A Mathematica Simulator of Quantum Circuits

V.P. Gerdt[1], A.N. Prokopenya[2]

[1]e-mail: gerdt@jinr.ru, Laboratory of Information Technologies, JINR, Dubna; [2]e-mail: propenya@brest.byu, Brest State Technical University, Brest, Belarus

In this note we briefly present the first version of our *Mathematica* package `QuantumCircuit` [1] for simulation of quantum circuits [2] and illustrate some of its features by simple examples.

To provide a user with a tool for designing and testing quantum algorithms, a simulator program must be user-friendly, allow to input arbitrary quantum circuit and able to calculate a unitary circuit matrix in the general case of n-qubit memory register.

There are quite a number of different quantum simulators implemented in different classical computer languages. Most of the simulators presented on the Website [3]. Among them there are several programs developed with *Mathematica*. But all they are not universal in a sense that it is not possible to analyze an arbitrary quantum algorithm within the framework of any of them. Our package `QuantumCircuit` satisfies this and other indicated requirements that makes it useful for designing and testing quantum algorithms. At the moment we focus mainly on constructing quantum circuits and computing the corresponding unitary matrices but the package is improved and extended to cover all types of calculations being necessary to simulate a quantum computer.

General structure of any quantum circuit can be readily understood from Fig. 1, where a quantum circuit implementing the Toffoli gate is depicted.

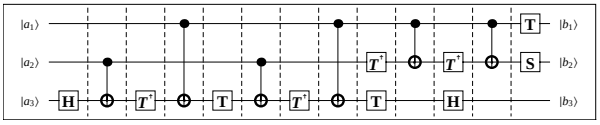

Figure 1: Implementation of the Toffoli gate using Hadamard (H), phase (S), controlled-NOT (CNOT) and $\pi/8$ (T) gates

The circuit is to be read from left-to-right. It means that a column of three qubits $|a_1\rangle$, $|a_2\rangle$, $|a_3\rangle$ in the left-hand side of the diagram determines an initial state of the memory register. Then it is successively acted on by different quantum gates and its final state is shown on the right-hand side of the diagram as a column of qubits $|b_1\rangle$, $|b_2\rangle$, $|b_3\rangle$. We have drawn vertical dashed lines in Fig. 1 to show clearly that evolution of the memory register is controlled by means of successive application of quantum gates to different qubits at each step of computation.

To generate and draw the circuit of Fig. 1 by means of package `QuantumCircuit`, it is sufficient to input the corresponding 3×13 matrix, as shown in Fig. 2, and to invoke the function **circuit[mat]** that depicts the circuit defined by the matrix **mat**.

```
mat = ( 1  1  1  C  1  1  1  C  1  C  1  C  T
        1  C  1  1  1  C  1  1  T' N  T' N  S
        H  N  T' N  T  N  T' N  T  1  H  1  1 ); circuit[mat]
```

Figure 2: A matrix corresponding to the circuit of Fig.1

It should be emphasized that a user can easily add or delete some row or column in the matrix **mat** or change some symbols replacing the corresponding quantum gates. Then the command **circuit[mat]** immediately visualizes a new quantum circuit. Afterwards, one can readily compute a unitary matrix corresponding to the quantum circuit by invoking function **matrixU[mat]**. The built-in data base of the package contains all basic gates [2].

Computational performance of package `QuantumCircuit` is illustrated by Fig. 3 where the timings were obtained on the notebook Toshiba Satellite A305 4 Gb RAM and with Intel Core2 Duo CPU, 2GHz. It should be noted that the unitary

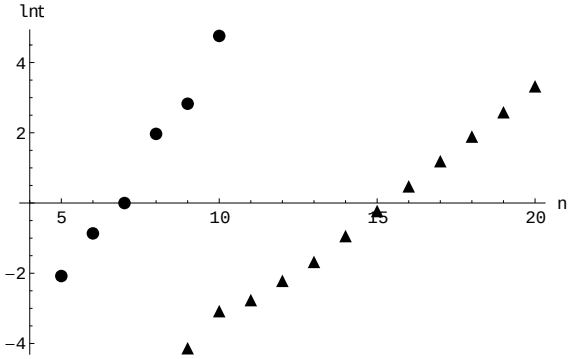

Figure 3: Calculation time of tensor product of n Hadamard (•) and Pauli-X (σ_1) (▲) matrices

matrix for the tensor product of ten Hadamard gates has size $2^{10} \times 2^{10}$ and occupies more than 1 Gb memory. In `QuantumCircuit` such matrices are stored as sparse arrays, and this enables to increase significantly the number of qubits in a circuit to be processed by the package. Thus, for a tensor product of 20 Pauli-X gates one needs only about 25 Mb memory.

To demonstrate application of the package "QuantumCircuit" to simulation of quantum algorithms we consider Grover's algorithm for searching

a marked item in an unstructured database [4]. Let

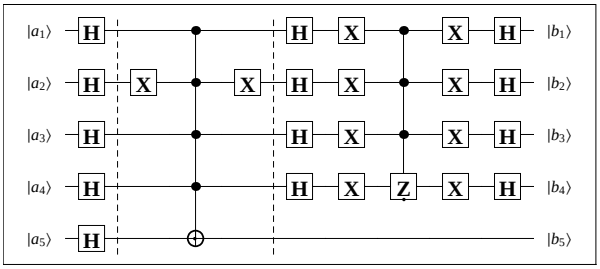

Figure 4: Quantum circuit implementing Grover's search algorithm ($n = 4, k = 11$)

a quantum memory register contain ($n = 4$) data qubits $|a_1\rangle$, $|a_2\rangle$, $|a_3\rangle$, $|a_4\rangle$ are originally prepared in the state $|0\rangle$ and one ancillary qubit $|a_5\rangle$ is in the state $|1\rangle$. It means that the initial state of the memory register is $|00001\rangle$ and the corresponding basis vector in the 32-dimensional Hilbert space has the second component equal to one and all other components equal to zero. Applying five Hadamard gates, one for each qubit, we obtain an equal superposition of all basis states of the five-qubit system. A quantum search subroutine bounded by two dashed lines in the diagram (Fig. 4) is a 4-bit binary function that outputs 1 if its input is some given integer ($k = 11$ in the case shown) and 0 otherwise. This subroutine together with a quantum circuit drawn on the right of the dashed line form one Grover's iteration.

Note that Grover's iteration can be applied to the memory register several times bringing it to some final state that can be measured in the computational basis. And if the number of iterations is equal to the integer part of $\pi/(4\arcsin(2^{-n/2})) = \pi/(4\arcsin(1/4)) = 3.108$ (see [2], [4], then the final state will be exactly $|k\rangle$ with very high probability.

To find a final state of the memory register after several Grover's iterations let us define two matrices **mat0** and **matG** (Fig. 5). The first one represents a column of Hadamard gates bringing initial state $|00001\rangle$ of the memory register to equal superposition of all basis states. The corresponding unitary $2^5 \times 2^5$ matrix **matU0** is given by the function **matrixU[mat0]** . The second matrix **matG** represents one Grover's iteration and the corresponding unitary matrix is given by **matrixU[matG]**.

```
initial = SparseArray[2 → 1, 2^5];
mat0 = {{H}, {H}, {H}, {H}, {H}}; matU0 = matrixU[mat0];

matG = ( 1 C 1 H X C X H
         X C X H X C X H
         1 C 1 H X C X H
         1 C 1 H X Z X H
         1 N 1 1 1 1 1 1 ); matU1 = matrixU[matG];

firstIteration = matU1.matU0.initial;
secondIteration = matU1.firstIteration;
thirdIteration = matU1.secondIteration
```

Figure 5: *Mathematica* code for computing a final vector after three Grover's iterations

Defining the initial state of the memory register as a sparse vector **initial**, one can act on it with operator **matU0** and then apply successively several Grover's iterations **matU1**. As a result we obtain a unit vector with $2^5 = 32$ components which determine probabilities of different basis states of the memory register. Remind that hidden item is encoded by the states of four qubits $|a_1\rangle$, $|a_2\rangle$, $|a_3\rangle$, $|a_4\rangle$, while the ancillary qubit $|a_5\rangle$ is finally in the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and may be found in both basis states $|0\rangle$ and $|1\rangle$ with equal probability. Therefore, a probability P to get k ($k = 0, 1, ..., 15$) as a result of measurement of the memory register $|a_1a_2a_3a_4\rangle$ is equal to a sum of the $2k$th and $(2k + 1)$th components squared of the final vector.

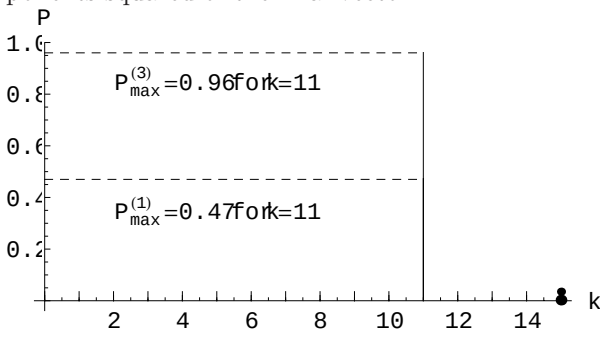

Figure 6: Probability distribution in the final state after one and three Grover's iterations

Fig. 6 shows that after one iteration a probability to get a correct number $k = 11$ as a result of measurement is equal to 47 percent, while after the third Grover's iteration a standard measurement in the computational basis gives 11 with probability 96 percent. It should be noted also that the fourth iteration decreases a probability to get a correct result to 58 percent. Thus, maximum probability to obtain correct result is reached if the number of iterations is equal to its optimal value that is determined as an integer part of $\pi/(4\arcsin(1/\sqrt{N}))$, where $N = 2^n$. For large values of N this number is $O(\sqrt{N})$ and, hence, Grover's algorithm provides a quadratic speed-up in solving the search problem in comparison with a classical computer which requires $O(N)$ applications of the subroutine.

References

[1] Gerdt V.P., Kragler R., Prokopenya A.N.: *A Mathematica Package for Simulation of Quantum Computation*. Lecture Notes in Computer Science 5743, Springer-Verlag, Berlin, 2009, pp.106-117.

[2] Nielsen M., Chuang I.: Quantum Computation and Quantum Information. Cambridge Univ. Press. 2000.

[3] `http://www.quantiki.org/wiki/index.php/List_of_QC_simulators`

[4] Grover L.K.: Quantum mechanics helps in searching for a needle in a haystack. Phys.Rev.Lett. **79** (1997) 325–328.