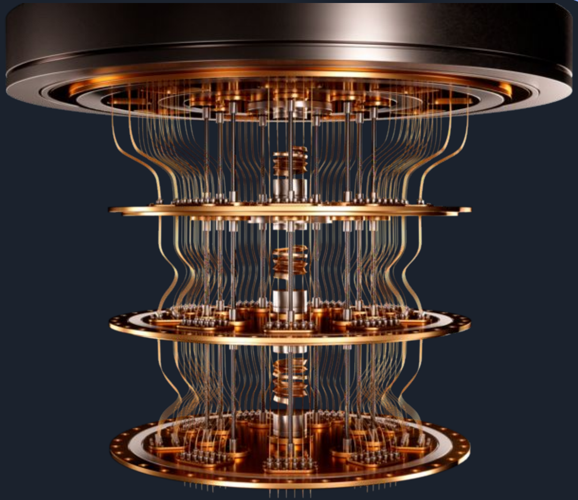
Quantum Mechanics Exploring

Quantum Computing

Presentation By: Nathan Pacey - 20153310
ENPH 344 - Introduction to Quantum Mechanics

Table of contents

Quantum Computing Information



Quantum computing uses qubits and quantum gates to perform calculations that are impossible with classical computers. It can solve certain problems efficiently, such as factoring large numbers and simulating quantum systems. In this report, Grover's Algorithm and PennyLane library will be used to improve a linear search algorithm.

PennyLane Introduction

PennyLane is a Python library for quantum machine learning and computing, supporting multiple hardware and software platforms. It allows researchers and developers to build, train, and deploy quantum computing models and integrate them with classical computing frameworks.

Background

Equation #1

$$|\psi\rangle = A|1\rangle + B|0\rangle$$

Equation #2

$$|A|^2 + |B|^2 = 1$$

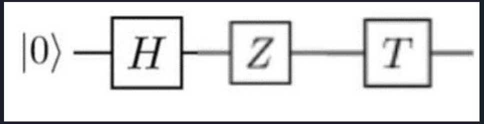
Quantum **Gates**

Quantum circuit gates are used to manipulate and control the state of qubits in quantum computers. They have unique properties compared to classical gates, such as quantum parallelism and the ability to create entanglement. Quantum circuit gates come in two types: single-qubit gates and multi-qubit gates. Examples of quantum circuit gates include the Pauli gates, the Hadamard gate, and the CNOT gate. The Penny Lane library includes a variety of quantum circuit gates.

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	$\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_y(\theta)}$	$RY(\theta)\|0\rangle = \cos\frac{\theta}{2}\|0\rangle + \sin\frac{\theta}{2}\|1\rangle$ $RY(\theta)\|1\rangle = -\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$

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.



Example #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.

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

```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()
```

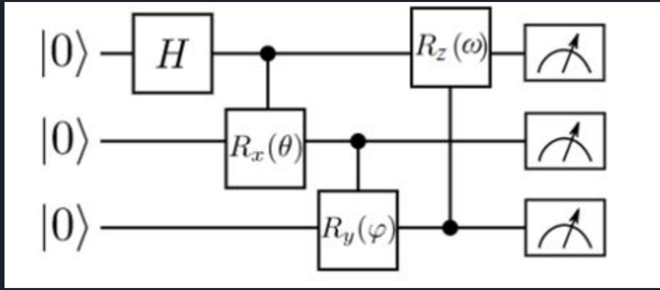
Quantum Circuits (Cont.)

```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]
```

Example #2

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



Introduction to Grover’s Algorithm

Grover's algorithm is a quantum algorithm that can search lists faster than classical algorithms by using quantum superposition and interference. It finds the desired element in a list of size N in O(sqrt(N)) iterations, compared to O(N) for classical algorithms. This allows it to search large lists efficiently.

Designing a Circuit to Apply Grover's Algorithm

Grover's algorithm uses a sequence of gates called the Grover iterate to amplify the probability of finding a desired element in a list. The Grover iterate is a unitary operator that applies the Hadamard operator, an oracle operator that marks the desired element, and the Hadamard operator again to create constructive and destructive interference between the elements in the list. This amplifies the probability of finding the desired element.

$$G = -I_s * H^N * O_w * H^N$$

Uniform Superposition and the Hadamard Gate

The equal superposition state is the starting point for Grover's algorithm and is a superposition of all possible states in the database. It is represented by Equation 5.

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

The equal superposition state is initialized using the Hadamard gate, represented by Equation 6. The state includes N elements in the database, represented by |i>.

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

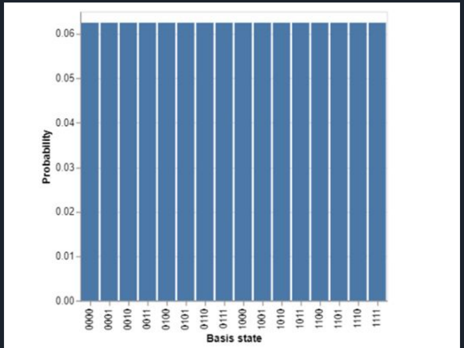
The Hadamard gate creates a superposition of the |0> and |1> states and can be applied to each qubit in the system to create the equal superposition state. The probability of observing each of the N states can be expressed by Equation 7.

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

Uniform Superposition and the Hadamard Gate (Cont.)

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

```python
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))

```



The probability of observing any of the given states after applying the Hadamard gate is equal, and the summation of the probabilities is 1.

Applying the Oracle Operator

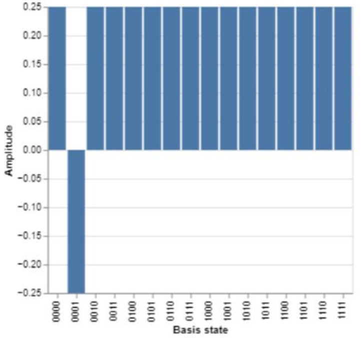
The oracle operator is represented mathematically by Equation 8.

$$O_w = I - 2|s\rangle\langle s|$$

In Grover's algorithm, the oracle operator flips the sign of the desired element's amplitude in a set of states. After applying the oracle, the target state has a negative amplitude of the same magnitude as the other states, as shown by Equation 9.

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

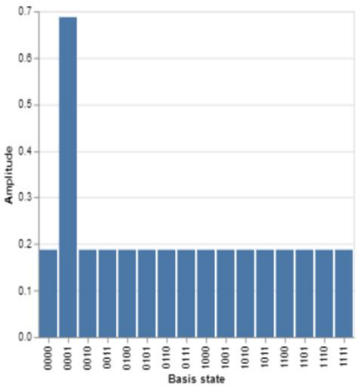
```
n_bits = 4
dev = qml.device("default.qubit", wires=n_bits)

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()

```



Applying the Diffusion Operator

The final step of Grover's algorithm is applying the Diffusion operator, which subtracts the outer product of the probability vector of each state from the identity matrix, resulting in the desired state having a greater final amplitude than the other states. This is shown in Equation 10 and implemented in the figures below.

```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

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()
```



$$D = 2\,|\psi\rangle\langle\psi| - I$$

Application of Grover’s Algorithm

A multi-variable search was conducted to compare the efficiency of Grover's algorithm to traditional algorithms. The search involved finding two groups in a 10-element array with equal sums and modifying the array so that the sum of the indices of the 1's equals the sum of the indices of the 0's. The quantum algorithm took a constant, linear number of iterations and was significantly faster than the traditional linear search.

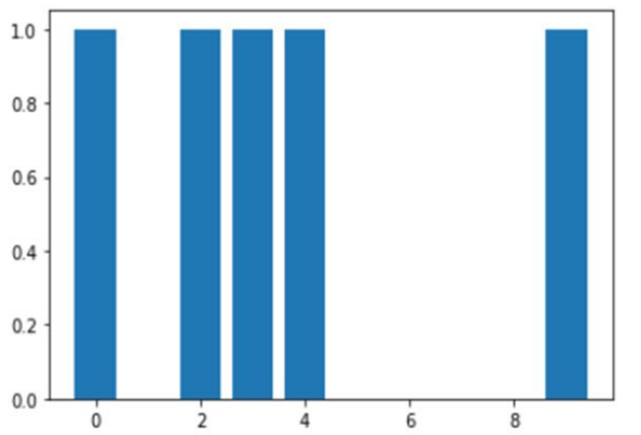
For example, if the given array was [44, 86, 18, 50, 30, 86, 67, 84, 96, 73] a possible solution would be [1, 1, 1, 0, 0, 0, 0, 0, 1, 1] since 44+86+18+96+73=317 and 50+30+86+67+84=317.
So, both groups of zeros and 1 have equal summations.

Conventional **Searching Algorithm**

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

```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)
```

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



Conventional Searching Algorithm (Cont.)

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.

```
# 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
```

Quantum Search Algorithm

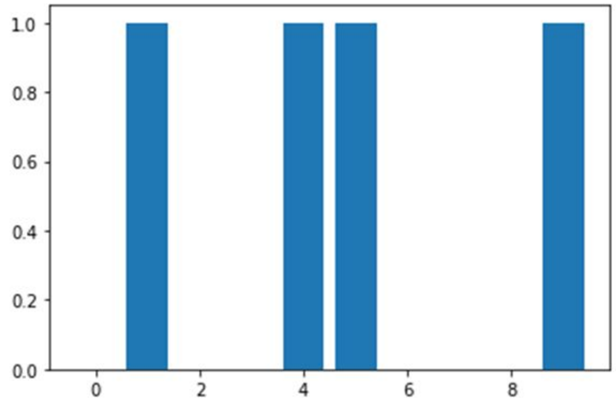
A quantum circuit was implemented using the Grover operator and additional functions to find two equal combinations in a random 10-element array. The circuit used a version of the Oracle operator that utilized a Quantum Fourier transform. The circuit was able to search for the desired groups and is shown below.

```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

# 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)
```

Quantum Search Algorithm (Cont.)

The quantum circuit correctly found equal groups in the supplied array. When the algorithm was run multiple times, each trial took the same number of iterations (12) to find a correct combination. This matches the expected result of √N iterations for Grover's algorithm, as compared to the average of N iterations for a classical computational searching algorithm. This demonstrates that a quantum circuit using Grover's algorithm can solve a searching problem in √N iterations.

```
[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>
```



Conclusion

This report introduced quantum computing and PennyLane, a Python library for quantum machine learning and computing. It explained quantum computing principles, including qubits and quantum gates, and how Grover's Algorithm can improve linear search algorithms. The report demonstrated the potential of quantum computing to solve complex problems and simulate quantum systems, and the role of tools like PennyLane in building and deploying quantum computing models.