Transparent simulation of quantum algorithms

A.A. Pomeransky, D.L. Shepelyansky and O.V. Zhirov

December 18, 2005

Abstract

We present a software library which allows to implement quantum computing algorithm in very transparent manner.

1 Introdiction.

Currently quantum computers are on the very beginning of practical implementation. A few qubits created in laboratories are still unsufficient to study quantum algorithms in practice and gain their advantage. A great challenge is going on search of a physical system which is efficient in practical quibits implementation, with unavoidable imperfections and external perturbations caused by enviroment. In parallel, a great activity is devoted to understanding the stability of quantum algorithms to imperfections of quantum qubits and gates, and to practical testing of them on common classical computers.

Each of quantum algorithm consists of many elements, a linear sequence of quantum gates, acting on quantum register. Each gate can be implemented by a piece of simple code, but the pattern of all these gates can be rather complicated, and final program can be huge. This makes the writing of such programs and their debugging be a difficult process, as well as their modifications can in principle break the code completely.

In fact, the object-oriented approach, which was proposed in programming about twenty years ago, can provide a very effective and elegant solution of this problem. Below we start with a paradigm, which turns the problem into a child game with bricks. In this way we introduce a set of "bricks", the objects which implement the quantum register and quantum gates. These objects are written in C++ programming language, and can be easily extended. As a result, the most code is incapsulated in these objects; the rules of composition of these objects are very simple, and the final code, the implementation of quantum algorithm becomes very compact and pictural.

2 The paradigm.

The heart of quantum computer is a *quantum register*, which is prepared in some quantum state $|X\rangle$. The process of quantum computation consist of a chain (sometimes very long!) of unitary transformations U applied to this register subsequently:

$$|X'\rangle = U_M U_{M-1} \dots U_2 U_1 \, |X\rangle \tag{1}$$

and final state $|X'\rangle$ contains some results of calculations. For a qubit the quantum state is described by two-component complex array

$$|q\rangle = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

while the quantum state of the register containing n_q qubits, which is a direct product of qubit quantum states

$$|X\rangle = |q_1\rangle \otimes |q_2\rangle \otimes \dots \otimes |q_{n_q}\rangle$$

is described by 2^{n_q}-component complex array. In turn, the gate matrices U_i, are $2^{n_q} \times 2^{n_q}$ arrays of complex numbers, too. One can see that dimensions of arrays grows exponentially with the number of qubits n_q: this is a main origin of exponential slowing down of simullations of quantum algorithms on common “classical” computers.

The number of quantum gates M in quantum algorithms is in general a power-like function of n_q. In fact, all of the possible gates can be reduced into very restricted set of elementary gates, each of that affects one or two qubits, leaving the rest untouched.

As a result, the classical code, simulating the quantum computation is a lengthy chain of several gates composed in a very complicated inhomogeneous pattern (very similar to DNA chain!). In order reduce mistakes in writing such code, we invent a simple human interface to objects of simulation.

2.1 Main objects and their properties.

The main object is the quantum register, or wavefunction of its quantum state. Corresponding type defined in header file `qubits.h` is `QBitsWaveFunction`:

```
#include ''qubits.h''
QBitsWaveFunction Psi(nq);   // nq is qubits number
```

One can perform on the quantum state the following operations:

```
QBitsWaveFunction Psi2(Psi)  // make a clone of the state Psi

QBitsWaveFunction Psi3(nq);
Psi3(nq)=Psi;                // copy of state Psi

double<complex> z;
...
Psi.RescaleBy(z);            // rescale by a complex factor;
```

```
Psi3.Sum(Psi2,Psi);          // calculate a sum of two wavefunctions

Psi.Allign();                // set all qubits up
```

One can perform multiplication of j-th quibit state by any Pauli matrix σ:

```
Psi.SigmaX(j);
Psi.SigmaY(j);
Psi.SigmaZ(j);
```

the Walsh-Hadamard transformation:

```
Psi.WH_tr(j);
```

and rotate j-th qubit state by angle Phi:

```
Psi.RotateQBit(j,Phi);
```

Next operation is useful to simulate interaction between two qubits:

```
Psi.InteractQubits(j1,j2,g);
```

which in fact corresponds to applying to register state the operator

$$\exp(-ig\sigma_z^{(j1)}\sigma_z^{(j2)})\,|X\rangle\,.$$

Next important operations are *Control-Not* and *Control-Control-Not* gates applied to qubits $j1$, $j2$ and $j3$:

```
Psi.Cnot(j1,j2);
Psi.CCnot(j1,j2,j3);
```

The contents of `Psi` can be printed by command `Psi.print();` to stdout, into three collumn form: first is plain index of the complex array, second and third are real and imaginary part of amplitude. The correspondence between plain index and states of qubits is:

$$j = s_1 + 2s_2 + \ldots + 2^{n_q-1}s_{n_q}$$

where s_i equal 1 and 0 for "up" and "down" states of i-th qubit, respectively.

In fact, in mutiplication and sum one can use more simpler interface:

```
Psi2=Psi*z;   // similar to Rescale by, but content of Psi remains unchanged
Psi2=z*Psi;

z=Psi1*Psi2;  // scalar product
Psi3=Psi1+Psi2; // sum of two states
```

2.2 Advanced interface to objects: basic operations and gates.

Addressing to gates as properties of wavefunction is not convenient. Instead we introduce a set of new objects, which can simplify appearence of code. Most of their definitions are collected in the header file `qAlgebra.h`. Let us start with prepared quantum state object Psi and applied to it several described above operations:

```
#include ''qAlgebra.h''
QBitsWaveFunction Psi(nq);   // nq is qubits number
...
Psi << X(j1,j2) << WH(j3,j4) << rotQBitBy(angle,j5,j6) << Cnot(ic,iq);
Psi << X(m1) << WH(m2) << rotQBitBy(angle,m3) << CCnot(ic1,ic2,iq);
```

At first single line we perform on Psi subsequently:

1. Apply matrix σ_x to qubits from $j1$-th to $j2$-th;
2. Apply Walsh-Hadamard transformation to qubits from $j3$-th to $j4$-th;
3. Rotate qubits from $j5$-th to $j6$-th by angle "`angle`";
4. Apply Control-Not gate, with control qubit `ic` and working qubit `iq`.

The single qubit operations are presented by the second line, where `m1`, `m2` and `m3` are indices of qubits; at the end of second line we give an example of Control-Control-Not gate operation. After all the object `Psi` contains the cumulative result of both lines.

The syntax of this interface is very simple and self-explaning: sequence of operations from left to right is very natural for human reading and direction of symbol "`<<`" point to the object, to which the action is applied.

3 Extentions.

3.1 Qubits interactions.

Interactions of qubits depend on their mutual positions. In our simulations we have assumed that they are placed on sites of simple rectangular lattice, and interactions among them and qubits imperfections are described by Hamiltonian

$$H_s = \sum_i a_i \sigma_i^z + \sum_{ij} b_{ij} \sigma_i^x \sigma_j^x$$

where couplings a_i, b_{ij} are random numbers, distributed homogeneously inside $[-\alpha, \alpha]$ and $[-\beta, \beta]$ respectively.

Implementation of object, providing simulation of these interactions is located in the header files `qLattice.h` and `qAlgebra.h`:

```
#include ''qLattice.h''
#include ''qAlgebra.h''

Lattice L(Lx,Ly,Alpha,Beta);   // create a lattice of size Lx by Ly qubits }

L.Rand();                      // generate a new set of random couplings
L.Perturb(Psi);                // introduce perturbations into state Psi
                               // by time evolution with Hamiltonian
```

In fact, the best way to introduce perturbation is

```
Psi << Pert();
```

3.2 Interaction with enviroment.

The underlying mechanism of decoherence is assumed a random flip of a qubit $|1\rangle \rightarrow |0\rangle$, with a rate Γ. The necessary initialization of the process is creating an object of type `Jump`:

```
#include ''qJump.h''
Jump(Gamma*dt);
```

Then one can simply act on the wave function `Psi`:

```
Psi << Jump(Gamma*dt);
```

As a result we get a state which either is the orignal one but damped by the probability amplitude of no qubit be flipped, or it start a new quantum branch.

4 Applications.

This library is used in simulations of Grover problem with imperfections (subdirectory `Grover`) and decoherence effects, caused by interactions with enviroment (subdirectory `DecoGr`). The former one is an earlier version of the library, where all objects are included in a single header file. The latter is the last version, actually described in this document.