CPSC 4110 Project Report

Roderick MacCrimmon, Mitchell Sulz-Martin, Riley Desrochers

Directory Layout:

- README.md – build instructions
- include
  - quantum.h – type definitions
  - functions.h – helper functions
  - cnot.h – controlled-NOT gate implementation
  - toffoli.h – Toffoli gate implementation
  - deutsch.h – Deutsch's algorithm implementation
- src
  - cnot
    - cnot.cpp – program demonstrating the CNOT gate
  - toffoli
    - toffoli.cpp – program demonstrating the Toffoli gate
  - deutsch
    - deutsch.cpp – program demonstrating Deutsch's algorithm
- test
  - test.cpp – a set of unit tests for the helper functions

Data types:

The data types we used to represent quantum systems are all aliases of types contained within the c++ standard library. These are defined in quantum.h, under the namespace quantum.

qubit - std::pair<std::complex, std::complex>
- a qubit is represented as a pair of complex numbers, representing the coefficients of the state in the $\{|0\rangle, |1\rangle\}$ basis.

state - std::vector<std::complex>
- the coefficients of a general quantum state in the $\{|0\rangle, |1\rangle\}^n$, typically representing a tensor product of several qubits. A two-element state represents the same thing as a qubit

op - std::vector<std::vector<std::complex>>
- A 2D array representing a complex matrix. While any complex matrix is allowed, we made the assumption that in practice, the matrix will be square and unitary.

Helper Functions:

We implemented a number of operations for these types in functions.h.

* operator - multiplication defined for op * op, which returns an op and op * state, which returns a state.

tensor - tensor product defined for op⊗op, which returns a new op, qubit⊗qubit, which returns a state, and state⊗qubit, which returns a state.

get_state - takes a vector of qubits $(q_0, q_1, ..., q_n)$ and returns the state $q_0 \otimes q_1 \otimes ... \otimes q_n$

measurement_prob

- Two versions are defined. One for a qubit, and one for states. The qubit version takes a Boolean value representing the desired outcome of $|0\rangle$or $|1\rangle$and returns the probability of measuring this state by calculating the norm of the correct coefficient squared, i.e. for state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$it returns either $|\alpha|^2$ or $|\beta|^2$.
- The state version additionally takes the total number of qubits and the number of the qubit to be measured. The probability is calculated as the sum of all coefficients for components with the desired outcome. e.g. If we have the state $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$, the probability of measuring qubit 0 in state $|1\rangle$ is $|\gamma|^2 + |\delta|^2$.

Controlled-NOT Gate

The controlled-NOT gates is implemented by the function controlled_not in cnot.h by multiplying the input state $|x\rangle \otimes |y\rangle$by the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and returning the resulting state. The demo program in cnot.cpp takes user input to set $|x\rangle$and $|y\rangle$ to $|0\rangle$ or $|1\rangle$and prints the probability of getting each of these basis states when taking a measurement of the qubits.

Toffoli Gate

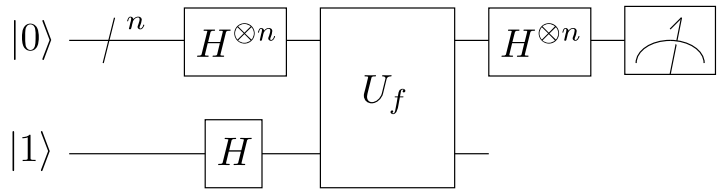
The Toffoli gate is implemented by the function toffoli_gate in toffoli.h by multiplying the input state $|x\rangle \otimes |y\rangle \otimes |z\rangle$ by the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

and returning the resulting state. The demo program in toffoli.cpp works similarly to cnot.cpp, taking user input and printing the probability of measuring each qubit in the state $|0\rangle$ or $|1\rangle$.

Deutsch's Algorithm

The following quantum circuit, with *n* set to 1, is implemented by the function deutsch in deutsch.h.



It takes as input the state $|01\rangle$. It first multiplies the state by the op $H \otimes H$ which acts the Hadamard operator on each qubit. We then use the observation that U_f maps the state

$$\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$$

to the state

$$\alpha|0, f(0)\rangle + \beta|0, \overline{f}(0)\rangle + \gamma|1, f(1)\rangle + \delta|1, \overline{f}(1)\rangle$$

which swaps the coefficients of the first two components if f(0) = 1 and swaps last two coefficients if f(1) = 1. Lastly, the state is multiplied by the op $H \otimes I$.

The demo program deutsch.cpp takes as input the values for f(0) and f(1). The resulting state is printed along with the probability of measuring the first qubit in the state $|0\rangle$ or $|1\rangle$.

What We Learned

- Measurement of a quantum state is not something you do mathematically – We were somewhat confused at first as to how we get the resulting state from measurement, but then realized we obviously cannot take a measurement, as that is something that is done physically. We can only predict the probabilities of the outcome.
- The matrix representing U_f is not always easily obtained – Initially our implementation of Deutsch's algorithm implemented everything with matrix operations. This worked fine for the two balanced functions represented by the identity and Pauli-X matrices. When we tried to implement constant functions with the matrices

  $$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

  we found that the controlled-U gate was not working as expected. This is of course because these matrices are not unitary, and thus not valid quantum gates. It seems that implementing these functions may require some sequence of gates or an approximation, so we changed to the approach described above.

Future Improvements

- Simulation of measurements – Currently we only display the probability of each outcome for measurements. For other systems it might be useful to be able to simulate the measurement and get the resulting state instead. This would allow you to simulate a large number of measurements and plot the results for example.
- Stricter checks on op data type – currently we make the assumptions about how we are using the op type. For more general use it would be good to enforce operators being unitary.