Final Submission

Bloch Sphere Simulator

Documentation

Quantum Computing and Its Applications

Budapest University of Technology and Economics

May 4th, 2025

Team members:
Akylai Atakanova, QRZ5BZ
Klevis Imeri, T4XGKO

I. Abstract

Quantum computing introduces new paradigms of computation based on the principles of quantum mechanics. *The Bloch sphere* serves as a fundamental tool for representing the states of a single qubit, providing a geometrical interpretation that aids in understanding and analysis. The aim of this project is to develop an interactive *Bloch sphere simulator* that allows users to dynamically visualize quantum state transformations through user-defined matrix operations. Implemented using *SvelteKit*, *Threlte*, and *mathjs*, the simulator enables real-time editing and visualization, bridging theoretical quantum mechanics with practical visualization techniques. The project successfully demonstrates the feasibility of lightweight web-based quantum state visualization and highlights potential directions for further expansion toward more complex quantum systems.

II. Introduction

Quantum computing is a rapidly advancing field that leverages the principles of quantum mechanics to perform computations that are infeasible for classical computers. At the core of quantum computing is the concept of the *qubit*, the quantum analog of the classical binary bit. Unlike classical bits, which exist in a state of either 0 or 1, qubits can exist in a superposition of both states simultaneously, enabling novel forms of computation.

Understanding the state of a qubit is crucial for grasping how quantum algorithms function. However, the abstract mathematical representations commonly used - complex vectors and matrices - can be challenging for students and practitioners to visualize intuitively. To address this gap, *the Bloch sphere* provides a powerful geometrical tool. It represents the pure states of a single qubit as points on the surface of a unit sphere, where different axes correspond to different bases of quantum measurement.

This project aims to develop a **Bloch Sphere Simulator**: an interactive web-based application that allows users to define and apply quantum operations and immediately observe the resulting changes in qubit states on the Bloch sphere. By offering a visual and interactive experience, the simulator facilitates a deeper understanding of fundamental concepts such as superposition, quantum gates, and state transformations.

This simulator is built using *SvelteKit* for the front-end framework, *Threlte* for 3D rendering via Three.js, and mathjs for handling complex numbers and matrix operations. Through the integration of real-time code editing and 3D visualization, the project demonstrates the effective use of modern web technologies in educational tools for quantum mechanics.

The primary goals of this project are:

- To provide an intuitive and interactive method for visualizing single-qubit states and operations.
- To offer a platform that bridges theoretical quantum concepts with practical understanding through visualization.
- To demonstrate the technical feasibility of real-time quantum state simulation using lightweight web frameworks.

This documentation details the motivation, design choices, implementation steps, challenges encountered, and future improvement possibilities for the Bloch Sphere Simulator project.

III. Project Objectives

The Bloch Sphere Simulator project was developed with a set of clear academic and technical objectives. These objectives guided both the design and implementation phases, ensuring that the resulting application would serve as an effective educational tool while demonstrating sound engineering practices.

The specific objectives of the project are as follows:

1. **Visualization of Qubit States**
   To create an interactive 3D visualization of qubit states using the Bloch sphere model, allowing users to observe the effects of quantum operations intuitively.
2. **Real-Time Simulation**
   To enable users to apply quantum gates and arbitrary operations to qubits via a code editor interface, and immediately reflect these operations on the Bloch sphere without requiring page reloads or recompilation.
3. **User Interaction and Experimentation**
   To encourage experimentation by allowing users to input custom matrix operations and quantum transformations through a dynamic, editable code panel integrated directly into the simulator.
4. **Educational Accessibility**
   To design the application so that it can be easily used by students and individuals who may have only a basic background in linear algebra and quantum mechanics, minimizing the barrier to entry.
5. **Efficient and Lightweight Web Application**
   To leverage modern web development frameworks (SvelteKit, Threlte, Math.js) to ensure that the application remains lightweight, responsive, and functional across a wide range of devices and browsers.
6. **Clean and Maintainable Codebase**
   To structure the code in a modular and extensible manner, facilitating future enhancements such as adding support for multiple qubits, mixed states, or animations.
7. **Error Handling and User Feedback**
   To implement robust error detection and handling mechanisms in user-provided code, ensuring that invalid operations or syntax mistakes do not crash the simulator but instead provide informative feedback.

IV. Theoretical Background

Understanding the Bloch Sphere Simulator requires a basic knowledge of quantum mechanics and linear algebra, particularly concepts related to qubits, complex numbers, and matrix operations. This section provides a theoretical foundation for these topics, emphasizing the mathematical structures that the simulator models and visualizes.

Qubits and Quantum States

In classical computing, a bit can exist in one of two states: 0 or 1. In quantum computing, however, a qubit (quantum bit) can exist in a linear superposition of both states simultaneously. Mathematically, a qubit is represented as:

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

where:

- $\alpha,\beta \in C$ (complex numbers)

- $|\alpha|^2+|\beta|^2=1$ (normalization condition)

Here, $|0\rangle$ and $|1\rangle$ are the computational basis states.

Complex Numbers and Quantum Amplitudes

Quantum states are described using complex numbers, which have a real and an imaginary part:

$$z=a+bi \qquad (a,b\in R)$$

The magnitude (or norm) of a complex number is crucial in quantum mechanics, as the probabilities of measurement outcomes are determined by the square of these magnitudes.

In the simulator, complex numbers are used extensively to model qubit states and quantum gates.

Matrix Representation of Qubit States

A single qubit state can be represented as a column vector:

$$|\psi\rangle=\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$$

Quantum gates, which manipulate qubit states, are represented as unitary matrices. A matrix U is unitary if:

$$U\dagger U=I$$

where $U\dagger$ is the conjugate transpose of U, and I is the identity matrix. Applying a quantum gate to a qubit involves matrix-vector multiplication:

$$|\psi'\rangle=U|\psi\rangle$$

The Bloch Sphere Representation

The Bloch Sphere is a geometrical representation of the pure state space of a single qubit. Each point on the surface of the unit sphere corresponds to a unique qubit state (up to a global phase).

A general qubit can be parameterized as:

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

where:

- θ is the polar angle $(0 \le \theta \le \pi)$
- ϕ is the azimuthal angle $(0 \le \phi < 2\pi)$

On the Bloch Sphere:

- $|0\rangle$ is at the north pole
- $|1\rangle$ is at the south pole
- Superpositions lie between these two extremes

This visualization makes it easier to understand complex quantum behaviors such as superposition and phase shifts.

Mathematical Representation of Qubits and Bloch Sphere Coordinates

To visualize the state of a qubit on the Bloch sphere, it is necessary to express the quantum state in a canonical, geometrically interpretable form.

A general qubit state is defined as a normalized linear combination of basis vectors:

$$|\psi\rangle=\alpha|0\rangle+\beta|1\rangle, \quad \alpha,\beta \in C, \quad |\alpha|^2+|\beta|^2=1$$

The complex coefficients α and β can be rewritten in polar form:

$$\alpha=r_\alpha e^{i\phi_\alpha},\beta=r_\beta e^{i\phi_\beta}$$

Multiplying the state by a global phase $e^{-i\phi_\alpha}$, which has no observable effect in quantum mechanics, we obtain:

$$|\psi\rangle\equiv e^{i\gamma}|\psi\rangle\equiv r_\alpha|0\rangle+r_\beta e^{i(\phi_\beta-\phi_\alpha)}|1\rangle$$

The expression $r_\alpha^2+r_\beta^2$ confirms that the magnitudes r_α and r_β lie on the unit circle and can be parameterized using a polar angle θ, where:

$$r_\alpha=\cos(\frac{\theta}{2}),r_\beta=\sin(\frac{\theta}{2}),\theta\in[0,\pi]$$

The relative phase $\phi=\phi_\beta-\phi_\alpha$ is called the **azimuthal angle**, and it varies within $[0,2\pi]$. Altogether, the qubit state becomes:

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

This is the standard form used to represent any pure qubit on the surface of the Bloch sphere, where:

- θ: polar angle (from the z-axis)
- ϕ: azimuthal angle (around the z-axis)

Even though global phase factors like $e^{i\gamma}$ exist, they do not influence measurements or interference patterns and are therefore not physically meaningful in most contexts.

To compute the Bloch angles θ and ϕ programmatically from a raw qubit vector [α,β], the convertQubit(qubit QubitVector) function is used.

```
export function convertQubit(qubit: QubitVector): QubitPolar {
  const [alpha, beta] = qubit
  // magnitudes
  const rAlpha: number = abs(alpha) as number;
  const rBeta: number = abs(beta) as number;
  // arguments (phases)
  const phiAlpha: number = arg(alpha) as number;
  const phiBeta: number = arg(beta) as number;
  // polar angle θ = 2 * arccos(|α|)
  const theta = 2 * Math.acos(rAlpha);
  // azimuthal angle φ = φ_β - φ_α, normalized to [0, 2π)
  let phi = phiBeta - phiAlpha;
```

```
  phi = ((phi % (2 * Math.PI)) + 2 * Math.PI) % (2 * Math.PI);

  return { theta, phi };
}
```

To support custom and advanced operations in the simulator, the createEvalContext() function and constants are available in the evaluation context, including both core math.js functions and domain-specific utilities. You can always expand this list by adding more functions specific for the application. For the moment I just added the ones which are more commonly used.

```
function createEvalContext() {
    // Create math instance with all functions
    const math = create(all, {});

    return {
      // Core math.js functions
      matrix: math.matrix,
      multiply: math.multiply,
      sqrt: math.sqrt,
      complex: math.complex,
      add: math.add,
      subtract: math.subtract,
      divide: math.divide,
      pow: math.pow,
      exp: math.exp,
      log: math.log,
      sin: math.sin,
      cos: math.cos,
      tan: math.tan,
      pi: math.pi,
      e: math.e,
      abs: math.abs,
      conj: math.conj,
      re: math.re,
      im: math.im,
      arg: math.arg,
      norm: math.norm,

      // Matrix-specific functions
      ctranspose: math.ctranspose,
      transpose: math.transpose,
      det: math.det,
      inv: math.inv,

      // Your custom functions
      show,
      clear,
      M,

      // Constants
      i: math.complex(0, 1),
      π: math.pi,

      // Namespace fallback (just in case)
```

```
      math
    };
}
```

NOTE: All the included items in the context can be used directly in the editor.
Please use M in the editor to create a matrix.
These functions enable rich user-defined expressions and quantum state manipulations within the simulator's interface.

For a more thorough mathematical review of how we did the transformations, please read the README.md. (It's easier to structure mathematical formulas in LaTeX inside Markdown.)

V. System Design and Architecture

The Bloch Sphere Simulator is designed as a modular and interactive web-based tool that visualizes the evolution of quantum states on the Bloch sphere. Its architecture follows a component-based structure using SvelteKit, ensuring reactivity, maintainability, and performance. This section provides an overview of the main modules and their responsibilities, along with the flow of data and user interaction throughout the system.

Overview

The system consists of the following key components:

- User Interface (UI): Responsible for accepting user input, displaying results, and enabling interaction.
- Text Editor: A custom code input component where users write JavaScript/TypeScript-style mathematical expressions.
- Rendering Engine: Uses Three.js via the Threlte library to render the Bloch Sphere and qubit states.
- Math Engine: Based on mathjs, which supports complex numbers, matrices, and symbolic parsing.
- Execution and State Management: Executes user-defined scripts, evaluates expressions, and dynamically updates the visualization.

These components interact seamlessly to create an intuitive experience for visualizing quantum computations.

Component Structure

page.svelte

This is the main layout file combining the two major interface elements: the TextEditor and the Bloch Sphere. It also includes the resizable layout mechanism that allows users to adjust the editor dynamically.

TextEditor.svelte

This component is a minimalistic code editor based on a contenteditable <pre> element.

- Supports syntax highlighting using highlight.js
- Executes code when Ctrl+Enter is pressed
- Convert user-defined matrix/vector expressions into meaningful operations.
- Extracts normalized complex vectors and invokes show(...) for visualization.

BlochSphere.svelte

Responsible for 3D rendering of qubit vectors on a virtual Bloch sphere using the Three.js-based Threlte framework.

- Renders the sphere with grid lines and axes.

- Accepts state vectors and maps them to 3D coordinates.
- Updates dynamically when new vectors are pushed.

show(...) Function

A central function embedded in the code execution environment that:

- Parses vectors to determine if they are valid quantum states.
- Converts unit-length complex vectors to 3D coordinates.
- Appends valid paths to the global paths array for rendering.

blochUtils.ts : just helper functions for converting and checking the format of vectors and matrices.
stores.ts : stores the qubits into one multidimensional array, which can be accessed globally. It is called a path because it contains paths of qubits.
tutorial.ts: stores the tutorial code.

The others are just 3D drawings, arrows, and labels.

Data Flow

User Input → TextEditor → eval(code) → show(vector) → paths[] → BlochSphere

- The user types quantum operations and vectors in the editor.
- On execution, the code is parsed and evaluated.
- Valid vectors (unit-norm complex vectors) are extracted and passed to the show() function.
- These vectors are converted into 3D points and added to the Bloch Sphere visualization.

Interactivity and Reactivity

The system leverages $state() and $effect() from SvelteKit's reactive syntax for dynamic updates.
The paths array is bound to rendering, so any valid vector pushed into it will trigger a redraw.
Mouse interactions (resize bar) are implemented using native DOM events.

VI. Implementation Details

This section elaborates on the internal implementation of the Bloch Sphere Simulator, focusing on the key components and the logic behind them. It includes a breakdown of the technologies used, the mathematical operations performed, and how code execution is handled securely and interactively. Here we only explain some sample parts of the code which were more important. The best way to understand a code base is to read the code therefore we will keep this part simple and the code clean so it is easy to understand.

Technologies and Libraries Used

- SvelteKit: A modern framework for building reactive web interfaces with support for file-based routing and component-based architecture.
- Threlte: A Svelte wrapper for Three.js that simplifies 3D rendering within Svelte components.
- Mathjs: A powerful library for symbolic computation, supporting matrices, complex numbers, and expression parsing.
- highlight.js: Used for syntax highlighting within the editable code editor.
- Three.js: A low-level 3D rendering engine for WebGL, accessed via Threlte.

Custom Code Editor (TextEditor.svelte)

The TextEditor component serves as a lightweight code editor with the following features:

- Syntax Highlighting: Applied using highlight.js to improve readability.

- ContentEditable: A <pre> tag is used instead of <textarea> to preserve formatting and allow inline syntax styling.
- Keyboard Shortcuts:
  → Ctrl + Enter: Executes the code.
  → Tab: Inserts two spaces, enhancing indentation.
- Dynamic Parsing and Evaluation:
  → Uses eval(code) to run user-defined code.
  → Encapsulated in a try-catch block to handle errors gracefully.

Safe Code Execution

Although eval is typically discouraged due to security risks, in this context, it is limited to a controlled environment because of the browser sandbox.

Vector Normalization and Validation

The `show` function performs critical filtering to ensure only valid quantum states are visualized:

```
export function matrixToQubitVector(matrix: Matrix<Complex>): QubitVector {
  if (!matrix) {
        throw new Error("Matrix is undefined or null");
  }
  const size = matrix.size();
  if (!Array.isArray(size) || size.length !== 2 || size[0] !== 2 || size[1] !== 1) {
        throw new Error(`Invalid matrix dimensions for qubit: expected [2,1], got [${size}]`);
  }

  const alpha = matrix.get([0, 0]);
  const beta = matrix.get([1, 0]);

  if (typeof alpha.re !== 'number' || typeof alpha.im !== 'number' ||
        typeof beta.re !== 'number' || typeof beta.im !== 'number') {
        throw new Error("Matrix elements are not valid Complex numbers");
  }

  const normSquared = alpha.re * alpha.re + alpha.im * alpha.im
                        + beta.re * beta.re + beta.im * beta.im;
  const tolerance = 1e-10;
  if (Math.abs(normSquared - 1) > tolerance) {
        throw new Error(`Qubit vector is not normalized: norm squared is ${normSquared}`);
  }

  return [alpha, beta];
}
```

Input Validation

- Checks if the matrix exists (`!matrix`).
- Ensures it's a **2x1 matrix** (required for a single qubit state).

Complex Number Check

- Verifies that both elements (α and β) are valid complex numbers (with `re` and `im` properties).

Normalization Check

- Computes the squared norm (`|α|² + |β|²`).
- Ensures it equals **1** (within a small tolerance, `1e-10`), meaning the qubit state is properly normalized.

Drag-to-Resize Editor

Implemented in `page.svelte`, this feature provides an adjustable split between the code editor and 3D canvas:

```
function handleMouseDown(event: MouseEvent) {
  isDragging = true;
  startX = event.clientX;
  startWidth = textEditorElement.offsetWidth;
}
```

- Tracks mouse movement and updates the editor width in real time.
- Restricts resizing within min/max bounds to ensure layout stability.

Bloch Sphere Rendering (BlochSphere.svelte)

- Uses Threlte's `<Canvas>` to embed a 3D scene.
- Accepts dynamically updated vectors and renders them as lines/arrows on the sphere.
- Axes and grid lines help orient the viewer and show the X, Y, Z basis.

VII. Testing & Debugging

Ensuring the correctness of quantum state visualization is critical in a Bloch sphere simulator, as even minor inaccuracies in phase or magnitude can significantly affect the interpretation of quantum behavior. In this section, we describe the methods used to test the mathematical computations, UI responsiveness, and rendering accuracy of the simulator.

Mathematical Consistency

To verify that the conversion from qubit vector notation (α,β) to spherical coordinates (θ,ϕ) was implemented correctly, we created a set of predefined test states with known Bloch sphere representations. Examples include:

- $|0\rangle=(1,0)\Rightarrow\theta=0,\phi=0$
- $|1\rangle=(0,1)\Rightarrow\theta=\pi,\phi=0$
- $|+\rangle=\frac{1}{\sqrt{2}}(1,1)\Rightarrow\theta=\frac{\pi}{2},\phi=0$
- $|-\rangle=\frac{1}{\sqrt{2}}(1,-1)\Rightarrow\theta=\frac{\pi}{2},\phi=\pi$

For each state, we verified that the computed angles matched the expected values within a reasonable numerical tolerance ($\leq 10^{-10}$) using JavaScript's `Math.acos, Math.atan2`, and complex number phase extraction via `mathjs`.

Additionally, we ensured that the global phase factor did not affect the output. For instance, multiplying both components of the qubit by a global phase resulted in the same vector direction, as expected in physical quantum state equivalence.

Interactive Input Testing

The simulator provides a text input interface for entering arbitrary qubit states using `mathjs` expressions. To validate this input system:

- Several malformed or invalid inputs were tested (e.g., missing brackets, imaginary-only states, norm violations) to ensure that errors were caught and handled gracefully.

- Valid edge cases such as (0,i), $(\frac{1}{\sqrt{2}},\frac{i}{\sqrt{2}})$, and complex inputs like $(\frac{1+i}{2},\frac{1-i}{2})$ were used to confirm robustness.

Norm Validation

Because qubit vectors must be normalized (i.e., $|\alpha|^2+|\beta|^2=1$), a check was implemented to automatically normalize any valid input. This logic was tested by inputting intentionally unnormalized vectors and verifying that the application both displayed the correct normalized version and rendered it properly on the Bloch sphere.

VIII. Results and Demonstration

This section presents the outcomes of the simulator's development, highlighting the key features, visual capabilities, and correctness of the implementation. It also demonstrates several example scenarios where the simulator can be used to represent and analyze qubit states.

Use Case Demonstrations

To illustrate its functionality, several canonical quantum states were entered into the simulator:

- **State $|0\rangle=(1,0)$**
  - Represented as a vector pointing along the positive Z-axis.
  - $\theta=0,\phi=0$
- **State $|1\rangle=(0,1)$**
  - The vector aligned with the negative Z-axis.
  - $\theta=\pi,\phi=0$
- **State $|+\rangle=\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$**
  - Lies on the equatorial plane, aligned with the X-axis.
  - $\theta=\frac{\pi}{2},\phi=0$
- **State $|-\rangle=\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$**
  - Also on the equator, but directed along the negative X-axis.
  - $\theta=\frac{\pi}{2},\phi=\pi$
- **State $|+i\rangle\frac{1}{\sqrt{2}}(|0\rangle+i|1\rangle)$**
  - Lies on the equatorial plane at +Y axis.
  - $\theta=\frac{\pi}{2},\phi=\frac{\pi}{2}$
- **State $|-i\rangle\frac{1}{\sqrt{2}}(|0\rangle-i|1\rangle)$**
  - Lies on the equatorial plane at +Y axis.
  - $\theta=\frac{\pi}{2},\phi=-\frac{\pi}{2}$

In each of these cases, the rendered vector corresponded accurately to the theoretical Bloch vector, confirming the correctness of the calculations and the visualization pipeline.

Interactive Tutorial

This section is designed to be interactive. Please access the BlochSphere program via https://klevisimeri.github.io/BlochSphere/ or run it locally using `bun vite`. Upon launching the program,

you will be guided through an interactive tutorial that features a series of tests and examples. The tutorial allows you to engage directly with the quantum simulator by uncommenting specific lines in the provided scripts. This hands-on approach demonstrates the program's capabilities, including the ability to define custom scripts and quantum gates.

Educational Benefits

By linking symbolic mathematical inputs with geometric representations, the simulator serves as a valuable learning aid for:

- Visualizing how complex qubit phases influence quantum states.
- Understanding the meaning of global vs. relative phase.
- Gaining intuition for superposition states and their angular dependencies.

The simulator allows users to quickly grasp the abstract nature of quantum states by seeing them mapped onto a tangible 3D sphere. This has practical educational utility in both introductory and advanced quantum computing coursework.

IX. Conclusion

This project aimed to develop an interactive Bloch Sphere Simulator capable of visualizing qubit states and enabling intuitive exploration of the relationship between quantum amplitudes and geometric representation. The simulator was implemented as a web-based application using modern JavaScript libraries such as Svelte for UI structure, Three.js for 3D rendering, and Math.js for symbolic and numerical processing of complex quantum inputs.

Throughout the development process, key theoretical concepts from quantum mechanics were integrated with practical software engineering techniques. The simulator converts standard qubit representations into Bloch vector form, allowing users to interactively manipulate states using either the computational basis (α,β) or spherical coordinates (θ,ϕ). This dual representation facilitates a deeper understanding of the geometric interpretation of quantum states and highlights important properties such as normalization, global phase invariance, and the role of relative phase in determining qubit position on the Bloch sphere.

The successful implementation of the simulator and its correct behavior in response to various canonical states (e.g., $|0\rangle$,$|1\rangle$,$|+\rangle$,$|-\rangle$, $|i\rangle$,$|-i\rangle$) demonstrate that the project fulfills its educational and functional goals. Moreover, the application provides a foundation that can be extended in future work — for example, by incorporating quantum gate operations, simulating quantum measurements, or enabling multi-qubit state visualization.

Overall, this project provided valuable hands-on experience in bridging the abstract mathematical framework of quantum computing with concrete, interactive visualization. It also reinforced principles of user interface design, numerical stability, and modular software architecture. As quantum computing becomes increasingly prominent in academic and industrial research, tools like this simulator play a critical role in making foundational concepts accessible and engaging to students and educators alike.

X. References

1. **Wikipedia contributors**. (n.d.). *Bloch sphere*. Wikipedia. Retrieved April 2025, from https://en.wikipedia.org/wiki/Bloch_sphere *Used for intuitive explanations of Bloch sphere geometry.*
2. **YouTube – Bloch Sphere Explained**. The Bloch Sphere (simply explained) Video on Bloch sphere interpretation and calculations.

3. **OpenAI ChatGPT**. (2025). *Assisted in documentation structuring, academic phrasing and spelling corrections*. Retrieved from https://chat.openai.com
4. **Grammarly**. (2025). *Grammar and writing enhancement tool*. Retrieved from https://www.grammarly.com