Quantum Visualizer

Nick Coffman

May 5, 2024

Abstract

With the onward march of quantum computing technology, many of us working within the domain of computer science are scrambling to intuitively grasp the concepts and thus become more fully capable of appreciating the relatively nascent technology as it seems to become more and more ubiquitous.

This project is an attempt to address the challenge of intuitively grasping the concepts underneath the hood of this impressive new technology.

The novel quantum concepts treated below include ***superposition, amplitude amplification,*** and ***interference***, which, in composite, figure significantly into one of the most famous and impressive of quantum algorithms, ***Grover's algorithm***.

1 Introduction

Some invaluable sources that have come to my aid toward the completion of this project include:

- Python
- IBM's Qiskit (v1.0.2) *An open-source quantum computing framework*
- Jupyter Notebook
- A virtual environment
- TeXShop
- A 2020 MacBook Pro running OS Sonoma 14.4.1

2 Quantum Concepts

2.1 Superposition

In classical computing, each bit represents a single state, and computations are performed on these definite states. This limitation requires that operations be performed sequentially, limiting the speed and efficiency of data processing. In contrast, superposition allows a quantum computer to process a multitude of possibilities simultaneously. Quantum superposition is a fundamental principle of quantum mechanics that allows quantum systems, such as qubits, to exist in multiple states simultaneously. Unlike classical bits, which are definitively zero or one, qubits can be both zero and one at the same time. This property is not merely a theoretical curiosity; it has practical applications that can revolutionize computing.

The figures below illustrate how the state of a single qubit can vary significantly, represented here by its position on the Bloch sphere, which symbolizes all the possible states in which a single qubit can exist.

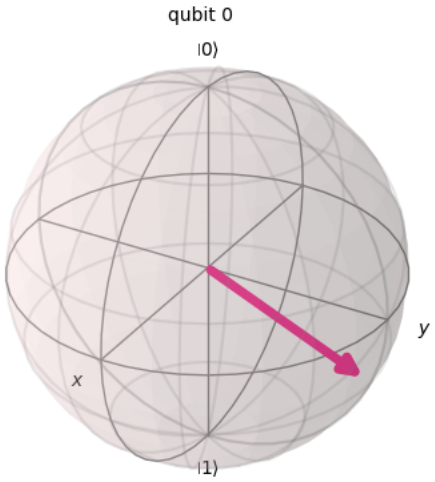


Figure 1: Bloch sphere representation of a qubit in a superposition state.

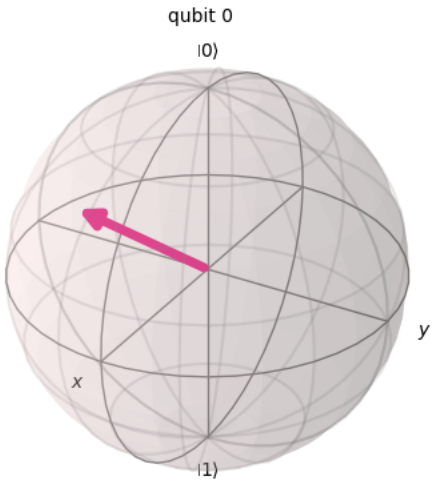


Figure 2: Another state of a qubit demonstrating the principle of superposition.

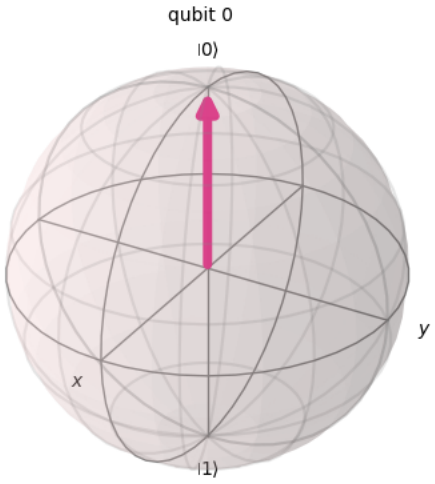


Figure 3: Final measured state of a qubit assuming the value of '0' and thus abandoning the state of superposition.

2.2 What is a Quantum Circuit?

Before proceeding, it should be pointed out that a quantum circuit is more or less analogous to a classical electronic circuit in that it involves connecting different components together to achieve a specific outcome. However, in the case of quantum computing, these components aren't resistors, capacitors, or wires, but qubits and their own quantum gates.

One such quantum gate that figures quite often into this demonstration for its importance is the *Hadamard gate*, which, when applied to a qubit, puts it into a state of superposition, where it will inhabit a state of both 0 and 1 values simultaneously. Below is a visual representation of a Hadamard gate being applied to a qubit:

Example 1: Single Qubit Hadamard Gate

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(1)  # Define a quantum circuit with 1 qubit.
qc.h(0)  # Apply a Hadamard gate to the qubit.
```

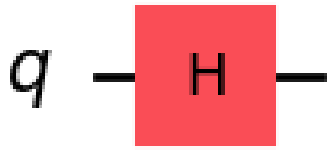


Figure 4: Quantum circuit with a single Hadamard gate.

Though a single qubit taken alone can prove quite powerful, for ease of demonstration, let us consider three Hadamard gates applying superposition to three separate qubits. While a classical computer with three bits can be in one of eight possible states at any given time, a quantum computer's three qubits can represent all eight possible states (000 to 111) simultaneously thanks to superposition. This capability exponentially increases with each additional qubit, leading to potential computing power that far exceeds that of classical computers.

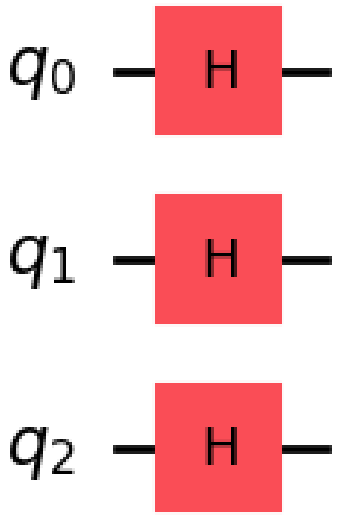


Figure 5: Quantum circuit applying Hadamard gates to two qubits.

2.3 Interference

Many of us who have ever been downwind of any discussion of the more famous concepts in quantum mechanics have likely heard of *interference*, likely by way of its most famous and bizarre demonstration: the double-slit experiment. This experiment demonstrated the less-than-intuitive idea that particles like photons and electrons behave both like particles and like waves. When they do behave like waves, they can *interfere* with each other just as waves in the ocean can. When these waves do interfere with one another, their interference can be classified as either *constructive* or *destructive*. Both of these classifications play a crucial role in the following section on *amplitude amplification*.

2.4 Amplitude Amplification

The final concept treated here is that of *amplitude amplification*. Amplitude amplification not only makes use of the previous two concepts, *superposition* and *interference*, but also is arguably the concept from which *Grover's algorithm* draws most of its computational strength.

You may recall from above the application of the *Hadamard gate*, which, when applied to one or more qubits, sets them into superposition. You may also recall that every additional qubit represents an exponential increase of computational power for our program.

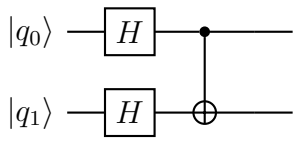
It may be useful to point out here that Grover's algorithm is ultimately a quantum search algorithm designed to be set upon unstructured datasets. This algorithm channels *interference*, both *constructive* and *destructive*, to manipulate the amplitudes of correct and incorrect states. That is to say, over multiple iterations, the sought-after datum (or correct state) becomes increasingly likely to be found while the rest of the unstructured dataset (or incorrect state) becomes increasingly likely to be properly ignored. This iterative process is what is referred to by *amplitude amplification*.

3 Grover's algorithm

3.1 The Oracle

Given the previous treatment of *amplitude amplification*, one may wonder how precisely Grover's algorithm knows which states are *correct* and which are *incorrect*. This is the role of the tool referred to as the ***Oracle***. The Oracle marks the correct solutions to the search problem by inverting the signs of the amplitude of the state or states that correspond to the correct answers. In other words, it identifies which items in a potentially vast and unstructured dataset are the ones we're looking for.

Visualized here is a *phase inversion* on the solution states, as performed by the Oracle. A phase inversion, in essence, is a controlled operation that changes how qubits behave when they're combined or interacted with.



Below is an implementation of the Oracle using Qiskit:

```
def grover_oracle(marked_states, num_qubits):
    # Initialize a new quantum circuit with a specified number of qubits.
    qc = QuantumCircuit(num_qubits)

    # Iterate over each target state in the list of marked states.
    for target in marked_states:
        # Reverse the target state string to match Qiskit's qubit ordering.
        rev_target = target[::-1]
        # Identify indices of the qubits that should be '0' in the marked state.
        zero_inds = [i for i, bit in enumerate(rev_target) if bit == '0']
        # Identify indices of the qubits that should be '1' (control qubits for the controlled-Z gate).
        control_qubits = [i for i, bit in enumerate(rev_target) if bit == '1']
        # Apply Pauli-X gates to qubits that need to be '0' in the oracle to prepare them for controlled operations.
        qc.x(zero_inds)

        # If there are control qubits, set up a multi-controlled Z gate.
        if control_qubits:
            qc.append(MCMT(gate='z', num_ctrl_qubits=len(control_qubits), num_target_qubits=1),
                      control_qubits + [zero_inds[0]])

        # Reapply Pauli-X gates to revert the states of zero indices to their original state.
        qc.x(zero_inds)


    return qc
```

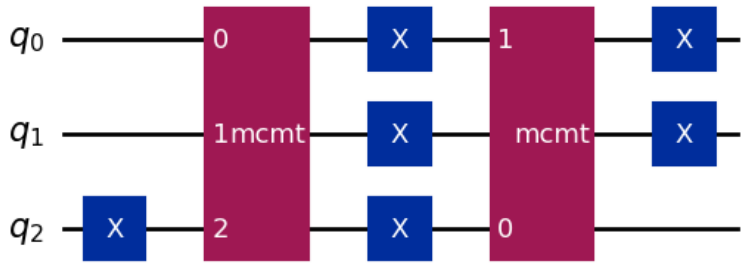
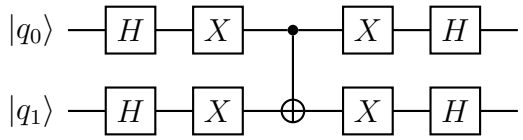


Figure 6: Pauli-X (NOT) gates (*here represented in blue*) and Multi-Controlled Multi-Target gates (*here represented in burgundy*) perform their markings on qubits corresponding to the sought-after data.

3.2 The Diffusion Operator

Once the *Oracle* has marked the correct states, we're free to employ the next tool within Grover's algorithm, the ***Diffusion Operator***. Its role is ultimately that of making use of *amplitude amplification* by amplifying the probability amplitudes of those states that were marked by the Oracle, using the mechanism of *interference*. This tool works by first applying a transformation that inverts all amplitudes about the average amplitude, augmenting the marked states' amplitudes (which had their signs flipped by the Oracle) and diminishing the unmarked states.

The diffusion operator is applied to spread the amplitude changes from the oracle across the entire superposition:



Below is an implementation of the Oracle and the Diffusion Operator using Qiskit:

```python
import math
from qiskit import QuantumCircuit
from qiskit.visualization import circuit_drawer

# Oracle to flip the phase of the |111> state now
def oracle(qc, target_state):
    # Apply controlled-Z gate across all qubits (assuming |111> is the target state)
    qc.cz(qubits[0], qubits[1])  # First CZ between the first and second qubits
    qc.cz(qubits[1], qubits[2])  # Second CZ between the second and third qubits

def diffusion_operator(qc, qubits):
    qc.h(qubits)  # Apply Hadamard gates to all qubits
    qc.x(qubits)  # Apply Pauli-X gates to all qubits

    # Apply a multi-controlled Z gate, which needs to apply a phase flip if all qubits are |1>
    # Using the third qubit as the target of the MCZ which is synthesized by a multi-controlled NOT
    qc.h(qubits[-1])  # Hadamard on the last qubit to transform Z to X basis
    qc.mcx(qubits[:-1], qubits[-1])  # Multi-controlled NOT with the last qubit as target
    qc.h(qubits[-1])  # Hadamard on the last qubit to transform back
    qc.x(qubits)  # Apply Pauli-X gates to all qubits
    qc.h(qubits)  # Apply Hadamard gates to all qubits
# Update qubits list to include three qubits
qubits = [0, 1, 2]  # Define qubits
# Initialize Quantum Circuit with 3 qubits
qc = QuantumCircuit(len(qubits))
# Apply Hadamard gates to all qubits
for q in qubits:
    qc.h(q)
oracle(qc, qubits)  # Apply the Oracle
diffusion_operator(qc, qubits)  # Apply the Diffusion Operator
```

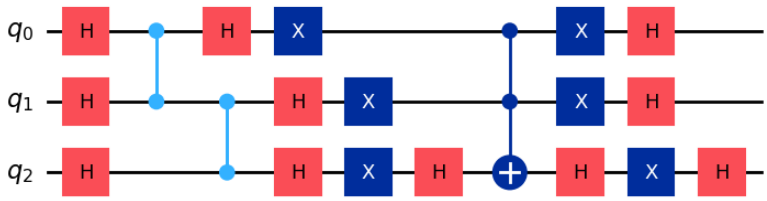


Figure 7: Here, we see several applications of Hadamard and Pauli-X gates, as well as a multi-controlled NOT gate (*here represented by the large blue dot with a "+"*), which implements the phase flip conditionally based on the state of all other qubits, as well as connections between qubits that are involved in two-qubit gates where both qubits equally influence the operation (*here represented by the blue connecting lines*).

3.3 Grover's Algorithm

At last, all composite parts are prepared to perform Grover's algorithm.

Taken step-by-step:

- 1. The program begins by applying Hadamard gates to all qubits, putting every qubit into superposition. This allows the algorithm to check all possible answers at once .
- 2. The Oracle then marks the correct solutions. This is done by changing the phase of those states, making them distinct from all others in a way that is meaningful but not directly observable.
- 3. The Grover Operator (The name given to the combination of the Oracle and the Diffusion Operator) then helps to amplify the probability of the marked states, which increases the likelihood that the that the correct answers are more likely to be observed when we finally measure the state of the qubits.
- 4. After applying the Grover Operator a calculated number of times (to increase the likelihood of finding the correct answer), all qubits are measured. The result of these measurements tells us which state is the most likely solution to the problem.

- Misc: The AerSimulator is used to simulate how a real quantum computer would perform this circuit.
- Misc: The circuit is transpiled (optimized and translated into a form compatible with the simulator) and run. The results from the simulation show how often each possible state (combination of qubits) was measured. In Grover's algorithm, the marked states should ideally be measured more frequently than others if the algorithm has worked correctly.

```
1
2  from qiskit import QuantumCircuit, transpile
3  from qiskit_aer import AerSimulator
4  from qiskit.visualization import plot_histogram,
       circuit_drawer
5  from qiskit.circuit.library import GroverOperator, MCMT
6  import math
7
8  def grover_oracle(marked_states, num_qubits):
9      qc = QuantumCircuit(num_qubits)
10     for target in marked_states:
11         rev_target = target[::-1]  # Reverse the bit-
       string to match Qiskit's qubit ordering
12         zero_inds = [i for i, bit in enumerate(
       rev_target) if bit == '0']
13         control_qubits = [i for i, bit in enumerate(
       rev_target) if bit == '1']
14         qc.x(zero_inds)  # Apply X gates to qubits
       corresponding to '0' in the target state
15         if control_qubits:
16             qc.append(MCMT(gate='z', num_ctrl_qubits=len
       (control_qubits), num_target_qubits=1),
       control_qubits + [zero_inds[0]])
17         qc.x(zero_inds)    # Reapply X gates to revert
       the initial state
18     return qc
19
20 # Define parameters for Grover's algorithm
21 num_qubits = 3
22 marked_states = ["011", "100"]
23 oracle_circuit = grover_oracle(marked_states, num_qubits
       )
24 grover_op = GroverOperator(oracle_circuit)
25
26 # Calculate the optimal number of iterations
27 optimal_num_iterations = math.floor(math.pi / (4 * math.
       asin(math.sqrt(len(marked_states) / 2**num_qubits))))
28
29 # Prepare the quantum circuit
```

```
qc = QuantumCircuit(num_qubits)
qc.h(range(num_qubits))  # Create an even superposition of all basis states
print("Initial Superposition State:")
display(qc.draw(output='mpl', style='iqp'))

# Applying Grover's Operator
qc.append(grover_op.power(optimal_num_iterations), range(num_qubits))

# Measurement
qc.measure_all()

# Set up the simulator
simulator = AerSimulator()

# Transpile the circuit for the simulator
transpiled_circuit = transpile(qc, simulator)

# Run the transpiled circuit and collect results
job = simulator.run(transpiled_circuit, shots=1024)
result = job.result()
```

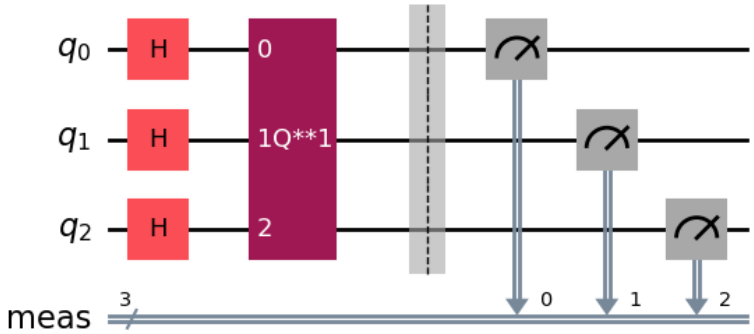


Figure 8: Final circuit with measurements in Grover's Algorithm. Included here are the familiar Hadamard gates as well as the Grover Operator *(represented here in burgundy)*, as well as the final measurement of each of the qubits *(represented here in gray)*.

3.4 Results and Interpretation

The following histogram generated by the Jupyter cell demonstrates the marked states being amplified as intended.

- X-axis (States): The x-axis lists all possible outcomes for the qubits being measured.
- Y-axis (Counts): The y-axis displays the number of times each state was measured, typically normalized by the total number of shots to show probabilities. If the algorithm is functioning as it should, the correct solutions (marked states) should appear more frequently.

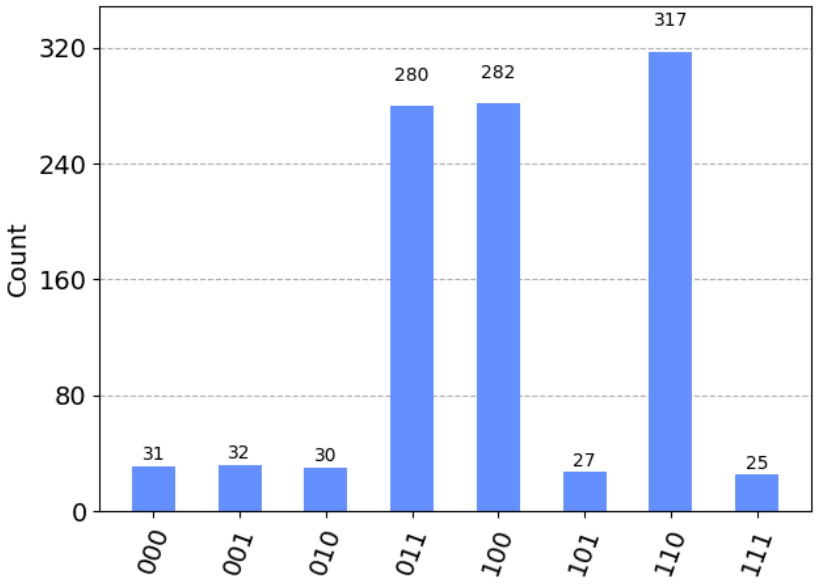


Figure 9: Histogram of the output probabilities.

4 Conclusion

This project has once again been an attempt to intuitively grasp some core quantum computing concepts such as superposition, interference, and amplitude amplification, as demonstrated through Grover's algorithm. Grasping these quantum concepts and their algorithmic implementations intuitively could facilitate seismic advancements in computer science. Quantum computing holds the potential to solve problems that are currently infeasible for classical computers. In the near-to-long term, the intuitive grasp of these concepts by computer scientists will be crucial. It will enable the development of new algorithms, enhance existing quantum protocols, and ultimately lead to the widespread adoption of quantum computing in solving real-world problems. The exploration of quantum computing technologies not only pushes the boundaries of what is computationally possible but also expands the horizons of human knowledge and capability.