Adiabatic Quantum Computing

C/CS/PHY 191

University of California, Berkeley

Vlad Goldenberg

Maciek Sakrejda

Outline

- Quantum Adiabatic Theorem
- Adiabatic Theorem applied to Quantum Computation
- Actual Implementation in “theory”
- Simulation using classical computing
- Object Oriented approach – JAVA

What is it?

- Based on *Adiabatic Theorem* of QM: A quantum system in its ground state will remain in its ground state provided that the hamiltonian H is varied slowly enough.
- Also, a quantum system whose energies are quantized that starts in the nth energy state will exist in the nth energy state provided that the hamiltonian is varied slowly enough.
- Vary the Hamiltonian slowly from an initial to final state so that it acts as though a unitary transformation occurred on the initial state, bringing it to a final state during some time **T**.

Implementation

- **Initialize register to desired input qubits. This is the initial state with which the computer will calculate the output state.**
- **Vary the Hamiltonian slowly toward the final Hamiltonian whose eigenstates encode the desired final states.**

Single qubit gate:

Let |i>, |i'> be the basis eigenstates input. In non-adiabatic quantum computation, we apply a gate, which is just a unitary transformation, U_t to the basis states to get the output states:

$$U_t|i\rangle = |f\rangle$$

$$U_t|i'\rangle = |f'\rangle$$

Implementation

We can think of {|i>, ||i'>} and {|f>,|f'>} as eigenbases of some initial and final Hamiltonians, respectively. Call these Hamiltonians $\mathbf{H_0}$ and $\mathbf{H_1}$, respectively. Then we can say:

$$H_0|i\rangle = E_i|i\rangle$$
$$H_0|i'\rangle = E_i'|i'\rangle$$
$$H_1|f\rangle = E_f|f\rangle$$
$$H_1|f'\rangle = E_f'|f'\rangle$$

In order to manipulate qubits using the adiabatic theorem, the Hamiltonian must be varied slowly from the initial to the final state. Let **T** be the final time, at the end of the process. Let **t** be the independent time variable. Define **s = t/T** such that during the evolution of the system, **0 < s < 1**. Then the Hamiltonian is a function of **s** such that:

$$H(s) = (1-s)H_0 + sH_1$$

Implementation

We see that if we apply **H(s)** on the input state until we reach **s = 1**, we will in effect be applying the unitary transformation $\mathbf{U_t}$ on the input state.

The form of **H(s)** suggested above is not always the one adequate for the implementation, as for example the two qubit CNOT gate, which requires the form:

$$H(s) = (1-s)H_0 + sH_1 + As(1-s)H_{01}$$ (Ali, Andrecut)

With **A=1**. This is necessary to meet the condition for the adiabatic theorem:

$$\delta_{\min} = \min[E_j(s) - E_k(s)]$$

$$\Delta_{\max} = \max_{s\in[0,1]} \left\| \frac{d}{ds} H(s) \right\|$$

$$T = \frac{\Delta_{\max}}{\varepsilon \delta_{\min}^{2}}$$

Discrete Simulation

$H(0)|i\rangle = E(0)|i\rangle$

.

.

.

$H(n)|t\rangle = E(n)|t\rangle$

.

.

.

$H(f)|f\rangle = E(f)|f\rangle$

$H(0)|i'\rangle = E'(0)|i'\rangle$

.

.

.

$H(n)|t'\rangle = E'(n)|t'\rangle$

.

.

.

$H(f)|f'\rangle = E'(f)|f'\rangle$

Classical Simulation

Object Oriented Programming

- Logic divided into components (objects)
- Each object has
  - State information about itself
  - Functions it can perform (methods)
- Core functionality is implemented through a set of objects interacting with each other

Classical Simulation

- Adiabatic quantum simulation requires two main components
  - N-Qubit states
  - N-Qubit operators
- Main functionality is a set of interactions between these

Classical Simulation

- N-Qubit States

```
public QState(int n) {
        numQubits = n;
        coeffs = new Complex[1 << n];
}
public QState(Complex [] coeffs) {
        this.coeffs = coeffs;
        int i = 0;
        // Calculates the log (base 2) of the length of the array--i.e.,
        // the number of qubits represented by the array of coefficients.
        for (int temp = 1; (temp & coeffs.length) == 0; temp <<= 1, ++i);
        this.numQubits = i;
}
```

Classical Simulation

- N-Qubit States (continued)

```
public QState tensor(QState that) {
        QState ret = new QState(this.numQubits + that.numQubits);
        for (int i = 0; i < this.coeffs.length; ++i) {
                for (int j = 0; j < that.coeffs.length; ++j) {
                        ret.coeffs[(i << that.numQubits) + j] =
                                this.coeffs[i].times(that.coeffs[j]);
                }
        }
        return ret;
}
```

Classical Simulation

- Next steps
  - Finish implementing Operator objects
  - Implement eigenvalue code
    - External library; probably Colt - Open Source Libraries for High Performance Scientific and Technical Computing in Java
  - Implement top-level logic

Classical Simulation

- Scope
  - Quantum properties are only simulated
  - Performance
  - Helpful in understanding system, but not very useful beyond this