Bloch sphere simulator

Made by: Bertalan Lichter, 13. 10. 2024. - 17. 11. 2024.

The goal of this project

This is my work for a university assignment and aims to create a simple Bloch-sphere simulator as a standalone python application.

I intend on this project to be made in such a way that it can easily be integrated into other projects of others. I hope that the core codebase is well-documented enough, so that the user can understand how the library works and how they can use it in their projects. You can check out the code documentation for more information. If you have any questions or suggestions, feel free to open an issue on the github page.

Background information

The Bloch sphere is a geometric representation of a qubit's state. It is a unit sphere in three-dimensional space, with the north and south poles representing the states $|0\rangle$ and $|1\rangle$, respectively. The equator represents a superposition of these states. Any point on the sphere represents a valid qubit state.

The Bloch sphere is a useful tool for visualizing quantum operations. A quantum gate can be represented as a rotation of the Bloch sphere. The axis of rotation is determined by the gate's matrix, and the angle of rotation is determined by the gate's eigenvalues.

An overview of a few Bloch-sphere simulators

Konstantin Herb's Bloch-sphere simulator

This is the work of a student at ETH Zürich, called Konstantin Herb. It is freely available online and has a very nice user interface with a clean, interactive visualization of what you are doing. The project is open-source with the source-code being available on github.

Main features

- the user can apply…
  - any rotation, defined by angles around the x, y and z axis
  - any rotation around a custom axis
  - popular quantum gates
    - X
    - Y
    - Z
    - H
    - S

        - $S^\dagger$
        - T
        - $T^\dagger$
    - pulses
- the user can rotate the sphere around freely and always see the axis, the current state and the history (represented by curves on the surface of the sphere)

Upsides

- very clean, intuitive interface
- open-source
- the ability to do any operation

Drawbacks

- the user cannot directly set a state without resetting the page
- the user cannot see the axis around which we rotate at each step

Bits and electronics' Bloch sphere simulator

This is yet another open-source project with a farily intuitive UI. The interface and the visualization is a bit less refined than the first project, we looked at, but nothing to be ashamed of.

This site presents us with a bit more information about the current state (such as the polar coordinates θ and ϕ, the Cartesian coordinates and the probability amplitudes α and β).

Main features

- the user can apply…
    - half-turn, quarter-turn and eighth-turn gates
        - eg.: H, P_X, $P_X^{1/2}$, $P_X^{1/4}$
    - custom gates that can be defined either by
        - a matrix
        - a rotation around a custom axis
    - lambda gates, defined by two angles:
        - θ - polar angle
        - ϕ - azimuthal angle
- URL-based state sharing

Upsides

- open-source
- the ability to do any operation
- the user can easily read the current state in multiple ways
- the user can share the current state via neat URLs

Drawbacks

- the user cannot directly set a state without resetting the page
- the visualization is a bit less refined

Attila Kun's Bloch-sphere simulator

This is a pet project of Attila Kun, a full-stack developer and BME alumnus. The project is open-source and available on github.

Although the UI is very simple with a very old style, the Bloch-sphere (to be more precise, only the axis, the state and the current operation's parameters) is very well put together. The user can see the axis around which we rotate at each step, as well as the path the state has taken. One very nice feature is the ability to directly set the state by entering the amplitudes or by dragging the state around.

Main features

- the user can apply…
  - any rotation, defined by a unitary matrix
  - popular quantum gates
    - X
    - Y
    - Z
    - H
- the user can set the state directly by…
  - $|0\rangle$, $|1\rangle$, $|+\rangle$ and $|-\rangle$ buttons
  - dragging the state vector around
  - entering the amplitudes directly
- save the current visualization as a .png

Upsides

- open-source
- the user can…
  - see much more information about the current operation
  - directly set the state
  - save the current visualization as a .png
  - apply any operation via a unitary matrix

Drawbacks

- the UI is very simple and old-fashioned
- the user cannot use many predefined operations
- it is limited in the ways the user can define a rotation

My Bloch Sphere Simulator

My aim is to create a library that can be independently used in other projects, alongside a simple Bloch-sphere simulator as a standalone python application. This project is open-source and available on github.

Specification

Main features

- the user can apply…
  - predefined gates
    - eg.: H, X, $S^\dagger$
    - custom gates, defined by a unitary matrix
- the user can set the state directly by defining the amplitudes (they are normalized automatically)
- the user can save the current visualization as a .png
- the visualization includes…
  - the Bloch sphere
  - the state vector
  - the current operation's parameters (path, axis, from and to states)
- the user can easily read…
  - the current state
  - the current operation's matrix

Third-party libraries

- numpy - for linear algebra operations
- matplotlib - for visualization
- tkinter - as the base for the GUI
- customtkinter - for nicer widgets (better looking UI)

Main implementation details

Backend

States The state is a column vector of size 2, where the first element is the amplitude of $|0\rangle$ and the second element is the amplitude of $|1\rangle$.

Gates Every gate is a wrapper for a unitary matrix. The user can apply predefined gates or custom gates defined by a matrix. Applying a gate to a state is a simple matrix multiplication. Determining the axis and angle of rotation is done by finding the eigenvectors and eigenvalues of the gate's matrix.

Frontend

Visualizing the Bloch sphere The Bloch sphere is visualized by a 3D plot in matplotlib. I chose this library due to familiarity, but one has to realize that drawing a Bloch-sphere only involves drawing spheres, parts of circles, arrows and axis (lines or arrows). The states (before and after the operation) and the axis of operation are quivers, and the path by a curve. The animation is also handled by matplotlib with the in-between states calculated by linear interpolation of the angle around the given axis.

Input validation Whenever a component of the unitary matrix is changed, the program checks if the matrix is unitary. If not, the user is notified by the font color of the matrix entry (the change will not take effect).

For the state vector, the program checks if the input is a valid complex number and that the sum of the squares of the amplitudes is 1. If the input is malformed, the user is notified by the font color of the entry (the change will not take effect) but if the sum of the squares is not 1, the program normalizes the vector.

User manual

Table of contents

Installation

To install the software, follow these steps:

- download the software from github. Use the Releases tab to download the latest version.
- open a terminal and navigate to the folder where you downloaded the software
- run the following command: `pip install DOWNLOADED.gz` where `DOWNLOADED.gz` is the name of the downloaded file
- navigate to `/src/app/` and run the following command: `python app_main.py`
- you should now see the app open up!

Usage

Overview

On the left side of the screen, you will see the Bloch sphere. You can rotate the sphere by clicking and dragging. The sphere will rotate around the axis that is perpendicular to the screen. The current state of the qubit is represented by a point on the sphere (red arrow). The state after the current operation is represented by a yellow arrow. The axis around which the rotation is performed is shown as blue dashed arrow. The transition between the two states is shown as an animated line that fades from red to yellow. The history is shown as fading dashed orange lines.

On the right side of the screen, you will see the controls. You can apply various operations to the qubit. You can also reset the qubit to a specific state.

Applying operations

Applying known gates To apply a known gate, click on the corresponding button. The gate's matrix will be loaded up and will be ready to use. After selecting the desired gate, click on the "APPLY" button to apply the gate to the current state.

Applying custom gates To apply a custom gate, simply modify the unitary matrix of the gate in the text box. The matrix should be a valid unitary matrix. After entering the matrix, click on the "APPLY" button to apply the gate to the current state. If the matrix is not valid, the matrix's text will be colored red.

Setting the state

To set the state, click on the corresponding button. The state will be set to the desired state. You can also enter the amplitudes directly in the text boxes. If the input is not a valid number, the text box will be colored red. If the input is a number for each component but the resulting vector is not normalized, the application will normalize the vector and highlight the state-vector's text in light blue to notify the user of the modification.

Evaluation of the text boxes

The text in the text boxes (for the matrix and the state vector) is evaluated upon hitting enter. The program tries to evaluate it using python's `eval` function. All `i`s are replaced with `j`s and the following locals are set for easier use:

- `np` - numpy (default import)
- `sqrt` - numpy.sqrt
- `sin` - numpy.sin
- `cos` - numpy.cos
- `tan` - numpy.tan

If the program cannot evaluate the text, the text box will be colored red.

History and undo

All operations along with their corresponding states are saved in the history. You can undo the last operation by clicking on the "UNDO" button. The history is saved in a stack, so you can undo multiple operations. The history is only updated upon hitting the "APPLY" button. The history stack is maximum 4 elements long. This is due to the fact that the history is shown on the Bloch sphere and the screen would get cluttered if the history was too long.

The history is shown as dashed orange lines. The older a line is, the more faded it is.

Code documentation

As I want to allow the user to use the library in their projects, I want to provide an overview of the more important classes and functions. This will allow the user to understand how the library works and how they can use it in their projects.

The UI's code is intentionally left out, as it is only a light wrapper around the library. The UI is not meant to be used in other projects, so it is not documented here.

Issues and contributions

If you find any inconsistencies or errors in the documentation, please let me know by opening an issue on the github page

If you want to directly contribute to the project, you can fork the repository and create a pull request. I will review the pull request and merge it if it is appropriate.

PyPi

Currently the library is not on PyPi but I plan to add it in the future. This will allow you to install the library using `pip` with an online repository.

Namespace bloch_simulator

Sub-modules

- bloch_simulator.gate
- bloch_simulator.state

Module bloch_simulator.gate

Classes

`Gate(args)` A class to represent a quantum gate. It is represented by a unitary matrix.

Create a new gate.

Args: args: The gate can be created in different ways: - With no arguments, the gate is the identity gate. - With a numpy array of size 2x2. - With a string representing the name of the gate: I, X, Y, Z, H.

Raises: ValueError: If the arguments are invalid.

Static methods

`from_rotation(axis: numpy.ndarray, angle: float)` Create a gate that represents a rotation around an axis.

Args: axis (np.ndarray): The axis of rotation. angle (float): The angle of rotation.

Raises: AssertionError: If the axis does not have three components.

Returns: Gate: The gate that represents the rotation.

Instance variables

`U` Return the matrix that represents the gate.

Returns: np.ndarray: The matrix that represents the gate.

`rotation_angle: float` Return the angle of rotation of the gate in the Bloch sphere.

Raises: ValueError: If the matrix does not have two eigenvalues.

Returns: float: The angle of rotation.

`rotation_axis` Return the axis of rotation of the gate in the Bloch sphere.

Returns: np.ndarray: The axis of rotation.

Methods

`apply(self, state: bloch_simulator.state.State) -> bloch_simulator.state.State`
Apply the gate to a state. The state is a column vector of size 2, where the first element is the amplitude of |0> and the second element is the amplitude of |1>.

Args: state (np.ndarray): The state to apply the gate to.

Returns: np.ndarray: The new state after applying the gate.

`calculate_trajectory(self, state_from: bloch_simulator.state.State, n_points: int)`
Calculates the trajectory of a gate applied to a state in the Bloch sphere.

Args: gate (Gate): The gate to apply. state_from (np.ndarray): The starting state. n_points (int): The number of points to calculate.

Returns: np.ndarray: The points in the Bloch sphere. Each column is a point in Cartesian coordinates.

`set_matrix(self, matrix: numpy.ndarray)` Set the matrix that represents the gate.

Module bloch_simulator.state

Classes

`State(*args)` Class to represent a quantum state. It is represented by a column vector.

Create a new state.

Args: *args: The state can be given in different ways: - With no arguments, the state is |0>. - With a numpy array of size 2. - With two numbers, the state is a column vector with those numbers as amplitudes. - With a string representing the name of the state: 0, 1, +, - (or |0>, |1>, |+>, |->).

Raises: ValueError: If the arguments are invalid.

Instance variables

`alpha: complex` Return the amplitude of |0>.

Returns: complex: The amplitude of |0>.

`beta: complex` Return the amplitude of |1>.

Returns: complex: The amplitude of |1>.

`bloch_coordinates: numpy.ndarray[float]` Return the Bloch coordinates of the state in Cartesian coordinates.

Returns: np.ndarray[float]: The Bloch coordinates of the state in Cartesian coordinates.

`phi: float` Return the phase of the state in the Bloch sphere.

Returns: float: The phase of the state.

`state: numpy.ndarray[complex]` Return the state.

Returns: np.ndarray: The state.

`theta: float` Return the angle of the state in the Bloch sphere.

Returns: float: The angle of the state.

Methods

```
`set_state(self, new_state: Union[numpy.ndarray, Self])`
:   Set the state.

    Args:
        new_state (np.ndarray): The new state.
```