using Hadamard gates.
- Use an **oracle** to mark the solution by flipping its phase.
- Apply the **Grover Diffusion Operator** to amplify probabilities.

8.3.3 Solution Code

```
from qiskit import QuantumCircuit

# Create a 2-qubit circuit
qc = QuantumCircuit(2, 2)

# Step 1: Apply Hadamard to both qubits (superposition)
qc.h([0, 1])

# Step 2: Oracle (Marking the solution)
qc.cz(0, 1)

# Step 3: Grover Diffusion Operator
qc.h([0, 1])
qc.z([0, 1])
qc.h([0, 1])

# Measure both qubits
qc.measure([0, 1], [0, 1])

qc.draw("mpl")
```

Expected Outcome: **Marked solution appears with high probability** after amplification.

—

8.4 Exercise 4: Implementing Shor's Algorithm (Factoring Numbers)

8.4.1 Task

Use **Shor's Algorithm** to factor a small integer **N = 15** into its prime components.

8.4.2 Hint

- The quantum part of Shor's Algorithm finds the **period r** in the function:

$$f(x) = a^x \mod N$$

- Quantum **Fourier Transform (QFT)** helps extract periodicity efficiently.
- Once r is found, use **greatest common divisors (GCD)** to compute the factors.

8.4.3 Solution Code

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.utils import QuantumInstance
from qiskit.algorithms.factorizers import Shor

# Define number to factorize
N = 15  # Factoring 15 into 3 × 5

# Initialize the quantum simulator
simulator = AerSimulator()
quantum_instance = QuantumInstance(simulator, shots=1024)

# Run Shor's algorithm
shor = Shor(quantum_instance=quantum_instance)
result = shor.factor(N)

print("Prime Factors:", result.factors)
```

Expected Output: **Factors of 15 → [3, 5]**

—

8.5 Exercise 5: Quantum Teleportation

8.5.1 Task

Use **quantum teleportation** to transfer the quantum state of one qubit to another.

8.5.2 Hint

- **Entangle two qubits** using a Bell state ($H + CNOT$).
- **Apply teleportation gates** to transfer quantum information.

- **Measure the sender qubit**, and apply appropriate correction gates to the receiver.

8.5.3 Solution Code

```
from qiskit import QuantumCircuit

# Create a 3-qubit circuit
qc = QuantumCircuit(3, 3)

# Step 1: Prepare qubit state to teleport (Apply Hadamard gate)
qc.h(0)

# Step 2: Create Bell pair (entangle q1 & q2)
qc.h(1)
qc.cx(1, 2)

# Step 3: Apply teleportation steps
qc.cx(0, 1)  # CNOT between qubit 0 and 1
qc.h(0)      # Hadamard on qubit 0

# Step 4: Measure qubit 0 and 1
qc.measure([0, 1], [0, 1])

# Step 5: Apply correction gates based on measurement results
qc.cx(1, 2)  # If q1 is 1, flip q2
qc.cz(0, 2)  # If q0 is 1, apply phase correction

qc.draw("mpl")
```

Expected Outcome: **Qubit 2 has the same quantum state as Qubit 0 before measurement**—successful **quantum teleportation!**

—