УДК 004.27
DOI: 10.25559/SITITO.18.202202.455-467

Original article

Emulators of Quantum Computers on Qubits and on Qudits

A. S. Andreev, P. V. Khrapov*
Bauman Moscow State Technical University, Moscow, Russian Federation
Address: 5, 2-nd Baumanskaya St., building 2, Moscow 105005, Russian Federation
* khrapov@bmstu.ru

Abstract

Quantum computing is still a developing, but an extremely promising area. The article lays out the main ideas behind quantum computing in simple terms. The topic of quantum computers based on qudits – multidimensional analogues of qubits, which have recently received much attention due to their efficiency, is also covered. The fundamentals of quantum mechanics, which are necessary for understanding the principles of operation of a quantum computer, such concepts as qubits and qudits, linear operators, the measurement process, etc are introduced. As an example of quantum computing, the principle of operation of the Deutsch-Jozsa algorithm, one of the first quantum algorithms to demonstrate their advantages, and its generalization to qudits, are analyzed in detail. The process of writing the simplest quantum computer emulator in the Python programming language is described step by step. The emulator operates with an arbitrary number of qubits and allows you to apply arbitrary operators to them and carry out multiple measurements of the final state of the qubit. A generalization of this emulator for working with qudits is given after that. To demonstrate the emulator we have written, we present programs that implement the Deutsch-Jozsa algorithm and its generalizations on it, and test them.

Keywords: quantum computer, qubit, qudit, quantum emulator, Deutsch-Jozsa algorithm

The authors declare no conflict of interest.

For citation: Andreev A.S., Khrapov P.V. Emulators of Quantum Computers on Qubits and on Qudits. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education.* 2022; 18(2):455-467. doi: https://doi.org/10.25559/SITITO.18.202202.455-467

Научная статья

Эмуляторы квантовых компьютеров на кубитах и на кудитах

А. С. Андреев, П. В. Храпов*
ФГБОУ ВО «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)», г. Москва, Российская Федерация
Адрес: 105005, Российская Федерация, г. Москва, ул. 2-я Бауманская, д. 5, к. 1
* khrapov@bmstu.ru

Аннотация
Квантовые вычисления – это развивающаяся и чрезвычайно перспективная область. В статье простыми словами изложены основные идеи, лежащие в основе квантовых вычислений. Также освещается тема квантовых компьютеров на основе кубитов – многомерных аналогов кубитов, которым в последнее время уделяется большое внимание благодаря их эффективности. Вводятся основы квантовой механики, которые необходимы для понимания принципов работы квантового компьютера, такие понятия, как кубиты и кудиты, линейные операторы, процесс измерения и т.д. В качестве примера квантовых вычислений подробно анализируются принцип работы алгоритма Дойча-Йожа, одного из первых квантовых алгоритмов, продемонстрировавшего свои преимущества, и его обобщение на кудиты. Пошагово описан процесс написания простейшего эмулятора квантового компьютера на языке программирования Python. Эмулятор работает с произвольным количеством кубитов и позволяет применять к ним произвольные операторы и проводить множественные измерения конечного состояния кубита. После этого дается обобщение этого эмулятора для работы с кудитами. Чтобы продемонстрировать написанный нами эмулятор, мы представляем программы, реализующие алгоритм Дойча-Йожа и его обобщения на нем, и тестируем их.

Ключевые слова: квантовый компьютер, кубит, кудит, квантовый эмулятор, алгоритм Дойча-Йожа

Авторы заявляют об отсутствии конфликта интересов.

Для цитирования: Андреев А. С., Храпов П. В. Эмуляторы квантовых компьютеров на кубитах и на кудитах // Современные информационные технологии и ИТ-образование. 2022. Т. 18, № 2. С. 455-467. doi: https://doi.org/10.25559/SITITO.18.202202.455-467

Introduction

Quantum computing is still a developing, but an extremely promising area.

Computations on quantum computers are carried out by applying operations to qubits, the quantum counterparts of classical bits. The main feature of qubits is stated in the principle of superposition: a qubit can be in several states at the same time. This allows using one operation to perform calculations for all states of the system at once, in contrast to a classical computer, in which calculations would have to be performed sequentially or in parallel. However, after this operation is executed, the qubit will also be in a state of superposition, and when we try to measure the results of calculations, we will find out the result of only one calculation, randomly selected from all possible ones. Fortunately, we can get a well-defined result with the help of certain additional operations, from which we can get general information about the calculations performed.

Since quantum computers are currently quite complex in design and not available for general use, one has to resort to emulating quantum algorithms on a classical computer to study them. A large number of quantum emulators have been created by now[1] [1]-[9], including the developments of such companies as Microsoft[2] and IBM[3], which even allow you to run algorithms on real quantum computers. However, it is useful to understand how an emulator works at a basic level for better understanding of this area.

Recently, computers based on the so-called "qudits", that is, multidimensional qubits, have become quite promising [10]-[19]. These are quantum systems that have not two, but three (qutrits) or more states. Qudits allow much fewer quantum systems to encode the same information, and therefore are a possible solution to the main problem of modern quantum computers – their instability. In May 2022, a group of scientists from the Russian Quantum Center received a patent for the physical implementation of a quantum computer based on qudits[4].

In this article, we will try to outline the main ideas of quantum computing, demonstrate them on the example of one of the first quantum algorithms, and then consolidate them using a simple emulation of a quantum computer on qubits and qudits in the Python language. The simplicity of the emulator will allow it to be used on normal personal computers, regardless of any additional software, and will make it available everywhere.

Qubits and operations on them

A qubit is a quantum system that has two possible measurable states[5]. These states are referred as $|0\rangle$ and $|1\rangle$. The superposition principle[6] states that a qubit can also be in a linear combination of states $|\psi\rangle = a|0\rangle + b|1\rangle$, where a and b are complex numbers [20]-[23]. Thus, the set of states of a qubit lies in a two-dimensional complex linear space, and for their designation, we can use the matrix notation

$$|\psi\rangle = \begin{pmatrix} a \\ b \end{pmatrix}.$$

Then

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ и } |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

When measuring a qubit in this state, the result can still be only $|0\rangle$ and $|1\rangle$, and it is impossible to predict in advance which state will be measured. The only available information is the probabilities of obtaining different results, equal to $aa^* = |a|^2$ and $bb^* = |b|^2$ for $|0\rangle$ and $|1\rangle$ respectively (a^* is a complex conjugate of a). The state of the qubit changes to correspond to the result of the measurement. Since nothing but $|0\rangle$ and $|1\rangle$ can be measured, the condition $|a|^2 + |b|^2 = 1$ is imposed on a and b.

Any qubit state transformations are given by a linear operator[7].

$$\hat{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$

Let $|\psi\rangle = a|0\rangle + b|1\rangle$ and $|\phi\rangle = c|0\rangle + d|1\rangle$ be two states of a qubit. We define their scalar product in the following formula

$$\langle\psi|\phi\rangle = |\psi\rangle^\dagger|\phi\rangle = \begin{pmatrix} a^* & b^* \end{pmatrix}\begin{pmatrix} c \\ d \end{pmatrix} = a^*c + b^*d,$$

where $|\psi\rangle^\dagger$ is the operation of vector transposition and complex conjugation of its components. For any state $|\psi\rangle$ $\langle\psi|\psi\rangle = 1$.

Let us define $|\hat{A}\psi\rangle = \hat{A}|\psi\rangle$, then $|\hat{A}\psi\rangle^\dagger = \langle\psi|\hat{A}^\dagger$.

$$\langle\hat{A}\psi|\phi\rangle = \langle\psi|\hat{A}^\dagger|\phi\rangle = \langle\psi|\hat{A}^\dagger\phi\rangle.$$

The operator $\hat{A}^\dagger$ is said to be conjugate to the operator $\hat{A}$. After any transformation $\hat{A}$, the following equality must hold:

$$1 = \langle\hat{A}\psi|\hat{A}\psi\rangle = \langle\psi|\hat{A}^\dagger\hat{A}|\psi\rangle,$$

[1] Boev A.S., Lvovsky A.I., Semenov A.M., et al. PolySimCIM Quantum Computing Software Emulator. Patent RF, no. 2021619034, 2021. Available at: https://www.elibrary.ru/item.asp?ysclid=lbo3rtj71q803318987&id=46313237 (accessed 16.03.2022). (In Russ.)

[2] Azure Quantum [Electronic resource]. 2022. Available at: https://azure.microsoft.com/ru-ru/services/quantum/#overview (accessed 16.03.2022). (In Russ.)

[3] Open-Source Quantum Development. *Qiskit* [Electronic resource]. 2022. Available at: https://qiskit.org (accessed 16.03.2022). (In Eng.)

[4] Patrakova A. Scientists from Russia patented a new quantum processor architecture. *CNews* [Electronic resource]. 26.05.2022. Available at: https://cnews.ru/link/n549297 (accessed 16.03.2022). (In Russ.)

[5] Nielsen M.A., Chuang I.L. Quantum Computation and Quantum Information. 10th ed. Cambridge University Press; 2011. 702 p. (In Eng.)

[6] Landau L.D., Lifshitz L.M. Quantum Mechanics: Non-Relativistic Theory. 3rd ed. Butterworth-Heinemann; 1981. 689 p. (In Eng.); Griffiths D.J. *Introduction to quantum mechanics*. Upper Saddle River, NJ: Pearson Prentice Hall; 2005. (In Eng.)

[7] Hughes C., Isaacson J., Perry A., Sun R.F., Turner, J. Quantum Computing for the Quantum Curious. Springer, Cham; 2021. 150 p. (In Eng.) doi: https://doi.org/10.1007/978-3-030-61601-4

which is obviously true if the operator is unitary, that is $\hat{A}^{\dagger} = \hat{A}^{-1}$. Therefore, all operators which we will deal with are unitary.

The simplest and at the same time quite common example of such operator is

$$\hat{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Since $\hat{X}|0\rangle = |1\rangle$ and $\hat{X}|1\rangle = |0\rangle$, $\hat{X}$ is called an analogue of the classical negation operation (NOT gate).

The Hadamard gate is another important operator.

$$\hat{H} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

It converts "pure" states into a "superposition" states and vice versa:

$$\hat{H}|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle),$$

$$\hat{H}|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle),$$

$$\hat{H}\hat{H} = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = \hat{I},$$

$$\hat{H}^{-1} = \hat{H}.$$

Systems of several qubits

Let us have two qubits in the states $|\psi_1\rangle = \begin{pmatrix} a \\ b \end{pmatrix}$ and $|\psi_2\rangle = \begin{pmatrix} c \\ d \end{pmatrix}$. The general state of the system can be specified using the Kronecker product:

$$|\psi_1\rangle \otimes |\psi_2\rangle = \begin{pmatrix} a\begin{pmatrix} c \\ d \end{pmatrix} \\ b\begin{pmatrix} c \\ d \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix}.$$

Shorthand notation $|\psi_1\rangle \otimes |\psi_2\rangle = |\psi_1\rangle|\psi_2\rangle = |\psi_1, \psi_2\rangle$ is also used. For example, the probability of finding a system in a state $|0\rangle \otimes |0\rangle = |00\rangle$ is equal to $|ac|^2 = |a|^2\,|c|^2$ (probabilities are multiplied). However, according to the principle of superposition, any linear combination of initial states is permissible, therefore, in the general case, the state vector cannot be decomposed into the product of the states of individual qubits. There is a correlation between measurements of individual qubits.

What do operators acting on pairs of qubits look like? Let $\hat{A}$ be the operator acting on the first qubit and $\hat{B}$ – the operator acting on the second one. Then

$$\hat{A}|\psi_1\rangle \otimes \hat{B}|\psi_2\rangle = (\hat{A} \otimes \hat{B})(|\psi_1\rangle \otimes |\psi_2\rangle),$$

$$\hat{A} \otimes \hat{B} = \begin{pmatrix} a_{11}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{12}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \\ a_{21}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{22}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \end{pmatrix} =$$

$$= \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}.$$

An important example of an operator acting on two qubits is $\widehat{CX}$, also called CNOT (Controlled NOT) gate. As the name suggests, it applies NOT gate depending on the state of the qubits. Namely, if the first qubit is in the state $|0\rangle$ CNOT does nothing, but if the state of the first qubit is $|1\rangle$, the NOT operation is applied to the second one. Let us write an expression for $\widehat{CX}$:

$$\widehat{CX} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \hat{I} + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \hat{X} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

A system of two qubits can be brought into the so-called "entangled state" using the operators $\hat{H}$ and $\widehat{CX}$:

$$\hat{H} \otimes \hat{I}|00\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} =$$

$$= |\psi\rangle,$$

$$\widehat{CX}|\psi\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} =$$

$$= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

In this state, the measurement of one qubit completely determines the state of the other, which can lead to interesting consequences.

The Deutsch-Jozsa algorithm

Now let us consider the simplest quantum algorithm, which is also one of the first, the Deutsch-Jozsa algorithm [24]. The Deutsch-Jozsa algorithm solves a rather artificial problem, but it clearly demonstrates the advantages of quantum computers and the basic principles of their operation.

Let a Boolean function of one variable f be given: $f: \{0; 1\} \to \{0; 1\}$. The main idea of the problem is to determine whether the function is constant or not. A classical computer has to compute both values of the function, therefore two computations are needed, but a quantum computer needs to compute it only once. It may seem that one saved computation is not much. This is true, but there is a simple generalization of this problem to the case of a function of many variables. As the number of variables increases, the number of computations required for a classical computer grows exponentially, while a quantum computer still requires only one computation.

Now let us consider only the simple case with one variable.

It can be shown that for any function f there exists a unitary operator $\hat{U}_f$, that takes the system from a pure state $|x\rangle|y\rangle$ to the state $|x\rangle|y \oplus f(x)\rangle$, which is comparable in complexity to one computation of $f(x)$ on a classical computer. Here the operation $\oplus$ denotes addition modulo 2.

Let us prepare our system in the state $|0\rangle|1\rangle$, then apply the Hadamard gate to each qubit.

$$\hat{H} \otimes \hat{H}|0\rangle|1\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle).$$

Now we apply the operator $\hat{U}_f$

$$\hat{U}_f\left(\frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)\right) =$$
$$= \frac{1}{2}\hat{U}_f(|0\rangle(|0\rangle - |1\rangle) + |1\rangle(|0\rangle - |1\rangle)) =$$
$$= \frac{1}{2}(|0\rangle(|0 \oplus f(0)\rangle - |1 \oplus f(0)\rangle)) +$$
$$+ \frac{1}{2}(|1\rangle(|0 \oplus f(1)\rangle - |1 \oplus f(1)\rangle)).$$

It can be noticed that

$$|0 \oplus f(0)\rangle - |1 \oplus f(0)\rangle = (-1)^{f(0)}(|0\rangle - |1\rangle).$$

If $f(0) = 0$, the state does not change, otherwise it just changes its sign. A similar transformation is also valid for $f(1)$. Thus, we get:

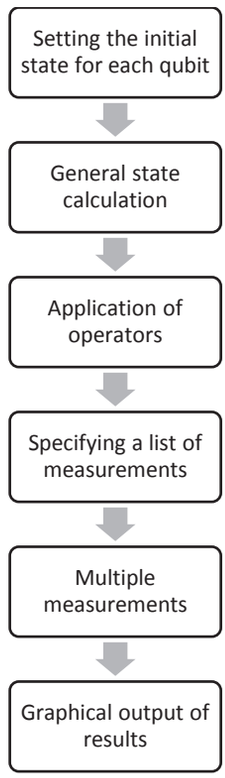
$$\frac{1}{2}\left((-1)^{f(0)}|0\rangle(|0\rangle - |1\rangle) + (-1)^{f(1)}|1\rangle(|0\rangle - |1\rangle)\right) =$$
$$= \frac{1}{2}\left((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\right)(|0\rangle - |1\rangle) =$$
$$= \frac{1}{2}(-1)^{f(0)}\left(|0\rangle + (-1)^{f(0)+f(1)}|1\rangle\right)(|0\rangle - |1\rangle)$$

$f(0) + f(1)$ is even if $f(0) = f(1)$, and is odd if $f(0) \neq f(1)$. With a constant function f the first qubit is in the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and, when the Hadamard gate is applied, goes into the state $|0\rangle$. For $f(0) \neq f(1)$ the first qubit is in the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and goes into the state $|1\rangle$. Thus, if we apply the $\hat{H} \otimes \hat{I}$ operator to our system, and then measure the state of the first qubit, we are guaranteed to know whether the function is constant.

Emulation of a quantum computer

Of course, the point of creating a quantum computer is to implement algorithms that a classical computer is not able to efficiently perform. However, an emulator is still a useful tool in the study of quantum algorithms. An emulator makes it possible to test small algorithms using commonly available tools - ordinary computers.

We use the Python programming language because of its simplicity, clarity, as well as a large number of libraries that can simplify our work. Python is one of the most widely used languages at the moment, including the field of quantum computing.



F i g. 1. Computation scheme

First, let us import the required libraries.

```
import numpy as np
import matplotlib.pyplot as plt
import itertools as it
```

The numpy library allows us to effectively handle vectors and matrices, itertools increases the efficiency of working with loops, and with the help of matplotlib we can display the results of measurements.

All information about our algorithm will be stored in the object of QuantumCircuit class

```
class QuantumCircuit:
  def __init__(self, nq, nb):
    self.nq = nq
    self.nb = nb
    zero = np.array([1, 0])
    self.initial_state = np.tile(zero, (nq, 1))
    self.qbits = np.array([])
    self.final_state = np.array([])
    self.bits = 0
    self.measurements = []
    self.results = np.zeros(2 ** self.nb)
```

The parameters n_q and n_b determine the number of qubits and classical bits in which the results of measurements of qubits are stored. The initial state of the system is stored in the variable initial_state as separate qubits so that they can be easily changed individually. By default, each qubit is in the state $|0\rangle$. The general state of the system after applying operators is stored in the final_state vector, a copy of the final state, over which measurements are made, is stored in the qubits vector, and the measurement results are stored in bits. The measuremets list contains information about all the measurements that will be taken after operators have been applied. Since, in general, the result of a quantum algorithm is probabilistic, it is useful to measure the final state of the qubits multiple times to find out the distribution of possible outcomes. The frequency of occurrence of each result is stored in a results vector.

Now let us describe the class methods required for computations.

set_initial_state(qbit, state) allows you to set an initial state for each qubit, which is then applied using the reset_state() function.

```
def set_initial_state(self, qudit, state):
  if (abs(state) ** 2).sum() == 1:
    self.initial_state[qudit] = state
  else:
    print("Incorrect state")

def reset_state(self):
  state = 1
  for s in self.initial_state:
    state = np.kron(state, s)
  self.final_state = state
  self.bits = 0
```

set_initial_state also checks the entered vector, the sum of the probabilities must always be equal to one. The function np.kron(state, s) denotes the Kronecker product $|state\rangle \otimes |s\rangle$.

The apply_operator method multiplies the matrix of operator $\hat{A}$ and the state vector.

```
def apply_operator(self, A):
    self.final_state = A @ self.final_state
```

An operator is often applied to one qubit, while the rest remain unchanged. Let n be the number of the qubit we want to change (numbering starts from zero). Then the complete operator looks like this

$$\hat{C} = \underbrace{\hat{I} \otimes \hat{I} \otimes \ldots \otimes \hat{I}}_{n} \otimes \hat{A} \otimes \underbrace{\hat{I} \otimes \hat{I} \otimes \ldots \otimes \hat{I}}_{n_q - n - 1} = \hat{I}_n \otimes \hat{A} \otimes \hat{I}_{n_q},$$

where $\hat{I}_k$ is identity matrix of size $2^k \times 2^k$.

For this purpose, we create a method apply1(A, qbit). The variable qbit here denotes the number of the qubit to which the operator is applied.

```
def apply1(self, A, qbit):
  I1 = np.identity(2**qbit)
  I2 = np.identity(2 ** (self.nq - qbit - 1))
  B = np.kron(I1, A)
  B = np.kron(B, I2)
  self.apply_operator(B)
```

Let's move on to the implementation of the measurement operation. The measurement function makes the measurement, and measure adds information about the desired measurement (the number of the measured qubit and the bit in which the result is written) to the list of measurements that will be carried out after the algorithm is executed.

```
def measurement(self, qbit, bit):
  all = np.arange(2 ** self.nq)
  indices1 = np.array([i for i in all if i & (2**qbit)], dtype=int)
  indices0 = np.delete(all, np.isin(all, indices1)) # indices0 = all \
indices1
  probability1 = (abs(self.qbits[indices1]) ** 2).sum()

  if np.random.rand() < probability1:
    self.qbits[indices1] /= np.sqrt(probability1)
    self.qbits[indices0] = np.zeros(2 ** (self.nq - 1))
    self.bits = self.bits | 2**bit
  else:
    self.qbits[indices0] /= np.sqrt(1 - probability1)
    self.qbits[indices1] = np.zeros(2 ** (self.nq - 1))


def measure(self, qbit, bit):
  self.measurements.append([qbit, bit])
```

First, indices of coordinates of the state vector, in which the desired qubit is in the state $|1\rangle$ (indices1) or $|0\rangle$ (indices0) are selected, and the probability of finding the qubit in the state $|1\rangle$ is calculated. For example, if the system is in the state $|\psi\rangle = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$ and we need to measure the second qubit, then $probability1 = |b|^2 + |d|^2$. Next, the np.random.rand() function generates a random number in the interval $[0; 1]$, and depending on its output, the system collapses into one of two states, and the measurement result is written to the self.bits variable.

The run() method performs all measurements once and returns their results.

```
def run(self):
  self.qbits = self.final_state
  for m in self.measurements:
    self.measurement(m[0], m[1])
  return self.bits
```

The simulate(num_of_iterations) method runs the algorithm a given number of times and returns the frequency of occurrence of each possible outcome.

```
def simulate(self, num_of_iterations):
  self.results = np.zeros(2 ** self.nb)
  for _ in range(num_of_iterations):
    self.results[self.run()] += 1
  self.results /= num_of_iterations
  return self.results
```

The barplot() method plots the distribution of results.

```
def barplot(self):
    labels = [] # column names
    for x in range(2 ** nb):
        b = bin(x)[2:] # trims '0b' at the beginning
        b = '0' * (nb - len(b)) + b # length alignment
        labels += [b]
    plt.bar(labels, self.results)
    plt.show()
```

Now let us create functions which apply the most common operators. This is quite simple for operators acting on a single qubit.

```
def h(self, qbit): # the Hadamard operator
    H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
    self.apply1(H, qbit)

def x(self, qbit): # the X operator
    X = np.array([[0, 1], [1, 0]])
    self.apply1(X, qbit)
```

The case with the operator $\widehat{CX}$ is more interesting. Let the number of the controlling qubit q_1 be less than the number q_2 of the qubit we want to transform. Then

$$\widehat{CX} = \hat{I}_{q_1} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \hat{I}_{n_q - q_1 + q_2 - 1} \otimes \hat{I} \otimes \hat{I}_{n_q - q_2 - 1} +$$
$$+ \hat{I}_{q_1} \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \hat{I}_{n_q - q_1 + q_2 - 1} \otimes \hat{X} \otimes \hat{I}_{n_q - q_2 - 1}$$
$$= \hat{I}_n \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \hat{I}_{n_q - q_1 - 1} +$$
$$+ \hat{I}_{q_1} \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \hat{I}_{n_q - q_1 + q_2 - 1} \otimes \hat{X} \otimes \hat{I}_{n_q - q_2 - 1}.$$

In the case of $q_2 < q_1$, the calculations are similar

```
def cx(self, q1, q2):
    X = np.array([[0, 1], [1, 0]])
    outer00 = np.diag([1, 0])
    outer11 = np.diag([0, 1])
    B1 = np.kron(np.identity(2 ** q1), outer00)
    B1 = np.kron(B1, np.identity(2 ** (self.nq - q1 - 1)))
    if q1 < q2:
        B2 = np.kron(np.identity(2**q1), outer11)
        B2 = np.kron(B2, np.identity(2**(self.nq-q1+q2-1)))
        B2 = np.kron(B2, X)
        B2 = np.kron(B2, np.identity(2**(self.nq-q2-1)))
    else:
        B2 = np.kron(np.identity(2 ** q2), X)
        B2 = np.kron(B2, np.identity(2 ** (self.nq - q2 + q1 - 1)))
        B2 = np.kron(B2, outer11)
        B2 = np.kron(B2, np.identity(2 ** (self.nq - q1 - 1)))
    B = B1 + B2
    self.apply_operator(B)
```

Now our compiler has all the basic features of a quantum computer. If needed, more operators can be added later.

Implementation of the Deutsch-Jozsa algorithm

Let us begin implementing the Deutsch–Jozsa algorithm. First, for each function $f: \{0; 1\} \to \{0; 1\}$ we need to define operators $\hat{F}$, transforming $|x\rangle|y\rangle$ into $|x\rangle|y \oplus f(x)\rangle\, x, y \in \{0; 1\}$.

Using create_operator(f) we can create the desired operator from the vector f that corresponds to function f (f[x] $= f(x)$).

```
def create_operator(f):
    Uf = np.zeros((4, 4))
    for i, j in it.product(range(2), repeat=2):
        Uf[i*2 + (j + f[i]) % 2, i*2+j] = 1
    return Uf
```

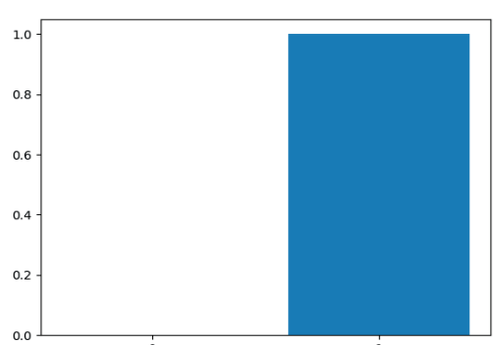
Now let us create a function that implements the Deutsch–Jozsa algorithm.

```
def deutsch_algorithm(f):
    nq = 2
    nb = 1
    qc = QuantumCircuit(nq, nb)
    Uf = create_operator(f)
    qc.set_initial_state(0, np.array([1, 0]))
    qc.set_initial_state(1, np.array([0, 1]))
    qc.reset_state()
    qc.h(0)
    qc.h(1)
    qc.apply_operator(Uf)
    qc.h(0)
    qc.measure(0, 0)
    qc.simulate(100)
    qc.barplot()
```

First, an object of the QuantumCircuit class is created, in which our algorithm will be stored. We only need two qubits and one bit for that. Then the system is initialized in the state $|0\rangle|1\rangle$ and the operators are applied, after which one hundred iterations (theoretically, one is enough, but it's better to be sure) are performed and the results are displayed.

Let us run the program:

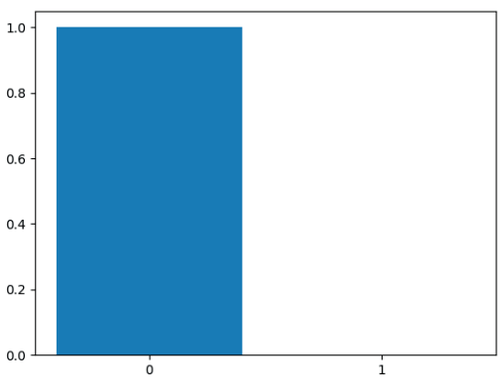
```
f = [0, 1]
deutsch_algorithm(f)
```



F i g. 2. Result of the Deutsch-Jozsa algorithm for a non-constant function

In 100% of cases, the first qubit is in the state, which is expected for a non-constant function. If instead of the operator we use the operator , we will see the opposite result.

```
g = [0, 0]
deutsch_algorithm(g)
```



F i g. 3. Result of the Deutsch-Jozsa algorithm for a constant function

Generalization of the Deutsch-Jozsa algorithm to qudits

The Deutsch-Jozsa algorithm can be rather simply generalized to quantum computers using qudits. Qudits are d-dimensional analogues of qubits, instead of two states $|0\rangle$ and $|1\rangle$ they have d different states $|0\rangle$, $|1\rangle$, $|2\rangle$, ..., $|d-1\rangle$.The state of one qudit is described by a d-dimensional normalized complex vector.

The problem for the generalized algorithm is formulated as follows: a function $f: \{0,1,\dots,d-1\} \to \{0,1,\dots,d-1\}$, can be either constant or balanced, that is, each of the d values of the function appears the same number of times, and we are required to determine what type f belongs to [25]. We consider the simplest case with a function of one variable, so the condition of it being balanced is equivalent to being bijection, but this algorithm can also be generalized to the case of functions of multiple variables.

This generalization has practically no differences from the original Deutsch–Jozsa algorithm in its structure. More complex operations for qudits, which are analogous to operations for qubits, are the only difference. We also need the operator $\hat{U}_f$, corresponding to the function f and taking the system from the pure state $|x\rangle|y\rangle$ to the state $|x\rangle|y \oplus f(x)\rangle$, where $\oplus$ now stands for addition modulo d.

First, let us consider a generalization of the Hadamard operator. It has the following form for a d-dimensional qudit:

$$\hat{H}_d = \frac{1}{\sqrt{d}} \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_d & \omega_d^2 & \omega_d^3 & \cdots & \omega_d^{d-1} \\ 1 & \omega_d^2 & \omega_d^4 & \omega_d^2 & \cdots & \omega_d^{2(d-1)} \\ 1 & \omega_d^3 & \omega_d^6 & \omega_d^2 & \cdots & \omega_d^{3(d-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_d^{d-1} & \omega_d^{2(d-1)} & \omega_d^{3(d-1)} & \cdots & \omega_d^{(d-1)^2} \end{pmatrix}.$$

where $\omega_n = e^{\frac{2\pi i}{n}}$. It is more convenient to represent this operator as a sum of *outer products* of the form $|j\rangle\langle k|$, which are linear operators. The action of such a linear operator on the vector $|\psi\rangle$ is quite intuitive: the vector $|j\rangle$ is multiplied by the inner product $\langle k|\psi\rangle$.

$$|j\rangle\langle k|\,|\psi\rangle = |j\rangle\langle k|\psi\rangle = \langle k|\psi\rangle \cdot |j\rangle$$

Then

$$\hat{H}_d = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \sum_{k=0}^{d-1} \omega_d^{jk} |j\rangle\langle k|.$$

Let us apply $\hat{H}_d$ to the state $|0\rangle$ and to the state $|1\rangle$

$$\frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \sum_{k=0}^{d-1} \omega_d^{jk} |j\rangle\langle k|0\rangle = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \omega_d^{j\cdot 0} |j\rangle\langle 0|0\rangle = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} |j\rangle$$

$$\frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \sum_{k=0}^{d-1} \omega_d^{jk} |j\rangle\langle k|1\rangle == \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \omega_d^{j\cdot 1} |j\rangle\langle 1|1\rangle = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \omega_d^{j} |j\rangle\,.$$

As in the original algorithm, we need two qudits in the state $|\psi_0\rangle = |0\rangle|1\rangle$. Applying the operator $\hat{H}_d$ to each of them, we get

$$|\psi_1\rangle = \hat{H}_d \otimes \hat{H}_d |\psi_0\rangle = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} |j\rangle \otimes \frac{1}{\sqrt{d}} \sum_{k=0}^{d-1} \omega_d^k |k\rangle.$$

Now we apply the operator $\hat{U}_f$

$$|\psi_2\rangle = \hat{U}_f |\psi_1\rangle = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \left(|j\rangle \otimes \frac{1}{\sqrt{d}} \sum_{k=0}^{d-1} \omega_d^k |k \oplus f(j)\rangle \right).$$

Addition of $f(j)$ in $|k \oplus f(j)\rangle$ is equivalent to a shift of coordinates by a constant (for a given j) number. Since $\omega_d^{k+d} = \omega_d^k$, we can rewrite $|\psi_2\rangle$ as

$$|\psi_2\rangle = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \left(|j\rangle \otimes \frac{1}{\sqrt{d}} \sum_{k=0}^{d-1} \omega_d^{k-f(j)} |k\rangle \right) = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} \omega_d^{-f(j)} |j\rangle \otimes \frac{1}{\sqrt{d}} \sum_{k=0}^{d-1} \omega_d^k |k\rangle.$$

Now, if $f(j) = f = const$, we can take the factor ω_n^{-f} out of the summation sign

$$|\psi_2\rangle = \omega_d^{-f} \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} |j\rangle \otimes \frac{1}{\sqrt{d}} \sum_{k=0}^{d-1} \omega_d^k |k\rangle\,.$$

Applying the operator $\hat{H}_d$ to the first qudit once again, we obtain

$$\frac{1}{\sqrt{d}}\sum_{m=0}^{d-1}\sum_{n=0}^{d-1}\omega_d^{mn}|m\rangle\langle n|\omega_d^{-f}\frac{1}{\sqrt{d}}\sum_{j=0}^{d-1}|j\rangle = \\ = \omega_d^{-f}\frac{1}{d}\sum_{m=0}^{d-1}\sum_{j=0}^{d-1}\omega_d^{mj}|m\rangle\langle j|j\rangle = \\ = \omega_d^{-f}\frac{1}{d}\sum_{m=0}^{d-1}\left(\sum_{j=0}^{d-1}\omega_d^{mj}\right)|m\rangle.$$

For $m = 0$, we have

$$\sum_{j=0}^{d-1}\omega_d^{0j} = d.$$

For $m \neq 0$, we have

$$\sum_{j=0}^{d-1}\omega_d^{mj} = \frac{1-\omega_d^{md}}{1-\omega_d^{m}} = 0.$$

Thus, for a constant function f, we have

$$|\psi_3\rangle = \hat{H}_d \otimes \hat{I}|\psi_2\rangle = \omega_d^{-f}|0\rangle \otimes \frac{1}{\sqrt{d}}\sum_{k=0}^{d-1}\omega_d^{k}|k\rangle.$$

The factor ω_d^{-f} is called the *global phase*, its presence does not change the properties of the state in any way. When measuring the first qudit, we will get $|0\rangle$ with one hundred percent probability.

If f is balanced, then after applying $\hat{H}_d$ to the first qudit, we will get

$$\frac{1}{\sqrt{d}}\sum_{m=0}^{d-1}\sum_{n=0}^{d-1}\omega_d^{mn}|m\rangle\langle n|\frac{1}{\sqrt{d}}\sum_{j=0}^{d-1}\omega_d^{-f(j)}|j\rangle = \\ = \frac{1}{d}\sum_{m=0}^{d-1}\left(\sum_{j=0}^{d-1}\omega_d^{mj-f(j)}\right)|m\rangle.$$

For $m = 0$, we have

$$\sum_{j=0}^{d-1}\omega_d^{mj-f(j)} = \sum_{j=0}^{d-1}\omega_d^{-f(j)} = \sum_{l=0}^{d-1}\omega_d^{l} = 0$$

since $f(j)$ runs through all values in $\{0,1,\dots,d-1\}$. Therefore, when measuring the first qudit, the probability of getting the state $|0\rangle$ is zero. We have proved that the output values of the algorithm on constant and balanced functions do not intersect, and therefore we can always determine the type of the function with one computation of it.

The result of the algorithm for a balanced function is generally not defined, since the coefficients for different $|m\rangle$ $(m \neq 0)$ are not necessarily equal to zero, but this does not affect the work of the algorithm.

Emulation of qudits

Now it's easy to generalize the emulator for qudits.

```
class QuantumCircuit:
  def __init__(self, d, nq, nd):
    self.dimension = d
    self.nq = nq
    self.nd = nd
    zero = np.array([1] + (d-1) * [0])
    self.initial_state = np.tile(zero, (nq, 1))
    self.qudits = np.array([])
    self.final_state = np.array([])
    self.digits = 0
    self.measurements = []
    self.results = np.zeros(d ** nd)
```

For the most part, the changes are rather cosmetic: renaming qubits to qudits, bits to digits, and so on. An important change is the additional self.dimension parameter, which defines the dimension of our qudits. So, wherever the number 2 was used as the dimension of qubits, it must be replaced by self.dimension. For this reason, the definition of the vector zero has also changed.

Functions such as set_initial_state, reset_state, apply_operator, measure, run remain completely unchanged (except for renaming). In the following two functions only the dimensionality of the qudits has been corrected:

```
def apply1(self, A, qudit):
  I1 = np.identity(self.dimension ** qudit)
  I2 = np.identity(self.dimension ** (self.nq - qudit - 1))
  B = np.kron(I1, A)
  B = np.kron(B, I2)
  self.apply_operator(B)

def simulate(self, num_of_iterations):
  self.results = np.zeros(self.dimension ** self.nd)
  for _ in range(num_of_iterations):
    self.results[self.run()] += 1
  self.results /= num_of_iterations
  return self.results
```

The measurement function requires the biggest changes:

```
def measurement(self, qudit, bit):
  indices0 = np.array([], dtype=int)
  bunch = np.arange(self.dimension ** (self.nq - qudit - 1))
  shift = self.dimension ** (self.nq - qudit)
  for i in range(self.dimension ** qudit):
    indices0 = np.concatenate((indices0, bunch))
    bunch = bunch + shift

  indices = [indices0 + k * self.dimension ** (self.nq - qudit - 1)
for k in range(self.dimension)]
  probabilities = [(abs(self.qudits[i]) ** 2).sum() for i in indices]

  rand_value = np.random.rand()
  for i, p in enumerate(probabilities):
    if rand_value < p:
      self.qudits[indices[i]] /= np.sqrt(probabilities[i])
```


```
        for k in range(self.dimension):
            if k != i:
                self.qudits[indices[k]] = np.zeros(self.dimension **
(self.nq - 1))
        self.digits += i * self.dimension ** bit
        break
    else:
        rand_value -= p
```

To begin with, the indices of the components corresponding to the states in which the measured qudit is equal to zero are selected into the indices0 vector. Such indices are arranged in evenly distributed bunches, the size of which depends on the qudit number. These bunches are appended to the vector indices0 with a shift by a constant number shift.

Then a list of all vectors with indices corresponding to the state number is created, and for all states the probability of its measurement is calculated (probabilities vector). After that, we create a random value rand_value and determine which interval it fell into in a loop. Further operations are similar to the case with qubits.

Changes in the barplot function are related to the labels and the increased number of states that need to be displayed. To create captions, it is convenient to write an additional function convert_base, which converts a number from the decimal number system to a representation with a dimension equal to the dimension of the qudit.

```
def convert_base(self, number):
    base = self.dimension
    string = ''
    for i in range(self.nd):
        string += str(number % base) + ' '
        number = number // base
    return string

def barplot(self):
    labels = [self.convert_base(n) for n in range(self.dimension **
self.nd)]
    plt.bar(labels, self.results)
    plt.show()
```

Creation of functions which apply operators to qudits is somewhat more complicated than for qubits, since the form of an operator depends on the dimension of the qudit, and working with qudits in general is not so common. We restrict ourselves to the Hadamard operator, which is sufficient for our purposes.

The operator is created by the function create_h, and then applied "manually" using the function apply1.

```
def create_h(self):
    d = self.dimension
    w = np.exp(2j * np.pi / d)
    H = np.zeros((d, d), dtype=np.csingle)
    for i, j in it.product(range(d), repeat=2):
        H[i, j] = w ** (i*j) / np.sqrt(d)
    return H
```

Thus, the creation of the general framework for the emulator is completed. Let us start implementing the generalized Deutsch–Jozsa algorithm.

The changes are not very significant again, we just need to replace the number 2 with the dimension of the qudit in certain places:

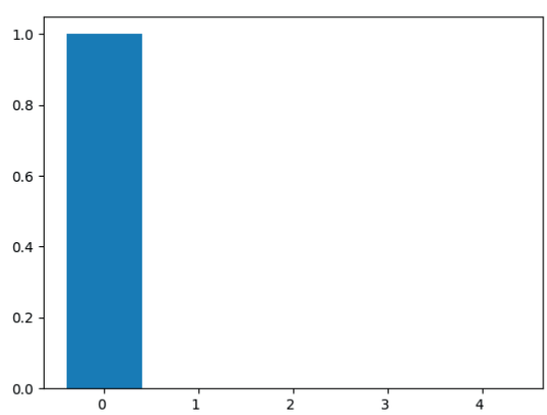
```
def create_operator(f):
    d = len(f)
    Uf = np.zeros((d**2, d**2))
    for i, j in it.product(range(d), repeat=2):
        Uf[i*d + (j + f[i]) % d, i*d+j] = 1
    return Uf

def deutsch_algorithm(f):
    d = len(f)
    nq = 2
    nd = 1
    qc = QuantumCircuit(d, nq, nd)
    Uf = create_operator(f)
    H = qc.create_h()
    qc.set_initial_state(0, np.array([1] + (d-1)*[0]))
    qc.set_initial_state(1, np.array((d-1)*[0] + [1]))
    qc.reset_state()
    qc.apply1(H, 0)
    qc.apply1(H, 1)
    qc.apply_operator(Uf)
    qc.apply1(H, 0)
    qc.measure(0, 0)
    qc.simulate(100)
    qc.barplot()
```

Let us test the algorithm on three functions:

1. Constant

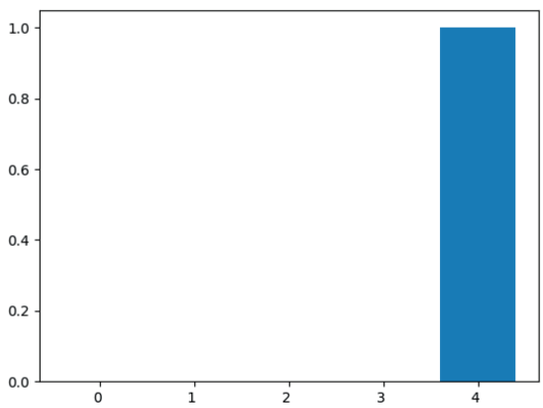
```
f = [0, 0, 0, 0, 0]
deutsch_algorithm(f)
```



F i g. 4. Result of the Deutsch-Jozsa algorithm for a constant function

2. Balanced (determined measurement result)

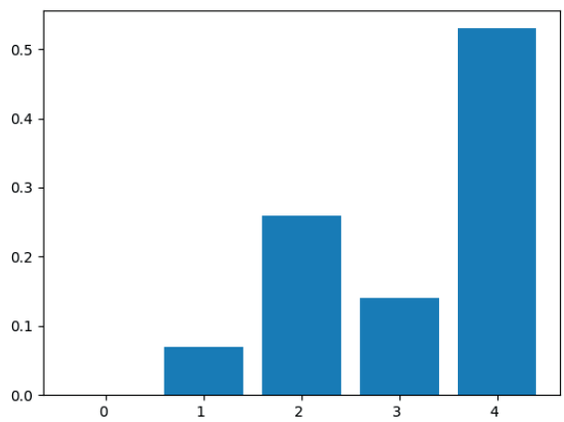
```
f = [0, 1, 2, 3, 4]
deutsch_algorithm(f)
```



F i g. 5. Result of the Deutsch-Jozsa algorithm for a balanced function

3. Balanced (undetermined measurement result)

```
f = [0, 2, 1, 3, 4]
deutsch_algorithm(f)
```



F i g. 6. Result of the Deutsch-Jozsa algorithm for a balanced function with undetermined measurement result

As you can see, the algorithm returns zero when the function is constant, and not zero when the function is balanced.

Conclusion

The authors tried to present the basics of quantum computing and their emulation using classical computers in an accessible form, as well as highlight current developments in the field such as the use of qudits.

A basic quantum computer emulator was developed and tested in the Python programming language. Thus, the authors showed the availability of quantum computing for any person with no access to a quantum computer.

The Deutsch–Jozsa algorithm and its generalization to qudits are analyzed in detail, and then implemented using the emulator to give a clear demonstration of the capabilities of quantum algorithms.

References

[1] Tretiak M.A., Shchekaturin A.E., Pilipenko I.A., Kravchenko V.O., Cherkesova L.V. Analysis of the advantages and disadvantages of quantum emulators on the example of interaction with the user. *Scientific Review. Technical science.* 2020; (5):27-37. Available at: https://www.elibrary.ru/item.asp?id=44149778 (accessed 16.03.2022). (In Russ., abstract in Eng.)

[2] Baskakov P.E., Khabovets Y.Yu., Pilipenko I.A., Kravchenko V.O., Cherkesova L.V. Tools for Performing and Emulating Quantum Computing. *Vestnik Novosibirskogo gosudarstvennogo universiteta. Seriâ: informacionnye tehnologii v obrazovanii* = Vestnik NSU. Series: Information Technologies. 2020; 18(2):43-53. (In Russ., abstract in Eng.) https://doi.org/10.25205/1818-7900-2020-18-2-43-53

[3] Kiktenko E.O., Nikolaeva A.S., Fedorov A.K. Quantum computing using multilevel quantum systems. *Nanoindustry*. 2020; 13(S4):649-651. (In Russ., abstract in Eng.) doi: https://doi.org/10.22184/1993-8578.2020.13.4s.649.651

[4] Smirnova T.S., Shvetskiy M.V. A visual emulator of the Bloch vector and sphere as a means of teaching quantum computing. *The Scientific Opinion.* 2021; (9):76-82. (In Russ., abstract in Eng.) doi: https://doi.org/10.25807/22224378_2021_9_76

[5] Grigoryeva G.M., Khodchenkov V.Yu. On the possibility of building a quantum computer emulator using XMM registers. *Sistemy komp'yuternoj matematiki i ih prilozheniya* = Computer Mathematics Systems and Their Applications. 2021; (22):113-116. Available at: https://www.elibrary.ru/item.asp?id=46649884 (accessed 16.03.2022). (In Russ., abstract in Eng.)

[6] Arute F., Arya K., Babbush R., et al. Quantum supremacy using a programmable superconducting processor. *Nature.* 2019; 574:505-510. (In Eng.) doi: https://doi.org/10.1038/s41586-019-1666-5

[7] Guzik V.P., Gushanskiy S.M. Development of emulator for quantum computers. *Izvestiya SFedU. Engineering Sciences.* 2010; (2):73-79. Available at: https://www.elibrary.ru/item.asp?id=13617268 (accessed 16.03.2022). (In Russ., abstract in Eng.)

[8] Solovyev V.M. Quantum Computers and Quantum Algorithms. Part 1. Quantum Computers. *Izvestiya of Saratov University. New Series. Series: Mathematics. Mechanics. Informatics.* 2015; 15(4):462-477. (In Russ., abstract in Eng.) doi: https://doi.org/10.18500/1816-9791-2015-15-4-462-477

[9] Solovyev V.M. Quantum Computers and Quantum Algorithms. Part 2. Quantum Algorithms. *Izvestiya of Saratov University. New Series. Series: Mathematics. Mechanics. Informatics.* 2016; 16(1):104-112. (In Russ., abstract in Eng.) doi: https://doi.org/10.18500/1816-9791-2016-16-1-104-112

[10] Ladd T., Jelezko F., Laflamme R., et al. Quantum computers. *Nature.* 2010; 464:45-53. (In Eng.) doi: https://doi.org/10.1038/nature08812
[11] Kiktenko E.O., Fedorov A.K., Man'ko O.V., Man'ko V.I. Multilevel superconducting circuits as two-qubit systems: Operations, state preparation, and entropic inequalities. *Physical Review A.* 2015; 91(4):042312. (In Eng.) doi: https://doi.org/10.1103/PhysRevA.91.042312
[12] Imany P., Jaramillo-Villegas J.A., Alshaykh M.S., et al. High-dimensional optical quantum logic in large operational spaces. *npj Quantum Information*. 2019; 5:59. (In Eng.) doi: https://doi.org/10.1038/s41534-019-0173-8
[13] Wang Y., Hu Z., Sanders B.C., Kais S. Qudits and High-Dimensional Quantum Computing. *Frontiers in Physics*. 2020; 8:589504. (In Eng.) doi: https://doi.org/10.3389/fphy.2020.589504
[14] Kiktenko E.O., Nikolaeva A.S., Xu P., Shlyapnikov G.V., Fedorov A.K. Scalable quantum computing with qudits on a graph. *Physical Review A.* 2020; 101(2):022304. (In Eng.) doi: https://doi.org/10.1103/PhysRevA.101.022304
[15] Moreno-Pineda E., Godfrin C., Balestro F., Wernsdorfer W., Ruben M. Molecular spin qudits for quantum algorithms. *Chemical Society Reviews*. 2018; 47(2), 501-513. (In Eng.) doi: https://doi.org/10.1039/C5CS00933B
[16] Ringbauer M., Meth M., Postler L., Stricker R., Blatt R., Schindler P., Monz T. A universal qudit quantum processor with trapped ions. *Nature Physics.* 2022; 18:1053-1057. (In Eng.) doi: https://doi.org/10.1038/s41567-022-01658-0
[17] Tacchino F., Chiesa A., Sessoli R., Tavernelli I., Carretta S. A proposal for using molecular spin qudits as quantum simulators of light–matter interactions. *Journal of Materials Chemistry C*. 2021; 9(32):10266-10275. (In Eng.) doi: https://doi.org/10.1039/D1TC00851J
[18] Lu H.H., Hu Z., Alshaykh M.S., Moore A.J., Wang Y., Imany P., Weiner A.M., Kais S. Quantum Phase Estimation with Time-Frequency Qudits in a Single Photon. *Advanced Quantum Technologies.* 2020; 3(2):1900074. (In Eng.) doi: https://doi.org/10.1002/qute.201900074
[19] Fischer L.E., Chiesa A., Tacchino F., Egger D.J., Carretta S., Tavernelli I. Towards universal gate synthesis and error correction in transmon qudits. *arXiv*:2212.04496. 2022. (In Eng.) doi: https://doi.org/10.48550/arXiv.2212.04496
[20] Chi Y., Huang J., Zhang Z., et al. A programmable qudit-based quantum processor. *Nature Communications*. 2022; 13:1166. (In Eng.) doi: https://doi.org/10.1038/s41467-022-28767-x
[21] Brennen G.K., O'Leary D.P., Bullock S.S. Criteria for exact qudit universality. *Physical Review A*. 2005; 71(5):052318. (In Eng.) doi: https://doi.org/10.1103/PhysRevA.71.052318
[22] Biamonte J., Wittek P., Pancotti N., et al. Quantum machine learning. *Nature.* 2017; 549:195-202. (In Eng.) doi: https://doi.org/10.1038/nature23474
[23] Aryte F., Arya K., Babbush R., et al. Quantum supremacy using a programmable supercon-ducting processor. *Nature.* 2019; 574:505-510. (In Eng.) doi: https://doi.org/10.1038/s41586-019-1666-5
[24] Deutsch D., Jozsa R. Rapid Solution of Problems by Quantum Computation. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences.* 1992; 439(1907):553-558. (In Eng.) doi: https://doi.org/10.1098/rspa.1992.0167
[25] Fan Y. A Generalization of the Deutsch-Jozsa Algorithm to Multi-Valued Quantum Logic. *37th International Symposium on Multiple-Valued Logic (ISMVL'07).* IEEE Computer Society, Oslo, Norway; 2007. p. 1-5. (In Eng.) doi: https://doi.org/10.1109/ISMVL.2007.3

Submitted 28.05.2022; approved after reviewing 30.06.2022; accepted for publication 09.07.2022.

About the authors:

Andrey S. Andreev, Student of the Department of Higher Mathematics, Faculty of Fundamental Sciences, Bauman Moscow State Technical University (5, 2-nd Baumanskaya St., building 2, Moscow 105005, Russian Federation), **ORCID: https://orcid.org/0000-0002-4124-4146,** andreevas3@student.bmstu.ru

Pavel V. Khrapov, Associate Professor of the Department of Higher Mathematics, Faculty of Fundamental Sciences, (5, 2-nd Baumanskaya St., building 2, Moscow 105005, Russian Federation), Cand.Sci. (Phys.-Math.), **ORCID: https://orcid.org/0000-0002-6269-0727,** khrapov@bmstu.ru

All authors have read and approved the final manuscript.

Список использованных источников

[1] Анализ достоинств и недостатков квантовых эмуляторов на примере взаимодействия с пользователем / М. А. Третьяк, А. Е. Щекатурин, И. А. Пилипенко [и др.] // Научное обозрение. Технические науки. 2020. № 5. С. 27-37. URL: https://www.elibrary.ru/item.asp?id=44149778 (дата обращения: 16.03.2022).
[2] Инструменты для выполнения и эмуляции квантовых вычислений / П. Е. Баскаков, Ю. Ю. Хабовец, И. А. Пилипенко [и др.] // Вестник НГУ. Серия: Информационные технологии. 2020. Т. 18, № 2. С. 43-53. doi: https://doi.org/10.25205/1818-7900-2020-18-2-43-53
[3] Киктенко Е. О., Николаева А. С., Федоров А. К. Квантовые вычисления с использованием многоуровневых квантовых систем // Наноиндустрия. 2020. Т. 13, № S4(99). С. 649-651. doi: https://doi.org/10.22184/1993-8578.2020.13.4s.649.651
[4] Смирнова Т. С., Швецкий М. В. Визуальный эмулятор вектора и сферы Блоха как средство обучения квантовым вычислениям // Научное мнение. 2021. № 9. С. 76-82. doi: https://doi.org/10.25807/22224378_2021_9_76

[5] Григорьева Г. М., Ходченков В. Ю. О возможности построения эмулятора квантового компьютера с использованием XMM регистров // Системы компьютерной математики и их приложения. 2021. № 22. С. 113-116. URL: https://www.elibrary.ru/item.asp?id=46649884 (дата обращения: 16.03.2022).
[6] Quantum supremacy using a programmable superconducting processor / F. Arute, K. Arya, R. Babbush [и др.] // Nature. 2019. Vol. 574. P. 505-510. doi: https://doi.org/10.1038/s41586-019-1666-5
[7] Гузик В. Ф., Гушанский С. М. Разработка эмуляторов для квантовых вычислителей // Известия ЮФУ. Технические науки. 2010. № 2(103). С. 73-79. URL: https://www.elibrary.ru/item.asp?id=13617268 (дата обращения: 16.03.2022).
[8] Соловьев В. М. Квантовые компьютеры и квантовые алгоритмы. Часть 1. квантовые компьютеры // Известия Саратовского университета. Новая серия. Серия: Математика. Механика. Информатика. 2015. Т. 15, № 4. С. 462-477. doi: https://doi.org/10.18500/1816-9791-2015-15-4-462-477
[9] Соловьев В. М. Квантовые компьютеры и квантовые алгоритмы часть 2. Квантовые алгоритмы // Известия Саратовского университета. Новая серия. Серия: Математика. Механика. Информатика. 2016. Т. 16, № 1. С. 104-112. doi: https://doi.org/10.18500/1816-9791-2016-16-1-104-112
[10] Quantum computers / T. Ladd, F. Jelezko, R. Laflamme [и др.] // Nature. 2010. Vol. 464. P. 45-53. doi: https://doi.org/10.1038/nature08812
[11] Multilevel superconducting circuits as two-qubit systems: Operations, state preparation, and entropic inequalities / E. O. Kiktenko [и др.] // Physical Review A. 2015. Vol. 91, issue 4. Article number: 042312. doi: https://doi.org/10.1103/PhysRevA.91.042312
[12] High-dimensional optical quantum logic in large operational spaces / P. Imany, J. A. Jaramillo-Villegas, M. S. Alshaykh [и др.] // npj Quantum Information. 2019. Vol. 5. Article number: 59. doi: https://doi.org/10.1038/s41534-019-0173-8
[13] Qudits and High-Dimensional Quantum Computing / Y. Wang [и др.] // Frontiers in Physics. 2020. Vol. 8. Article number: 589504. doi: https://doi.org/10.3389/fphy.2020.589504
[14] Scalable quantum computing with qudits on a graph / E. O. Kiktenko [и др.] // Physical Review A. 2020. Vol. 101, issue 2. Article number: 022304. doi: https://doi.org/10.1103/PhysRevA.101.022304
[15] Molecular spin qudits for quantum algorithms / E. Moreno-Pineda [и др.] // Chemical Society Reviews. 2018. Vol. 47, issue 2. P. 501-513. doi: https://doi.org/10.1039/C5CS00933B
[16] A universal qudit quantum processor with trapped ions / M. Ringbauer [и др.] // Nature Physics. 2022. Vol. 18. P. 1053-1057. doi: https://doi.org/10.1038/s41567-022-01658-0
[17] A proposal for using molecular spin qudits as quantum simulators of light–matter interactions / F. Tacchino [и др.] // Journal of Materials Chemistry C. 2021. Vol. 9, issue 32. P. 10266-10275. doi: https://doi.org/10.1039/D1TC00851J
[18] Quantum Phase Estimation with Time-Frequency Qudits in a Single Photon / H. H. Lu [и др.] // Advanced Quantum Technologies. 2020. Vol. 3, issue 2. Article number: 1900074. doi: https://doi.org/10.1002/qute.201900074
[19] Towards universal gate synthesis and error correction in transmon qudits // L. E. Fischer [и др.] // arXiv:2212.04496. 2022. doi: https://doi.org/10.48550/arXiv.2212.04496
[20] A programmable qudit-based quantum processor / Y. Chi, J. Huang, Z. Zhang [и др.] // Nature Communications. 2022. Vol. 13. Article number: 1166. doi: https://doi.org/10.1038/s41467-022-28767-x
[21] Brennen G. K., O'Leary D. P., Bullock S. S. Criteria for exact qudit universality // Physical Review A. 2005. Vol. 71, issue 5. Article number: 052318. doi: https://doi.org/10.1103/PhysRevA.71.052318
[22] Quantum machine learning / J. Biamonte, P. Wittek, N. Pancotti [и др.] // Nature. 2017. Vol. 549. P. 195-202. doi: https://doi.org/10.1038/nature23474
[23] Quantum supremacy using a programmable supercon-ducting processor / F. Aryte, K. Arya, R. Babbush [и др.] // Nature. 2019. Vol. 574. P. 505-510. doi: https://doi.org/10.1038/s41586-019-1666-5
[24] Deutsch D., Jozsa R. Rapid Solution of Problems by Quantum Computation // Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences. 1992. Vol. 439, issue 1907. P. 553-558. doi: https://doi.org/10.1098/rspa.1992.0167
[25] Fan Y. A Generalization of the Deutsch-Jozsa Algorithm to Multi-Valued Quantum Logic // 37th International Symposium on Multiple-Valued Logic (ISMVL'07). Oslo, Norway: IEEE Computer Society, 2007. P. 1-5. doi: https://doi.org/10.1109/ISMVL.2007.3

Поступила 28.05.2022; одобрена после рецензирования 30.06.2022; принята к публикации 09.07.2022.

Об авторах:

Андреев Андрей Сергеевич, студент кафедры высшей математики, факультет фундаментальных наук, ФГБОУ ВО «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (105005, Российская Федерация, г. Москва, ул. 2-я Бауманская, д. 5, к. 1), **ORCID: https://orcid.org/0000-0002-4124-4146,** andreevas3@student.bmstu.ru

Храпов Павел Васильевич, доцент кафедры высшей математики, факультет фундаментальных наук, ФГБОУ ВО «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (105005, Российская Федерация, г. Москва, ул. 2-я Бауманская, д. 5, к. 1), кандидат физико-математических наук, **ORCID: https://orcid.org/0000-0002-6269-0727,** khrapov@bmstu.ru

Все авторы прочитали и одобрили окончательный вариант рукописи.