Quantum Algorithm Simulator (QAS)

Kevin Coltin

April 2014

Contents

1 Introduction

QAS is a library of functions and classes for simulating the action of a quantum computer using C++. It contains two classes—`QubitSystem` and `QuantumGate`—that simulate the two primary structures used in quantum computing, quantum bits (qubits) and gates that operate on systems of one or more qubits.

Encapsulation is used to enforce the limitations that make programming for a quantum computer fundamentally different than for a classical computer. The operations that can be performed on an object of the class `QubitSystem`, which represents a system of one or more possibly entangled qubits, are essentially limited to applying the action of a unitary quantum gate on the system and "observing" the system, which collapses it to a pure state. This allows the user to design, implement, and test algorithms in a natural way under the constraints inherent to quantum computing.

The QAS library has an extremely compact and intuitive interface, making it easy to learn and use for anyone familiar with C++. It fits easily within a larger C++program, so that mixing classical and quantum programming is effortless. For examples of use, see the unit tests in the `test` directory.

2 The QubitSystem class

An object of the QubitSystem class represents a system of one or more qubits. Representing multiple qubits in a single object—rather than the more obvious object-oriented approach of having each instance of the class be a single qubit—makes it possible to realistically model the phenomenon of entanglement, wherein performing operations or measurements on one or more qubits may affect arbitrarily many other qubits.

The data members of a QubitSystem object are as follows:

- n: Nonnegative integer representing the number of qubits in the system.

- coeffs: Array of complex numbers representing the coefficients on each basis state of the system. Since a system of n qubits has 2^n possible basis states, the length of coeffs is always 2^n. The kth entry, coeffs[k], represents the complex coefficient (the amplitude) on the basis state whose individual qubit values form the binary integer k represented by n binary digits. For example, if $\texttt{n} = 4$, then coeffs[0] would be the coefficient on the basis state $|0000\rangle$, coeffs[13] would be the coefficient on the basis state $|1101\rangle$, and so on. The sum of the squares of the amplitudes is restricted to always equal one.

The following constructors exist for the QubitSystem class. The class also has a corresponding init method for each constructor, so calling

```
QubitSystem q;
q.init(args);
```

is always equivalent to calling

```
QubitSystem q(args);
```

- QubitSystem (): Default constructor. Creates a system containing a single qubit in the state $|0\rangle$.

- QubitSystem (int n): Creates a system containing n qubits in the state $|00...0\rangle$.

- QubitSystem (int n, int state): Creates a system containing n qubits in the pure state representing the integer state in binary digits. For example, calling QubitSystem q(3, 2) would initialize a system in the state $|010\rangle$.

- QubitSystem (int n, const std::string state): Creates a system containing n qubits in the pure state represented by the binary string state. For example, calling QubitSystem q(4, "1100") would initialize a system in the state $|1100\rangle$.

Note that the copy constructor and copy assignment operator for the class are not public methods: this prevents violations of the no-cloning theorem, which states that it is impossible to copy the state of an arbitrary system of qubits.

The other public methods of the class are:

- `int N ()`: Returns the value of the private member variable `n`, the number of qubits in the system.
- `int measure ()`: Performs a measurement on the system of qubits. This collapses the system to a pure state, where the amplitudes (represented by the array `coeffs`) are zero on all states except the observed state. The kth state (i.e., the state whose qubits form the binary digits of the integer k) is observed with probability given by $|$`coeffs[k]`$|^2$.

  For example, suppose that `q` is a `QubitSystem` with `q.n`$= 2$ and `q.coeffs`$= \{1/\sqrt{2}, 0, 0.5 + 0.5i, 0\}$, i.e. a system of two qubits in the mixed state $1/\sqrt{2}|00\rangle + (0.5 + 0.5i)|10\rangle$. Then, calling `q.measure()` would return the integers 0 or 2 each with 50% probability.
- `int measure (int bit_index)`: Performs a measurement on a single qubit in the system, the `bit_index`th qubit in order from left to right, starting with one. E.g., if the system `q` is in the pure state $|0110\rangle$, then `q.measure(1)` or `q.measure(4)` will return 0, and `q.measure(2)` or `q.measure(3)` will return 1.
- `std::string smeasure ()`: Like calling `measure()`, but returns the state as a string of binary digits rather than an integer. E.g., calling `q.smeasure()` when `q` is in the pure state $|010\rangle$ will return the string `"010"`.
- `int * ameasure ()`: Like calling `measure()`, but returns the state as an array of binary digits rather than an integer. E.g., calling `q.smeasure()` when `q` is in the pure state $|010\rangle$ will return the array `{0, 1, 0}`.

3 The `QuantumGate` class

An object of the class `QuantumGate` represents a single quantum gate that operates on a system of one or more qubits.

A gate is represented by a unitary matrix M such that performing the action of the gate to a system of qubits is equivalent to left multiplying the vector of coefficients on the basis states of the system by M. Suppose for example that q is a system of two qubits in the state $a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle)$, and G is a gate with matrix

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(the swap gate). When G acts on q, the coefficient vector (a, b, c, d) is multiplied by M, which changes the state of q to be $a|00\rangle + c|01\rangle + b|10\rangle + d|11\rangle$.

Objects of the class `QuantumGate` have the following data members:

- `n`: Positive integer representing the number of qubits that the gate operates on. For instance, the swap gate in the example above operates on two qubits.

- `matrix`: A two-dimensional array of complex numbers representing the coefficients of the unitary matrix that represents the action of the quantum gate. The dimensions of the matrix are always $2^n \times 2^n$, since a gate operating on a system of n qubits must operate on a vector of amplitudes of 2^n possible basis states.

The `QuantumGate` class includes the following constructors:

- `QuantumGate (int n)`: Default constructor for a quantum gate operating on `n` qubits. The gate is initialized to the identity gate—that is, the gate represented by the $2^n \times 2^n$ identity matrix, whose application has no effect on a qubit system.

- `QuantumGate (int n, std::complex<double> **matrix, bool byref)`: Creates a gate operating on `n` qubits defined by the given matrix. The second argument must be a two-dimensional array of size $2^n \times 2^n$. If the optional argument `byref` is omitted or is `true`, then the member `this->matrix` is assigned by reference to equal the argument `matrix`. If `byref` is `true`, then `this->matrix` is a copy of the argument `matrix`.

- `QuantumGate (int n, const complex<double *vector)`: Creates a gate operating on `n` qubits, defined by the matrix whose coefficients are given by the argument `vector` in row-major order. `vector` must be a complex array of length 2^{2n}.

The class also implements a copy constructor and copy assignment operator that work in the usual way, i.e. by creating a new `QuantumGate` whose fields `n` and `matrix` are equal (by value, not reference) to those of the original gate.

The class also contains the following public methods and friend functions:

- `int N ()`: Returns the value of the private member variable `n`, the number of qubits that the gate acts on.

- `void set (int i, int j, std::complex<double> val)` and `std::complex<double> operator() (int i, int j)`: Respectively, setter and getter methods for the `(i,j)`th entry of `matrix`.

- `QuantumGate H ()`: Returns the quantum gate defined by the conjugate transpose of the current gate's matrix. Analogous to taking the conjugate transpose of the member variable `matrix`.

- `void act (QubitSystem *q, int index)` and the operator `*`: The method `act` applies the action of the quantum gate on a (proper or improper) subset of the given system of qubits. Specifically, if `q` is a system of n qubits and the gate operates on m qubits (where $m \leq n$ of necessity), then the action of the gate is applied to the `index`th through (`index`$+m-1$)th qubits in the system. If the optional argument `index` is omitted, it is set to 1.

The operator * is shorthand for calling `act` with one argument, that is:

```
   G * q;
```

is equivalent to

```
   G.act(&q);.
```

Note that the * operator takes an address `QubitSystem&` whereas the `act` method takes a reference `QubitSystem*`. The * operator is to be preferred whenever the second argument is unnecessary as it is closer to the mathematical notation.

For example, let `G` be the swap gate as defined above, and let `q` be a system of qubits in the state

$$a|000\rangle + b|001\rangle + c|010\rangle + d|011\rangle + e|100\rangle + f|101\rangle + g|110\rangle + h|111\rangle.$$

Then, taking

```
   G.act(&q);
```

or

```
   G * q;
```

would put the system into the state

$$a|000\rangle + b|001\rangle + e|010\rangle + f|011\rangle + c|100\rangle + d|101\rangle + g|110\rangle + h|111\rangle,$$

while taking

```
   G.act(&q, 2);
```

instead would put the system into the state

$$a|000\rangle + c|001\rangle + b|010\rangle + d|011\rangle + e|100\rangle + g|101\rangle + f|110\rangle + h|111\rangle.$$

Note: The method `act` (and hence the * operator) always checks that the gate's matrix is unitary before applying it to a system of qubits. If the programmer creates a gate that is not unitary, an exception will be thrown whenever the gate is used.

- Various linear-algebra operations on quantum gates: The operators +, -, *, and % are respectively used to add, subtract, multiply using standard matrix multiplication, and multiply using the tensor product two quantum gates. The operator ^ raises a gate's matrix to an integer power, and the function `QuantumGate tensor_pow (const QuantumGate &g, int e)` returns the tensor product of a gate `g` tensored with itself `e` times.

4 The file gate_factory.cpp

The file `gate_factory.cpp` contains factory functions for creating various types of commonly used quantum gates, such as the Pauli-X gate (also known as the quantum NOT gate), swap gate, and Toffoli gate. Each function in the file is explained by a brief comment preceding it.

5 Quantum algorithms

The following algorithms are included in the QAS system:

- `void qft (QubitSystem *q)`: Performs the quanum Fourier transform on a system of qubits. This is equivalent to performing the discrete Fourier transform in-place on the vector of amplitudes `q->coeffs`.

- `int grover_search (const std::string &match_text, const std::string *list, int n)`: Uses Grover's search algorithm to search the array of strings `list` for the string `match_text`. It returns the index `k` such that the string `list[k]` is the same as `match_text`. If two or more such values `k` exist, it may return any one of them. If no element of `list` matches `match_text`, returns -1.

- `int grover_invert (int (*f) (int), int y, int n)`: Uses Grover's algorithm to invert a function. The function `f` must map a sequence of `n` bits (which can be thought of as the binary representation of an integer between 0 and $2^n - 1$ inclusive) to another sequence of bits of any positive length (which can again be thought of as a binary integer). Calling `grover_invert(f, y, n)` returns the integer $x \in \{0, 1, \ldots, 2^n - 1\}$ such that $f(x) = y$. If two or more such values x exist, it may return any one of them. If no such x exists, it returns -1.

- `int shor_factor (int n)`: Implements Shor's factorization algorithm. Given an integer n, it returns a non-trivial prime factor of n if n is composite. If n is prime, returns 1.