BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES
Vol. 58, No. 1, 2010

Extending scientific computing system with structural quantum programming capabilities

P. GAWRON*, J. KLAMKA, J.A. MISZCZAK, and R. WINIARCZYK

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

Abstract. We present the basic high-level structures used for developing quantum programming languages. The presented structures are commonly used in many existing quantum programming languages and we use quantum pseudo-code based on QCL quantum programming language to describe them.

We also present the implementation of introduced structures in GNU Octave language for scientific computing. Procedures used in the implementation are available as a package quantum-octave providing library of functions, which facilitates the simulation of quantum computing. This package allows also to incorporate high-level programming concepts into the simulation in GNU Octave and Matlab. As such it connects features unique for higl-level quantum programming languages, with the full palette of efficient computational routines commonly available in modern scientific computing systems.

To present the major features of the described package we provide the implementation of selected quantum algorithms. We also show how quantum errors can be taken into account during the simulation of quantum algorithms using quantum-octave package. This is possible thanks to the ability to operate on density matrices implemented in quantum-octave.

Key words: quantum information, quantum programming, models of quantum computation.

1. Introduction

Quantum information theory main to harness the quantum nature of information carriers in order to develop more efficient algorithms and more secure communication protocols [1–4]. Unfortunately counterintuitive laws of quantum mechanics make the development of new quantum information processing procedures highly non-trivial task. This can be seen as one of the reasons why only few truly quantum algorithms were proposed [5, 6].

As the laws of quantum mechanics are in many cases very different from those we know from the classical world. That is why one needs to seek for the novel methods for describing information processing which involves quantum elements. To this day several formal models were proposed for the description of quantum computation process [7–12].

The most popular of them is the quantum circuit model [7], which is tightly connected to the physical operations implemented in laboratory. It allows to operate with the basic ingredients of quantum information processing – namely qubits, unitary evolution operators and measurements. However, it does not provide too much flexibility concerning operations on more sophisticated elements required to develop scalable algorithms and protocols eg. quantum registers or classical controlling operations.

Another model widely used in the study of theoretical aspects of quantum information processing is the quantum Turing machine [7]. This model is mainly used in the analysis of quantum complexity problems [13]. Its main advantage it that it provides method of comparing efficiency of classical and quantum algorithms. Unfortunately quantum Turing machine, in analogy to its classical counterpart, operates on very basic data and thus it cannot be easily used to construct quantum algorithms.

Both quantum circuit model and quantum Turing machine share some serious drawback concerning lack of support for high-level programming and very limited flexibility. These problems were addressed to be the recent research in the area of quantum programming languages [14–16].

Quantum programming languages are based on the Quantum Random Access Machine (QRAM) model. QRAM is equivalent, with respect to its computational power, to the quantum circuit model or quantum Turing machine. However, it has strictly distinguished two parts: quantum and classical. The quantum part is responsible for performing parts of a algorithm which cannot be computed efficient by a classical machine. The classical part, which is used to control quantum computation. This model is used as a basis for most quantum programming languages [14].

Among high-level programming languages designed for quantum computers we can distinguish imperative and functional languages. The later are seen by many research as the means of providing robust and scalable methods for developing quantum algorithms [17]. However, we focus on the imperative paradigm as it provides more efficient way of implementing high-level quantum programming concepts.

This paper is organized as follows. In Sec. 2 we briefly describe the QRAM model of quantum computer and introduce the quantum pseudocode, which was designed to describe this model. In Sec. 3 we introduce high-level programming structures used in quantum programming languages. These struc-

*e-mail: gawron@iitis.pl

ture are based on the QRAM model of quantum computer. In Sec. 4 the implementation of presented concepts is described and quantum-octave package is presented. In Sec. 5 implementation of quantum algorithms using quantum-octave package is presented. Also the analysis of quantum errors is provided in the case of quantum search algorithm, Finally Sec. 6 summarizes the presented work and provides reader with some concluding remarks.

2. QRAM model of quantum computation

Quantum random access machine is interesting for us since it provides convenient model for developing quantum programming languages. However, these languages and basic concepts used to develop them are our main area of interest. For this reason here we provide only the very basic introduction to the QRAM model. Detailed description of this model is given in [18] and [19] together with the description of hybrid architecture used in quantum programming.

2.1. Classical RAM model. As in many situations in quantum information science, the QRAM models is based on the concepts developed to describe classical computational process – in this case on the Random Access Machine (RAM) model. The classical model of random access machine (RAM) is the example of more general register machines [20–22].

The Random Access Machine consists of an unbounded sequence of memory registers and finite number of arithmetic registers. Each register may hold an arbitrary integer number. The programme for the RAM is a finite sequence of instructions $\Pi = (\pi_1, \ldots, \pi_n)$. At each step of execution register i holds an integer r_i and the machine executes instruction π_κ, where κ is the value of the programme counter. Arithmetic operations are allowed to compute the address of a memory register.

Despite the difference in the construction between Turing machine and RAM, it can be easily shown that Turing machine can simulate any RAM machine with polynomial slowdown only [21]. The main advantage of the RAM models is its resemblance with the real-world computers.

2.2. Quantum RAM model and quantum pseudocode. Quantum Random Access Machine (QRAM) model is build as an extension of the classical RAM model. Its main goal is to provide the ability to exploit quantum resources. Moreover, it can be used to perform any kind of classical computation. The QRAM allows us to control operations performed on quantum registers and provides the set of instructions for defining them. Schematic presentation of such architecture is provided in Fig. 1.

The quantum part of QRAM model is used to generate probability distribution. This is achieved by performing measurement on quantum registers. The obtained probability distribution has to be analysed using classical computer.

Quantum algorithms are, in most of the cases, described using the mixture of quantum gates, mathematical formulas and classical algorithms. The first attempt to provide a uniform method of describing quantum algorithms was made in [23], where the author introduces a high-level notation based on the notation known from computer science textbooks [24, 25].

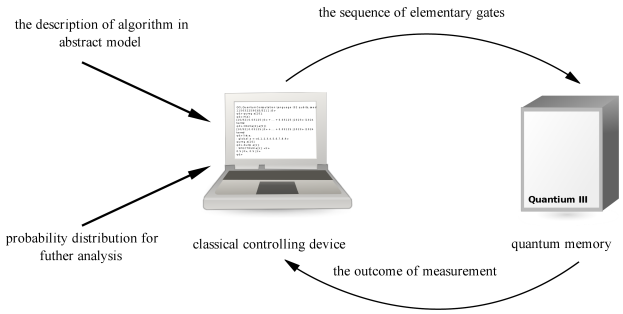

Fig. 1. The model of classically controlled quantum machine [19]. Classical computer is responsible for performing unitary operations on quantum memory. The results of quantum computation are received in the form of measurement results

In [18] Knill introduced the first formalized language for description of quantum algorithms. Moreover, it was tightly connected with the model of Quantum Random Access Machine.

Quantum pseudocode proposed by Knill [18] is based on conventions for classical pseudocode proposed in [24, Chapter 1]. Classical pseudocode was designed to be readable by professional programmers, as well as people who had done a little programming. Quantum pseudocode introduces operations on quantum registers. It also allows to distinguish between classical and quantum registers. In quantum pseudocode quantum registers are distinguished with an underline. They can be introduced by applying quantum operations to classical registers or by calling a subroutine which returns a quantum state. In order to convert a quantum register into a classical register measurement operation has to be performed.

The example of quantum pseudocode is presented in Listing 1. It shows the main advantage of QRAM model over quantum circuits model – the ability to incorporate classical control into the description of quantum algorithm.

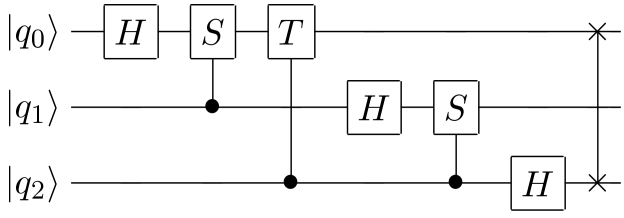

Fig. 2. Quantum circuit representing quantum Fourier transform for three qubits. Elementary gates used in this circuit are described in Ref. 1

Procedure: FOURIER $(\underline{a}, d)$
Input: A quantum register $\underline{a}$ with d qubits. Qubits are numbered form 0 to $d-1$.
Output: The amplitudes of $\underline{a}$ are Fourier transformed over $\mathbb{Z}_{2^d}$.

```
C: assign value to classical variable
ω ← e^{i2π/2^d}
C: perform sequence of gates
for i = d − 1 to i = 0
  for j = d − 1 to j = i + 1
  if a_j then R_{ω^{2^{d−i−1+j}}}(a_i)
    C: number of loops executing phase depends on
    C: the required accuracy of the procedure
  H(a_i)

C: change the order of qubits
for j = 0 to j = d/2 − 1
  SWAP(a_j, a_{d−a−j})
```

Listing 1: Quantum pseudoceode for quantum Fourier transform on d qubits. Quantum circuit for this operation with $d = 3$ is presented in Fig. 2.

Operation $\mathcal{H}(\underline{a_i})$ executes a quantum Hadamard gate on a quantum register $\underline{a_i}$ and $\mathcal{SWAP}(\underline{a_i}, \underline{a_j})$ performs SWAP gate between $\underline{a_i}$ and $\underline{a_j}$. Operation $\mathcal{R}_\phi(\underline{a_i})$ executes a quantum gate $R(\phi)$ is defined as

$$R(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}, \tag{1}$$

on the quantum register $\underline{a_i}$. Using conditional construction

if $\underline{a_j}$ **then** $\mathcal{R}_\phi(\underline{a_i})$

it is easy to define controlled phase shift gate. Similar construction exists in QCL quantum programming language [19]. In Sec. 4 we describe implementation of this construction in quantum-octave.

The measurement of a quantum register can be indicated using an assignment

$a_j \leftarrow \underline{a_j}$.

2.3. Requirements for quantum programming language. Taking into account QRAM model we can formulate basic requirements which have to be fulfilled by any quantum programming language [9, 15, 26].

- **Completeness:** Language must allow to express any quantum circuit and thus enable the programmer to code every valid quantum programme written as a quantum circuit.
- **Extensibility:** Language must include, as its subset, the language implementing some high level classical computing paradigm. This is important since some parts of quantum algorithms (for example Shor's algorithm) require nontrivial classical computation.
- **Separability:** Quantum and classical parts of the language should be separated. This allows to execute any classical computation on purely classical machine without using any quantum resources.
- **Expressivity:** Language has to provide high level elements for facilitating the quantum algorithms coding.
- **Independence:** The language must be independent from any particular physical implementation of a quantum machine. It should be possible to compile a given programme for different architectures without introducing any changes in its source code.

3. High-level programming structures

3.1. Quantum memory. Quantum memory is a set of qubits indexed by integer numbers. Quantum register is a set of indices pointing to distinct qubits. We will denote those registers as $r_1, r_2, \ldots$ or in case of single qubits as $q_1, q_2, \ldots$. The state of quantum memory is a quantum state of size equal to the number of qubits. In case of quantum-octave we operate on density matrices (although some operations on state vectors are allowed). We will denote the state of the quantum memory by ρ.

Following operations on quantum memory are allowed:

- Allocation of new register of size n:
$$\rho_{t+1} = \rho_t \otimes |\underbrace{0\ldots0}_{n}\rangle\langle\underbrace{0\ldots0}_{n}|, \tag{2}$$
where $\otimes$ denotes tensor product, $|\cdot\rangle$ the column vector and $\langle\cdot|$ the dual vector.
- Deallocation of a register indexed by register r:
$$\rho_{t+1} = \mathrm{Tr}_r\left(\rho_t\right), \tag{3}$$
where $\mathrm{Tr}_r\left(\rho\right)$ denotes partial trace of ρ with regard to the subsystem indexed by r.
- Unitary evolution U of the quantum memory:
$$\rho_{t+1} = U\rho_t U^\dagger. \tag{4}$$
- Application of quantum channel K_i on the quantum memory:
$$\rho_{t+1} = \sum_i K_i \rho_t {K_i}^\dagger. \tag{5}$$
- Measurement in computational basis:
$$\rho_{t+1} = \sum_i |i\rangle\langle i|\rho_t|i\rangle\langle i|, \tag{6}$$
$$P(i) = \mathrm{Tr}\left(|i\rangle\langle i|\rho_t\right), \tag{7}$$
where i enumerates the states of computational basis.

For a solid introduction to quantum computation the reader may refer to book by Nielsen and Chuang [1], where all the needed notions are explained in detail.

In quantum computation, construction of the unitary gate is the essential part of quantum algorithm (program) design process. It is a difficult task to write a quantum program using only elementary set of gates *ie.* CNot and one qubit rotations. Therefore it is desirable to introduce some techniques that facilitate the process of composition of complex quantum gates. Some of those techniques are presented below. We will refer to implementation of those techniques in quantum-octave which is described in details in Sec. 4.

3.2. Composed and controlled gates Composed gate. Given one-qubit unitary gate G, quantum register r, and size of the gate s we can construct composed gate U_r^s according to the formula:

$$U_r^s = \bigotimes_{i=1}^{s} X_i, \text{ where } X_i = \begin{cases} G & \text{if} \quad i \in r, \\ \mathbb{I} & \text{if} \quad i \notin r \end{cases}. \tag{8}$$

Controlled gate with multiple controls. Given one-qubit unitary gate G, quantum register r_c we call control, and quantum register r_t we call target, and size of the gate s we can construct controlled gate $U^s_{r_t|r_c}$ according to the formula:

$$\begin{aligned} U^s_{r_t|r_c} &= \bigotimes_{i=1}^{s} X_i + \bigotimes_{i=1}^{s} Y_i, \quad \text{where} \\ X_i &= \begin{cases} |0\rangle\langle 0| & \text{if} \quad i \in r_c, \\ \mathbb{I} & \text{if} \quad i \notin r_c, \end{cases} \\ Y_i &= \begin{cases} G & \text{if} \quad i \in r_t, \\ |1\rangle\langle 1| & \text{if} \quad i \in r_c, \\ \mathbb{I} & \text{if} \quad i \notin r_c \cup r_t \end{cases} \end{aligned} . \tag{9}$$

We assume that $r_t \cap r_c = \varnothing$. Sometimes we will omit the size parameter s.

3.3. Conditionals. One of high-level technique used in quantum programming are quantum conditions [27]. The main idea behind quantum conditions is construction of quantum gates controlled by predicates on control registers.

Condition on quantum variable. The if-then-else structure controlled by a quantum variable and acting on a quantum variable was introduced in QCL [19, 28].

In Fig. 3 the realisation of this concept is presented. If qubit q_0 is in the state $|1\rangle$ the G_1 gate is applied to qubit q_1, otherwise the gate G_2 is applied.

We may write this circuit in the following way:

$$IF_{q_0}(G_1 q_1) ELSE (G_2 q_1) = Not_{q_0} G_{2 q_1|q_0} Not_{q_0} G_{1 q_1|q_0}. \tag{10}$$

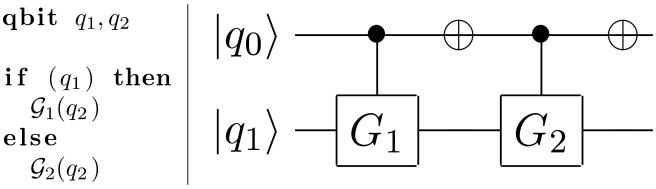

Fig. 3. Example of simple quantum if-then-else structure

For a given control register r_c, target register r_t and two quantum gates G_1 and G_2, we may define define quantum condition in the more general way,

$$IF_{r_c}(G_1 r_t) ELSE (G_2 r_t) = \\ = \prod_{i \in \mathcal{P}(r_t) \setminus \{\varnothing\}} \left(Not_i G_{2 r_t|r_c} Not_i \right) G_{1 r_t|r_c}, \tag{11}$$

where $\mathcal{P}(\cdot)$ denotes the power set.

Condition on mixture of classical and quantum variables. One may consider relation between state of the quantum register and value of the classical variable. In our notation by $[[x]]_r$ we will denote numerical value of ordered in ascending order elements of the register x in regard to register r, for example the value of $[[\{4,9\}]]_{\{2,4,7,9\}}$ is 10. By $[r]$ we will denote the value of the register in order to use it as argument for arithmetic comparison. For example $[r] < 4$ means: "all those values of r that are less than four."

Code and circuit in Fig. 4 show the idea and implementation of conditional operation controlled by expression 'less than' operating on classical constant and quantum register.

Less than

Example in pseudocode	Example in `quantum-octave`

qnibble r
qbit q_1, q_2
 	
if $(r < 4)$ **then**
$\mathcal{G}_1(q_2)$
else
$\mathcal{G}_2(q_1)$
	`r=newregister(4);`
	`q1=newregister(1);`
	`q2=newregister(1);`
	
	`qif(...`
	`qrlt(qureg(q1),4),...`
	`{G1,qureg(q2)}, ...`
	`{G2,qureg(q1)})`

Quantum Circuit

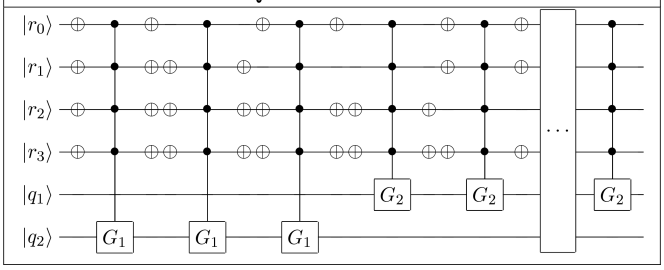

Fig. 4. Example of quantum conditional operation with inequality

In the general case, gate implementing any relation (marked as $\circledast$) can be constructed in the following way:

$$IF_{[r_c]\circledast N}(G_1 r_t) ELSE (G_2 r_t) = \\ = \prod_{i \in \mathcal{F}} \left(Not_i G_{2 r_t|r_c} Not_i \right) \prod_{i \in \mathcal{T}} \left(Not_i G_{1 r_t|r_c} Not_i \right), \tag{12}$$

where sets $\mathcal{T}$ and $\mathcal{F}$ are defined as follows:

$$\mathcal{T} = \mathcal{P}(r_c) \setminus \{x | x \in \mathcal{P}(r_c) \wedge [[x]]_{r_c} \circledast N\}, \tag{13}$$

$$\mathcal{F} = \mathcal{P}(r_c) \setminus \{x | x \in \mathcal{P}(r_c) \wedge \neg([[x]]_{r_c} \circledast N)\}. \tag{14}$$

Note that $\mathcal{T} \cup \mathcal{F} = \mathcal{P}(r_c)$.

In quantum-octave standard arithmetic relations $=$, $\neq$, $<$, $>$, $\leq$, $\geq$ are implemented.

3.4. Expressions. We may consider more complicated expression on quantum registers. In example logical operators and quantum pointers. Logical operators allow to apply an controlled operation to the target register only if a given logical expression on control registers is true. A quantum pointer allows to apply controlled gate on the target register selected by the state of the control register.

Logical expressions on quantum variables. The gate that implements logical expression (denoted here by $\diamond$) is constructed according to the following equation:

$$IF_{[r_{c_1}]\circledast_1 N_1 \diamond [r_{c_2}]\circledast_2 N_2}(G_1 r_t) ELSE (G_2 r_t) = \\ = \prod_{i \in \mathcal{F}} \left(Not_i G_{2 r_t|r_c} Not_i \right) \prod_{i \in \mathcal{T}} \left(Not_i G_{1 r_t|r_c} Not_i \right), \tag{15}$$

where sets $\mathcal{T}$ and $\mathcal{F}$ are defined as follows:

$$\mathcal{T} = \mathcal{P}(r_c) \setminus \{x_1 \cup x_2 | x_1 \subseteq r_{c_1}, x_2 \subseteq r_{c_2} \wedge \\ \wedge \left([[x_1]]_{r_{c_1}} \circledast_1 N_1 \diamond [[x_2]]_{r_{c_2}} \circledast_2 N_2\right)\}, \tag{16}$$

$$\mathcal{F} = \mathcal{P}(r_c) \setminus \{x_1 \cup x_2 | x_1 \subseteq r_{c_1}, x_2 \subseteq r_{c_2} \wedge \\ \wedge \neg \left([[x_1]]_{r_{c_1}} \circledast_1 N_1 \diamond [[x_2]]_{r_{c_2}} \circledast_2 N_2\right)\} \tag{17}$$

and $r_c = r_{c_1} \cup r_{c_2}$.

An example of quantum conditional gate controlled by logical expression defined on quantum registers is presented in Figs. 5 and 6.

And

Example in pseudocode	Example in `quantum-octave`

qbit q_1, q_2, q_3, q_4
 	
if (q_1 **and** q_2) **then**
  $\mathcal{G}_1(q_3)$
else
  $\mathcal{G}_2(q_4)$
	q1=newregister(1);
	q2=newregister(1);
	q3=newregister(1);
	q4=newregister(1);
	
	qif(...
	 qrand(...
	  qreq(qureg(q1),1),...
	  qreq(qureg(q2),1)),..
	 {G1,qureg(q3)}, ...
	 {G2,qureg(q4)})

Quantum Circuit

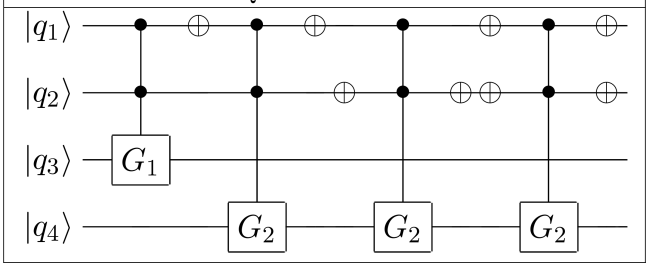

Fig. 5. Example of quantum conditional operation with "and" operator

Or

Example in pseudocode	Example in `quantum-octave`

qbit q_1, q_2, q_3, q_4
 	
if (q_1 **or** q_2) **then**
  $\mathcal{G}_1(q_3)$
else
  $\mathcal{G}_2(q_4)$
	q1=newregister(1);
	q2=newregister(1);
	q3=newregister(1);
	q4=newregister(1);
	
	qif(...
	 qror(...
	  qreq(qureg(q1),1),...
	  qreq(qureg(q2),1)),..
	 {G1,qureg(q3)}, ...
	 {G2,qureg(q4)})

Quantum Circuit

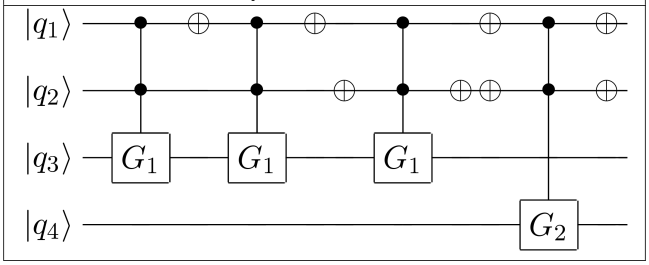

Fig. 6. Example of quantum conditional operation with "or" operator

Quantum pointers. In analogy to concept of pointers and indirect addressing in classical programming, one may introduce quantum pointers. The idea is to use control register to control on which of the target registers an operation should be applied.

Assume one has the n-bit control register and set of 2^n-bit target registers. The control register stores the address of target register to which given unitary operation shall be applied. In order to visualise the use of a quantum pointer an example is shown in Fig. 7.

Pointer

Example in pseudocode	Example in `quantum-octave`

qreg[2] q_1
qnibble q_2
 	
if($*q_1$) **then**
  $\mathcal{G}(q_2)$
	q1=newregister(2);
	q2=newregister(4);
	
	qpointer(G,qureg(q1),qureg(q2))

Quantum Circuit

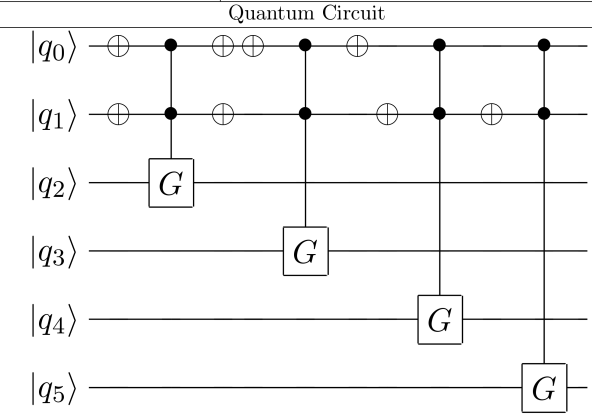

Fig. 7. Example of simple quantum conditional operation controlled by quantum pointer

Formally, quantum pointer controlled by register r_c with target r_t is constructed in the following way:

$$POINT_{r_t}(G_{[r_c]}) = \prod_{i \in \mathcal{P}(r_c)} \left(Not_{r_c \setminus i} G_{[[i]]_{r_c} | r_c} Not_{r_c \setminus i}\right). \tag{18}$$

4. Package quantum-octave

The package quantum-octave [29–31] provides a quantum programming, simulation and analysis language implemented as a library of functions for GNU Octave [32].

GNU Octave is computer algebra system (CAS) and high level programming language designed primarily to perform numerical calculations. The basic data structure in Octave is the matrix (integer, real or complex), therefore it is natural choice for the basis for implementation of quantum programming language.

GNU Octave supports sparse matrices and distributed computing in shared and distributed memory models. GNU Octave is very flexible and easily extendible tool. It is also free software and it can be used in a wide range of operating systems.

4.1. Design choices. The main goal of the design of quantum-octave is to provide a flexible and useful tool for simulation of quantum information processing. Therefore it is based on GNU Octave a high-level scientific programming language. This allows for a seamless integration of very efficient matrix operations and numerical procedures with the library of specialized functions provided by quantum-octave. As GNU Octave is to large extent compatible with Matlab, provided functions can be also used to simulate and analyse quantum algorithms in Matlab.

One of the unique features of quantum-octave is its ability to operate on both pure and mixed quantum states. It allows to perform unitary as well as non-unitary evolution represented by quantum channels.

Quantum gates can be constructed by the user in various ways: by calling provided subroutines, by building their own subroutines, by using quantum control structures. Additionally the user can build and use quantum channels or use those already provided. Most of the quantum-octave functions operate on quantum registers and therefore the quantum operations build with their use are re-allocable.

A good illustration of those features is presented in the following listing 2 that contains the implementation of Quantum Fourier transform in quantum-octave. It can be compared to pseudo code version of the same procedure listed in Listing 1.

```
function ret=dft(gatesize)
  n=gatesize;
  cir=id(n);
  for i=[1:n]
    for j=[1:i-1]
      cir=circuit(cir,cphase(pi/(2^(i-j)),j,i,gatesize));
    endfor
    cir=circuit(cir,productgate(h,i,gatesize));
  endfor
  ret=cir=circuit(cir, flip(n));
endfunction
```

Listing 2: Quantum Fourier transform in quantum-octave

Package quantum-octave can operate on sparse and full matrices depending of users choice. Sparse matrices need much less memory to store but operations on them may be slower. Full matrices tend to consume huge memory space, but operations on them are generally faster. In case of full matrices it should be possible to operate on states of size up to ten qubits on a contemporary workstation. Sparse matrices should allow to simulate the quantum systems of up to 20 qubits.

Although quantum-octave is not, strictly speaking, a programming language ready to program real quantum devices, with some effort it can be transformed in such a way that it would be able to compile high level programs to some sort of quantum assembler. One should note that broad range of functions allowing the analysis of states is implemented in the package. Those function are described in the following section.

4.2. Description. quantum-octave is designed to allow the user to operate on different levels of abstraction. User can prepare complex gates and quantum channels from basic primitives such as single qubit rotations, controlled gates, single qubit channels. Most of the functions that form this library operate on quantum registers, which makes the preparation of quantum gates and channels very "natural". The library is implemented in such a way that depending of user's choice it may operate on full or sparse matrices.

quantum-octave can work in two modes: as a library or as programming language/simulator. Library mode is default. To move to language/simulator mode one has call `quantum_octave_init()` function. In case of the second mode quantum-octave allocates and manages an internal quantum state and maintains the quantum registers. Such functions as `evolve()`, `applychannel()`, `measurecompbasis()` operate directly on the internal state. Listings of Deustch's 3 and Grover's 4 algorithms show use of language/simulation mode.

Convention. Following conventions are used in quantum-octave.

Quantum register is horizontal integer vector containing indices of qubits starting from one.
Ket is vertical complex vector.
Bra is horizontal complex vector.
Density matrix is complex square matrix always of dimension n-th power of two by n-th power of two.
Binary string is 0,1-horizontal vector, that encodes a binary number. Order of bits is from MSB to LSB.
Size of the gate or channel is always given in terms of number of qubits it acts on. If size is written in square brackets it means that it can be omitted if the gate or channel acts on the whole system and quantum-octave was initialised.

Quantum gates.

Package quantum-octave supplies set of elementary gates known in quantum computation.

- `sx`, `sy`, `sz` – return one-qubit Pauli operators `sx` – σ_x, `sy` – σ_y, `sz` – σ_z.
- `id(n)` – returns identity matrix: $\mathbb{I}_n$.
- `roty(a)`, `rotz(a)`, `rotx(a)` – return rotation matrix by angle `a` around appropriate axis.
- `qft(n)` – returns quantum Fourier transform on `n` qubits.
- `swap(size, qubits)` – returns swap gate of a given `size` that swaps `qubits` given as two-element vector.
- `qubitpermutation(permutation)` – returns unitary gate that performs given `permutation`.
- `h` – returns one-qubit Hadamard gate.
- `phase(p0,p1)` – returns one-qubit phase gate, with $p0, p1$ phase parameters.

Basic functions. Following functions are essential to prepare a quantum state and to implement a quantum algorithm, protocol or game.

- `ket(binvec)` – returns ket for given binary string.
- `ketn(int,size)` – returns a ket of size 2^{size} for given `integer` number.

– `state(pure_state)` – returns density matrix for a given ket.
– `mixstates(a1,mixed_state1,[a2,mixed_state2,...])` – returns convex combination of density matrices with coefficients `a1, a2, ....`
– `productgate(gate,targetreg[,size])` – returns a controlled gate of a given `size` that applies given `gate` on `target register`. See Eq. 8.
– `controlledgate(gate,controlreg,targetreg [,size])` – returns a controlled gate of given `size` that applies `gate` on specified `target register` and is controlled by `control register`. See Eq. 9.

Quantum conditional operations. The functions listed below implement quantum conditional operations, quantum expressions and pointers. They are useful to simplify the implementation.

– `qif(expression,ifpart,elsepart,size)` – returns quantum gate of given `size`, controlled by `expression` that applies `ifpart` if expression is true and `elsepart` if expression is false. `ifpart` and `elsepart` are cellarrays in the form: {`gate, target_register`}. See Eq. (11).
– `qreq(register,integer)` – returns expression: [`register`] equals `integer`. See Eq. (12).
– `qrne(register,integer)` – returns expression: [`register`] not equal `integer`.
– `qrge(register,integer)` – returns expression: [`register`] is greater or equal to `integer`.
– `qrgt(register,integer)` – returns expression: [`register`] is greater than `integer`.
– `qrle(register,integer)` – returns expression: [`register`] is lesser or equal to `integer`.
– `qrlt(register,integer)` – returns expression: [`register`] is lesser than `integer`.
– `qrin(register,set)` – returns expression: [`register`] is in `set`.
– `qror(expr1,expr2)` – returns logical or on expressions `expr1` and `expr2`. See Eq. (15).
– `qrand(expr1,expr2)` – returns logical and on expressions `expr1` and `expr2`.
– `qpointer(gate,contrregister,targteregister [,size])` – returns quantum gate of given `size`, controlled by `controll register` that applies `gate` on `target register`. See Eq. (18).

Evolution, channels and measurement. The following group of functions allows to control the evolution of quantum states, and introduces the application of channels and measurement.

– `evolve(evolution[,state])` – applies unitary `evolution` to the `state`, returns the result of the evolution. See Eq. (4).
– `channel(name,p)` – returns Kraus operators acting on one qubit, parametrised by `p` allowed `names` are: `"depolarizing"`, `"amplitudedamping"`, `"phasedamping"`, `"bitflip"`, `"phaseflip"` and `"bitphaseflip"`.
– `localchannel(kraus, targetreg[, chsize])` – returns a channel being the extension of defined by `kraus` operators channel, acting on `target register`.
– `applychannel(elements[,state])` – applies on the `state` non unitary evolution defined by set of Kraus operators (`elements`), returns the result of the evolution. See Eq. (5).
– `ptrace(state, targetreg)` – returns reduced density matrix for the `state` with `target register` traced out. See Eq. (3).
– `circuit(gate[, gate])` – returns circuit composed of the sequence of `gates`.
– `measurecompbasis([state])` – returns the probability distribution of the σ_z measurement on the given `state`.
– `isunitary(gate)` – returns `true` if the `gate` is unitary, otherwise returns `false`.
– `ischannel(operators)` – returns `true` if the `operators` form valid quantum channel, otherwise returns `false`.
– `collapse(distribution)` – chooses and returns a basis state at random according to `distribution`.

Computation and control. Following functions allows to control the quantum heap and configure the behaviour of the library.

– `quantum_octave_init()` – initialises the simulated system, creates quantum state with zero qubits allocated and empty list of registers.
– `set_quantum_octave_sparse([true | false])` – switches on or off use of sparse matrices by all quantum-octave functions.
– `newregister(size)` – creates new register of given `size`, allocates qubits on quantum heap, returns register id.
– `clearregister(regid)` – removes `regid` register from quantum heap. Traces out appropriate qubits from the internal state.
– `qureg(regid)` – returns quantum register to which `regid` points.
– `getstate()` – returns the internal quantum state.

Well known states. Some of the states commonly used in quantum algorithms are implemented in the library as separate functions.

– `ghz(n)` – returns Greenberger-Horne-Zeilinger state for `n` qubits: $\frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle)^{\otimes n}$.
– `phip` – returns Bell $|\Phi^+\rangle$ state: $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.
– `phim` – returns Bell $|\Phi^-\rangle$ state: $\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$.
– `psip` – returns Bell $|\Psi^+\rangle$ state: $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$.
– `psim` – returns Bell $|\Psi^-\rangle$ state: $\frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$.
– `maximallymixed(n)` – return density matrix maximally mixed state: $\frac{1}{n}\mathbb{I}_n$.

– `wernersinglet(a)` – returns 2-qubit Werner state: $a(|00\rangle - |11\rangle)(\langle 00| - \langle 11|) + (1-a)\frac{\mathbb{I}}{4}$.

Analysis. Package quantum-octave provides standard functions for analysis of quantum states, widely used in quantum information literature. Among them the most important are:

– `negativity(state, qubits)` – computes negativity of the `state` in respect to `qubits`.
– `entropy(state)` – computes Von Neuman entropy of the `state`.
– `concurrence(state)` – computes concurrence of the `state`.
– `fidelity(rho, sigma)` – computes fidelity between density matrices `rho` and `sigma`.
– `fidelitypuremixed(psi, rho)` – computes fidelity between ket `psi` and density matrix `sigma`.
– `tracenorm(state)` – computes trace norm of the `state`.
– `partialtranspose(state, targetreg)` – returns matrix being partial transposition of `state` matrix in regard to `target` register.

The next section presents the applications of quantum-octave and various programming techniques for solutions of quantum programming problems.

5. Examples and applications

In what follows the applications of quantum-octave and various high-level programming techniques are discussed. It is shown how quantum processes, such as algorithms may be implemented, simulated and analysed with this tool.

5.1. Deutsch's problem. One of the simplest quantum algorithms is Deutsch's algorithm. Although it may seem trivial, this algorithm shows two very important features of quantum computation. First, by taking advantage of superposition one can compute any binary function for all its arguments in one step and second, that it is only possible to retrieve an information about property of a function and not on its values.

Let's assume that we have a black box that is usually called the oracle. This box computes a function $f : \{0,1\} \to \{0,1\}$. We do not know if that function is constant $f(0) = f(1)$ or injective $f(0) = \overline{f(1)}$. In classical case we have to ask the oracle twice to check which kind the function f is. But in quantum case it is possible to solve this problem asking the oracle only once.

The algorithm goes as follows:

1. Prepare the state: $|\Psi\rangle = |0\rangle \otimes |1\rangle$,.
2. Apply the Hadamard $H^{\otimes 2}$ gate on the state $|\Psi\rangle$, you will get
$$|\Psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{19}$$
3. Apply the gate $U_f : |x\rangle \otimes |y\rangle \to |x\rangle \otimes |f(x) \oplus y\rangle$ on the state $|\Psi_1\rangle$; you will get:
$$|\Psi_2\rangle = \begin{cases} \pm\frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \text{for constant } f, \\ \pm\frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \text{for injective } f. \end{cases} \tag{20}$$
4. Apply $H \otimes \mathbb{I}$ on the state $|\Psi_2\rangle$; you will get:
$$|\Psi_3\rangle = \begin{cases} \pm|0\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \text{for constant } f, \\ \pm|1\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \text{for injective } f. \end{cases} \tag{21}$$
5. Measure state of the first qubit, you will get $|0\rangle$ in case of constant function, $|1\rangle$ for injective function.

Quantum circuit representation of Deutsch's algorithm is presented in Fig. 8. The U_f gate provide a reversible implementation of function f and the symbol  denotes a measurement.

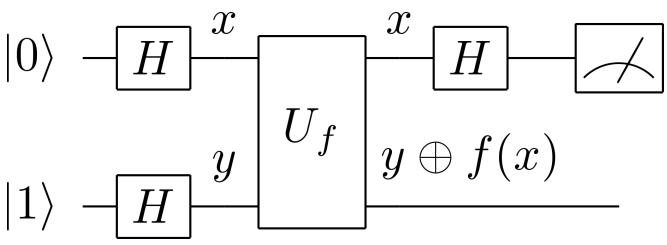

Fig. 8. Deutsch's algorithm

The implementation of Deutsch's algorithm presented in listing 3 is an introductory example of application of quantum-octave for simulation of a quantum algorithm with all basic steps of computation: initialization of the quantum computer, unitary evolution and measurement.

Below we have the description of simulation steps (compare with circuit in Fig. 8):

line 5 : initialisation of the simulator,
lines 7, 8 : allocation of registers,
lines 10 to 17 : definition of all four possible oracles,
line 19 : application of Not on second qubit,
line 20 : application of $H \otimes H$,
line 21 : application of the oracle,
line 22 : application of $H \otimes \mathbb{I}$,
line 24 : tracing out of second register,
line 26 : return the probability distribution of the measurement outcome.

```
# input: identifier of the function
# output: state after execution of Deutsch's algorithm
function ret = deutsch(num)
 # initialize the simulation
 quantum_octave_init();
 # declare and allocate registers
 r1=newregister(1);
 r2=newregister(1);
 # declare functions
 f{1}=id(2);
 f{2}=productgate(sx,qureg(r2));
 f{3}=qif(qreq(qureg(r1),1),...
   {sx,qureg(r2)},...
   {id,qureg(r2)});
 f{4}=qif(qreq(qureg(r1),0),...
   {sx,qureg(r2)},...
   {id,qureg(r2)});
 # do the algorithm
 evolve(productgate(sx,qureg(r2)));
 evolve(productgate(h,[qureg(r1),qureg(r2)]));
 evolve(f{num});
 evolve(productgate(h,qureg(r1)));
 # throw away second register
  clearregister(qureg(r2));
 # return the outcome
 ret=measurecompbasis();
endfunction
```

Listening 3: Deutsch algorithm in quantum-octave

5.2. Grover's algorithm. To illustrate more advanced usage of the presented concepts we use the quantum algorithm for searching a unordered database. The algorithms was proposed by Grover [33–35] and its detailed description and analysis can be found in [36, 37]. Here we present implementation of Grover's algorithm which presents the features of quantum-octave related to the observation of quantum errors. We show the propagation of initial errors during the execution of the algorithm.

Grover's search algorithm is one of the most important quantum algorithms. This especially true since many algorithmic problems can be reduced to exhaustive search. However, like in the case of any quantum procedure, the efficiency of the algorithm depends on the ability to avoid errors during the procedure. Thus, it is important how quantum errors affect the executions of the algorithm.

Statement of the problem. Let X be a set and let $f : X \to \{0, 1\}$, such that

$$f(x) = \begin{cases} 1 \Leftrightarrow x = x_0 \\ 0 \Leftrightarrow x \neq x_0 \end{cases}, \ x \in X, \tag{22}$$

for some marked $x_0 \in X$.

For the simplicity we assume that X is a set of binary strings of length n. Therefore

$$|X| = 2^n$$

and

$$f : \{0, 1\}^n \to \{0, 1\}.$$

We can map the set X to the set of states over $\mathcal{H}^{\otimes n}$ in the natural way as

$$x \leftrightarrow |x\rangle. \tag{23}$$

The goal of the algorithm is to find the marked element. This is achieved by the amplification of the apropriate amplitude [36, 37].

The algorithm. The Grover's algorithm is composed of two main procedures: the oracle and the diffusion.

Oracle. By oracle we understand a function that marks one defined element. In the case of this algorithm, the marking of the element is done by negation of the amplitude of the state that we search for.

With the use of elementary quantum gates the oracle can be constructed using ancilla $|q\rangle$ in the following way:

$$O|x\rangle|q\rangle = |x\rangle|q \otimes f(x)\rangle. \tag{24}$$

If the register $|q\rangle$ is prepared in the state:

$$|q\rangle = H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}, \tag{25}$$

then by substitution, Eq. (24) is re-transformed to:

$$O|x\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}} = (-1)^{f(x)}|x\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}}, \tag{26}$$

and by tracing out the ancilla we get:

$$O|x\rangle = -(-1)^{f(x)}|x\rangle. \tag{27}$$

Thus the oracle marks a given state by inverting its amplitude.

Diffusion. The operator D rotates any state around the state

$$|\psi\rangle = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle, \tag{28}$$

D may written in the following form:

$$D = -H^{\otimes n}(2|0\rangle\langle 0| - \mathbb{I})H^{\otimes n} = 2|\psi\rangle\langle\psi| - \mathbb{I}. \tag{29}$$

Grover iteration. The first step of the algorithm is to apply Hadamard gate $H^{\otimes n}$ on all the qubits. Then we apply gate $G = DO$ several times.

Number of iterations. Application of diffusion operator on the base state $|n\rangle$ gives

$$-H^{\otimes n}I_0H^{\otimes n}|n\rangle = -|x_0\rangle + \frac{2}{N}\sum_y |y\rangle. \tag{30}$$

Application of this operator on any state gives

$$\begin{aligned} D|x\rangle &= \sum_x \alpha_x(-|x\rangle + \frac{2}{N}y\sum_y |y\rangle) \\ &= \sum_x(-\alpha_x + 2s)|x\rangle, \end{aligned}$$

where

$$s = \frac{1}{N}\sum_{\mathbf{x}}\alpha_{\mathbf{x}} \tag{31}$$

is arithmetic mean of coefficients $\alpha_x,\ x = 0, \ldots, 2^n - 1$. k-fold application of Grover's iteration G on initial state $|s\rangle$ leads to [36]:

$$G^k|s\rangle = \alpha_k \sum_{x \neq x_0} |x\rangle + \beta_k |x_0\rangle, \tag{32}$$

with real coefficients:

$$\begin{aligned} \alpha_k &= \frac{1}{\sqrt{N-1}} \cos{(2k+1)\,\theta}, \\ \beta_k &= \sin{(2k+1)\,\theta}, \end{aligned} \tag{33}$$

where θ is an angle that fulfils the relation:

$$\sin(\theta) = \frac{1}{\sqrt{N}}. \tag{34}$$

Therefore the coefficients α_k, β_k are periodic functions of k. After several iteration amplitude of β_k rises and the others drop. The influence of the marked state $|x_0\rangle$ on the state of the register is that initial state $|s\rangle$ evolves towards the marked state.

The β_k attains its maximum after approximately $\frac{\pi}{4}\sqrt{N}$ steps. Then it begins to fall.

The number of steps needed to transfer the initial state towards the marked state is of $O(\sqrt{n})$. In the classical case the number of steps is of $O(n)$.

Measurement. The last step of the Grover's algorithm is the measurement. Probability of obtaining of the proper result is $|\beta_k|^2$.

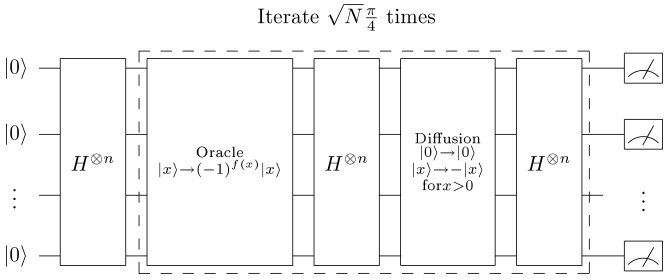

Fig. 9. The circuit for Grover's algorithm

Graphical interpretation. There exists a very nice graphical interpretation of Grover's algorithm.

Let $|\alpha\rangle$ denotes the sum of states orthogonal to the state we are searching for $|x_0\rangle$

$$|\alpha\rangle = \frac{1}{\sqrt{2^n - 1}} \sum_{x \neq x_0} |x\rangle, \tag{35}$$

and for consistence we will write $|\beta\rangle = |x_0\rangle$. Then, on the plane spanned by $|\alpha\rangle$ and $|\beta\rangle$, we can observe of evolution of the state vector.

By putting values from Eq. (33) into Eq. (32) we get following relation:

$$G^k|s\rangle = \cos((2k+1)\theta)|\alpha\rangle + \sin((2k+1)\theta)|\beta\rangle. \tag{36}$$

Exemplar behavior of this equation for 2^3 states is presented in Fig. 10.

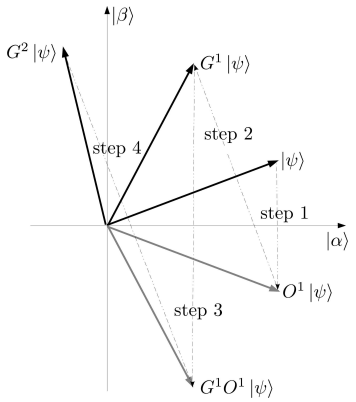

Fig. 10. Visualisation of Grover's algorithm after Ref. 1. Projection on plane spanned by $|\alpha\rangle$ and $|\beta\rangle$. Vector $|\psi\rangle$ is flat superposition of all the possible states

Implementation. Listing 4 presents the implementation of function `grover`. We will apply quantum noise at the end of each Grover iteration and observe its influence on its efficiency.

To simulate this behavior we will insert the code from Listing 5 after line 21 of the implementation.

```
1  # function implementing Grover's algorithm
   # input: number we are looking for, size of the system
3  # output: probability distribution after
   #      execution of the algorithm
5  function ret = grover(num, S)
    # initialize the simulation
7   quantum_octave_init();
    # allocate register
9   r1=newregister(S);
   # number of elements
11 N=2^length(qureg(r1))
   # calculate number of iterations
13 k = floor((pi/4)*sqrt(N));
   # prepare the system in flat superposition of base states
15 evolve(productgate(h,qureg(r1)));
   # Grover iterations
17 for i = 1:k
    # ask the oracle
19  evolve(oracle(num, qureg(r1)));
    # diffuse
21  evolve(diffuse(qureg(r1)));
   endfor
23 # return probability distribution of base states
    ret=measurecompbasis();
25 endfunction


27 # function implementing oracle
   # input: number to mark, size of the system
29 # output: gate implementing oracle of the size 2^l
   function ret = oracle (num, register)
31  l=length(register);
    ret = id(l);
33  ret(num+1,num+1) = -1;
   endfunction
35
   # function implementing oracle
37 # input: register on which implement diffusion
   # output: gate implementing diffusion of the size 2^l
39 function ret = diffuse(register)
    l=length(register);
41  ret = circuit(...
        productgate(h,register,l),...
43      (2*ketn(0,l)*bran(0,l) - id(l)),...
        productgate(h,register,l)...
45     );
   endfunction
```

Listening 4: Grover's algorithm in quantum-octave

```
applychannel(
    localchannel(
        channel(channelname,p),qureg(r1)
    )
);
```

Listening 5: Adding noise to Grover's algorithm

Simulation results. The results of the simulation of noisy Grover's algorithm acting on system of size from three to six qubits when system is affected by noise modelled with depolarizing channel are shown in Fig. 11. One may observe that rate of successful application of the algorithm drops quickly with raising amount of noise. This effect is more significant for larger systems. This result clearly indicates that it is not possible to successfully implement Grover's algorithm in presence of large amounts of noise if no error correction scheme is applied.

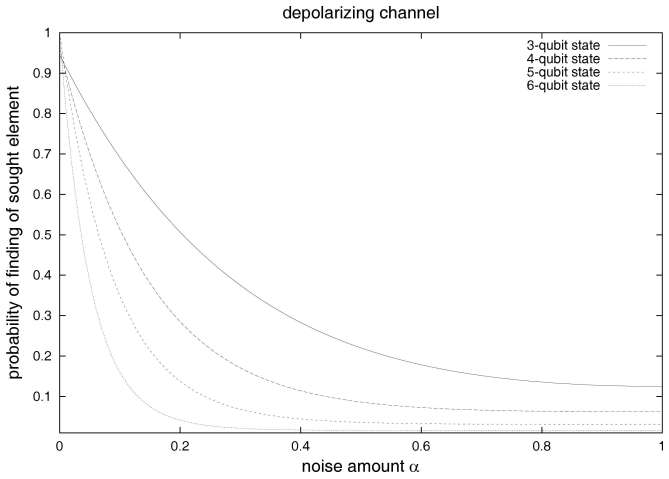

Fig. 11. Influence of depolarizing channel parametrized by single real number α on probability of successful finding of sought element in Grover's algorithm implemented with 3, 4, 5 and 6 qubits

6. Summary

We have introduced an original solution to the problem of simulation of quantum processes. This solution is provided by quantum-octave a library that is build upon GNU Octave high level programming language, which provides high-level quantum programming structures.

Although, strictly speaking, quantum-octave is not a programming language but a library, together with GNU Octave, it is very convenient and flexible tool. Programs written in quantum programming languages, such as QCL, can be easily rewritten using this library, thanks to the use of quantum memory, registers and routines. Scalable programs can be easily implemented in quantum-octave so the programmer does not have to think about details of the implementation.

Acknowledgements. We acknowledge the financial support by the Polish Ministry of Science and Higher Education under the grant number N519 012 31/1957 and by the Polish research network LFPPI. The numerical calculations presented in this work were performed on the `Leming` server of The Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences.

Package quantum-octave is distributed as free software and it can be downloaded from the project web-page [38].

REFERENCES

[1] M.A. Nielsen and I.L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, 2000.

[2] M. Hirvensalo, *Quantum Computing*, Springer, Berlin, 2001.

[3] S. Bugajski, J. Klamka, and S. Wegrzyn, "Foudations of quantum computing, Part I", *Archives of Theoretical and Applied Informatics* 13 (1), 97–142 (2001), (in Polish).

[4] S. Bugajski, J. Klamka, and S. Wegrzyn, "Foudations of quantum computing. Part II", *Archives of Theoretical and Applied Informatics* 13 (1), 137–149 (2001), (in Polish).

[5] P.W. Shor, "Why haven't more quantum algorithms been found?", *ACM* 50 (1), 87–90 (2003).

[6] P.W. Shor, "Progress in quantum algorithms", *Quantum Information Processing* 3, 1–5 (2004).

[7] D. Deutsch, "Quantum theory, the Church-Turing principle and the universal quantum computer", *Proc. Roy. Soc. Lond.* A 400, 97 (1985).

[8] D. Deutsch, "Quantum computational networks", *Proc. Roy. Soc. Lond.* A 425, 73 (1989).

[9] S. Bettelli, L. Serafini, and T. Calarco, "Toward an architecture for quantum programming", *Eur. Phys. J. D* 25 (2), 181–200 (2003).

[10] S. Gudder, "Quantum computational logic", *Int. J. Theoretical Physics* 1 (42), 39–47 (2003).

[11] A. van Tonder, "A lambda calculus for quantum computation", *SIAM J.COMPUT.* 33, 1109 (2004).

[12] C. Moore and J.P. Crutchfield, "Quantum automata and quantum grammars", *Theoretical Computer Science* 237 (1–2), 275–306 (2000).

[13] E. Bernstein and U. Vazirani, "Quantum complexity theory", *SIAM J. on Computing* 26 (5), 1411–1473 (1997).

[14] S. Gay, "Quantum programming languages: Survey and bibliography", *Bull. Eur. Association for Theoretical Computer Science* 1, CD-ROM (2005).

[15] J.A. Miszczak, *Probabilistic Aspects of Quantum Programming Languages*, PhD Thesis, The Institute of Theoretical and Applied Informatics PAS, Warsaw, 2008.

[16] S. Gay, *Bibliography on Quantum Programming Languages*, web-page http://www.dcs.gla.ac.uk/~simon/quantum/, 2007.

[17] T. Altenkirch and J. Grattage, "A functional quantum programming language", *Proc. Annual IEEE Symposium on Logic in Computer Science* 1, 249–258 (2005).

[18] E. Knill, "Conventions for quantum pseudocode", *Technical Report LAUR-96-2724* 1, CD-ROM (1996).

[19] B. Oeme, *Structured Quantum Programming*, PhD Thesis, Technical University of Vienna, Vienna, 2003.

[20] S.A. Cook and R.A. Reckhow, "Time-bounded random access machines", *Proc. forth Annual ACM Symposium on Theory of Computing* 1, 73–80 (1973).

[21] C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley Publishing Company, New York, 1994.

[22] J.C. Shepherdson and H.E. Strugis, "Computability of recursive functions", *J. ACM* 10 (2), 217–255 1963.

[23] R. Cleve and D.P. DiVincenzo, "Schumacher's quantum data compression as a quantum computation", *Phys. Rev.* A 54 (4), 2636–2650 (1996).

[24] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, London, 2001.

[25] J.E. Hopcroft and J.D. Ullman, *Introduction to the Theory of Automata, Language*, PWN Scientific Publishing House, Warsaw, 2003, (in Polish).

[26] S. Bettelli, *Toward an Architecture for Quantum Programming*, PhD Thesis, Università di Trento, Trento, 2002.

[27] P. Gawron, *High Level Programming in Quantum Computer Science*, PhD Thesis, The Institute of Theoretical and Applied Informatics PAS, Warsaw, 2008.

[28] B. Oemer, *Quantum Programming in QCL*, Master Thesis, TU Viena, Vienna, 2000.

[29] P. Gawron and J.A. Miszczak, "Didactic tools for teaching quantum informatics", *Annales UMCS Informatica* AI 1 (2), 77–79 (2004).

[30] P. Gawron and J.A. Miszczak, "Simulations of quantum systems evolution with quantum-octave package", *Annales UMCS Informatica* AI 1 (2), 52–63 (2004).

[31] P. Gawron and J.A. Miszczak, "Numerical simulations of mixed states quantum computation", *Int. J. Quan. Inf.* 3 (1), 195–199 (2005).

[32] J.W. Eaton, *GNU Octave Manual*, Network Theory Limited, New York, 2002.

[33] L. Grover, "A fast quantum mechanical algorithm for database search", *Proc. 28th Annual ACM Symposium on the Theory of Computation* 1, 212–219 (1996).

[34] L.K. Grover, "Quantum mechanics helps in searching for a needle in a haystack", *Phys. Rev. Lett.* 79, 325 (1997).

[35] L.K. Grover, "A framework for fast quantum mechanical algorithms", *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)* 1, 53–62 (1998).

[36] S. Bugajski, "Quantum search", *Archives of Theoretical and Applied Informatics* 13 (2), 143–150 (2001).

[37] S.J. Lomonaco, "Grover's quantum search algorithm", *Proc. Symposia in Applied Mathematics* 58, 181–192 (2002).

[38] Project quantum-octave, http://quantum-octave.sf.net/, 2007.