

Quantum Computing

Marta Bubel Ewelina Kolba

21-05-2022

Implementation a simple version of various logic functions and the Deutsch two qubit quantum algorithm.

Table of contents

1	Introduction to Quantum Computing	3
1.1	Superposition	3
1.2	Entanglement	3
1.3	Fragility	3
1.4	No cloning	3
2	Logic gates	4
2.1	Classical logic gates	4
2.1.1	NOT	4
2.1.2	AND	4
2.1.3	OR	4
2.1.4	NAND	5
2.1.5	NOR	5
2.1.6	XNOR	5
2.2	Quantum gates	6
2.2.1	The X-Gate	6
2.2.2	The Y & Z-gate	7
2.2.3	The CNOT-gate	8
2.2.4	The Toffoli-gate	9
2.2.5	The Hadamard-gate	10
3	Deutsch algorithm	11
3.1	The Deutsch algorithm code:	12
3.2	The Deutsch algorithm scheme:	13
	References	14

1 Introduction to Quantum Computing

Quantum Computers are different from the digital computing that drives today's data centers, cloud environments, PCs and other devices. Digital computation requires data to be encoded into binary digits (bits), each of which is always in one of two definite states (0 or 1). A quantum bit is a quantum system that has two horizontal (degrees of freedom). However, quantum computation uses quantum bits (qubits), which can be in multiple states simultaneously. As a result, operations on qubits can amount to a large number of calculations in parallel. It has been shown that in theory, some specific problems should be solable in much less time on a quantum computer than using the best known algorithms for a conventional computer. Here are four key concepts that are the foundation of quantum computing.

1.1 Superposition

Classical physics can be either 0 or 1 bit. In quantum physics a qubit would be both 0 and 1 and spin simultaneously up and down. One way to represent this with mathematics is to use two orthogonal vectors.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Orthogonal Vectors

1.2 Entanglement

Entanglements gives quantum computing the ability to scale exponentially. If one qubit simultaneously represents two states, two qubits represents four states when coupled together. They can no longer be treated independently, they now form a coupled or entangled, super state. As more qubits link together, the number of states exponentially increase, which could lead to a computer with astronomically large computing power.

1.3 Fragility

Quantum states are quite fragile. If you measure, observe, touch or perturb any of these states, they collapse to a classical state. The states don't stick around for very long, which is why quantum computers are currently hard to build.

1.4 No cloning

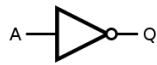
A corollary to fragility is the "No cloning theorem". In classical physics, if we have two bits, one can copy or eavesdrop and recreate the information. In contrast, the information entangled within a set of qubits will be lost if someone tries to observe or copy them. A quantum state cannot be copied without the sender or receiver realizing this. This concept serves as the basis of quantum communications.

2 Logic gates

A logic gate is an idealized or physical device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. Logic circuits include such devices as multiplexers, registers, arithmetic logic units (ALUs), and computer memory, all the way up through complete microprocessors, which may contain more than 100 million gates.

2.1 Classical logic gates

2.1.1 NOT



NOT - gate

INPUT		OUTPUT
A		
0		1
1		0

*NOT -
Truth Table*

2.1.2 AND



AND - gate

INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

*AND -
Truth Table*

2.1.3 OR

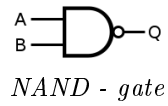


OR - gate

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

*OR - Truth
Table*

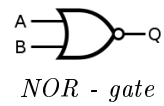
2.1.4 NAND



INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0

*NAND -
Truth Table*

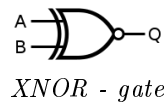
2.1.5 NOR



INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	0

*NOR -
Truth Table*

2.1.6 XNOR



INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	1

*XNOR -
Truth Table*

2.2 Quantum gates

2.2.1 The X-Gate

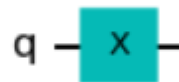
It simply flips the bit value: 0 becomes 1 and 1 becomes 0. For qubits, it is an operation called x that does the job of the NOT. $|0\rangle \rightarrow |1\rangle$ $|1\rangle \rightarrow |0\rangle$ The X-gate is represented by the Pauli-X matrix. Qbit negation is defined by the following transformations:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

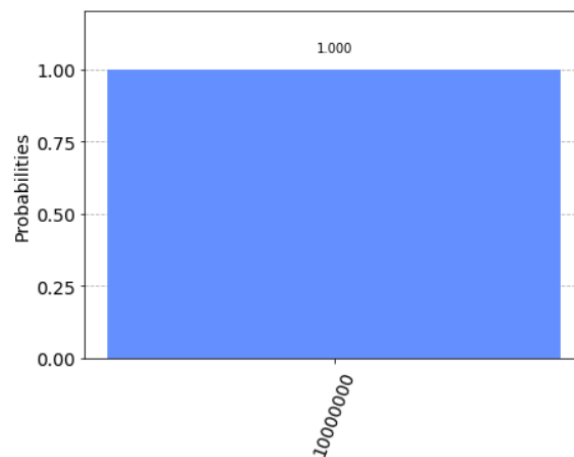
The X-Gate

```
qc = QuantumCircuit(1)
qc.x(0)
qc.draw()
```

The X-Gate Code



The X-Gate Schema



The X-Gate simulation

2.2.2 The Y & Z-gate

Y-gate: $|0\rangle \rightarrow i|1\rangle$

$|1\rangle \rightarrow -i|0\rangle$

Z-gate: $|0\rangle \rightarrow |0\rangle$

$|1\rangle \rightarrow -|1\rangle$


Similarly to the X-gate, the Y & Z Pauli matrices also act as the Y & Z-gates:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

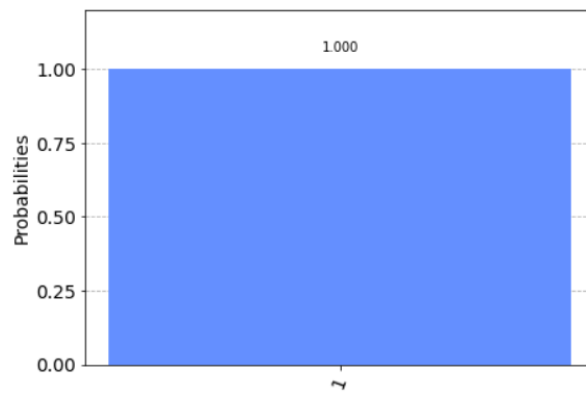
$$Y = -i|0\rangle\langle 1| + i|1\rangle\langle 0| \quad Z = |0\rangle\langle 0| - |1\rangle\langle 1|$$

The Y & Z-Gate

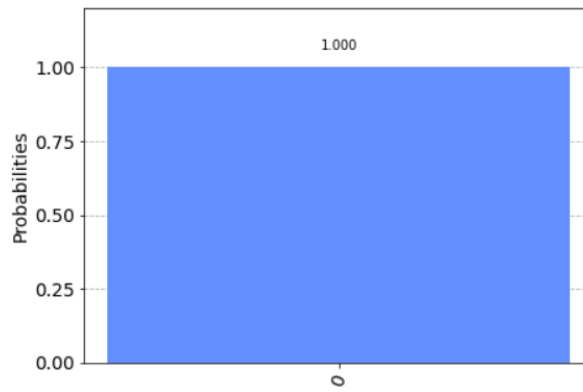
```
qc.y(0)
qc.z(0)
qc.draw()
```



The Y & Z-Gate Code
The Y & Z-Gate Schema



The Y-Gate simulation



The Z-gate simulation

2.2.3 The CNOT-gate

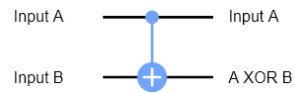
In quantum computers, the job of the XOR gate is done by the controlled-NOT gate. Since that's quite a long name, we usually just call it the CNOT. In Qiskit its name is `cx`, which is even shorter.

Input 1	Input 2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

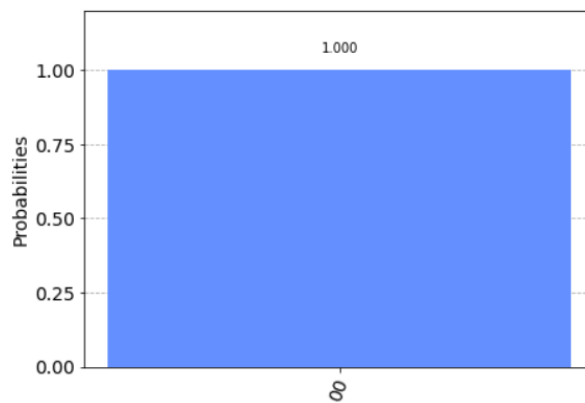
*The CNOT-gate
Truth Table*

```
qc_cnot = QuantumCircuit(2)
qc_cnot.cx(0,1)
qc_cnot.draw()
```

The CNOT-gate Code



The CNOT-gate Schema



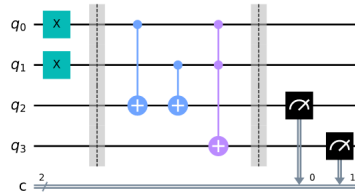
The CNOT-Gate simulation

2.2.4 The Toffoli-gate

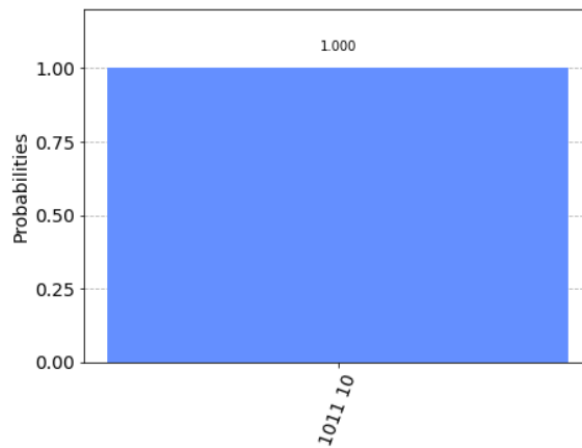
This new gate is called the Toffoli. For those of you who are familiar with Boolean logic gates, it is basically an AND gate. In Qiskit, the Toffoli is represented with the `ccx` command.

```
qc_ha = QuantumCircuit(4,2)
qc_ha.x(0)
qc_ha.x(1)
qc_ha.barrier()
qc_ha.cx(0,2)
qc_ha.cx(1,2)
qc_ha.ccx(0,1,3)
qc_ha.barrier()
qc_ha.measure(2,0)
qc_ha.measure(3,1)
qc_ha.draw()
```

The Toffoli-gate Code



The Toffoli-gate Schema



The Toffoli-Gate simulation

2.2.5 The Hadamard-gate

The Hadamard gate (H-gate) is a fundamental quantum gate. It allows us to move away from the poles of the Bloch sphere and create a superposition of $|0\rangle$ and $|1\rangle$. This gate represents rotation by Π around the $X + Z$ axis.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The Hadamard-gate

$$H|0\rangle = |+\rangle$$

$$H|1\rangle = |-\rangle$$

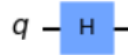
The hadamard-gate

$$|0\rangle \rightarrow (|0\rangle + |1\rangle) / \sqrt{2}$$

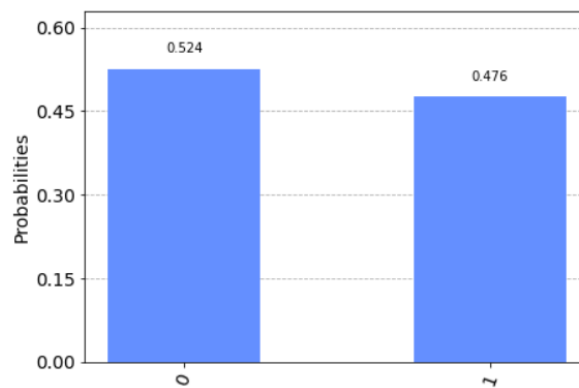
$$|1\rangle \rightarrow (|0\rangle - |1\rangle) / \sqrt{2}$$

```
qc_H = QuantumCircuit(1)
qc_H.h(0)
qc_H.draw()
```

The Hadamard-gate Code



The Hadamard-gate Schema



The Hadamard-Gate simulation

3 Deutsch algorithm

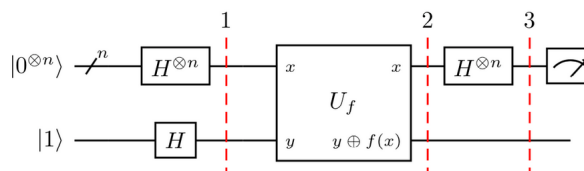
The Deutsch-Jozsa algorithm, was the first example of a quantum algorithm that performs better than the best classical algorithm. There is given a hidden Boolean function f , which takes as input a string of bits, and returns either 0 or 1, that is:

$$f(\{x_0, x_1, x_2, \dots\}) \rightarrow 0 \text{ or } 1, \text{ where } x_n \text{ is } 0 \text{ or } 1$$

The boolean function

The property of the given Boolean function is that it is guaranteed to either be balanced or constant. A constant function returns all 0's or all 1's for any input, while a balanced function returns 0's for exactly half of all inputs and 1's for the other half. The Deutsch-Jozsa algorithm task is to determine whether the given function is balanced or constant. This problem could be solve by quantum computer provided it has the function f implemented as a quantum oracle, which maps the state $|x\rangle|y\rangle$ to state $|x\rangle|y \oplus f(x)\rangle$ (where \oplus is addition modulo 2).

Below is the generic circuit for the Deutsch-Jozsa algorithm.



*The generic circuit for
the Deutsch-Jozsa
algorithm*

The steps of the algorithm:

1. Prepare two quantum registers. The first is an n -qubit register initialized to $|0\rangle$, and the second is a one-qubit register initialized to $|1\rangle$.
2. Apply a Hadamard gate to each qubit.
3. Apply the quantum oracle $|x\rangle|y\rangle$ to $|x\rangle|y \oplus f(x)\rangle$.
4. Apply a Hadamard gate to each qubit in the first register.
5. Measure the first register. Notice that the probability of measuring which evaluates to 1 if $f(x)$ is constant and 0 if $f(x)$ is balanced.

When the oracle is constant, it has no effect on the input qubits, and the quantum states before and after querying the oracle are the same. When the oracle is balanced, phase kickback adds a negative phase to exactly half states. The quantum state after querying the oracle is orthogonal to the quantum state before querying the oracle. Thus, in Step 4, when applying the H-gates, we must end up with a quantum state that is orthogonal to $|00 \dots 0\rangle$. This means we should never measure the all-zero state.

3.1 The Deutch algorithm code:

```
# initialization
import numpy as np

# importing Qiskit
from qiskit import IBMQ, Aer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, assemble, transpile

# import basic plot tools
from qiskit.visualization import plot_histogram
n = 3

const_oracle = QuantumCircuit(n+1)

output = np.random.randint(2)
if output == 1:
    const_oracle.x(n)

const_oracle.draw()
balanced_oracle = QuantumCircuit(n+1)
b_str = "101"

# Place X-gates
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)
# Use barrier as divider
balanced_oracle.barrier()

# Controlled-NOT gates
for qubit in range(n):
    balanced_oracle.cx(qubit, n)

balanced_oracle.barrier()

# Place X-gates
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

dj_circuit = QuantumCircuit(n+1, n)
# Apply H-gates
for qubit in range(n):
    dj_circuit.h(qubit)

# Put qubit in state  $|-\rangle$ 
dj_circuit.x(n)
dj_circuit.h(n)
dj_circuit.draw()

# Add oracle
dj_circuit += balanced_oracle

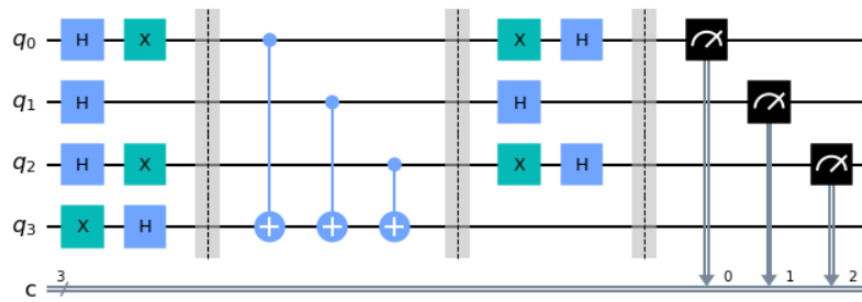
# Repeat H-gates
for qubit in range(n):
    dj_circuit.h(qubit)
dj_circuit.barrier()

# Measure
for i in range(n):
    dj_circuit.measure(i, i)

# Display circuit
dj_circuit.draw()

# use local simulator
aer_sim = Aer.get_backend('aer_simulator')
qobj = assemble(dj_circuit, aer_sim)
results = aer_sim.run(qobj).result()
answer = results.get_counts()
#plot_histogram(answer)
```

3.2 The Deutch algorithm scheme:



The Deutch Algorithm scheme

References

- [1] MARK OSKIN, *Quantum Computing - Lecture Notes*
- [2] DAVID McMAHON, *Quantum Computing Explained*, A John Wiley & Sons, Inc., Publication, 2007.
- [3] QISKIT.ORG, *Qiskit 0.36.2 documentation*