# Installing Libraries

!pip install qiskit

!pip install pylatexenc

!pip install qiskit-aer

# Importing libraries

from qiskit import \*

from qiskit.tools.monitor import job\_monitor

from qiskit.tools.visualization import plot\_state\_qsphere

from qiskit.visualization import plot\_histogram, plot\_bloch\_multivector, array\_to\_latex

##== creating a circuit (Defining Circuit)  ==##

circuit = QuantumCircuit(1, 1)

#                       (q, c) we can only change the initial state of Q-bit not classical bits or cbit.

## initialize a different state

# [0, 1] is a column vector that represent state |1> and [1, 0] represents |0>

# i.e., [0, 1] = |1>     [1, 0] = |0>

initial\_state = [0,1]

circuit.initialize(initial\_state, 0)

##== Apply measurement  ==##

# measure F^n has two arguments, qbit and cbit

# in our case we only have one qbit i.e., 0th qbit which we want to store in a cbit

circuit.measure(0, 0)

# measure\_all F^n measures all the qbits and stores it into the cbit

# sometimes we want to measure specific qbits for that we wont be needing

# measure\_all F^n

# circuit.measure\_all()

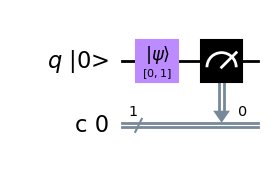
##== drawing a circuit  ==##

#  mpl means "Math Plot Library"

circuit.draw(output = 'mpl', initial\_state = True)

#here circuit flow will be left to right

#classical bits are for measurement. Its used to store the result in the memory.



# What do we have to do after building a Circuit?

1. Build a circuit

2. Debug/run it on a simulation

3. Run it on real Quantum Device

# How to run it on a Simulator?

# First, we need to understand the purpose of the simulation. From a simulator we get 3 outputs

1. State Vector

2. Unitary Matrix

3. Result after measurement

# Building a Circuit of one Qbit

circuit = QuantumCircuit(1)

# Apply Hadamard Gate

circuit.h(0)

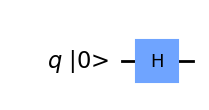
#        (0)

# here the argument in the Hadamard F^n i.e., h() is the nth qbit

# in our case it the 0th qbit that we want to use

# Drawing a circuit

circuit.draw(output = 'mpl', initial\_state=True)



# after building this circuit

# we want to find out the state vector

# here state in our case is just two, 0 and 1

## to run this on simulator we first have to do the following

#here Aer is a package on which we are calling get\_backend() F^n

simulator = Aer.get\_backend('aer\_simulator')

## save the statevector

# we have to save the state of the circuit using the following F^n

circuit.save\_statevector()

## Assemble our Circuit

# next we have to assemble the circuit into a Quantum Object

# so that we can run that Quantum Object into the Simulator

qobj = assemble(circuit)

## Run on Simulator

# inside the argument of run() F^n we are calling the Quantum object that

# we have created i.e., qobj in our case

# the out put of this is being saved into job

job = simulator.run(qobj)

## Monitoring the process

# To find whether there was an issue or an error in our job

# we are using a job\_monitor() F^n

job\_monitor(job)

## Get Results from job

# in result there are a lot of information being stored at

# to access that we can use print

result = job.result()

## Get State Vector from result

final\_state = result.get\_statevector()

Job Status: job has successfully run

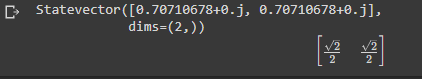
# Visualize the Final State/ State Vector

print(final\_state)

array\_to\_latex(final\_state)

# here the +0.j is the complex part

# and the root that is showing is the latex view of the result



plot\_bloch\_multivector(final\_state)

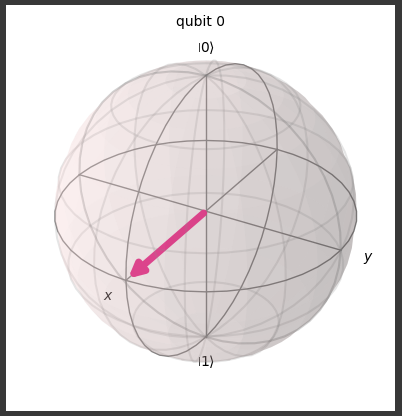
#since its not pointing to either |0> or |1> meaning both state are existing

# this is why its called superposition.

# We apply hadamard gate to make a qbit go into superposition

# |0> state means 0% probability of existing there.

# |1> state means 100% probability of existing there.



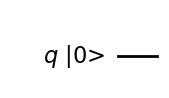
# Unitary Matrix

# create a circuit

circuit = QuantumCircuit(1) #here camel case is used

# draw circuit

circuit.draw(output="mpl", initial\_state=True)



# running simulator

simulator = Aer.get\_backend('aer\_simulator')

# save unitary

circuit.save\_unitary()

# assemble

qobj = assemble(circuit)

# run on simulator

job = simulator.run(qobj)

# monitor the process

job\_monitor(job)

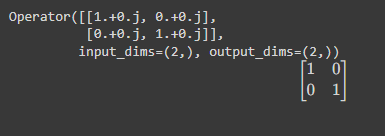
# get the result

result = job.result()

# get the unitary from the result

unitary = result.get\_unitary()

Job Status: job has successfully run



## Now using Identity Gate

# create a circuit

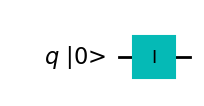
circuit = QuantumCircuit(1) #here camel case is used

# applying identity gate

circuit.id(0) # to the 0th Qbit

# draw circuit

circuit.draw(output="mpl", initial\_state=True)



# running simulator

simulator = Aer.get\_backend('aer\_simulator')

# save unitary

circuit.save\_unitary()

# assemble

qobj = assemble(circuit)

# run on simulator

job = simulator.run(qobj)

# monitor the process

job\_monitor(job)

# get the result

result = job.result()

# get the unitary from the result

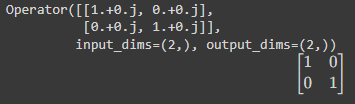
unitary = result.get\_unitary()

Job Status: job has successfully run

print(unitary)

array\_to\_latex(unitary)

#the result is the same as before!



## Now using Hadamard Gate

# create a circuit

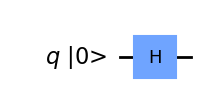
circuit = QuantumCircuit(1) #here camel case is used

# applying identity gate

circuit.h(0) # to the 0th Qbit

# draw circuit

circuit.draw(output="mpl", initial\_state=True)



# running simulator

simulator = Aer.get\_backend('aer\_simulator')

# save unitary

circuit.save\_unitary()

# assemble

qobj = assemble(circuit)

# run on simulator

job = simulator.run(qobj)

# monitor the process

job\_monitor(job)

# get the result

result = job.result()

# get the unitary from the result

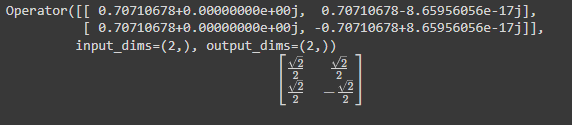
unitary = result.get\_unitary()

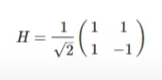
Job Status: job has successfully run

print(unitary)

array\_to\_latex(unitary)

#the result is different!





This is the actual representation of Hadamard gate in matrix form!

## Now using two Hadamard Gate

# create a circuit

circuit = QuantumCircuit(1) #here camel case is used

# applying identity gate

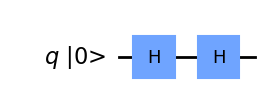
circuit.h(0) # to the 0th Qbit

#applying another gate

circuit.h(0)

# draw circuit

circuit.draw(output="mpl", initial\_state=True)



# running simulator

simulator = Aer.get\_backend('aer\_simulator')

# save unitary

circuit.save\_unitary()

# assemble

qobj = assemble(circuit)

# run on simulator

job = simulator.run(qobj)

# monitor the process

job\_monitor(job)

# get the result

result = job.result()

# get the unitary from the result

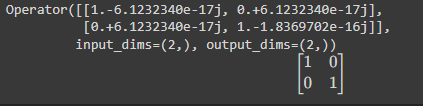
unitary = result.get\_unitary()

Job Status: job has successfully run

print(unitary)

array\_to\_latex(unitary)

#the result is different!



Here, two Hadamard gates are multiplied to give the answer as a unitary matrix.

Note: The quantum gates are inverse to each other, meaning if we apply same gate to each other, they cancel out!

## To see the sate vector of this circuit we have to do the following

# running simulator

simulator = Aer.get\_backend('aer\_simulator')

# save State Vector

circuit.save\_statevector()

# assemble

qobj = assemble(circuit)

# run on simulator

job = simulator.run(qobj)

# monitor the process

job\_monitor(job)

# get the result

result = job.result()

# get the unitary from the result

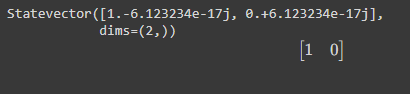
final\_state = result.get\_statevector()

Job Status: job has successfully run

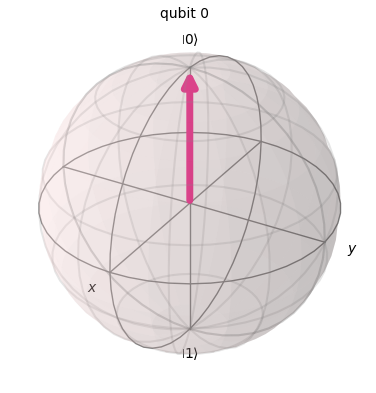
print(final\_state)

array\_to\_latex(final\_state)

#the State vector is shown!



plot\_bloch\_multivector(final\_state)



Since the Hadamard gates are cancelling out, the state is returning to its initial state which is |0>