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 `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 $|1100\rangle$.

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 $|\texttt{coeffs[k]}|^2$.

  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

4 The file `gate_factory.cpp`

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.