Project Plan – Quantum Circuit Simulator – Circuit 2 team

Scope

Features

Features listed below are from the project description made for Quantum Circuit Simulator by one of our teammates. The list does not include all of them, but the ones we think we are able to achieve during the project. Other features and longer description of the project can be found here.

Basic Features

1. Basic Quantum Components: Implement core quantum gates such as Pauli-X (NOT), Pauli-Y, Pauli-Z, Hadamard (H), and CNOT gates.
2. Graphical User Interface (GUI): Develop a graphical interface for users to design and simulate quantum circuits. The interface should allow users to:
    a. Add, move, connect, and remove quantum gates and qubits.
    b. Visualize the evolution of quantum states.
3. Circuit Persistence: Allow users to save and load quantum circuits to and from files, preserving their structure and configurations.
4. Quantum State Calculations: Compute and display the final state vector of the quantum circuit after the gates are applied.
5. Ready-made Quantum Circuits: Include several pre-built circuits (e.g., Bell state generation, quantum teleportation) to demonstrate the program's functionality.

Additional Features

At least quantum measurement, quantum noise models, QFT and circuit optimization features from the "Additional Features" list of the original project document.

Advanced Features

At least compiling circuits to some gates set format from the "Additional Features" list of the original project document.

Usage

Our software should be used as a desktop application. It has a simple user interface with quantum wires and slots for quantum gates. When the user has placed the gates to the wires, there is a button which evaluates the circuit. The results will be shown as a

state vector. There is also an option to save the circuit or load it from the file and add or delete qubits. The sketch of the GUI is presented in Figure 1.

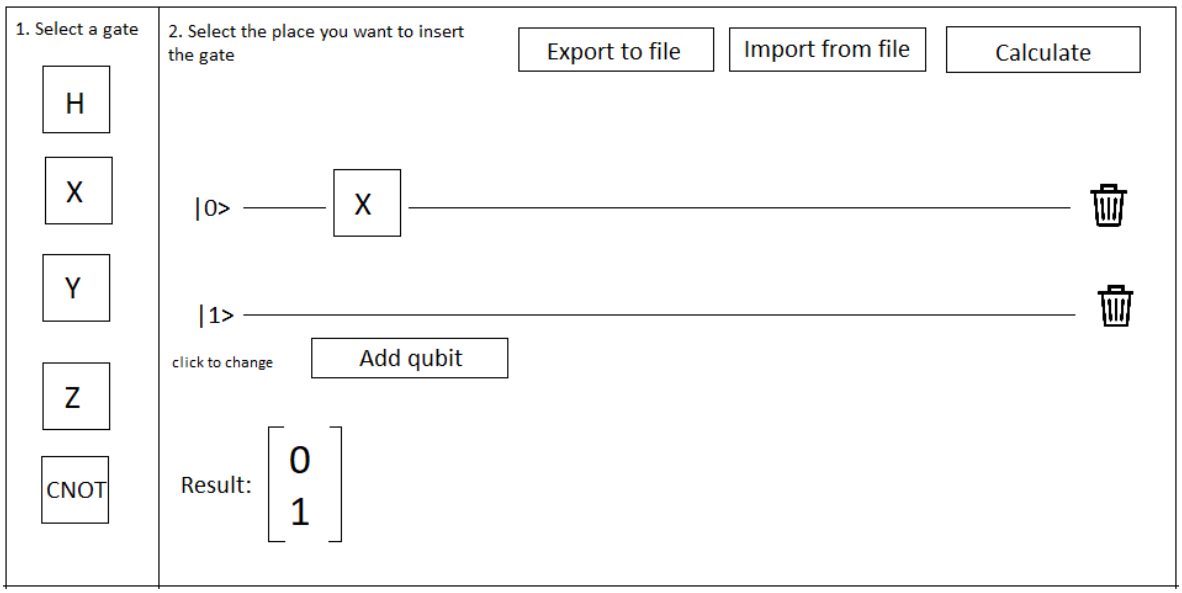

Figure 1: The GUI sketch of the final software.

How Does It Work

It uses linear algebra to represent the state of qubits as vectors (to satisfy the quantum mechanical principles of superposition and entanglement where qubits exist in a combination of 0 and 1 states and entanglement by having the tensor product defined on them) and quantum gates as unitary matrices (representing the Hamiltonian from the Schrödinger equation to allow the time evolution of the state vector). The simulator updates the qubit state by applying these matrices. Quantum gates, like the Hadamard or CNOT, manipulate the state vector using operations derived from quantum mechanics, such as interference and measurement collapse (partial and measurements are done according to the Born rule).

States

Each physical system has a state space, which in our case is a complex vector space. We can represent isolated systems (qubits) by a unit vector in the state space. In our case the state will be represented as a column vector usually denoted using the Dirac notation in the form of

$$|\psi>$$

Composite systems, (multi-qubit systems) can be composed using the tensor product where ψ is the composite system

$$|\psi> = |\upsilon> \otimes \quad |\varphi>$$

Time evolution

In quantum computing we define the time evolution of our state as a discretized version of the Hamiltonian from the Schrodinger equation

$$\mathrm{H}\psi = \mathrm{E}\psi$$

In quantum computation we use unitary matrices often referred to as gates, which take state $|\psi_1>$ to $|\psi_2>$ applying gates to states is basically a simple matrix multiplication, in the case of multi qubits gates one would have to pad the unitary with 2x2 identity matrices.

Gates

The underlying theory behind quantum computation was laid by Fredkin and his proposal of reversible computation, which classical computers are not capable of hence why we require the gates to be unitary. Here U represents an unitary matrix.

$$U\dot{U}^{-1} = 1$$

Program Structure

Software Architecture

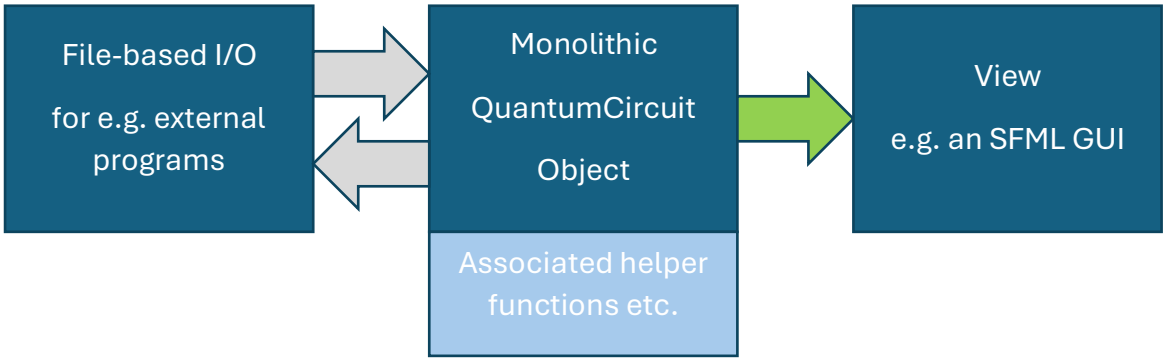

Figure 2: A monolithic QuantumCircuit object with associated helper functions can communicate bi-directionally using file-based I/O and one-directionally to a View such as an SFML GUI.

Class Structures

At the time of writing, our program will revolve around the following classes:

- class QuantumCircuit
    - Getters and setters for interacting with GUI changes

    - AddGate() to any position in the circuit, RemoveGate() from some position
        - A data structure that allows arbitrary addition and removal of gates and controls
    - Evaluate(), computes the result
    - ReadFromFile(...), WriteToFile(...)
        - Using a suitable format
    - Reset(int no_of_qubits)
    - StateVector<Qubit> state_vector_ member (initially zero state)
- class SomeGate : public Eigen::MatrixXcd for a particular X
    - Could store on which qubit is it applied
- class Qubit : public Eigen::Vector2cd
    - Flip() or similar or reconstruct to make [0 1] <-> [1 0]
- class StateVector : public Eigen::VectorXcd
    - Takes a list of qubits as argument and creates the combined StateVector of the qubits
    - Computes the probabilites from the quantum state
    - ApplyGate(array/vec<array/vec<Gate>>)
- MeasureAll()
    - Measure all qubits in the system
    - Could return the results in either a string format or as a list of integers (“101”/[1,0,1])
    - Functionality: for each complex probability amplitude in the state vector it takes the square of their absolute value to get the classical probability of measuring that state.

Libraries

- Eigen
    - A template library for linear algebra. We aim to use most matrix operations from Eigen directly
- SMFL
    - For the GUI and file I/O
- Doxygen

Work organization

Matti	QuantumCircuit class
Henna	GUI/SFML, external communications
Bence	Gate classes
Emma	StateVector and related
Ha	Reading and writing circuits to files

The work that needs to be done is specified in GitLab repository as issues. When someone starts working on an issue, they assign the issue to themselves. A new branch is created from dev and the issue is solved there. In the merge request the issue is mentioned, so they are traceable. The merge request ideally contains the information about the issue, solution and testing instructions for the reviewer. When the reviewer approves the change, it is merged to dev. After every sprint the dev is merged to master.

Sprints

- **Sprint 1: Project planning** (18.10. - 1.11.)
  - Finalize the project plan, set up Gitlab repository, and prepare for feature implementation
    - Goal: Plan the project and initialize repository so that it is easier to start implementing features
- **Sprint 2: Initial implementation** (1.11. - 15.11.)
  - Implement and test basic features: gate classes, basic GUI layout.
    - Goal: Basic features done. Additional features started
- **Sprint 3: Features complete** (15.11. - 29.11.)
  - Add and tests additional features, including measurement, quantum noise models, and QFT.
    - Goal: Additional features done. Advanced features started
- **Sprint 4: Finalization** (29.11. - 13.12.)
  - Polish the software, complete documentation, and prepare for project demo
    - Goal: Wrap up the project

We have weekly meetings every Friday, where we go over the work we have done during the week and the work we need to do before the next meeting. We will also discuss about problems if we face any.