Assignment 1

Installation of Qiskit

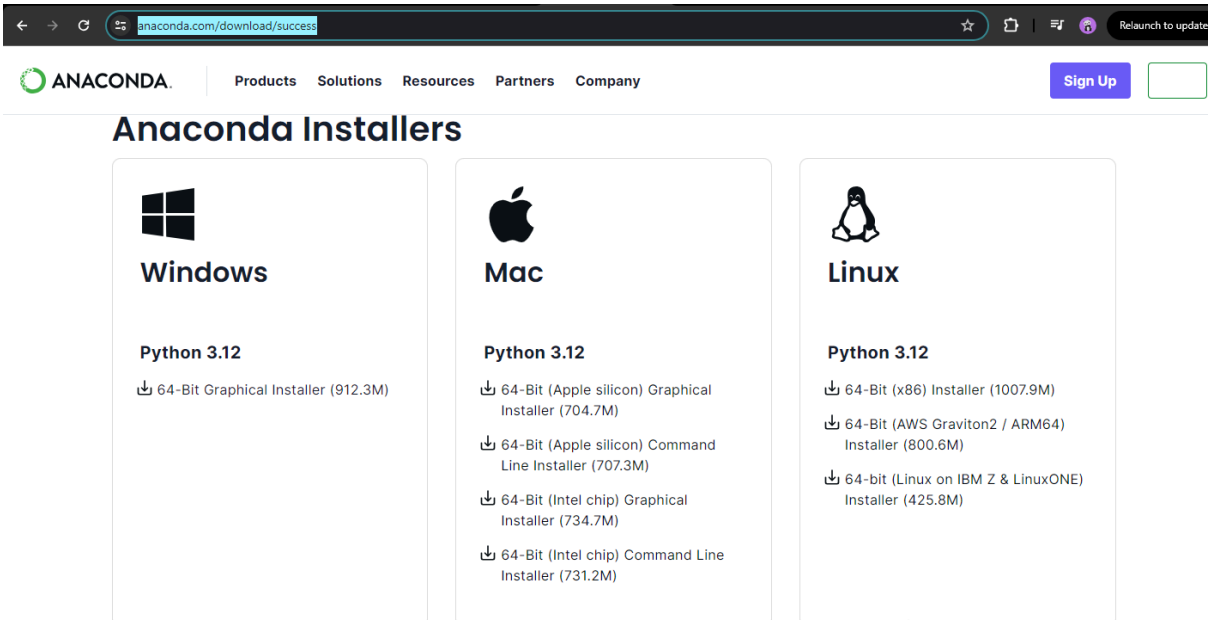
Step 1 :

Open command prompt (cmd prompt) and type: python –version (if you have python version installed then proceed else install python). and also : go to search and type Anaconda.

```
C:\Users\student>python --version
Python 3.12.1

C:\Users\student>
```

If Anaconda is not installed then install it from:

https://www.anaconda.com/download/success



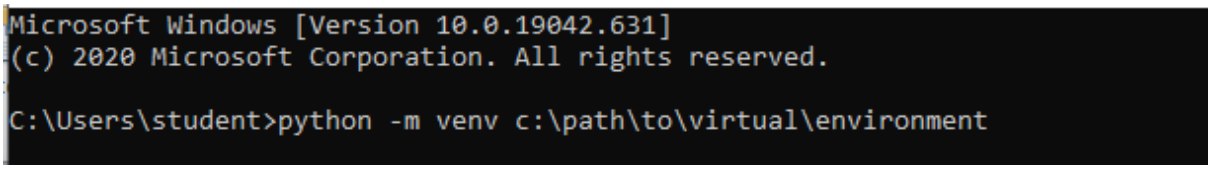
Step 2 :

Open IBM Qiskit installation : https://docs.quantum.ibm.com/guides/install-qiskit

Step 3 :

Write this as it is (do not do anything of your own)

python -m venv c:\path\to\virtual\environment



And after this copy commands as they are.

Copy the commands sequentially you see on the website and run them on cmd prompt.

Step 4 :

After running all the commands and if you see the message “No such file or directory” then just simply again write in cmd prompt “jupyter notebook” and click enter.

Now,search the path mentioned after the above step. For example my path mentioned was :

C:\Users\student\anaconda3\etc\jupyter

A similar path format will be mentioned just try to track it in your system. And locate “anaconda3” then open “etc”. That’s all.

Now to open the jupyter notebook again just write “Jupyter notebook” in cmd and enter. This will open jupyter notebook to make a new file click “new” and select “Python 3” and then write the code in environment in the below like area. After writing code just click on “run”.

Conclusion:

Hence, we have successfully installed Quiskit in Jupyter environment.

Assignment 2

Implementation of Linear algebra on Quiskit

Implement State Vectors using Quiskit-

A) **Creation of state Vectros.**
B) **Check the vectors are valid or not**
C) **Display the vectors using histogram.**

Theory :

1. Importing the Required Library

- First, you import the `Statevector` class from Qiskit's quantum information module. This class is used to represent and manipulate quantum state vectors.

2. Creating a Quantum State Vector

- You can create a quantum state vector that represents a specific quantum state. For example, if you want to represent the quantum state |0⟩|0\rangle|0⟩ (which is a common basis state in quantum computing), you can use the `from_label` method of the `Statevector` class with the label `'0'`.

3. Displaying the Quantum State Vector

- Once the state vector is created, you can display it to see its components. The state vector is typically represented as a complex-valued vector, where each entry corresponds to the amplitude of the respective basis state.

4. Checking the Validity of the State Vector

- In quantum mechanics, a valid state vector must be normalized, meaning the sum of the absolute squares of its components must equal 1. The `Statevector` class has a method called `is_valid` that checks this condition and returns `True` if the vector is valid, and `False` otherwise.

Example Process in Words:

1. **Import the Statevector Class**: Start by importing the `Statevector` class from Qiskit's quantum information module.
2. **Create the State Vector**: Use the `from_label` method to create a state vector for the quantum state |0⟩|0\rangle|0⟩. This method creates a vector where the state |0⟩|0\rangle|0⟩ has an amplitude of 1 and |1⟩|1\rangle|1⟩ has an amplitude of 0.
3. **Display the State Vector**: Print out the state vector to see its components. For |0⟩|0\rangle|0⟩, the vector will typically look like `[1.+0.j, 0.+0.j]`, indicating that the amplitude for the |0⟩|0\rangle|0⟩ state is 1 and for |1⟩|1\rangle|1⟩ is 0.
4. **Validate the State Vector**: Use the `is_valid` method to check if the state vector is properly normalized. If the vector is valid, the method returns `True`.

```
from qiskit.quantum_info import Statevector
from numpy import sqrt
u = Statevector([1 / sqrt(2), 1 / sqrt(2)])
v = Statevector([(1 + 2.0j) / 3, -2 / 3])
w = Statevector([1 / 3, 2 / 3])
x = Statevector([ 3, 2])
print("State vectors u, v, and w have been defined.")
```

Output :

```
State vectors u, v, and w have been defined.
```

```
display(u.draw("latex"))
display(u.draw("latex"))
display(x.draw("latex"))
```

Output

$$\frac{\sqrt{2}}{2}|0\rangle + \frac{\sqrt{2}}{2}|1\rangle$$

$$\frac{\sqrt{2}}{2}|0\rangle + \frac{\sqrt{2}}{2}|1\rangle$$

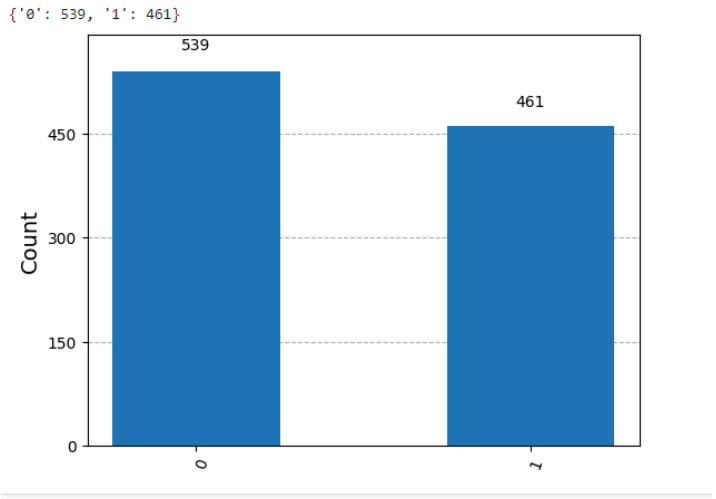
$$3|0\rangle + 2|1\rangle$$

```
display(u.is_valid())
display(x.is_valid())
```

Output

True
False

```
from qiskit.visualization import plot_histogram
statistics = v.sample_counts(1000)
display(statistics)
plot_histogram(statistics)
```



Conclusion:

This process allows you to create, visualize, and validate quantum state vectors, which are fundamental in describing the state of quantum systems. This ensures that the quantum states you work with are physically meaningful and adhere to the principles of quantum mechanics.

Assignment 3

Linear algebra: Vector operations, Vector multiplication, Tensor products

- **Install Qiskit** (if not already installed).
- **Import necessary modules** from Qiskit.
  Import Statevector and numpy
- **Create quantum state vectors**.
  You can create quantum state vectors using the `Statevector` class. For example, let's create state vectors for |0⟩|0\rangle|0⟩ and |1⟩|1\rangle|1⟩:
- **Compute the dot product** of two vectors.
  The dot product between two quantum state vectors can be calculated using the `numpy` library:
- **Compute the tensor product** of two vectors.

The tensor product (also known as the Kronecker product) between two state vectors can be computed using Qiskit's `tensor` method:

Inner Product/ dot product of two vectors:

```
from qiskit.quantum_info import Statevector

import numpy as np

# State vector for |0⟩

state_vector_0 = Statevector.from_label('0')

# State vector for |1⟩

state_vector_1 = Statevector.from_label('1')

# Convert state vectors to numpy arrays

vec_0 = state_vector_0.data

vec_1 = state_vector_1.data

# Compute the dot product (inner product)

dot_product = np.vdot(vec_0, vec_1)

print("Dot product:", dot_product)
```

Output:

```
Dot product: 0j
```

Tensor Product of two vectors:

```
# Compute the tensor product
tensor_product = state_vector_0.tensor(state_vector_1)
print("Tensor product:", tensor_product)
from qiskit.quantum_info import Statevector
import numpy as np

# Step 1: Create quantum state vectors
state_vector_0 = Statevector.from_label('0')
state_vector_1 = Statevector.from_label('1')

# Step 2: Compute the dot product (inner product)
vec_0 = state_vector_0.data
vec_1 = state_vector_1.data
dot_product = np.vdot(vec_0, vec_1)
print("Dot product:", dot_product)
# Step 3: Compute the tensor product
tensor_product = state_vector_0.tensor(state_vector_1)
print("Tensor product:", tensor_product)
```

Output:

```
Tensor product: Statevector([0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
            dims=(2, 2))
```

```
from qiskit.quantum_info import Statevector
import numpy as np
```

```
# Step 1: Create quantum state vectors
state_vector_0 = Statevector.from_label('0')
state_vector_1 = Statevector.from_label('1')
# Step 2: Compute the dot product (inner product)
vec_0 = state_vector_0.data
vec_1 = state_vector_1.data
dot_product = np.vdot(vec_0, vec_1)
print("Dot product:", dot_product)
# Step 3: Compute the tensor product
tensor_product = state_vector_0.tensor(state_vector_1)
print("Tensor product:", tensor_product)
```

Output:

```
Dot product: 0j
Tensor product: Statevector([0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
            dims=(2, 2))
```

Assignment 4

Implementation of Identity matrix:1 Qubit, 2 Qubits, 3 Qubits

Theory

Identity Matrix is a square matrix whose all diagonal elements are equal to 1 and rest all elements are zero. Identity Matrix is also known as Unit Matrix. In simple terms, all diagonal elements are equal to 1 and rest are zero. The main condition for Identity matrix is that it should be a square matrix i.e. the no of rows should be equal to zero.

Identity Matrix Definition (Unit Matrix)

A unit matrix, or identity matrix, is a square matrix whose principal diagonal elements are ones, and the rest of the elements of the matrix are zeros. An identity matrix is always a square matrix and is expressed as "I." For example, "In" is the identity matrix of order n, i.e., it has "n" rows and columns.

It is a scalar matrix as the diagonal elements are same. When the product of any two square matrices is an identity matrix, then the matrices are said to be inverses of each other. The rank of an identity matrix of order "n × n" is n, as it has "n" linearly independent rows (or columns). The matrix given below represents an identity matrix of order "n by n."

$$I_{n\times n} = \begin{bmatrix} 1 & 0 & . & . & . & 0 \\ 0 & 1 & . & . & . & 0 \\ . & . & 1 & & & . \\ . & . & & 1 & & . \\ . & . & & & 1 & . \\ 0 & 0 & . & . & . & 1 \end{bmatrix}_{n\times n}$$

Unit matrix of order n × n

Examples of Identity Matrix

A square matrix P = [aij] is said to be an identity matrix when aij = 1 for i = j and aij = 0 for i ≠ j.

⇒ The matrix given below is an identity matrix of order "2 × 2."

$$I_{2\times2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_{2\times2}$$

⇒ The matrix given below is an identity matrix of order "3 × 3."

$$I_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}_{3\times3}$$

PROGRAM :

```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Operator

# Function to create identity matrix for n qubits
def identity_matrix(n_qubits):
    # Create a quantum circuit with n qubits
    qc = QuantumCircuit(n_qubits)

    # Apply the identity gate to all qubits using the `id()` method
    for i in range(n_qubits):
        qc.id(i)

    # Convert the quantum circuit to an operator (matrix)
    identity_matrix_nq = Operator(qc).data

    return identity_matrix_nq

# Example: Identity matrix for 1, 2, and 3 qubits
identity_matrix_1q = identity_matrix(1)
identity_matrix_2q = identity_matrix(2)
identity_matrix_3q = identity_matrix(3)

print("Identity matrix for 1 qubit:")
print(identity_matrix_1q)

print("\nIdentity matrix for 2 qubits:")
print(identity_matrix_2q)

print("\nIdentity matrix for 3 qubits:")
print(identity_matrix_3q)
```

```
Identity matrix for 1 qubit:
[[1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j]]

Identity matrix for 2 qubits:
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]

Identity matrix for 3 qubits:
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
```

ASSIGNMENT 5

Implementation of 1- Qubit Gate

a) **Pauli X,**
b) **Pauli Y**
c) **Pauli Z gate**
d) **Hadamard Gate**

Create a Quantum Circuit:

Code

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)

qc.qubits
```

Explanation

This indicates that qc.qubits returns a list containing the qubits in the quantum circuit qc. Each qubit is represented by a Qubit object, which shows the quantum register it belongs to (QuantumRegister(2, 'q')) and its index (0 or 1 in this case).

Output:

```
[Qubit(QuantumRegister(2, 'q'), 0), Qubit(QuantumRegister(2, 'q'), 1
```

Add X Gate to Qubits 0

The Pauli X gate flips the state of the qubit from |0⟩ to |1⟩ and vice versa, essentially performing a classical NOT operation in the context of quantum computing. This gate is fundamental and frequently used in quantum algorithms and circuits.

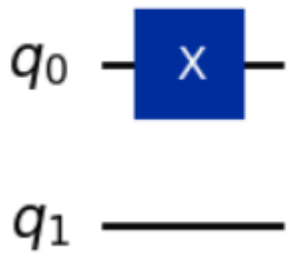
Code

```
qc.x(0)  # Add X-gate to qubit 0
qc.data
```

Output

```
[CircuitInstruction(operation=Instruction(name='x', num_qubits=1, num_clbits=0,
params=[]), qubits=(Qubit(QuantumRegister(2, 'q'), 0),), clbits=())]
```

```
qc.draw("mpl")
```



Add H gate to Qubit 0

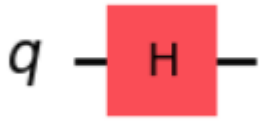
Code:

```
from qiskit.circuit.library import HGate

qc = QuantumCircuit(1)

qc.append(

   HGate(),  # New HGate instruction

   [0]       # Apply to qubit 0

)

qc.draw("mpl")
```

Here's what each part of the code does:

- `qc = QuantumCircuit(1)`: Creates a quantum circuit with 1 qubit.
- `HGate()`: Represents the Hadamard gate (`H`) as an instruction that can be appended to a quantum circuit.
- `qc.append(HGate(), [0])`: Appends the Hadamard gate (`H`) instruction to qubit 0 of the quantum circuit `qc`.
- `plot_circuit(qc, style='mpl')`: Draws the quantum circuit `qc` using matplotlib (`"mpl"` style).

Output:



Add Y Gate to Qubits 0

```
from qiskit import QuantumCircuit

# Create a quantum circuit with 1 qubit

qc = QuantumCircuit(1)

# Apply a Y gate to qubit 0

qc.y(0)

# Draw the circuit using matplotlib

qc.draw('mpl')
```

1. `qc = QuantumCircuit(1)`: Creates a quantum circuit with 1 qubit.
2. `qc.y(0)`: Applies a Y gate (Pauli Y gate) to qubit 0.
3. `qc.draw('mpl')`: Draws the quantum circuit using matplotlib (`'mpl'` style).

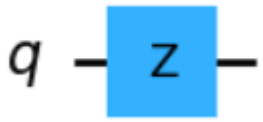
Output :

Add Z Gate to Qubits 0

```
from qiskit import QuantumCircuit

# Create a quantum circuit with 1 qubit

qc = QuantumCircuit(1)
```

```
# Apply a Z gate to qubit 0

qc.z(0)

# Draw the circuit using matplotlib

qc.draw('mpl')
```

1. `qc = QuantumCircuit(1)`: Creates a quantum circuit with 1 qubit.
2. `qc.z(0)`: Applies a Z gate (Pauli Z gate) to qubit 0.
3. `qc.draw('mpl')`: Draws the quantum circuit using matplotlib (`'mpl'` style).



ASSIGNMENT 5

Implementation of 2- Quibits Quantum Gates

a)CNOT Gate

b) Phase Gate

c)Swap Gate

ASSIGNMENT 6

Implementation of three Qubits FREDKIN Gate

Explanation:

1. **Imports**:
   - `QuantumCircuit`, `transpile`, and `assemble` are imported from `qiskit`.
   - `plot_histogram` is imported from `qiskit.visualization`.
   - `Aer` is imported from `qiskit_aer`.
   - `Sampler` is used to run the circuit simulation.
2. **Create and Configure Circuit**:
   - Define a quantum circuit with 3 qubits and apply the Fredkin gate.
3. **Add Measurements**:
   - Add measurements to observe the results.
4. **Print Circuit Diagram**:
   - Display the quantum circuit layout.
5. **Simulate the Circuit**:
   - Set up the simulator backend with `Aer`, transpile and assemble the circuit, and use `Sampler` for running the simulation.
6. **Get and Plot Results**:
   - Retrieve measurement results and visualize them with a histogram.

FREDKIN GATE PROGRAM:

```
from qiskit import QuantumCircuit, transpile, assemble

from qiskit.visualization import plot_histogram

from qiskit_aer import Aer

from qiskit.primitives import Sampler

# Create a Quantum Circuit with 3 qubits

qc = QuantumCircuit(3)

# Apply the Fredkin gate (control qubit 0, target qubits 1 and 2)

qc.cswap(0, 1, 2)

# Add measurement to all qubits

qc.measure_all()

# Print the circuit diagram
```

```
print("Quantum Circuit:")

print(qc.draw())

# Simulate the circuit

backend = Aer.get_backend('aer_simulator')

compiled_circuit = transpile(qc, backend)

qobj = assemble(compiled_circuit)

sampler = Sampler(backend)

result = sampler.run(qc, shots=1024).result()

# Get the results

counts = result.get_counts()

print("Measurement Results:", counts)

# Plot the results

plot_histogram(counts)
```

OUTPUT: