Annales UMCS Informatica AI 2 (2004) 47-56

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Simulations of quantum systems evolution with quantum-octave package

Piotr Gawron, Jarosław A. Miszczak*

Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences, Bałtycka 5, 44-100 Gliwice, Poland

Abstract

Article presents package of functions for GNU Octave computer algebra system. Those functions were designed to perform simple but not necessary efficient simulations of quantum systems, especially quantum computers. The most important feature of this package is the ability to perform calculations with mixed states.

We describe application of quantum-octave package for simulation of Grovers algorithm, which is one of the most important quantum algorithms. We also list other possible calculations, which can be performed with this package.

1. Introduction

The simulation of quantum computers is an important issue in quantum informatics. No one expects that physical quantum computers will be built in less than 20-30 years. That is why simulation of quantum systems is the only way for testing existing quantum algorithms and observing quantum effects which are connected with processing and transferring of quantum information.

In this article we present quantum-octave [1] package of functions for GNU Octave [2,3] computer algebra system (CAS), which allows to observe unitary evolution of quantum systems. This package was developed by authors as a tool for simple calculations on pure and mixed quantum states. GNU Octave was selected as a target CAS because of its flexibility and portability. Described package allows performing many types of calculations. One of its usages is calculation of entanglement measures for pure and mixed states. It can also be used to observe quantum errors and application of quantum error correcting codes.

* Corresponding author: *e-mail address*: miszczak@iitis.gliwice.pl

2. Description of package

In this section we briefly describe functions provided by quantum-octave package and present its possibilities. For a more detailed description of function please refer to on-line help distributed with source files [1]. We also present implementation of Deutsch algorithm [4,5,6] as an example of available commands. In the next section we present implementation of Grovers algorithm [7-9] which allows to perform computation on mixed states.

Functions described here were present in version test3 of quantum-octave. Most of them probably won't be changed in near future, but it is possible that some internals will be changed.

GNU Octave is computer algebra system (CAS) primary designed to perform numerical rather then symbolical calculations. Basic data structure in Octave is matrix, but it is also possible to operate on C-like structures. Syntax of Octave language is very similar to MatLab [10] language. GNU Octave is distributed as a GPL [11] software and it can be used in a wide range of UNIX like operating systems, including Linux.

Low level quantum computation is based on finite dimensional matrix calculus. GNU Octave was chosen by authors to create quantum computation package, because its language was developed to operate on matrices and it implements wide range of functions useful for matrix-based computing. That was important because quantum-octave was meant to be a tool for performing simple simulations quickly. Because simplicity was primary advantage of this package not every operation is implemented in the most efficient way. It is possible to simulate up to 10 qubits using quantum-octave.

Most of existing tools designed for testing quantum algorithms are based on quantum gate array model [5,6]. According to our knowledge only QuCalc [3,12] package for Mathematica [14] is an exception. In this model quantum operations are described as a unitary matrix and the states of quantum computers are described as unit vectors in Hilbert space. Computation process is described as a matrix multiplication. For realistic physical experiments it is important to include quantum errors. From theoretical point of view it is also interesting to observe quantum effects connected with operations on mixed quantum states.

Another interesting issue in quantum information is the problem of entanglement and its connections with mixed states [15-17]. There is no unique entanglement measure for mixed states, and most of existing measures cannot be used for numerical calculations. But it is possible to calculate some measures based on the numerical properties of density matrix.

For those reasons we include mixed states (density matrices) formalism in quantum-octave.

Package quantum-octave allows a user to operate on different levels of abstraction. Lower level functions allow to assembly complex quantum gates,

including gates with arbitrary controlled and target qubits, to prepare pure states and arbitrary mixtures of pure states. The examples of low level commands are:

- **`Ket`** – produces pure quantum state,
- **`State`** – produces density matrix for a given pure state,
- **`MixStates`** – produces mixture of density matrices,
- **`ProductGate`** – allows to build many qubit gates,
- **`ControlledGate`** – allows to build controlled gate with many control and target qubits,
- **`CNot = ControlledGate(2,Not,[1],[2])`**.

The last equality indicates that `CNot` is an abbreviation for `ControlledGate(2,Not,[1],[2])`. Function takes `ControlledGate` for arguments. First argument defines size of output gate and second elementary gate to be controlled. Any one-qubit gate can be used as a second argument for `ControlledGate` function. Third and fourth arguments of this function represent controlled and target qubits respectively and function won't work properly if they overlap.

```
octave:1> s1 = State(Normalize(Ket([1,0])+Ket([0,1])))
                          s1 =

              0.00000   0.00000   0.00000   0.00000
              0.00000   0.50000   0.50000   0.00000
              0.00000   0.50000   0.50000   0.00000
              0.00000   0.00000   0.00000   0.00000

octave:2> s2 = State(Normalize(Ket([1,1])+Ket([0,1])))
                          s2 =

              0.00000   0.00000   0.00000   0.00000
              0.00000   0.50000   0.00000   0.50000
              0.00000   0.00000   0.00000   0.00000
              0.00000   0.50000   0.00000   0.50000

              octave:3> s3 = MixStates (s1,s2)
                          s3 =

              0.00000   0.00000   0.00000   0.00000
              0.00000   0.50000   0.25000   0.25000
              0.00000   0.25000   0.25000   0.00000
              0.00000   0.25000   0.00000   0.25000
```

Fig. 1. Example of mixing pure states using **`Normalize`** and **`MixStates`** functions

`MixStates` command was designed to produce arbitrary mixture of pure quantum states. One should remember that superposition of pure states is not

normalized and function `Normalize` should be used before mixing states to preserve proper interpretation of results. Numerical values of mixture coefficients do not have to sum to one – proper density matrix will be produced automatically.

In Fig. 1 example of `MixStates` command is presented. First we produce density matrices `s1 s2`, for pure states using function `State`. Last command produces uniform mixture of states `s1` and `s2`.

Second group of commands allow controlling evolution of quantum states. Commands from this group are:

– `Measure`
– `Evolve`

Function `Evolve` implements unitary evolution of mixed quantum states and it is performed according to the standard rule [13]

$$P(t)\,AP(0)\,A^{+}, \tag{1}$$

where *P(0)* is state of the system in time $t = 0$ and a unitary matrix A represents evolution and A^{+} represents a conjugate matrix. State of the system after time t is represented by matrix $P(t)$.

Function `Measure` implements orthogonal measurement and it returns a vector representing probability distribution on space of results of measurement. This distribution can be plotted using `PlotProbs` command described below. Only orthogonal measurements are supported, but authors are planning to extend function `Measure` in future.

Package also provides commands useful for visualization of the results. Some of them are based on GNU Octave command `plot`, which allows producing two dimensional plots of functions. Commands from this group are:

– `PrintAmps`
– `PrintProbs`
– `PlotAmps`
– `PlotProbs`

Commands `PrintAmps` and `PlotAmps` for a given ket vector return numerical values of amplitudes for every base state. First command returns list of values while second one produces a two dimensional plot. One should note that in the second case real or imaginary part of amplitude can be plotted and behavior of function is determinated by its third argument.

`PrintProbs` and `PlotProbs` can be used for visualization of probability distributions obtained with `Measure` command. Those functions accept vector representing probability distributions as an argument.

In Fig. 2 source code for Deutsch algorithm is presented. Function `deutsch`[1] allows starting quantum calculation from arbitrary mixed state. In our case we start from ground state `State((Ket [0,0]))`.

```
# input: identifier of function and initial state
# output: state after execution of Deutsch's algorithm
function ret = deutsch( num, state )
# preparation of gates corresponding to
# set of functions {0,1}->{0,1}
f1 = Id(2);
f2 = kron(Id(1), Not);
f3 = CNot(2, [1], [2]);
f4 = kron(Not, Id(1))* CNot(2, [1], [2])* kron(Not, Id(1));
# choice of function
if (num == 1)
f=f1;
elseif (num == 2)
f=f2;
elseif (num == 3)
f=f3;
elseif (num == 4)
f=f4;
else
# error: no such function
endif
# preparation of algorithm gate
algorithm = kron(H,H) * kron(Id(1),Not) * f * kron(H,
Id(1));
# execution of Deutsch's algorithm
ret = Evolve(algorithm,state);
endfunction
```

Fig. 2. Deutsch algorithm in quantum-octave

Fig. 3 presents probability distributions of results after final measurement in Deutsch's algorithm. Plots were obtained with `PlotProbs` function.

[1] See directory `examples` in `quantum-octave` distribution.

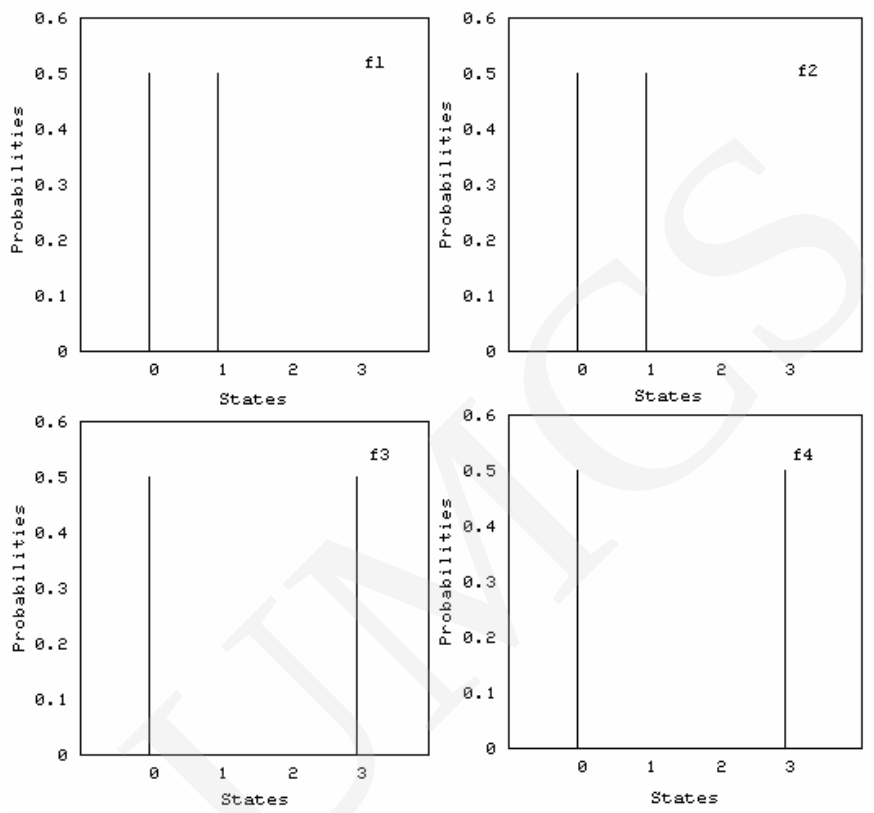

Fig. 3. Probability distributions after execution of Deutsch algorithm for all four functions

3. Simulation of Grover's algorithm

In [18] requirements for physical realization of useful quantum computer are listed. One of them is the requirement that starting state for execution of any algorithm should be ground state of the system we operate on. In this section we present result of Grover's algorithm simulation for different initial states.

Grover's algorithm is one of the most important quantum algorithms. Potentially it could be used to search an unsorted database. Detailed description of this algorithm can be found in [7-9]. We present here implementation of Grover's algorithm in quantum-octave which allows observation of quantum errors propagation during execution of algorithm.

Source code in Fig. 4 presents implementation of function `grover`. This function accepts integer number as a first parameter and a mixed state as a second parameter. This number could be interpret as database key. First parameter represents number for which probability of measurement will be increased. Second argument must be a proper mixed state. As an example pure ground state modified by some mixture of pure base states can be used.

State `sm` defined below is a mixture of 5 base states

`sm = MixStates (`*x0,* s0, *x1,* s1, *x2,* s2, *x3,* s3, *x4,* s4`),` (2)

parameters *x0, x1, x2, x3,x4* can be changed to change statistical contents of state in canonical base. In the first example we have *x0=0.9* and *x1=x2=x3=x4=0.1* and in the second example we put *x0=0.5* and *x1=x2=x3=x4=0.2.* Note that those coefficients do not sum to 1 because they are implicitly normalized.

States *s0 ,..., s4* are defined as:

```
s0 = State (Ket([0,0,0])),
s1 = State (Ket([0,0,1])),
s2 = State (Ket([0,1,0])),
s3 = State (Ket([0,1,1])),
s4 = State (Ket([1,0,0])).
```

```
# input: num – quantum database key, state – initial state
  # output: state after execution of Grover's algorithm
          function ret = grover( num, state )
                # get number of qubits
             qubits = log2(size(state)(1));
                # list of all qubits
                tvec = [1:qubits];
           # number of Grover's iterations
        k = ceil((pi/8)*sqrt(size(state)(1)));
         # Welsh-Hadamrd gate of register size
          bigh = ProductGate(qubits,H,tvec);
                # Grover iterations
# Step 1: preparation, set uniform distribution on register
                ret = Evolve(bigh,state);
                # Step 2: iterate k-times
                      for i = 1:k
       # Step 2.1: perform oracle gate on register
          ret = Evolve(oracle(num, qubits),ret);
     # Step 2.2: perform diffusion gate on register
          ret = Evolve(diffuse(qubits),ret);
                        endfor
                      endfunction

          # input: num – quantum database key,
        # qubits – number of qubits in register
  # output: gate that performs phase flip on base state
                    labeled by num
              # on given number of qubits
          function ret = oracle (num, qubits)
                ret =  eye (2^qubits);
                ret(num+1,num+1) = -1;
                      endfunction

   # input: num – qubits – number of qubits in register
```

```
# output: gate that performs inversion about the mean
                    on given number
                      # of qubits
          function ret = diffuse(qubits)
                tvec = [1:qubits];
              tmp = zeros(1,qubits);
        ret = ProductGate(qubits,H,tvec);
   ret = ret*(2*Projection(tmp) - Id(qubits));
      ret = ret*ProductGate(qubits,H,tvec);
                    endfunction
```

Fig. 4. Source code for Grover iteration with starting state as a parameter

Using first state as a starting point for `grover`[2] function we get probability distribution presented in Fig. 4. In Fig. 5 probability distribution for second starting state is plotted.

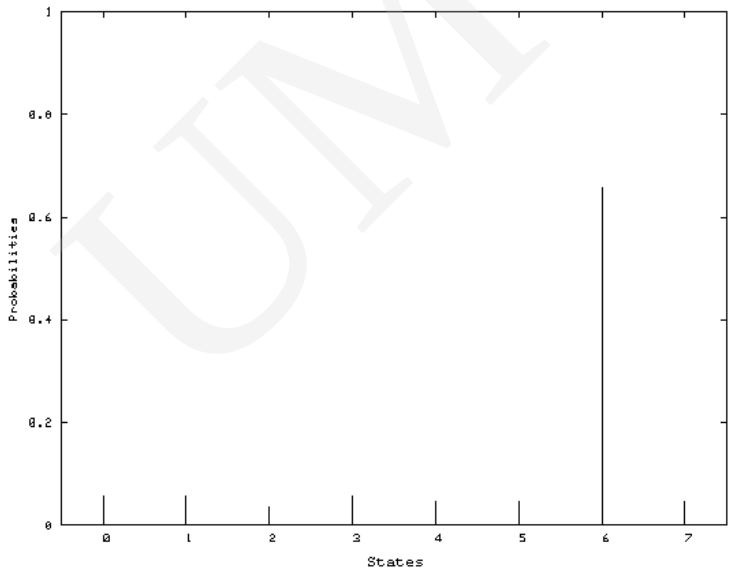

Fig. 5. Probability distribution obtained after Grovers algorithm with starting state `sm1` defined as mixture (2) with x0=0.9 and x1=x2=x3=x4=0.1

We can see that starting from state other than ground state we lose information about result and there is stronger probability that we obtain wrong answer, which means that we measure the state different from predicted. In physical realization this means that we have to repeat experiment to minimize probabilities of error. If the mixture is not pure enough all information will be

[2] See directory `examples` in `quantum-octave` distribution.

lost (see Fig. 5). Strong modification of mixture is reflected by higher probability of error.

This process occurs in physical situations, because obtaining pure ground state is a hard task in some implementations. For example in NMR based quantum computing experiments so called pseudo-pure state is used. It contains only small amount of ground state in statistical contents and problem of observation a result is resolved during the measurement.

4. Conclusions

We have shown that `quantum-octave` package can be useful to simulate, analyze and observe evolution of states during execution of quantum algorithms.

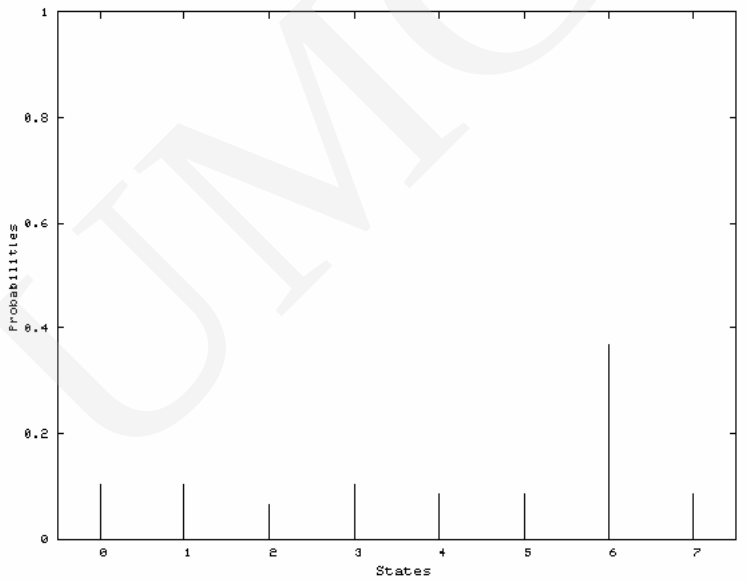

Fig. 6. Probability distribution obtained after Grovers algorithm with starting state `sm2` defined as mixture (2) with x0=0.5 and x1=x2=x3=x4=0.2

It is also significant that observation of errors propagation can be easily realized with quantum-octave. It is possible to introduce errors during execution of algorithms, for example by changing unitary transformations for operations in algorithm or by introducing operations which describe quantum errors.

This work was financially supported by the grant from the State Committee for Scientific Research, Republic of Poland, KBN grant no. 7 T11C 017 21.

References

[1] Project quantum-octave, http://quantum-octave.sourceforge.net/.
[2] GNU Octave home page, http://www.octave.org/.

[3] OctaveForge, http://octave.sourceforge.net/.
[4] Deutsch D., *Quantum theory, the Church-Turing principle and the universal quantum computer*, Proc. Roy. Soc. Lond., A 400 (1985).
[5] Deutsch D., *Quantum computation networks*, Proc. Roy. Soc. Lond., A 425 (1989).
[6] Mosca M., *Quantum computer algorithms*, Ph.D. thesis, University of Oxford, (1999). Grover, Lov K., *Quantum Mechanics Helps in Searching for a Needle in a Haystack*, Phys. Rev. Lett., 79 (1997) 325.
[7] Grover, Lov K., *A framework for fast quantum mechanical algorithms,* in: Proceedings of 30th Annual ACM Symposium on Theory of Computing (STOC), (1998) 53.
[8] Jozsa, R., Searching in Grover's search algorithm, arXiv:quant-ph/9706033.
[9] MathLab home page, http://www.mathworks.com/.
[10] GNU general public license, http://www.gnu.org/.
[11] Dumais P., Touchette H., QuCalc – *Quantum Calculator,* http://crypto.cs.mcgill.ca/QuCalc/.
[12] Phillips F., *Quantum Computation*, The Mathematica Journal, 8(1) (2001).
[13] Mathematica home page, http://www.wolfram.com/.
[14] Preskill, J., *Lecture notes on physics: Quantum computation,* http://www.theory.caltech.edu/people/preskill/ph229/.
[15] Werner R.F., *Quantum states with Einstein-Rosen-Podolsky correlations admitting a hidden-variable model*, Phys. Rev. A, 40 (1989) 4277.
[16] Bruß D., *Characterizing entanglement*, J. Math. Phys, 43 (2002) 4327.
[17] DiVincenzo D.P., Loss D., *Quantum information is physical*, Superlattices and Microstructures 23, 419 (1998), arXiv: cond-mat/9710259.