CPSC4110 - Quantum Computing

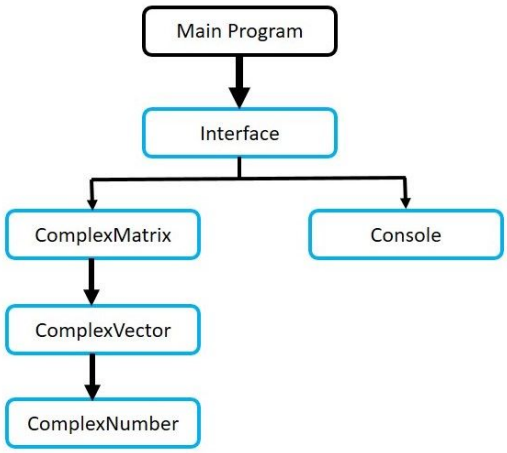
Project Report

Group Members:

- Brett Reqnier
- Daylend de Grasse
- Mitra Kazemzadeh
- Shah Jabeen Sajid

Overall Design

- The basic design of the program is provided on the right side which explains that the main programs calls the interface and interface calls ComplexMatrix and Console class. Furthermore, ComplexMatrix calls ComplexVector, that calls the ComplexNumber.
- The Interface class implements some main functions such as Toffoli, CNOT and Deutsch's Algorithm that calls functions in ComplexMatrix and Console class for further implementation.
- Where ComplexMatrix calls ComplexVector and ComplexVector calls ComplexNumber for other functionality.
- The detailed diagram is provided on the last page.



Running Steps

- The user can run the program with command "*quantum*". It beings by running the main program that calls the interface class.
- The program provides the user with three options:
    1. CNOT
    2. Toffoli
    3. Deutsch's
- At this stage the user can only insert 1, 2, 3 or 4, any other input to the program will result in an error being displayed.
- Upon selecting 1. CNOT, the program asks for the control bit and the target bit. Here the user can insert only 1 or 0. Entering any other value to the program will output an error. The program will output the result of calculating CNOT on the target bit.
- Upon selecting 2. Toffoli, the program will ask the user to insert the first control bit, the second control bit, and the target bit. Here the user can insert only 1 or 0 for all these options.
- Upon selecting 3. Deutsch's, the program will ask the user for the values in f(0) and f(1) value that can be either 1 or 0.
- From this point forward the program will calculate the superposition of the target bit, first applying a Hadamard matrix, next applying the calculation of the Uf function, and then applying a Hadamard to the target qubit again.

- At the last part we observe the state of the qubit, by evaluating the target bit at most 2 times. We have 4 evaluations, we will check if the target qubit is ±|0>, if it is then the program will output that the function is constant. If the target qubit is ±|1> then the program will output that the function is balanced.
- After giving the output of a specific function the program again gives the options to select until the user terminated the program.

Data Structure

The program incorporates fives classes for implementing the CNOT function, Toffoli function and Deutsch's Algorithm.

1. ComplexNumber

- Which consists of two data members i.e real value and imaginary value.
- and consists of some functions that are allocated for a specific complex number passed into it such as: Add, Product, Conjugate, as well as Getter/Setter functions, for our real and imaginary numbers, called Real and Imaginary.

2. ComplexVector

- Contains a list of ComplexNumbers.
- It consists of functions that are allocated for a specific complex vector passed into it. This data structure is used to function as the columns of our ComplexMatrix.
- Some of the functions that are contained in this class are: Add, DotProduct, Conjugate, and Insert.

3. ComplexMatrix

- Has two data members, a vector of ComplexVectors, a ComplexNumber as scalar to the matrix.
- It has functions to make a ComplexMatrix requiring the entry of rows and columns, overloading of the operators: =, +, and [], respectfully in the previous order they are used to: add two complex matrices, multiply two complex matrices, and get a row of the from the vector of ComplexVectors.
- We have functions to check if the ComplexMatrix is: hermitian, identity or unitary.
- As well we have functions for returning the conjugate, transpose, and adjoint of a matrix.
- Lastly it has a function for the tensor product of two matrices and return functions for well known matrices: 2x2 Hadamard matrix, 4x4 CNOT matrix, and 8x8 TOFFOLI matrix.

4. Interface

- This class is to interact with the user, giving the user options to choose from and getting users input and showing the results accordingly.
- It has two data members, *option*: which is used to hold user's chosen option, and *qubit:* which is a 2x1 complex matrix and would be initialized to either|0> or |1> based on users input.
- Additionally the interface has a function for returning a ComplexMatrix that is in a basis state, called int qubit_input(string msg) which displays a passed in string and requests for an input of 0 or 1.
- ComplexMatrix create_qubit(int v) creates a ComplexMatrix in one of the basis states |0> or |1> based on the passed integer.

- ComplexMatrix ControlNOT_function() would implement the Controlled-Not gate. It takes two inputs as the control qubit and the input qubit, as a basis state |0> or |1>, and calculates their tensor product to have 4x1 complex matrix. Then the we multiply the CNOT matrix with the tensor product, and return the result as a 4x1 complex matrix.
- ComplexMatrix Toffoli-function() implements the toffoli gate. It takes two control qubits and a target qubit from the user as |0> or |1> and calculates their tensor product of all three bits, resulting in a 8x1 matrix. Next we multiply the Toffoli Matrix, which is an 8x8 matrix, with the tensor product, which returns an appropriate 8x1 matrix that has been passed through a Toffoli Gate.
- void Deutsch() requires the user to input the values of f(0), and f(1). From there we perform an Hadamard to both the target qubit and the control qubit. We then perform the Uf calculations on the target qubit, respectfully as $(-1)^{f(x)}|x> \otimes (1/\sqrt{2}^)*(|0> - |1>)$. Lastly the target qubit is passed through another Hadamard, and we evaluate up to 2 times on the target qubit to determine if it is constant or balanced. In the case where the target qubit is ±|0> the function will output that the function is constant, and if the target qubit is ±|1> the function outputs that the function is balanced.

5. Console

- This class is used for taking the input from the user and printing the result on the console.
- It has functions like Print, PrintIn and GetInteger that takes in a string and process it for a specific functionality.

Lesson Learned

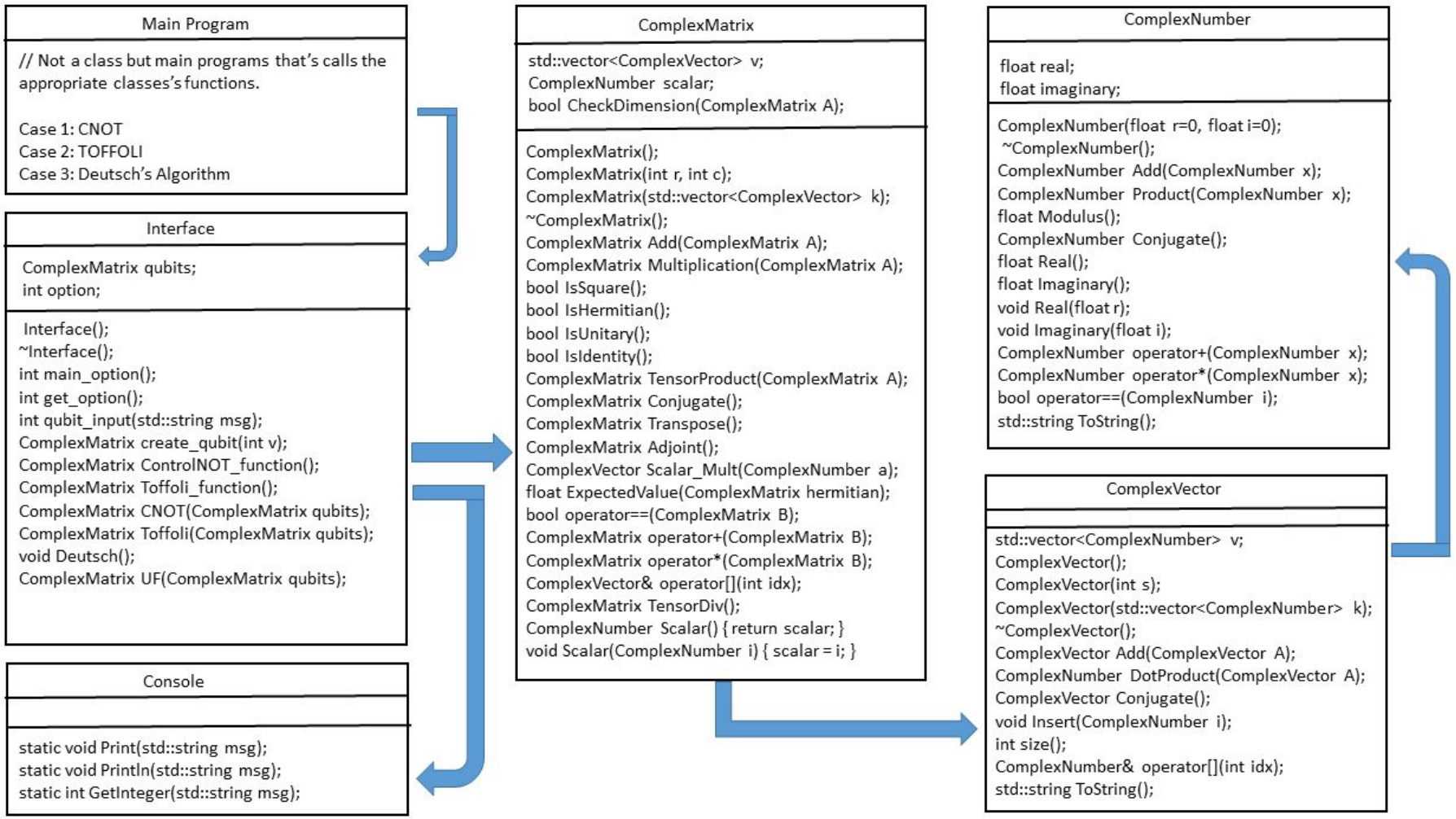
The project helped us learn how quantum computing can be incorporated in a classical computer. Implementation is much easier once the groundwork is laid and most of the basic functions such as adding and multiplying vectors were incorporated into the program. This project helped us have a deeper understanding of the usefulness of Quantum Computers, as displayed in Deutsch's algorithm we can see that the classical computer needs to perform 2 evaluations to determine if the function is constant or balanced, while a quantum computer would only require one.

Improvement

We tried to implement everything from scratch without using any embedded libraries such as Complex Numbers, this gave us the most control over our data structures but took additional time and planning. In order to eradicate any possible errors, some test cases were incorporated that ensures the basic functionality of the program. The boundary test cases were embedded but some other checks could have been incorporated. Additionally, the interface works but was lackluster, which could be easily fixed given time.

Issues

Handling the factorization of complex matrices to reflect their scalars proved difficult, since it involved a more complicated data structure for handling said factors. We know for sure that all user input cases were tested, and any invalid input would not crash the program.


Main Program
// Not a class but main programs that's calls the appropriate classes's functions.
Case 1: CNOT
Case 2: TOFFOLI
Case 3: Deutsch's Algorithm
Interface
ComplexMatrix qubits;
int option;
Interface();
~Interface();
int main_option();
int get_option();
int qubit_input(std::string msg);
ComplexMatrix create_qubit(int v);
ComplexMatrix ControlNOT_function();
ComplexMatrix Toffoli_function();
ComplexMatrix CNOT(ComplexMatrix qubits);
ComplexMatrix Toffoli(ComplexMatrix qubits);
void Deutsch();
ComplexMatrix UF(ComplexMatrix qubits);
Console
static void Print(std::string msg);
static void Println(std::string msg);
static int GetInteger(std::string msg);
ComplexMatrix
std::vector<ComplexVector> v;
ComplexNumber scalar;
bool CheckDimension(ComplexMatrix A);
ComplexMatrix();
ComplexMatrix(int r, int c);
ComplexMatrix(std::vector<ComplexVector> k);
~ComplexMatrix();
ComplexMatrix Add(ComplexMatrix A);
ComplexMatrix Multiplication(ComplexMatrix A);
bool IsSquare();
bool IsHermitian();
bool IsUnitary();
bool IsIdentity();
ComplexMatrix TensorProduct(ComplexMatrix A);
ComplexMatrix Conjugate();
ComplexMatrix Transpose();
ComplexMatrix Adjoint();
ComplexVector Scalar_Mult(ComplexNumber a);
float ExpectedValue(ComplexMatrix hermitian);
bool operator==(ComplexMatrix B);
ComplexMatrix operator+(ComplexMatrix B);
ComplexMatrix operator*(ComplexMatrix B);
ComplexVector& operator[](int idx);
ComplexMatrix TensorDiv();
ComplexNumber Scalar() { return scalar; }
void Scalar(ComplexNumber i) { scalar = i; }
ComplexNumber
float real;
float imaginary;
ComplexNumber(float r=0, float i=0);
~ComplexNumber();
ComplexNumber Add(ComplexNumber x);
ComplexNumber Product(ComplexNumber x);
float Modulus();
ComplexNumber Conjugate();
float Real();
float Imaginary();
void Real(float r);
void Imaginary(float i);
ComplexNumber operator+(ComplexNumber x);
ComplexNumber operator*(ComplexNumber x);
bool operator==(ComplexNumber i);
std::string ToString();
ComplexVector
std::vector<ComplexNumber> v;
ComplexVector();
ComplexVector(int s);
ComplexVector(std::vector<ComplexNumber> k);
~ComplexVector();
ComplexVector Add(ComplexVector A);
ComplexNumber DotProduct(ComplexVector A);
ComplexVector Conjugate();
void Insert(ComplexNumber i);
int size();
ComplexNumber& operator[](int idx);
std::string ToString();