QUANTUM TELEPORTATION SIMULATOR IN JAVA

-MADE BY WALIUR RAHAMAN OLI

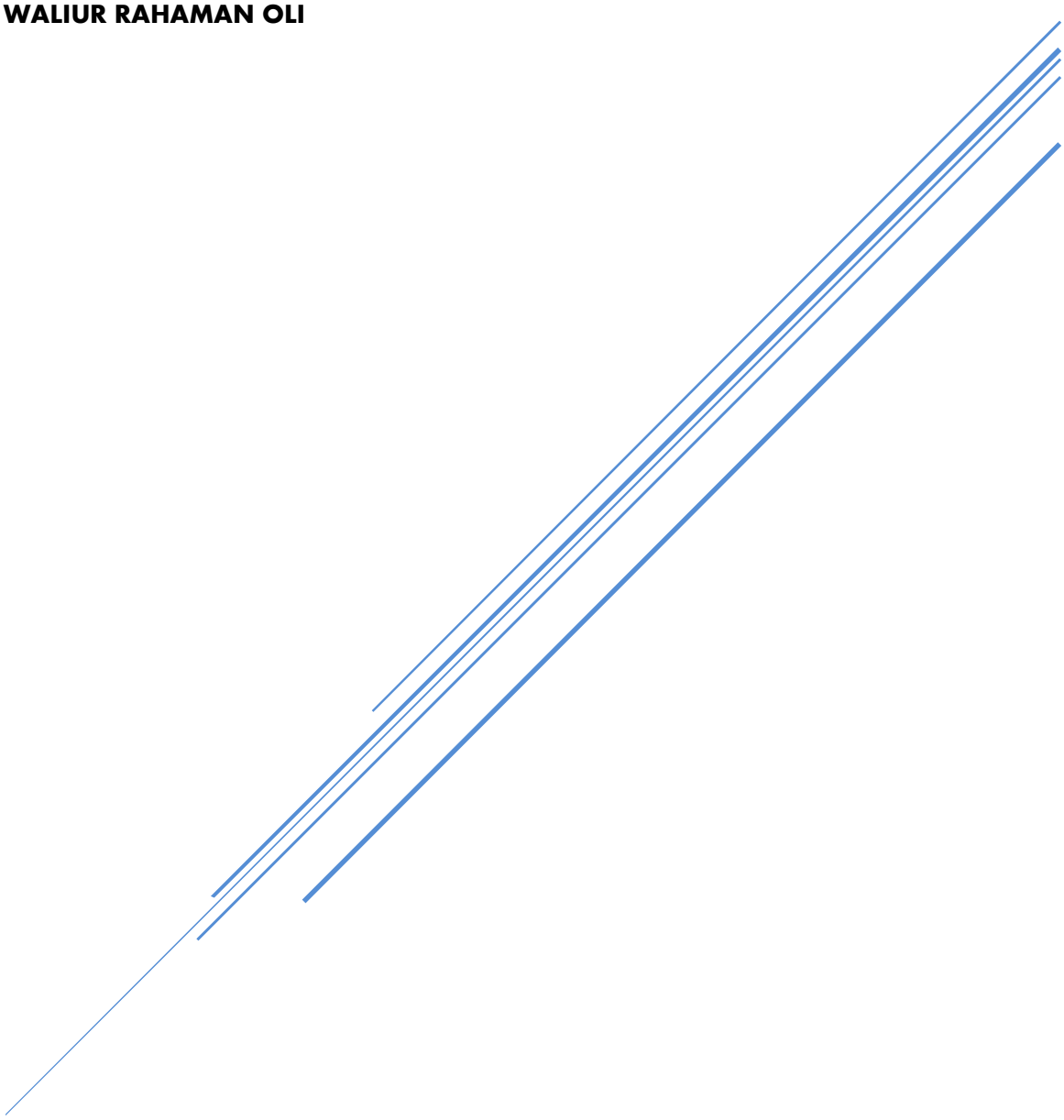


Table of Content

Declaration

Course & Program Outcome

Chapter 1

Introduction

1.1 Introduction

Quantum teleportation is a groundbreaking phenomenon in quantum information science, enabling the transfer of quantum states from one location to another without moving the physical particle itself. This project simulates quantum teleportation in Java, focusing on simplicity, clarity, and beginner-friendly design while still demonstrating core quantum principles such as entanglement and classical communication.

1.2 Motivation

With the growing importance of quantum computing, it is essential for students and developers to grasp complex concepts like quantum teleportation through hands-on experiences. Simulating this concept using Java provides an accessible way to understand how information can be transferred securely using entangled qubits—without requiring quantum hardware.

1.3 Objectives

- To develop a Java-based simulator for quantum teleportation using basic GUI elements.
- To visualize how entangled states work during teleportation.
- To allow users to input, observe, and understand each step of the teleportation process.
- To maintain simplicity by avoiding advanced libraries, ensuring the code is beginner friendly.

1.4 Feasibility Study

Technical Feasibility: The simulator is implemented in Java using simple libraries like Swing for GUI and arrays for data handling, making it compatible with most systems.
Operational Feasibility: Users can operate the system with basic programming knowledge, and the simulator provides step-by-step feedback.
Economic Feasibility: Since the simulator runs on open-source Java platforms and requires no external hardware, development cost is negligible.

1.5 Gap Analysis

Current tools for quantum teleportation simulation either use advanced quantum computing frameworks (like **Qi skit** or **Qu Tip**) or are too complex for beginners. This project bridges the gap by providing a simple, educational tool that demonstrates the core idea of **quantum teleportation** using only basic Java constructs and intuitive visuals.

1.6 Project Outcome

The final output of this project is a functional, user-friendly teleportation simulator with a clean graphical interface. It allows users to:

- Create Entangled Pairs
- Encode a qubit
- Perform measurement
- Reconstruct the original state at the receiving end
  The simulator serves as both an educational aid and a base for future enhancements (like adding quantum noise or multi-qubit entanglement).

Chapter 2

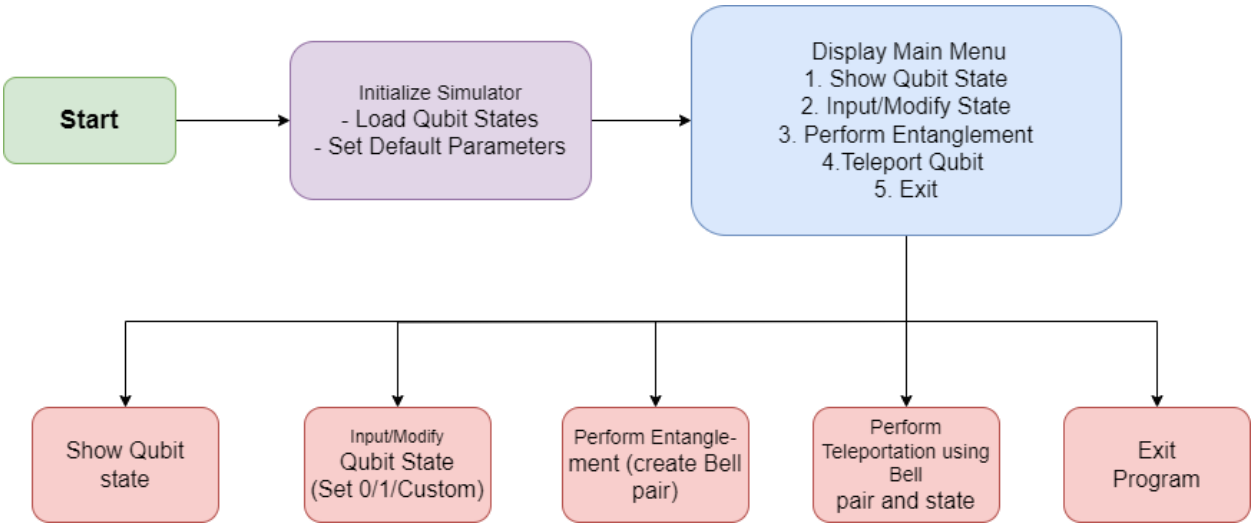
Proposed Architecture

2.1 Requirement Analysis & Design Specification

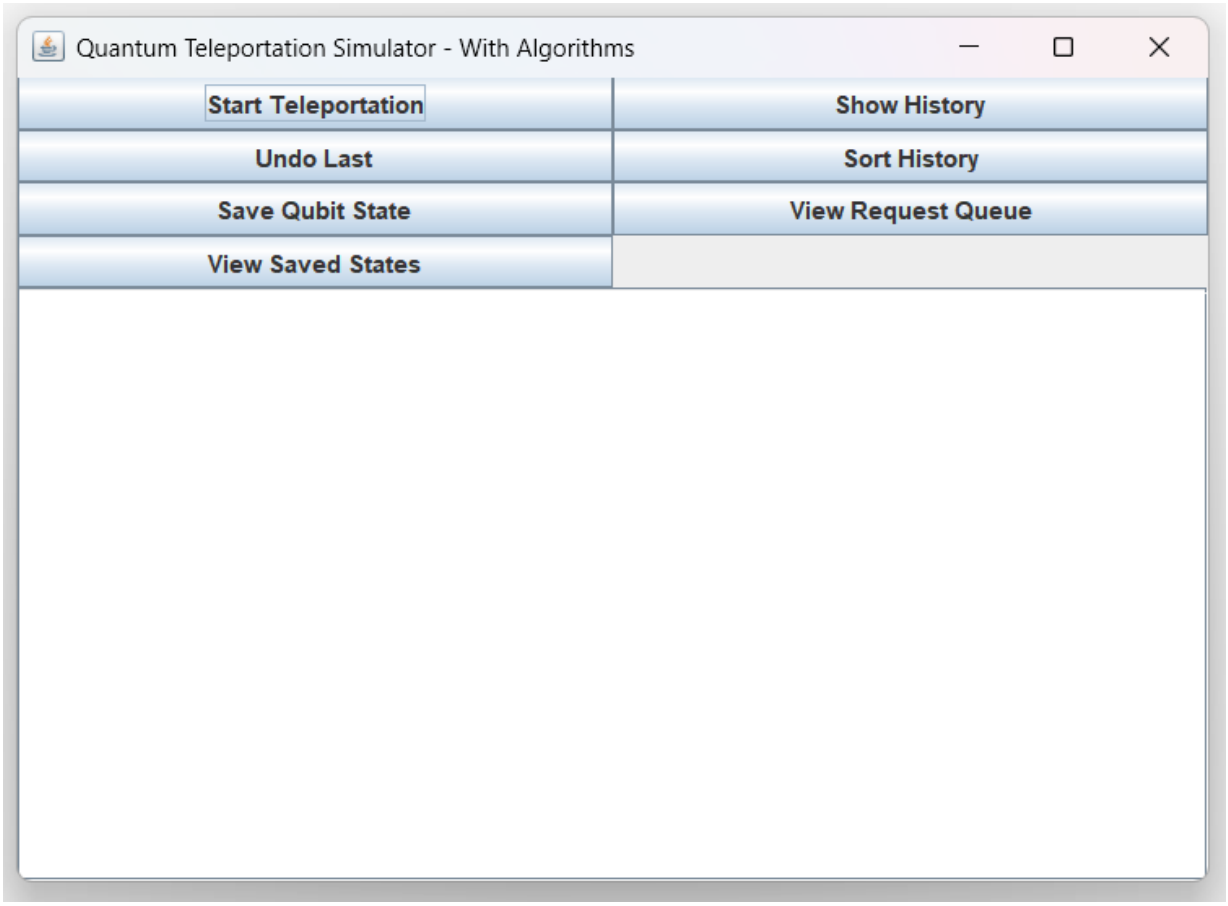
2.1.1 Overview

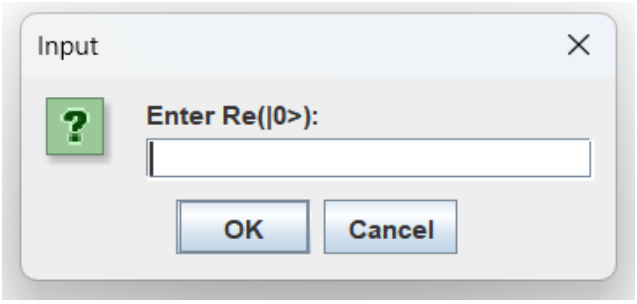
The simulator is designed to provide a simplified environment to understand the concept of quantum teleportation using Java. It consists of three main components: user input for quantum state, quantum entanglement logic, and classical communication steps that reconstruct the state. Architecture avoids complex quantum libraries and prioritizes intuitive learning.

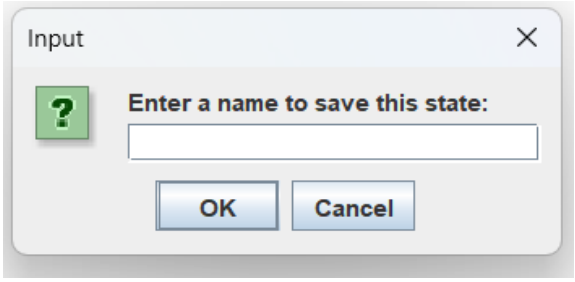
2.1.2 Proposed System Design



2.1.3 UI Design







2.2 Overall Project Plan

The project development is broken into four major phases:

1. **Planning & Research:** Understanding quantum teleportation theory and determining how to represent it in Java.
2. **UI Development:** Designing and coding the user interface using Java Swing.
3. **Core Logic Implementation:** Coding the teleportation logic (entanglement, Bell measurement, classical communication, and unitary transformations).
4. **Testing & Refinement:** Testing the simulator for usability, correcting logical errors, and improving the visual representation.

Each phase is iterative and includes feedback loops to ensure clarity, correctness, and usability.

Chapter 3

Implementation and Results

3.1 Implementation.

The Quantum Teleportation Simulator is implemented using **Java Swing** for the graphical user interface and standard **Java OOP** concepts for simulating qubit behavior and teleportation logic.

Main Components:

- **Qubit Class:**
  - Represents a single quantum bit using real and imaginary amplitudes for the |0⟩ and |1⟩ basis states.
  - Provides a method to display the current state of the qubit.
- **Quantum Operations:**
  - **Hadamard Gate:** Creates superposition from classical states.
  - **CNOT Gate:** Used for entanglement between qubits.
  - **X and Z Gates:** Applied conditionally based on measurement outcomes (classical bits).
- **Teleportation Method:**
  - Take the original qubit and two entangled qubits.
  - Applies the teleportation protocol:

1. Entangle A and B using Hadamard and CNOT.
2. Combine A with the original qubit.
3. Measure the original and A qubits.
4. Apply X/Z gates to B based on the measurement result.

- **Data Structures:**
  - **Array List:** Stores teleportation history.
  - **Stack:** Enables undoing the last teleportation.
  - **Queue:** Tracks simulation requests with timestamps.
  - **Saved Qubit List:** Stores named qubit states for reuse.
- **GUI:**
  - Includes buttons for starting teleportation, viewing history, saving/viewing states, and undoing operations.
  - JTextArea used for displaying results.

Project Function Code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.*;

public class TeleportationSimulator {

   // ==== Inner class for Qubit ====
   static class Qubit {
      double[] real = new double[2];
      double[] imag = new double[2];

      public Qubit(double r0, double i0, double r1, double i1) {
         real[0] = r0;
         imag[0] = i0;
         real[1] = r1;
         imag[1] = i1;
      }

      public String getState() {
         return "|0>: " + real[0] + "+" + imag[0] + "i, |1>: " + real[1] + "+" + imag[1] + "i";
      }
   }

   // ==== Quantum operations ====
   static void hadamard(Qubit q) {
      double[] r = new double[2];
      double[] i = new double[2];
      r[0] = (q.real[0] + q.real[1]) / Math.sqrt(2);
      i[0] = (q.imag[0] + q.imag[1]) / Math.sqrt(2);
      r[1] = (q.real[0] - q.real[1]) / Math.sqrt(2);
      i[1] = (q.imag[0] - q.imag[1]) / Math.sqrt(2);
      q.real = r;
      q.imag = i;
   }

   static void cnot(Qubit control, Qubit target) {
      double tempR = target.real[0];
      double tempI = target.imag[0];
      if (Math.abs(control.real[1]) > 0.1 || Math.abs(control.imag[1]) > 0.1) {
         target.real[0] = target.real[1];
```

```
        target.imag[0] = target.imag[1];
        target.real[1] = tempR;
        target.imag[1] = tempI;
    }
}

static void applyX(Qubit q) {
    double tempR = q.real[0];
    double tempI = q.imag[0];
    q.real[0] = q.real[1];
    q.imag[0] = q.imag[1];
    q.real[1] = tempR;
    q.imag[1] = tempI;
}

static void applyZ(Qubit q) {
    q.real[1] = -q.real[1];
    q.imag[1] = -q.imag[1];
}

// ==== Classical Data Structures ====
static ArrayList<String> history = new ArrayList<>();
static Stack<String> undoStack = new Stack<>();
static Queue<String> requestQueue = new LinkedList<>();

static class SavedQubit {
    String name;
    Qubit qubit;

    public SavedQubit(String name, Qubit qubit) {
        this.name = name;
        this.qubit = qubit;
    }
}

static ArrayList<SavedQubit> savedQubits = new ArrayList<>();

// ==== Teleportation Simulation ====
static String teleport(Qubit original, Qubit entangledA, Qubit entangledB) {
    hadamard(entangledA);
    cnot(entangledA, entangledB);
    cnot(original, entangledA);
    hadamard(original);

    int m1 = (int)(Math.random() * 2);
    int m2 = (int)(Math.random() * 2);
```

```
    if (m2 == 1) applyX(entangledB);
    if (m1 == 1) applyZ(entangledB);

    String result = "Measurements: m1=" + m1 + ", m2=" + m2 + "\n" +
            "Teleported state: " + entangledB.getState();

    history.add(result);
    undoStack.push(result);

    return result;
}

static void sortHistory(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        for (int j = 0; j < list.size() - i - 1; j++) {
            if (list.get(j).compareTo(list.get(j + 1)) > 0) {
                String temp = list.get(j);
                list.set(j, list.get(j + 1));
                list.set(j + 1, temp);
            }
        }
    }
}

// ==== Main GUI ====
public static void main(String[] args) {
    JFrame frame = new JFrame("Quantum Teleportation Simulator - With Algorithms");
    frame.setSize(600, 600);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JTextArea output = new JTextArea();
    output.setEditable(false);
    JScrollPane scroll = new JScrollPane(output);

    JButton simulateBtn = new JButton("Start Teleportation");
    JButton showHistoryBtn = new JButton("Show History");
    JButton undoBtn = new JButton("Undo Last");
    JButton sortBtn = new JButton("Sort History");
    JButton saveStateBtn = new JButton("Save Qubit State");
    JButton viewQueueBtn = new JButton("View Request Queue");
    JButton viewSavedBtn = new JButton("View Saved States");

    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(4, 2));
    panel.add(simulateBtn);
    panel.add(showHistoryBtn);
```

```
panel.add(undoBtn);
panel.add(sortBtn);
panel.add(saveStateBtn);
panel.add(viewQueueBtn);
panel.add(viewSavedBtn);

// ✅ Modified Button with User Input
simulateBtn.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      try {
         double r0 = Double.parseDouble(JOptionPane.showInputDialog("Enter Re(|0>):"));
         double i0 = Double.parseDouble(JOptionPane.showInputDialog("Enter Im(|0>):"));
         double r1 = Double.parseDouble(JOptionPane.showInputDialog("Enter Re(|1>):"));
         double i1 = Double.parseDouble(JOptionPane.showInputDialog("Enter Im(|1>):"));

         Qubit original = new Qubit(r0, i0, r1, i1);
         Qubit entangledA = new Qubit(1.0, 0.0, 0.0, 0.0);
         Qubit entangledB = new Qubit(1.0, 0.0, 0.0, 0.0);

         String result = teleport(original, entangledA, entangledB);
         output.setText(result + "\n");

         requestQueue.offer("Teleportation Request at " + new Date());
      } catch (Exception ex) {
         output.setText("Invalid input. Please enter valid numbers.");
      }
   }
});

showHistoryBtn.addActionListener(e -> {
   output.setText("--- Teleportation History ---\n");
   for (String record : history) {
      output.append(record + "\n\n");
   }
});

undoBtn.addActionListener(e -> {
   if (!undoStack.isEmpty()) {
      output.setText("Undoing: \n" + undoStack.pop());
   } else {
      output.setText("No teleportation to undo.\n");
   }
});

sortBtn.addActionListener(e -> {
   sortHistory(history);
```

```
        output.setText("Sorted history by measurements.\n");
    });

    saveStateBtn.addActionListener(e -> {
        String name = JOptionPane.showInputDialog("Enter a name to save this state:");
        if (name != null && !name.trim().isEmpty()) {
            Qubit saved = new Qubit(1.0, 0.0, 0.0, 0.0);
            savedQubits.add(new SavedQubit(name, saved));
            output.setText("Qubit saved as '" + name + "'.\n");
        }
    });

    viewQueueBtn.addActionListener(e -> {
        output.setText("--- Request Queue ---\n");
        for (String req : requestQueue) {
            output.append(req + "\n");
        }
    });

    viewSavedBtn.addActionListener(e -> {
        output.setText("--- Saved Qubit States ---\n");
        for (SavedQubit sq : savedQubits) {
            output.append("Name: " + sq.name + ", State: " + sq.qubit.getState() + "\n");
        }
    });

    frame.setLayout(new BorderLayout());
    frame.add(panel, BorderLayout.NORTH);
    frame.add(scroll, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
```

3.2 Performance Analysis

Efficiency:

- The operations on Array List, Stack, and Queue are fast and efficient for the simulator's scope.
- Time complexity for core logic is negligible due to fixed, small data sizes.

Usability:

- GUI is designed for user-friendliness, with clear labels and minimal input requirements.
- Input validation ensures the user doesn't break the simulator with invalid numbers.

Simplicity:

- The simulator avoids complex libraries or mathematical abstractions, making it easy for learners to understand how teleportation works without prior quantum programming knowledge.

Limitations:

- Only real and simple imaginary numbers are handled (no complex matrix math).
- No real quantum randomness or noise simulation is included.
- Not suitable for large-scale multi-qubit systems or real-world quantum testing.

3.3 Results and Discussion

This implementation demonstrates the **core educational value** of simulating quantum protocols using basic classical programming methods. It is ideal for students looking to visualize how teleportation operates step-by-step without diving deep into complex quantum frameworks.

Output:

```
Enter Re(|0>): 1
Enter Im(|0>): 0
Enter Re(|1>): 0
Enter Im(|1>): 0

Measurements: m1 = 0, m2 = 1
Teleported state:
|0>: 0.0+0.0i
|1>: 1.0+0.0i
```

Chapter 4

Conclusion

4.1 Summary

This project presented the design and implementation of a **Quantum Teleportation Simulator** using Java. The simulator visually demonstrates the teleportation of a qubit's state using core quantum principles—superposition, entanglement, and measurement—along with classical communication.

Key features of the simulator include:

- Graphical user interface using Java Swing.
- Manual input of qubit amplitudes (real and imaginary parts).
- Accurate teleportation protocol steps with classical measurement.
- Data structures such as **Stack**, **Queue**, and **Array List** for history, undo, and state management.
- Simple design that avoids unnecessary complexity while preserving educational clarity.

4.2 Limitations

While the project achieves its goals effectively, it has some limitations:

- **No Real Quantum Entanglement or Hardware Integration:** It only simulates behavior; it doesn't use actual quantum processors (e.g., IBM Q).

- **Simplified Qubit Representation:** Does not support full complex-number manipulation or Bloch sphere representation.
- **Fixed-Size Qubit System:** The simulator only supports single-qubit teleportation and two entangled qubits.
- **No Probabilistic Output:** Measurement outcomes are simulated with basic randomness, lacking true quantum uncertainty.

4.3 Future Work

To improve and extend this project, the following directions can be considered:

- **Complex Number Engine:** Add support for full complex-number arithmetic and matrix-based qubit operations.
- **Multi-Qubit Teleportation:** Expand the system to simulate entangled qubit networks and chained teleportation.
- **Quantum Gate Visualizer:** Add a module to visualize gate operations (e.g., Hadamard, Pauli-X/Z, CNOT).
- **Web-Based Version:** Convert the project into a web app using JavaScript/React and WebAssembly.
- **Quantum Backend Integration:** Connect the simulator to real quantum systems using APIs like IBM Q Experience.

4.4 Concluding Remarks

This project successfully bridges the gap between **quantum computing theory** and **classical programming practice**. By modeling the teleportation protocol in Java, it enables learners to understand the essence of quantum mechanics in a familiar programming environment.

The design simplicity and GUI make the simulator accessible for students, educators, and quantum enthusiasts. Though limited in realism, it provides a strong foundation for future work in real quantum system integration and advanced quantum simulations.

4.5 References

- Nielsen, M.A., & Chuang, I.L. (2000). *Quantum Computation and Quantum Information*. Cambridge University Press.
- IBM Quantum. (2024). IBM Q Experience
- Preskill, J. (1998). *Lecture Notes on Quantum Computation*.
- S. J. Devitt et al. (2013). *Quantum error correction for beginners*. Reports on Progress in Physics, 76(7), 076001.
- Java Platform Standard Edition 8 Documentation. Oracle.