ENPH 344 Quantum Mechanics: Exploring Quantum Computing Utilizing Xanadu's PennyLane

Grover's Algorithm Compared to a Conventional Algorithm

Submitted by: Nathan Pacey

Student Number; 20153310

Email: 18njp@queensu.ca

Date: November 26, 2022

1 Background Information

1.0 Quantum Computing Introduction

Quantum computing is a new type of computing that uses the principles of quantum mechanics, such as superposition, entanglement, and interference, to enable powerful calculations and processes that are not possible with traditional computing methods. Quantum computing is based on the idea that information can be represented as qubits (quantum bits) instead of the traditional binary digits (bits) used in classical computing. The basic principle differentiating a qubit to a regular bit is that a qubit is constantly in a superposition state of both 1 and 0 [1]. Quantum computers solve complex problem by sending qubits through various quantum gates. Quantum computing has the potential to be used in a variety of fields. One of the key advantages of quantum computing is its ability to efficiently solve certain problems that are difficult or impossible for classical computers to solve. For example, quantum computers can quickly factorize large numbers, a problem that is central to many forms of modern cryptography. Another, advantage of quantum computing is its ability to simulate complex quantum systems. This allows scientists to study the behavior of molecules, materials, and other systems that are difficult to study with classical computers. Research using quantum computing could lead to new breakthroughs in fields such as chemistry, materials science, and drug discovery [2].

In this report I am going to explain how to apply some fundamental properties of quantum computing using Xanadu's Python library PennyLane including how to improve a linear search algorithm using Grover's Algorithm [3].

1.1 PennyLane Introduction

PennyLane is an open-source Python library for quantum machine learning and quantum computing. It is designed to enable researchers and developers to build, train, and deploy quantum computing models, and to integrate them with classical computing frameworks. One of the key features of PennyLane is its support for a wide range of quantum computing hardware and software platforms, including different types of quantum computers, simulators, and emulators [3]. This allows users to easily experiment with different quantum computing technologies, and to compare their performance on various tasks [4].

1.2 Qubits and Quantum Computing States

Since all qubits can either be represented by the '1' state or the '0' state the overall probability of a specific qubit can be represented in an orthonormal basis where the multiples of each state represent the amplitude of the qubit in the state's axis as seen below in Equation 1.

$$|\psi\rangle = A|1\rangle + B|0\rangle \tag{1}$$

Since the amplitude of the qubit is represented by a complex number one must take the summation of the normalized amplitudes to get the total probability of 1 as seen in Equation 2.

$$|A|^2 + |B|^2 = 1 \tag{2}$$

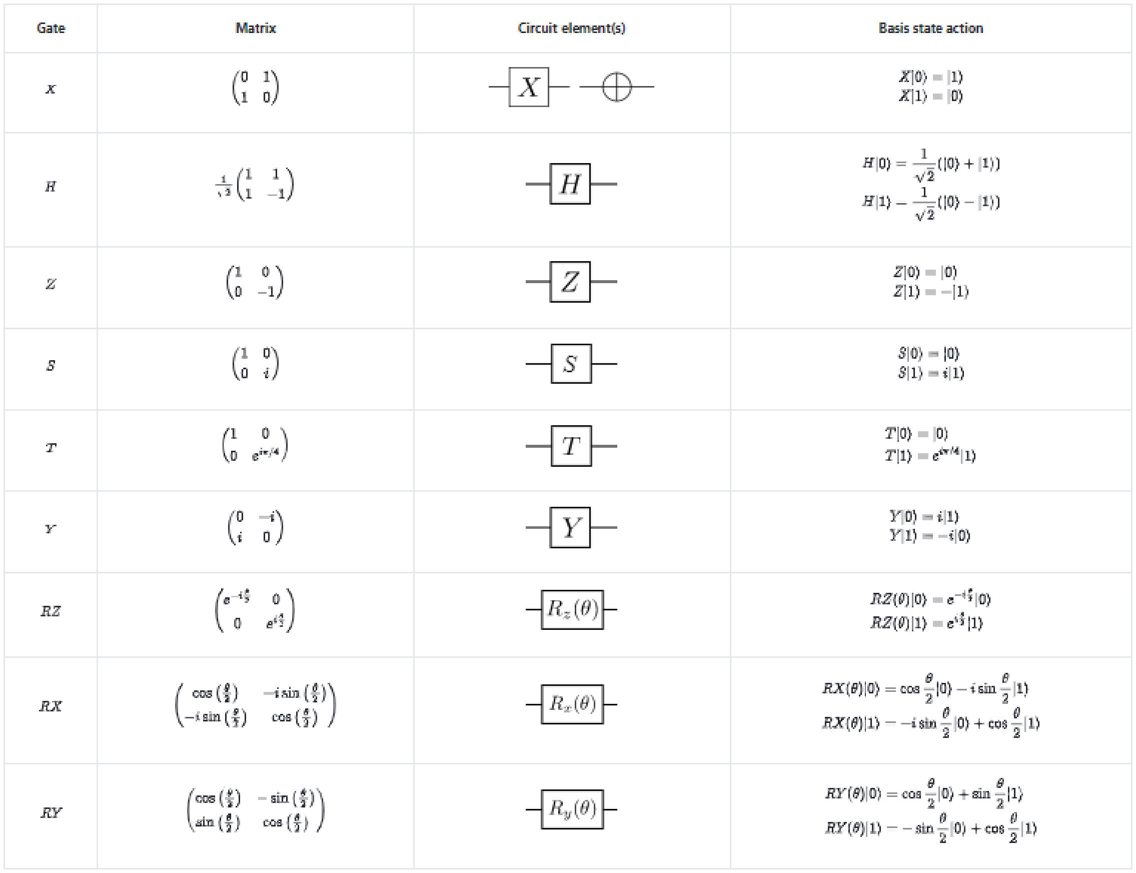
These properties allow qubits to manipulated and observed in a similar manner to sub-atomic particles [4].

1.3 Quantum Gates

Quantum computing circuit gates are a fundamental concept in the field of quantum computing. They are used to manipulate and control the quantum state of quantum bits. Quantum circuit gates are analogous to classical computer gates, which are used to manipulate and control the state of classical bits in a classical computer. However, quantum circuit gates have some unique properties that distinguish them from classical gates, due to the principles of quantum mechanics. One key difference between quantum and classical circuit gates is that quantum gates can perform operations on multiple qubits simultaneously, whereas classical gates can only operate on one bit at a time. This property, known as quantum parallelism, allows quantum computers to perform certain tasks much faster than classical computers. Another important feature of quantum circuit gates is that they can be used to create entanglement, a phenomenon in which the state of one qubit becomes correlated with the state of another qubit, even when they are separated by large distances. Entanglement is a key resource in many quantum algorithms and protocols and is one of the key features that makes quantum computers so powerful [4].

There are many different types of quantum circuit gates, including single-qubit gates and multi-qubit gates. Single-qubit gates operate on a single qubit, while multi-qubit gates operate on two or more qubits. Some common examples of quantum circuit gates include the Pauli gates, the Hadamard gate, and the CNOT gate. Below is a table of the quantum gates available using the Penny Lane library as well as their respective action on the state of a qubit [4].

Table 1. Quantum Circuit Gates and respective operations from the Xanadu Codebook [4].



Gate	Matrix	Circuit element(s)	Basis state action
X	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\boxed{X}$ $\oplus$	$X\|0\rangle = \|1\rangle$ $X\|1\rangle = \|0\rangle$
H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\boxed{H}$	$H\|0\rangle = \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$ $H\|1\rangle = \frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$
Z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\boxed{Z}$	$Z\|0\rangle = \|0\rangle$ $Z\|1\rangle = -\|1\rangle$
S	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	$\boxed{S}$	$S\|0\rangle = \|0\rangle$ $S\|1\rangle = i\|1\rangle$
T	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$	$\boxed{T}$	$T\|0\rangle = \|0\rangle$ $T\|1\rangle = e^{i\pi/4}\|1\rangle$
Y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\boxed{Y}$	$Y\|0\rangle = i\|1\rangle$ $Y\|1\rangle = -i\|0\rangle$
RZ	$\begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$	$\boxed{R_z(\theta)}$	$RZ(\theta)\|0\rangle = e^{-i\frac{\theta}{2}}\|0\rangle$ $RZ(\theta)\|1\rangle = e^{i\frac{\theta}{2}}\|1\rangle$
RX	$\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_x(\theta)}$	$RX(\theta)\|0\rangle = \cos\frac{\theta}{2}\|0\rangle - i\sin\frac{\theta}{2}\|1\rangle$ $RX(\theta)\|1\rangle = -i\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$
RY	$\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_y(\theta)}$	$RY(\theta)\|0\rangle = \cos\frac{\theta}{2}\|0\rangle + \sin\frac{\theta}{2}\|1\rangle$ $RY(\theta)\|1\rangle = -\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$

These gates are used in various combinations to perform a wide range of quantum algorithms and protocols. It should also be noted that there are certain combinations of gates that can span all possible sets of single qubit operations called universal gate sets. Sets include the Hadamard (H) and Phase Gate (T) gates or the Rotational gates (RX, RY and RZ) [4].

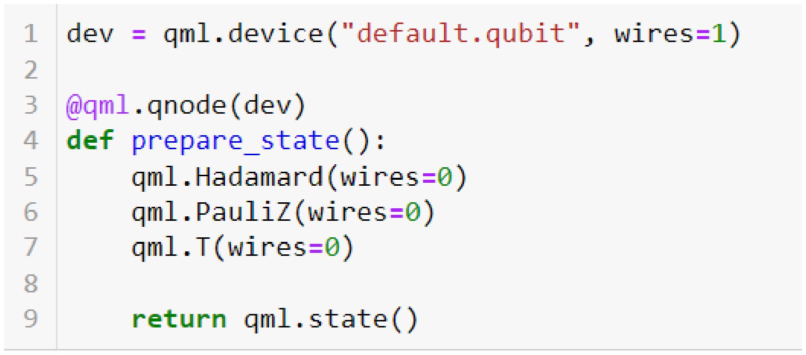
1.4 Quantum Circuits

To implement quantum computing algorithms such as Grover's algorithm a quantum circuit is used. Quantum circuits consist of a set of initial qubit states in a quantum register that get passed to different gates through a set of wires. To view the results of a quantum circuit measurements can be taken using tools like probes.

As an example, we can use the relations as shown in Table 1, we can make a single wire quantum circuit that represents Equation 3 as shown below. Note that this example was taken from Xanadu's Codercise I.8.1 [4].

$$|\psi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}e^{\frac{5}{4}\pi i} \tag{3}$$

Using the following Python code as shown below in Figure 1.



```python
dev = qml.device("default.qubit", wires=1)

@qml.qnode(dev)
def prepare_state():
    qml.Hadamard(wires=0)
    qml.PauliZ(wires=0)
    qml.T(wires=0)

    return qml.state()
```

Figure 1. Single wire quantum circuit implementation using PennyLane [4].

As shown by the code the first step is to create a quantum device 'dev' and define the number of wires needed. Next, we can declare a function that manipulates the quantum device by using the '@qml.qnode(dev)' tag above the function. Within the function we can declare the quantum gates applied to the device in an order that represents the algorithm we are trying to implement. Note that the single wire circuit implemented above can also be represented by the following diagram.

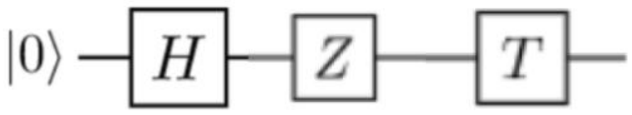


Figure 2. Circuit diagram of the single wire circuit implemented [4].

To implement more complex algorithms like Grover's algorithm we have to implement a circuit with multiple qubits. Codercise I.12.3 as shown in Figure 3 below shows how one can implement a multi-bit quantum circuit using PennyLane [4].

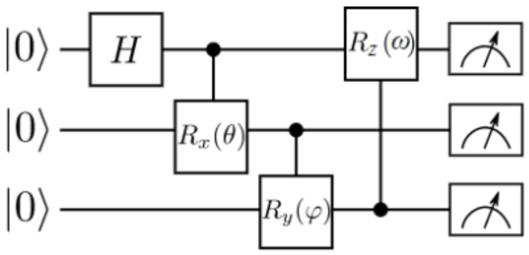
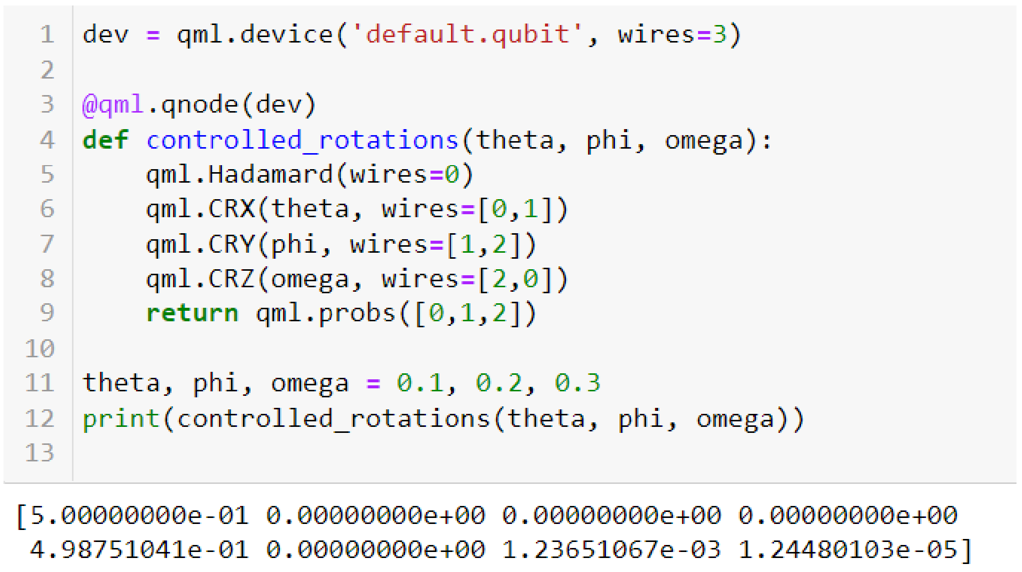


Figure 3. Example of a multi-wired circuit from the Xanadu Codebook [4].

Where the following Python code shown in Figure 4 is the implementation of this circuit.



```python
dev = qml.device('default.qubit', wires=3)

@qml.qnode(dev)
def controlled_rotations(theta, phi, omega):
    qml.Hadamard(wires=0)
    qml.CRX(theta, wires=[0,1])
    qml.CRY(phi, wires=[1,2])
    qml.CRZ(omega, wires=[2,0])
    return qml.probs([0,1,2])

theta, phi, omega = 0.1, 0.2, 0.3
print(controlled_rotations(theta, phi, omega))

```

```
[5.00000000e-01 0.00000000e+00 0.00000000e+00 0.00000000e+00
 4.98751041e-01 0.00000000e+00 1.23651067e-03 1.24480103e-05]
```

Figure 4. Implementation of the given quantum circuit.

It can bee seen that certain quantum gates can span over several wires by setting the gate's wire attribute to specific wires within the circuit. As well, to see the results of the quantum circuit when given initial quantum states one can return 'qml.probs' with the specific wire in which observation occurs specified.

2 Building Grover's Algorithm

2.1 Introduction to Grover's Algorithm

Grover's algorithm is a quantum algorithm that can be used to search an unstructured database or list in a time significantly faster than what is possible with classical algorithms. The algorithm works by using quantum superposition and quantum interference to amplify the probability of finding the desired element in the list. This is done by iteratively applying a unitary operator known as the Grover iterate to the initial state of the system, which consists of all possible elements in the list. The main advantage of Grover's algorithm is that it can find the desired element in a list of size N with only O($\sqrt{N}$) iterations, compared to O(N) iterations for classical algorithms. This means that it can search an unsorted list of one million elements in only 1,000 iterations, while a classical algorithm would require an average of 500,000 iterations [5].

2.2 Designing a Circuit to Apply Grover's Algorithm

For a circuit with a defined number of input states Grover's algorithm can be implemented using a Grover iterate, which is a sequence of gates that amplifies the probability of finding the desired element in the database. The Grover iterate is a unitary operator that is applied to the equal superposition state in order to amplify the probability of finding the desired element in the database. It is represented mathematically by Equation 4 [5].

$$G = -I_s * H^N * O_w * H^N \tag{4}$$

Where I_s is the identity operator acting on the search space, H^N is the Hadamard operator applied to all N qubits, O_w is the Oracle operator that marks the desired element, and H^N is the Hadamard operator applied again to all N qubits.

The effect of the Grover iterate is to create constructive and destructive interference between the different elements in the database, which amplifies the probability of finding the desired element. This is done by applying the Oracle operator O_w, which marks the desired element by flipping its phase, and then applying the Hadamard operator H^N, which creates interference between the marked and unmarked elements [4].

2.3 Uniform Superposition and the Hadamard Gate

The starting point for the algorithm is the quantum state known as the equal superposition state, which is a superposition of all possible states in the database. This state is represented mathematically by the Equation 5 [4].

$$|s\rangle = \frac{1}{\sqrt{N}} * (|0\rangle + |1\rangle + |2\rangle + \cdots + |i\rangle) \tag{5}$$

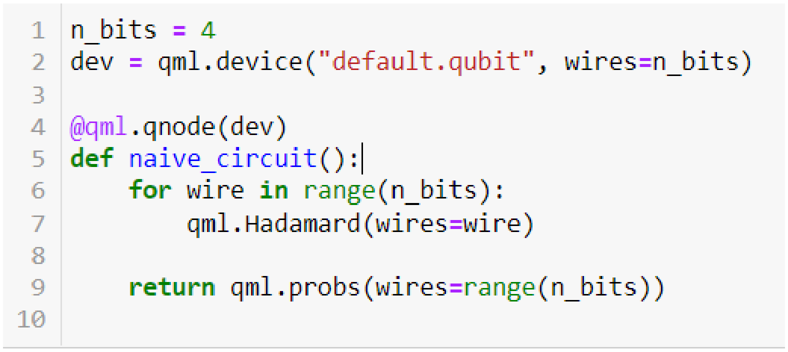
Where N is the number of elements in the database, and |i> represents the i-th element in the database. To initialize the qubits in the equal superposition the Hadamard gate is used, which is represented mathematically by Equation 6 [4].

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{6}$$

The Hadamard gate creates a superposition of the |0> and |1> states and can be applied to each qubit in the system to create the equal superposition state. The probability of observing each of the N states can be expressed by Equation 7.

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_i^N |i\rangle \tag{7}$$

Note that each state i can only below to the '0' or '1' state. When the summation is taken over all possible states the probability will approach 1. We can observe the effect of the Hadamard gate on a list of possible 4-bit states using Codercise A.1.1 as shown below in Figure 5 [4].



```
n_bits = 4
dev = qml.device("default.qubit", wires=n_bits)

@qml.qnode(dev)
def naive_circuit():
    for wire in range(n_bits):
        qml.Hadamard(wires=wire)

    return qml.probs(wires=range(n_bits))

```

Figure 5. Implementation of the Hadamard gate to create uniform superposition of the given states [4].

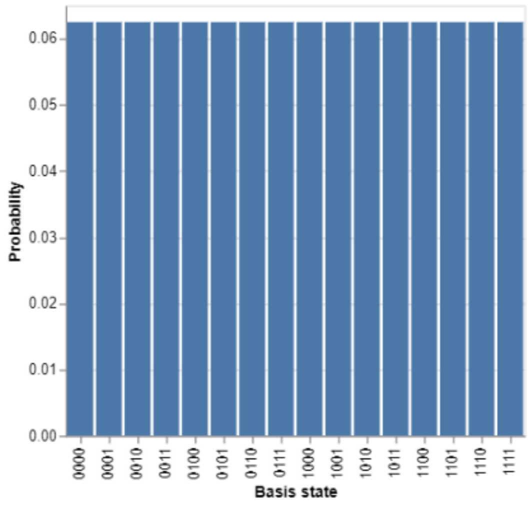


Figure 6. Probability output of the quantum circuit creating uniform super position of the given states [4].

From Figure 6 it can be seen that the probability of observing any of the given states after applying the Hadamard gate is equal, and the summation of the probabilities is 1.

2.4 Applying the Oracle Operator

The Oracle operator is represented mathematically by Equation 8 [4].

$$O_w = I - 2|s\rangle\langle s| \tag{8}$$

Where I is the identity operator and s is the target state in a given set of states. The Oracle operator is used in Grover's algorithm to flip the sign (positive to negative) of the amplitude of the desired element. Furthermore, after applying the Oracle to a superposition set the target state would have a negative amplitude with the same magnitude as the other states. This can be shown mathematically by Equation 9.

$$O_w = \begin{cases} -|i\rangle \; , i = s \\ \;\; |i\rangle \, , i \neq s \end{cases} \tag{9}$$

This phenomenon can be seen in the following code and plot as shown in Figure 7 and 8, taken from Codercise G.1.1. where the desired state is given by the array combo = [0,0,0,1] [4].

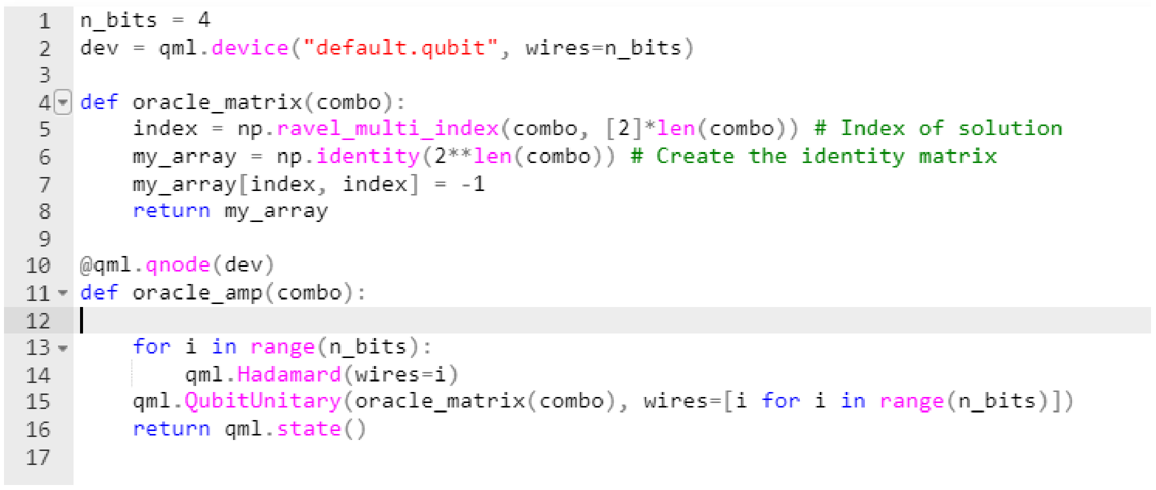

```python
n_bits = 4
dev = qml.device("default.qubit", wires=n_bits)

def oracle_matrix(combo):
    index = np.ravel_multi_index(combo, [2]*len(combo)) # Index of solution
    my_array = np.identity(2**len(combo)) # Create the identity matrix
    my_array[index, index] = -1
    return my_array

@qml.qnode(dev)
def oracle_amp(combo):

    for i in range(n_bits):
        qml.Hadamard(wires=i)
    qml.QubitUnitary(oracle_matrix(combo), wires=[i for i in range(n_bits)])
    return qml.state()
```

Figure 7. Applying the Oracle Operator to the Quantum Circuit of equal super position states [4].

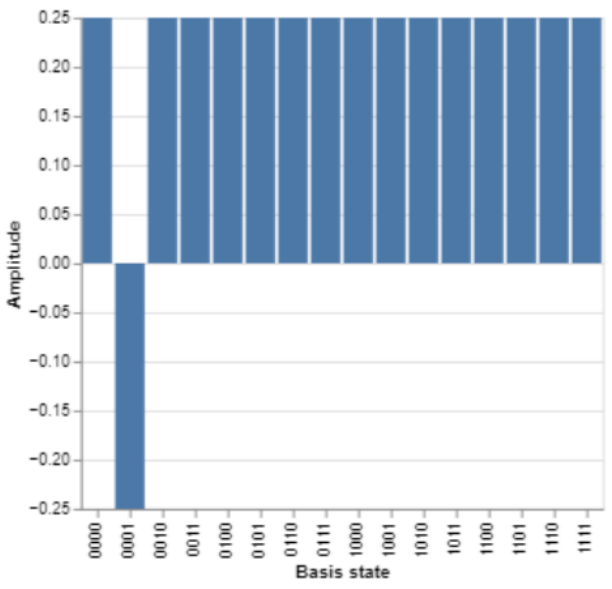


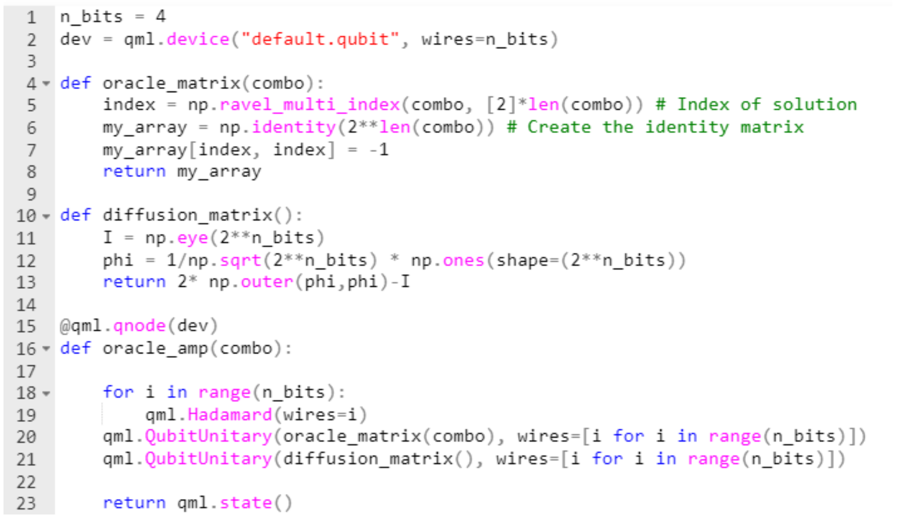
Figure 8. Resulting Amplitudes of the given quantum states after applying the Oracle operation to the Quantum Circuit [4].

2.5 Applying the Diffusion Operator

The final step of Grover's algorithm is applying the Diffusion Operator. The Diffusion operator takes two times the outer product of the probability vector (as given by Equation 7) of each given state and subtracts it from the identity matrix as shown below in Equation 10.

$$D = 2\,|\psi\rangle\langle\psi| - I \tag{10}$$

This has the effect of parsing the states as given after the Oracle, in which the desired state has a final amplitude that is much greater than all the other states. The implementation and effect of the Diffusion operator on the 4-bit system can be seen below in Figures 9 and 10.



```
n_bits = 4
dev = qml.device("default.qubit", wires=n_bits)

def oracle_matrix(combo):
    index = np.ravel_multi_index(combo, [2]*len(combo)) # Index of solution
    my_array = np.identity(2**len(combo)) # Create the identity matrix
    my_array[index, index] = -1
    return my_array

def diffusion_matrix():
    I = np.eye(2**n_bits)
    phi = 1/np.sqrt(2**n_bits) * np.ones(shape=(2**n_bits))
    return 2* np.outer(phi,phi)-I

@qml.qnode(dev)
def oracle_amp(combo):

    for i in range(n_bits):
        qml.Hadamard(wires=i)
    qml.QubitUnitary(oracle_matrix(combo), wires=[i for i in range(n_bits)])
    qml.QubitUnitary(diffusion_matrix(), wires=[i for i in range(n_bits)])

    return qml.state()
```

Figure 9. Applying the Diffusion operator to the Quantum Circuit [4].

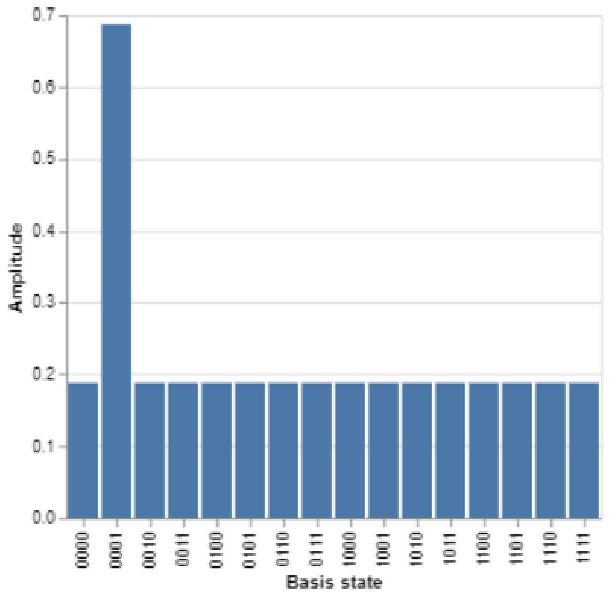


Figure 10. Resulting Amplitudes of the given states after applying Grover's Algorithm.

These results as shown in Figure 10, represent the implementation of Grover's Algorithm for a 4-bit combination of states in which 0001 was the target. To get an outcome in which the target amplitude approaches 1 the algorithm can be run multiple times until an optimal amplitude is found.

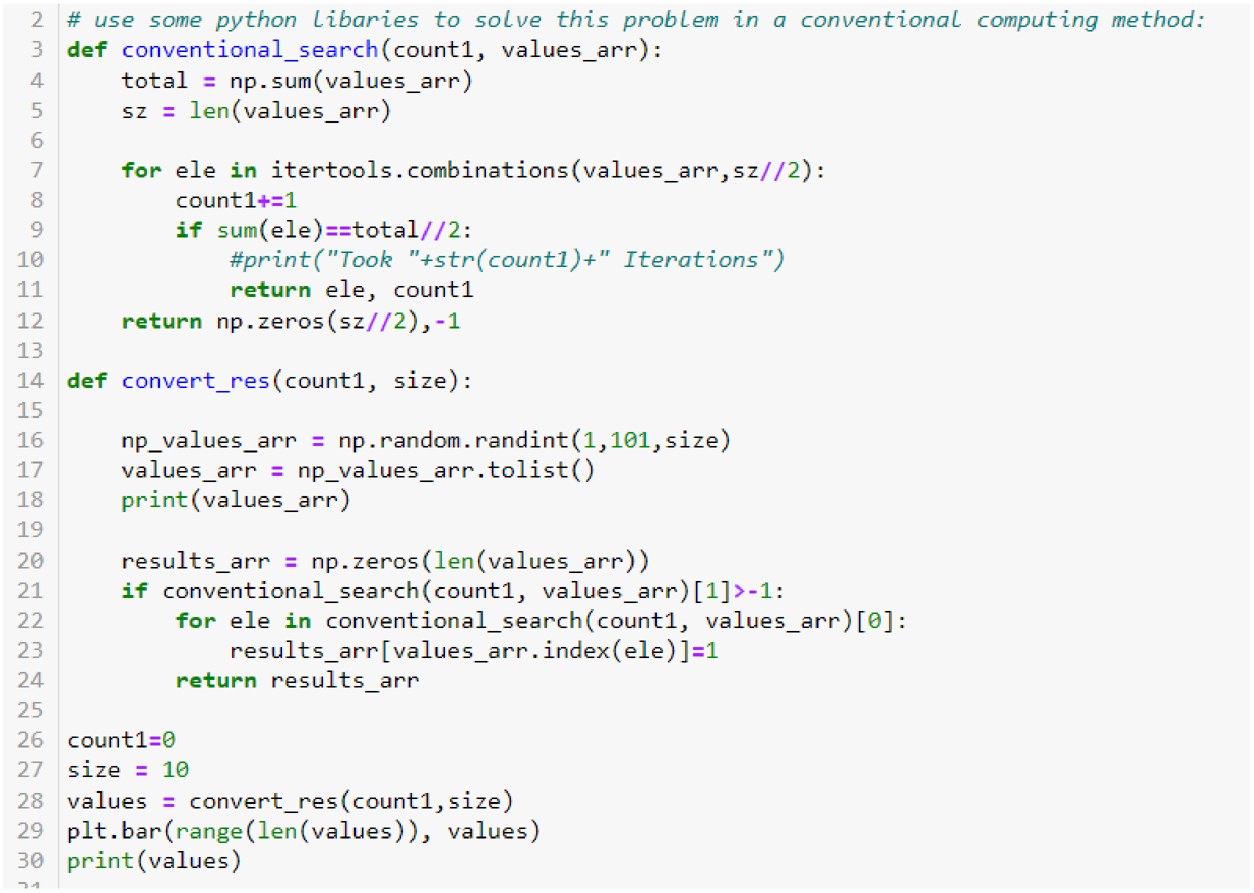
3. Application of Grover's Algorithm

To illustrate the benefit of Grover's algorithm when compared to traditional computational algorithms a multi-variable search was used. An example of a searching algorithm was tested in which the algorithm had to compare various combinations of a 10-element array containing random integers from 0 to 100 in order to identify two groupings from the array that had equal sums. The algorithm was also required to output a modified 10-element array where the sum of the indices of the 1's in the original array was equal to the sum of the indices of the 0's in the original array. The goal of this example was to illustrate that the number of iterations that the quantum algorithm takes would not only be linear and constant but also be significantly less than the conventional linear search [5].

For example, if the given array was [44, 86, 18, 50, 30, 86, 67, 84, 96, 73] a possible solution would be [1, 1, 1, 0, 0, 0, 0, 0, 1, 1] since 44+86+18+96+73=317 and 50+30+86+67+84=317. So, both groups of zeros and 1 have equal summations.

3.1 Conventional Searching Algorithm Implementation

To implement the conventional algorithm a linear search was used along with the combinations function from the itertools library as seen below in Figure 11.



```python
# use some python libaries to solve this problem in a conventional computing method:
def conventional_search(count1, values_arr):
    total = np.sum(values_arr)
    sz = len(values_arr)

    for ele in itertools.combinations(values_arr,sz//2):
        count1+=1
        if sum(ele)==total//2:
            #print("Took "+str(count1)+" Iterations")
            return ele, count1
    return np.zeros(sz//2),-1

def convert_res(count1, size):

    np_values_arr = np.random.randint(1,101,size)
    values_arr = np_values_arr.tolist()
    print(values_arr)

    results_arr = np.zeros(len(values_arr))
    if conventional_search(count1, values_arr)[1]>-1:
        for ele in conventional_search(count1, values_arr)[0]:
            results_arr[values_arr.index(ele)]=1
        return results_arr

count1=0
size = 10
values = convert_res(count1,size)
plt.bar(range(len(values)), values)
print(values)
```

Figure 11. Conventional computational searching algorithm for the given problem.

The results of the conventional searching algorithm given a random 10-element array can be seen in Figure 12 below.

```
[39, 55, 21, 12, 82, 63, 5, 87, 10, 65]
[1. 0. 1. 1. 1. 0. 0. 0. 0. 1.]
```

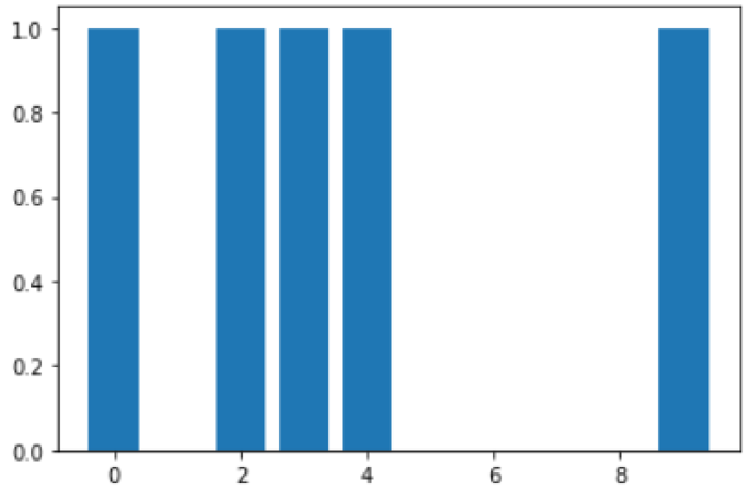
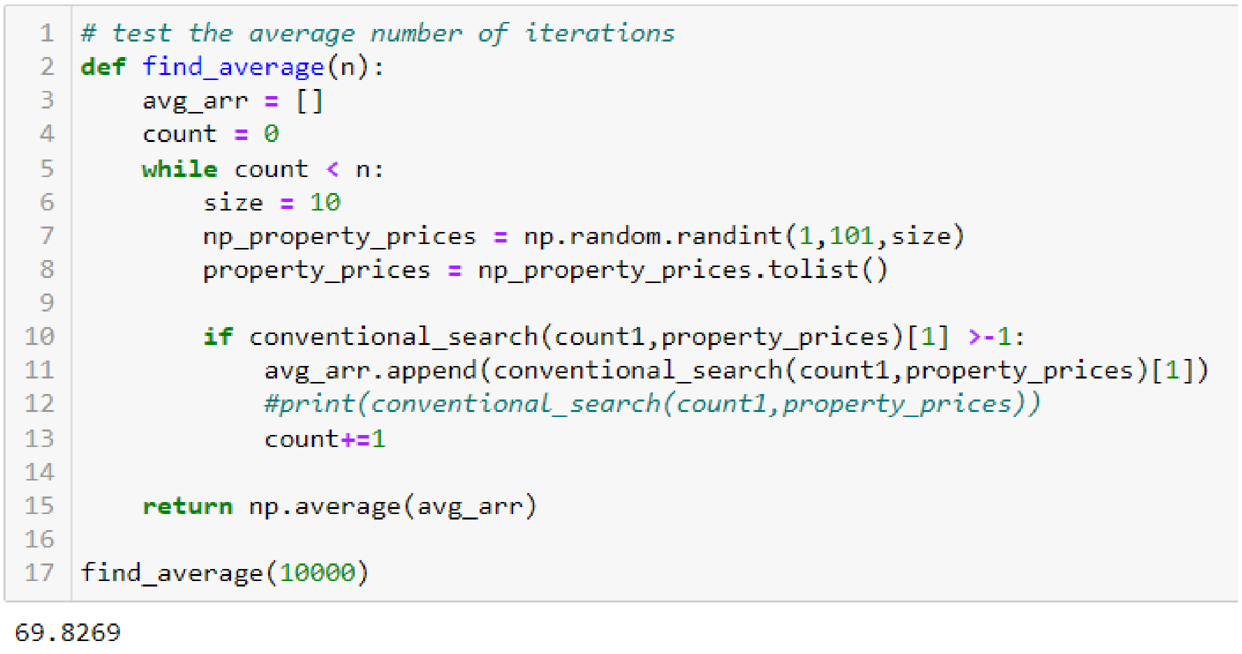


Figure 12. Results from the conventional computational searching algorithm.

As shown in Figure 12 the algorithm was able to create two groups of equal values from the initial array.

To observe the average number of iterations it takes to find an equal combination the following code ran the functions 100,000 times and took the average of the iterations of each trial, as seen below in Figure 13.



```
# test the average number of iterations
def find_average(n):
    avg_arr = []
    count = 0
    while count < n:
        size = 10
        np_property_prices = np.random.randint(1,101,size)
        property_prices = np_property_prices.tolist()

        if conventional_search(count1,property_prices)[1] >-1:
            avg_arr.append(conventional_search(count1,property_prices)[1])
            #print(conventional_search(count1,property_prices))
            count+=1

    return np.average(avg_arr)

find_average(10000)
```

```
69.8269
```

Figure 13. Running the conventional algorithm 100000 times to find the average number of iterations to find the combinations.

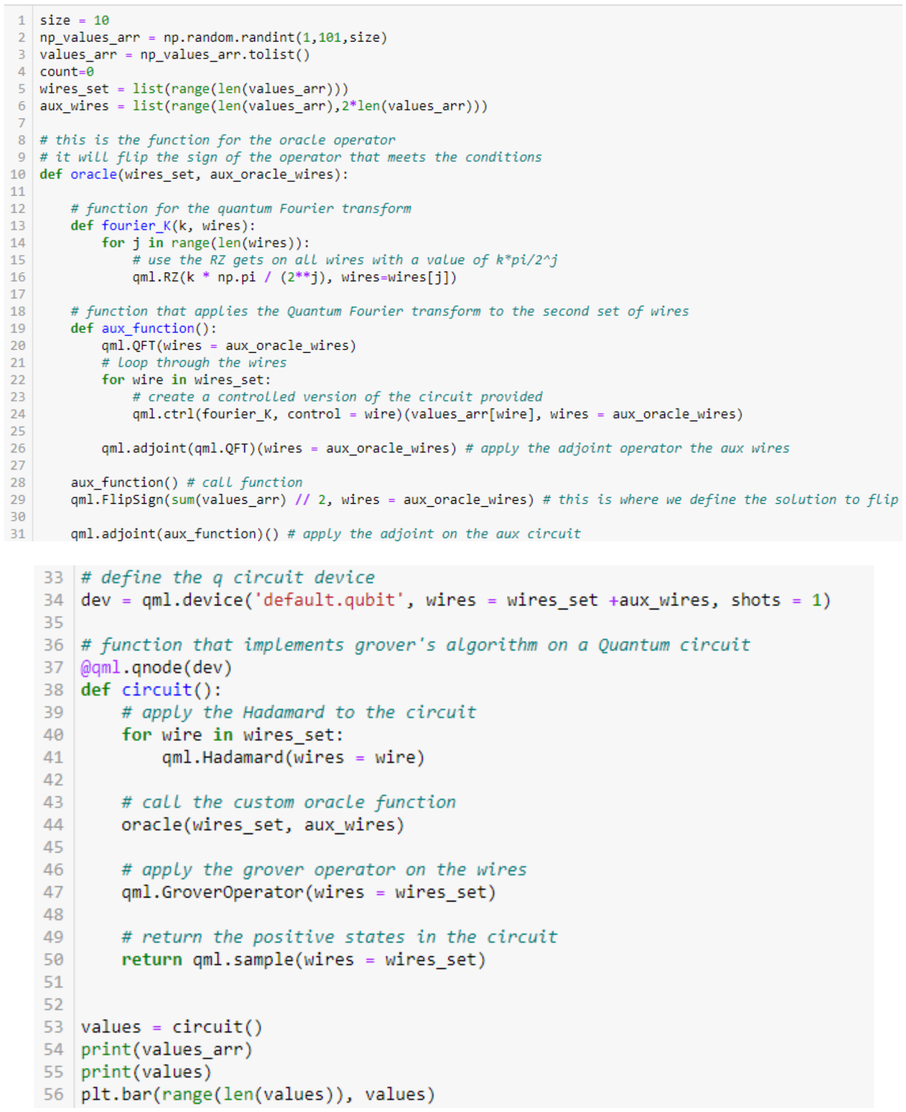
It can be seen that the average number of iterations to find equal combinations of a 10-element array is about 70 (representing N/2). Therefore, the N of the system is about 140.

3.2 Quantum Search Algorithm

To implement a quantum circuit that can find two equal combinations of a random 10 element array the Grover Operator along with some additional quantum functions were used. Since the problem is more complex than the 4-bit example shown previously the circuit had to implement a version of the Oracle Operator that utilizes a Quantum Fourier transform (QFT).

The QFT operates on a quantum state, which is represented by a complex-valued wave function. Given a quantum state with N quantum bits (qubits), the QFT maps this state to a new state with the same number of qubits, in which the qubits are in a superposition of all N possible states. The QFT can be thought of as a quantum version of the DFT, which maps a set of N complex numbers to another set of N complex numbers [7].

The following Python code implemented a quantum circuit able to search for the desired groupings within a given 10-element array as shown below in Figure 14.

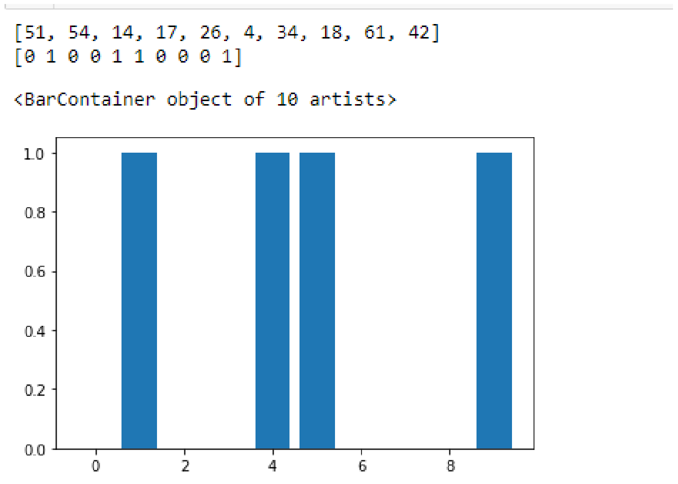


```python
size = 10
np_values_arr = np.random.randint(1,101,size)
values_arr = np_values_arr.tolist()
count=0
wires_set = list(range(len(values_arr)))
aux_wires = list(range(len(values_arr),2*len(values_arr)))

# this is the function for the oracle operator
# it will flip the sign of the operator that meets the conditions
def oracle(wires_set, aux_oracle_wires):

    # function for the quantum Fourier transform
    def fourier_K(k, wires):
        for j in range(len(wires)):
            # use the RZ gets on all wires with a value of k*pi/2^j
            qml.RZ(k * np.pi / (2**j), wires=wires[j])

    # function that applies the Quantum Fourier transform to the second set of wires
    def aux_function():
        qml.QFT(wires = aux_oracle_wires)
        # loop through the wires
        for wire in wires_set:
            # create a controlled version of the circuit provided
            qml.ctrl(fourier_K, control = wire)(values_arr[wire], wires = aux_oracle_wires)

        qml.adjoint(qml.QFT)(wires = aux_oracle_wires) # apply the adjoint operator the aux wires

    aux_function() # call function
    qml.FlipSign(sum(values_arr) // 2, wires = aux_oracle_wires) # this is where we define the solution to flip

    qml.adjoint(aux_function)() # apply the adjoint on the aux circuit
```

```python
# define the q circuit device
dev = qml.device('default.qubit', wires = wires_set +aux_wires, shots = 1)

# function that implements grover's algorithm on a Quantum circuit
@qml.qnode(dev)
def circuit():
    # apply the Hadamard to the circuit
    for wire in wires_set:
        qml.Hadamard(wires = wire)

    # call the custom oracle function
    oracle(wires_set, aux_wires)

    # apply the grover operator on the wires
    qml.GroverOperator(wires = wires_set)

    # return the positive states in the circuit
    return qml.sample(wires = wires_set)


values = circuit()
print(values_arr)
print(values)
plt.bar(range(len(values)), values)
```

Figure 14. Quantum Circuit utilizing a modified Grover's algorithm for a multi-variable problem [5].

The results of the modified quantum algorithm can be seen below in Figure 15.



```
[51, 54, 14, 17, 26, 4, 34, 18, 61, 42]
[0 1 0 0 1 1 0 0 0 1]

<BarContainer object of 10 artists>
```

Figure 15. Resulting groupings of the 10-element array after utilizing the quantum circuit.

As shown above the quantum circuit correctly outputs the equal groups of the supplied array. After running the quantum searching algorithm several times with a counter on each loop it was observed that each trial took the same number of iterations to find a correct combination which was 12. This matches the expected results since the average iterations N found using the conventional algorithm was about 140 and the expected number of iterations using Grover's algorithm is $\sqrt{N}$ which is about 11.81. Therefore, it can be seen from this example that a quantum circuit utilizing Grover's algorithm can solve a searching problem in $\sqrt{N}$ iterations when compared to a classical computational searching algorithm [5]. Note that the benefit of a quantum algorithm over a conventional algorithm only grows as the number of elements in the array increases. Furthermore, as the number of variables or elements in a list increase the difference in average iterations between a quantum algorithm and conventional algorithm increases.

Conclusion

In conclusion, this report has provided an introduction to quantum computing and the use of PennyLane, a Python library for quantum machine learning and quantum computing. The report has described the basic principles of quantum computing, including the representation of information as qubits and the use of quantum gates to manipulate and control the quantum state of qubits. The report has also explained how Grover's Algorithm can be used to improve the performance of linear search algorithms in quantum computing. Overall, the report has demonstrated the potential of quantum computing to solve complex problems and simulate quantum systems and has highlighted the importance of tools like PennyLane in enabling researchers and developers to build and deploy quantum computing models.

References

[1] Kurzgasagt - In a Nutshell, "Quantum Computers Explained – Limits of Human Technology," Youtube, 8 December 2015. [Online]. Available: https://www.youtube.com/watch?v=JhHMJCUmq28&ab_channel=Kurzgesagt%E2%80%93InaNutshell. [Accessed 2 December 2022].

[2] Quanta Magazine, "Quantum Computers, Explained With Quantum Physics," Youtube, 8 June 2021. [Online]. Available: https://www.youtube.com/watch?v=jHoEjvuPoB8&ab_channel=QuantaMagazine. [Accessed 1 December 2022].

[3] Xanadu, "About PennyLane," Xanadu, [Online]. Available: https://pennylane.ai/. [Accessed 4 December 2022].

[4] XANADU, "XANADU QUANTUM CODEBOOK," XANADU, [Online]. Available: https://codebook.xanadu.ai/. [Accessed 22 November 2022].

[5] O. Lockwood, "Grover's Algorithm and Diffusion Operator: a Complete Example Walkthrough," Youtube, 14 June 2021. [Online]. Available: https://www.youtube.com/watch?v=qluzL6d7i4w&t=1s&ab_channel=OwenLockwood. [Accessed 3 December 2022].

[6] Xanadu, "Grover's Algorithm | PennyLane Tutorial," Xanadu, 15 November 2022. [Online]. Available: https://www.youtube.com/watch?v=KeJqcnpPluc&ab_channel=Xanadu. [Accessed 10 December 2022].