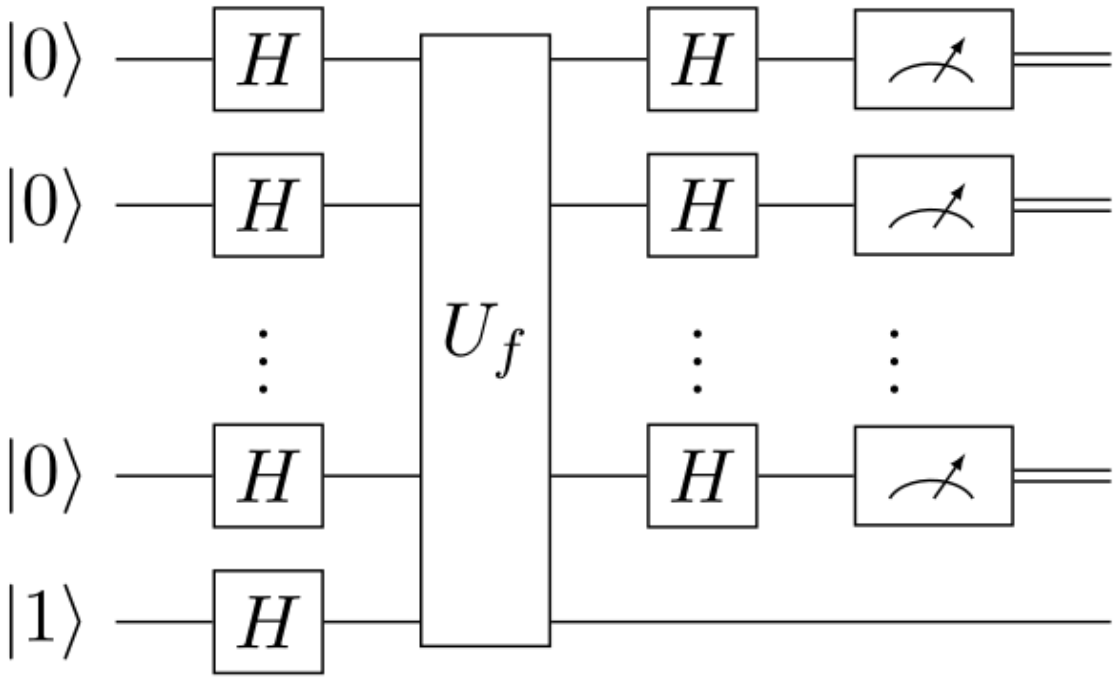


Implementation of the Deutsch-Jozsa Quantum Algorithm

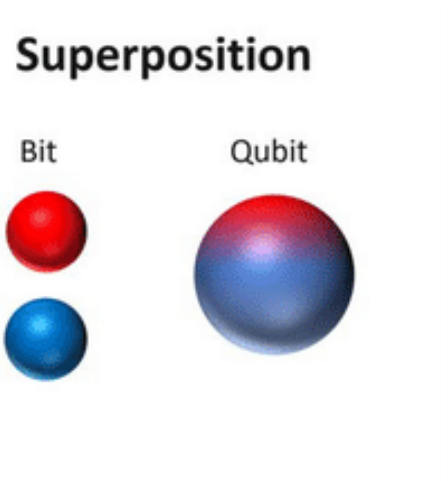
Ethan Linton

Table of Contents

Introduction

Quantum computers offer a significant edge in tackling certain problem sets. Their computational capabilities stem from utilising the principles of quantum superposition, and more importantly, entanglement in-which I'll explain more in-depth further on.

Superposition allows objects present within a quantum environment to exist in one or more states or locations simultaneously. What this means, is that an object present within a quantum environment can be in two states at a single time while remaining a single object i.e.



Instead of just being in one state, like a regular computer bit being either 0 or 1, a quantum bit (qubit) can be both 0 and 1 simultaneously. This allows quantum computers to explore many potential solutions simultaneously, making them incredibly powerful for certain types of problems.

In comparison, a classical computer (12-bit) can store 1 out of 12 to the power 2 = 4096 (4k) possible combinations. On the other hand, a 12-QUBIT quantum system can store 12 to the power 2 = 4096 possible values *at the same time*, thus showing an advantage in solving problems.

Throughout this document, I will be exploring the usage of the quantum algorithm "Deutsch Jozsa" and practical implementation of such algorithm via the use of a real quantum computer offered by the IBM Cloud Q experience.

Classical compute programming

The instructions you write in code aren't directly executed by the computer. Instead, a compiler translates human-written code into machine code, which the computer understands and executes. This machine code consists of specific sets of 1s and 0s that instruct the computer on which bits to manipulate and where to send electrical signals.

The processor runs this machine code, directing the computer to perform mathematical tasks using logic gates. These gates manipulate the actual bits within the computer to execute computations. Serving as the foundation of classical computing, logic gates take electrical signals as inputs and produce different signals as outputs based on the inputs received. For instance, an 'AND' gate permits current flow only when both input wires carry current, indicating both inputs are 1s. By linking these logic gates, we construct more intricate circuits.

The code transforms into machine code, which the processor reads and utilizes logic gates for operation.

Quantum computing

Quantum computers are also created using high level languages, these languages are created on classical computers, compiled on classical computers, and executed on quantum computers.

Quantum computers utilizes quantum mechanics such as Superposition and entanglement. These two mechanics set the foundation for quantum computing. Superposition and entanglement in quantum algorithms to solve problems.

Superposition refers to a quantum system's ability to exist in multiple states simultaneously, with the specific state uncertain until measurement occurs. These states could involve properties like energy, spin, or momentum, varying among different qubits. Upon measurement, the superposition collapses into a definite state, determined by mathematical coefficients associated with each state in the superposition. The mechanism behind this collapse remains a subject of debate.

Entanglement is like having two quantum particles that become intertwined in such a way that the state of one particle instantly influences the state of the other, regardless of the distance between them. This means that if you measure the state of one particle, it immediately determines the state of the other particle, even if they are light-years apart. In quantum computing, the combination of superposition and entanglement in quantum algorithms to solve problems.

How do we implement these quantum algorithms?
To explain how quantum algorithms are implemented, let's delve into implementing programs on neutral atom quantum computers. This process remains consistent across all quantum computers, with differences mainly arising in software/hardware interaction.

In a neutral atom quantum computer, each atom serves as a qubit. Quantum logic gates manipulate qubits in a predictable manner. In such systems, these gates are crafted by directing lasers at individual atoms (qubits) using specific frequencies. The choice of frequency depends on the intended logic gate (like Hadamard gate, Z Gate, etc.). Simply put, selecting the frequency aims to shift the qubit from state 1 to state 0. This method, involving laser manipulation of atoms to alter their composition, is called superposition, and it facilitates the creation of single qubit logic gates.

But what about multi-qubit logic gates? This is where entanglement steps in. Instead of relying on superposition, as we do for single-qubit logic gates, we employ entanglement to craft multi-qubit logic gates. In quantum computing, all logic gates must possess an equal number of inputs and outputs. This differs from classical computing, where a classical 'AND' logic gate can have two inputs and one output.

Implementing multi-qubit logic gates varies depending on the platform, but in most cases, it involves forcing a known interaction between two qubits. In a neutral atom system, this often entails elevating one of our atoms to a state called 'Rydberg'—a high-energy state with the electron distant from the nucleus. The resulting charge separation between the positive nucleus and the negative electron generates a dipole moment, creating an electric field. This electric field interacts with nearby atoms, preventing them from entering a Rydberg state. Now, as the high-energy atom dictates the allowable states for the other atoms, they become entangled. These entangled multi-qubit logic gates pose greater implementation challenges compared to single superposition logic gates because they involve forcing qubits to interact with each other, potentially leading to interaction with the environment and subsequent loss of information.

How do we program a quantum computer?

Quantum programming involves designing quantum logic gates and then applying algorithms to these gates. This parallels the concept of machine code in classical computing; it's the specific sequence of logic gates applied to our quantum bits to perform computations. These sequences of logic gates are called circuits.

When it comes to real world scenarios, we would never program using logic gates – similarly for classical computing we use programming languages, this is the same concept for quantum computing. Quantum computers are not coded directly on the quantum computer. Classical computers are already perfectly good to write and compile code for quantum computers.
The process from code to running such compiled code via a quantum processor is as follows:
Test quantum code for compiler issues -

1. Code turns into machine code
2. Machine code is read by the processor
3. Processor utilizes Logic gates

The running of compiled quantum code:

1. Qubits are acted on by Quantum Gates
2. Quantum Gates combine to make Quantum Circuits
3. Quantum Circuits are used by Quantum processors

Throughout this project I will be utilizing the Google colab research notebook to code the Deutsch Jozsa algorithm. I will be coding in Qiskit which is IBM's coding library that was written in Python for quantum computing. After completion of code – I will run the code which in turn will compile the code and translate it from machine language into a series of quantum logic gates which need to be applied to qubits. These compiled quantum logic gates are interpreted by my classical computer which triggers the actual quantum hardware to fire the specific laser pulses that correspond to the desired gates. The end result is measured and then we get our answer.

The Deutsch Problem overview

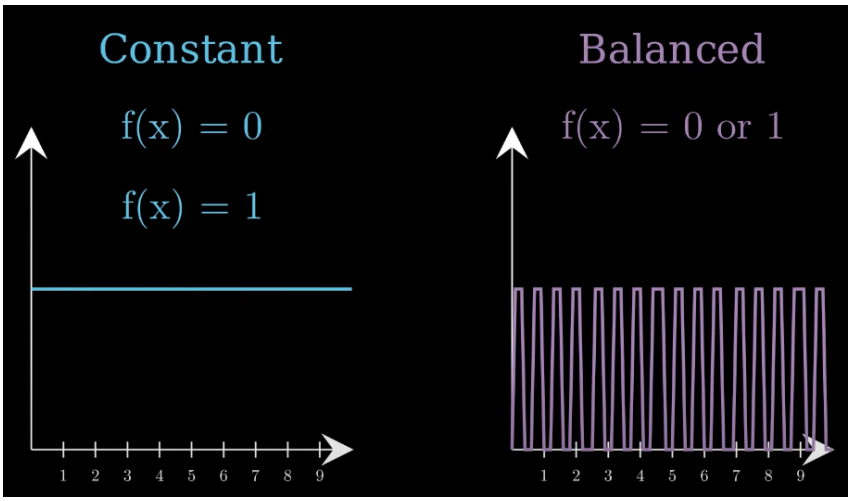
The Deutsch Jozsa algorithm is what I'll be coding. We want to decide if a random function given is either constant or balanced. Balanced functions return 0 half the time, and 1 half the time. A constant function returns 0 every time or 1 every time.

$f_A(0) = 0$ and $f_A(1) = 0$ → Constant

$f_B(0) = 0$ and $f_B(1) = 1$ → Balanced

$f_C(0) = 1$ and $f_C(1) = 0$ → Balanced

$f_D(0) = 1$ and $f_D(1) = 1$ → Constant



In a classical computer, it will take 3 iterations of the functions to tell me if the function is either balanced or constant, this is because we will need to evaluate either f(0) = f(1) (Constant), and F(0) != F(1) (Balanced). Three iterations of the function are chosen based on the worst case scenario – for example, if the first 2 iterations measure at balanced, or if both iterations measure the same output that being both 1, or both 0 (constant). A third measure will have to be conducted to ensure 100% evaluation.

When implementing the Deutsch Jozsa algorithm on a quantum computer – I will only need to run a single iteration of the function – I can create an equal superposition of all the possible states of our quantum computer and then run the function on them. If the output is 0, the function is considered constant, if the output is 1 then the function is considered balanced. This overall, gives me a factor of 2 speedup via the use of quantum computing in comparison to classical computing.

Considering this computation thus speed increase of quantum computing, there's a catch. Quantum mechanics operate probabilistically. When a quantum computer measures a quantum superposition state, it yields one of the possible states of the system—either 0 or 1, exclusively. However, due to quantum computers' inherent noise, there's a chance of obtaining an incorrect measurement, even if the Deutsch-Jozsa algorithm perfectly collapses the state into either 0 or 1. Currently, there's a 1% probability of obtaining the wrong state via the Deutsch-Jozsa algorithm in quantum computing. This 1% failure rate can be mitigated by running the quantum computer multiple times.

While this might seem counterintuitive—since a classical computer only requires 2 iterations while the quantum computer needs 1 iteration with several repeats—the quantum approach remains superior. Although the quantum algorithm implemented in this project utilizes only 2 qubits, making it feasible even for a classical computer, other quantum algorithms involving more qubits will exhibit a significant speed increase compared to their classical computing counterparts.

$$K(n) = 2^{n-1} + 1 \approx O(2^n)$$

The above describes that for a function that takes n input bits, we would have to run our function about 2 to the N times in the worst case to tell whether it’s a constant or balanced function on a classical computer. This process gets very long once n bits are 1000, or even a million. When the number of input bits is small, the classical solution is great whereas our noisy quantum computer needs thousands of runs for averages but as N starts to increase the classical computing operations exponentially increases thus very slow operations occur, but quantum computing has very minimal increase.

The Deutsch Jozsa Algorithm implementation

This section describes how I implemented the Deutsch Jozsa quantum algorithm utilizing cloud resources, including an IBM Cloud Quantum Computer.

Firstly, I navigated to https://colab.research.google.com and created a new notebook. The following code snippets include the following:

- Installation and import of required libraries
- Select the function I will pass to the algorithm (Constant or balanced function)
- Initialize the Deutsch Jozsa algorithm
- Backend setup – connection to the IBM Quantum computer
- Run the algorithm and plot measurement.

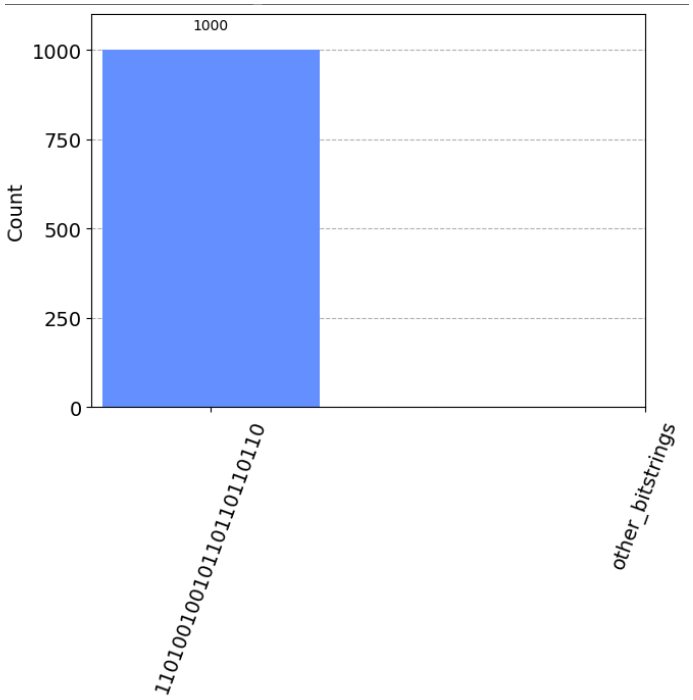
Please refer to the embedded comments for more information.

```
#install the libraries and import them
!pip install pylatexenc
!pip install qiskit
!pip install qiskit-aer
!pip install qiskit_ibm_runtime

# Import required libraries
import qiskit
from qiskit import QuantumCircuit, transpile
from qiskit_ibm_runtime import QiskitRuntimeService
from qiskit.circuit import Parameter
from qiskit_ibm_runtime import EstimatorV2 as Estimator
from qiskit.transpiler.preset_passmanagers import
generate_preset_pass_manager
from qiskit.quantum_info import Pauli, SparsePauliOp
import numpy as np
import matplotlib.pyplot as plt

# Print Qiskit version
print(qiskit.__version__)

# Additional imports for visualization
from qiskit import ClassicalRegister, QuantumRegister
from qiskit.visualization import plot_histogram

import numpy as np

# Generate random values for oracle type and value
oracleType = np.random.randint(2)
oracleValue = np.random.randint(2)
n = 23  # Number of qubits

# Determine the type of oracle and print the result
```

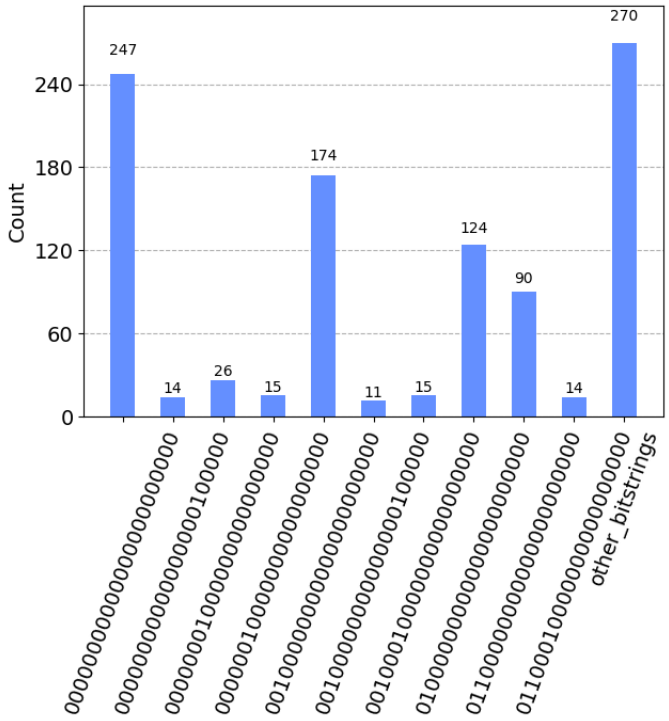
```
if oracleType == 0:
    print(f'The oracle returns a constant value: {oracleValue}')
else:
    print("The oracle returns a balanced function")
    a = np.random.randint(1, 2**n)  # A parameter used later in the
balanced function

from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

# Initialize the algorithm (Circuit)
n = 23  # Number of qubits
circuitName = 'DeutschJozsa'
qr = QuantumRegister(n + 1)
cr = ClassicalRegister(n)
djCircuit = QuantumCircuit(qr, cr)

# The algorithm
# Step 1
djCircuit.x(qr[n])  # Apply a NOT gate to the last qubit to set it to
the state |1>

# Step 2 - The Hadamard Gates
for i in range(n + 1):
    djCircuit.h(qr[i])
djCircuit.barrier()

# Step 3 - Apply the function
if oracleType == 0:
    if oracleValue == 1:
        djCircuit.x(qr[n])  # Apply NOT gate if the function is
constant at 1
else:
    for i in range(n):
        if a & (1 << i):
            djCircuit.cx(qr[i], qr[n])  # Apply CNOT gate if the
function is balanced

djCircuit.barrier()

# Step 4 - Apply Hadamard gates to the first n qubits again
for i in range(n):
    djCircuit.h(qr[i])
djCircuit.barrier()

# Measurement
for i in range(n):
    djCircuit.measure(qr[i], cr[i])  # Measure each qubit to the
corresponding classical bit
```

```
Set up the backend:

service = QiskitRuntimeService(channel='ibm_quantum',token='HIDDEN’)

# Get the quantum backends
backend_sim = service.get_backend('ibmq_qasm_simulator')
backend_qc = service.get_backend('ibm_brisbane')
backend_qc_notbusy = service.least_busy(operational=True,
simulator=False)

# Print the names of the backends
print(f"Simulator backend: {backend_sim.name}")
print(f"Quantum computer backend: {backend_qc.name}")
print(f"Least busy quantum computer backend:
{backend_qc_notbusy.name}")

# Select the backend
# backend = backend_sim
backend = backend_qc_notbusy  # The actual quantum computer
print(f"Backend: {backend.name}")

# Transpile the circuit for the selected backend
transpiled_circuit = transpile(djCircuit, backend=backend)

# Define the number of shots
shots = 1000

# Execute the circuit on the backend
job = backend.run(transpiled_circuit, shots=shots)
results = job.result()
answer = results.get_counts()

# Set a threshold for significant measurements
threshold = int(0.01 * shots)

# Filter the results to exclude measurements below the threshold
filteredAnswer = {k: v for k, v in answer.items() if v >= threshold}
removedCounts = np.sum([v for k, v in answer.items() if v < threshold])
filteredAnswer['other_bitstrings'] = removedCounts

# Plot the histogram of the filtered results
plot_histogram(filteredAnswer)
```

I firstly ran the algorithm via the quantum simulator offered by IBM. The result (shown below) measured as a balanced state. As mentioned previously, due to resources (number of qubits) such algorithm can be run via the quantum simulator.



Next I ran the algorithm against real quantum hardware via IBM's Quantum platform. The result (shown below) measured as mostly constant (247) with other_bitstrings holding a count of 270. This shines a light on how noisy such quantum systems are, and the importance of running multiple iterations to ensure results are accurate. Although such results point more towards a balanced state, there is still a big chance of the function actually being constant.



Conclusion

Much of the ongoing research in the quantum domain revolves around reducing the error rates of qubits to enhance the performance of algorithms such as the Deutsch-Josza algorithm. The issue I demonstrated through the measurements from the real quantum system primarily effects our ability to scale up quantum systems by adding more qubits. Presently, increasing the number of qubits on a chip leads to increased interference, thus increasing the noise problem.

Bibliography

Computer, I. t. (n.d.). *Deutsch Algorithm*. Retrieved from YouTube: https://www.youtube.com/watch?v=QQS5iI0LH1I&t

David Collins, K. W. (n.d.). *Deutsch-Jozsa algorithm as a test of quantum computation.*

Lab, L. (n.d.). *How To Code A Quantum Computer*. Retrieved from YouTube: https://www.youtube.com/watch?v=JOJ5zihcd6Q

Lab, L. (n.d.). *I Coded a Real Quantum Computer*. Retrieved from YouTube: https://www.youtube.com/watch?v=Gl9GVLW451s

Stephan Gulde*, M. R.-K. (n.d.). *Implementation of the Deutsch– Jozsa algorithm on an ion-trap quantum computer.*