Actividad 2 - Laboratorio Quantum Computing

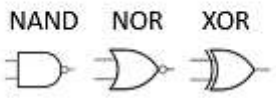
Arif Morán Velázquez/A01234442

Lab 1

In [22]:
```
from qiskit import *
from qiskit.visualization import plot_histogram
from qiskit_ibm_runtime import QiskitRuntimeService
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
```

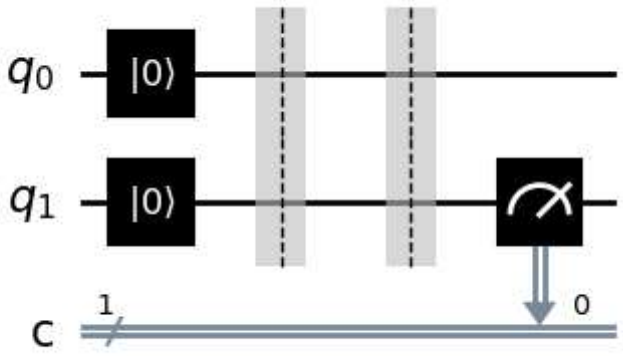
In [15]:
```
img = Image.open('tt.png')
display(img)
```

X	Y	AND(X,Y)	OR(X,Y)	NAND(X,Y)	NOR(X,Y)	XOR(X,Y)
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0



In [49]:
```
qc = QuantumCircuit(2, 1)
qc.reset(range(2))

    # barrier between input state and gate operation
qc.barrier()
    # program for quantum XOR gate goes




    # barrier between input state and gate operation
qc.barrier()
qc.measure(1,0)

qc.draw()
```

Out[49]:



In [53]:
```
#We'll run the program on a simulator
backend = Aer.get_backend('aer_simulator')
    #Since the output will be deterministic, we can use just a single shot to ge
job = backend.run(qc, shots=1, memory=True)
output = job.result().get_memory()[0]
output
```

Out[53]: '0'

In [6]:
```
def NOT(inp):
    """An NOT gate.

    Parameters:
        inp (str): Input, encoded in qubit 0.

    Returns:
        QuantumCircuit: Output NOT circuit.
        str: Output value measured from qubit 0.
    """

    qc = QuantumCircuit(1, 1) # A quantum circuit with a single qubit and a sing
    qc.reset(0)

    # We encode '0' as the qubit state |0⟩, and '1' as |1⟩
    # Since the qubit is initially |0⟩, we don't need to do anything for an inpu
    # For an input of '1', we do an x to rotate the |0⟩ to |1⟩
    if inp=='1':
        qc.x(0)

    # barrier between input state and gate operation
    qc.barrier()

    # Now we've encoded the input, we can do a NOT on it using x
    qc.x(0)

    #barrier between gate operation and measurement
    qc.barrier()

    # Finally, we extract the |0⟩/|1⟩ output of the qubit and encode it in the b
    qc.measure(0,0)
    qc.draw()

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
```

```
    # Since the output will be deterministic, we can use just a single shot to g
    job = backend.run(qc, shots=1, memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

In [62]:

```
def XOR(inp1,inp2):
    """An XOR gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output XOR circuit.
        str: Output value measured from qubit 1.
    """

    qc = QuantumCircuit(2, 1)
    qc.reset(range(2))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    # barrier between input state and gate operation
    qc.barrier()
    # program for quantum XOR gate goes

    qc.cx(0,1)

    # barrier between input state and gate operation
    qc.barrier()

    qc.measure(1,0) # output from qubit 1 is measured

    #We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    #Since the output will be deterministic, we can use just a single shot to ge
    job = backend.run(qc, shots=1, memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

In [66]:

```
def AND(inp1,inp2):
    """An AND gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output XOR circuit.
        str: Output value measured from qubit 2.
    """
    qc = QuantumCircuit(3, 1)
    qc.reset(range(2))
```

```
    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()

    # this is where your program for quantum AND gate goes
    qc.ccx(0,1,2)




    qc.barrier()
    qc.measure(2, 0) # output from qubit 2 is measured

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    # Since the output will be deterministic, we can use just a single shot to g
    job = backend.run(qc, shots=1, memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

In [68]:

```
def NAND(inp1,inp2):
    """An NAND gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output NAND circuit.
        str: Output value measured from qubit 2.
    """
    qc = QuantumCircuit(3, 1)
    qc.reset(range(3))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()

    # this is where your program for quantum NAND gate goes
    qc.ccx(0,1,2)
    qc.x(2)




    qc.barrier()
    qc.measure(2, 0) # output from qubit 2 is measured

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    # Since the output will be deterministic, we can use just a single shot to g
```

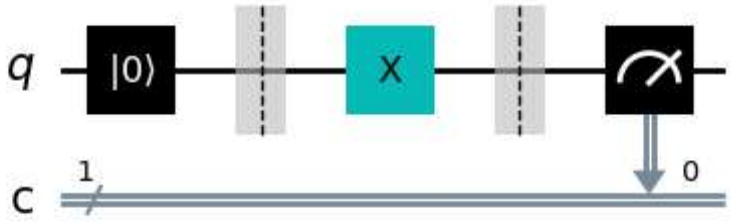
```
    job = backend.run(qc,shots=1,memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

In [83]:
```
def OR(inp1,inp2):
    """An OR gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output XOR circuit.
        str: Output value measured from qubit 2.
    """

    qc = QuantumCircuit(3, 1)
    qc.reset(range(3))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()
    qc.x(0)  # Set qubit 0 to state |1⟩
    qc.x(1)  # Set qubit 1 to state |1⟩
    qc.x(2)  # Set qubit 2 to state |1⟩
    qc.ccx(0, 1, 2)  # Controlled-controlled-X gate (Toffoli gate) with qubits 0



    qc.barrier()
    qc.measure(2, 0) # output from qubit 2 is measured

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    # Since the output will be deterministic, we can use just a single shot to g
    job = backend.run(qc,shots=1,memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

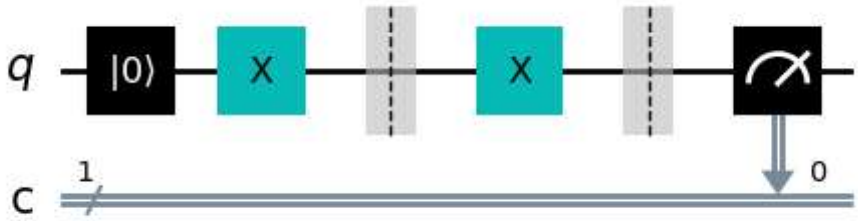
NOT

In [18]:
```
## Test the function
for inp in ['0', '1']:
    qc, out = NOT(inp)
    print('NOT with input',inp,'gives output',out)
    display(qc.draw())
    print('\n')
```

```
NOT with input 0 gives output 1
```



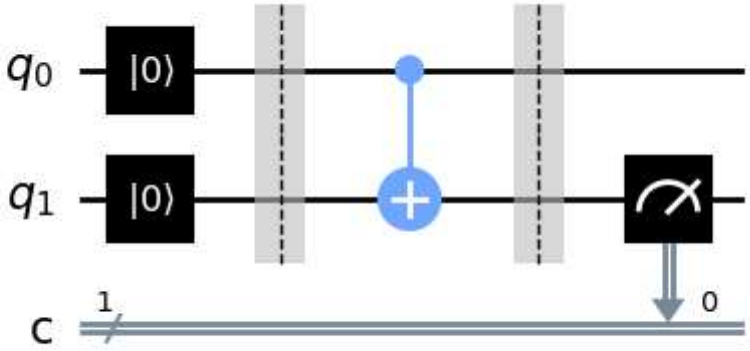
NOT with input 1 gives output 0



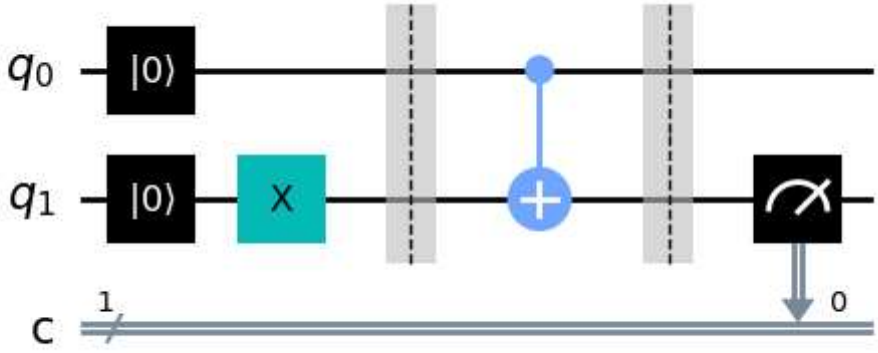
xor

In [86]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = XOR(inp1, inp2)
        print('XOR with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

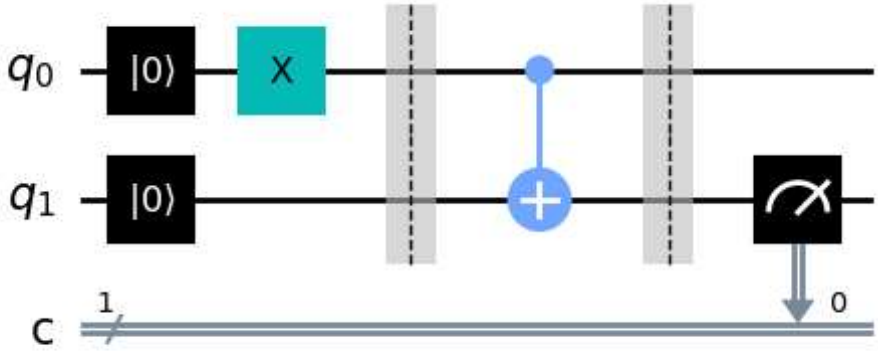
XOR with inputs 0 0 gives output 0



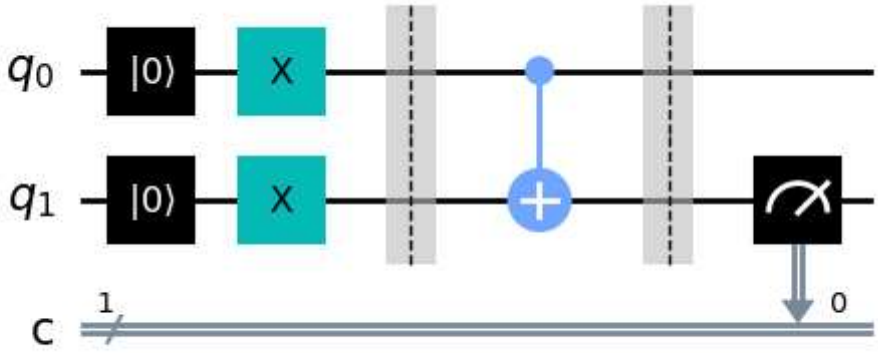
XOR with inputs 0 1 gives output 1



XOR with inputs 1 0 gives output 1



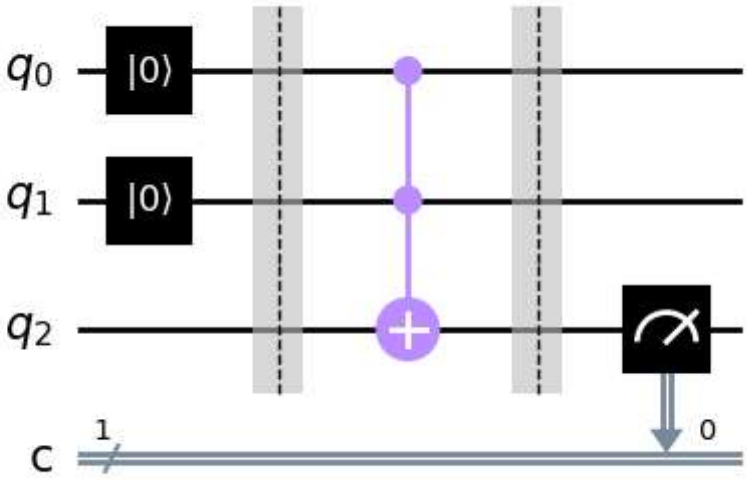
XOR with inputs 1 1 gives output 0



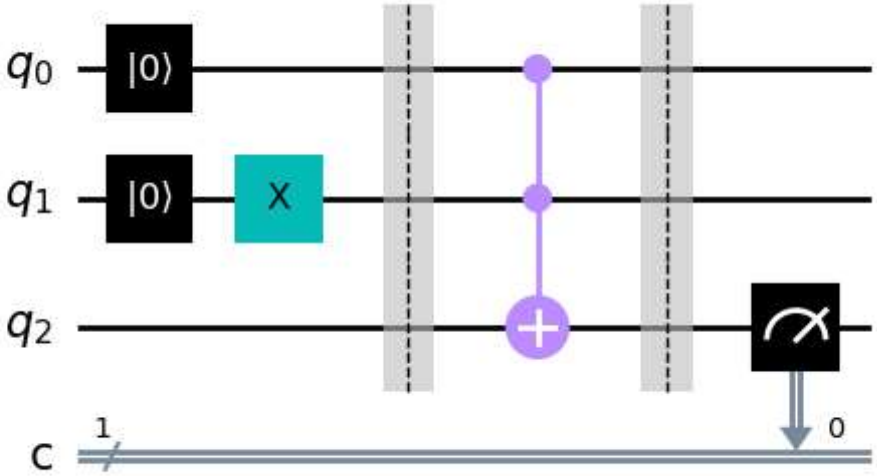
AND

In [88]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = AND(inp1, inp2)
        print('AND with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

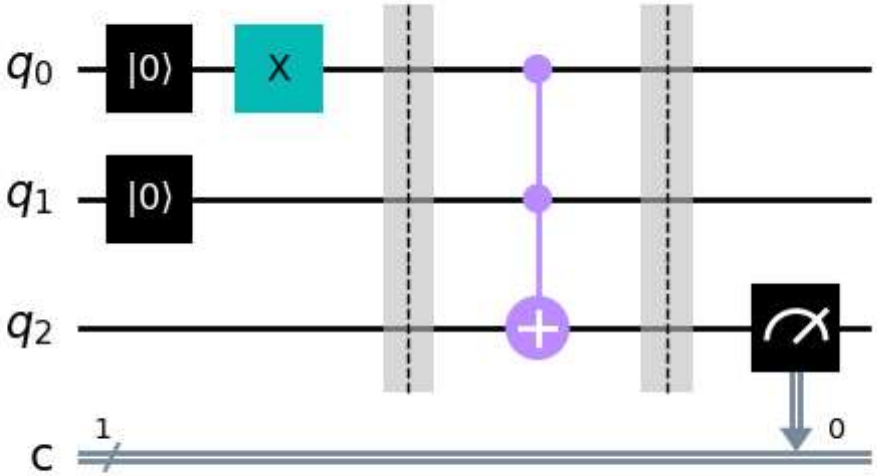
AND with inputs 0 0 gives output 0



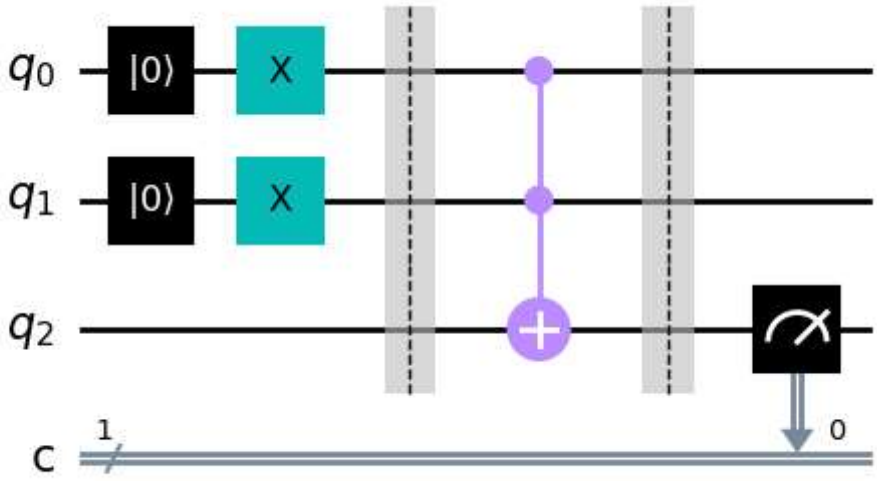
AND with inputs 0 1 gives output 0



AND with inputs 1 0 gives output 0



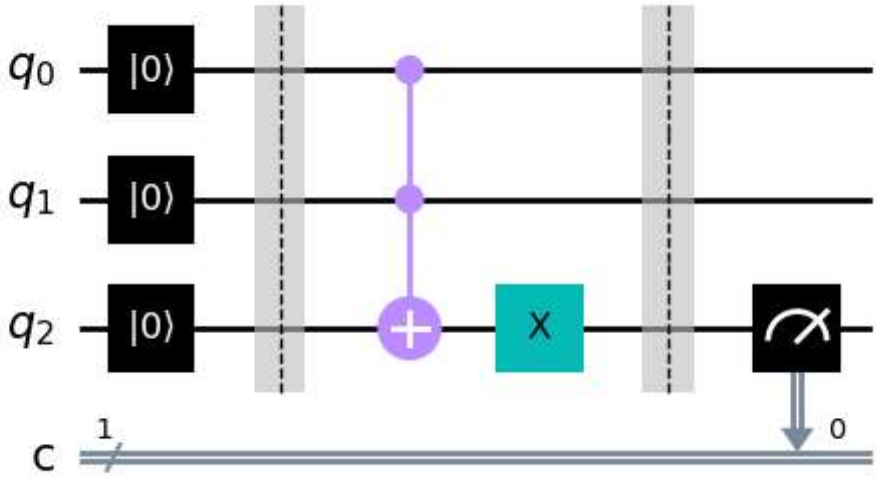
AND with inputs 1 1 gives output 1



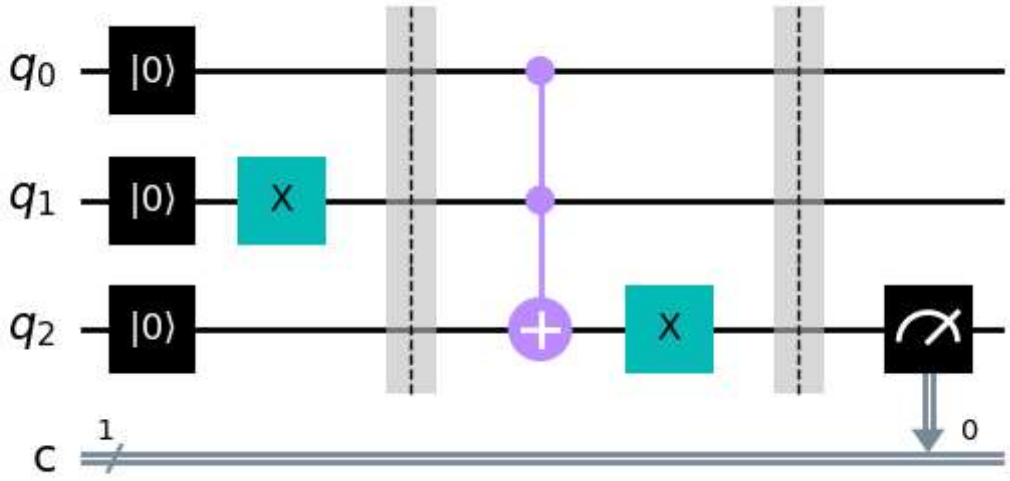
NAND

In [89]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = NAND(inp1, inp2)
        print('NAND with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

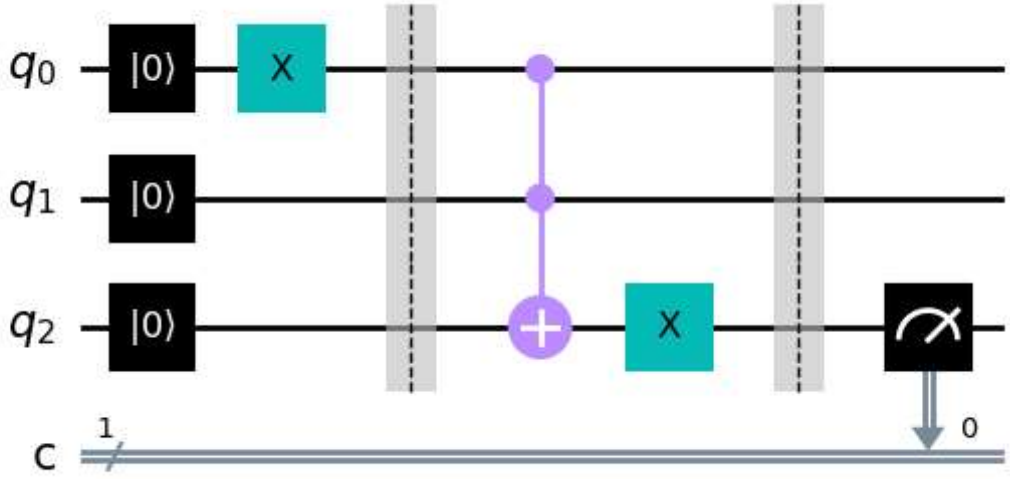
```
NAND with inputs 0 0 gives output 1
```



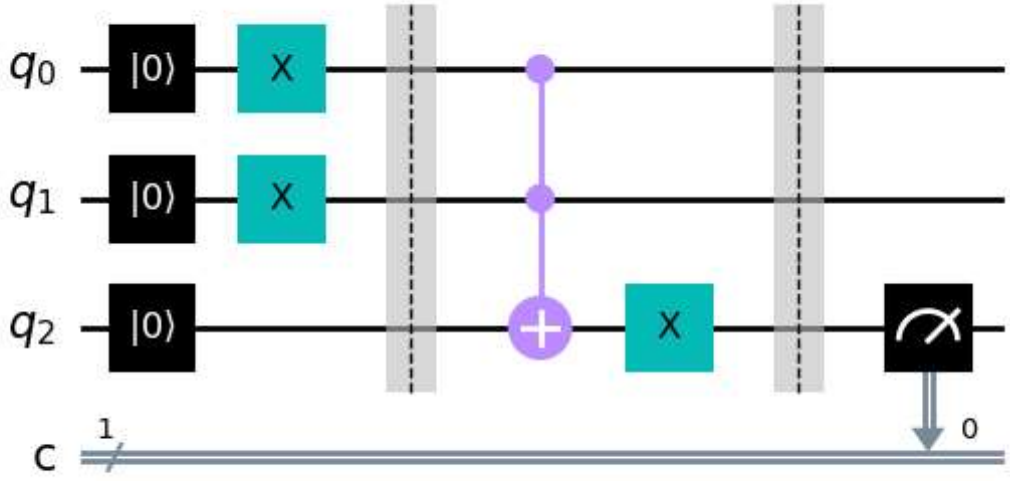

```
NAND with inputs 0 1 gives output 1
```



NAND with inputs 1 0 gives output 1



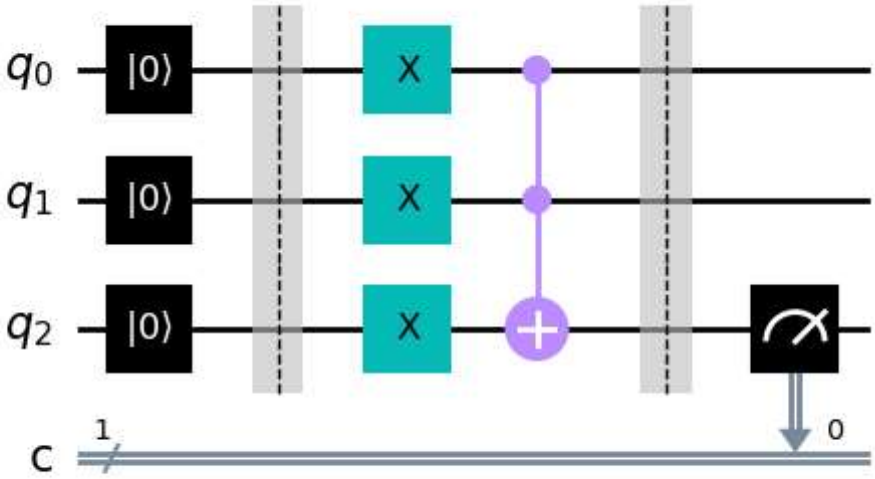
NAND with inputs 1 1 gives output 0



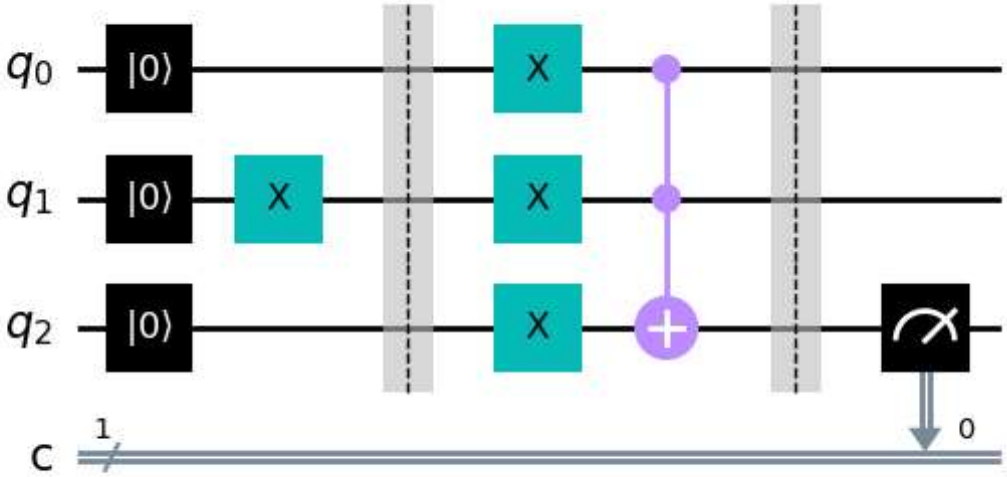
OR

In [90]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = OR(inp1, inp2)
        print('OR with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

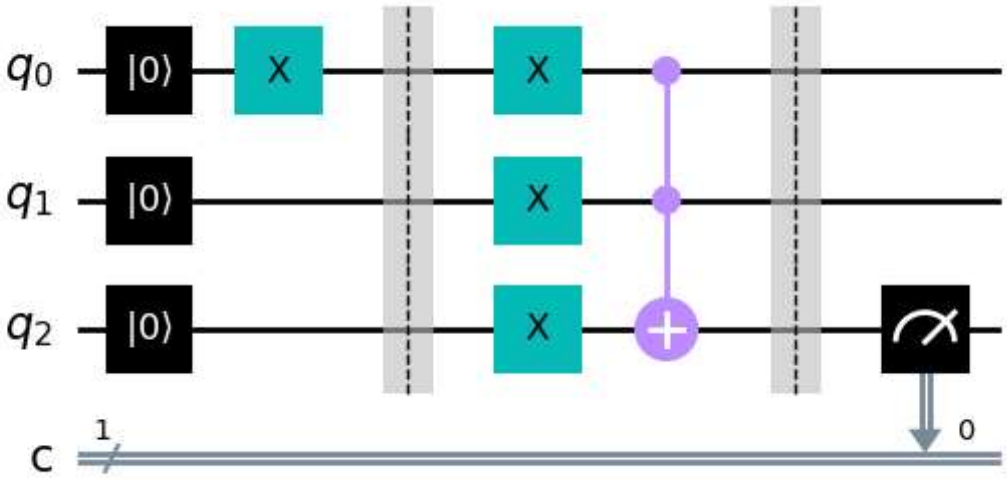
```
OR with inputs 0 0 gives output 0
```



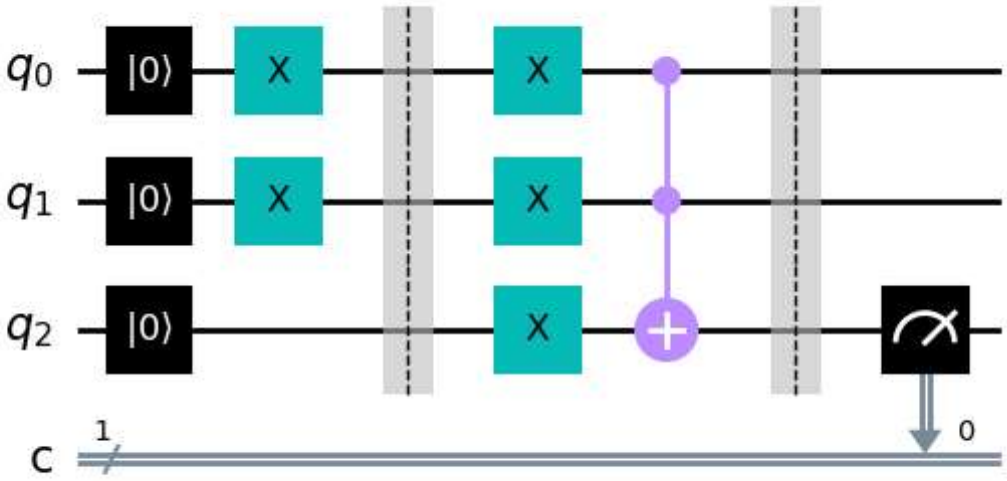

```
OR with inputs 0 1 gives output 1
```




```
OR with inputs 1 0 gives output 1
```



OR with inputs 1 1 gives output 1



Run in quantum System

In [27]:
```
IBMQ.load_account()
```

Out[27]:
```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

In [29]:
```
IBMQ.providers()
provider = IBMQ.get_provider('ibm-q')
provider.backends()
```

Out[29]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', pro
ject='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')
>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open', p
roject='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open', project
='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q', group
='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open', pr
oject='main')>,
 <IBMQBackend('ibmq_jakarta') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibm_lagos') from IBMQ(hub='ibm-q', group='open', project='main')
>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibm_perth') from IBMQ(hub='ibm-q', group='open', project='main')
>]
```

In [33]:
```
import qiskit.tools.jupyter

backend_ex = provider.get_backend('ibm_lagos')
backend_ex
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;pa
dding-top: 1%;padding-bottom: 1…
```

Out[33]:
```
<IBMQBackend('ibm_lagos') from IBMQ(hub='ibm-q', group='open', project='main')>
```

In [34]:
```
backends = provider.backends(filters = lambda x:x.configuration().n_qubits >= 2
                             and x.status().operational==True)
backends
```

Out[34]:
```
[<IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')
>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibmq_jakarta') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibm_lagos') from IBMQ(hub='ibm-q', group='open', project='main')
>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibm_perth') from IBMQ(hub='ibm-q', group='open', project='main')
>]
```

In [91]:
```
from qiskit.providers.ibmq import least_busy
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qub
```

```
                                                not x.configuration().simulator and x.st
backend
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;pa
dding-top: 1%;padding-bottom: 1…
```

Out[91]:
```
<IBMQBackend('ibm_lagos') from IBMQ(hub='ibm-q', group='open', project='main')>
```

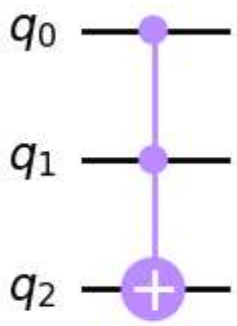
In [93]:
```
# run this cell
backend = provider.get_backend('ibm_lagos')
backend
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;pa
dding-top: 1%;padding-bottom: 1…
```

Out[93]:
```
<IBMQBackend('ibm_lagos') from IBMQ(hub='ibm-q', group='open', project='main')>
```

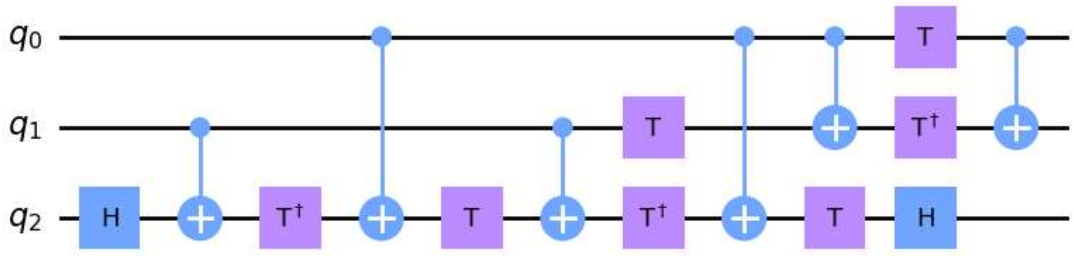
In [95]:
```
qc_and = QuantumCircuit(3)
qc_and.ccx(0,1,2)
print('AND gate')
display(qc_and.draw())
print('\n\nTranspiled AND gate with all the required connectivity')
qc_and.decompose().draw()
```

```
AND gate
```




```
Transpiled AND gate with all the required connectivity
```

Out[95]:



In [39]:
```
from qiskit.tools.monitor import job_monitor
```

In [96]:
```
# run the cell to define AND gate for real quantum system

def AND(inp1, inp2, backend, layout):

    qc = QuantumCircuit(3, 1)
    qc.reset(range(3))
```

```
    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()
    qc.ccx(0, 1, 2)
    qc.barrier()
    qc.measure(2, 0)

    qc_trans = transpile(qc, backend, initial_layout=layout, optimization_level=
    job = backend.run(qc_trans, shots=8192)
    print(job.job_id())
    job_monitor(job)

    output = job.result().get_counts()

    return qc_trans, output
```

In [99]:
```
AND(1,1,backend,[0,1,2])
```

```
ck1aj0fp8blo0kjfv76g
Job Status: job has successfully run
```

Out[99]:
```
(<qiskit.circuit.quantumcircuit.QuantumCircuit at 0x7fa268348100>,
 {'0': 2156, '1': 6036})
```

In [1]:
```
2156/8192
```

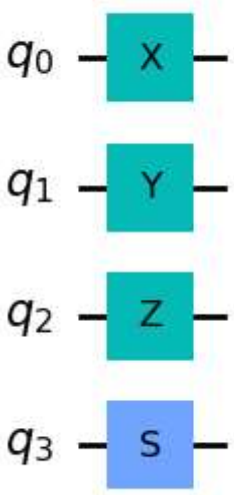
Out[1]:
```
0.26318359375
```

LAB 2

In [102…
```
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, execute
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from qiskit.providers.aer import QasmSimulator

backend = Aer.get_backend('statevector_simulator')
```

In [103…
```
qc1 = QuantumCircuit(4)

# perform gate operations on individual qubits
qc1.x(0)
qc1.y(1)
qc1.z(2)
qc1.s(3)

# Draw circuit
qc1.draw()
```

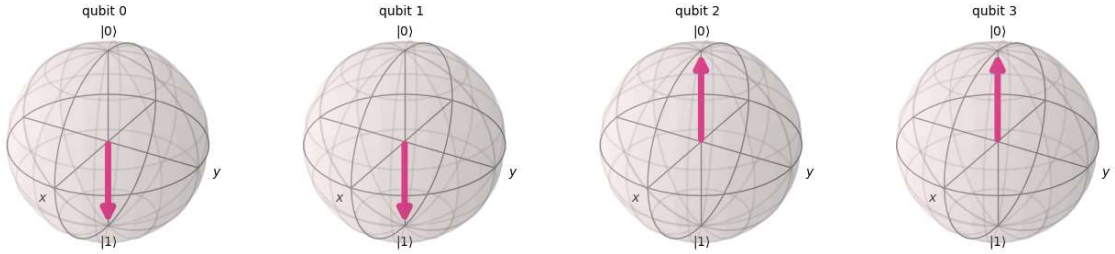
Out[103]:



In [104…

```
# Plot blochshere
out1 = execute(qc1,backend).result().get_statevector()
plot_bloch_multivector(out1)
```

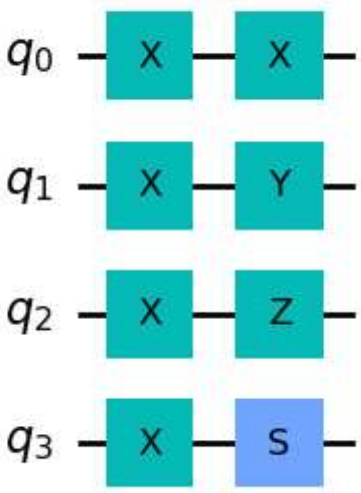
Out[104]:



In [105…

```
qc2 = QuantumCircuit(4)

# initialize qubits
qc2.x(range(4))

# perform gate operations on individual qubits
qc2.x(0)
qc2.y(1)
qc2.z(2)
qc2.s(3)

# Draw circuit
qc2.draw()
```

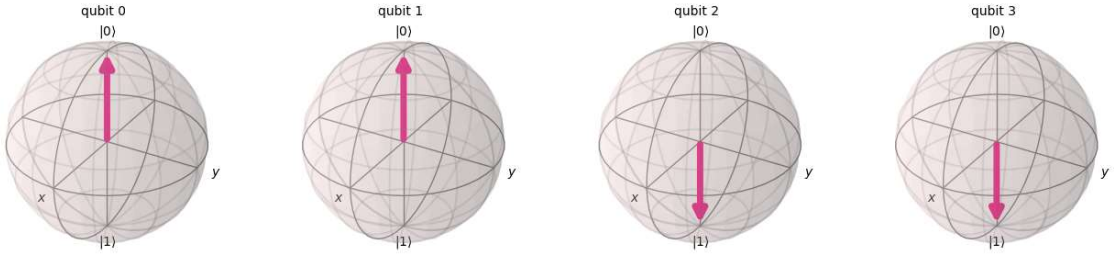
Out[105]:



In [106…

```
# Plot blochshere
out2 = execute(qc2,backend).result().get_statevector()
plot_bloch_multivector(out2)
```

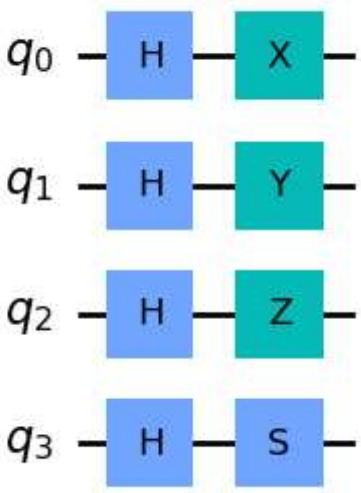
Out[106]:



In [107…

```
qc3 = QuantumCircuit(4)

# initialize qubits
qc3.h(range(4))

# perform gate operations on individual qubits
qc3.x(0)
qc3.y(1)
qc3.z(2)
qc3.s(3)

# Draw circuit
qc3.draw()
```

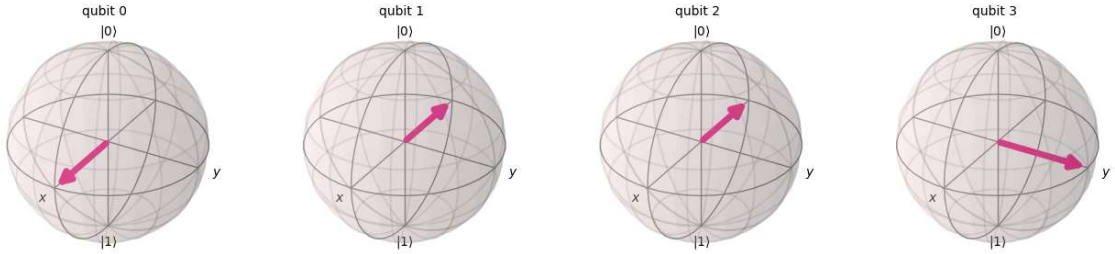
Out[107]:



In [108…

```
# Plot blochshere
out3 = execute(qc3,backend).result().get_statevector()
plot_bloch_multivector(out3)
```

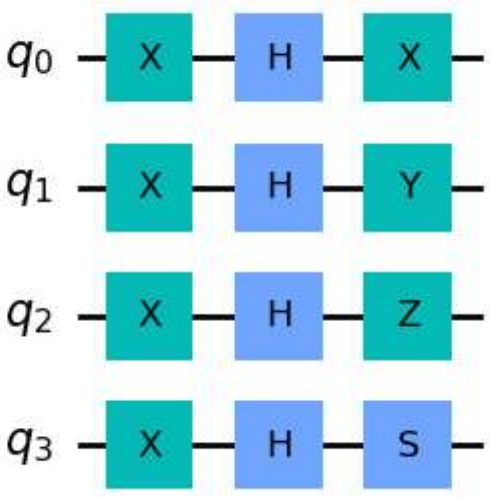
Out[108]:



In [109…

```
qc4 = QuantumCircuit(4)

# initialize qubits
qc4.x(range(4))
qc4.h(range(4))

# perform gate operations on individual qubits
qc4.x(0)
qc4.y(1)
qc4.z(2)
qc4.s(3)

# Draw circuit
qc4.draw()
```

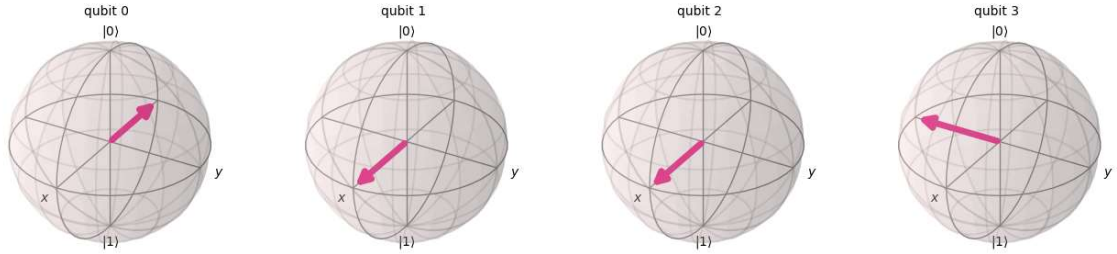
Out[109]:



In [110…

```
# Plot blochshere
out4 = execute(qc4,backend).result().get_statevector()
plot_bloch_multivector(out4)
```

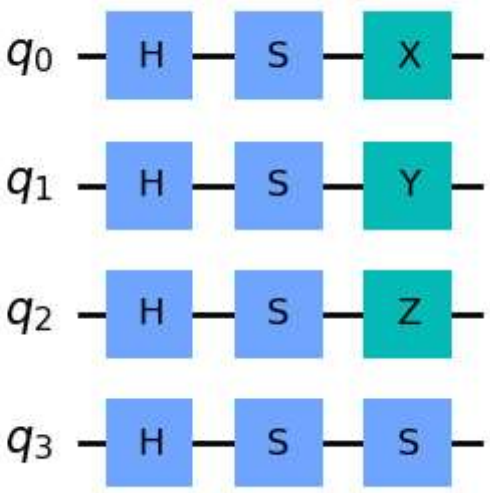
Out[110]:



In [111…

```
qc5 = QuantumCircuit(4)

# initialize qubits
qc5.h(range(4))
qc5.s(range(4))

# perform gate operations on individual qubits
qc5.x(0)
qc5.y(1)
qc5.z(2)
qc5.s(3)

# Draw circuit
qc5.draw()
```

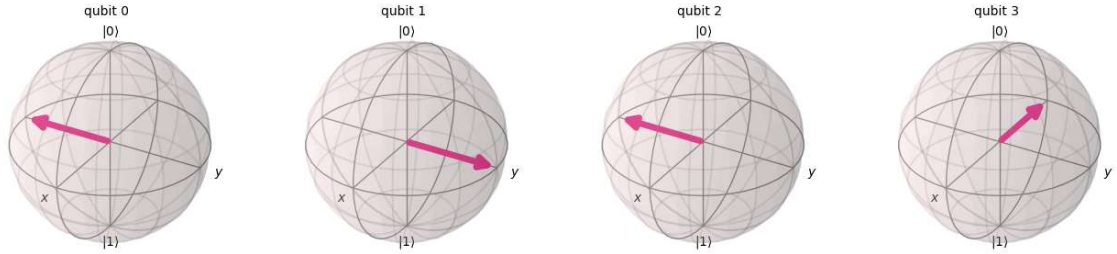
Out[111]:



In [112…

```
# Plot blochshere
out5 = execute(qc5,backend).result().get_statevector()
plot_bloch_multivector(out5)
```

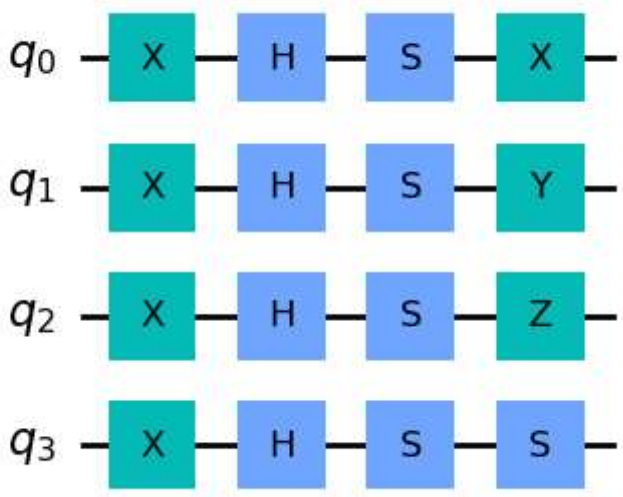
Out[112]:



In [113…

```
qc6 = QuantumCircuit(4)

# initialize qubits
qc6.x(range(4))
qc6.h(range(4))
qc6.s(range(4))

# perform gate operations on individual qubits
qc6.x(0)
qc6.y(1)
qc6.z(2)
qc6.s(3)

# Draw circuit
qc6.draw()
```

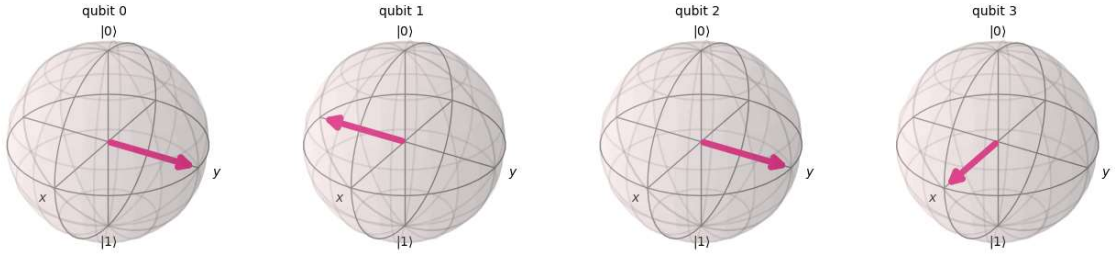
Out[113]:



In [114…

```
# Plot blochshere
out6 = execute(qc6,backend).result().get_statevector()
plot_bloch_multivector(out6)
```

Out[114]:



lab 3

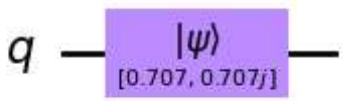
In [5]:

```
from qiskit import *
import numpy as np
from numpy import linalg as la
from qiskit.tools.monitor import job_monitor
import qiskit.tools.jupyter
```

In [15]:

```
qc = QuantumCircuit(1)

#### your code goes here
alpha = 1/np.sqrt(2) #np.random.rand() + 1j * np.random.rand()
beta =1j/np.sqrt(2) #np.random.rand() + 1j * np.random.rand()

vector = [alpha, beta]
qc.initialize(vector, 0)

qc.draw('mpl')
```

Out[15]:



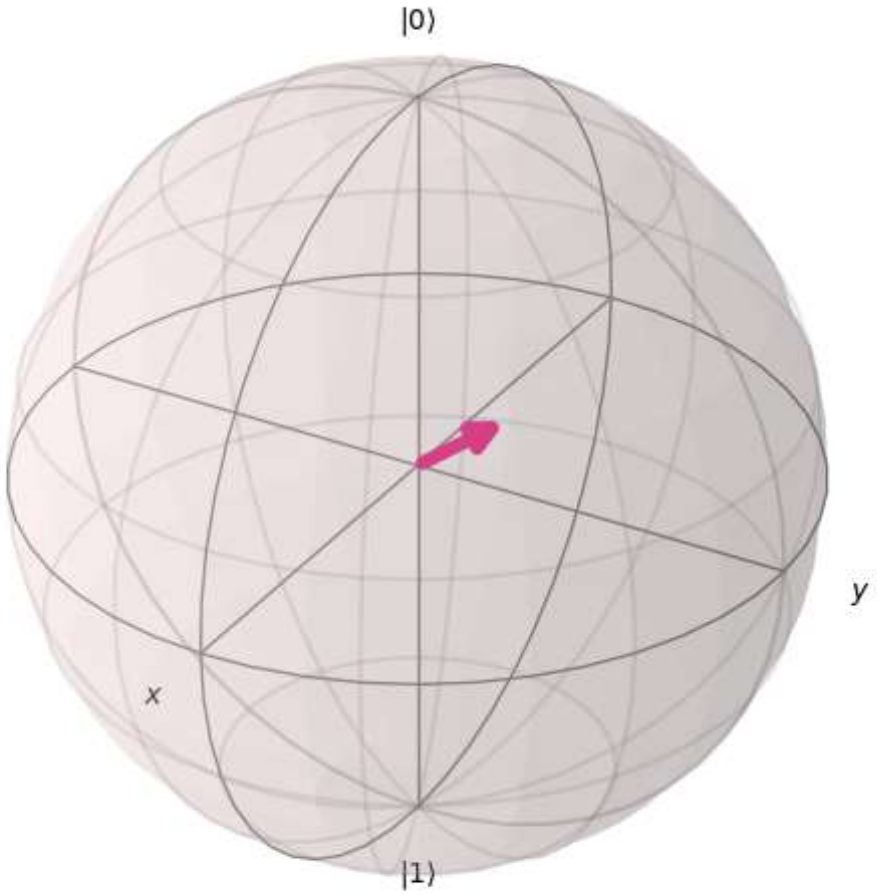
In [18]:
```
# z measurement of qubit 0
measure_z = QuantumCircuit(1,1)
measure_z.measure(0,0)

# x measurement of qubit 0
measure_x = QuantumCircuit(1,1)
# your code goes here
measure_x.measure(0,0)









# y measurement of qubit 0
measure_y = QuantumCircuit(1,1)
# your code goes here
measure_y.measure(0,0)









shots = 2**14 # number of samples used for statistics
sim = Aer.get_backend('qasm_simulator')
bloch_vector_measure = []
for measure_circuit in [measure_x, measure_y, measure_z]:

    # run the circuit with a the selected measurement and get the number of samp
    counts = execute(measure_circuit, sim, shots=shots).result().get_counts()

    # calculate the probabilities for each bit value
    probs = {}
    for output in ['0','1']:
        if output in counts:
            probs[output] = counts[output]/shots
        else:
            probs[output] = 0

    bloch_vector_measure.append( probs['0'] -  probs['1'] )

# normalizing the bloch sphere vector
bloch_vector = bloch_vector_measure/la.norm(bloch_vector_measure)

print('The bloch sphere coordinates are [{0:4.3f}, {1:4.3f}, {2:4.3f}]'
      .format(*bloch_vector))
```

```
The bloch sphere coordinates are [0.577, 0.577, 0.577]
```

In [19]:
```
from kaleidoscope.interactive import bloch_sphere

bloch_sphere(bloch_vector, vectors_annotation=True)
```

In [20]:
```
from qiskit.visualization import plot_bloch_vector

plot_bloch_vector( bloch_vector )
```

Out[20]:



In [27]:
```
# circuit for the state Tri1
Tri1 = QuantumCircuit(2)
# your code goes here
```

```
Tri1 = QuantumCircuit(2)
Tri1.h(0)  # Apply a Hadamard gate on qubit 0
Tri1.cx(0, 1)
#Tri1.measure_all()


# circuit for the state Tri2
Tri2 = QuantumCircuit(2)
# your code goes here
Tri2.h(0)  # Apply a Hadamard gate on qubit 0
Tri2.cx(0, 1)  # Apply a CNOT gate with qubit 0 as the control and qubit 1 as th
Tri2.z(0)
#Tri2.measure_all()


# circuit for the state Tri3
Tri3 = QuantumCircuit(2)
# your code goes here
Tri3.h(0)  # Apply a Hadamard gate on qubit 0
Tri3.cx(0, 1)  # Apply a CNOT gate with qubit 0 as the control and qubit 1 as th
Tri3.x(0)
#Tri3.measure_all()



# circuit for the state Sing
Sing = QuantumCircuit(2)
# your code goes here
Sing.h(0)  # Apply a Hadamard gate on qubit 0
Sing.cx(0, 1)  # Apply a CNOT gate with qubit 0 as the control and qubit 1 as th
Sing.x(0)  # Apply an X gate to qubit 0 (this changes 0 to 1 and 1 to 0)
Sing.z(0)
#Sing.measure_all()
```

In [23]:
```
# <ZZ>
measure_ZZ = QuantumCircuit(2)
measure_ZZ.measure_all()

# <XX>
measure_XX = QuantumCircuit(2)
# your code goes here
measure_XX.h(0)  # Apply a Hadamard gate to qubit 0
measure_XX.h(1)  # Apply a Hadamard gate to qubit 1
measure_XX.measure_all()

# <YY>
measure_YY = QuantumCircuit(2)
# your code goes here
measure_YY.h(0)  # Apply a Hadamard gate to qubit 0
measure_YY.h(1)  # Apply a Hadamard gate to qubit 1
measure_YY.sdg(0)  # Apply the conjugate of the S gate to qubit 0
measure_YY.sdg(1)  # Apply the conjugate of the S gate to qubit 1
measure_YY.measure_all()
```

In [38]:
```
shots = 2**14 # number of samples used for statistics

A = 1.47e-6 #unit of A is eV
E_sim = []
for state_init in [Tri1,Tri2,Tri3,Sing]:
```

```
    Energy_meas = []
    for measure_circuit in [measure_XX, measure_YY, measure_ZZ]:

        # run the circuit with a the selected measurement and get the number of
        qc = state_init.compose(measure_circuit)
        counts = execute(qc, sim, shots=shots).result().get_counts()

        # calculate the probabilities for each computational basis
        probs = {}
        for output in ['00','01', '10', '11']:
            if output in counts:
                probs[output] = counts[output]/shots
            else:
                probs[output] = 0

        Energy_meas.append( probs['00'] - probs['01'] - probs['10'] + probs['11'

    E_sim.append(A * np.sum(np.array(Energy_meas)))
```

In [43]:
```
# Run this cell to print out your results

print('Energy expection value of the state Tri1 : {:.3e} eV'.format(E_sim[0]))
print('Energy expection value of the state Tri2 : {:.3e} eV'.format(E_sim[1]))
print('Energy expection value of the state Tri3 : {:.3e} eV'.format(E_sim[2]))
print('Energy expection value of the state Sing : {:.3e} eV'.format(E_sim[3]))
```

```
Energy expection value of the state Tri1 : 1.453e-06 eV
Energy expection value of the state Tri2 : 1.479e-06 eV
Energy expection value of the state Tri3 : -1.470e-06 eV
Energy expection value of the state Sing : -1.485e-06 eV
```

In [44]:
```
# reduced plank constant in (eV) and the speed of light(cgs units)
hbar, c = 4.1357e-15, 3e10

# energy difference between the triplets and singlet
E_del = abs(E_sim[0] - E_sim[3])

# frequency associated with the energy difference
f = E_del/hbar

# convert frequency to wavelength in (cm)
wavelength = c/f

print('The wavelength of the radiation from the transition\
 in the hyperfine structure is : {:.1f} cm'.format(wavelength))
```

```
The wavelength of the radiation from the transition in the hyperfine structure
is : 42.2 cm
```

In [33]:
```
provider = IBMQ.load_account()
```

```
/tmp/ipykernel_59/2020123530.py:1: DeprecationWarning:

The qiskit.IBMQ entrypoint and the qiskit-ibmq-provider package (accessible fro
m 'qiskit.providers.ibmq`) are deprecated and will be removed in a future relea
se. Instead you should use the qiskit-ibm-provider package which is accessible
from 'qiskit_ibm_provider'. You can install it with 'pip install qiskit_ibm_pro
vider'. Just replace 'qiskit.IBMQ' with 'qiskit_ibm_provider.IBMProvider'
```

In [45]:
```
from qiskit.providers.ibmq import least_busy
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qub
                                       not x.configuration().simulator and x.st
backend
```

Out[45]:
```
<IBMQBackend('ibm_perth') from IBMQ(hub='ibm-q', group='open', project='main')>
```

In [54]:
```
backend = provider.get_backend('ibm_perth')

# assign your choice for the initial layout to the list variable `initial_layout
initial_layout =[3,5]
```

In [55]:
```
qc_all = [state_init.compose(measure_circuit) for state_init in [Tri1,Tri2,Tri3,
          for measure_circuit in [measure_XX, measure_YY, measure_ZZ]]

shots = 8192
job = execute(qc_all, backend, initial_layout=initial_layout, optimization_level
print(job.job_id())
job_monitor(job)
```

```
ck1i1o0cal0hh8h4al2g
Job Status: job is queued (None)
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[55], line 7
      5 job = execute(qc_all, backend, initial_layout=initial_layout, optimizat
ion_level=3, shots=shots)
      6 print(job.job_id())
----> 7 job_monitor(job)

File /opt/conda/lib/python3.10/site-packages/qiskit/tools/monitor/job_monitor.p
y:105, in job_monitor(job, interval, quiet, output, line_discipline)
    102 else:
    103     _interval_set = True
--> 105 _text_checker(
    106     job, interval, _interval_set, quiet=quiet, output=output, line_disc
ipline=line_discipline
    107 )

File /opt/conda/lib/python3.10/site-packages/qiskit/tools/monitor/job_monitor.p
y:44, in _text_checker(job, interval, _interval_set, quiet, output, line_discip
line)
     42     print("{}{}: {}".format(line_discipline, "Job Status", msg), end
="", file=output)
     43 while status.name not in ["DONE", "CANCELLED", "ERROR"]:
---> 44     time.sleep(interval)
     45     status = job.status()
     46     msg = status.value

KeyboardInterrupt:
```

In [ ]:
```
# getting the results of your job
results = job.result()
```

In [ ]:
```
## To access the results of the completed job
results = backend.retrieve_job('job_id').result()
```

In [ ]:
```
def Energy(results, shots):
    """Compute the energy levels of the hydrogen ground state.

    Parameters:
        results (obj): results, results from executing the circuits for measurin
        shots (int): shots, number of shots used for the circuit execution.

    Returns:
        Energy (list): energy values of the four different hydrogen ground state
    """
    E = []
    A = 1.47e-6

    for ind_state in range(4):
        Energy_meas = []
        for ind_comp in range(3):
            counts = results.get_counts(ind_state*3+ind_comp)

            # calculate the probabilities for each computational basis
            probs = {}
            for output in ['00','01', '10', '11']:
                if output in counts:
                    probs[output] = counts[output]/shots
                else:
                    probs[output] = 0

            Energy_meas.append( probs['00'] - probs['01'] - probs['10'] + probs[

        E.append(A * np.sum(np.array(Energy_meas)))

    return E
```

In [ ]:
```
E = Energy(results, shots)

print('Energy expection value of the state Tri1 : {:.3e} eV'.format(E[0]))
print('Energy expection value of the state Tri2 : {:.3e} eV'.format(E[1]))
print('Energy expection value of the state Tri3 : {:.3e} eV'.format(E[2]))
print('Energy expection value of the state Sing : {:.3e} eV'.format(E[3]))
```

Error mitigation

In [ ]:
```
from qiskit.ignis.mitigation.measurement import *
```

In [ ]:
```
# your code to create the circuits, meas_calibs, goes here
meas_calibs, state_labels =



# execute meas_calibs on your choice of the backend
job = execute(meas_calibs, backend, shots = shots)
print(job.job_id())
job_monitor(job)
cal_results = job.result()
## To access the results of the completed job
#cal_results = backend.retrieve_job('job_id').result()


# your code to obtain the measurement filter object, 'meas_filter', goes here
```

In [ ]:
```
results_new = meas_filter.apply(results)
```

In [ ]:
```
E_new = Energy(results_new, shots)

print('Energy expection value of the state Tri1 : {:.3e} eV'.format(E_new[0]))
print('Energy expection value of the state Tri2 : {:.3e} eV'.format(E_new[1]))
print('Energy expection value of the state Tri3 : {:.3e} eV'.format(E_new[2]))
print('Energy expection value of the state Sing : {:.3e} eV'.format(E_new[3]))
```

In [ ]:
```
# results for the energy estimation from the simulation,
# execution on a quantum system without error mitigation and
# with error mitigation in numpy array format
Energy_exact, Energy_exp_orig, Energy_exp_new = np.array(E_sim), np.array(E), np
```

In [ ]:
```
# Calculate the relative errors of the energy values without error mitigation
# and assign to the numpy array variable `Err_rel_orig` of size 4
Err_rel_orig =
```

In [ ]:
```
np.set_printoptions(precision=3)

print('The relative errors of the energy values for four bell basis\
 without measurement error mitigation : {}'.format(Err_rel_orig))
```

In [ ]:
```
np.set_printoptions(precision=3)

print('The relative errors of the energy values for four bell basis\
 with measurement error mitigation : {}'.format(Err_rel_new))
```

Lab 4

In [56]:
```
# Importing standard Qiskit libraries
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit import Aer, assemble
from qiskit.visualization import plot_histogram, plot_bloch_multivector, plot_st

# Ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

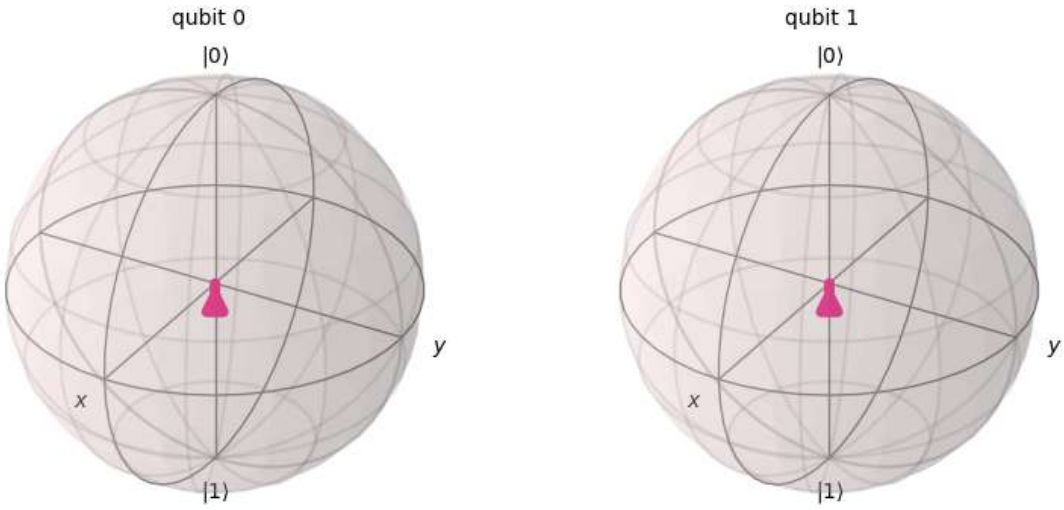
```
# Define backend
sim = Aer.get_backend('aer_simulator')
```

In [57]:
```
def createBellStates(inp1, inp2):
    qc = QuantumCircuit(2)
    qc.reset(range(2))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()

    qc.h(0)
    qc.cx(0,1)

    qc.save_statevector()
    qobj = assemble(qc)
    result = sim.run(qobj).result()
    state = result.get_statevector()

    return qc, state, result
```

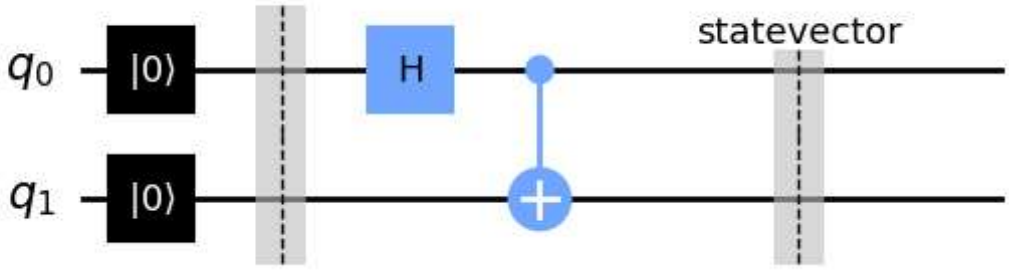
In [58]:
```
print('Note: Since these qubits are in entangled state, their state cannot be wr

inp1 = 0
inp2 = 1

qc, state, result = createBellStates(inp1, inp2)

display(plot_bloch_multivector(state))

# Uncomment below code in order to explore other states
for inp2 in ['0', '1']:
    for inp1 in ['0', '1']:
        qc, state, result = createBellStates(inp1, inp2)

        print('For inputs',inp2,inp1,'Representation of Entangled States are:')

        # Uncomment any of the below functions to visualize the resulting quantu

        # Draw the quantum circuit
        display(qc.draw())

        # Plot states on QSphere
        display(plot_state_qsphere(state))

        # Plot states on Bloch Multivector
        display(plot_bloch_multivector(state))

        # Plot histogram
        display(plot_histogram(result.get_counts()))

        # Plot state matrix like a city
        display(plot_state_city(state))

        # Represent state matix using Pauli operators as the basis
        display(plot_state_paulivec(state))
```

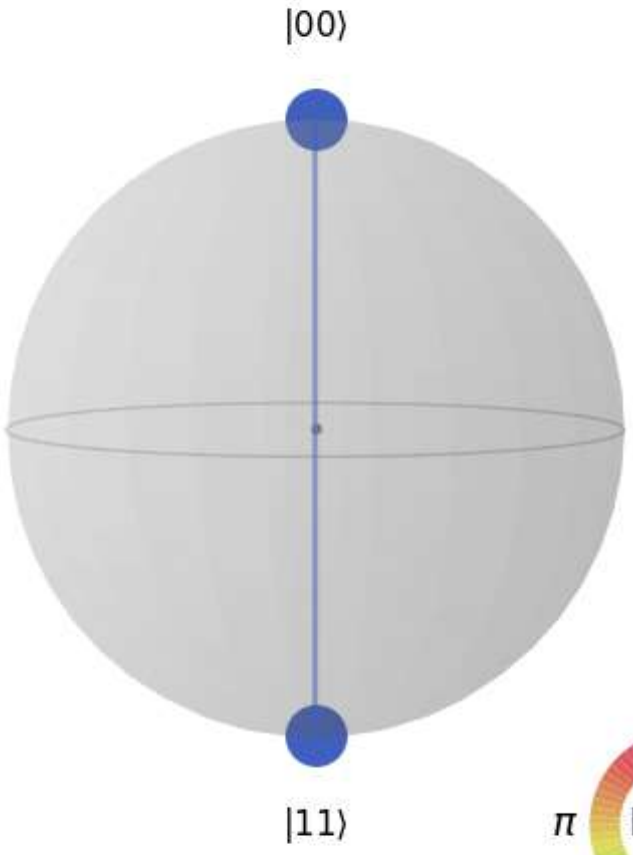
```
        # Plot state matrix as Hinton representation
        display(plot_state_hinton(state))

        #print('\n')'''
```

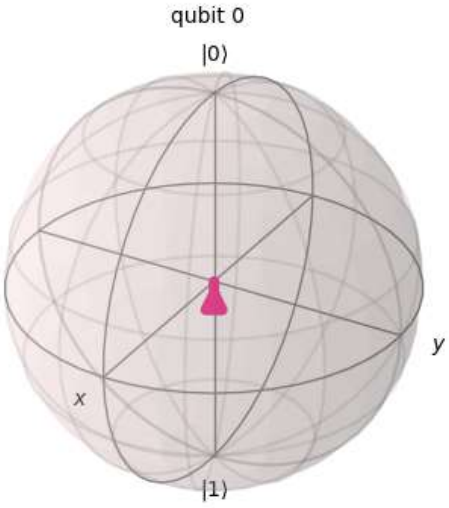
Note: Since these qubits are in entangled state, their state cannot be written as two separate qubit states. This also means that we lose information when we try to plot our state on separate Bloch spheres as seen below.

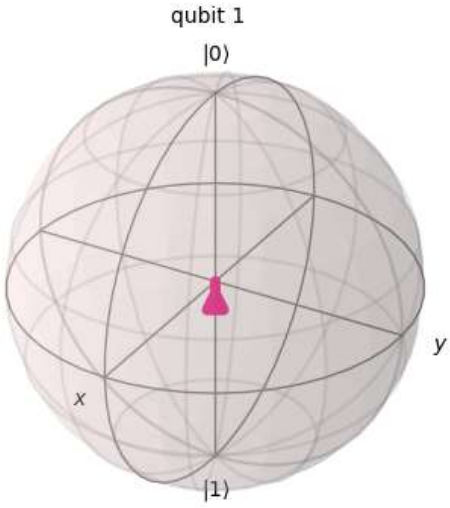


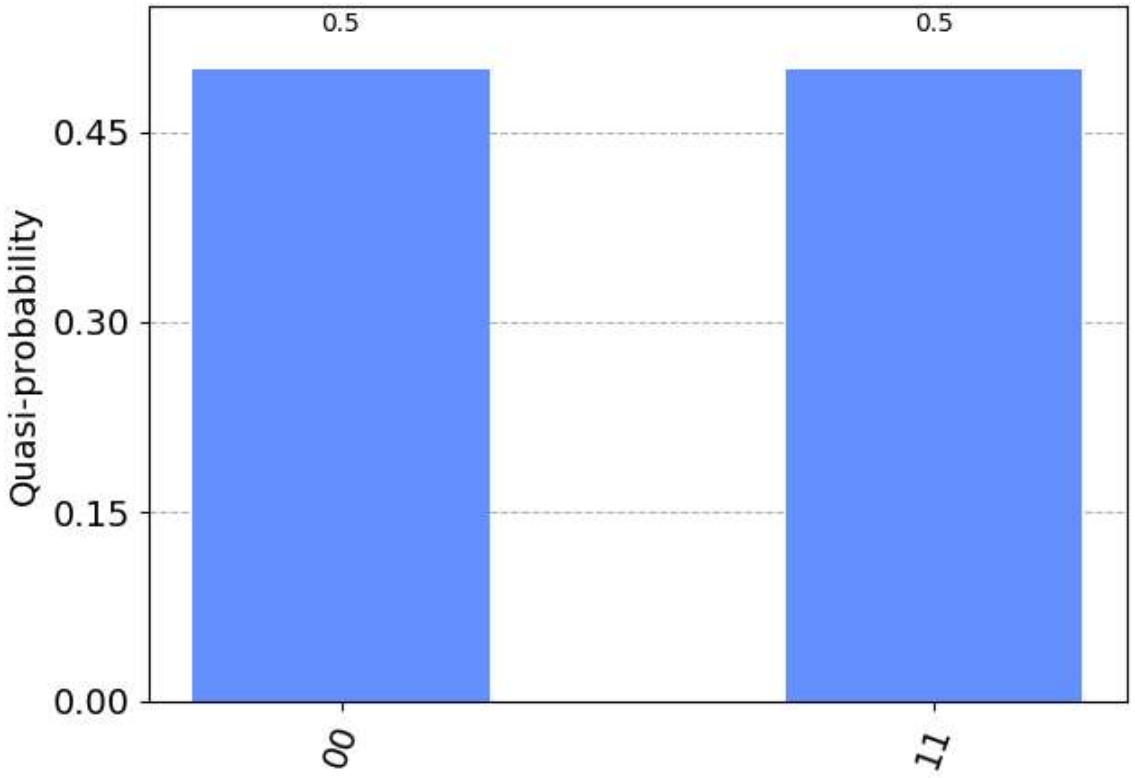
For inputs 0 0 Representation of Entangled States are:

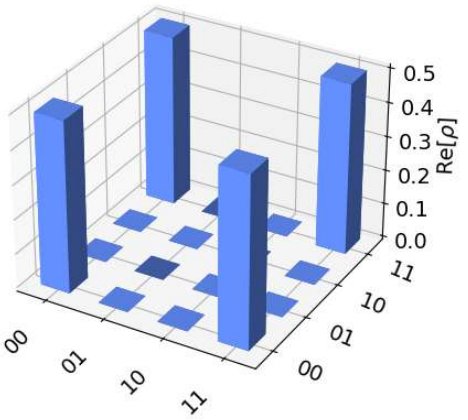


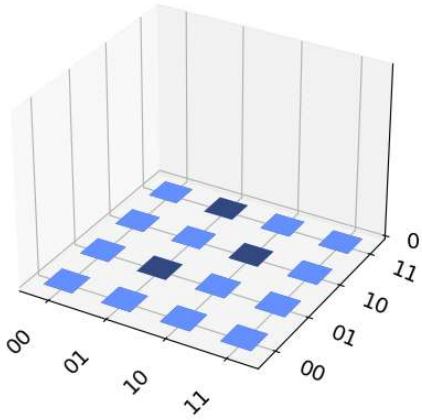

|00⟩
|11⟩
π/2
π
Phase
0
3π/2

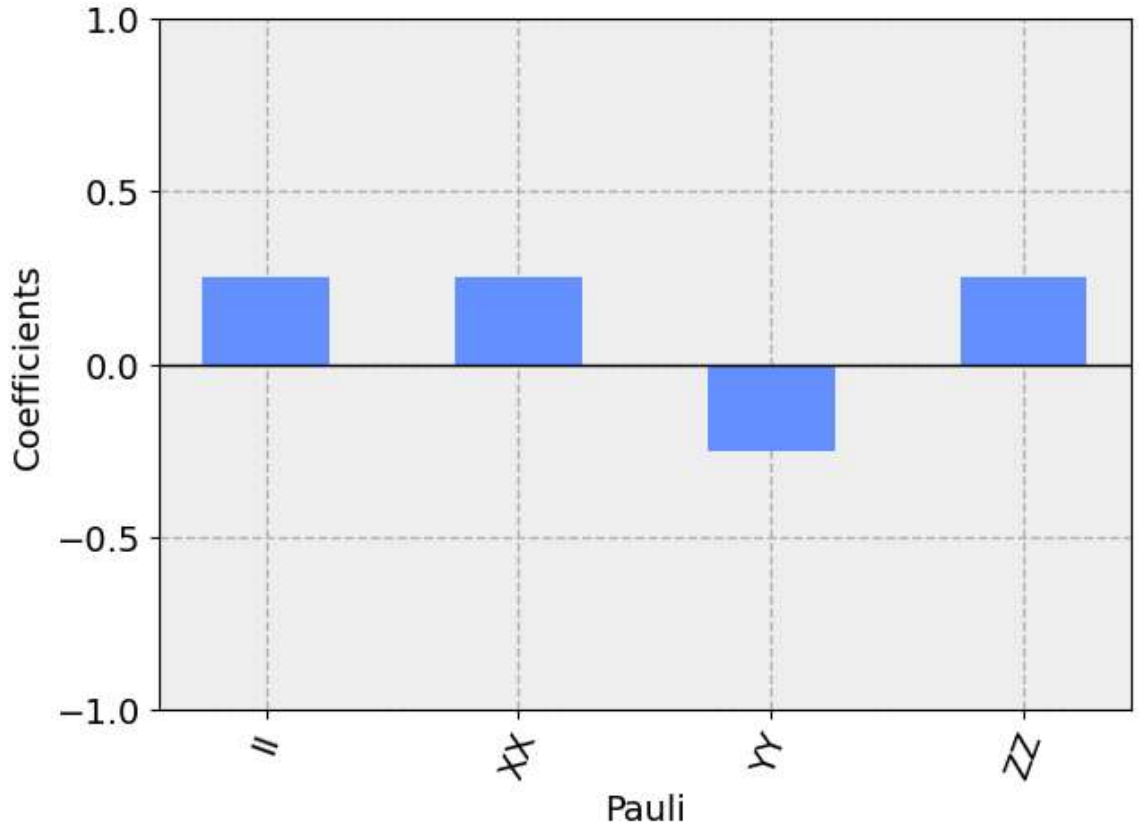

qubit 0
|0⟩
y
x
|1⟩

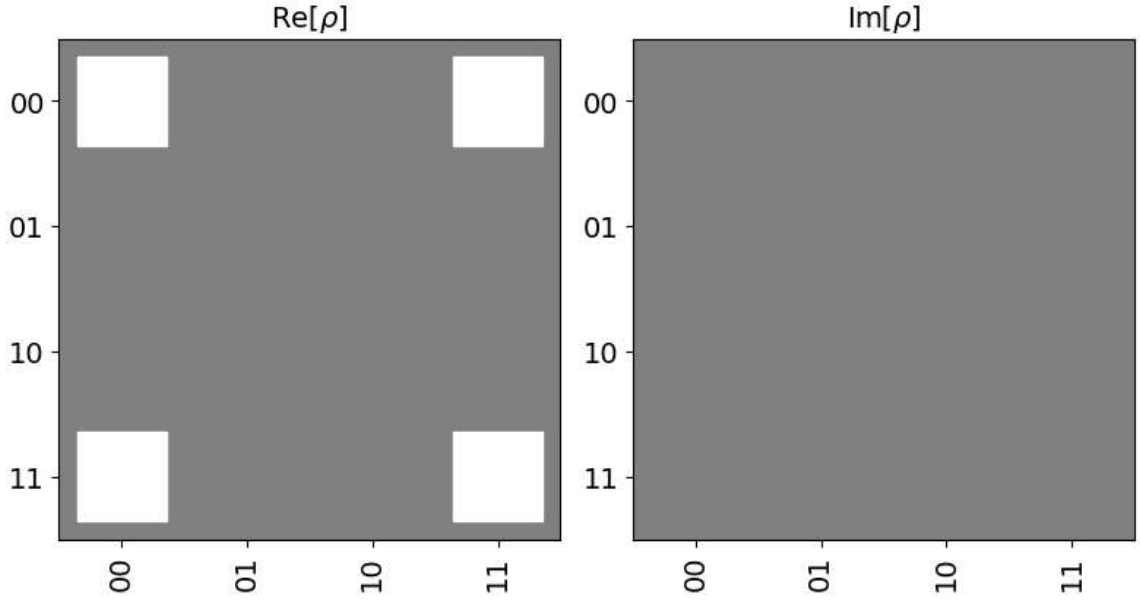

qubit 1
|0⟩
y
x
|1⟩


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00
11

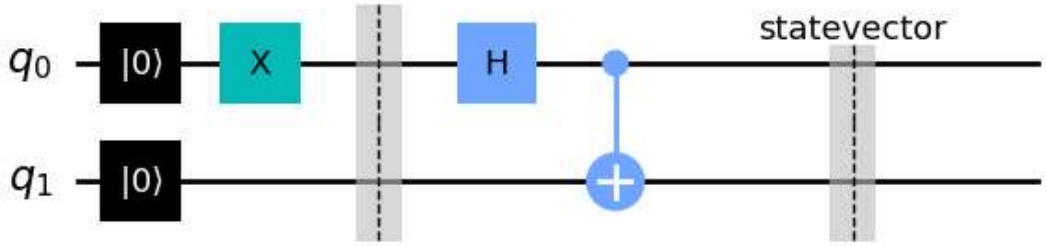

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
00
01
10
11
00
01
10
11

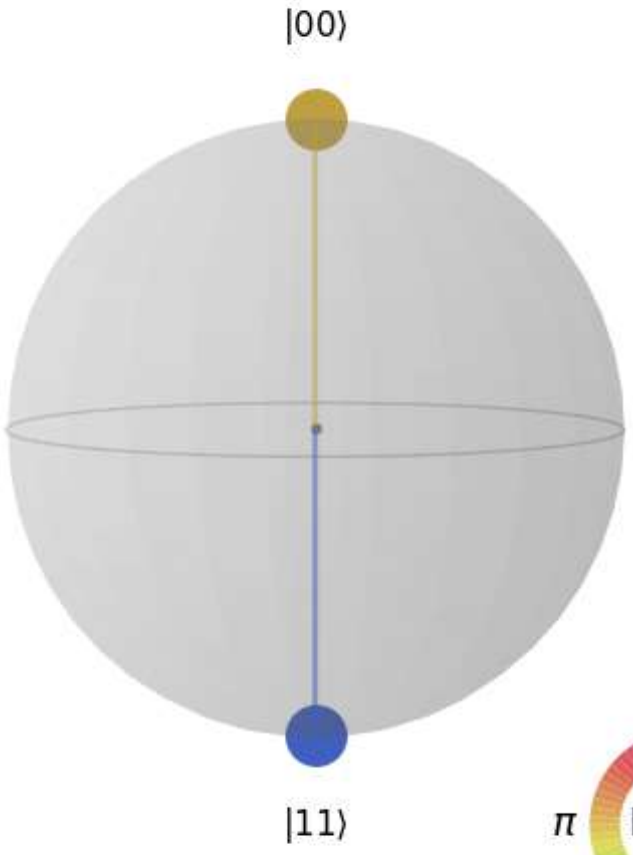

00
01
10
11
00
01
10
11

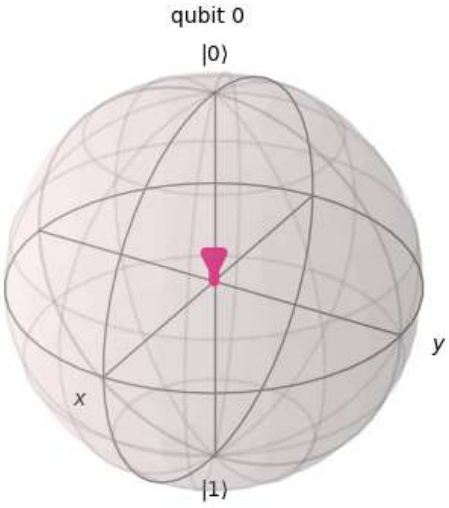


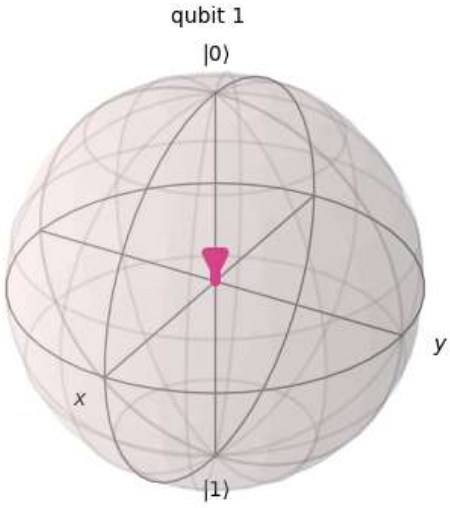


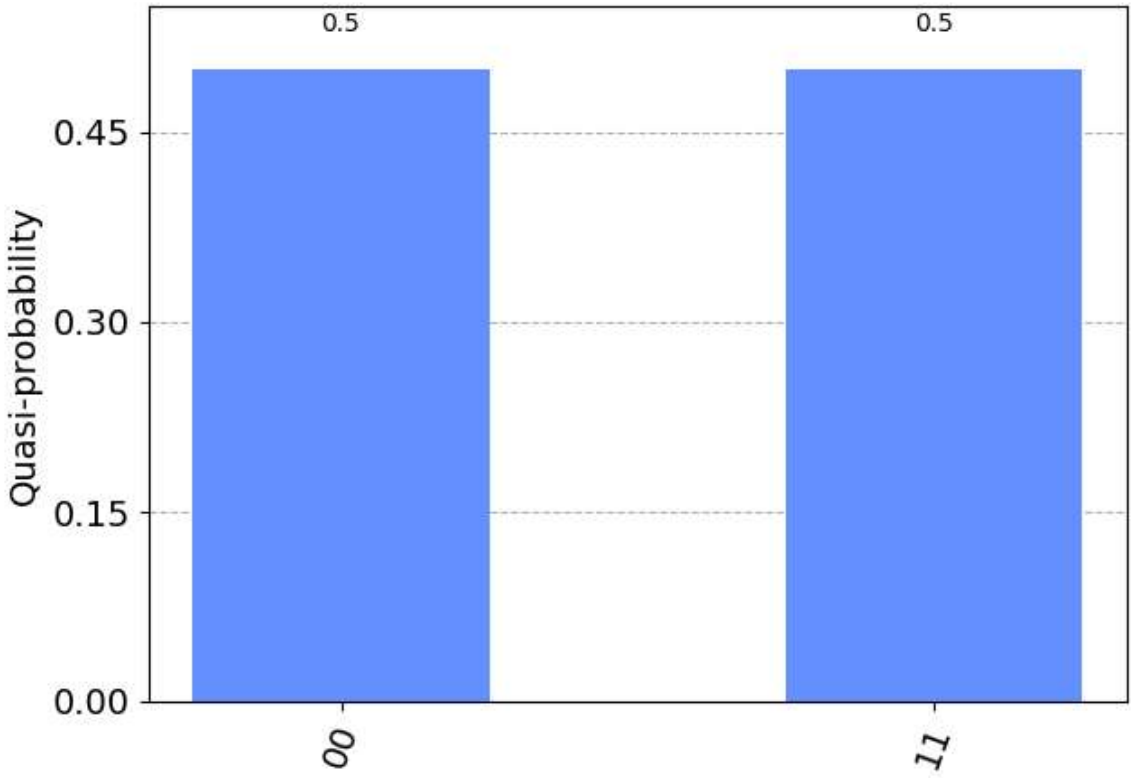
For inputs 0 1 Representation of Entangled States are:

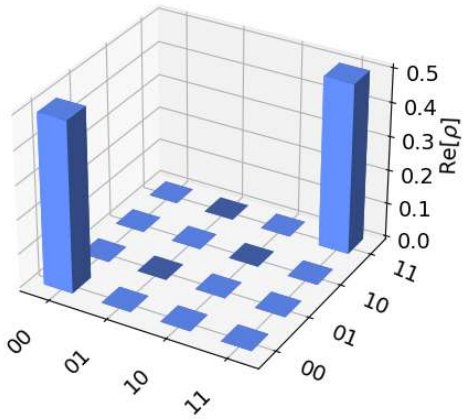


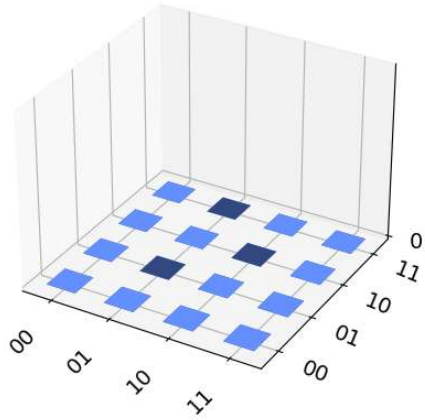

|00⟩
|11⟩
π/2
π
Phase
0
3π/2

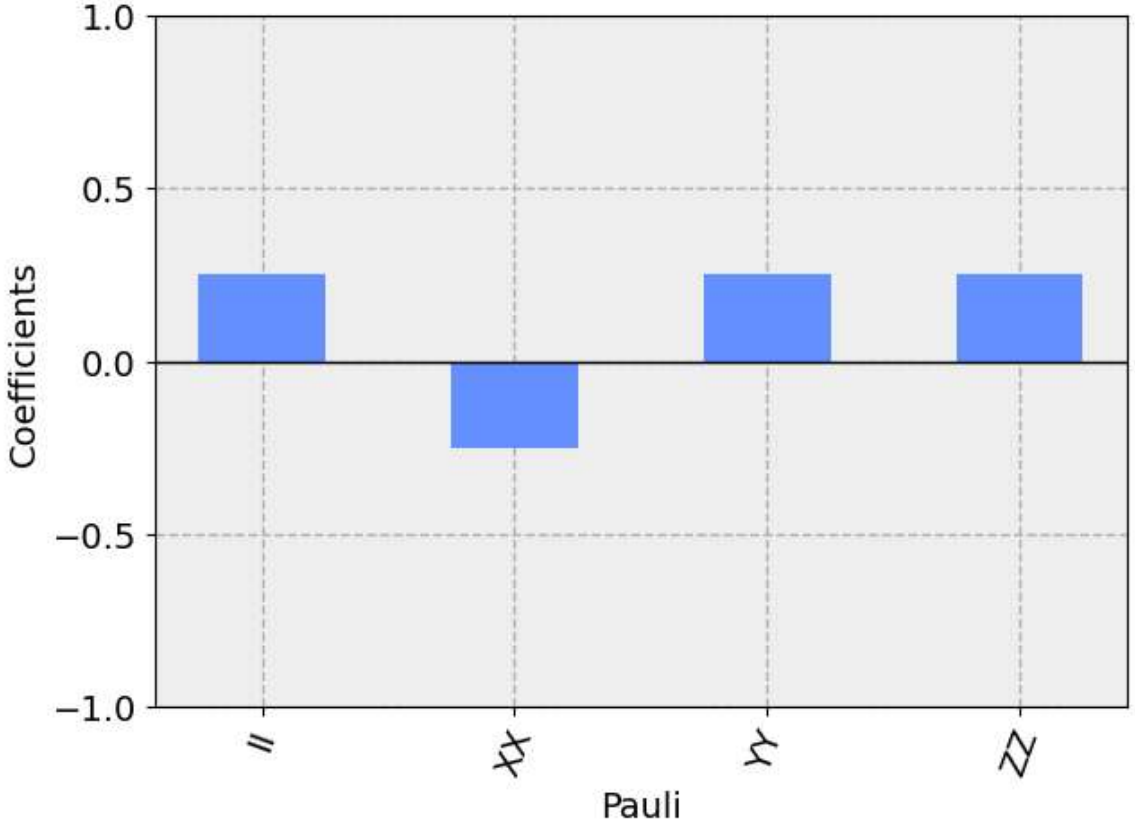

qubit 0
|0⟩
y
x
|1⟩

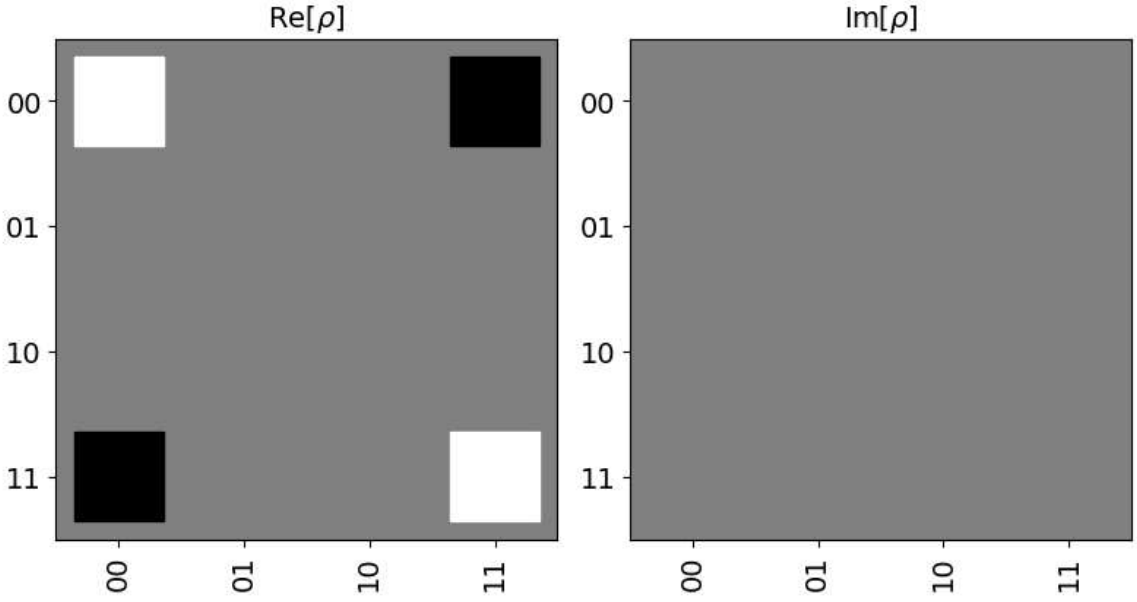

qubit 1
|0⟩
y
x
|1⟩


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00
11

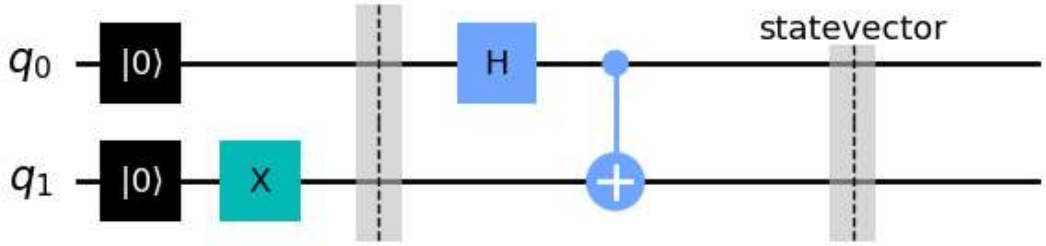

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
00
01
10
11
00
01
10
11

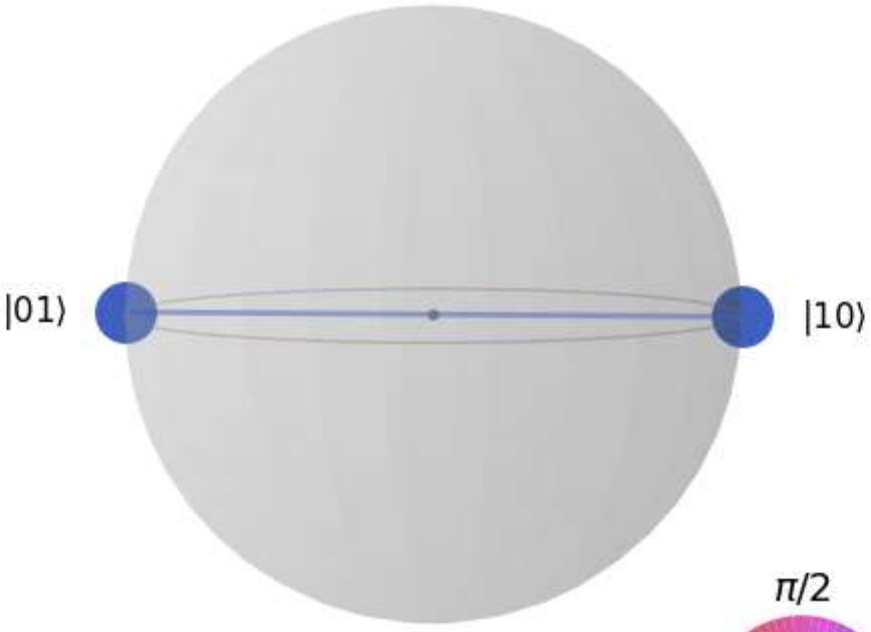

00
01
10
11
00
01
10
11

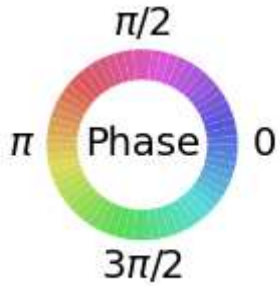


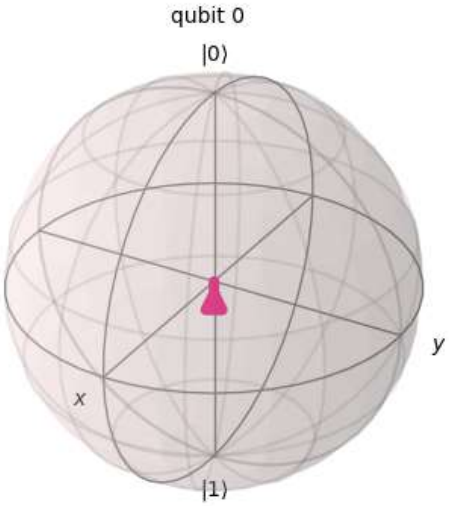


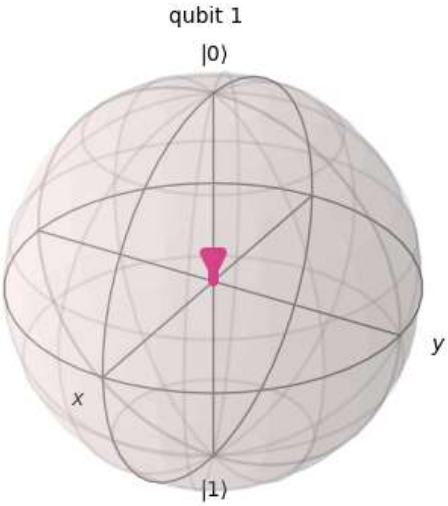
For inputs 1 0 Representation of Entangled States are:

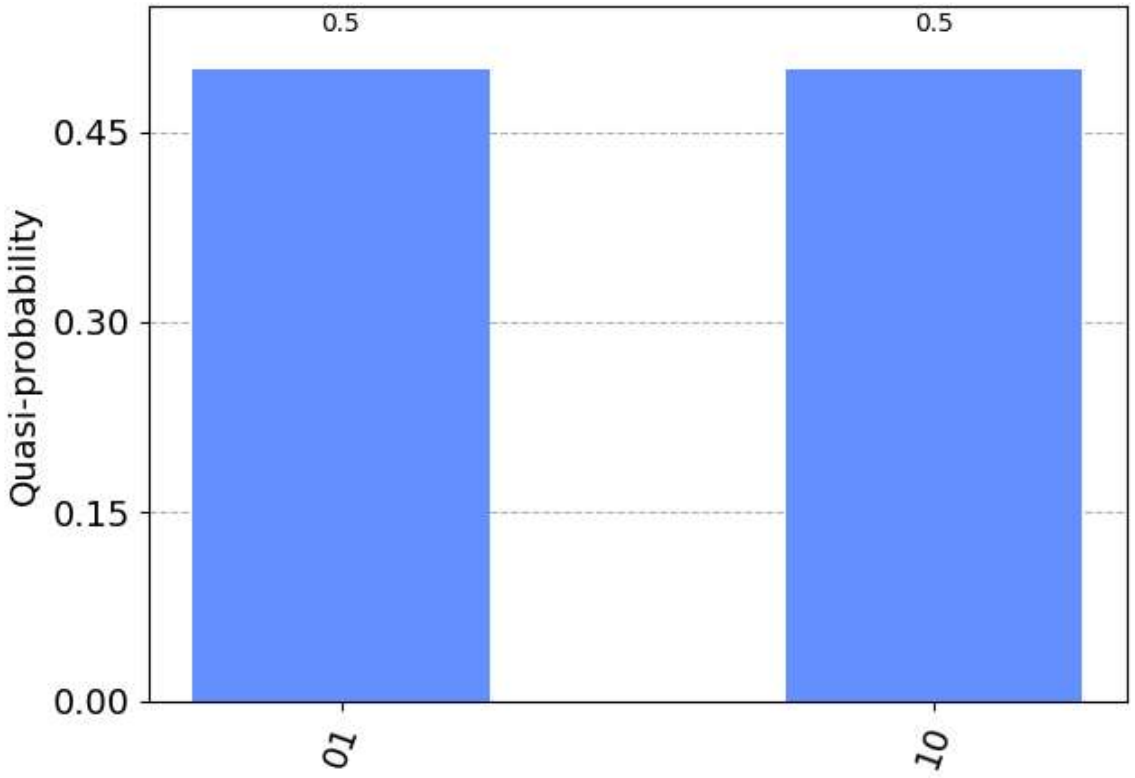


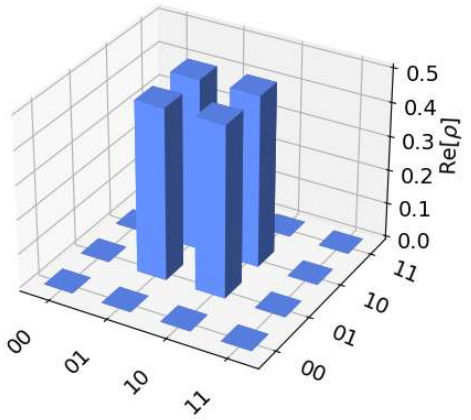

|01⟩
|10⟩

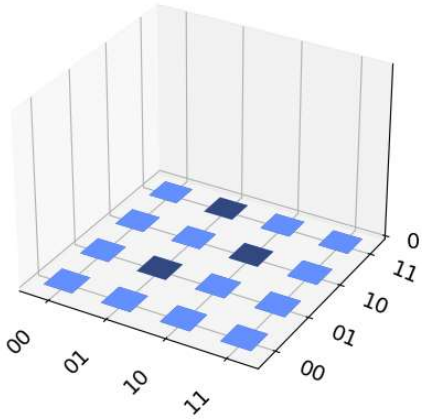

π/2
π
Phase
0
3π/2

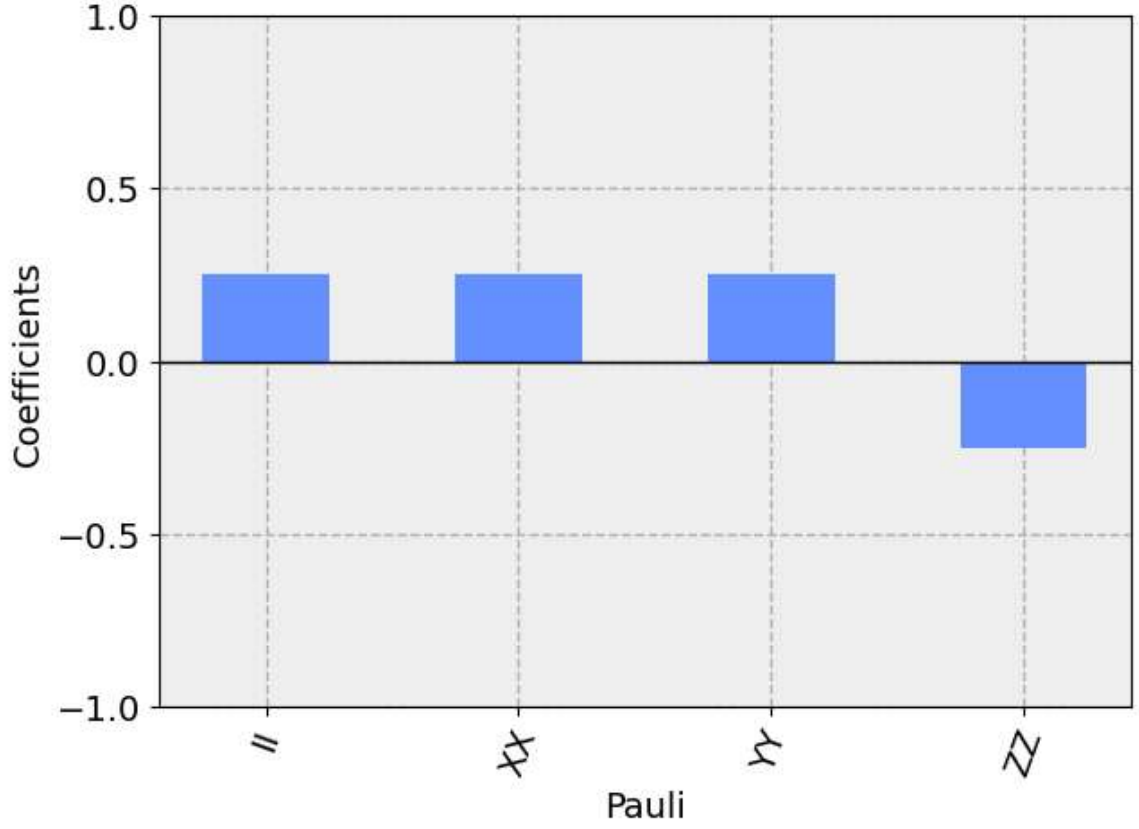

qubit 0
|0⟩
y
x
|1⟩

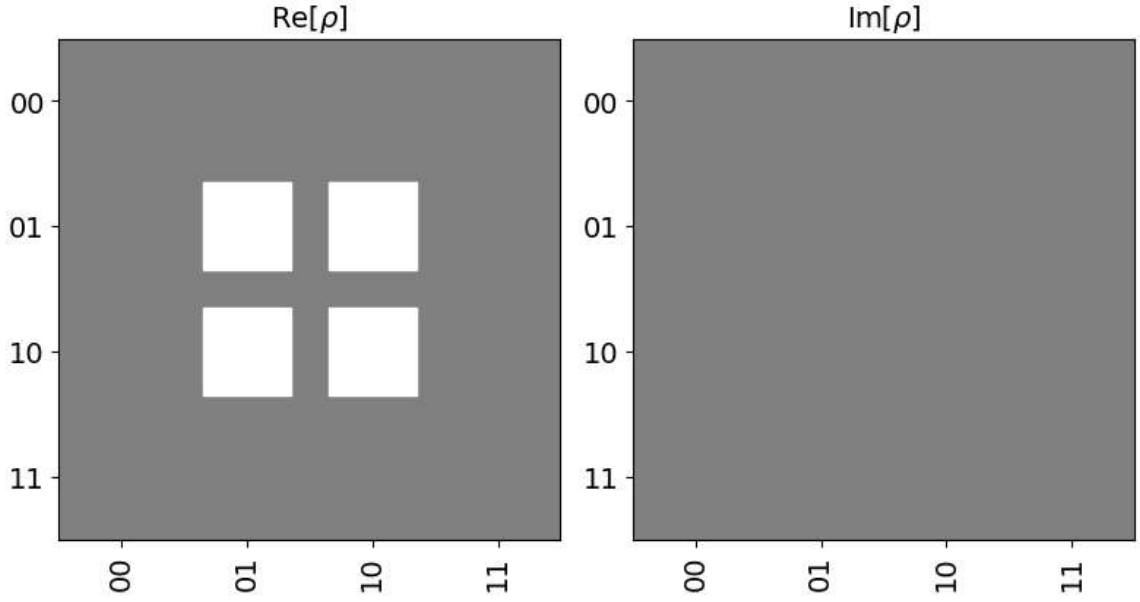

qubit 1
|0⟩
y
x
|1⟩


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
01
10

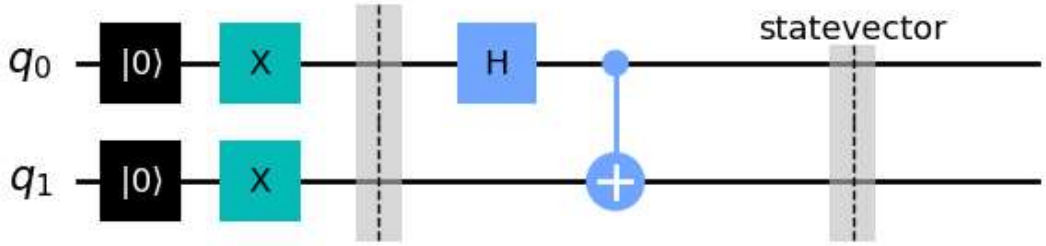

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
00
01
10
11
00
01
10
11

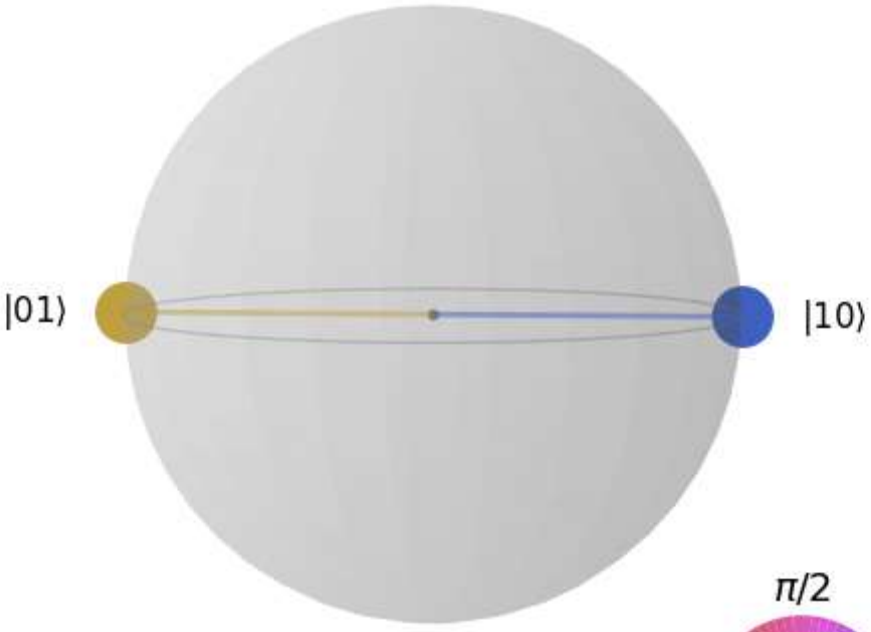

00
01
10
11
00
01
10
11

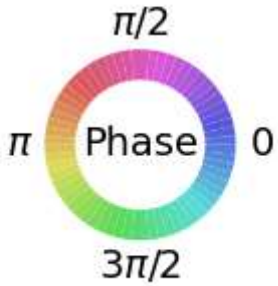


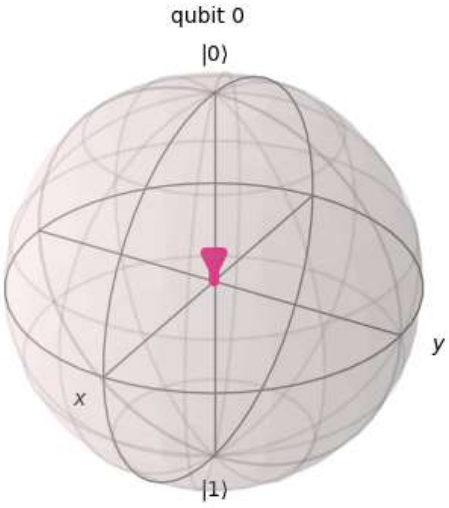


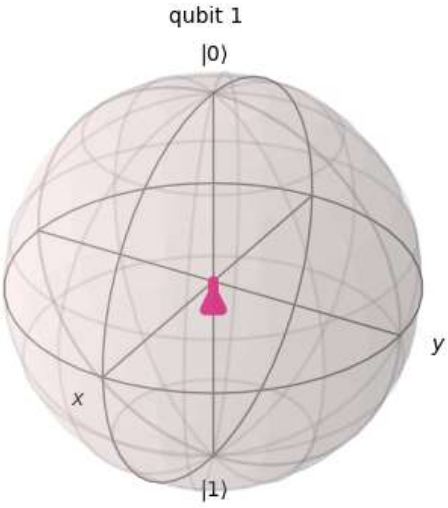
For inputs 1 1 Representation of Entangled States are:

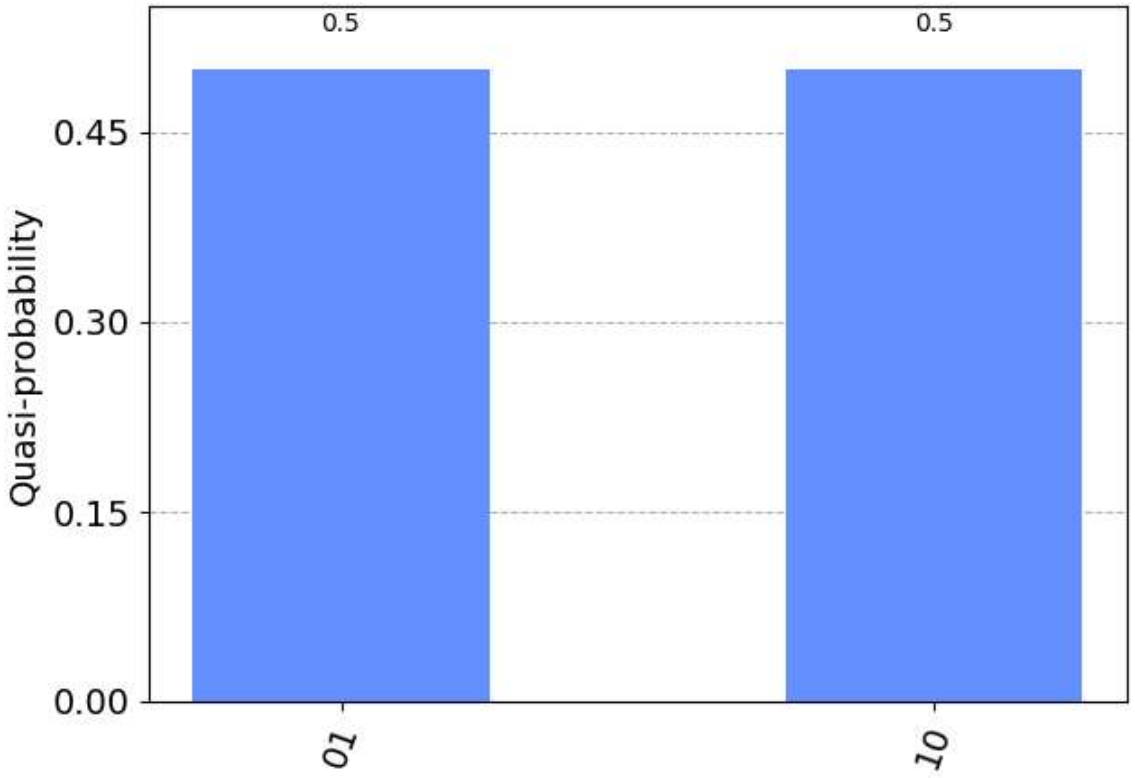


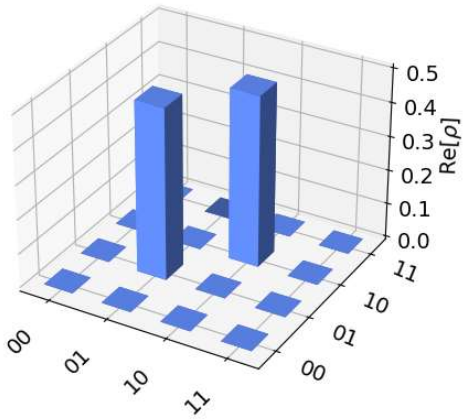

|01⟩
|10⟩

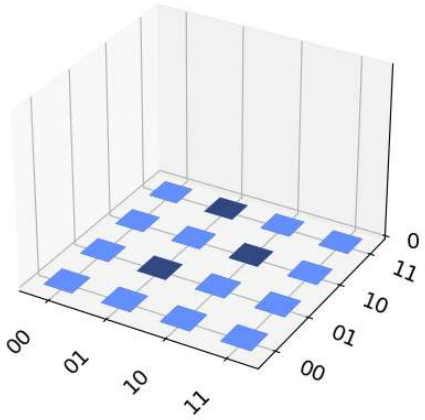

π/2
π
Phase
0
3π/2

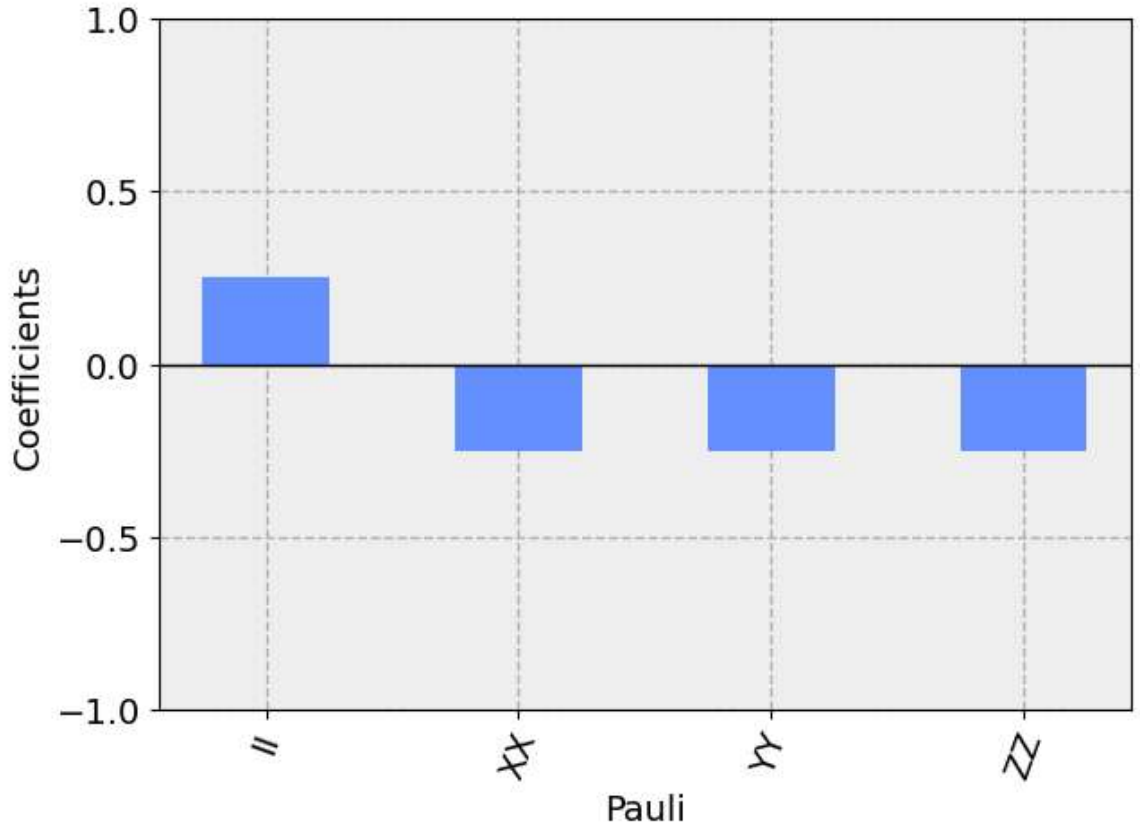

qubit 0
|0⟩
y
x
|1⟩

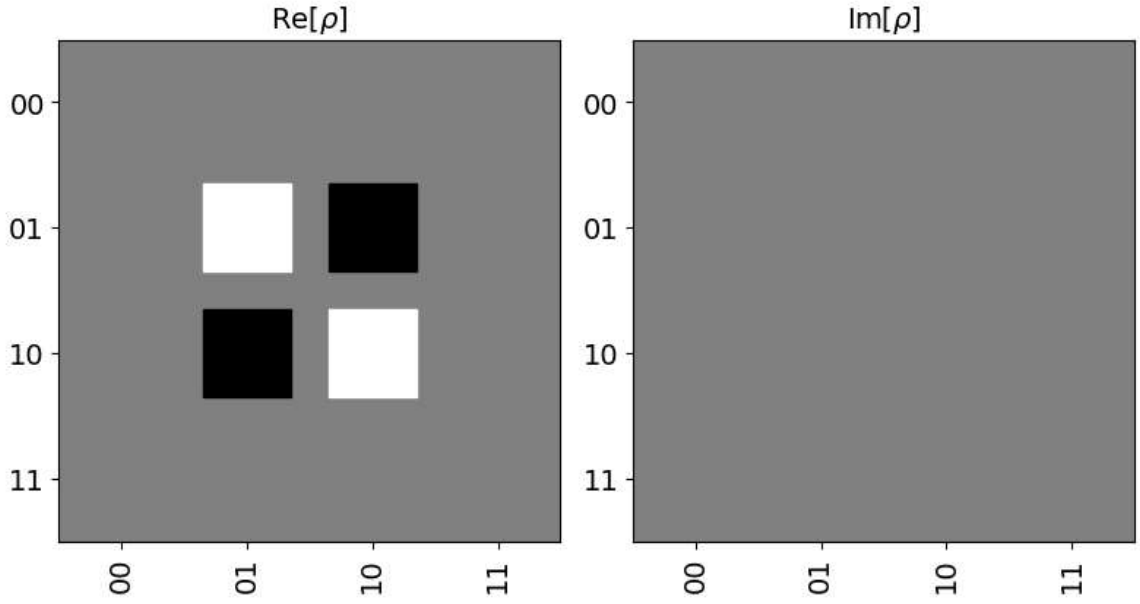

qubit 1
|0⟩
y
x
|1⟩


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
01
10


0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
00
01
10
11
00
01
10
11


00
01
10
11
00
01
10
11





part 2

In [59]:
```
from qiskit import IBMQ, execute
from qiskit.providers.ibmq import least_busy
from qiskit.tools import job_monitor

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qub
                                       and not x.configuration().simulator
                                       and x.status().operational==True))
```

```
ibmqfactory.load_account:WARNING:2023-09-14 15:56:19,479: Credentials are alrea
dy in use. The existing account in the session will be replaced.
```

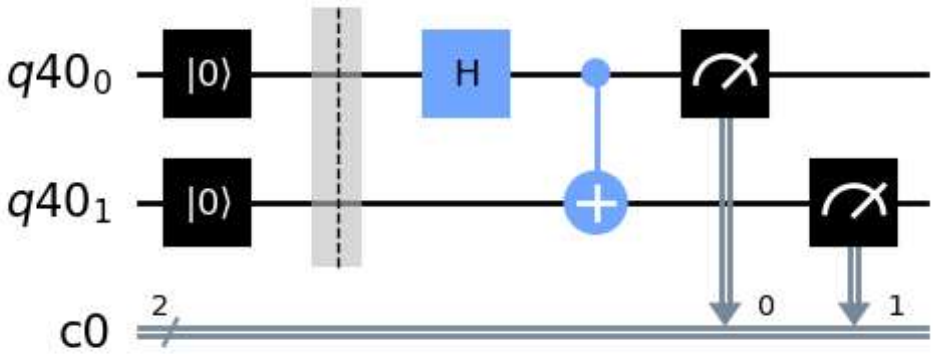
In [122…

```
def createBSRealDevice(inp1, inp2):
    qr = QuantumRegister(2)
    cr = ClassicalRegister(2)
    qc = QuantumCircuit(qr, cr)
    qc.reset(range(2))

    if inp1 == 1:
        qc.x(0)
    if inp2 == 1:
        qc.x(1)

    qc.barrier()

    qc.h(0)
    qc.cx(0,1)

    qc.measure(qr, cr)

    job = execute(qc, backend=backend, shots=100)
    job_monitor(job)
    result = job.result()

    return qc, result
```

In [123…

```
inp1 = 0
inp2 = 0

print('For inputs',inp2,inp1,'Representation of Entangled States are,')

#first results
qc, first_result = createBSRealDevice(inp1, inp2)
first_counts = first_result.get_counts()

# Draw the quantum circuit
display(qc.draw())
```

```
For inputs 0 0 Representation of Entangled States are,
Job Status: job has successfully run
```



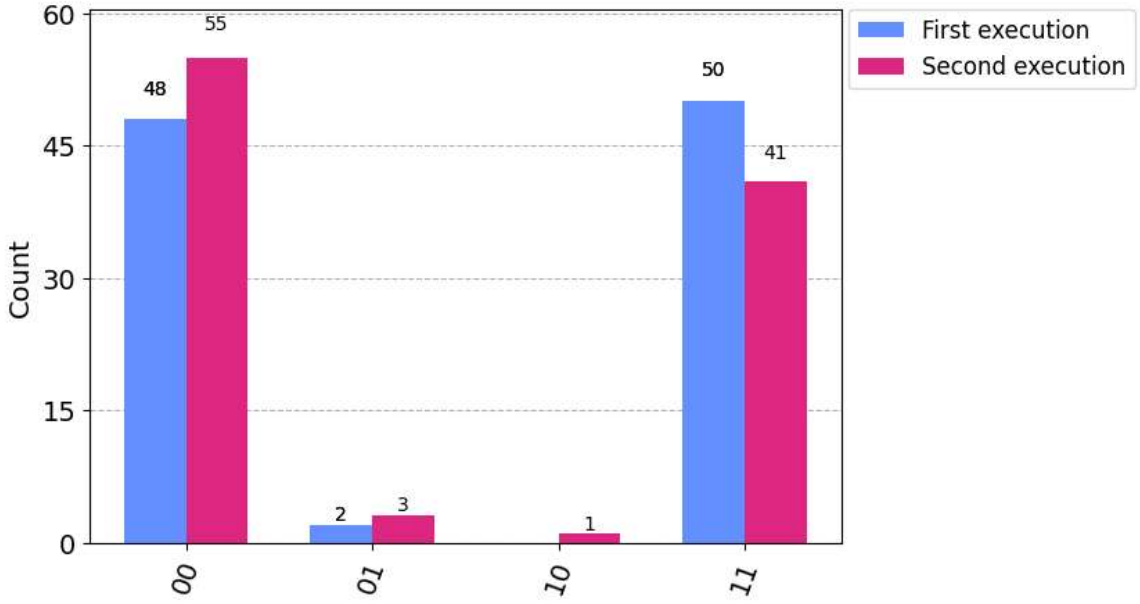
In [124…

```
#second results
qc, second_result = createBSRealDevice(inp1, inp2)
second_counts = second_result.get_counts()


# Plot results on histogram with legend
```

```
legend = ['First execution', 'Second execution']
plot_histogram([first_counts, second_counts], legend=legend)
```

```
Job Status: job has successfully run
```

Out[124]:



In [60]:

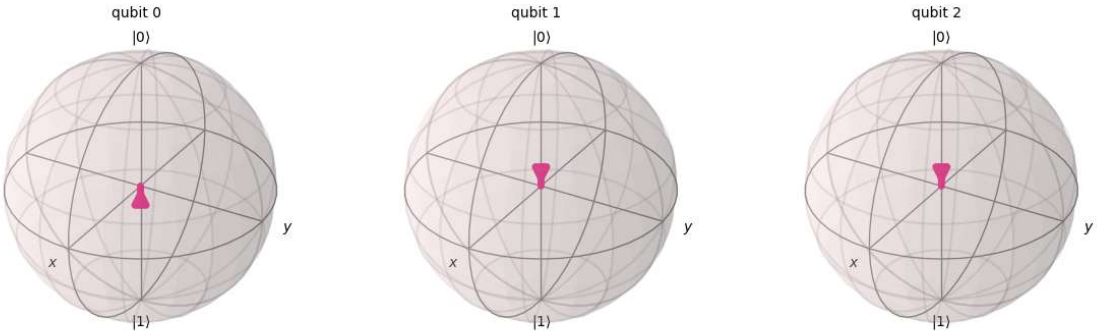
```
def ghzCircuit(inp1, inp2, inp3):

    qc = QuantumCircuit(3)
    qc.reset(range(3))

    if inp1 == 1:
        qc.x(0)
    if inp2 == 1:
        qc.x(1)
    if inp3 == 1:
        qc.x(2)

    qc.barrier()

    qc.h(0)
    qc.cx(0,1)
    qc.cx(0,2)

    qc.save_statevector()
    qobj = assemble(qc)
    result = sim.run(qobj).result()
    state = result.get_statevector()

    return qc, state, result
```

In [61]:

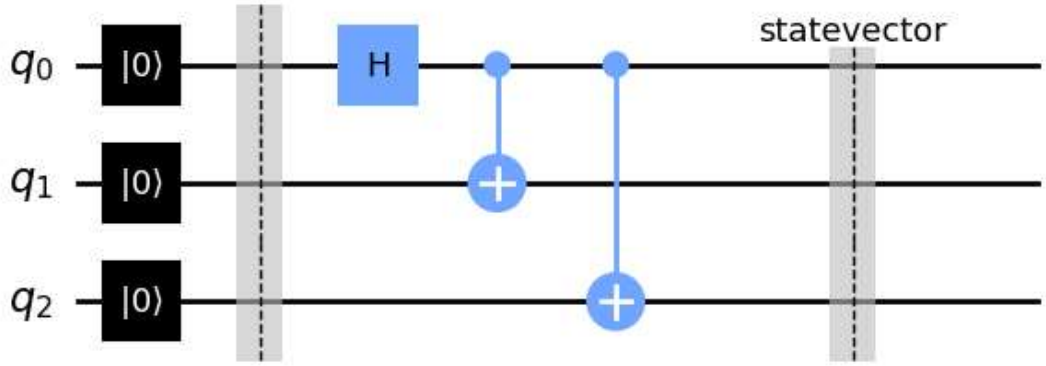
```
print('Note: Since these qubits are in entangled state, their state cannot be wr

inp1 = 0
inp2 = 1
inp3 = 1

qc, state, result = ghzCircuit(inp1, inp2, inp3)

display(plot_bloch_multivector(state))

# Uncomment below code in order to explore other states
for inp3 in ['0','1']:
```

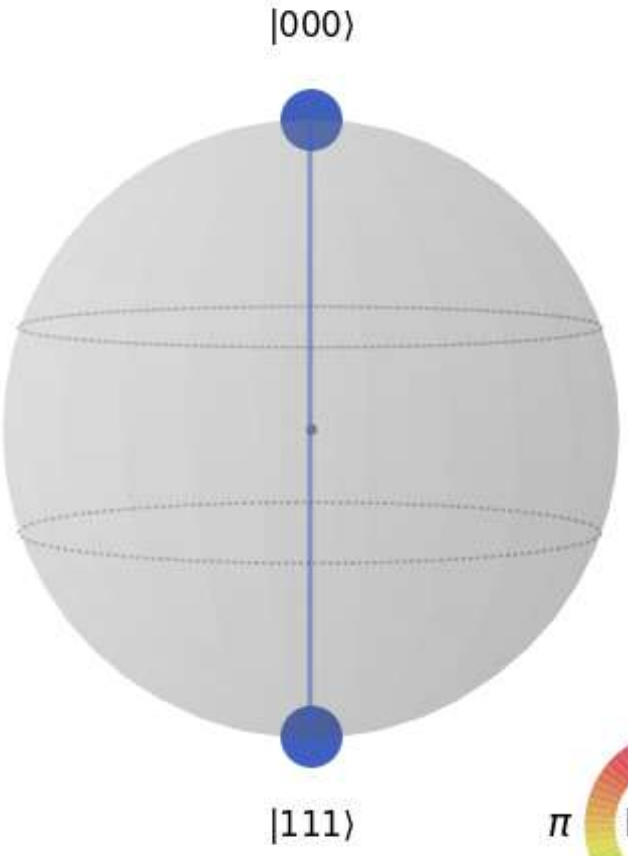
```
    for inp2 in ['0','1']:
        for inp1 in ['0','1']:
            qc, state, result = ghzCircuit(inp1, inp2, inp3)

            print('For inputs',inp3,inp2,inp1,'Representation of GHZ States are:

            # Uncomment any of the below functions to visualize the resulting qu

            # Draw the quantum circuit
            display(qc.draw())

            # Plot states on QSphere
            display(plot_state_qsphere(state))

            # Plot states on Bloch Multivector
            display(plot_bloch_multivector(state))

            # Plot histogram
            display(plot_histogram(result.get_counts()))

            # Plot state matrix like a city
            display(plot_state_city(state))

            # Represent state matix using Pauli operators as the basis
            display(plot_state_paulivec(state))

            #Plot state matrix as Hinton representation
            display(plot_state_hinton(state))

            #print('\n')
```

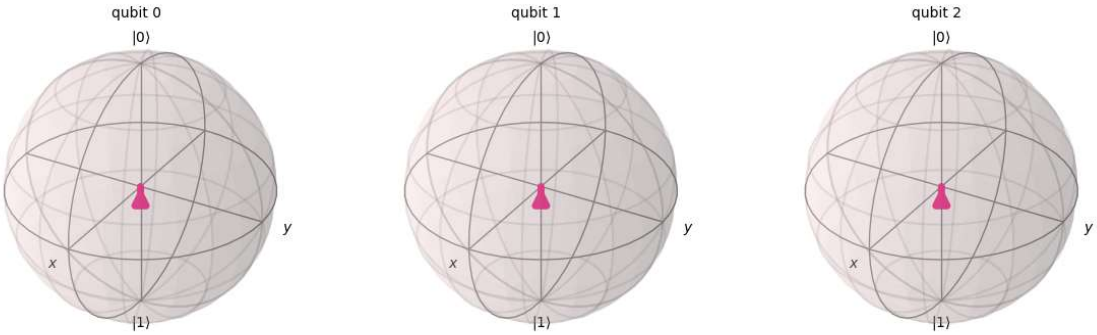
Note: Since these qubits are in entangled state, their state cannot be written as two separate qubit states. This also means that we lose information when we try to plot our state on separate Bloch spheres as seen below.

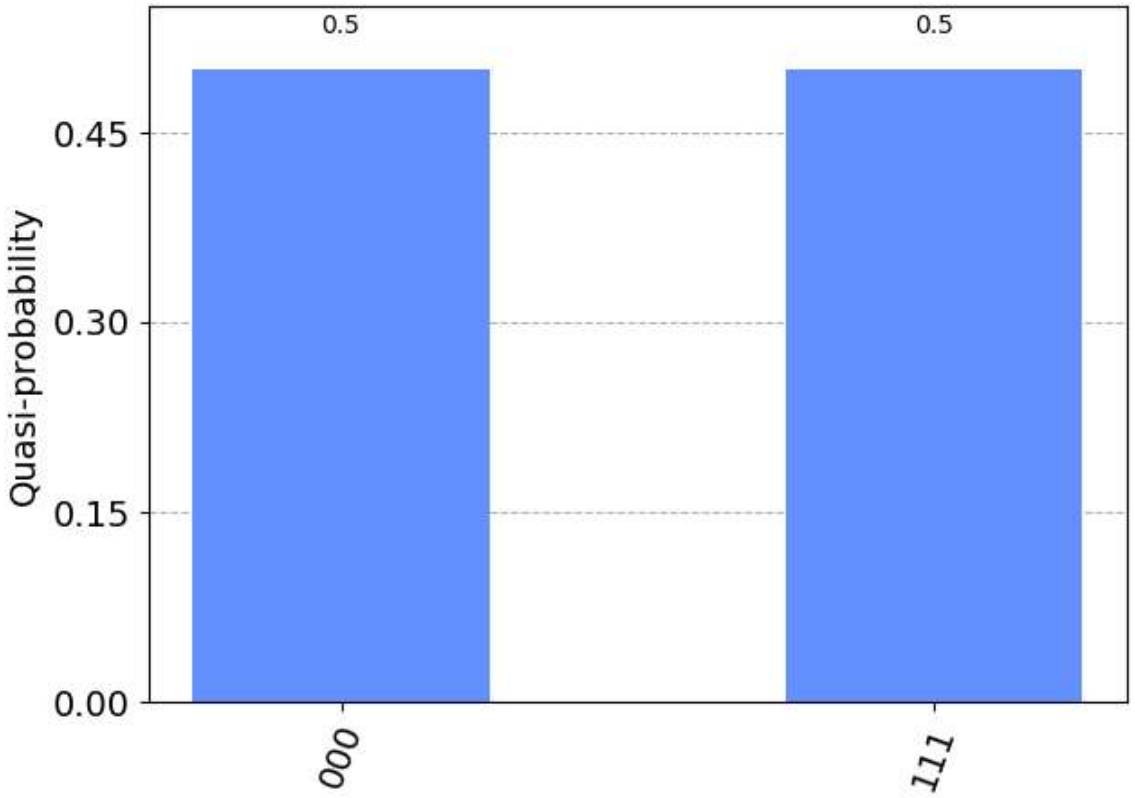


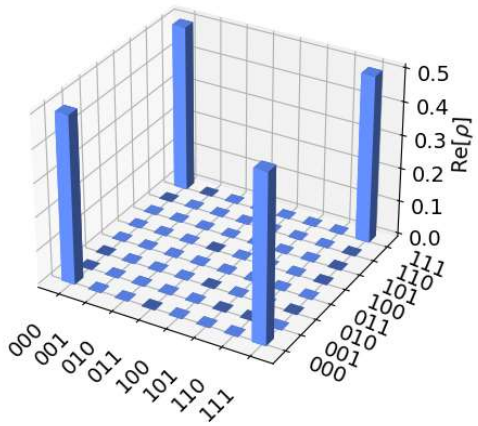
For inputs 0 0 0 Representation of GHZ States are:

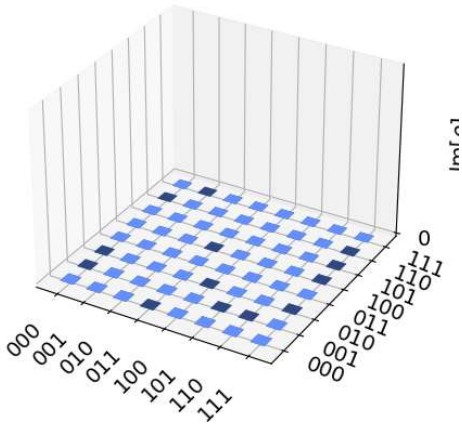


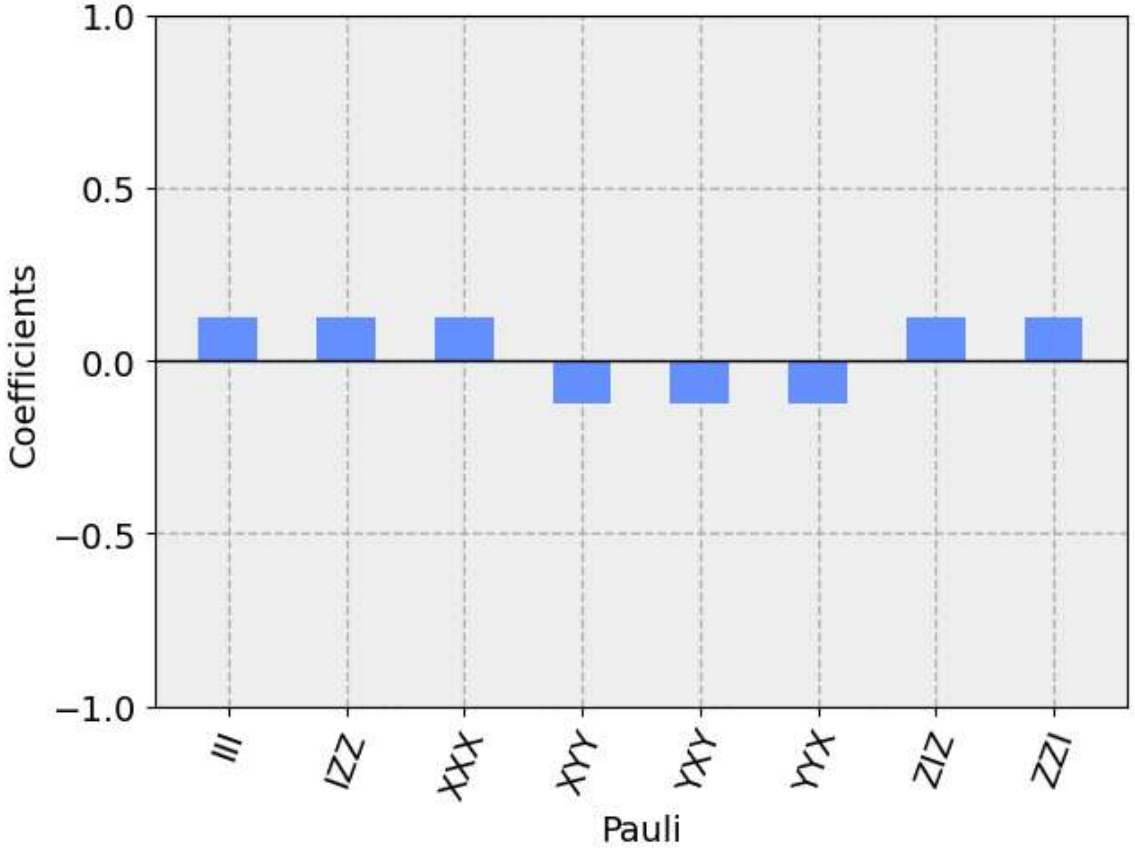

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

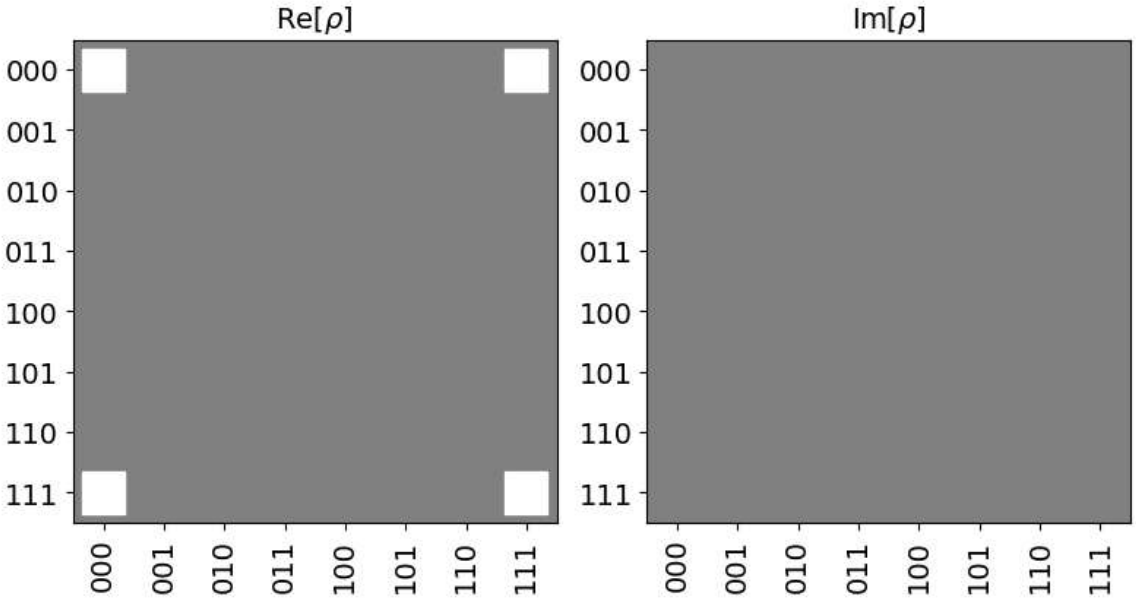

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111

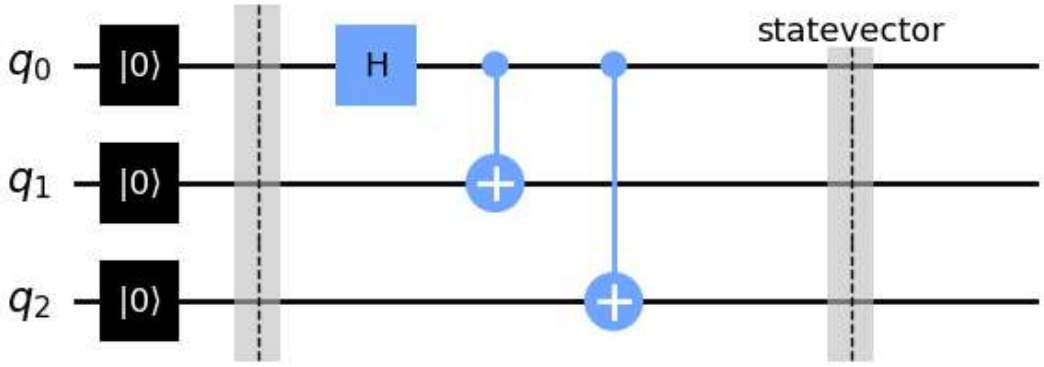

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111

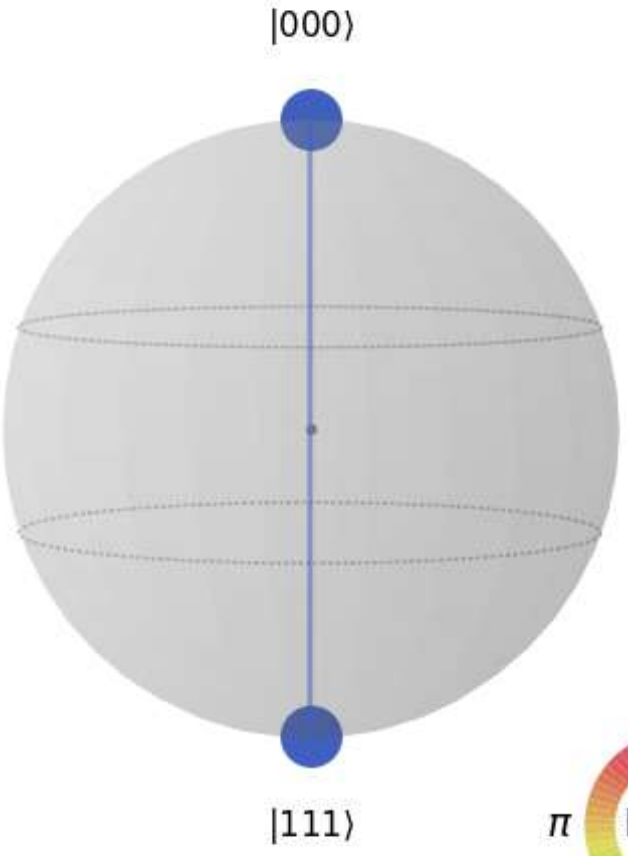

0
000
001
010
011
100
101
110
111

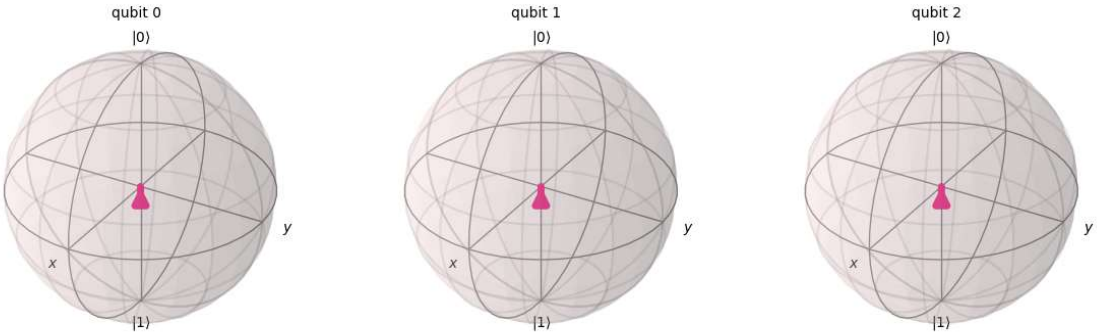


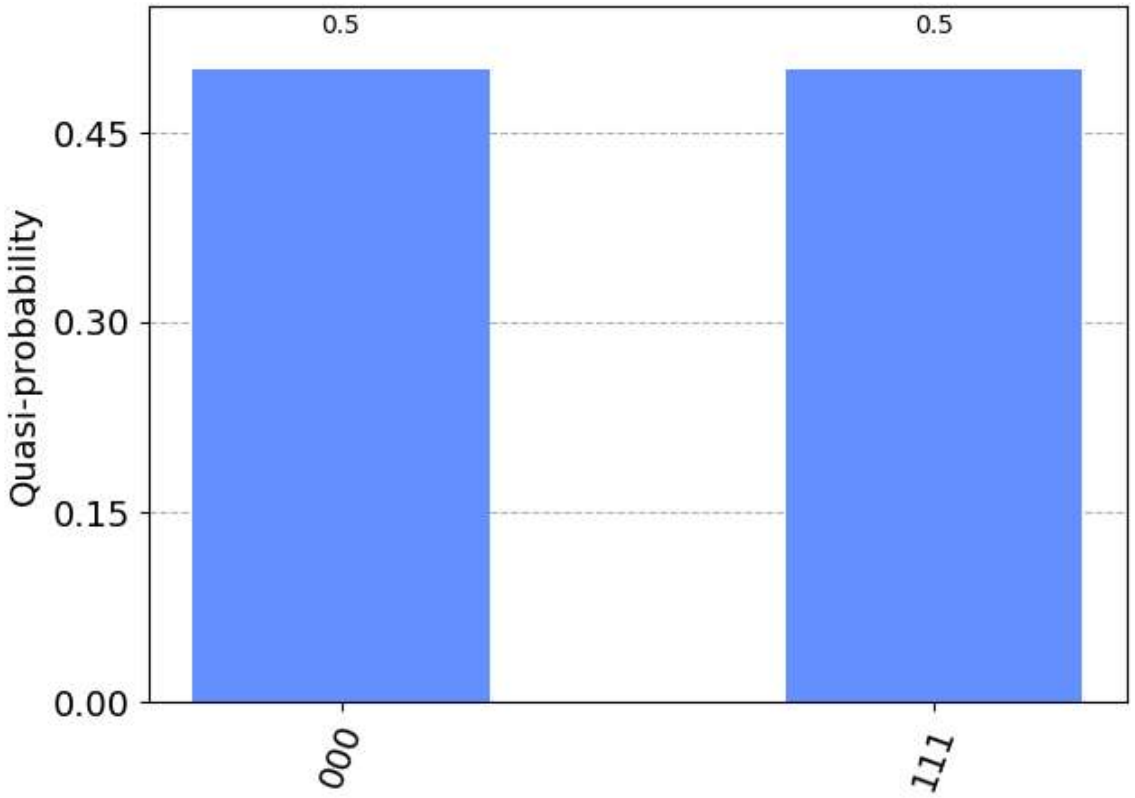


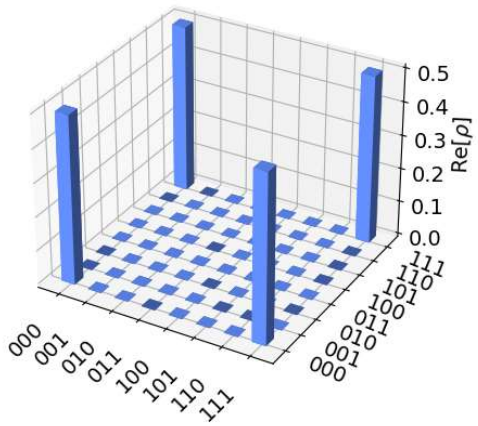
For inputs 0 0 1 Representation of GHZ States are:

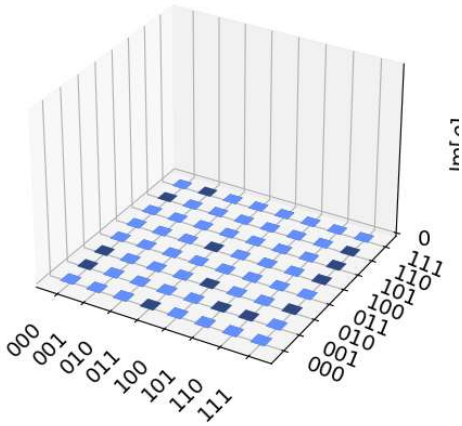


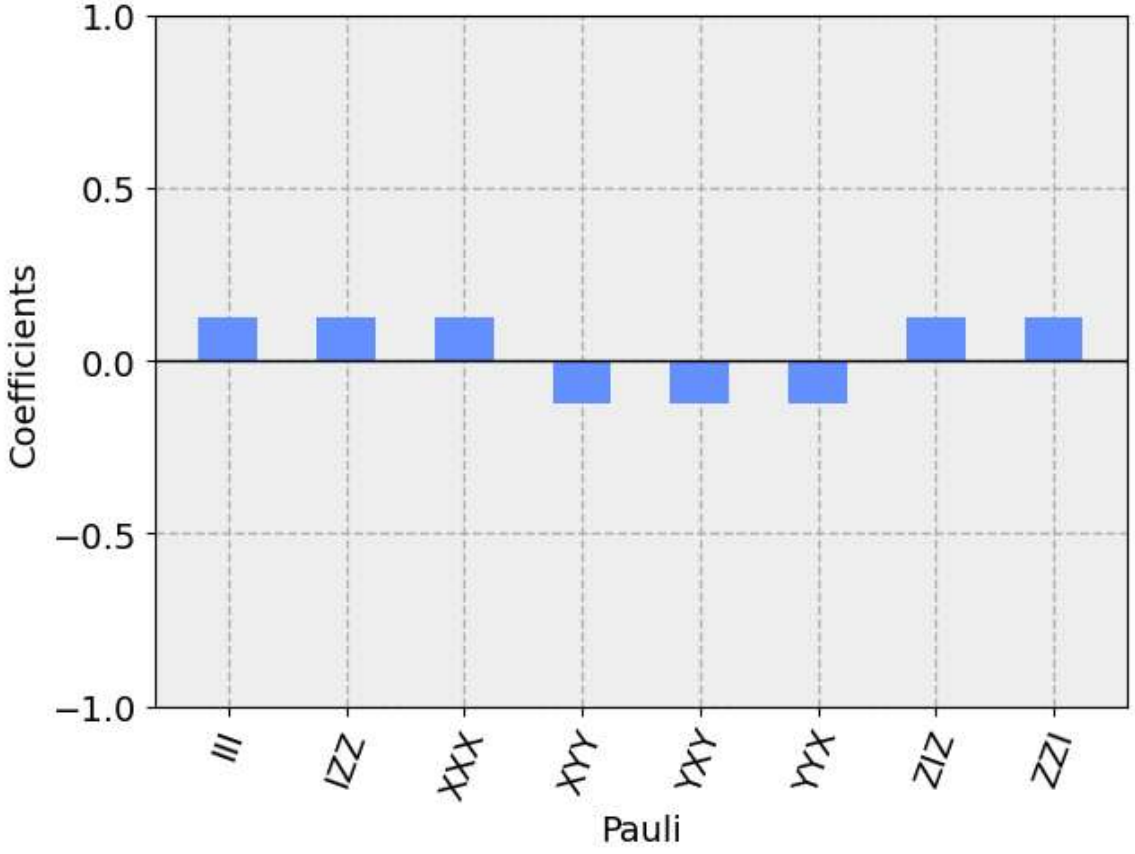

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

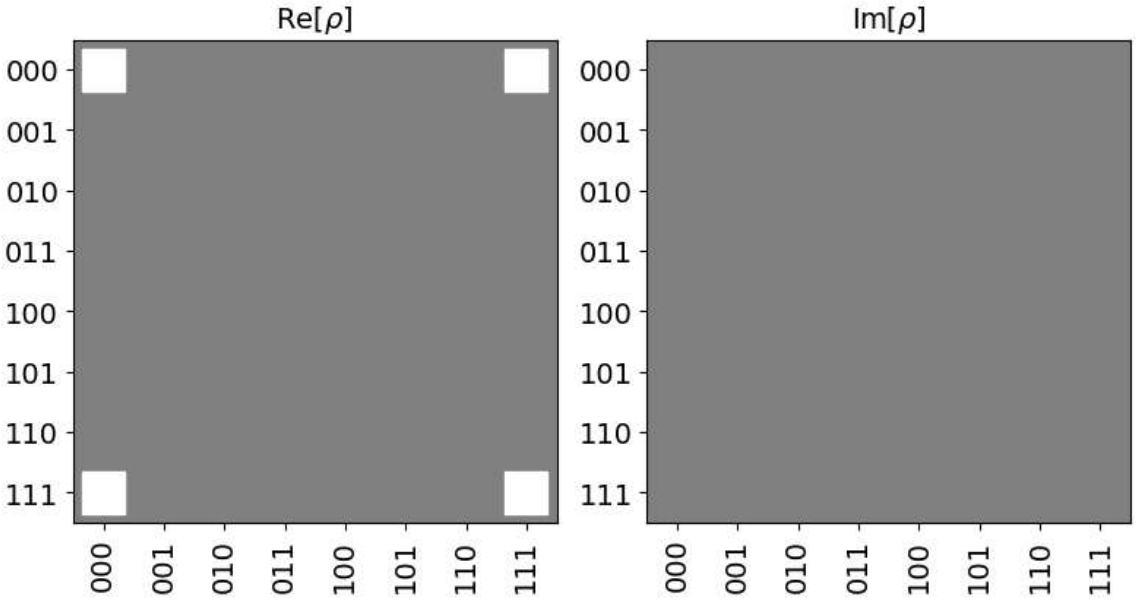

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111

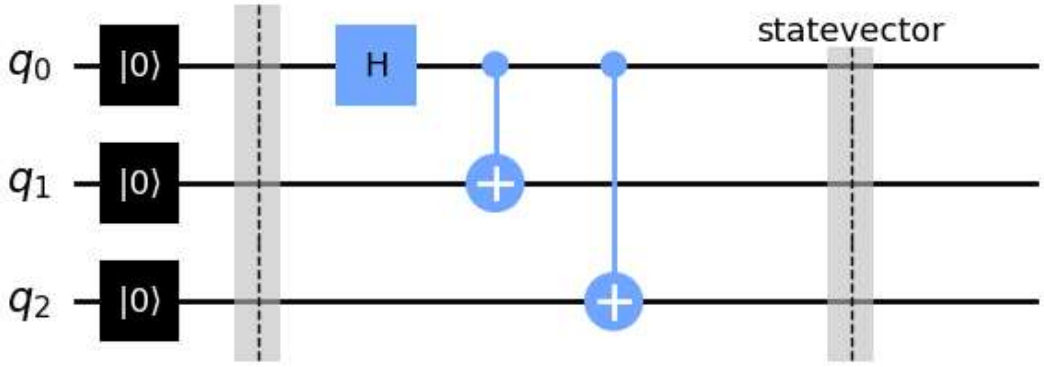

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111

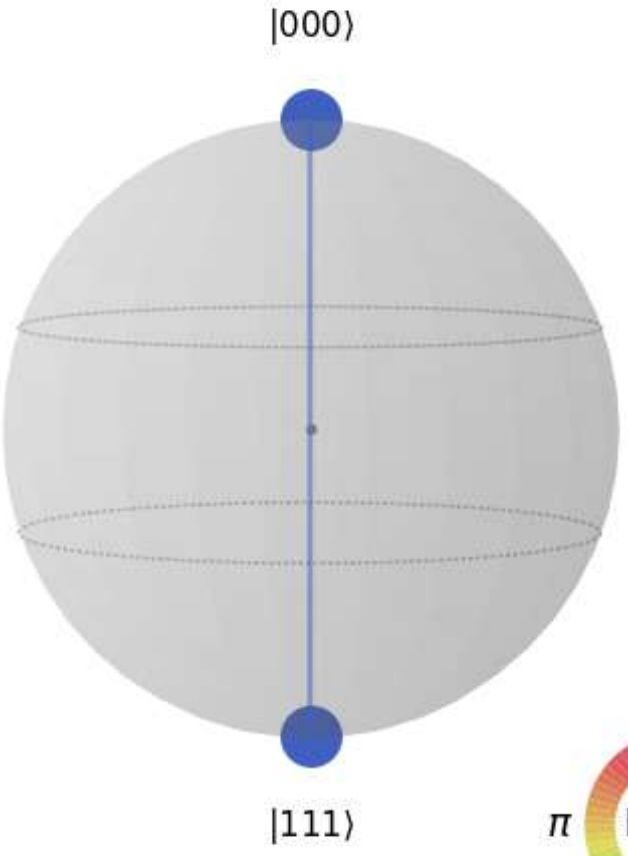

0
000
001
010
011
100
101
110
111

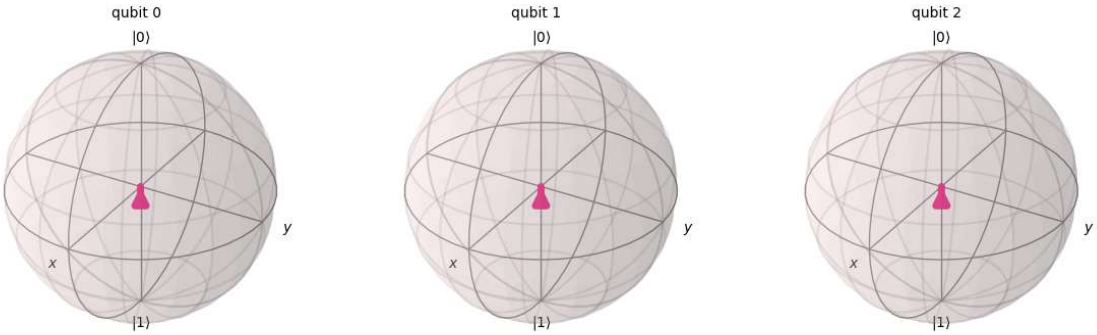


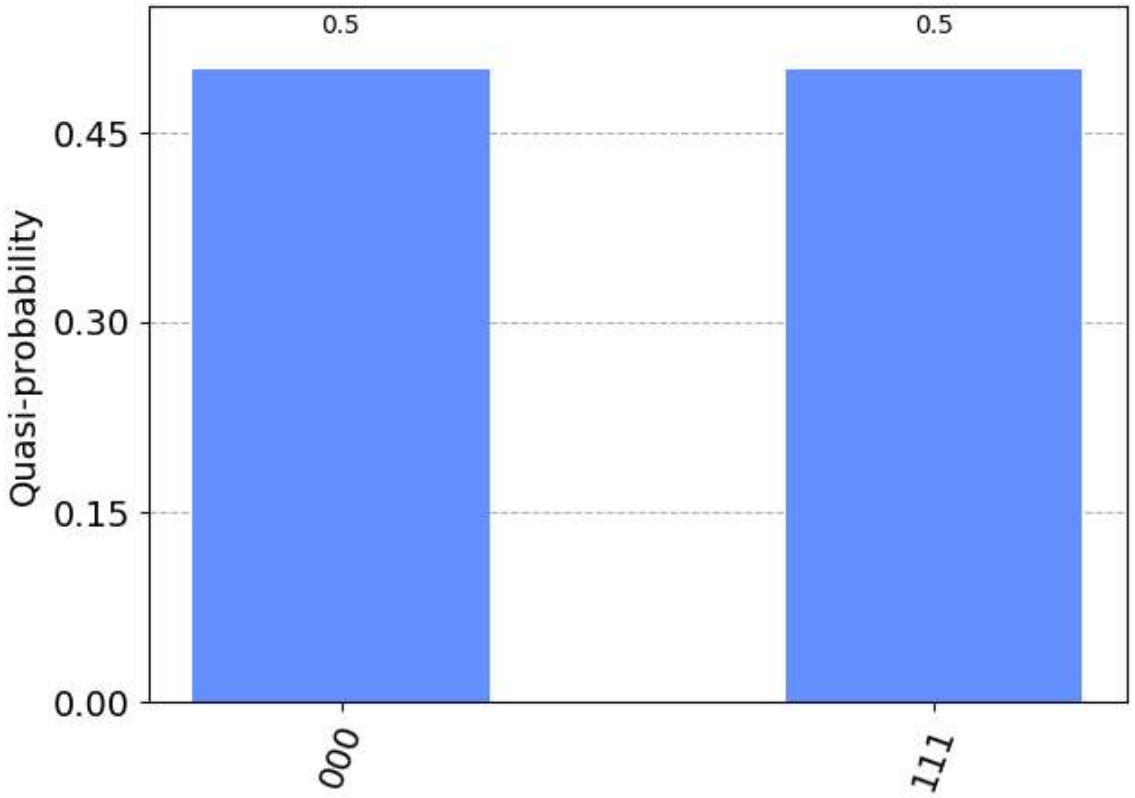


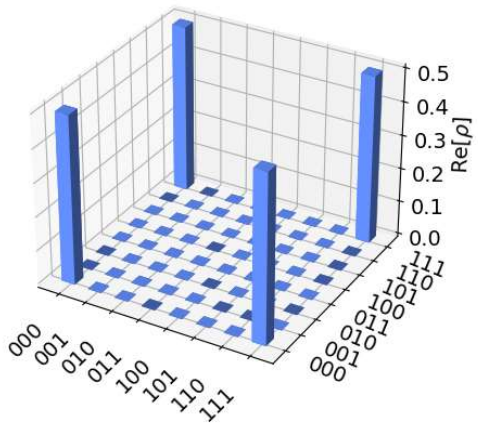
For inputs 0 1 0 Representation of GHZ States are:

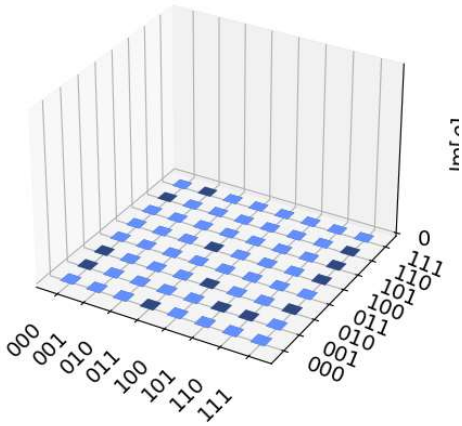


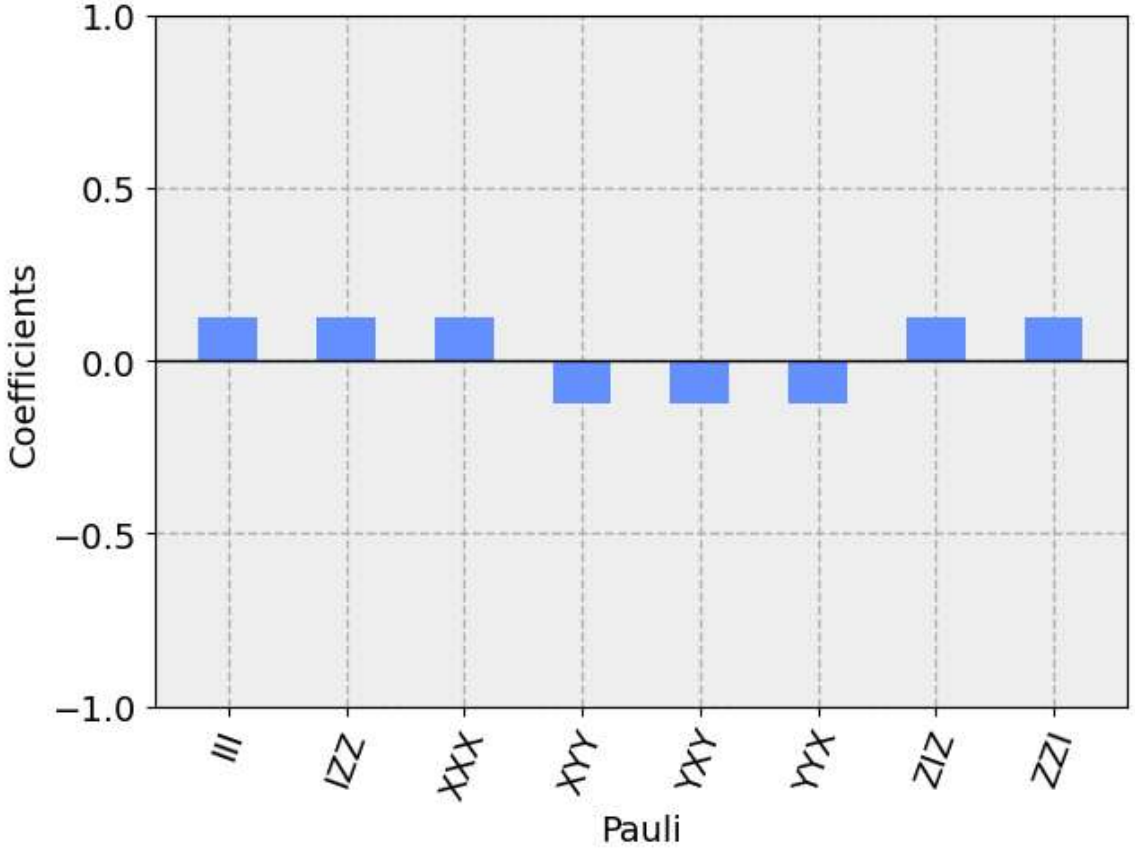

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

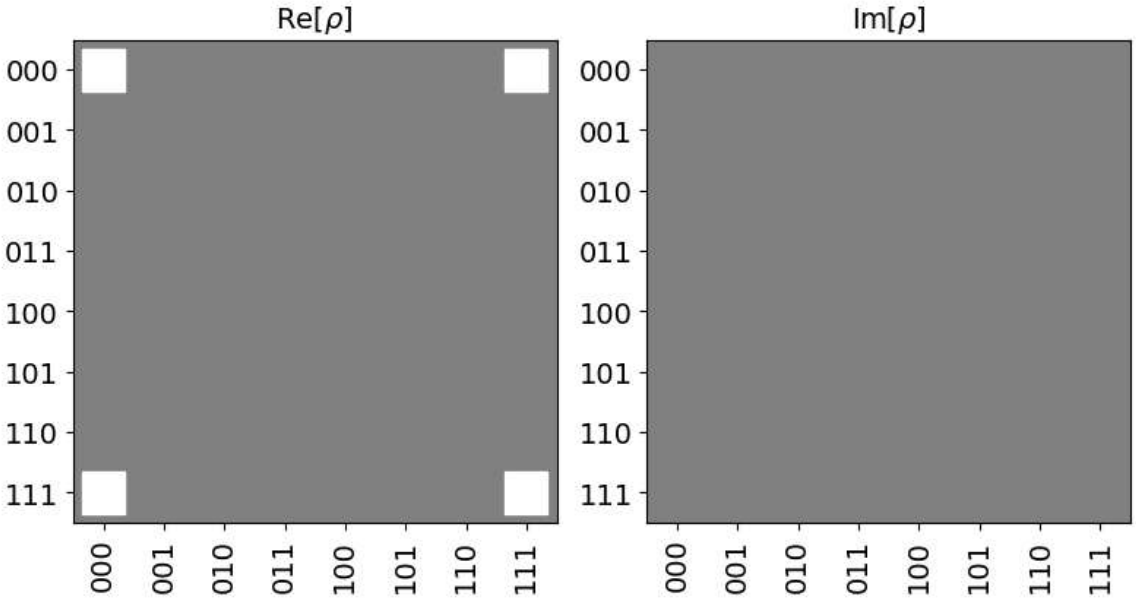

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111

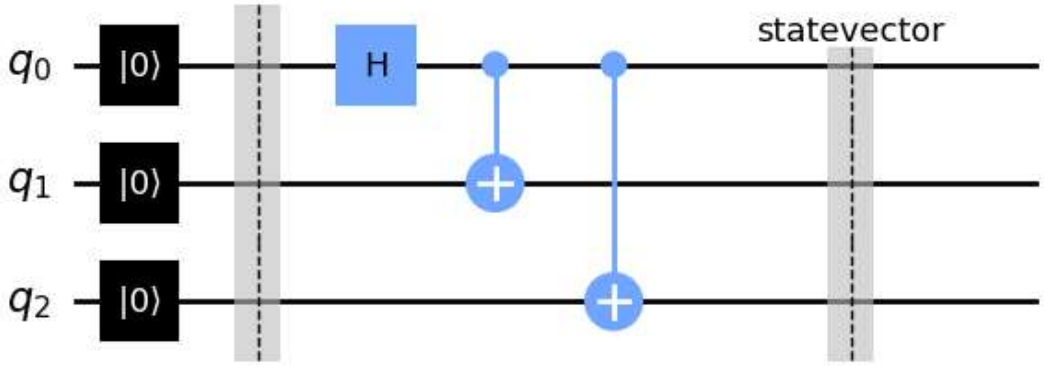

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111

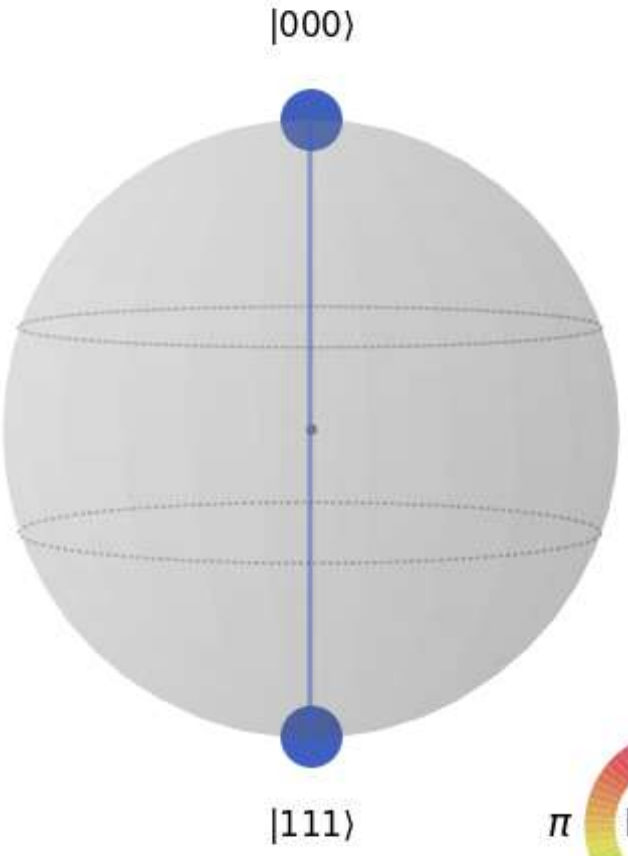

0
000
001
010
011
100
101
110
111

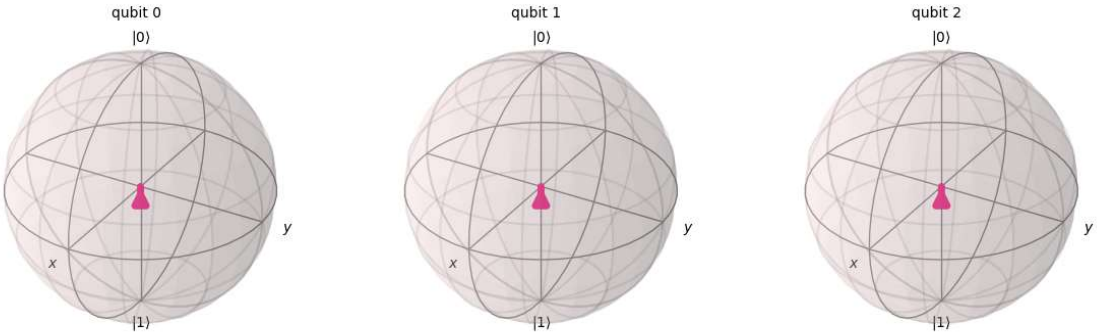


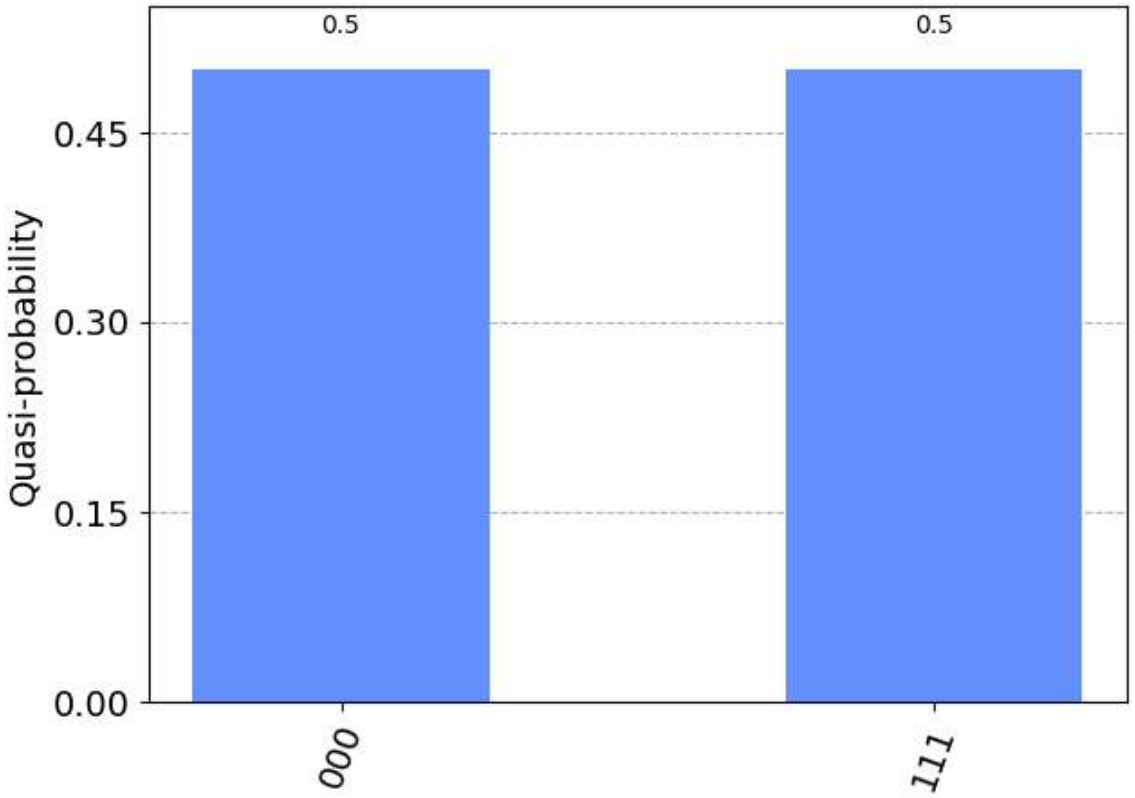


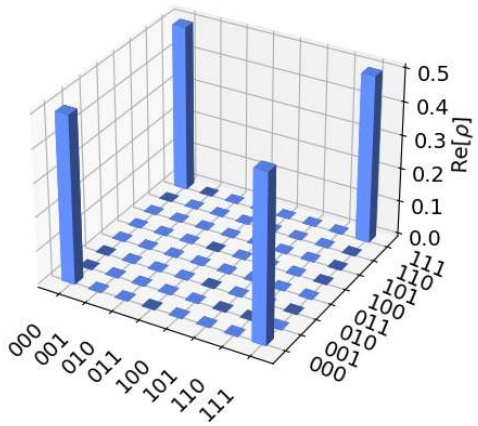
For inputs 0 1 1 Representation of GHZ States are:

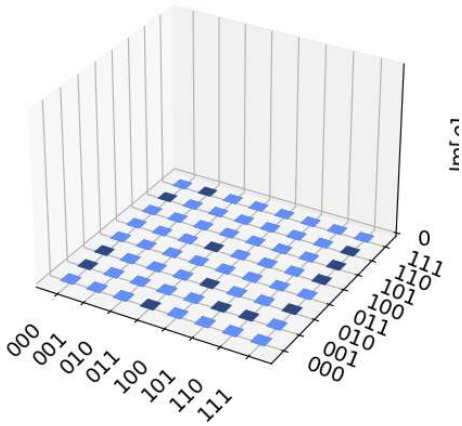


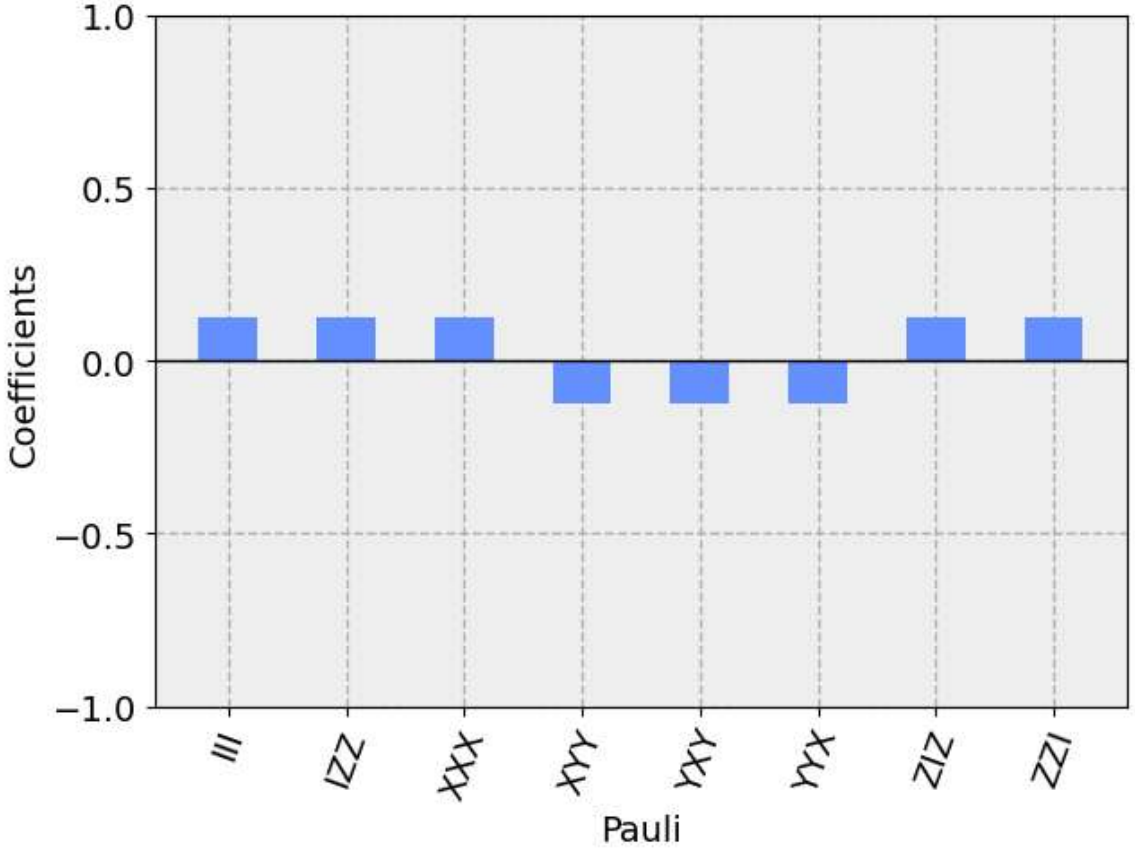

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

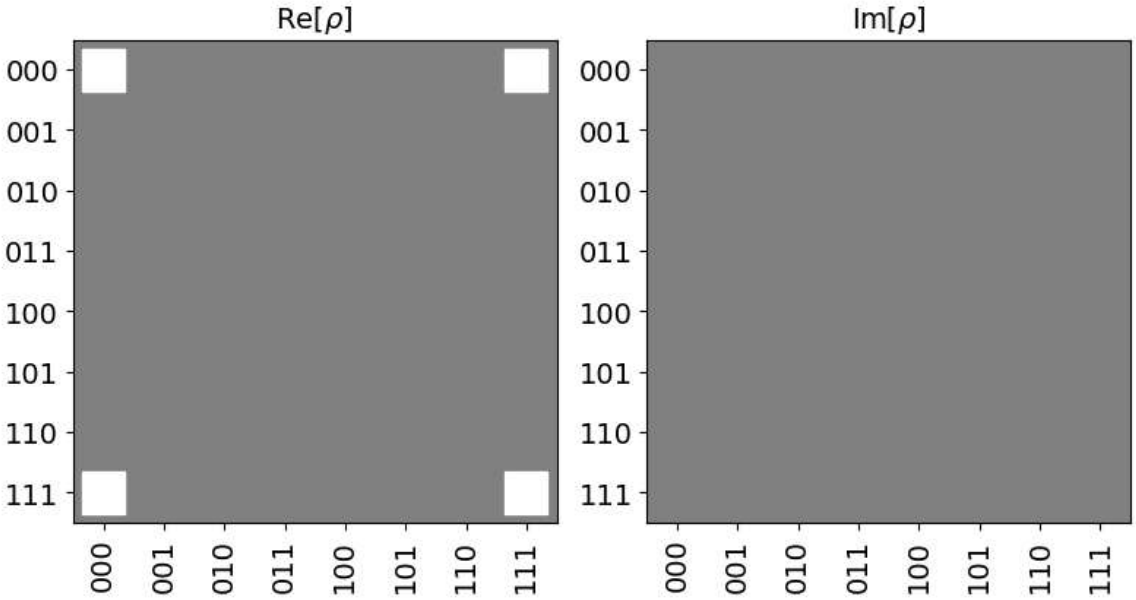

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111

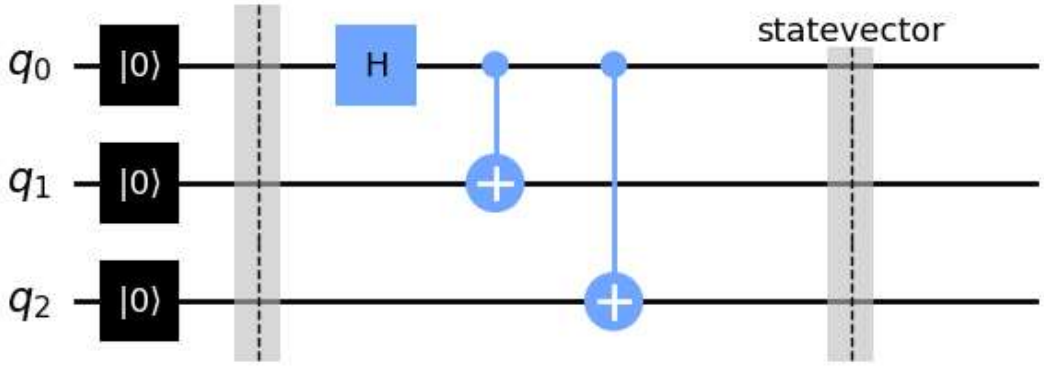

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111

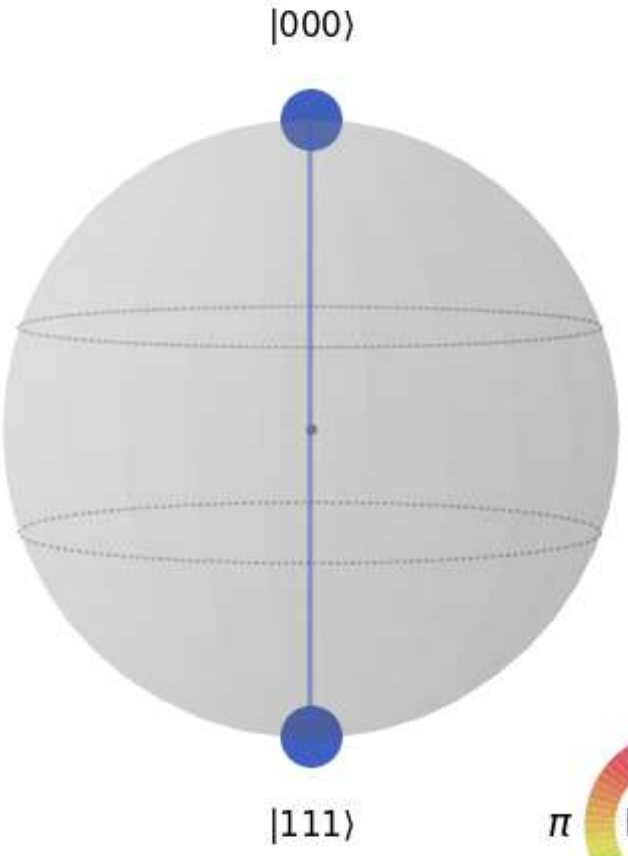

0
000
001
010
011
100
101
110
111

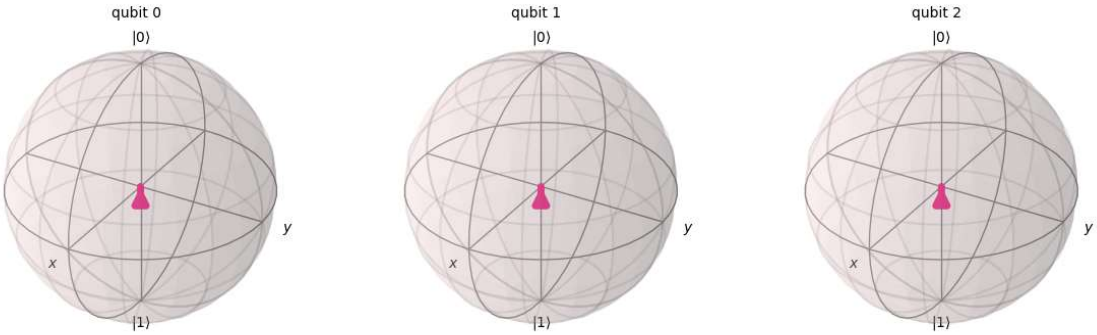


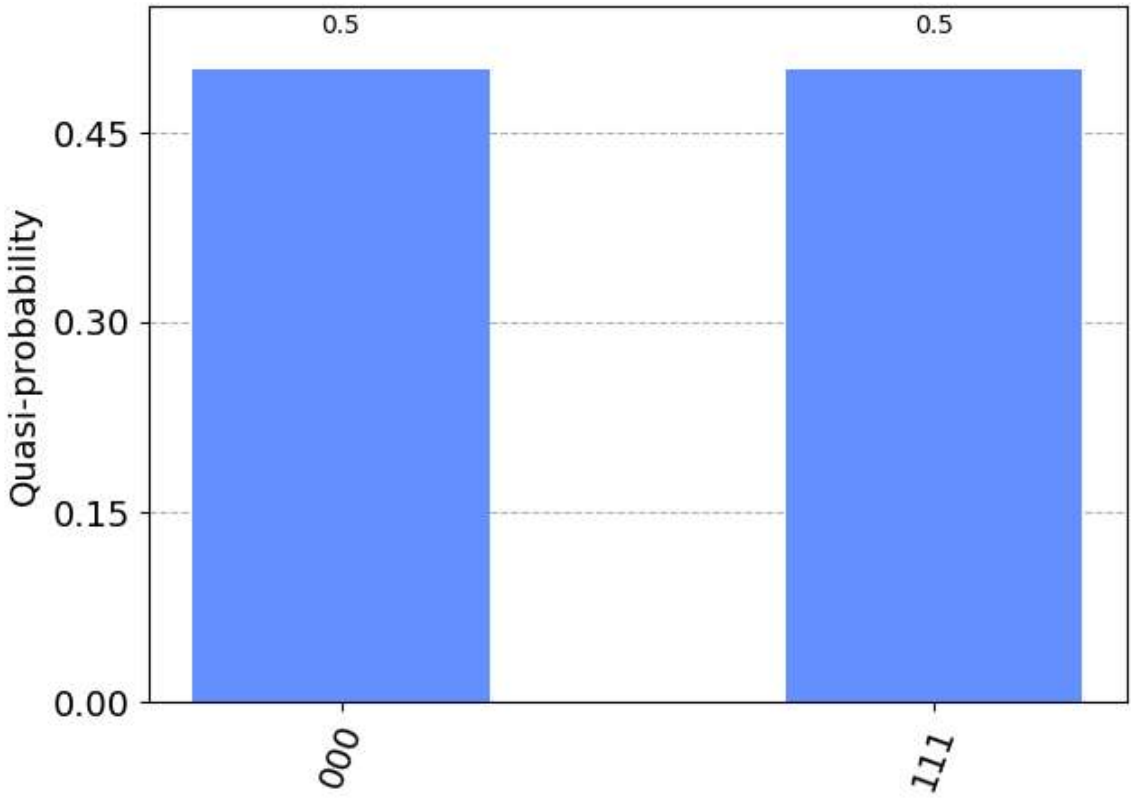


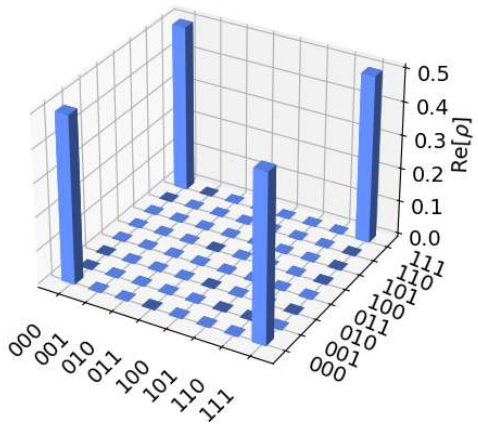
For inputs 1 0 0 Representation of GHZ States are:

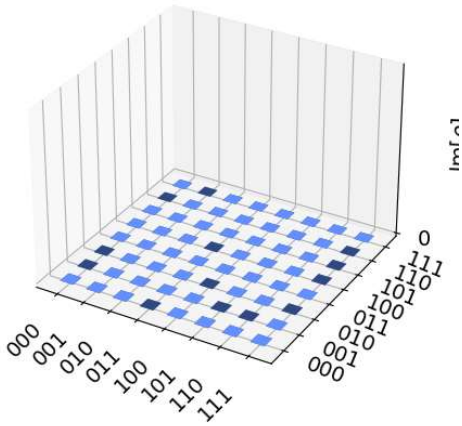


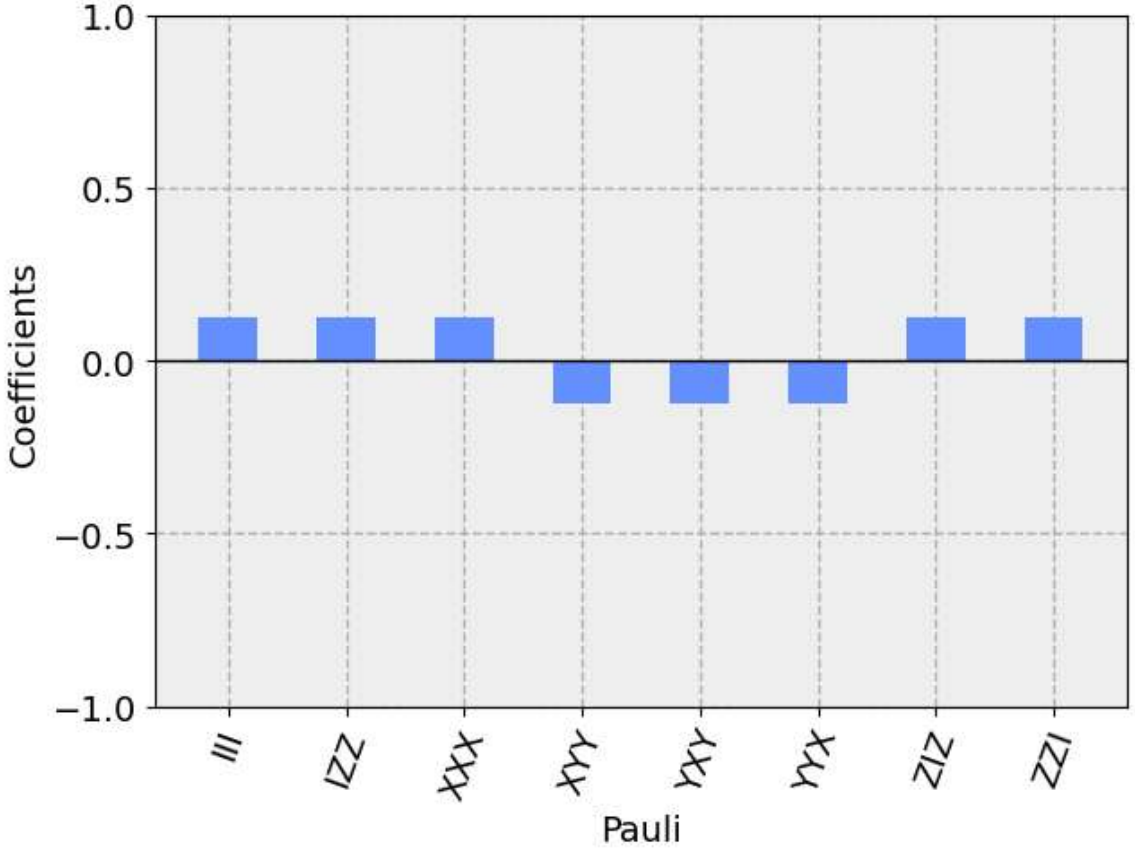

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

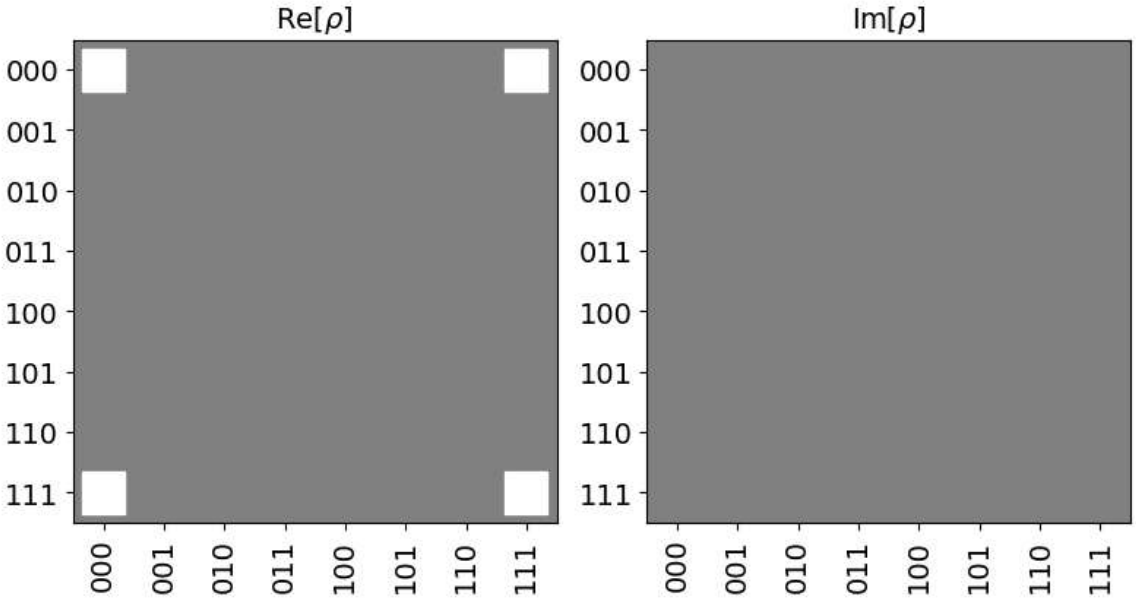

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111

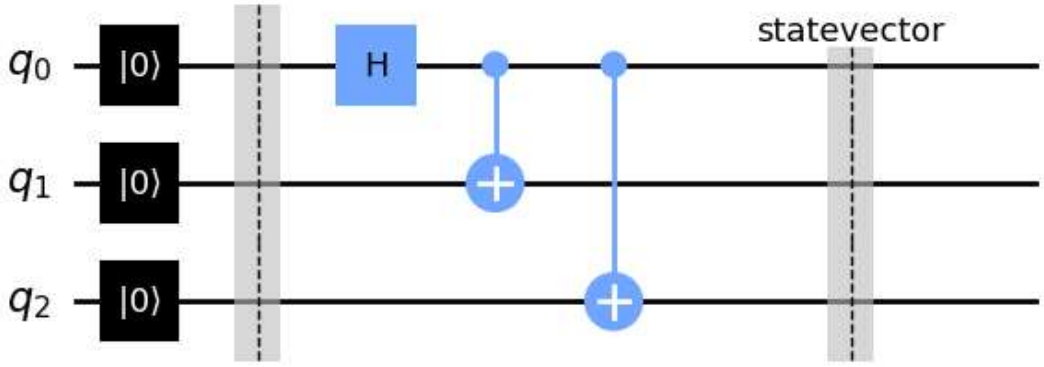

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111

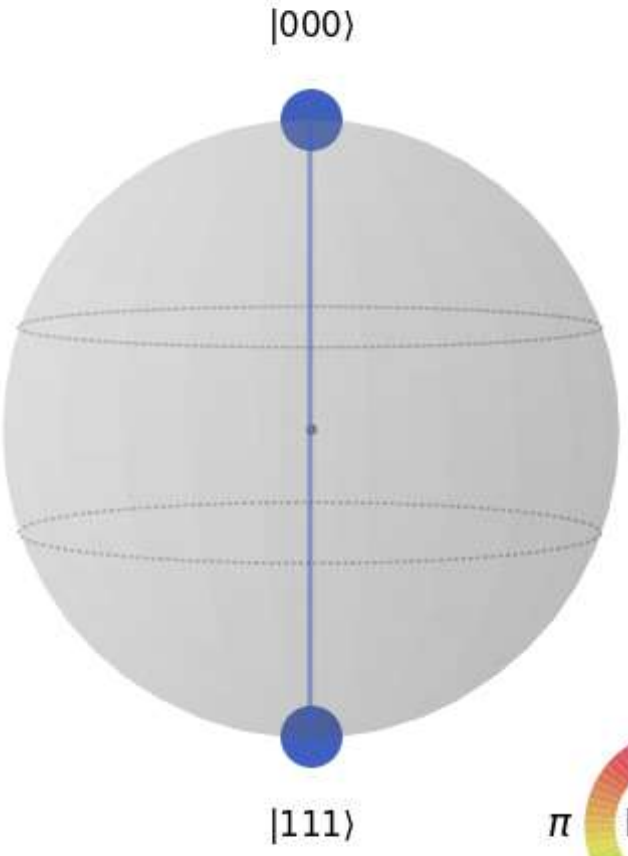

0
000
001
010
011
100
101
110
111

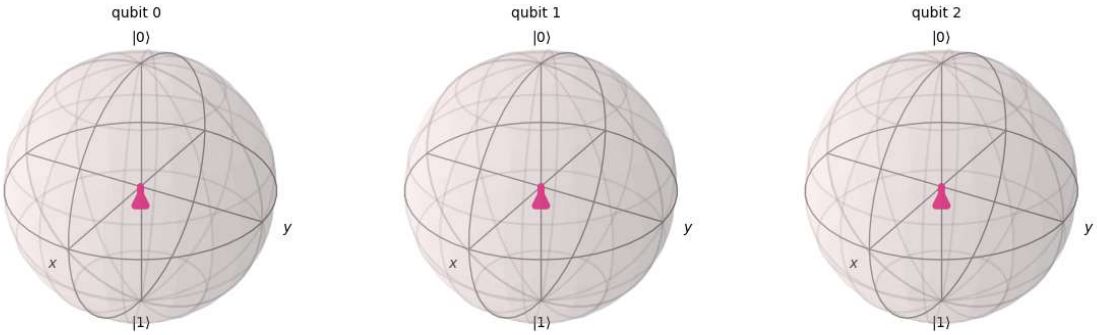


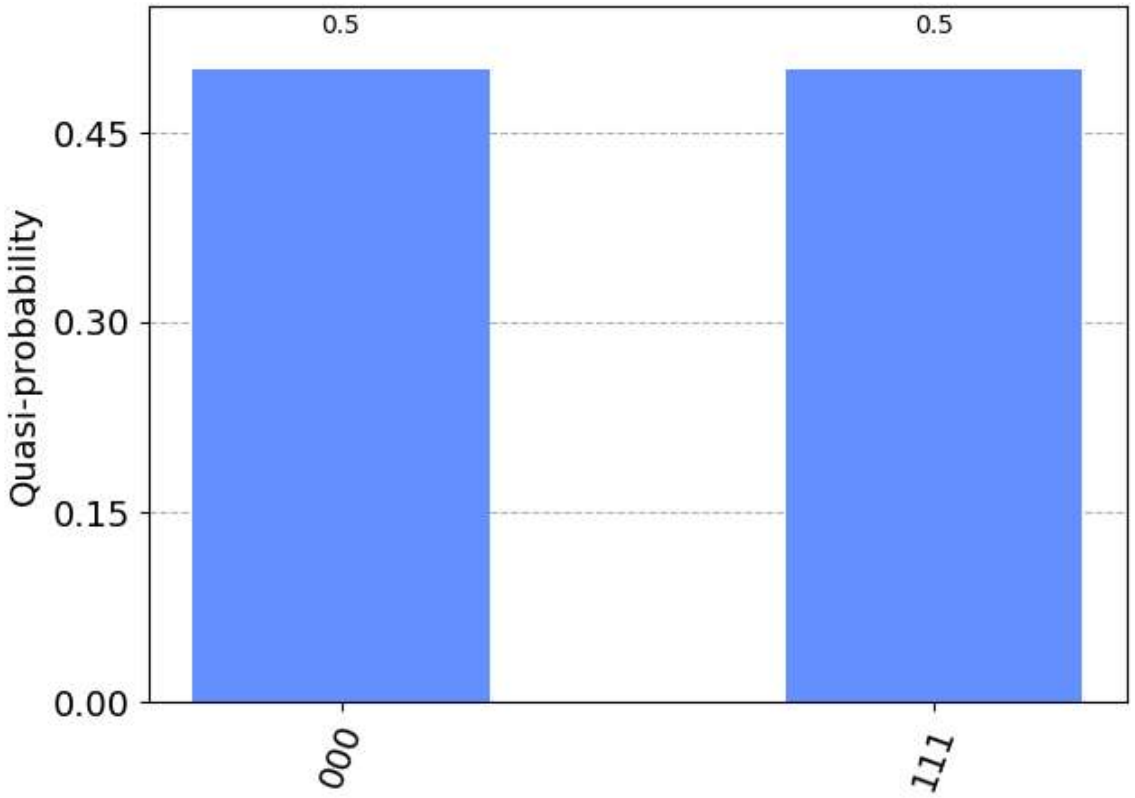


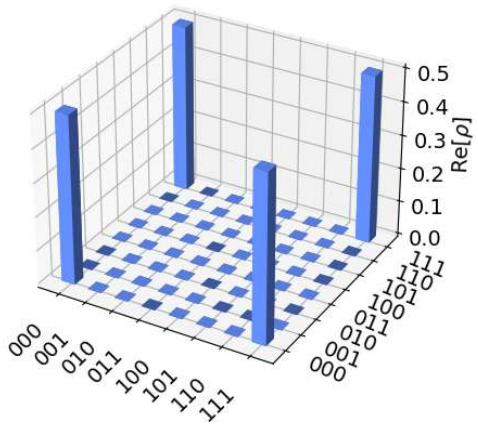
For inputs 1 0 1 Representation of GHZ States are:

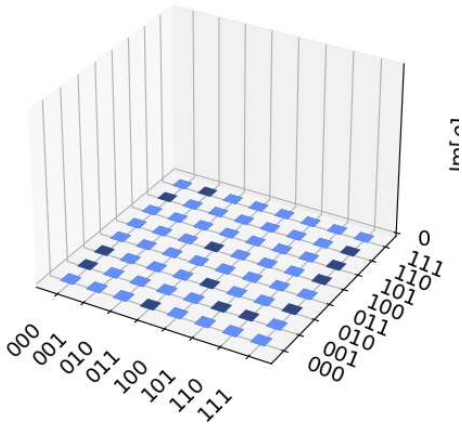


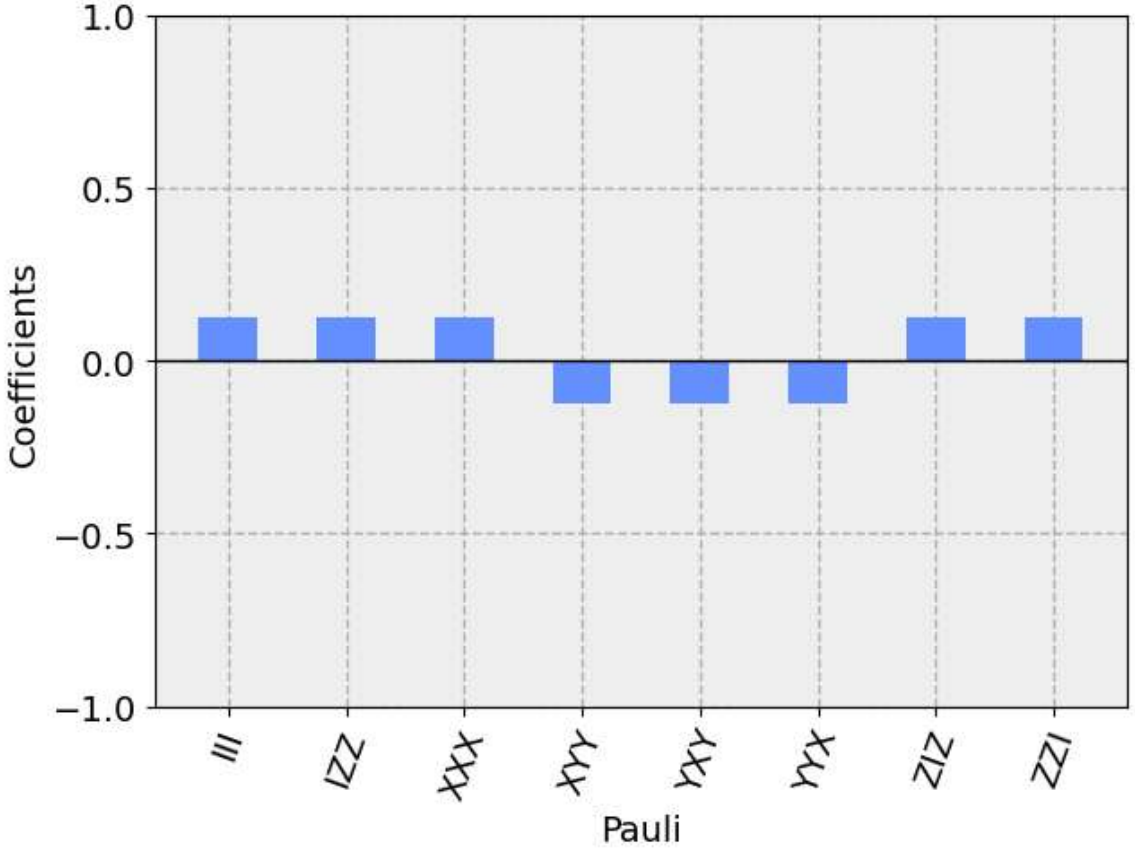

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

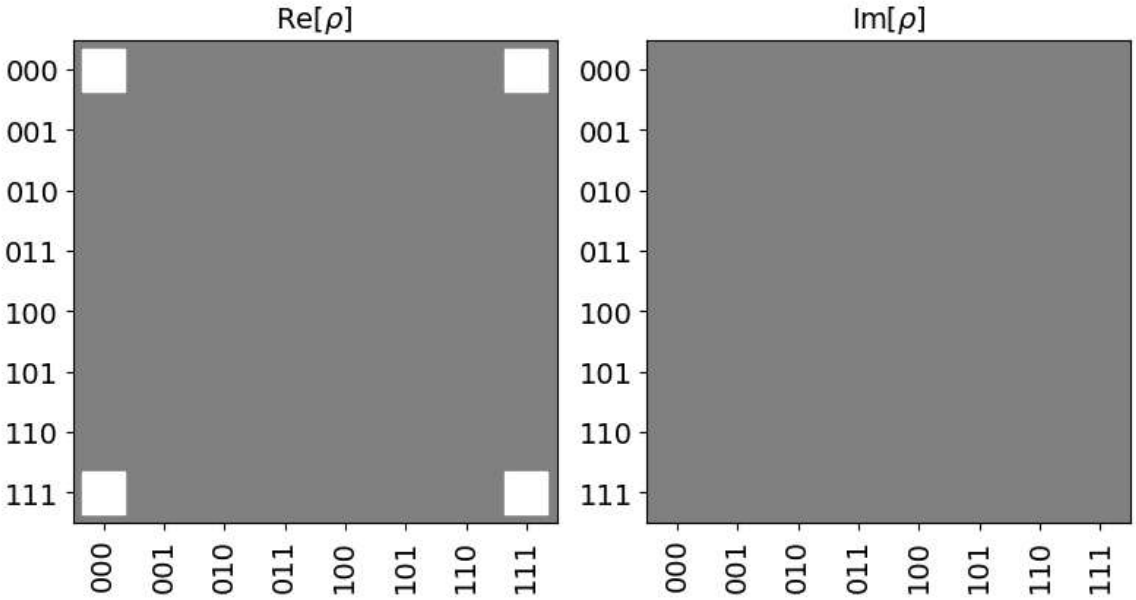

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111

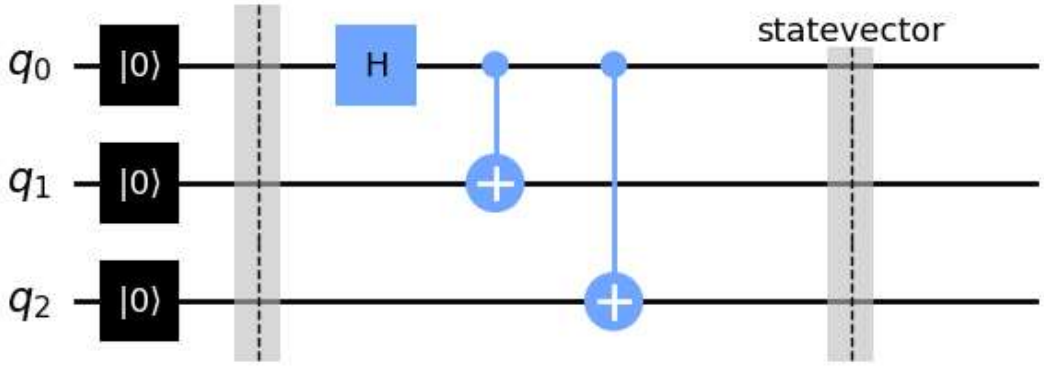

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111

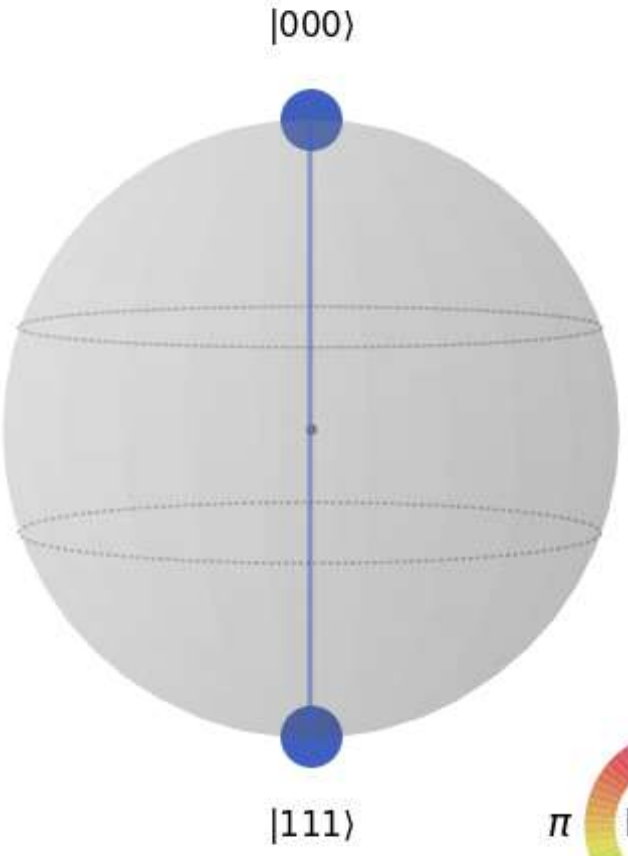

0
000
001
010
011
100
101
110
111

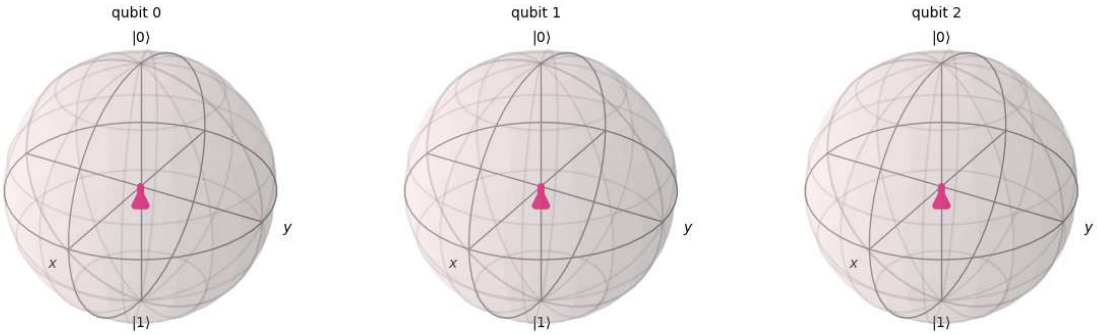


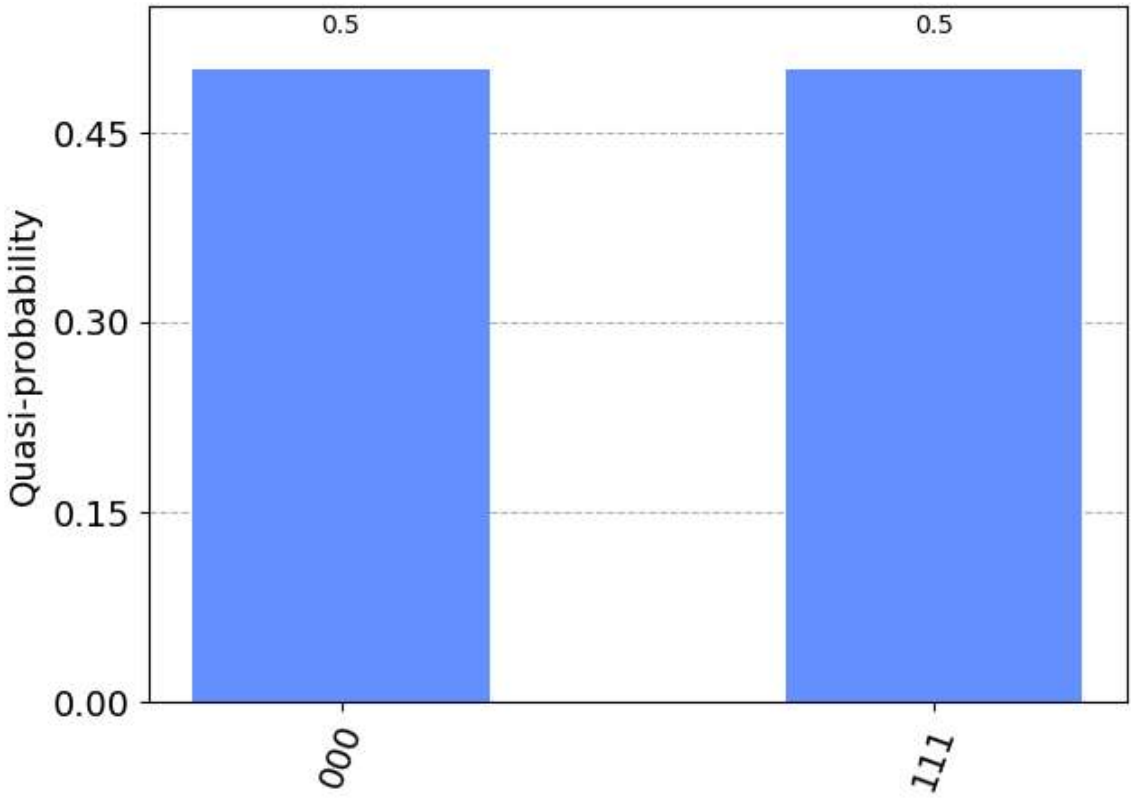


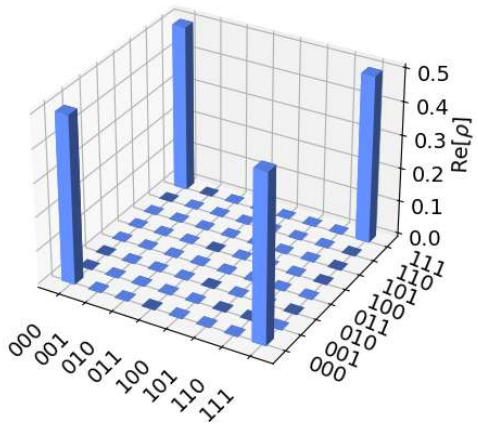
For inputs 1 1 0 Representation of GHZ States are:

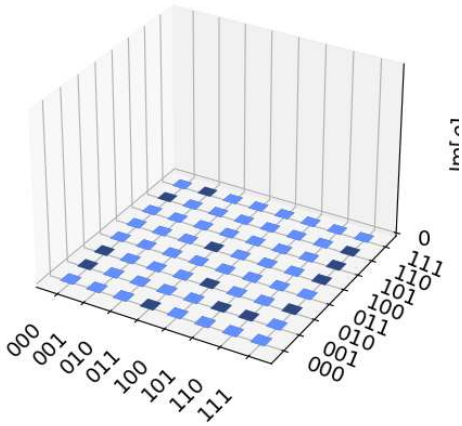


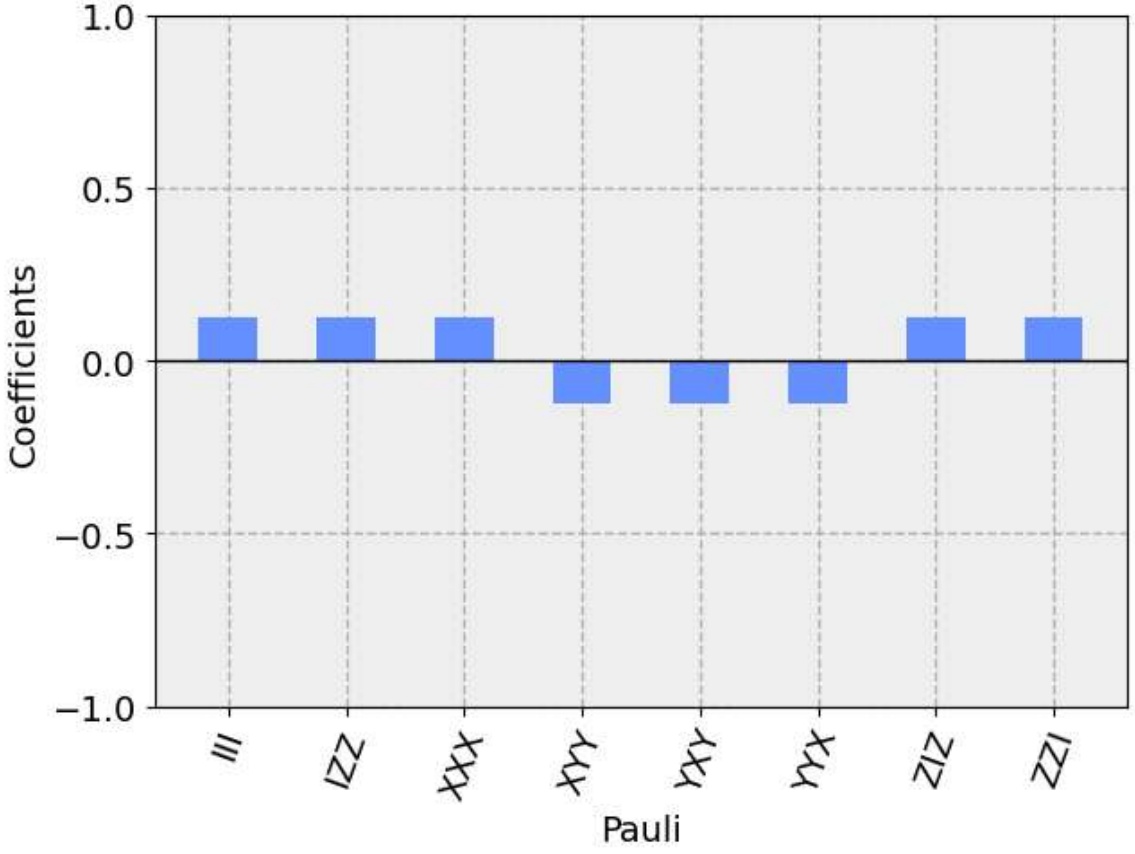

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

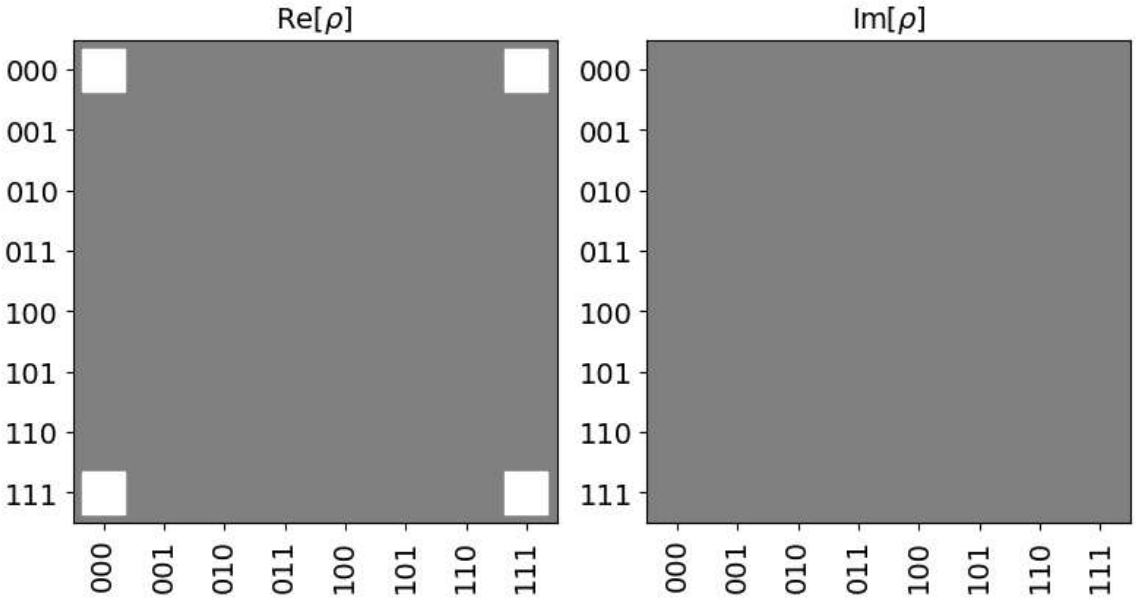

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111

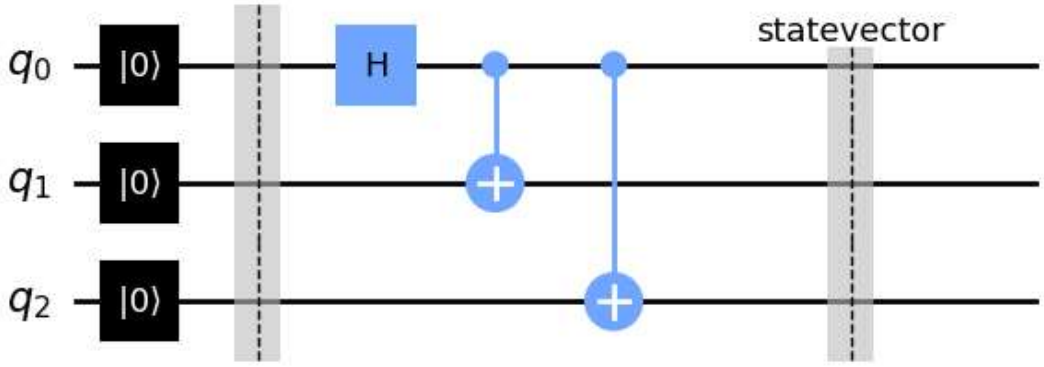

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111

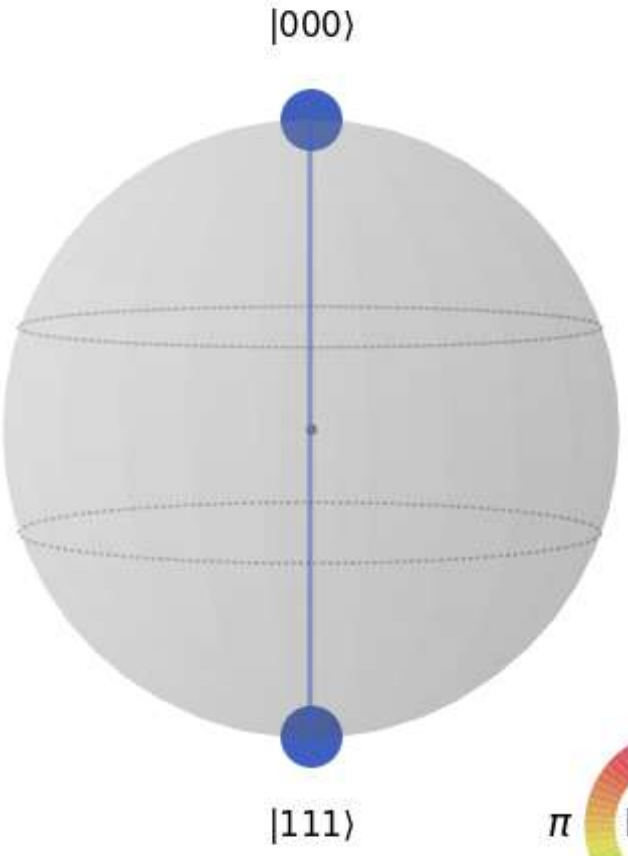

0
000
001
010
011
100
101
110
111

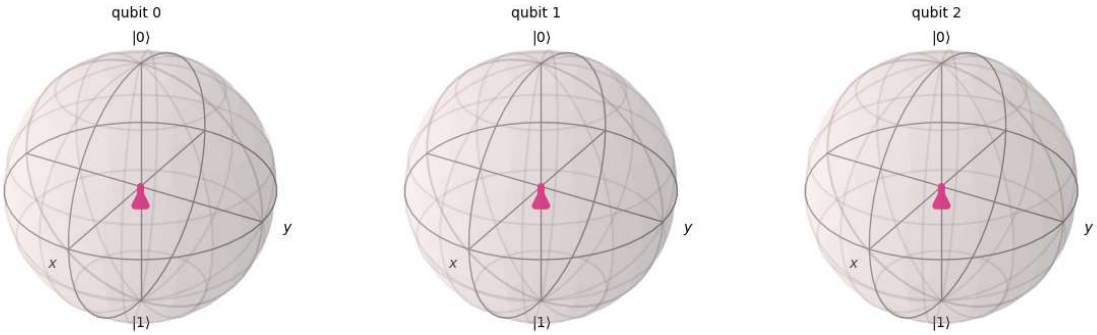


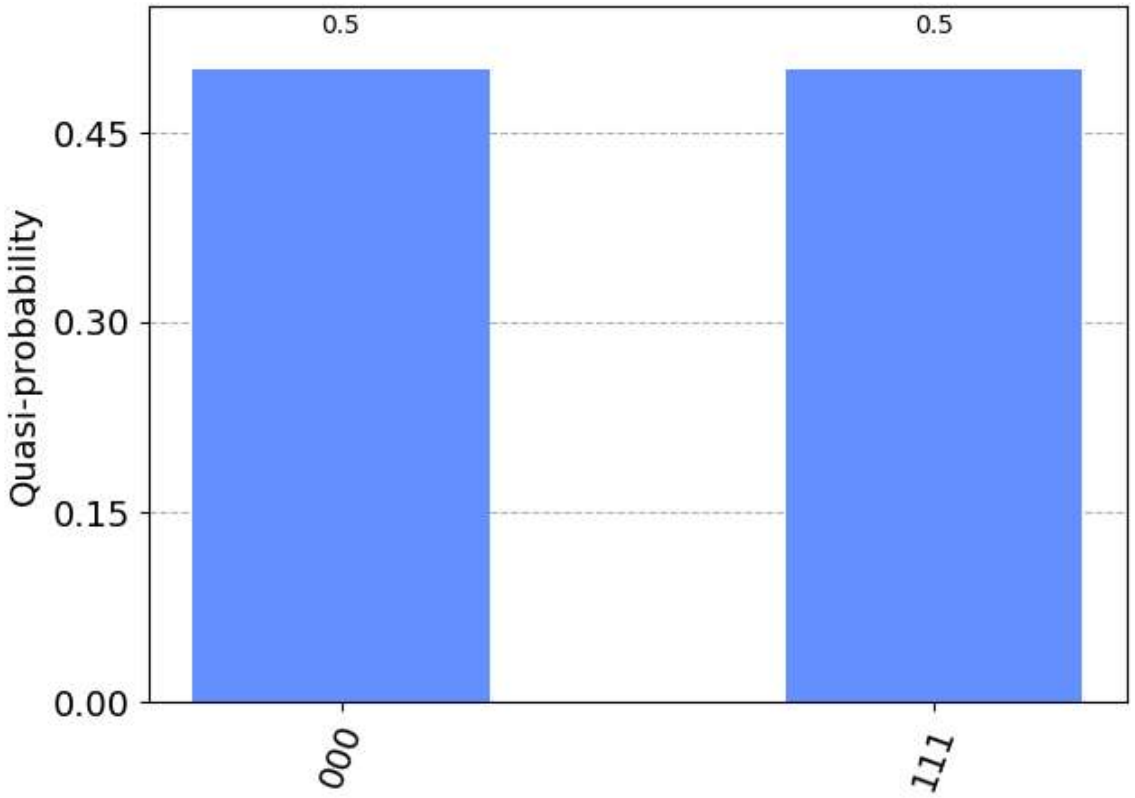


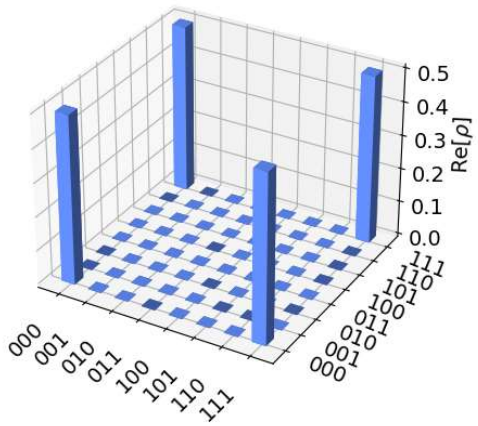
For inputs 1 1 1 Representation of GHZ States are:

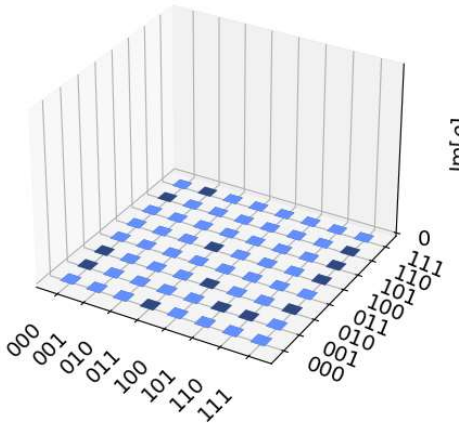


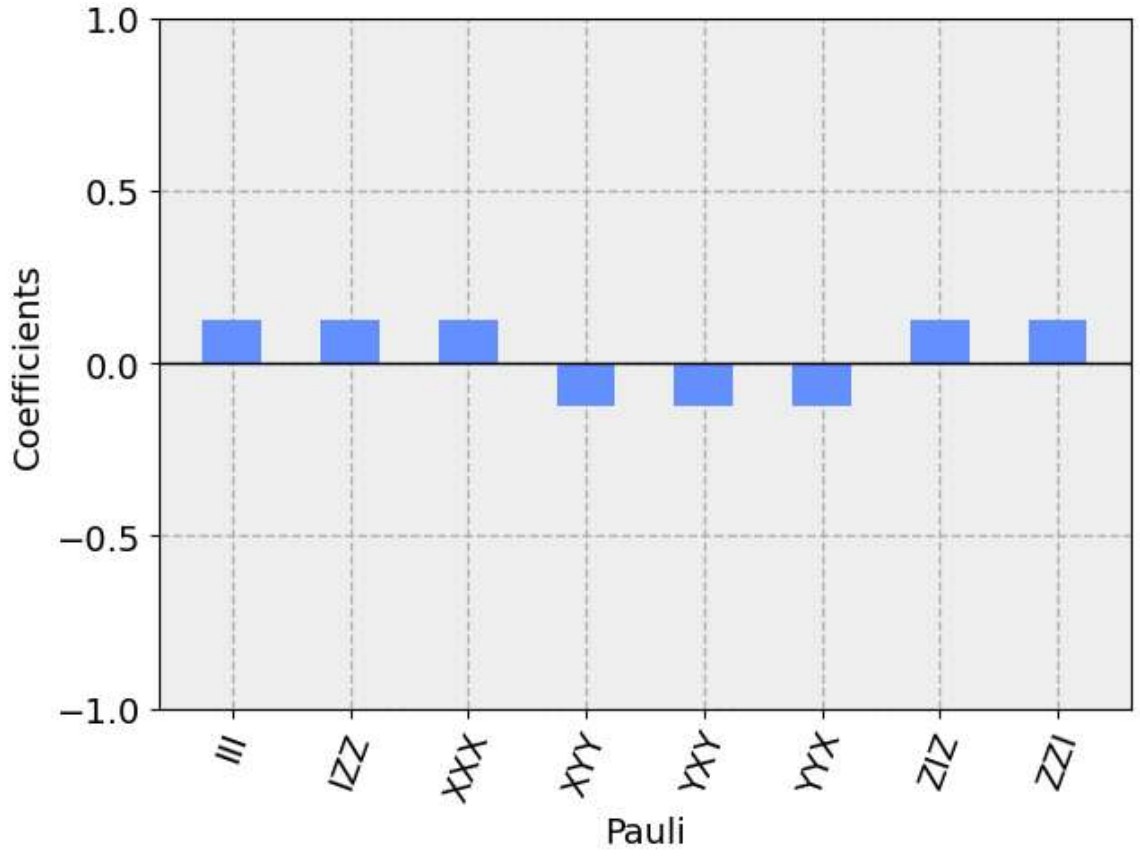

|000⟩
|111⟩
π/2
π
Phase
0
3π/2

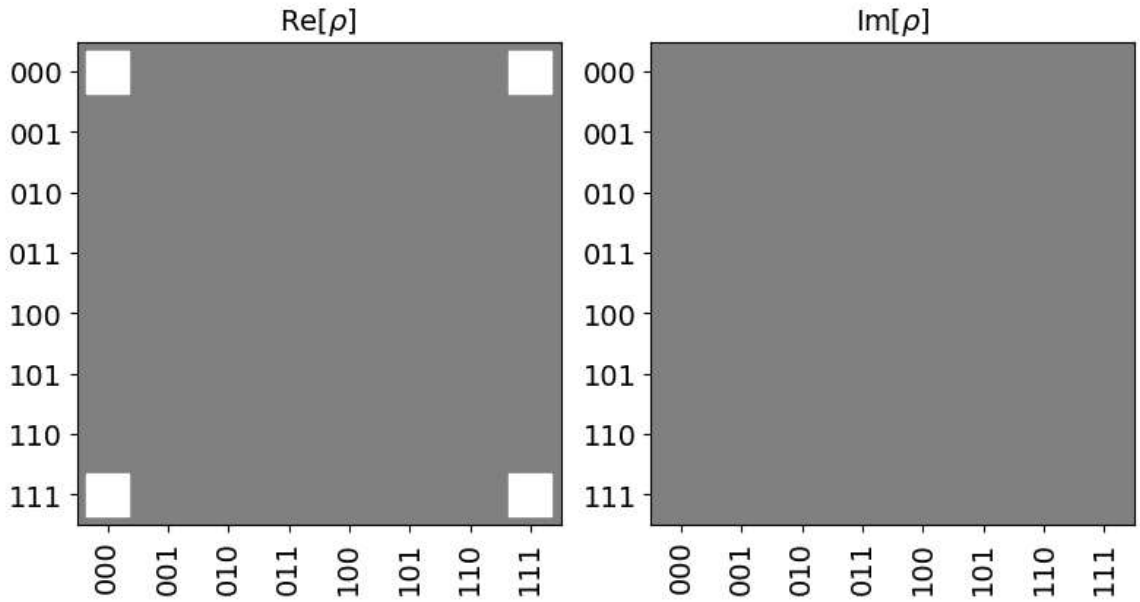

qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y


0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
000
111


0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]
000
001
010
011
100
101
110
111


0
000
001
010
011
100
101
110
111



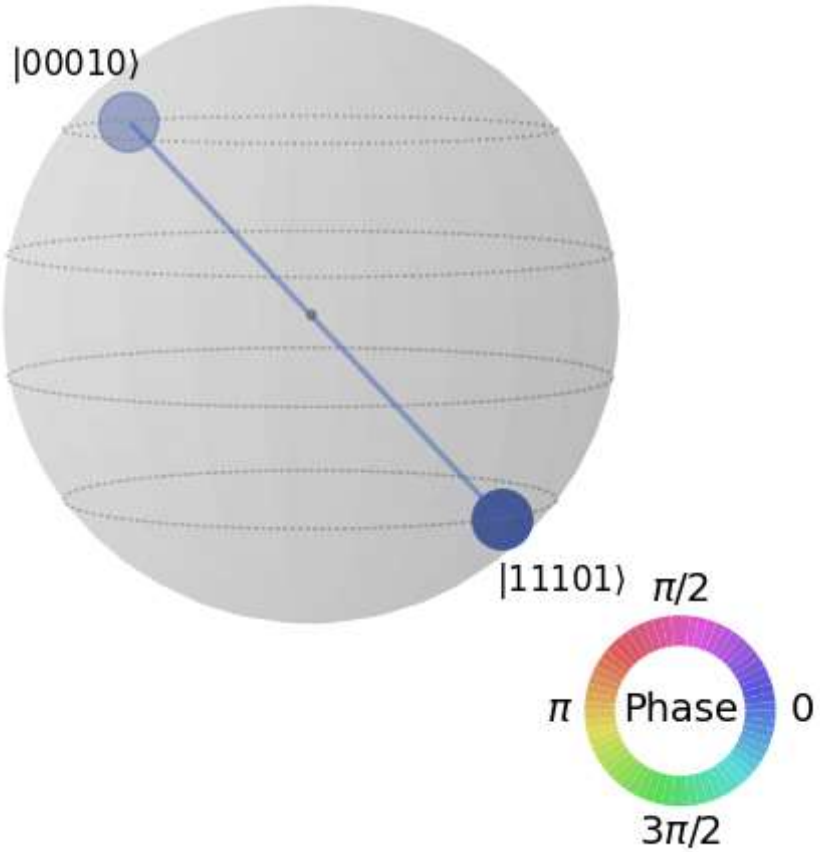


In [62]:
```
def ghz5QCircuit(inp1, inp2, inp3, inp4, inp5):

    qc = QuantumCircuit(5)
    #qc.reset(range(5))

    if inp1 == 1:
        qc.x(0)
    if inp2 == 1:
        qc.x(1)
    if inp3 == 1:
        qc.x(2)
    if inp4 == 1:
        qc.x(3)
    if inp5 == 1:
        qc.x(4)
```

```
    qc.barrier()

    qc.h(0)
    qc.cx(0,1)
    qc.cx(0,2)
    qc.cx(0,3)
    qc.cx(0,4)

    qc.save_statevector()
    qobj = assemble(qc)
    result = sim.run(qobj).result()
    state = result.get_statevector()

    return qc, state, result
```

In [63]:
```
# Explore GHZ States for input 00010. Note: the input has been stated in little-
inp1 = 0
inp2 = 1
inp3 = 0
inp4 = 0
inp5 = 0

qc, state, result = ghz5QCircuit(inp1, inp2, inp3, inp4, inp5)

print('For inputs',inp5,inp4,inp3,inp2,inp1,'Representation of GHZ States are:')
display(plot_state_qsphere(state))
print('\n')
```

```
For inputs 0 0 0 1 0 Representation of GHZ States are:
```

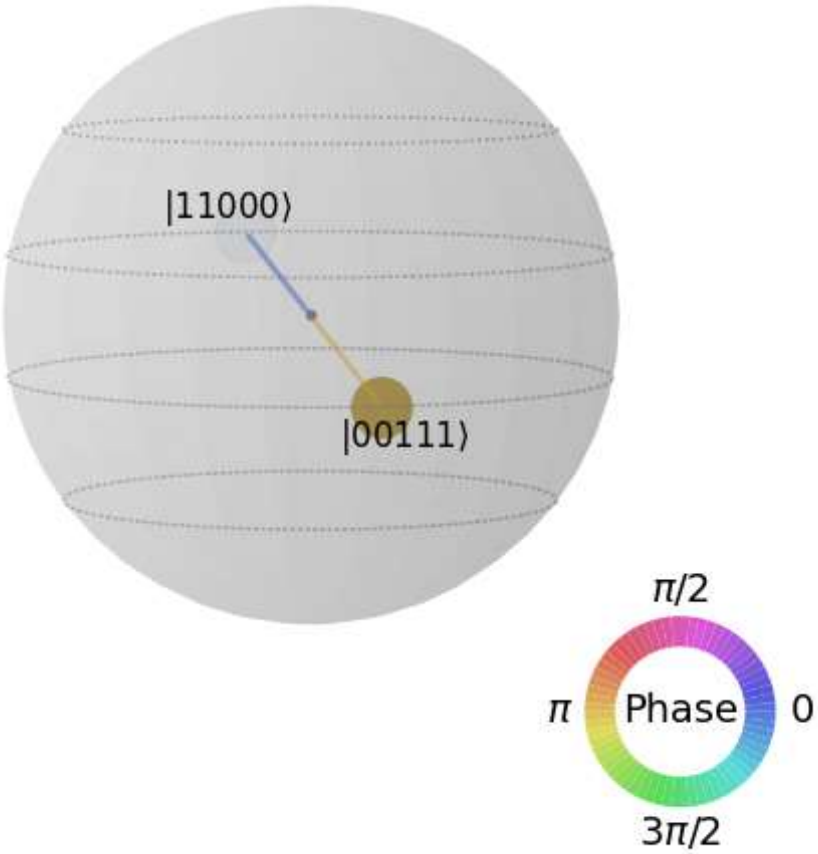


In [ ]:

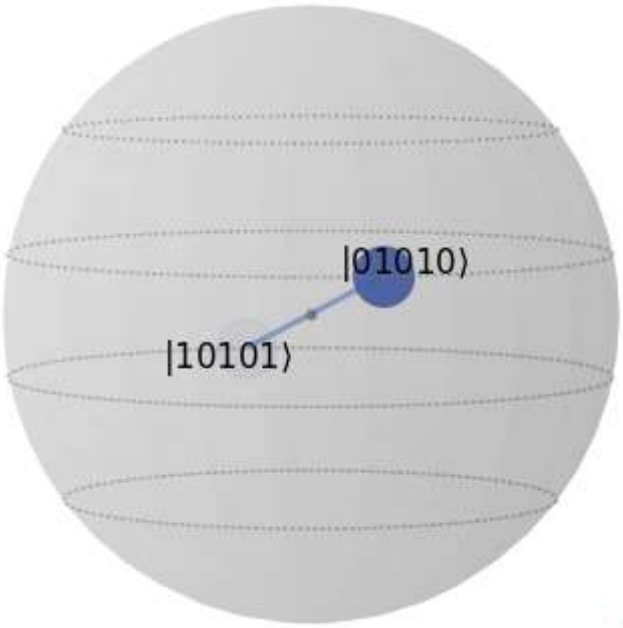
```
# Explore GHZ States for input 11001. Note: the input has been stated in little-
inp1 = 1
inp2 = 0
inp3 = 0
inp4 = 1
inp5 = 1

qc, state, result = ghz5QCircuit(inp1, inp2, inp3, inp4, inp5)

print('For inputs',inp5,inp4,inp3,inp2,inp1,'Representation of GHZ States are:')
display(plot_state_qsphere(state))
print('\n')

# Explore GHZ States for input 01010. Note: the input has been stated in little-
inp1 = 0
inp2 = 1
inp3 = 0
inp4 = 1
inp5 = 0

qc, state, result = ghz5QCircuit(inp1, inp2, inp3, inp4, inp5)

print('For inputs',inp5,inp4,inp3,inp2,inp1,'Representation of GHZ States are:')
display(plot_state_qsphere(state))
print('\n')
```

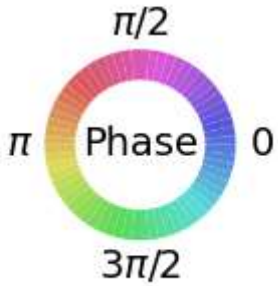
```
# Uncomment below code in order to explore other states
for inp5 in ['0','1']:
    for inp4 in ['0','1']:
        for inp3 in ['0','1']:
            for inp2 in ['0','1']:
                for inp1 in ['0','1']:
                    qc, state, result = ghz5QCircuit(inp1, inp2, inp3, inp4, inp

                    #print('For inputs',inp5,inp4,inp3,inp2,inp1,'Representation

                    # Uncomment any of the below functions to visualize the resu

                    # Draw the quantum circuit
                    display(qc.draw())

                    # Plot states on QSphere
                    display(plot_state_qsphere(state))

                    # Plot states on Bloch Multivector
                    display(plot_bloch_multivector(state))

                    # Plot histogram
                    display(plot_histogram(result.get_counts()))

                    # Plot state matrix like a city
                    display(plot_state_city(state))

                    # Represent state matix using Pauli operators as the basis
                    display(plot_state_paulivec(state))

                    # Plot state matrix as Hinton representation
                    display(plot_state_hinton(state))

                    #print('\n')
```

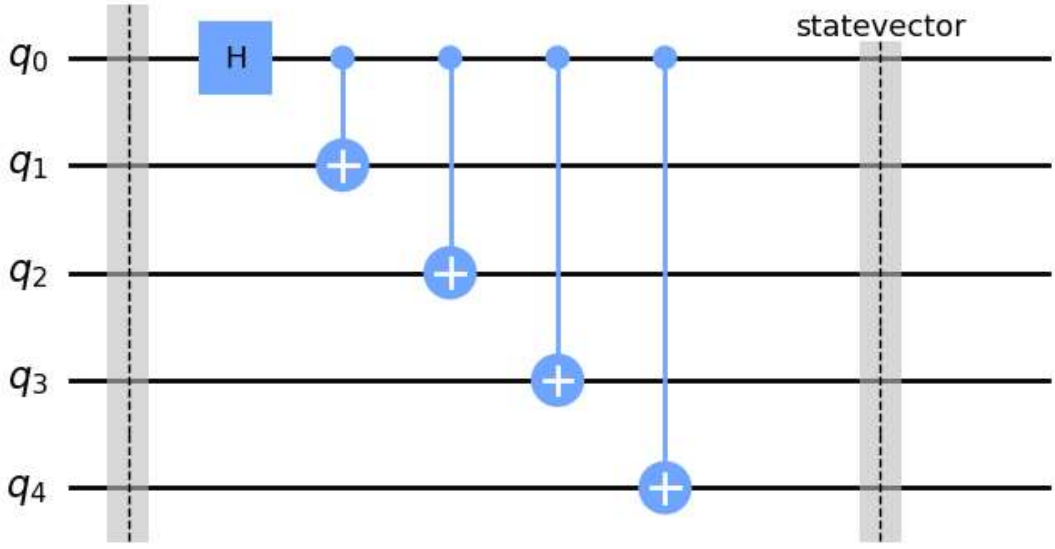
```
For inputs 1 1 0 0 1 Representation of GHZ States are:
```

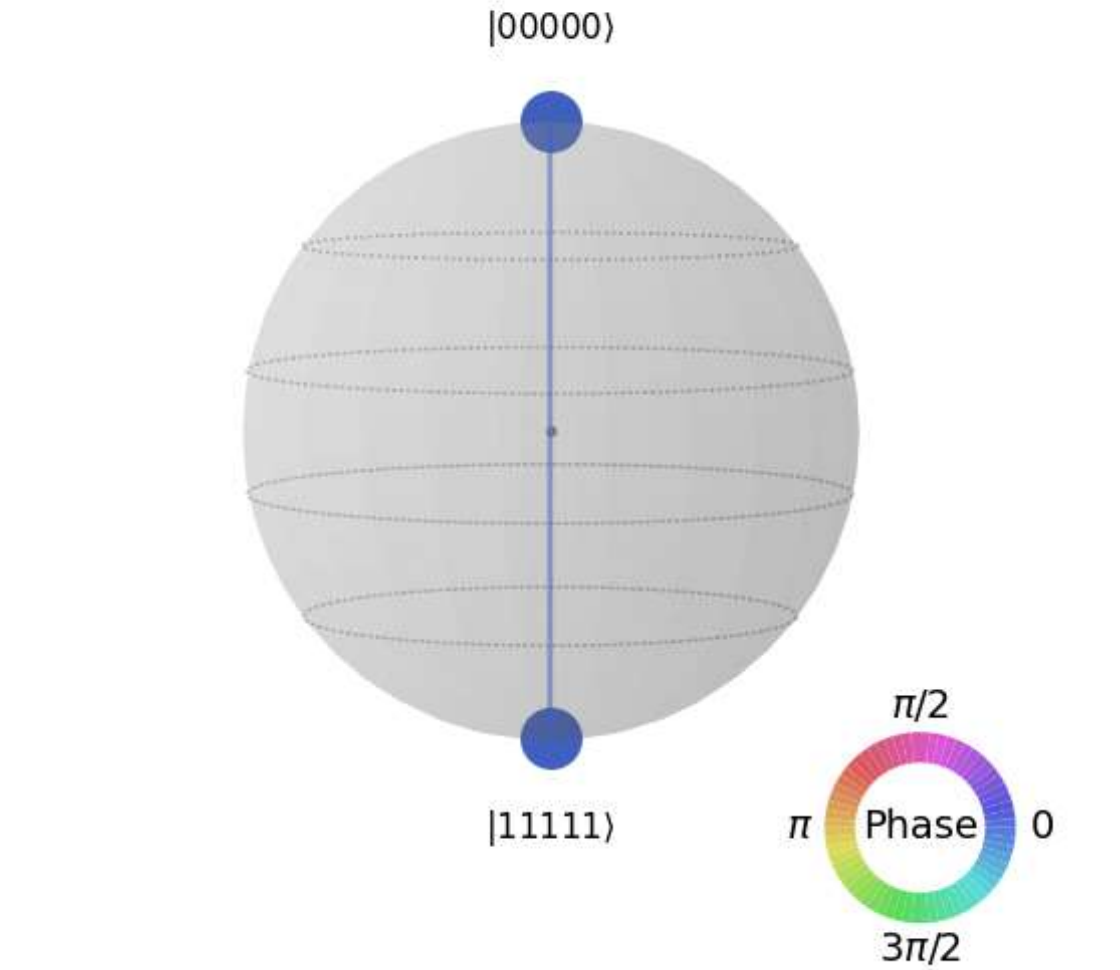


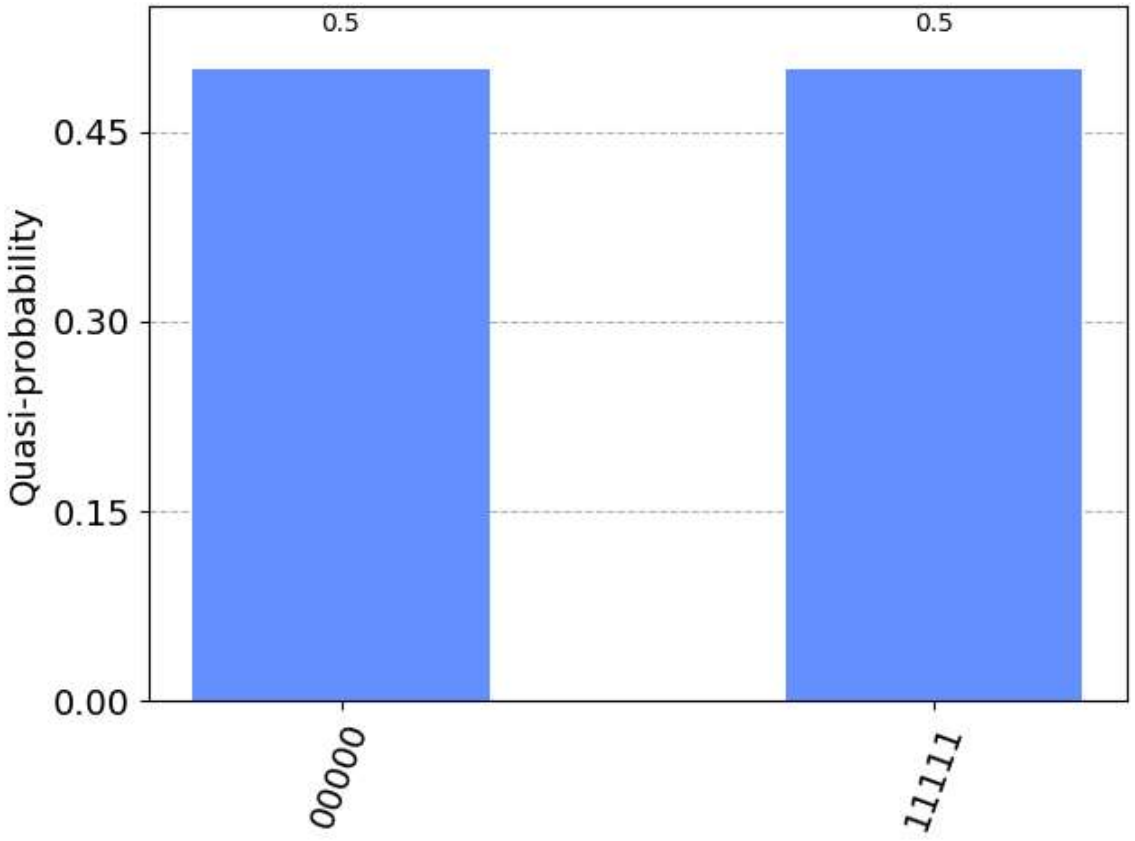
For inputs 0 1 0 1 0 Representation of GHZ States are:

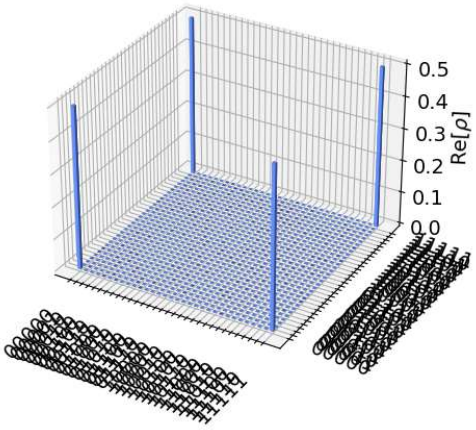

|01010⟩
|10101⟩

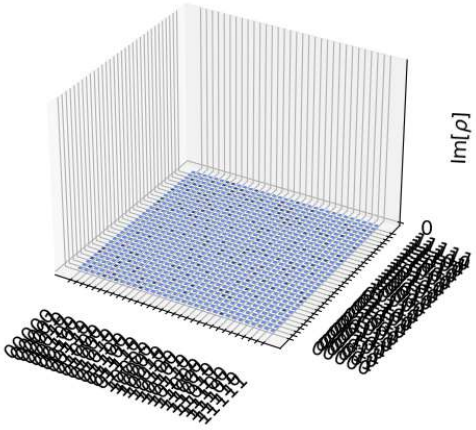

π/2
π
Phase
0
3π/2

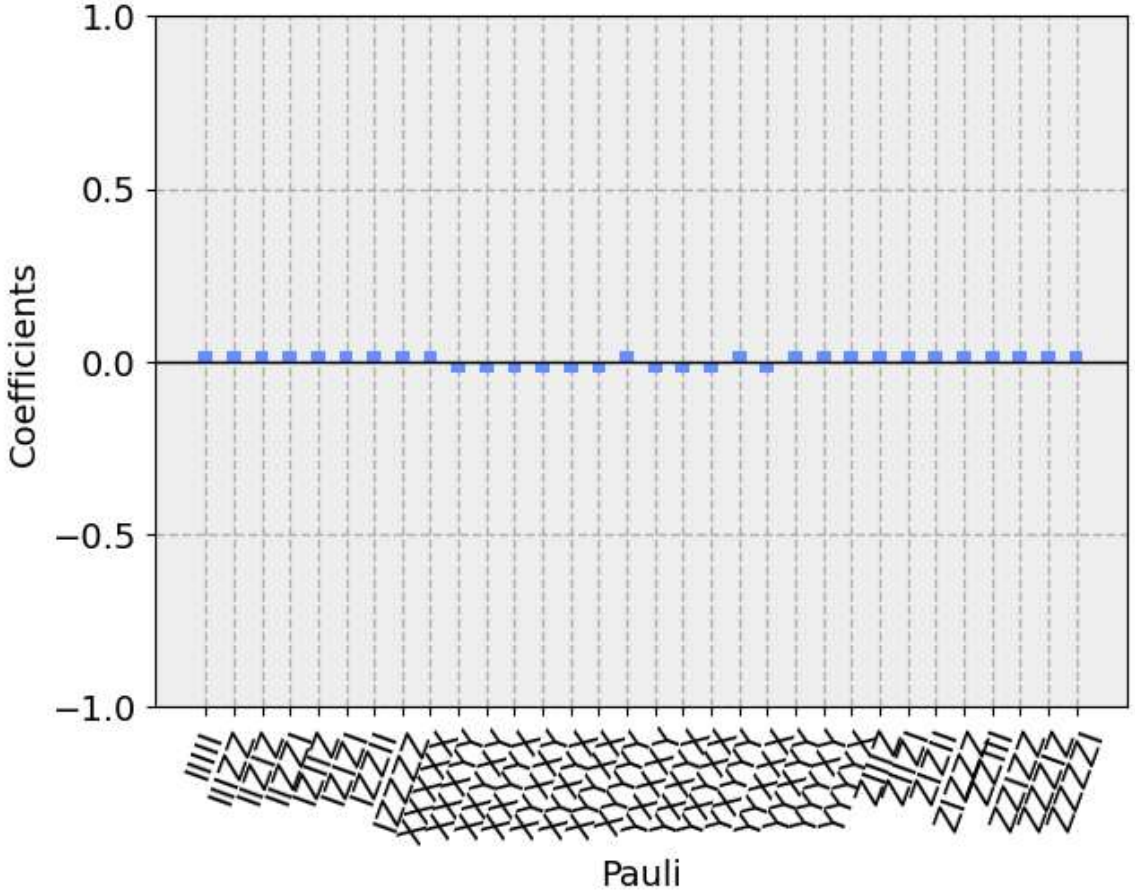

q0
q1
q2
q3
q4
H
statevector

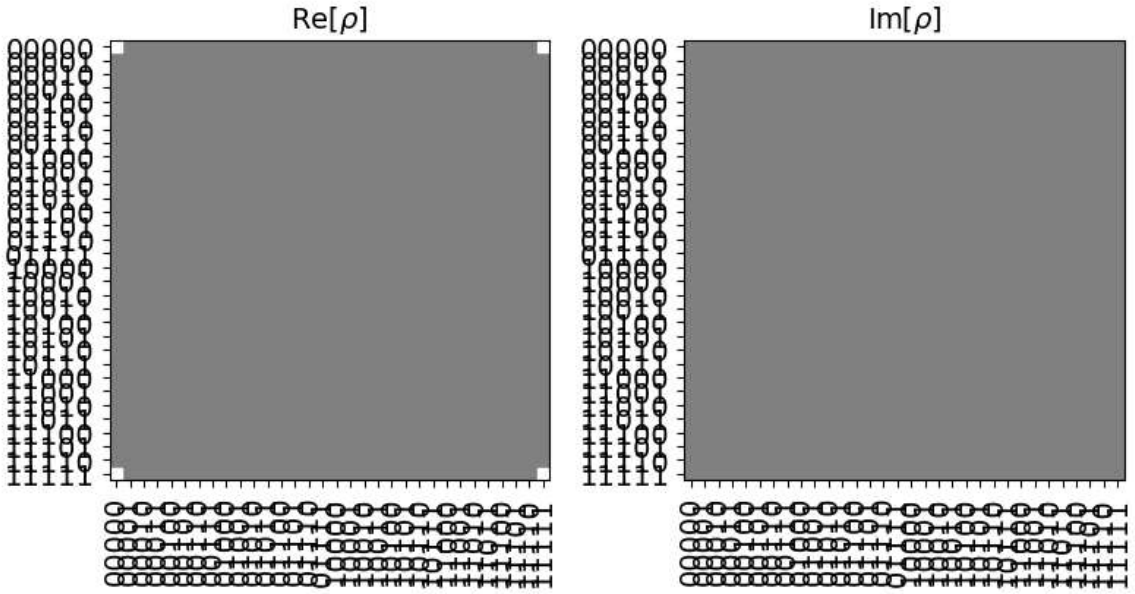

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2
qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

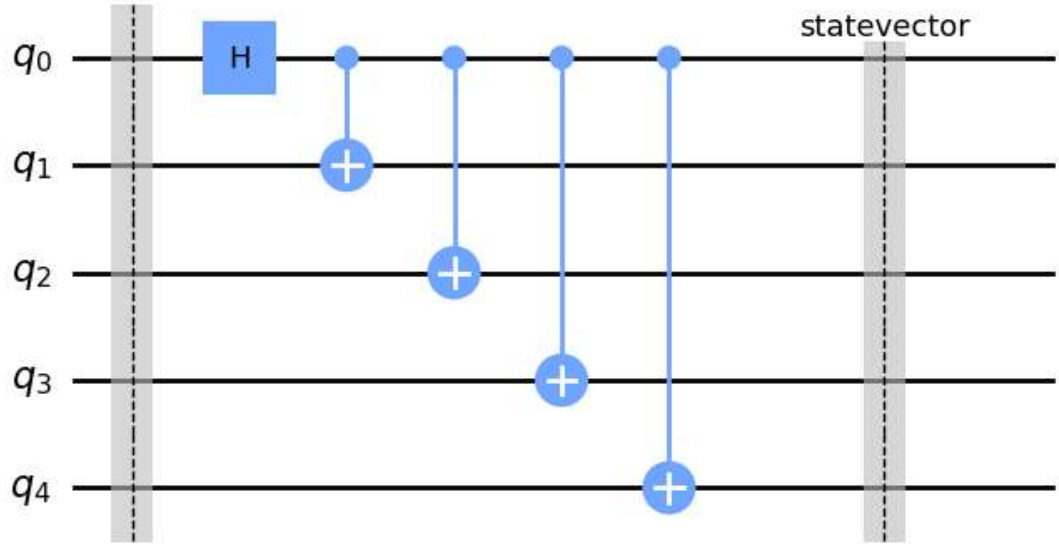

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

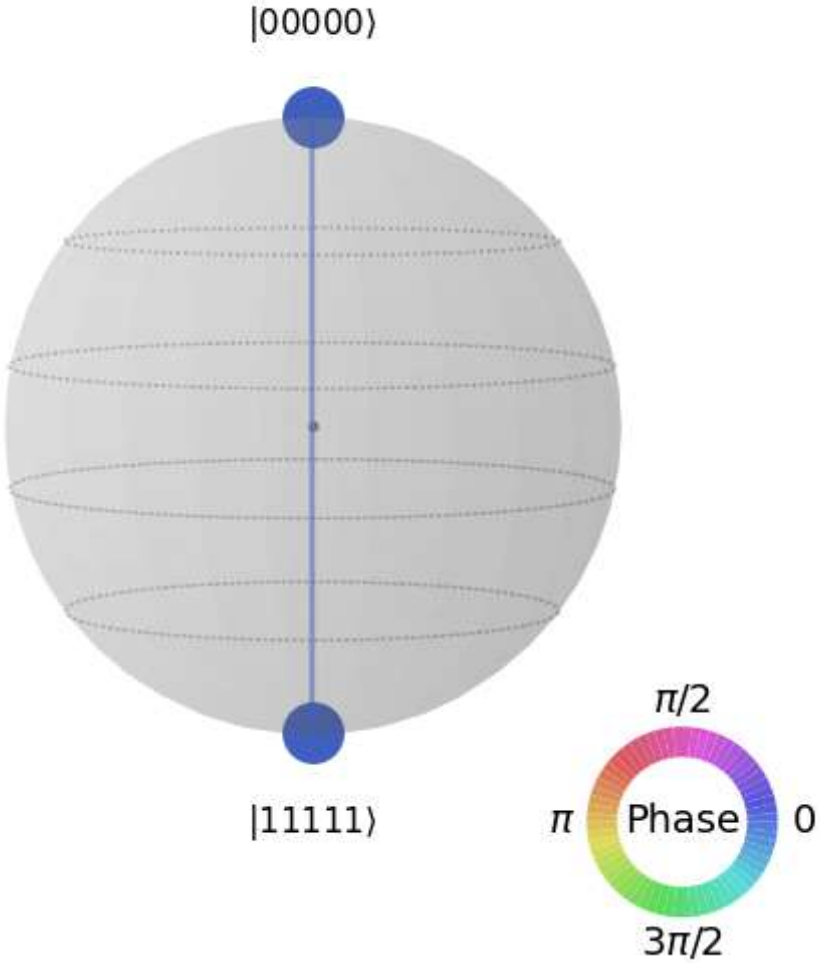

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

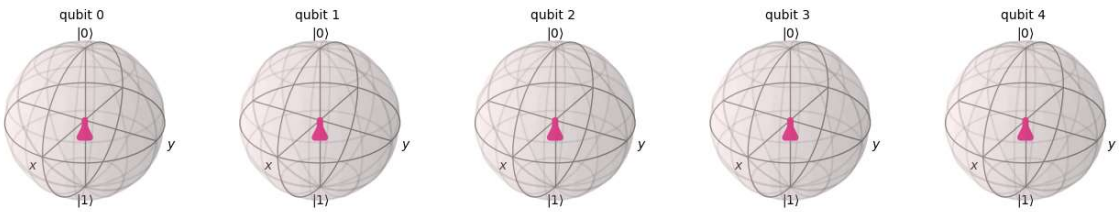

Im[ρ]
0

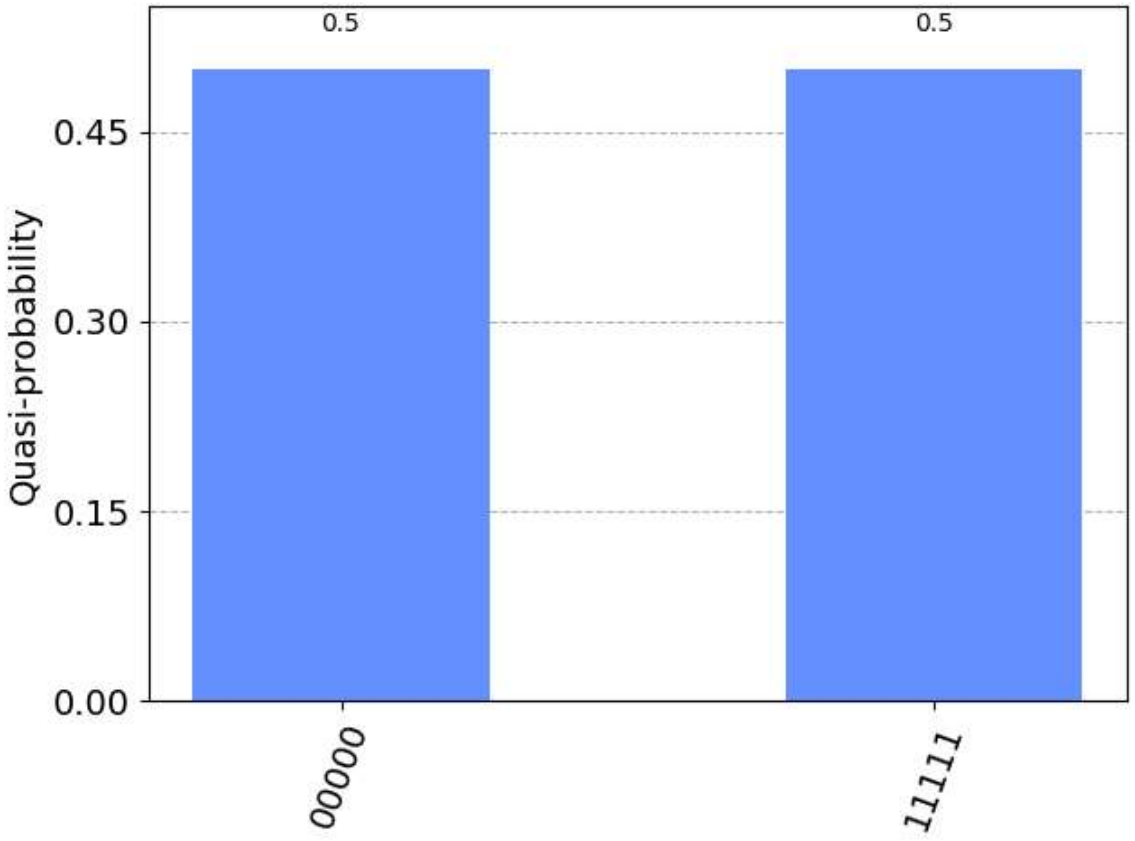

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

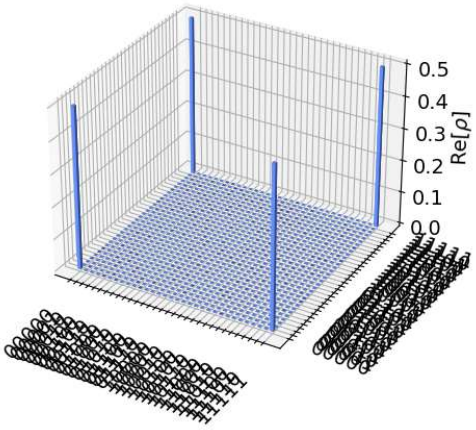

Re[ρ]
Im[ρ]

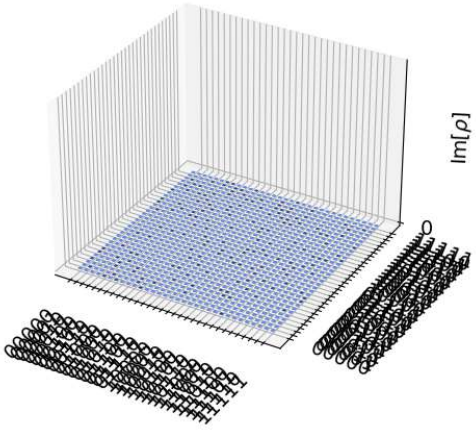

statevector
H
q_0
q_1
q_2
q_3
q_4

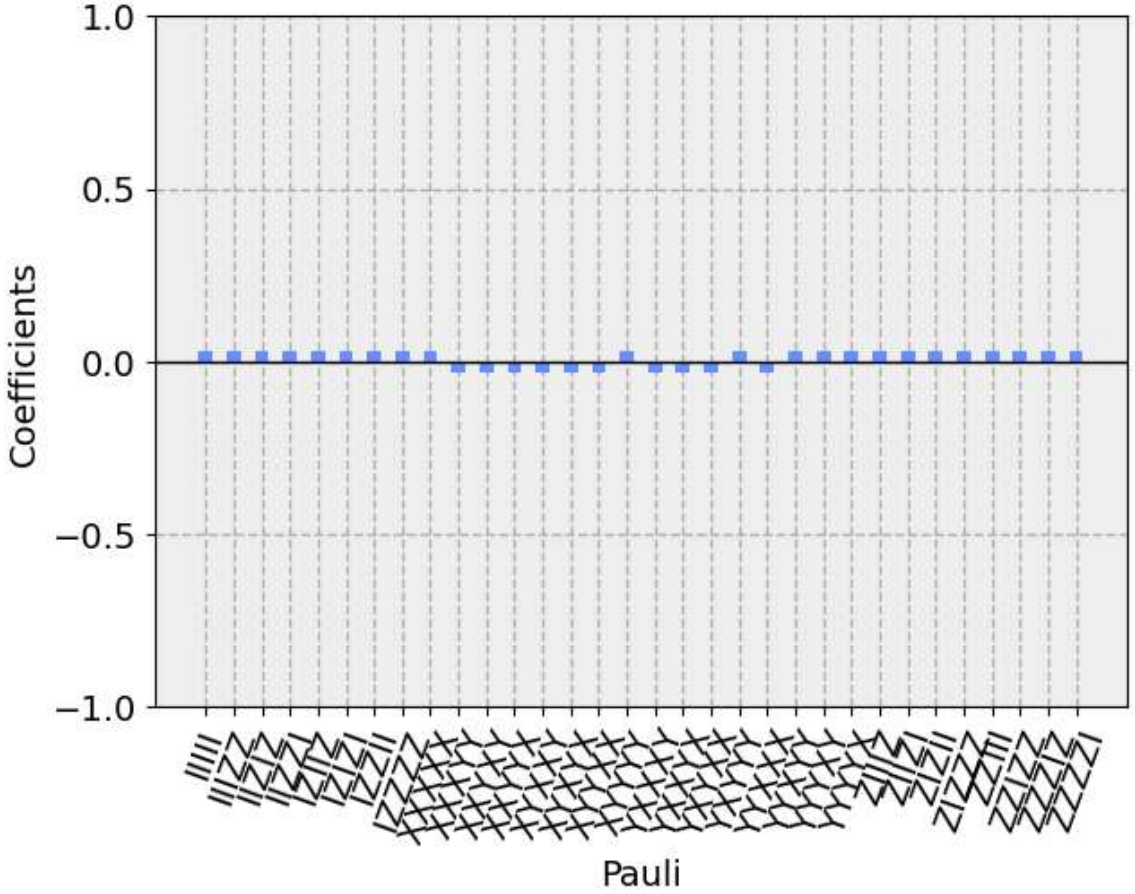

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

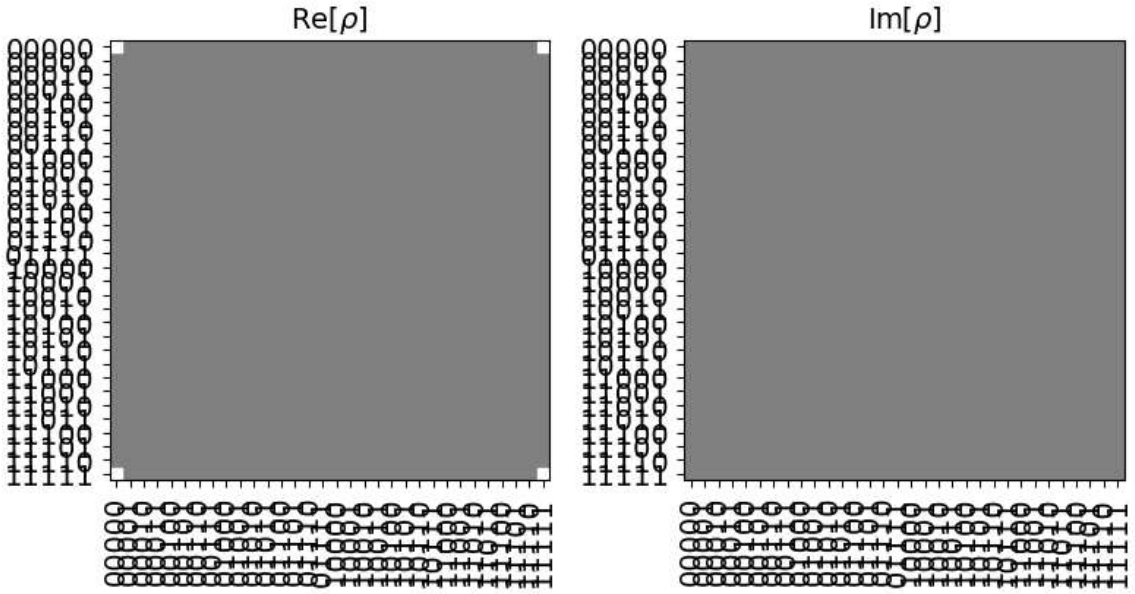

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

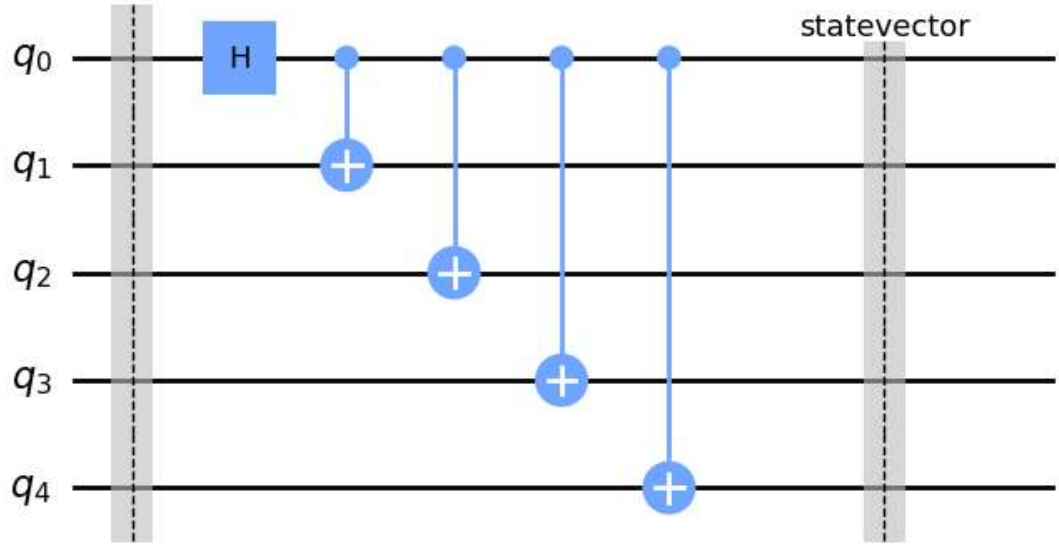

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

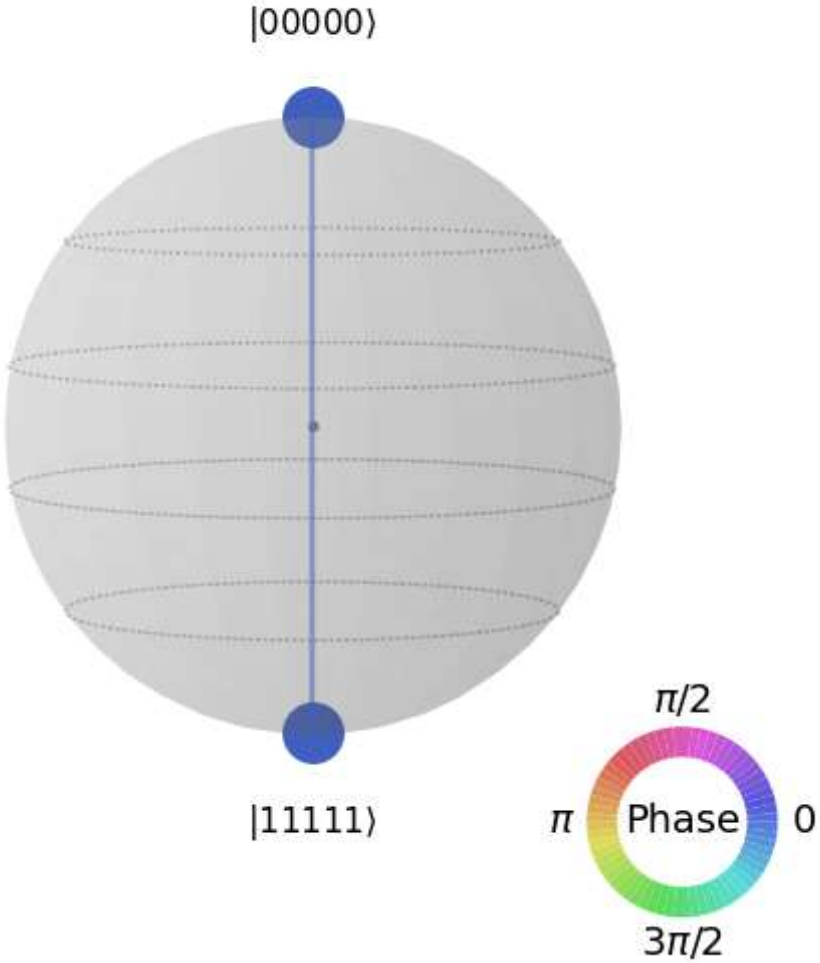

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

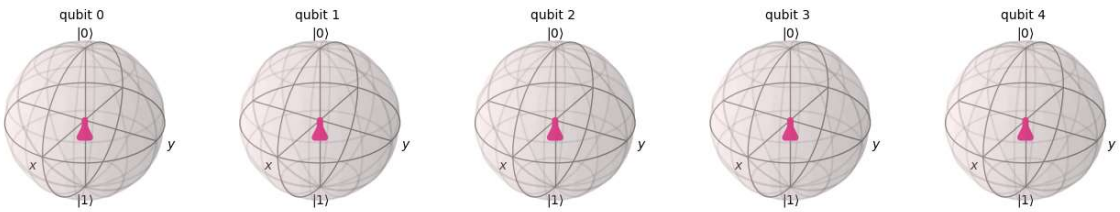

Im[ρ]
0

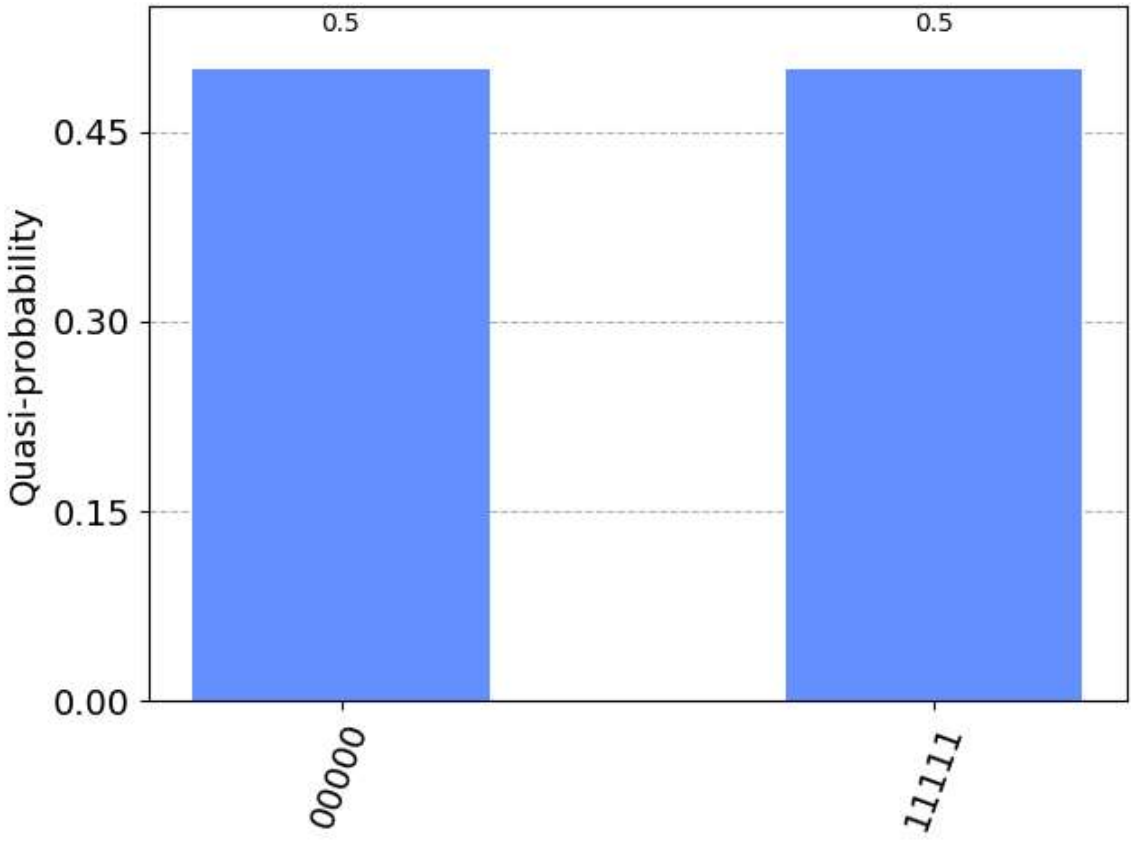

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

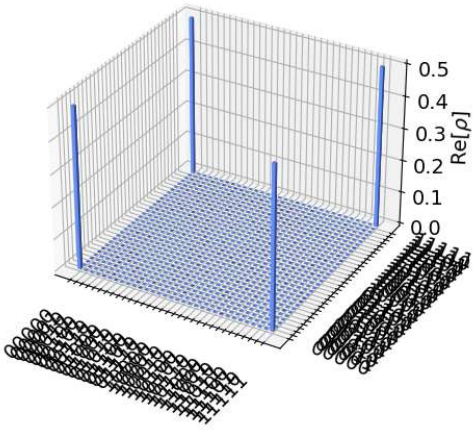

Re[ρ]
Im[ρ]

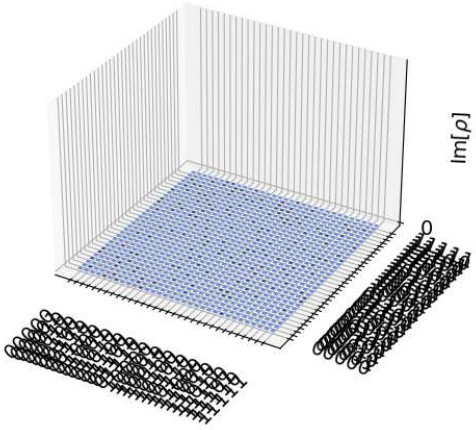

statevector
H
q_0
q_1
q_2
q_3
q_4

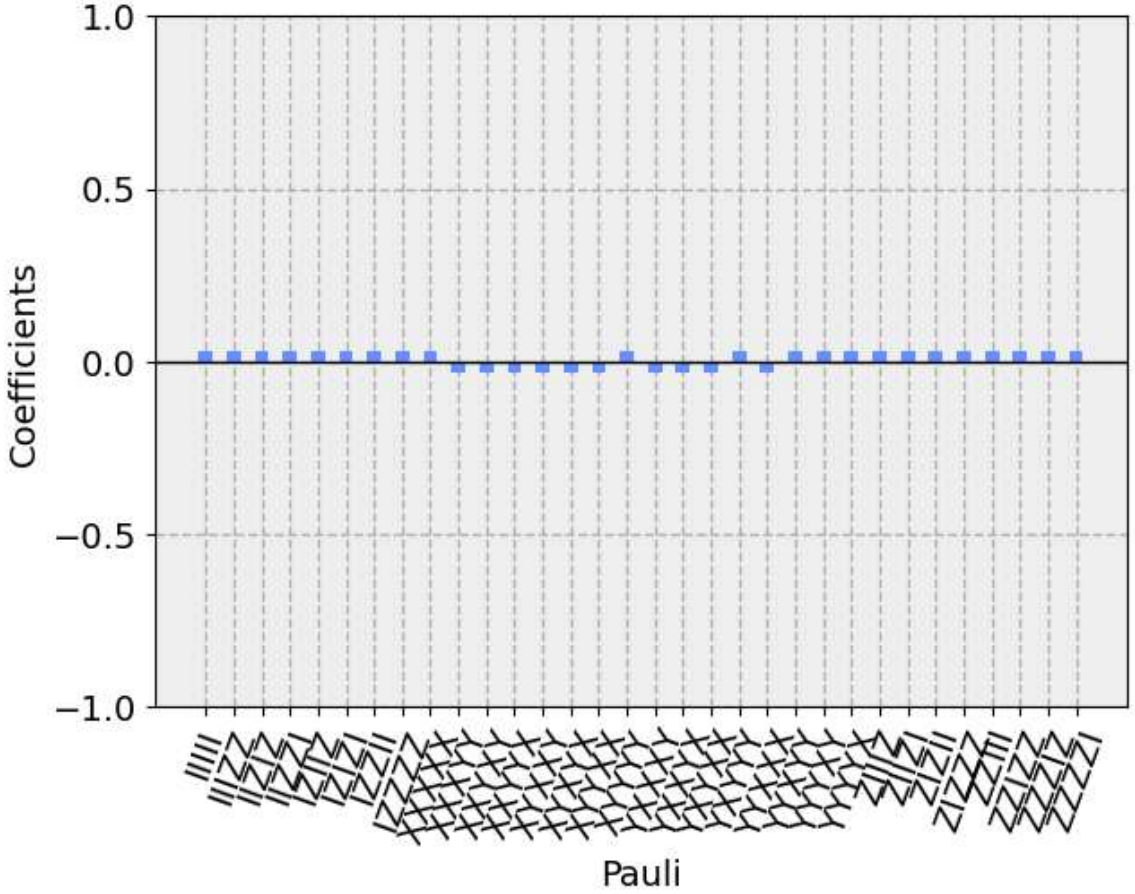

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

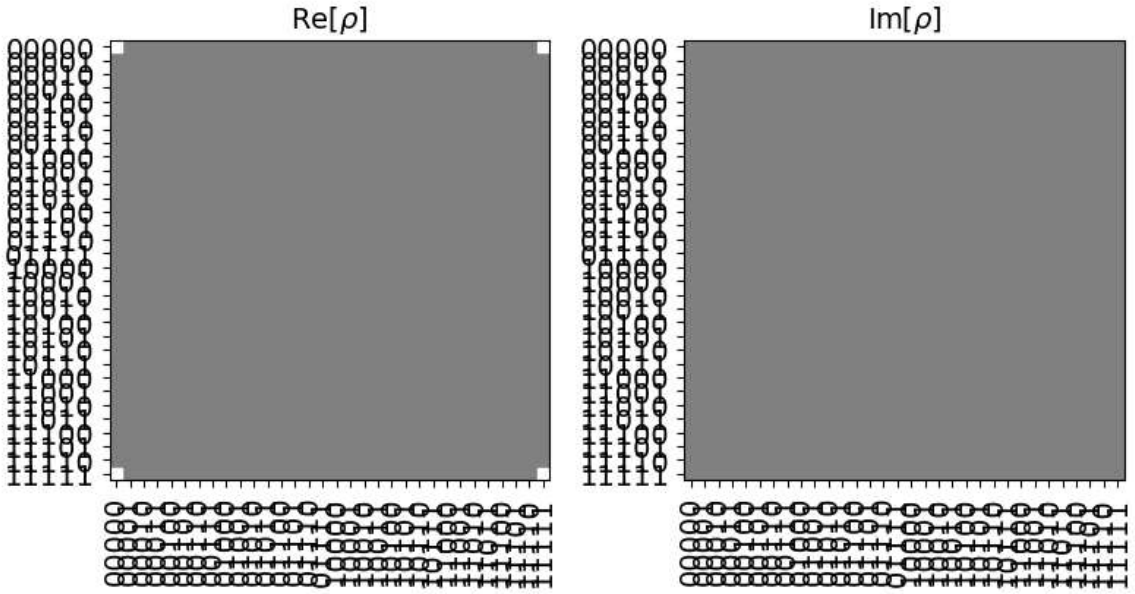

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

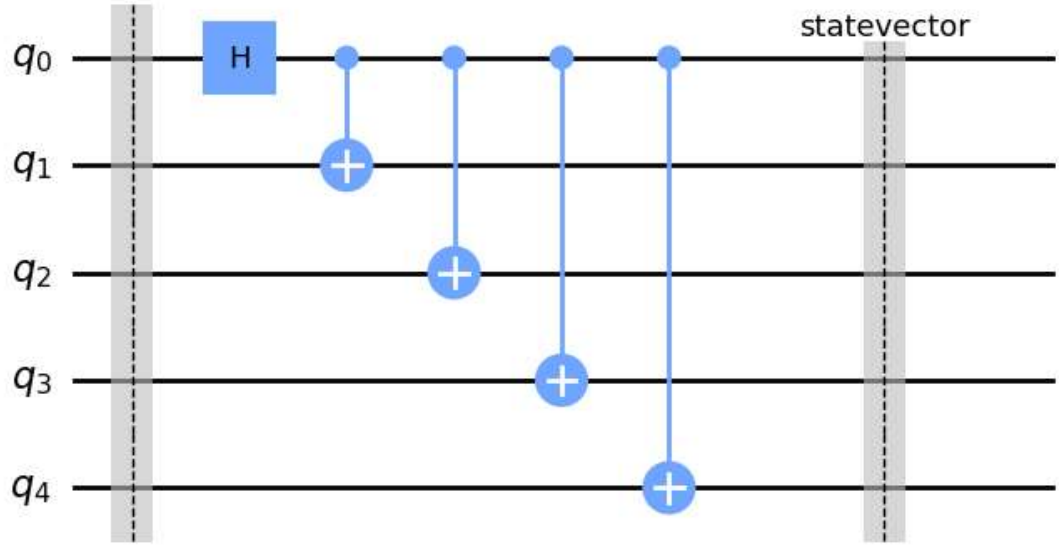

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

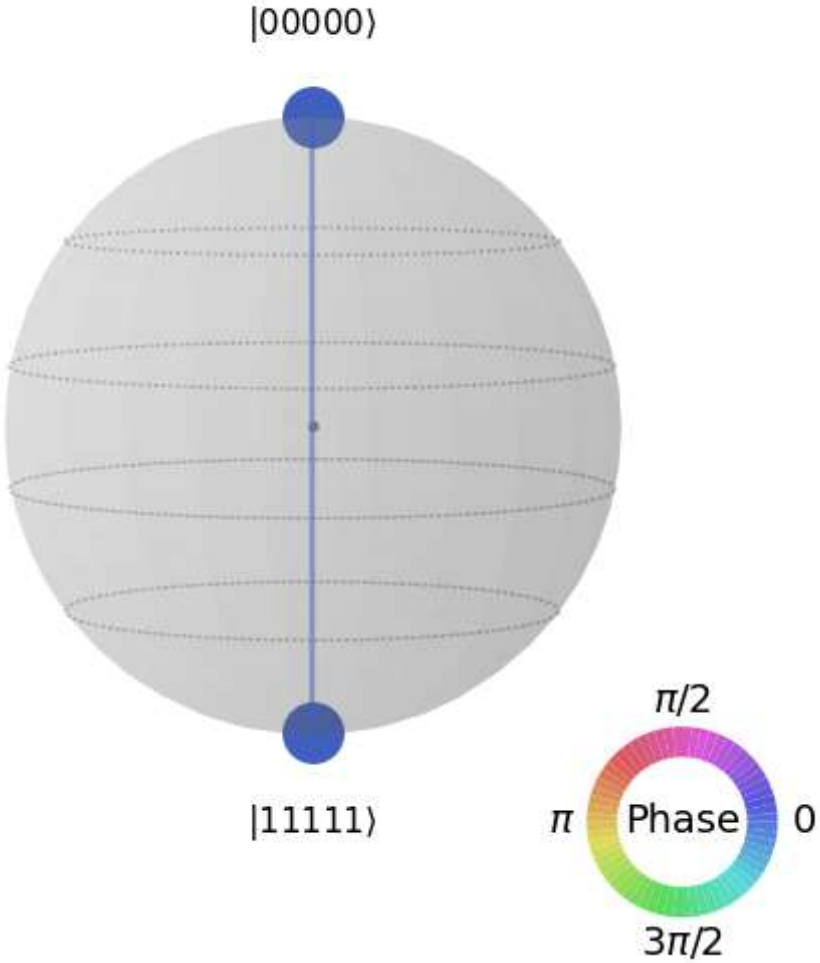

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

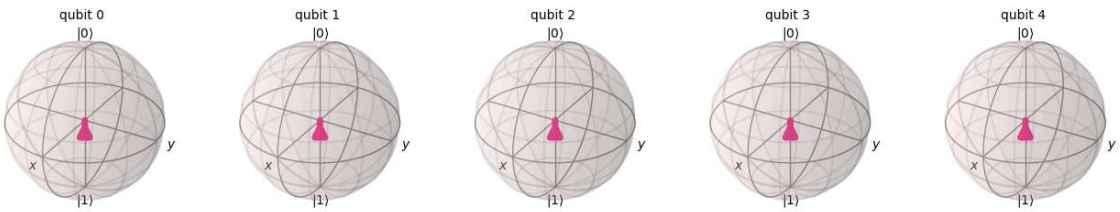

Im[ρ]
0

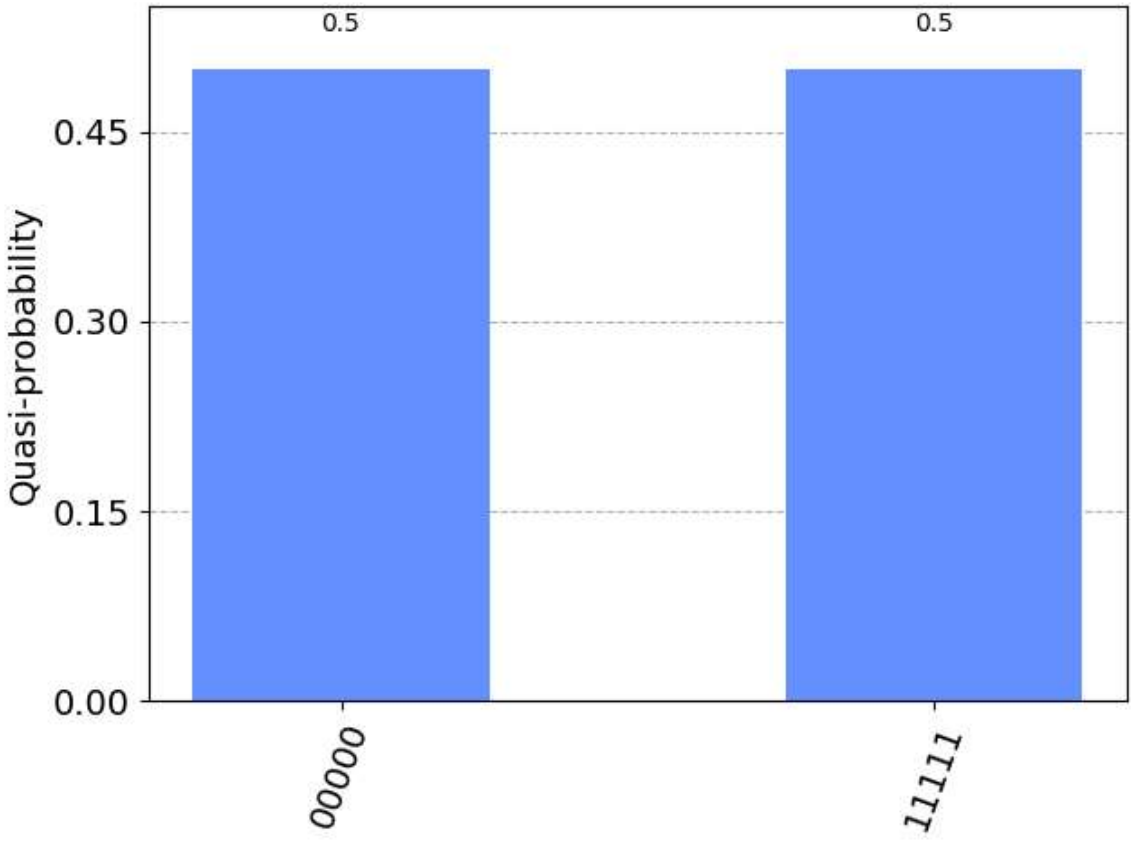

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

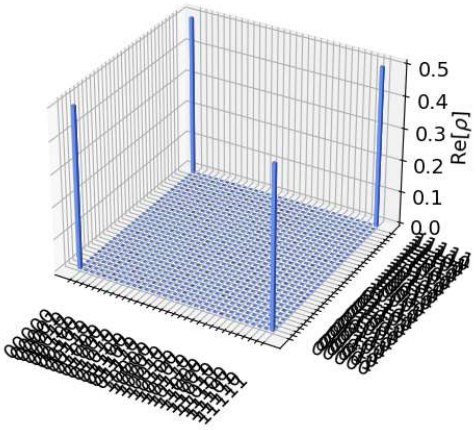

Re[ρ]
Im[ρ]

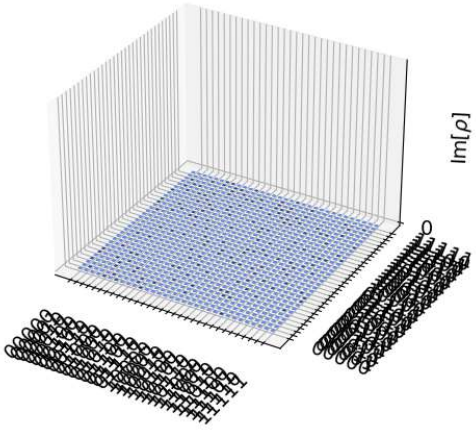

statevector
H
q_0
q_1
q_2
q_3
q_4

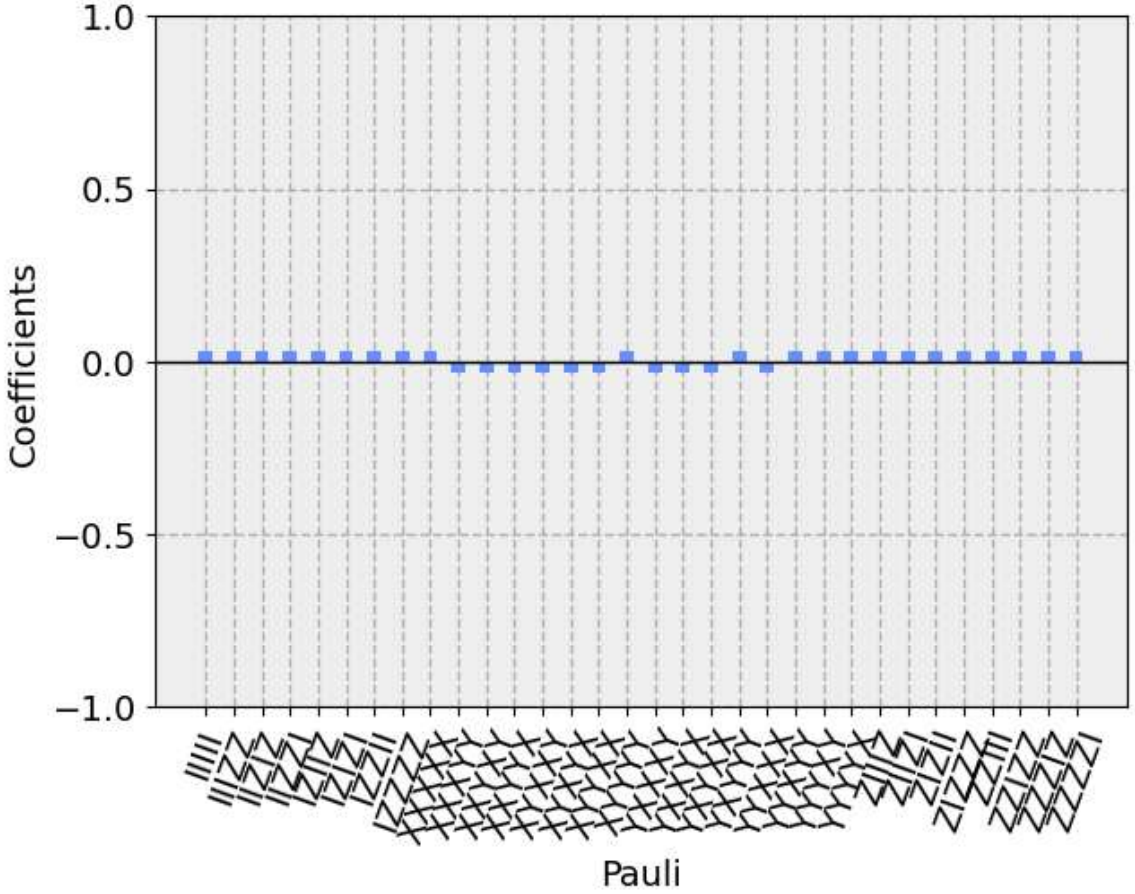

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

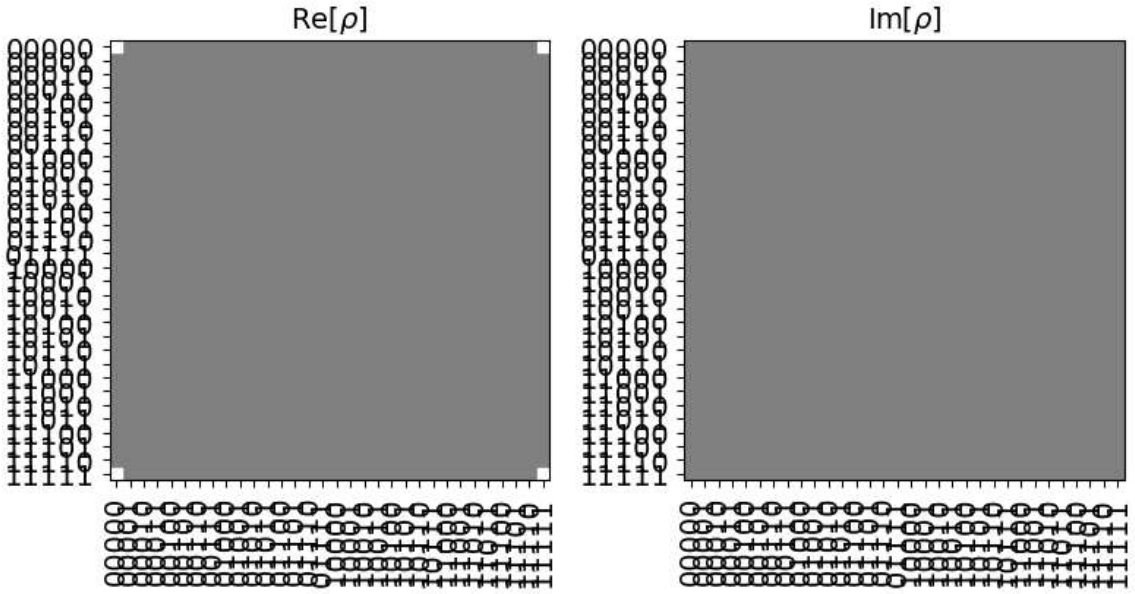

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

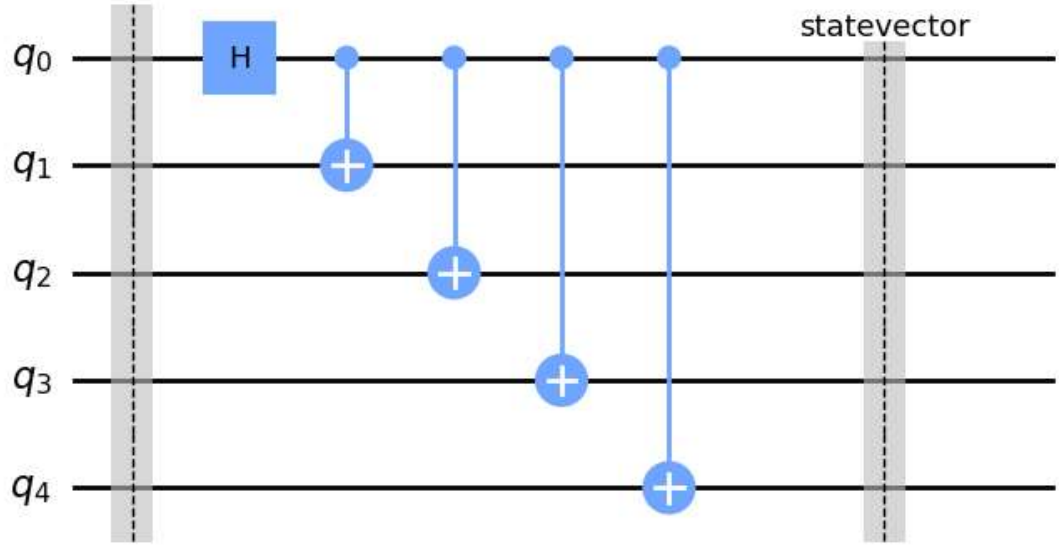

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

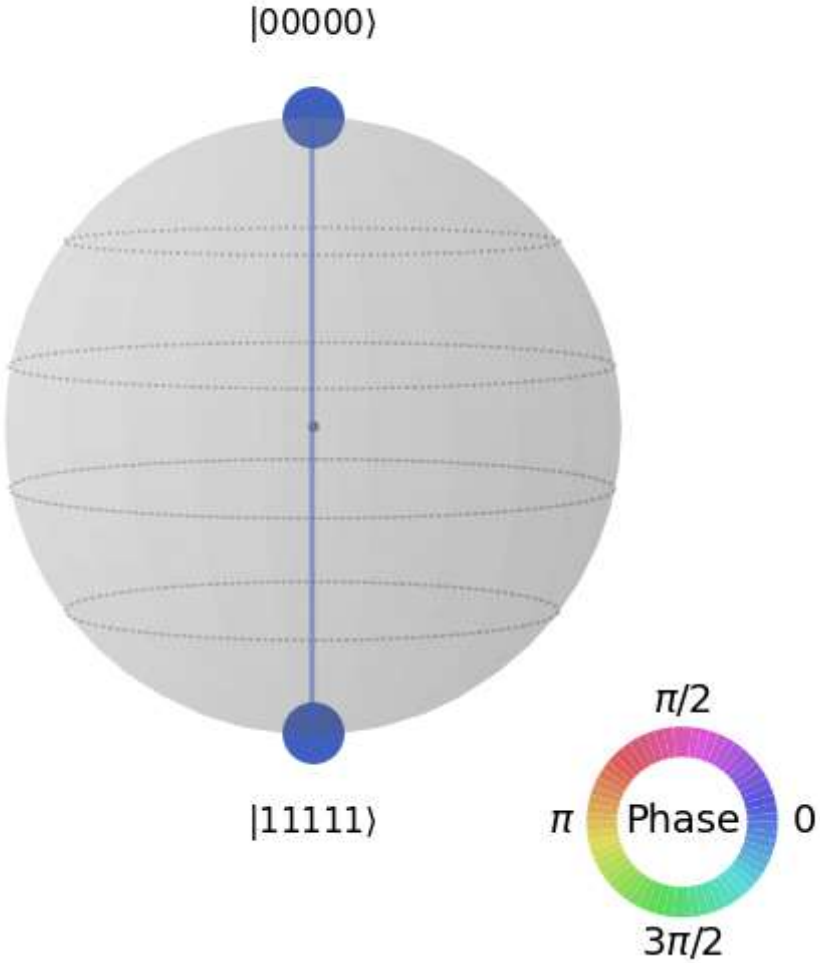

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

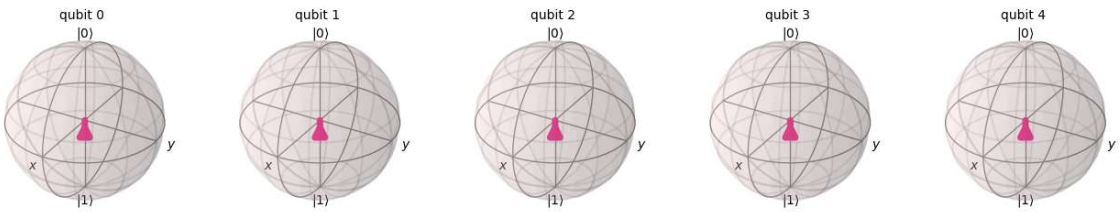

Im[ρ]
0

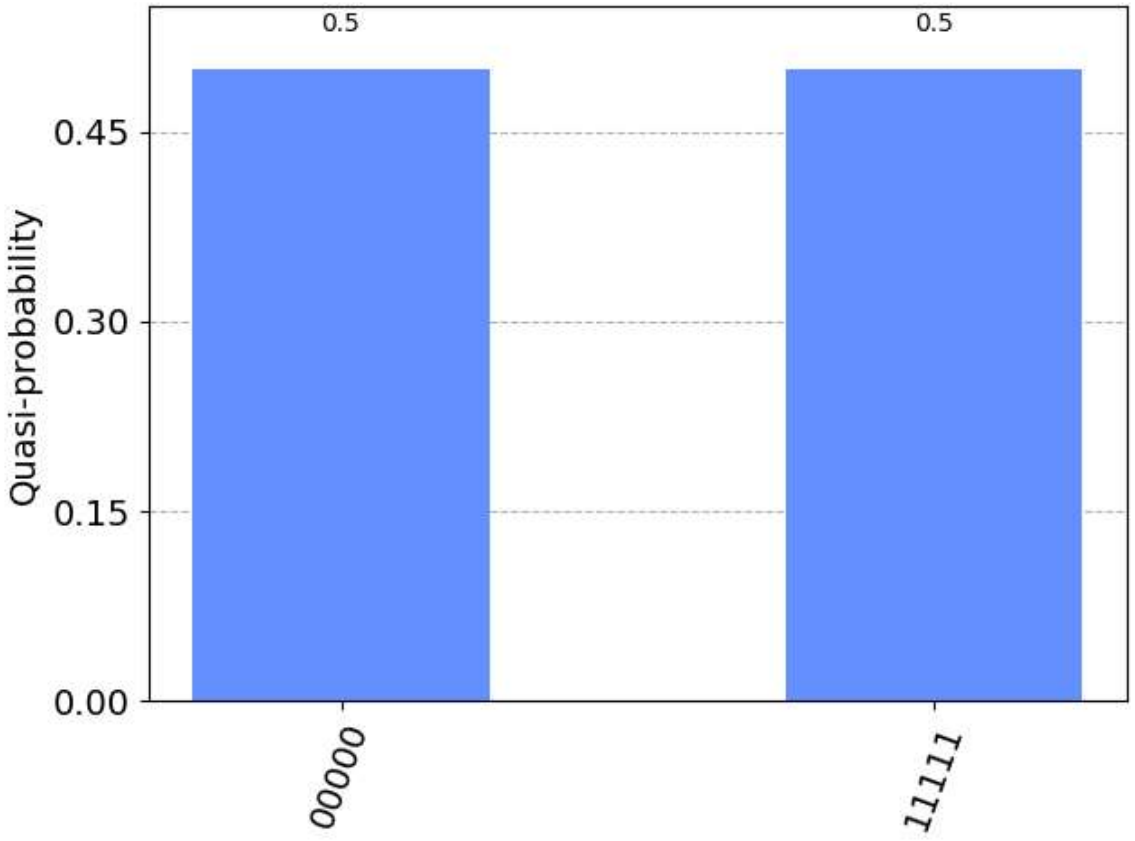

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

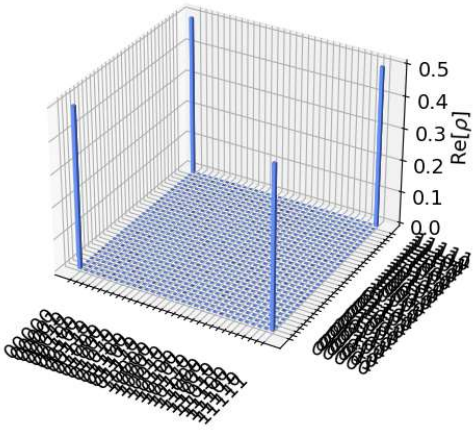

Re[ρ]
Im[ρ]

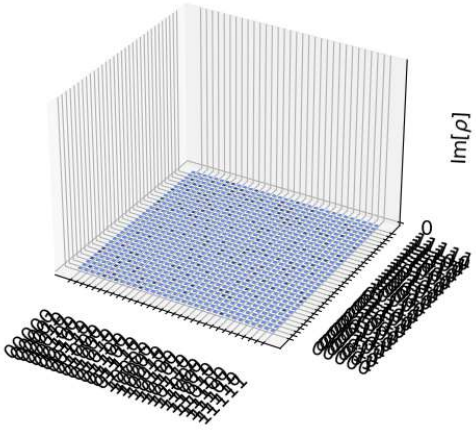

statevector
H
q_0
q_1
q_2
q_3
q_4

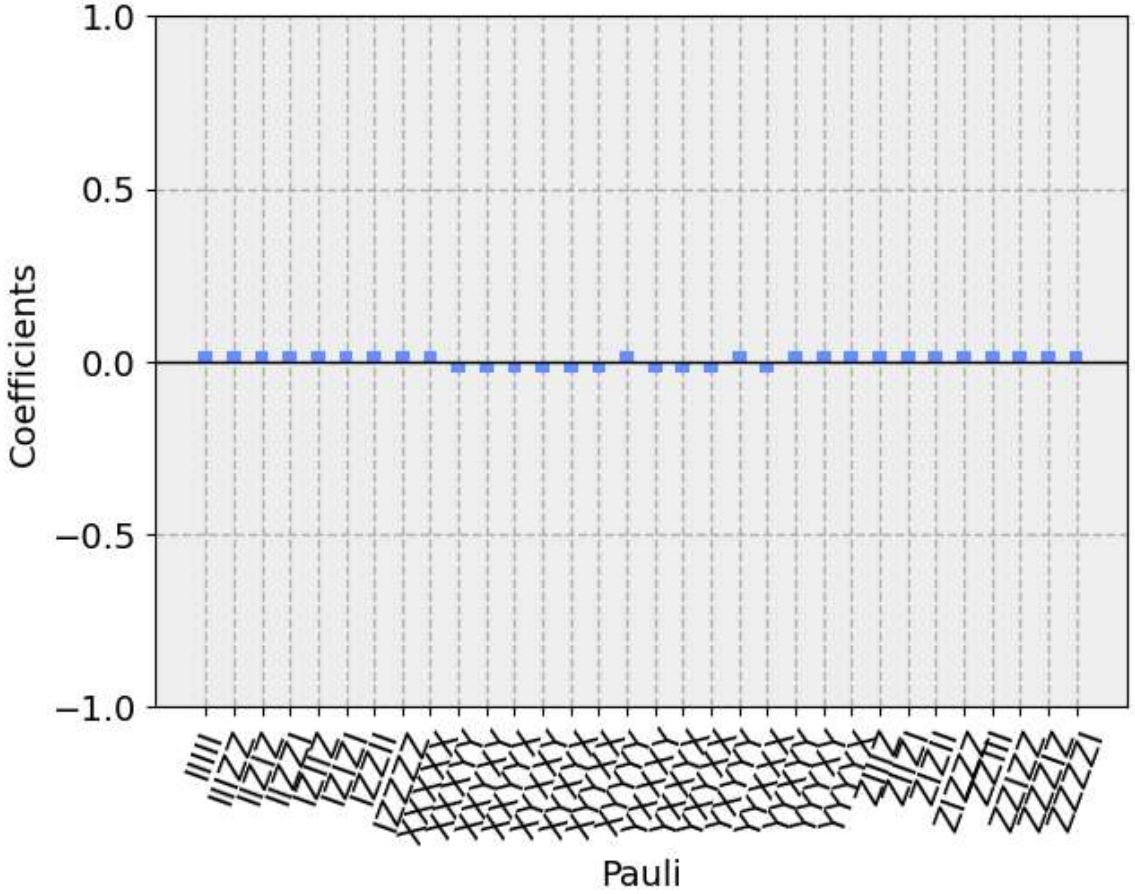

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

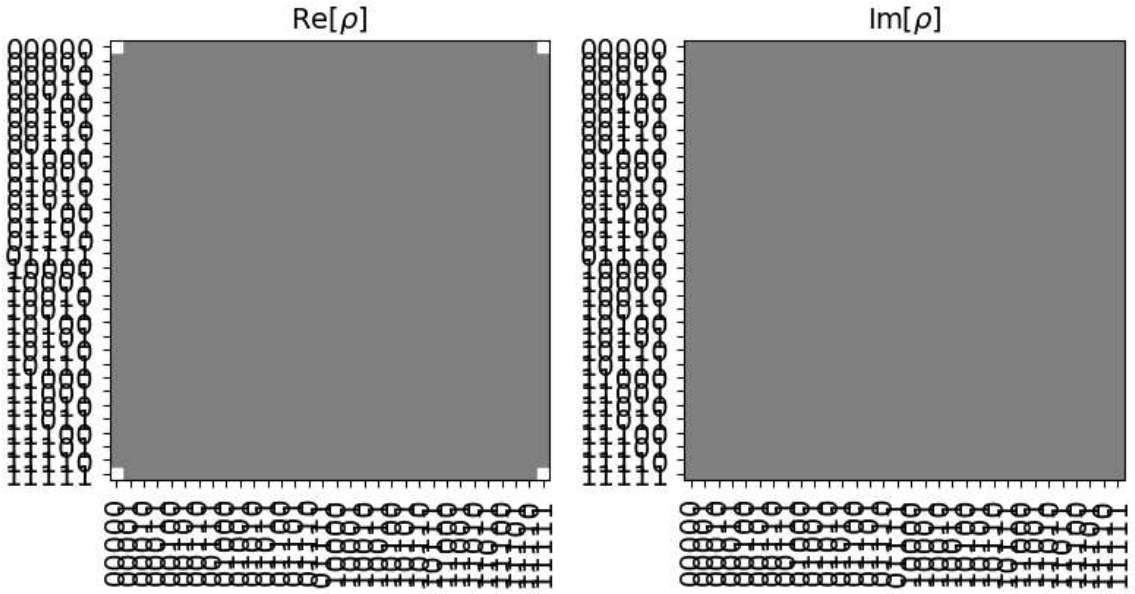

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

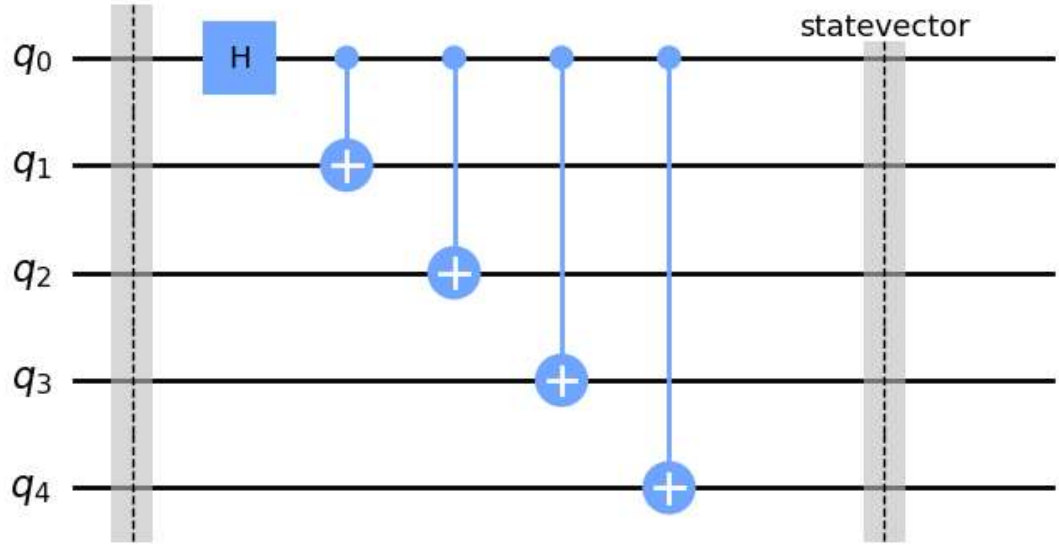

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

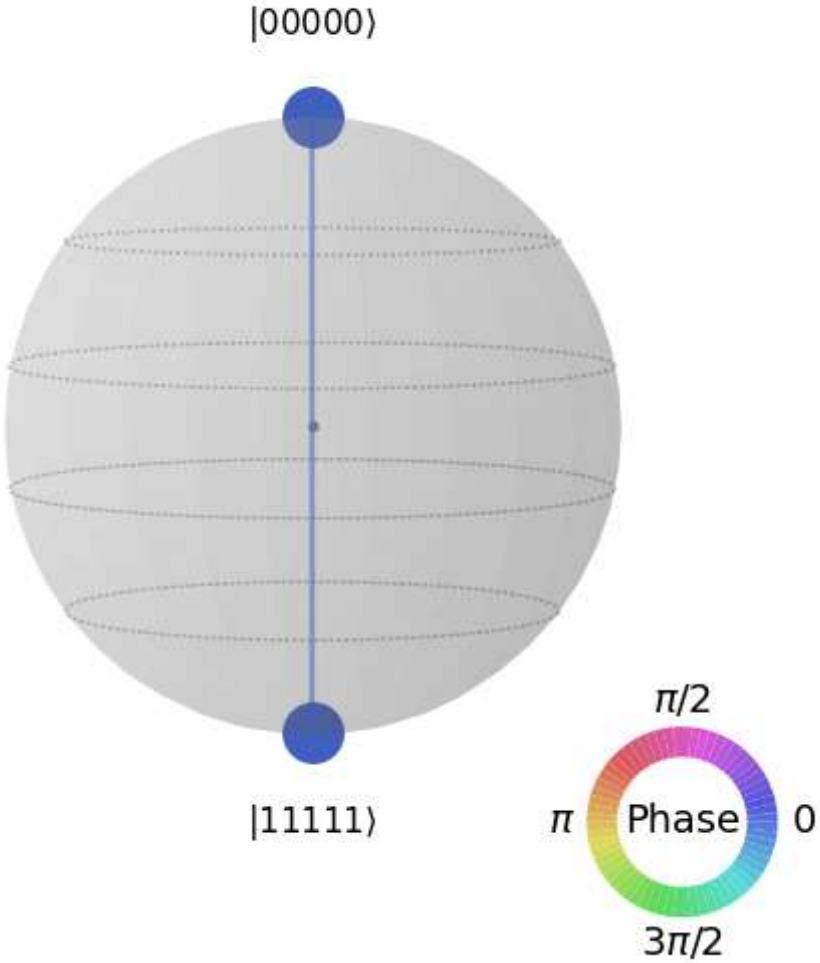

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

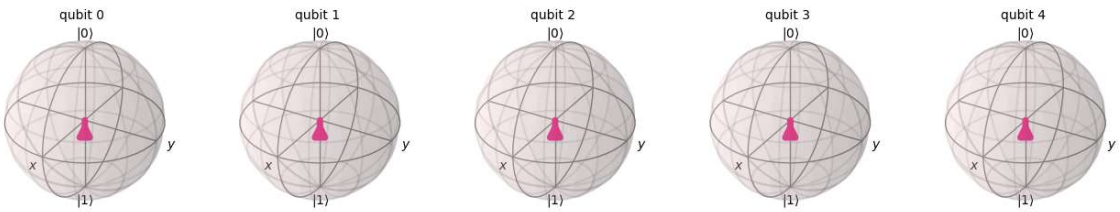

Im[ρ]
0

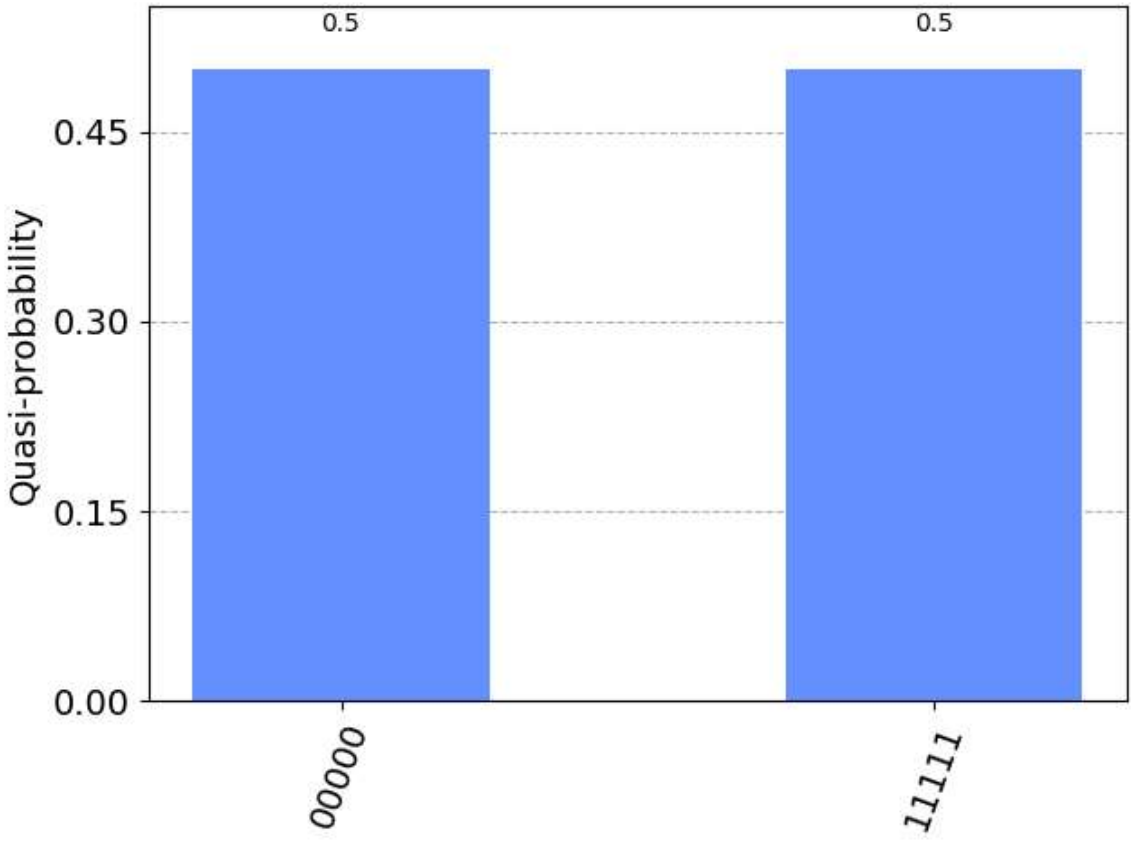

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

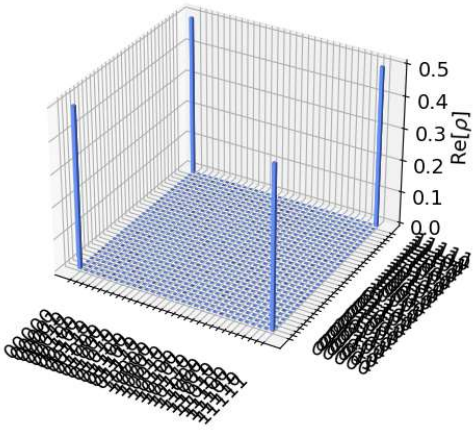

Re[ρ]
Im[ρ]

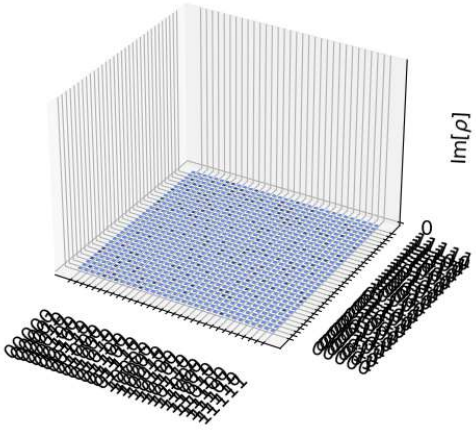

statevector
H
q_0
q_1
q_2
q_3
q_4

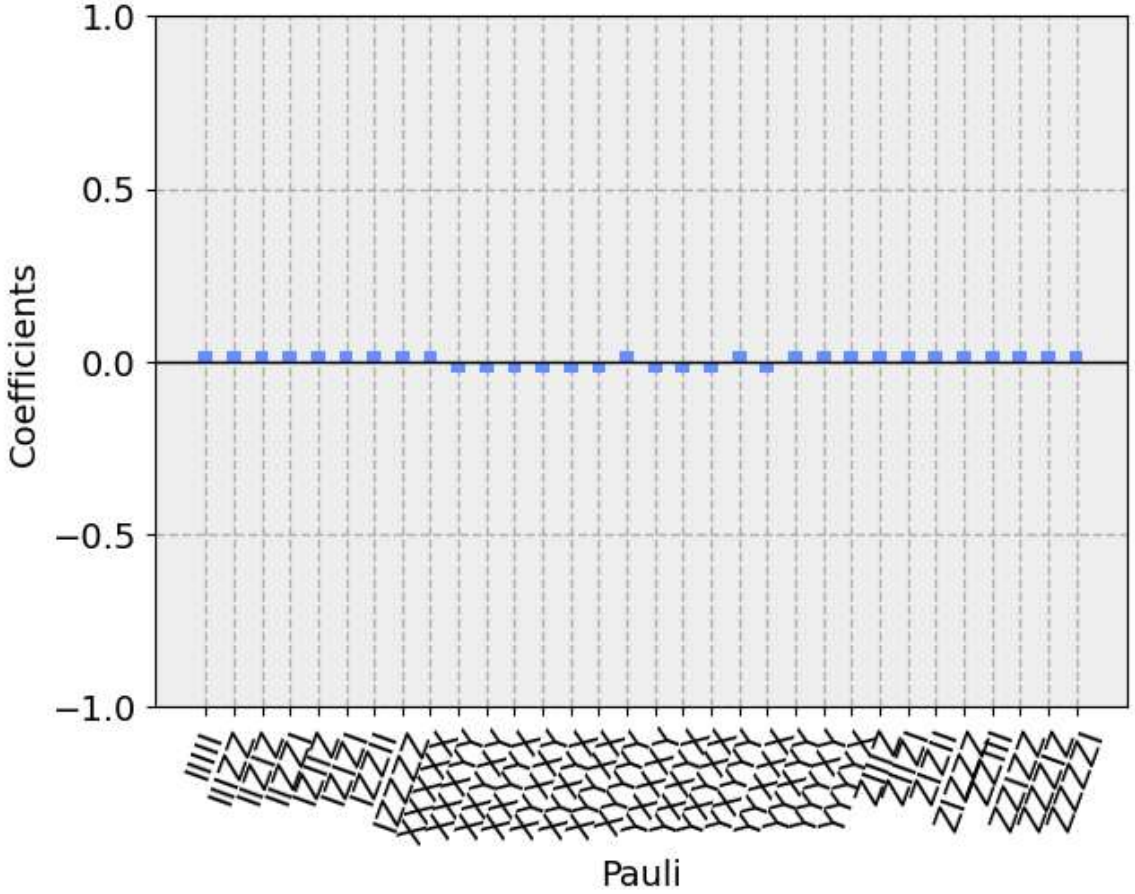

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

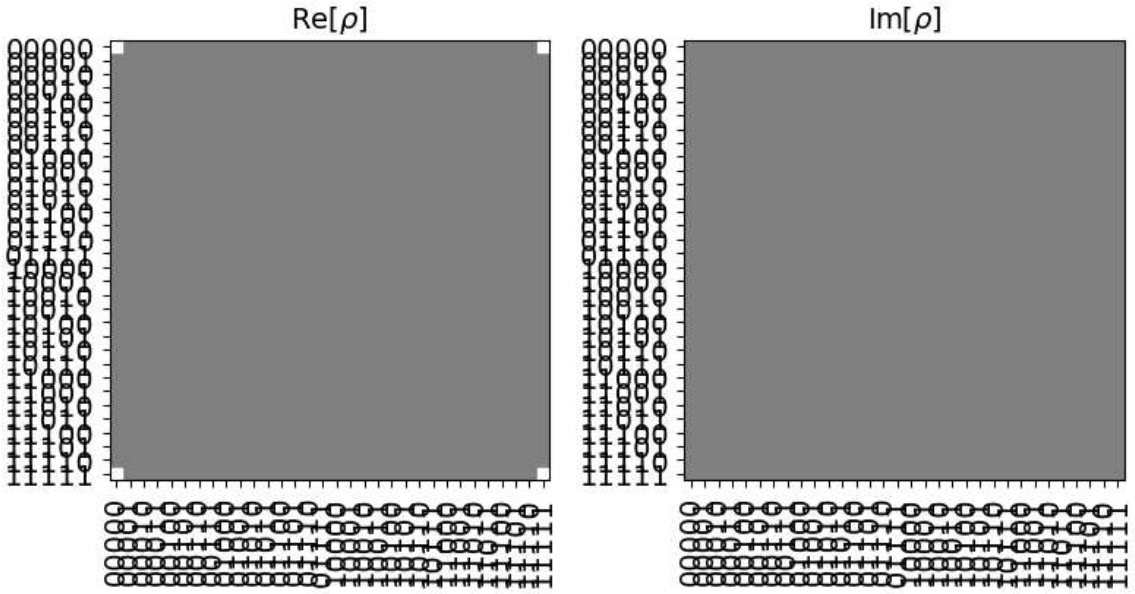

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

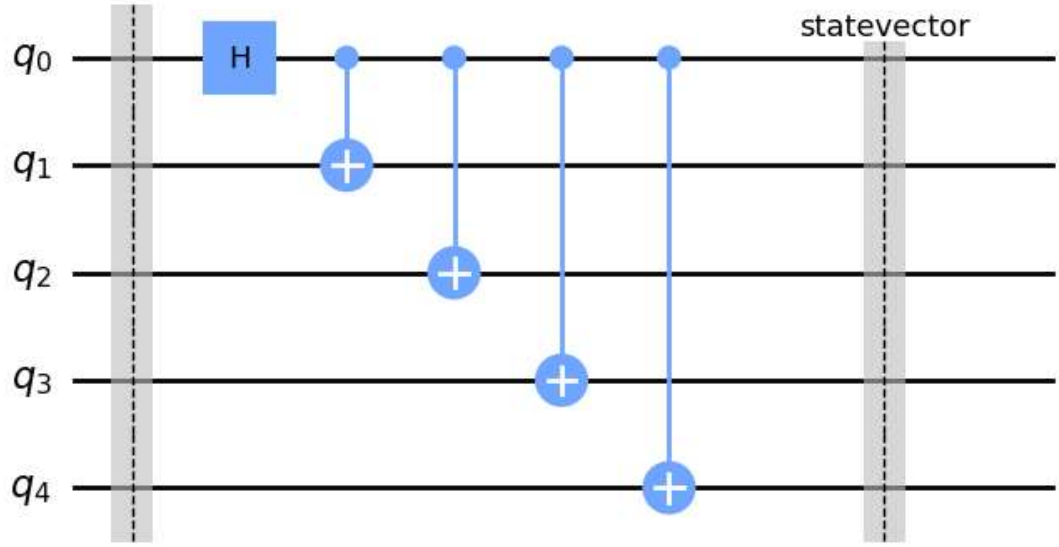

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

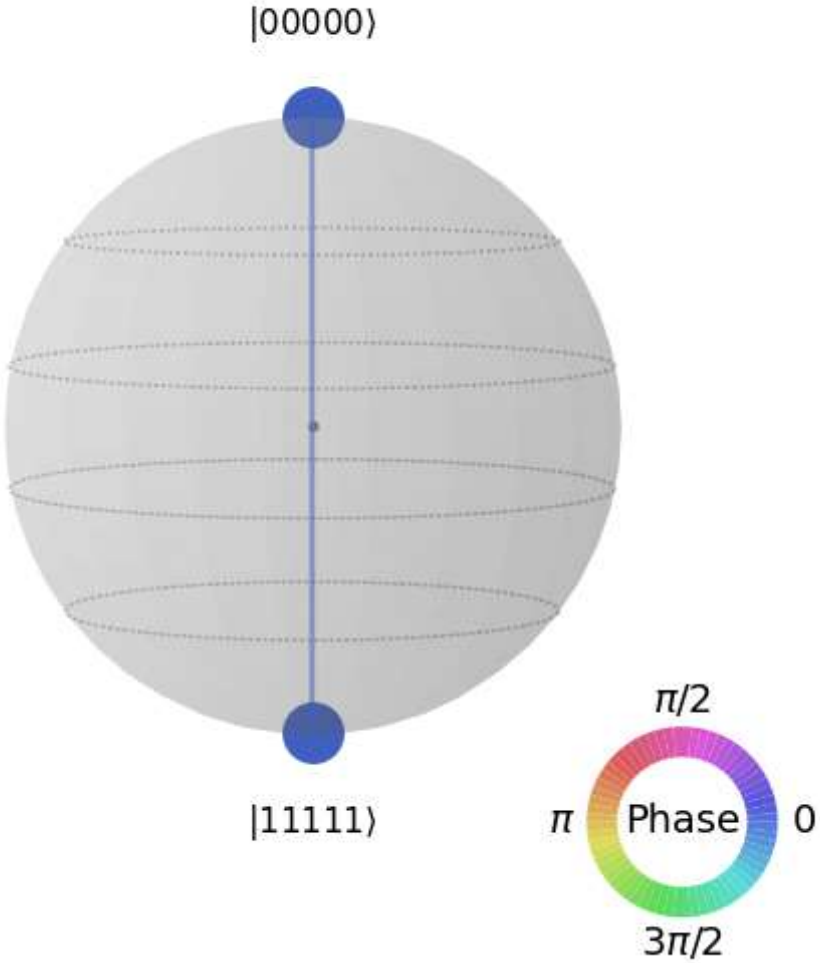

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

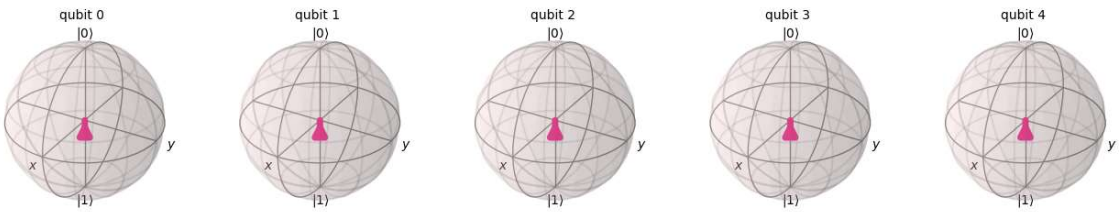

Im[ρ]
0

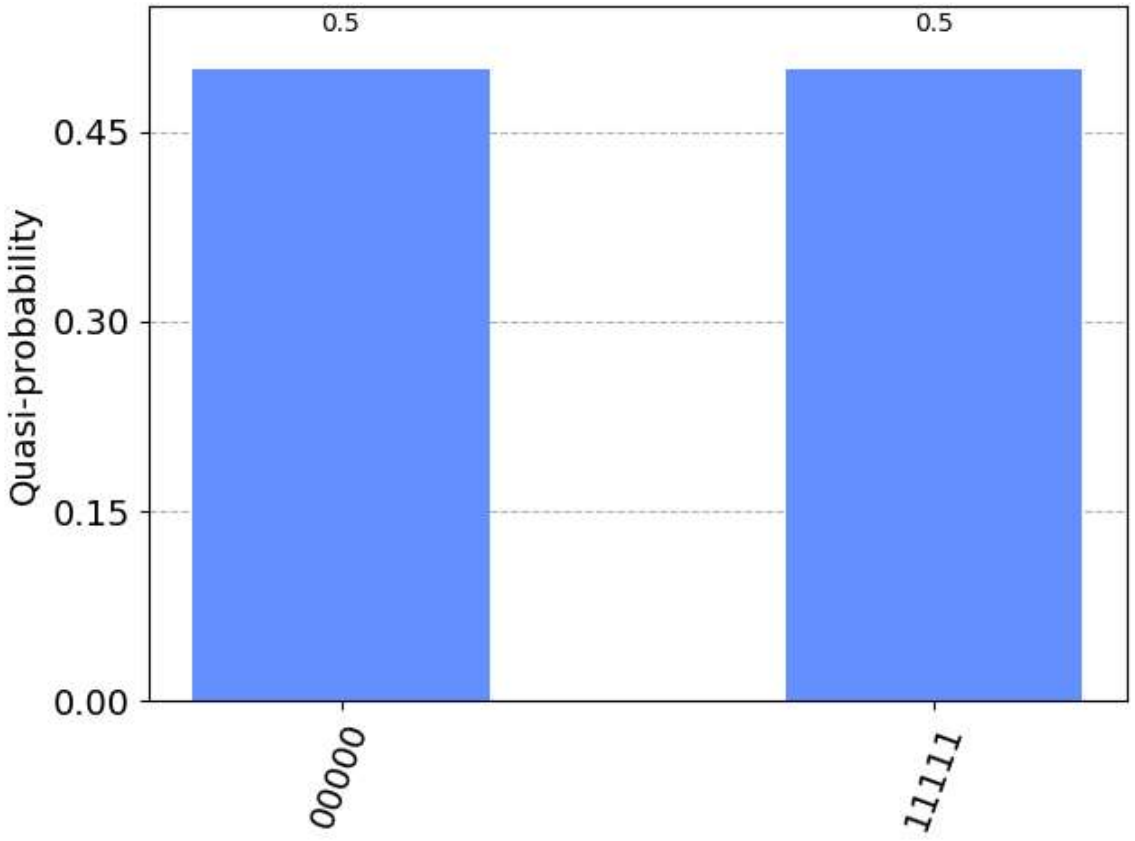

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

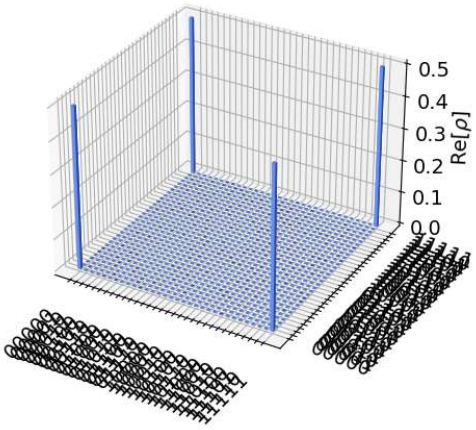

Re[ρ]
Im[ρ]

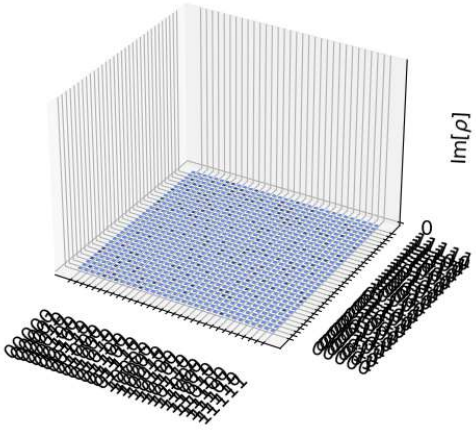

statevector
H
q_0
q_1
q_2
q_3
q_4

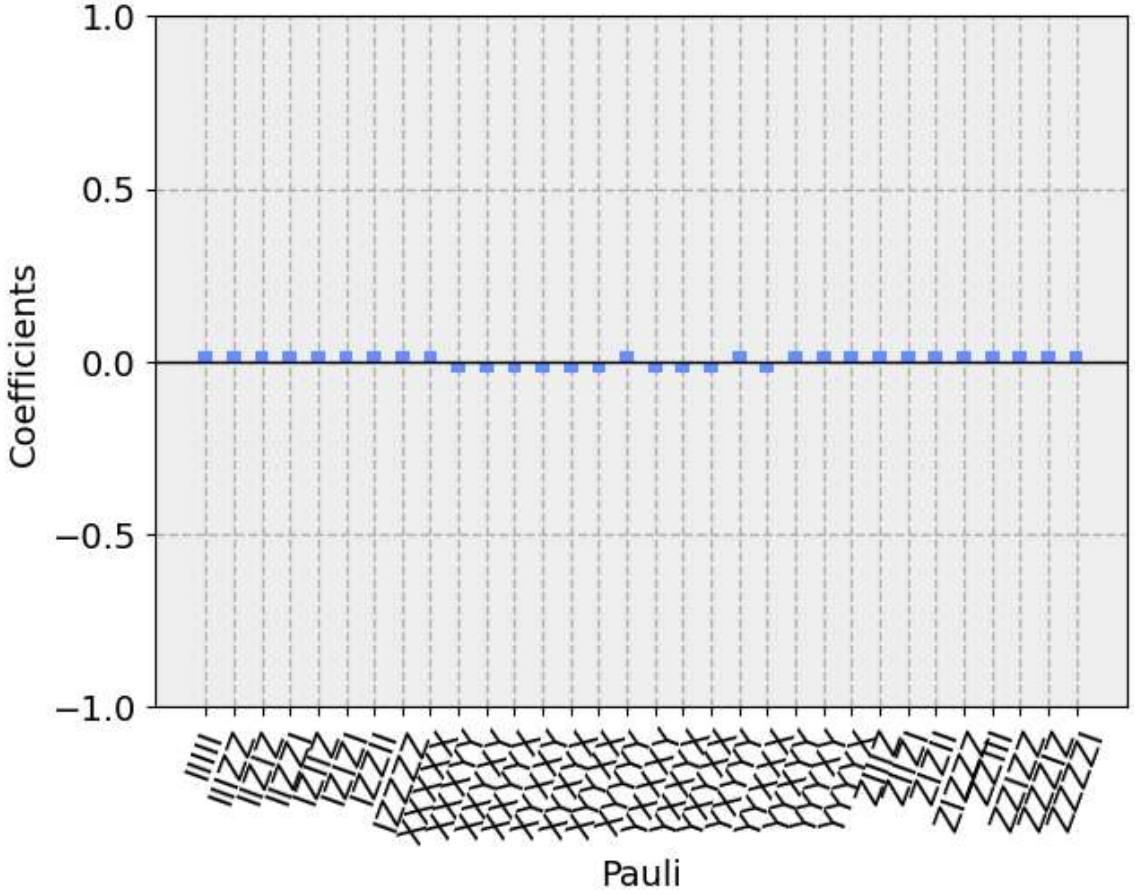

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

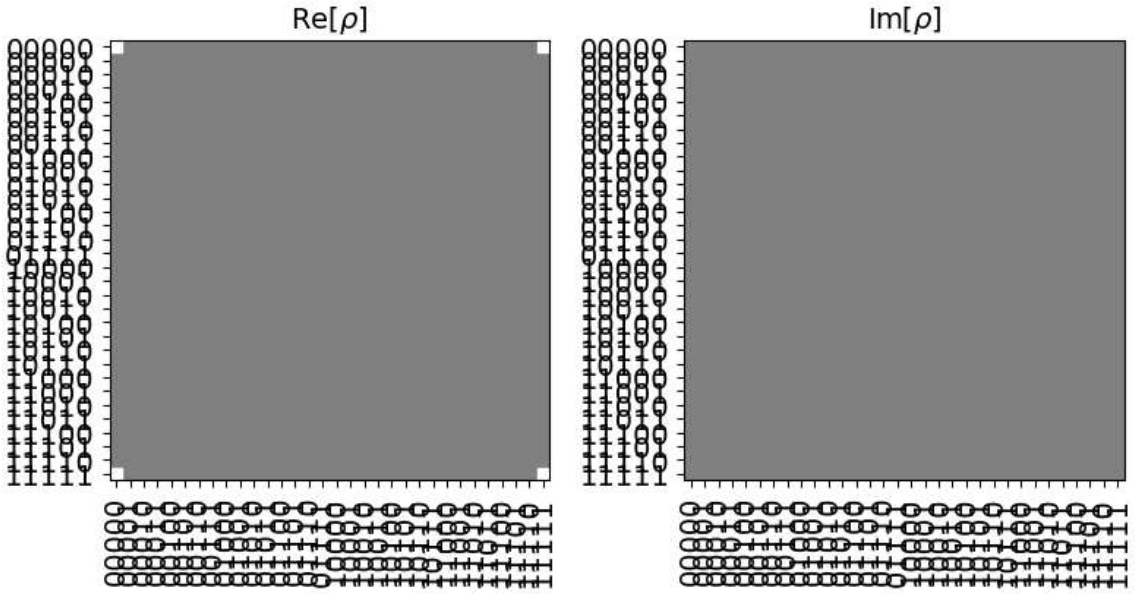

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

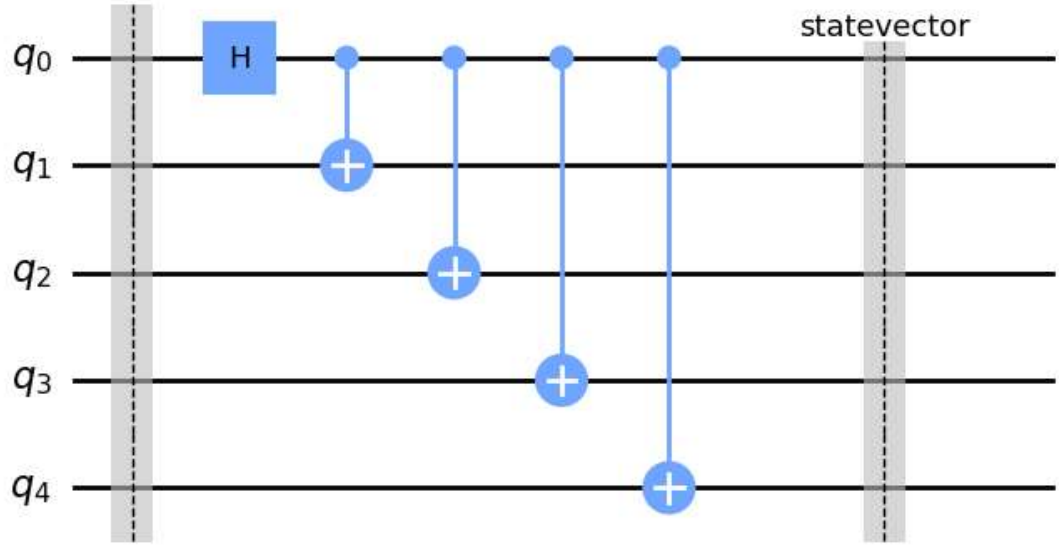

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

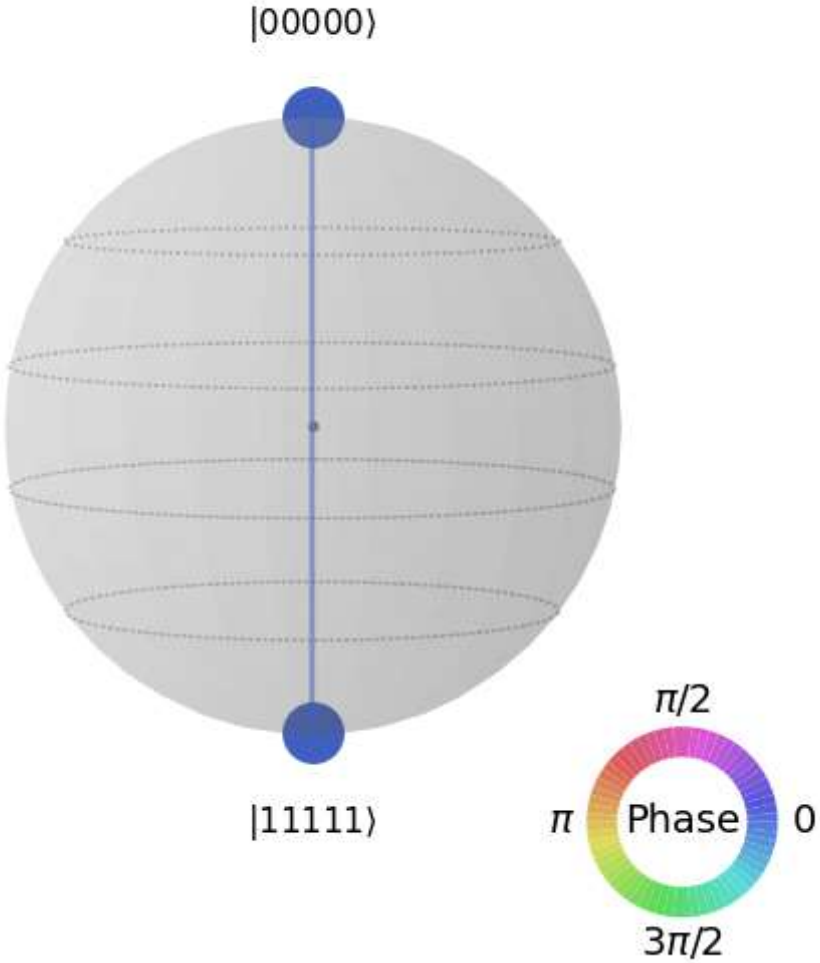

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

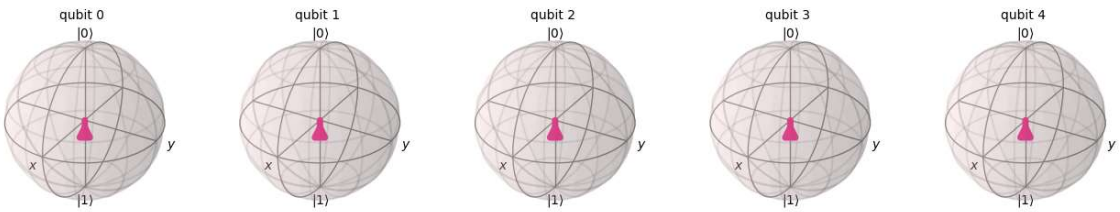

Im[ρ]
0

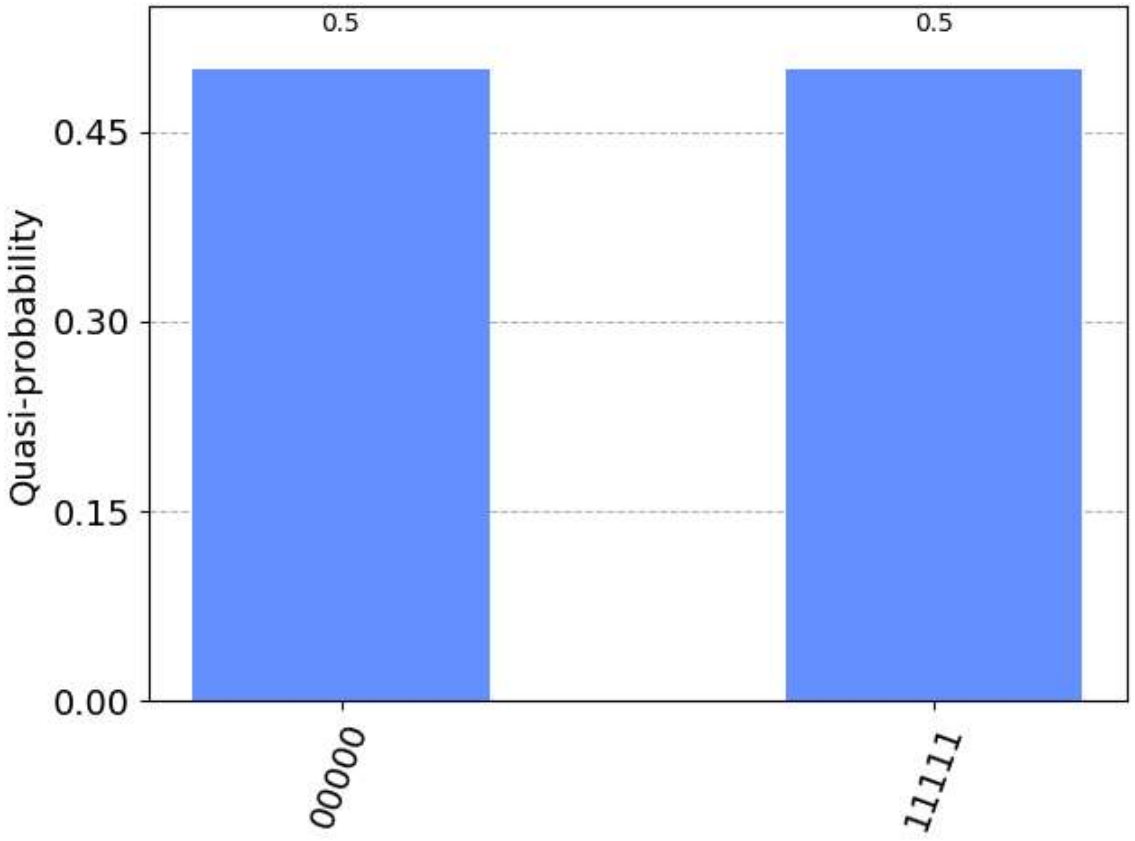

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

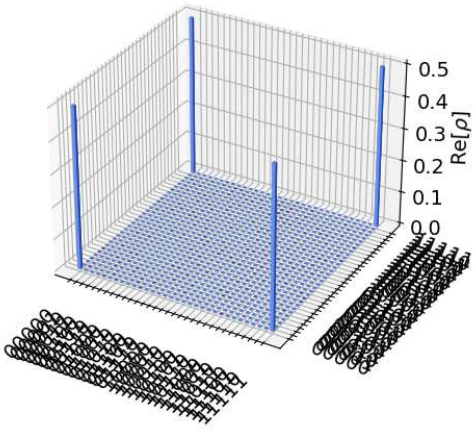

Re[ρ]
Im[ρ]

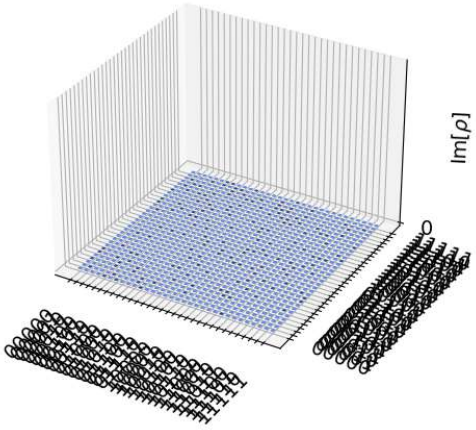

statevector
H
q_0
q_1
q_2
q_3
q_4

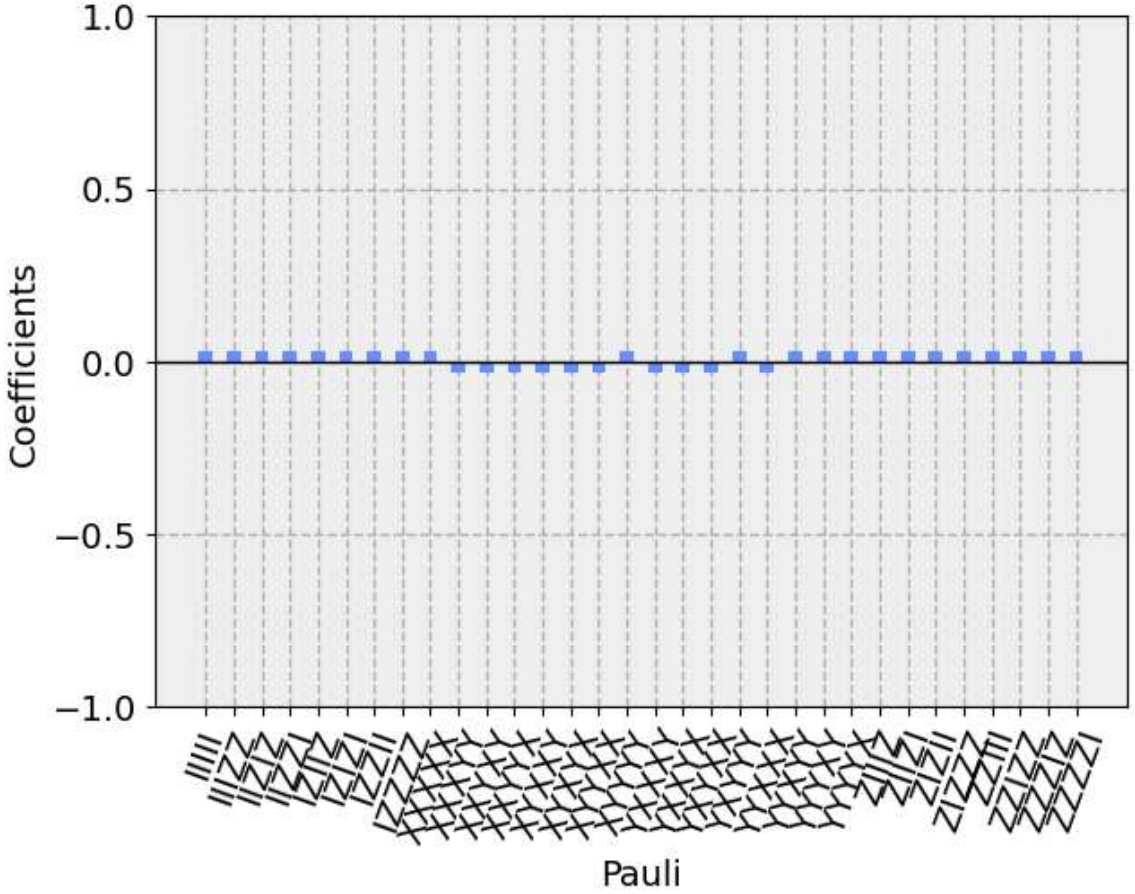

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

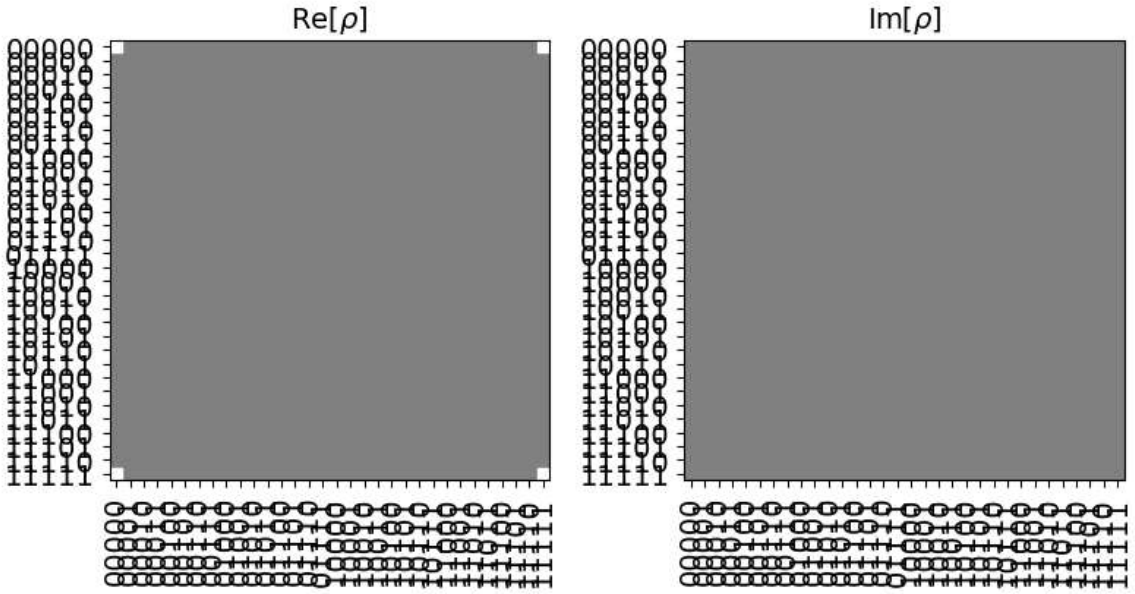

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

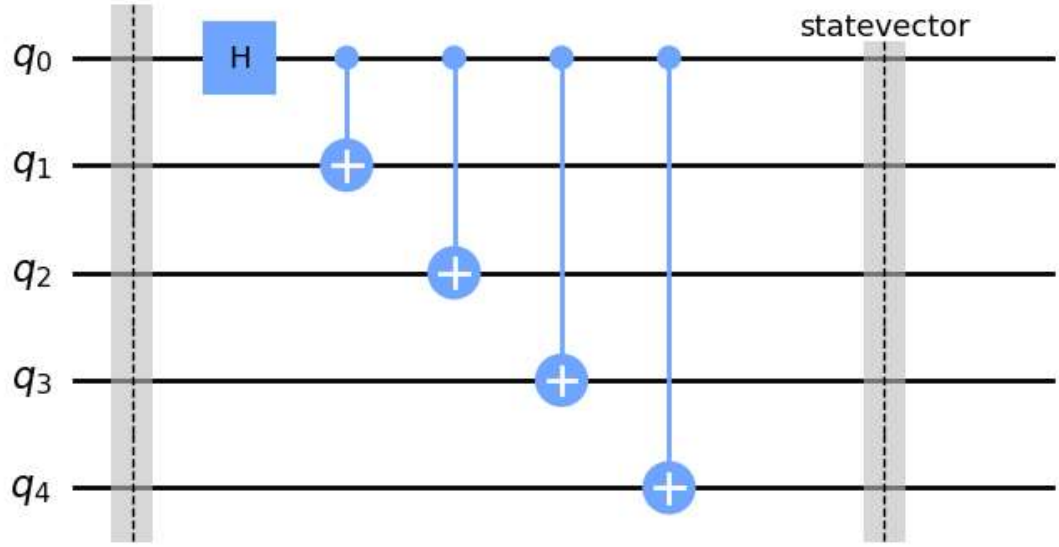

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

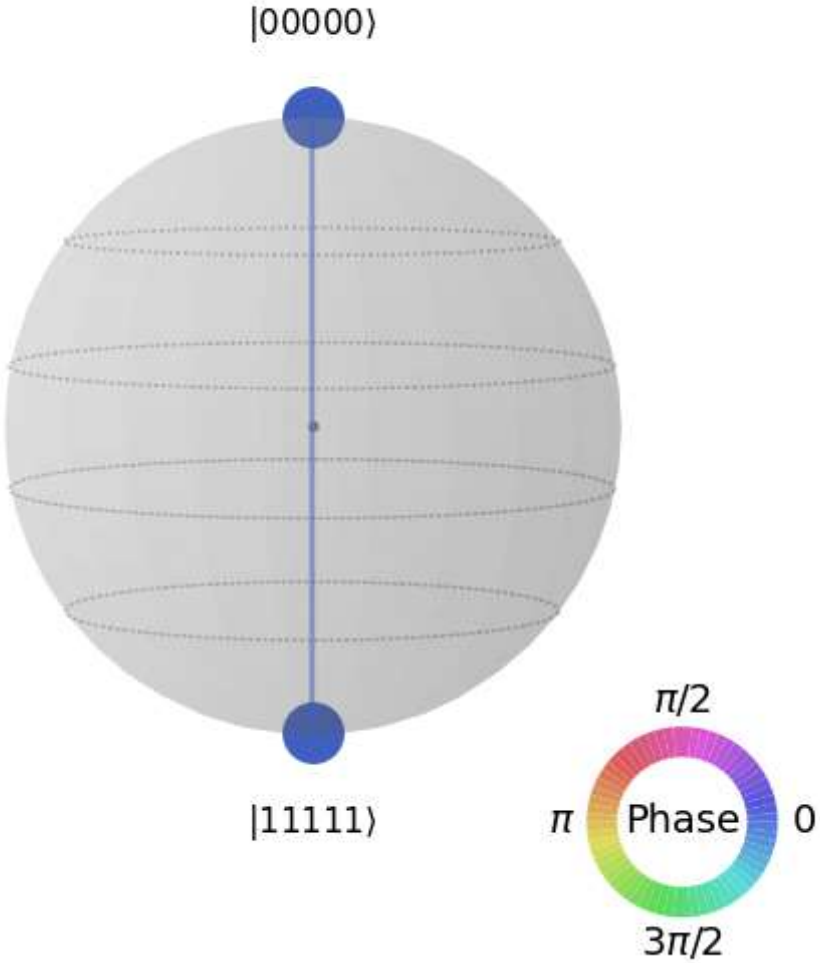

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

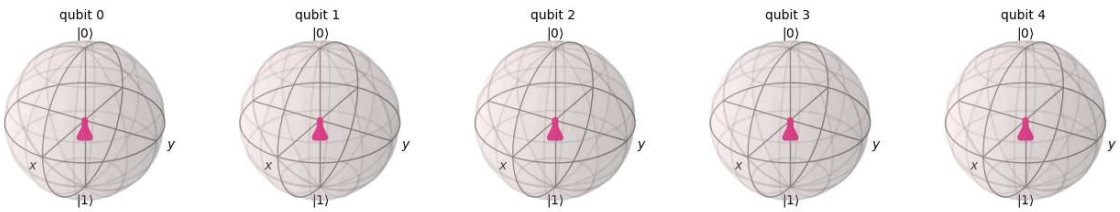

Im[ρ]
0

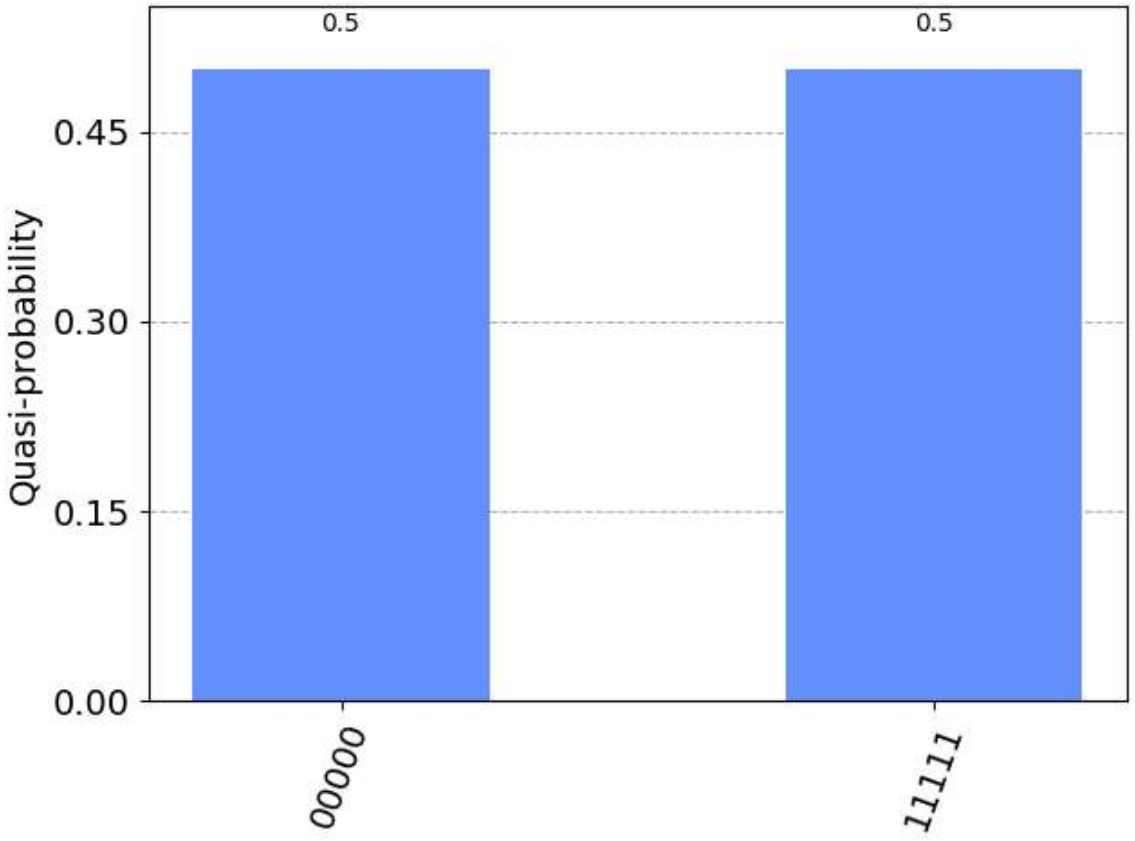

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

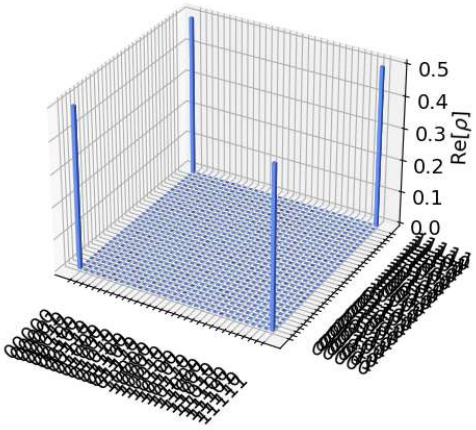

Re[ρ]
Im[ρ]

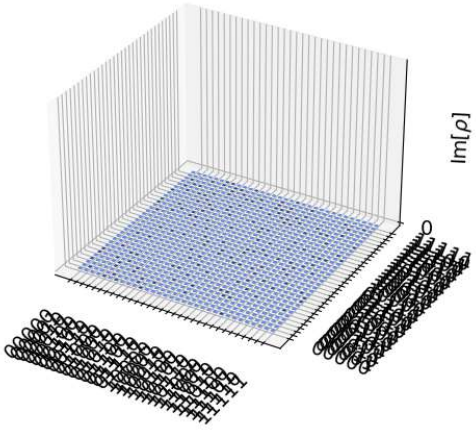

statevector
H
q_0
q_1
q_2
q_3
q_4

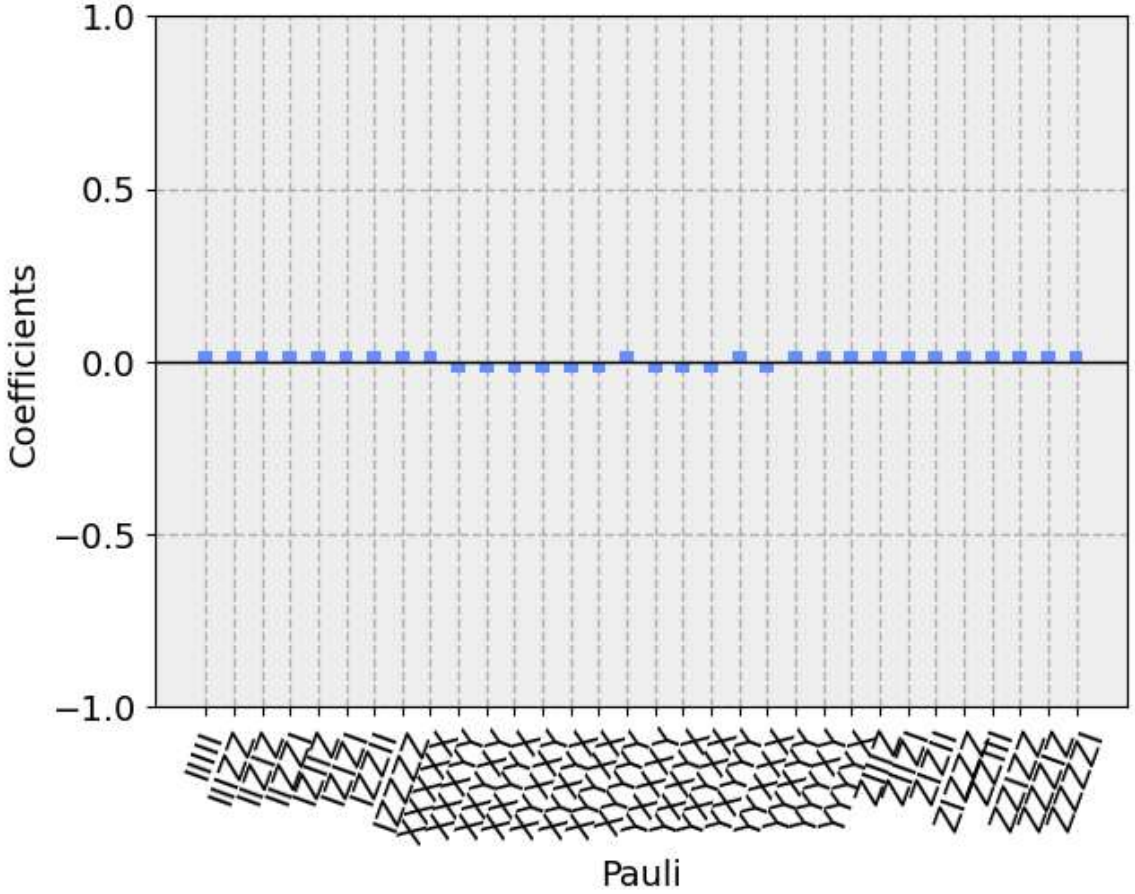

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

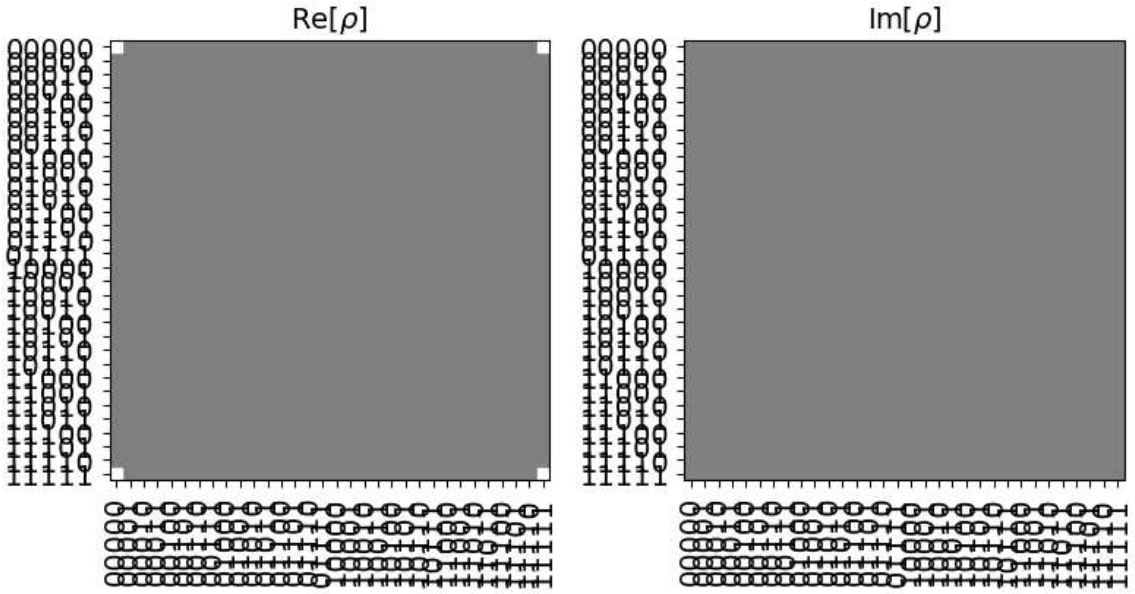

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

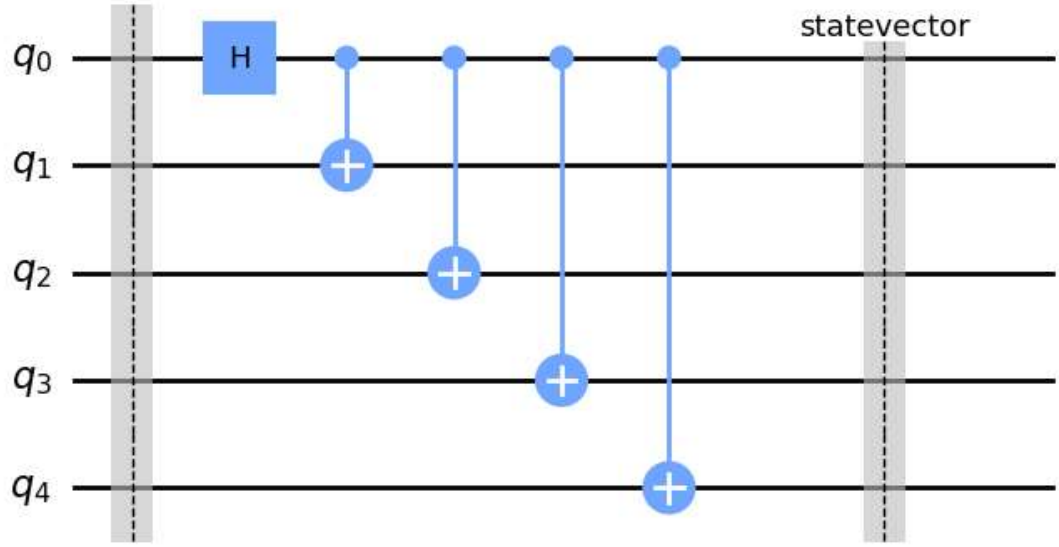

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

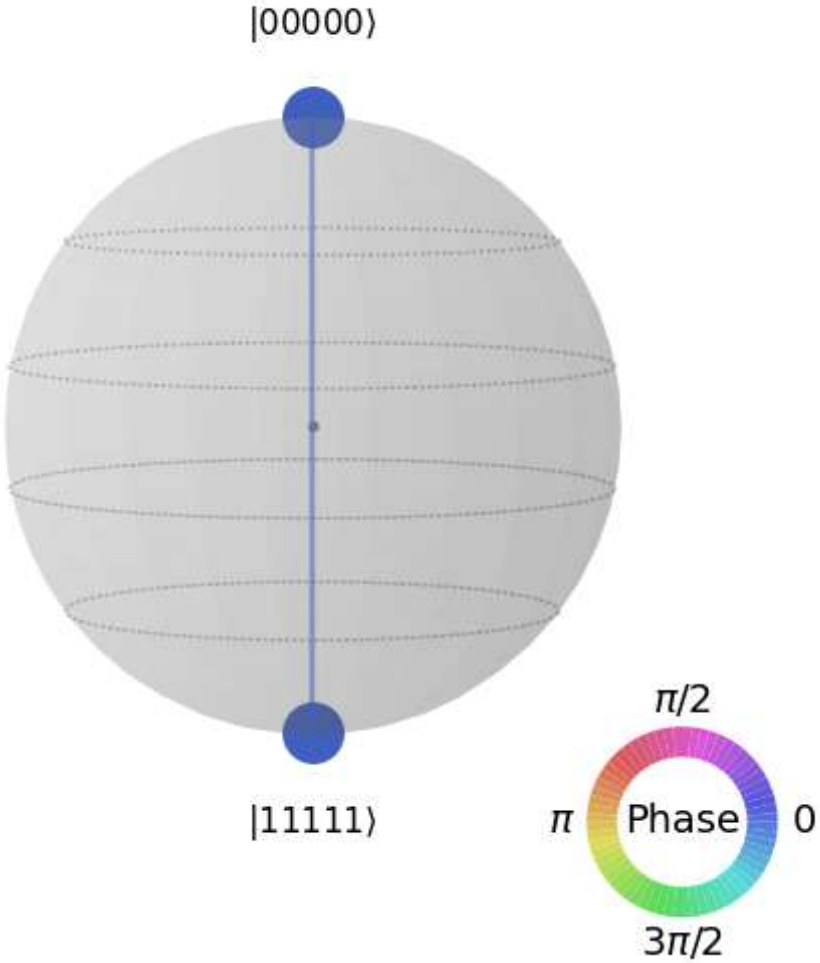

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

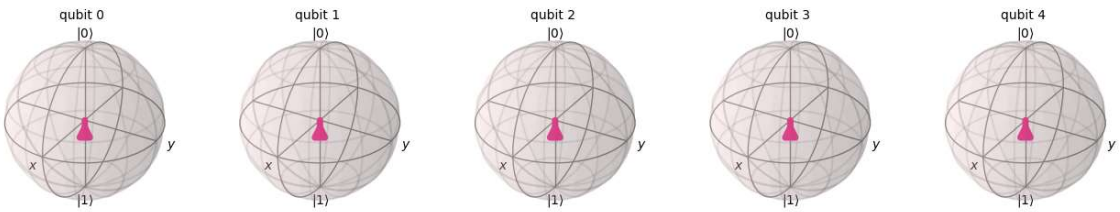

Im[ρ]
0

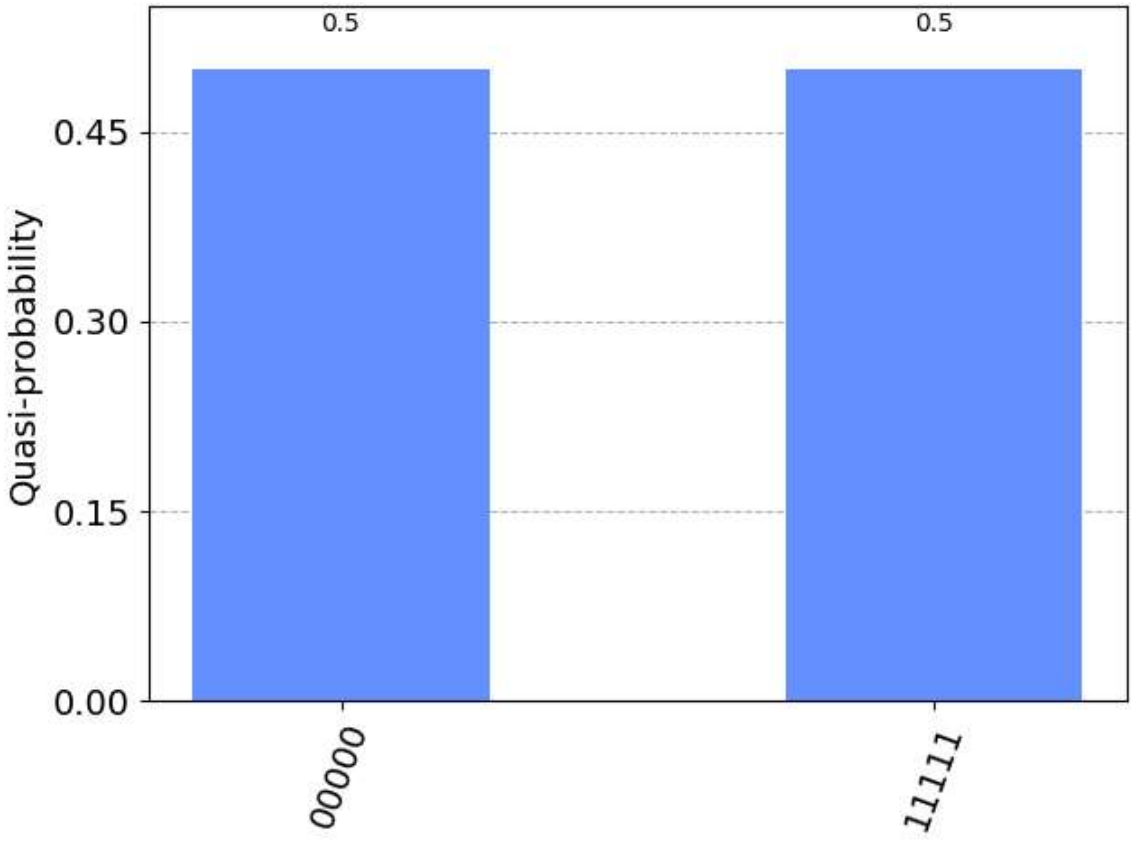

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

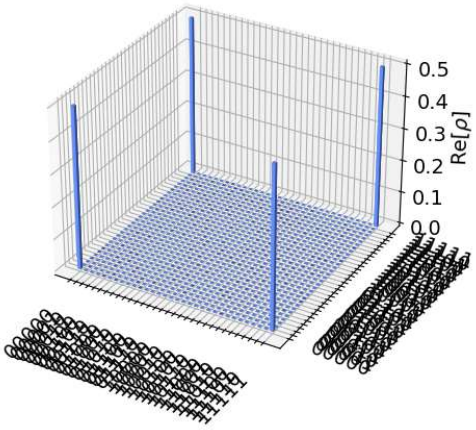

Re[ρ]
Im[ρ]

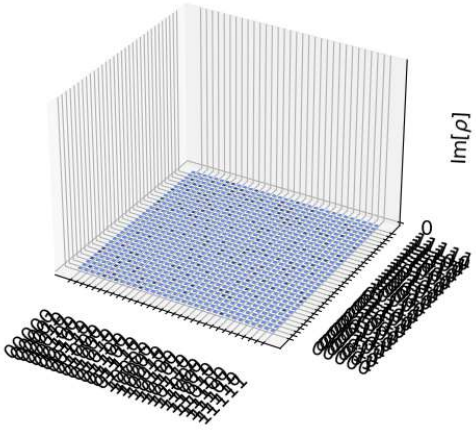

statevector
H
q_0
q_1
q_2
q_3
q_4

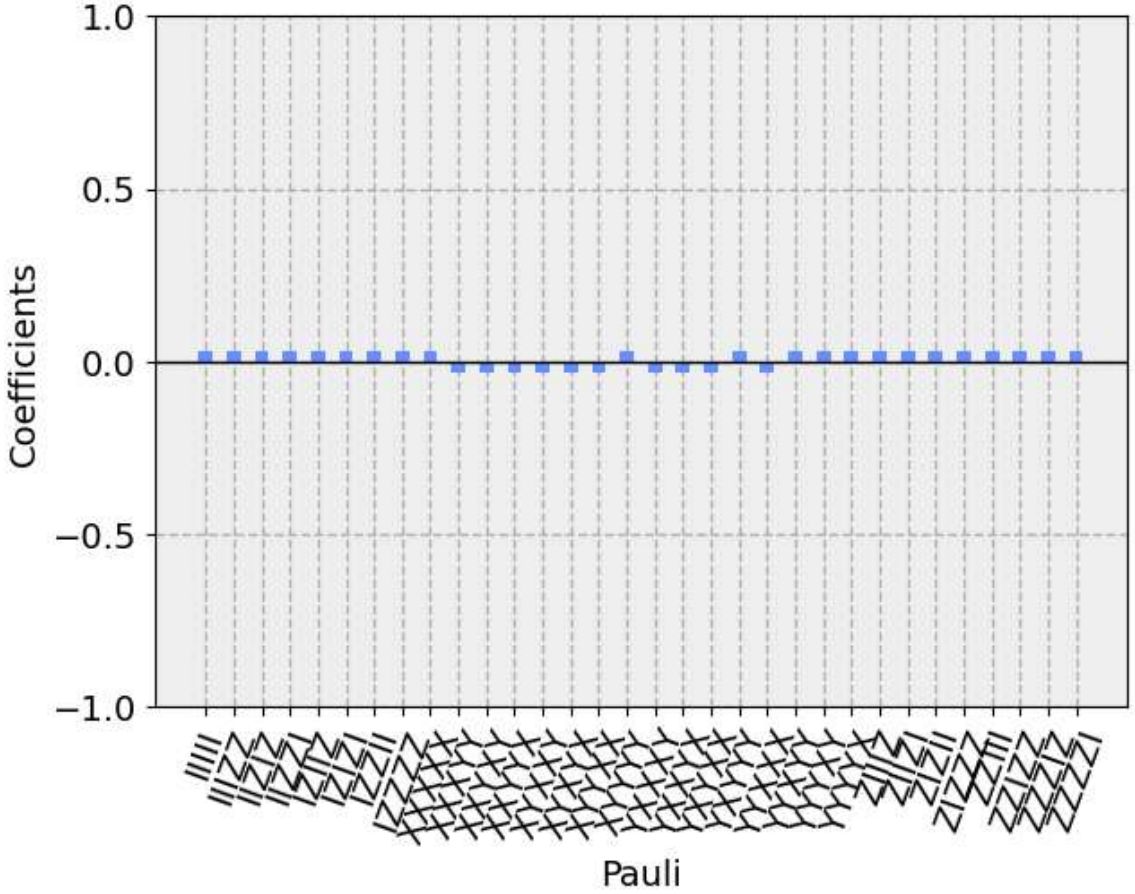

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

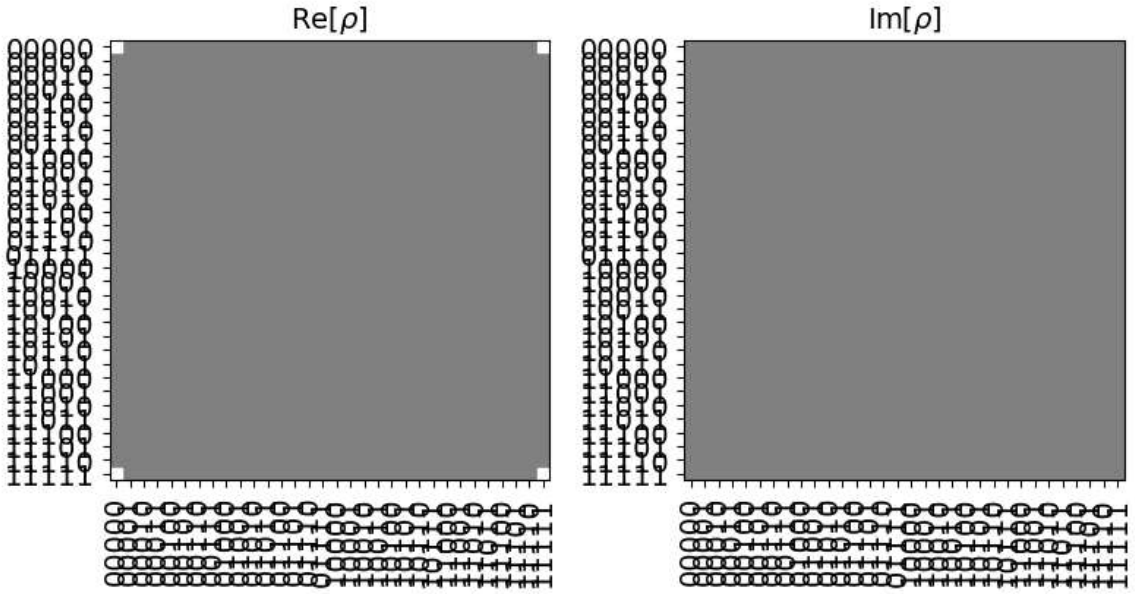

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

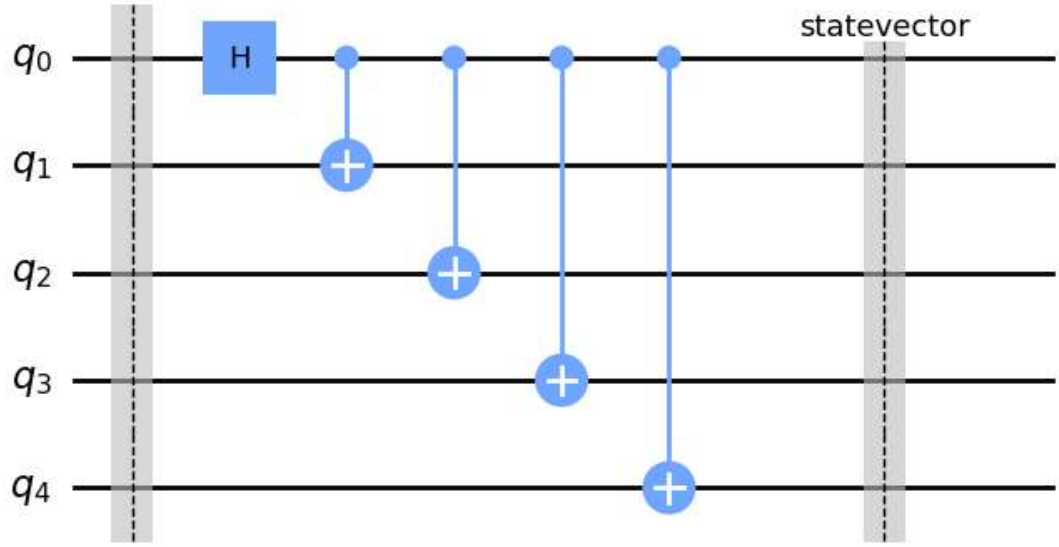

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

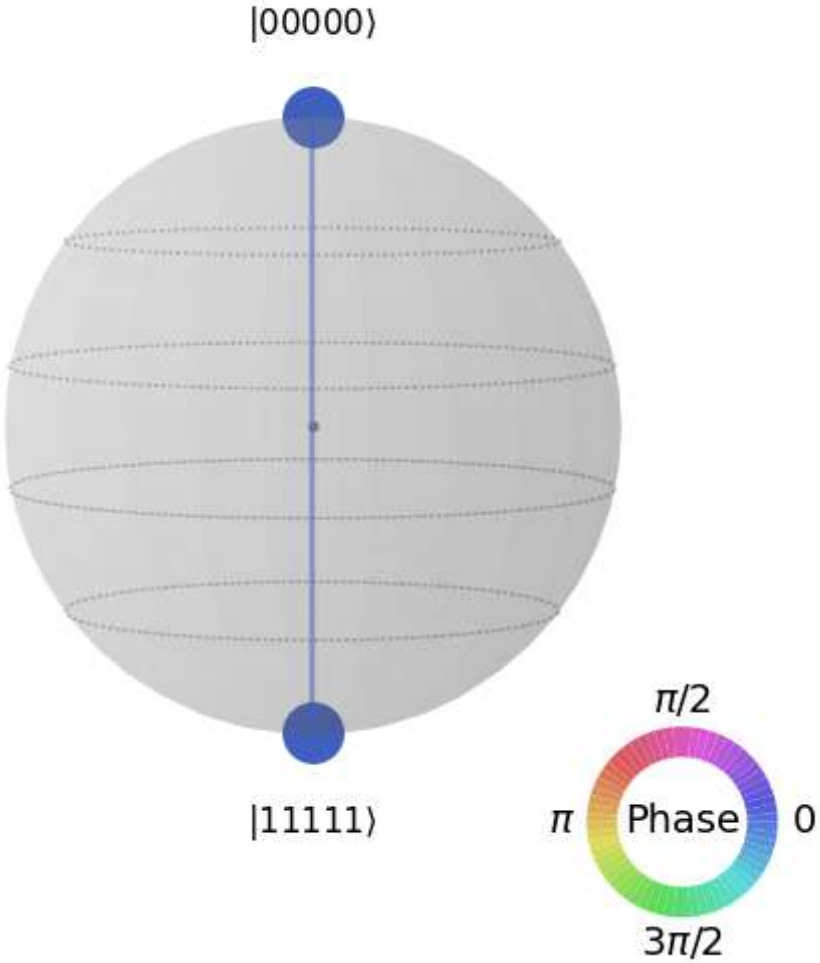

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

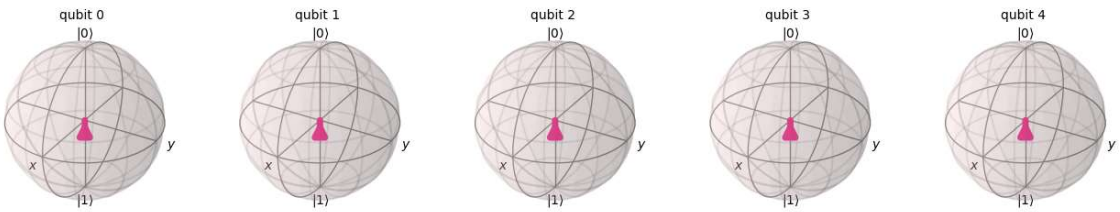

Im[ρ]
0

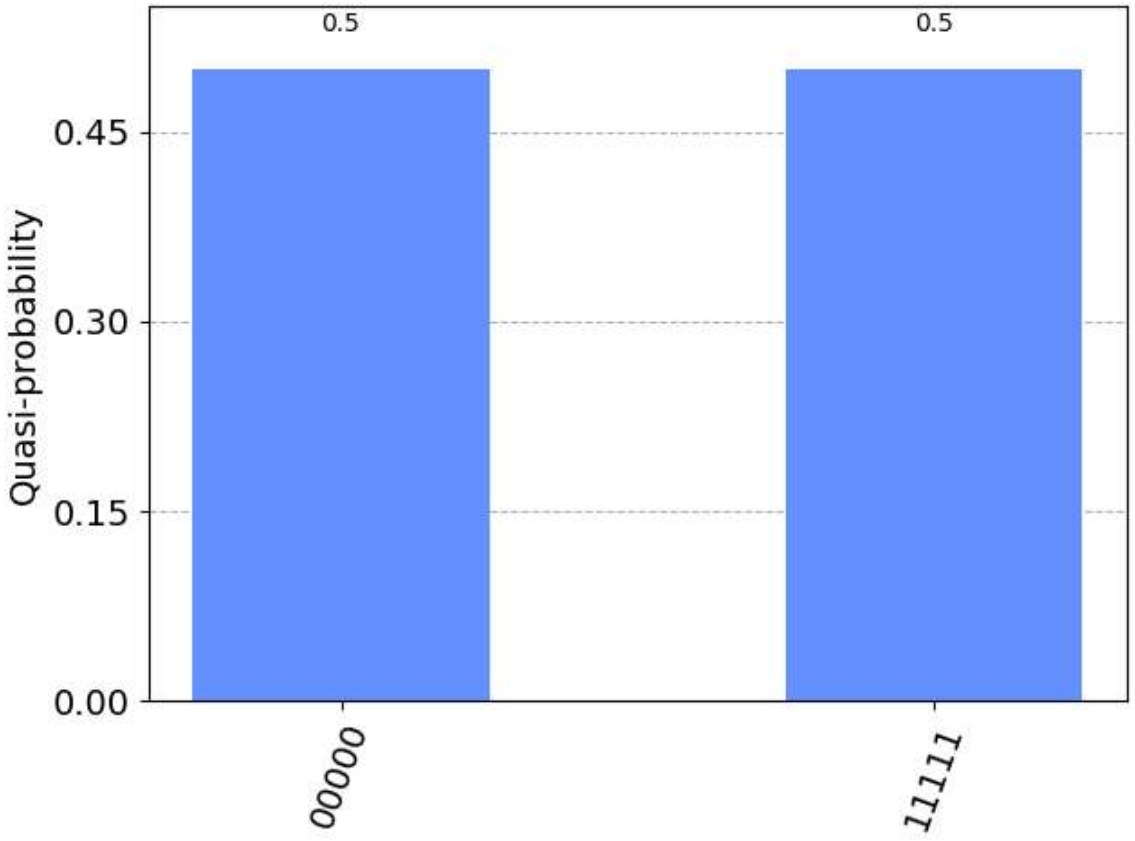

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

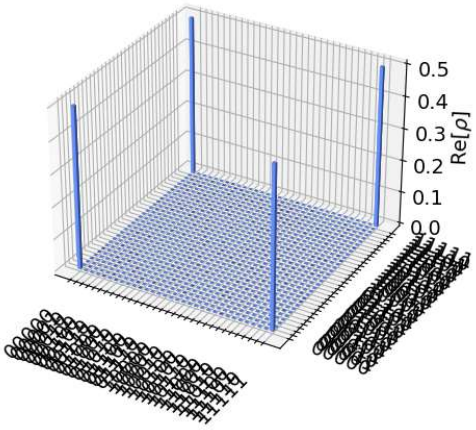

Re[ρ]
Im[ρ]

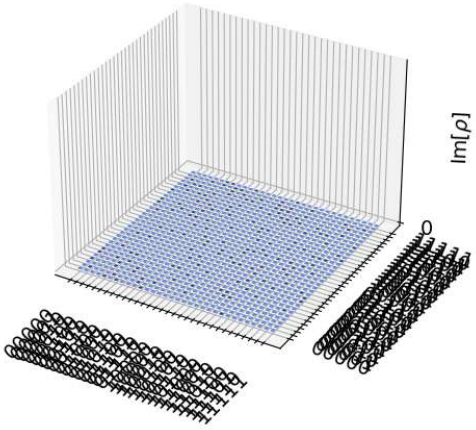

statevector
H
q_0
q_1
q_2
q_3
q_4

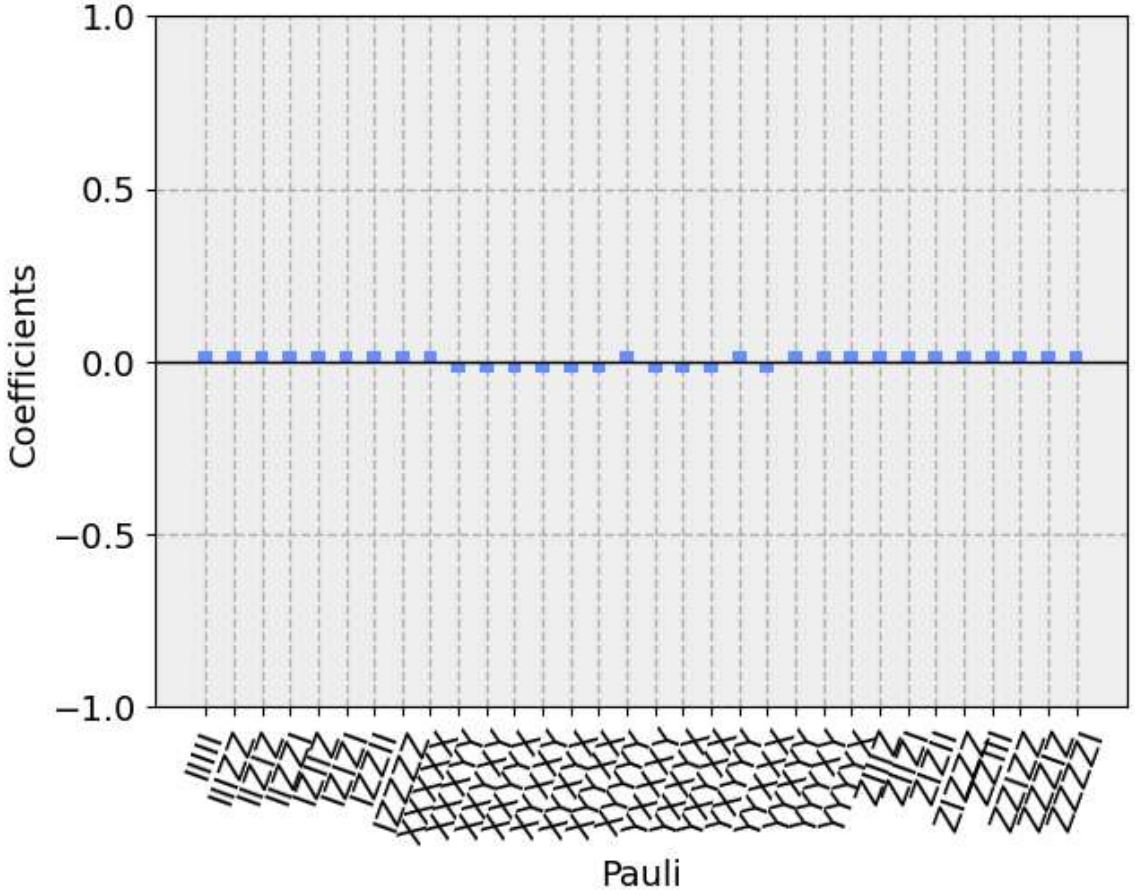

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

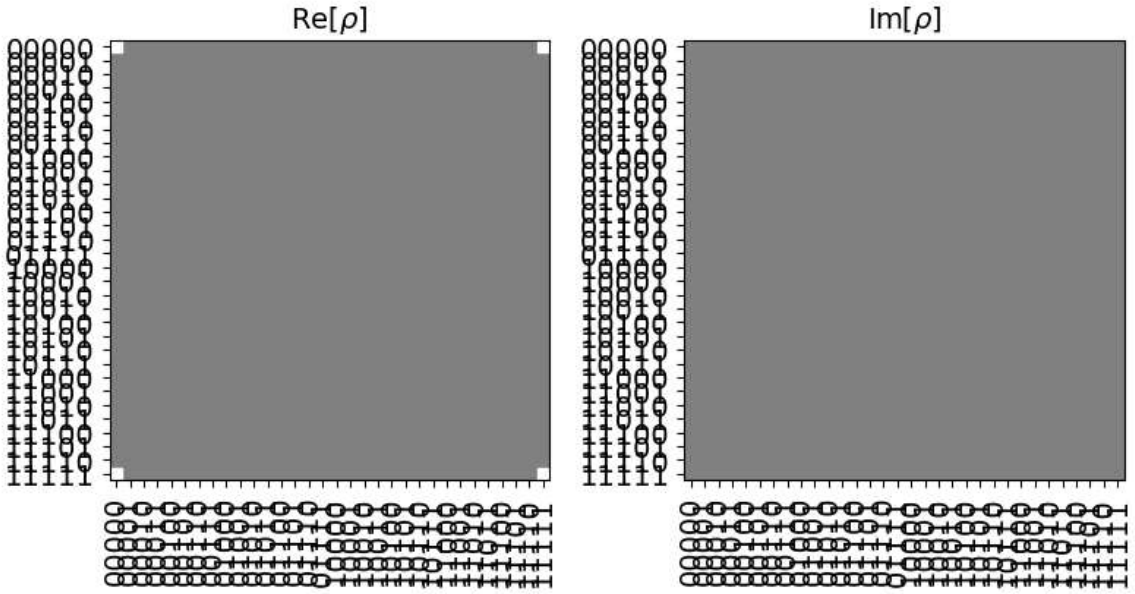

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

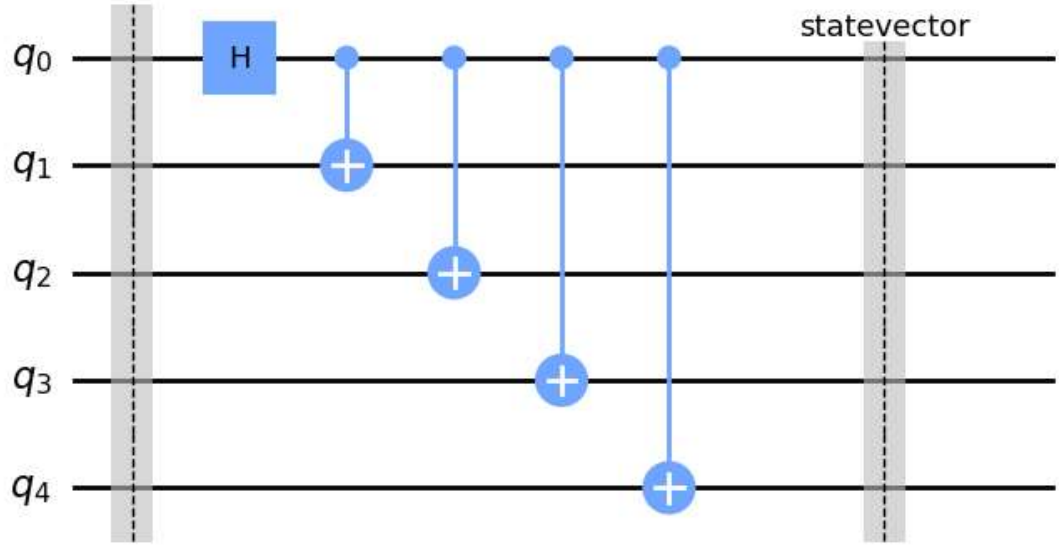

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

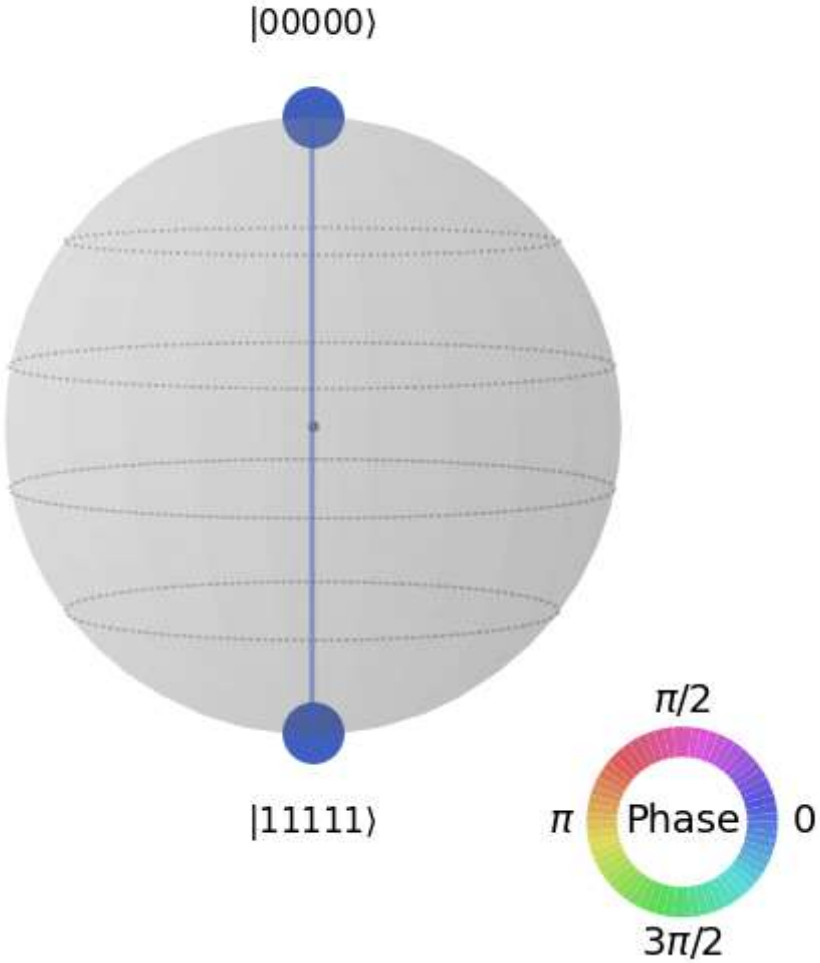

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

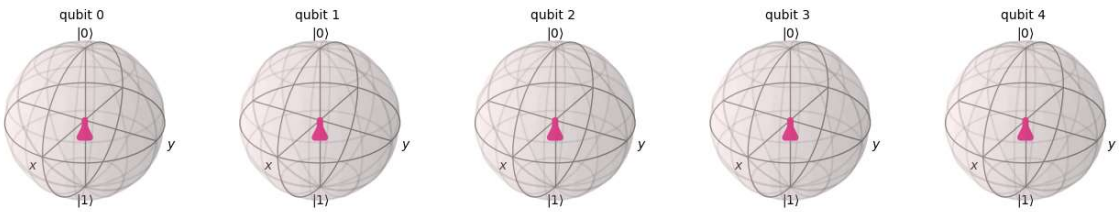

Im[ρ]
0

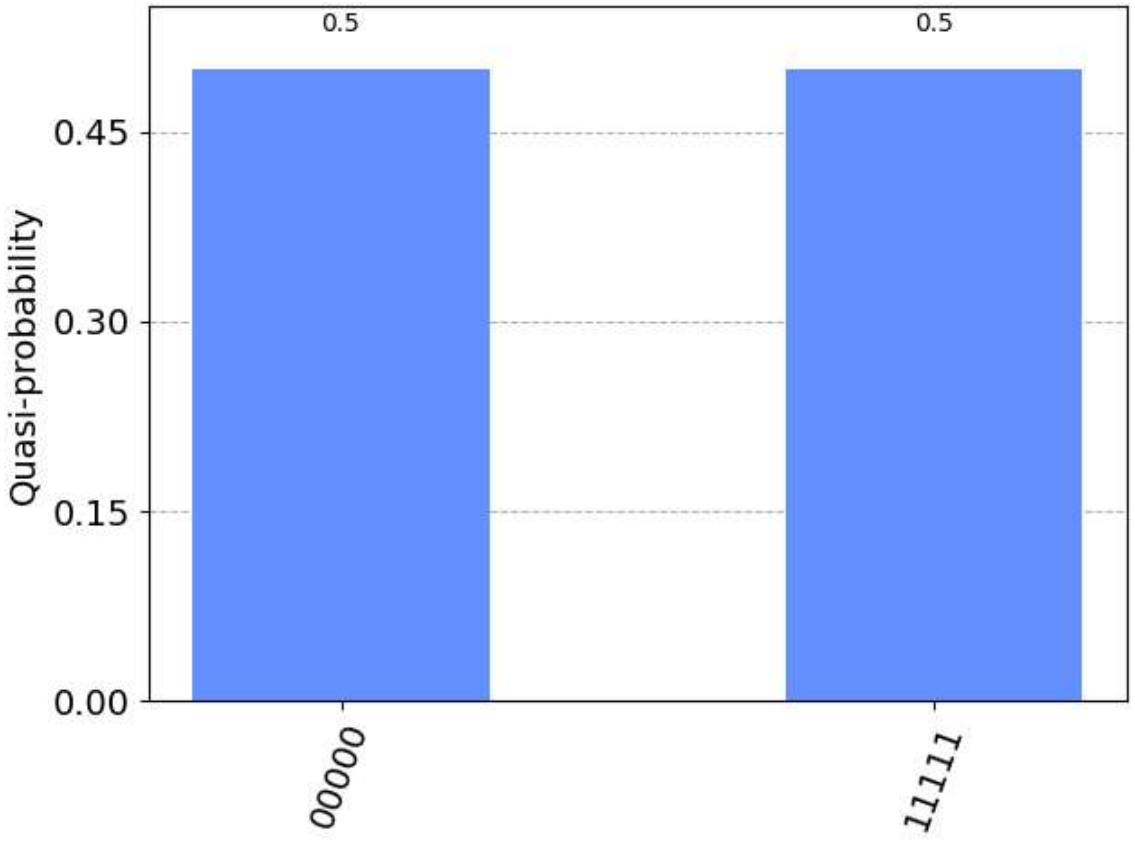

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

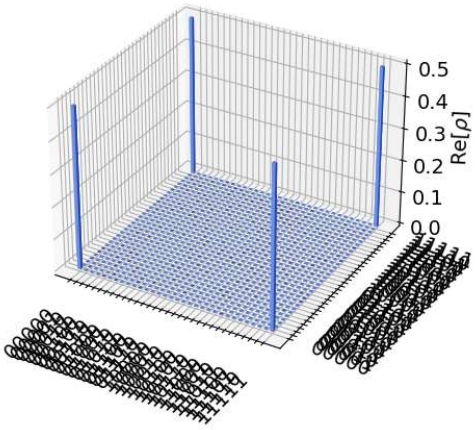

Re[ρ]
Im[ρ]

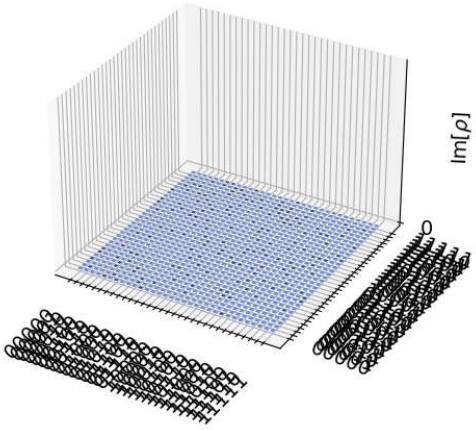

statevector
H
q_0
q_1
q_2
q_3
q_4

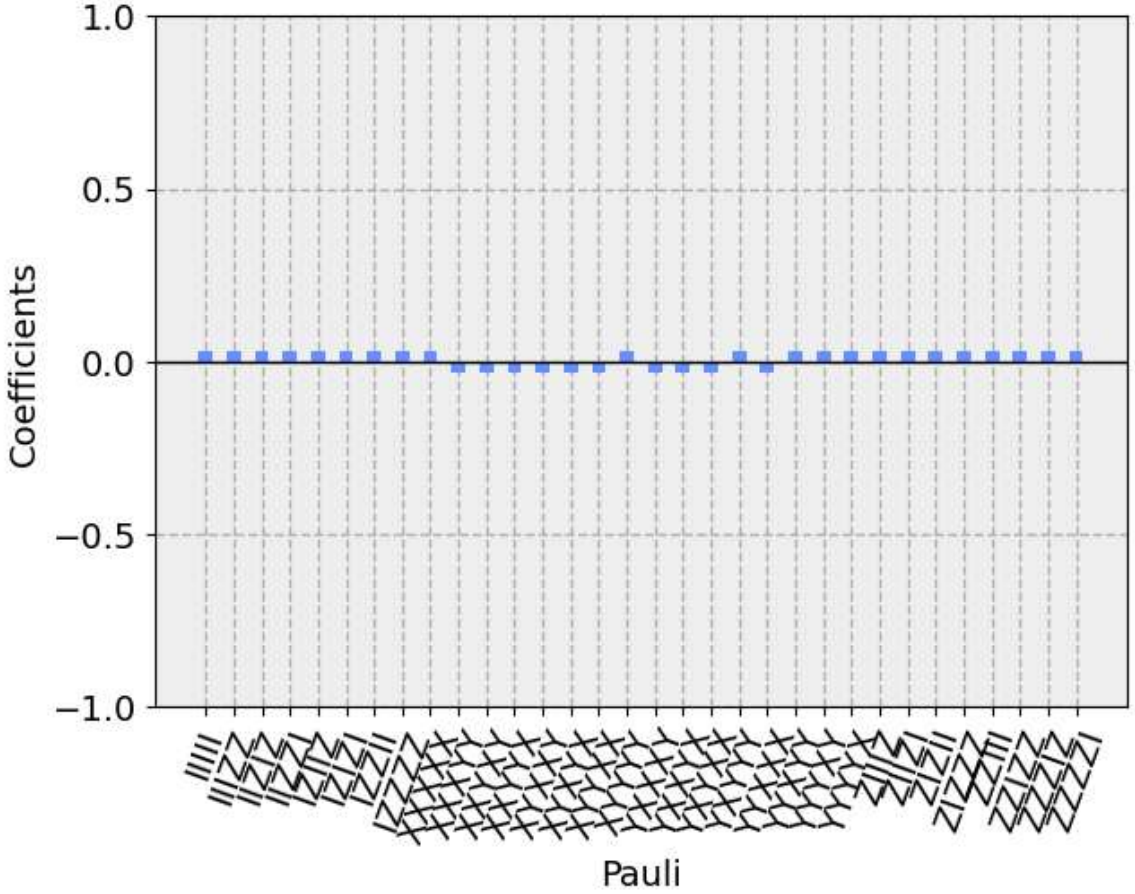

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

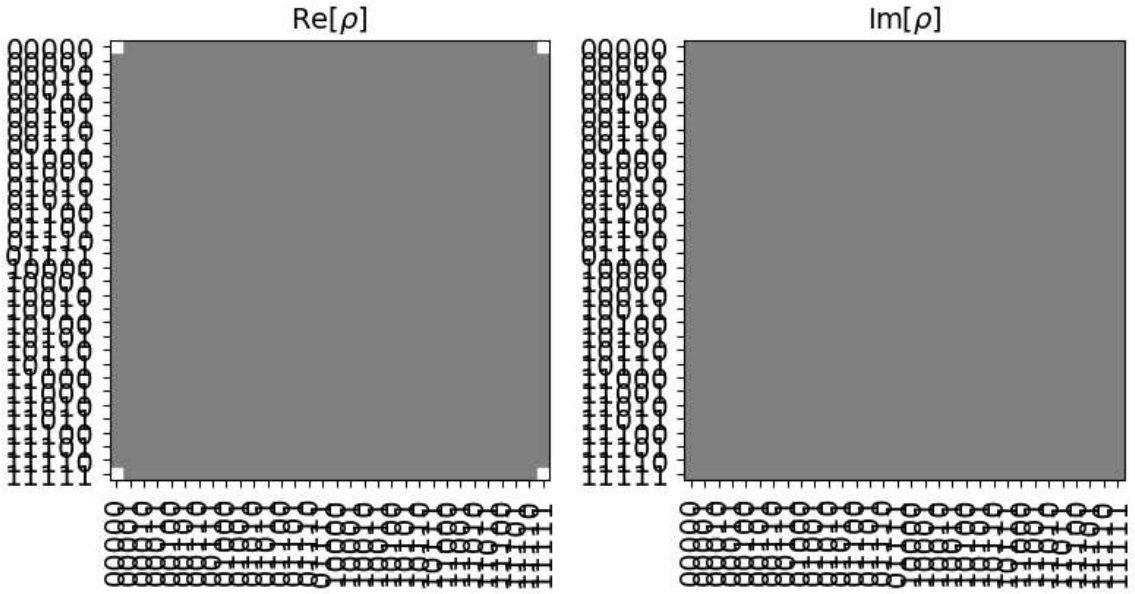

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

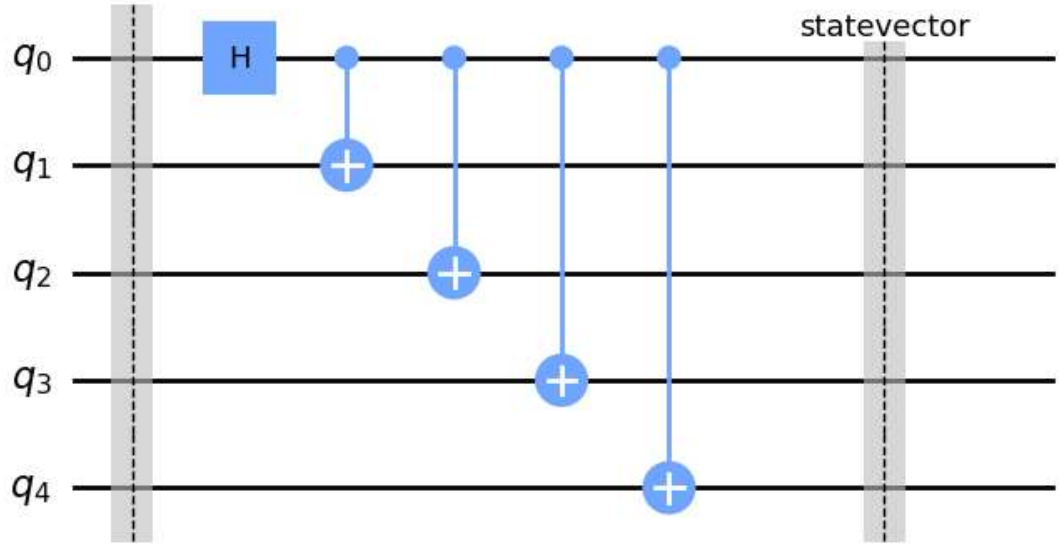

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

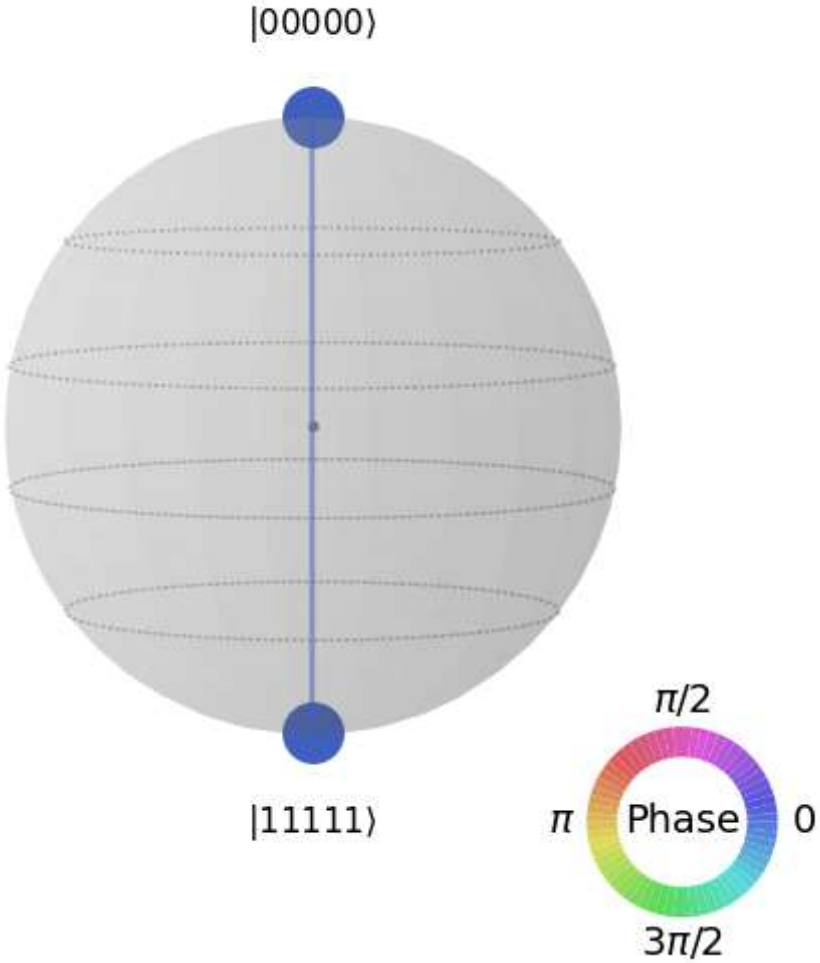

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

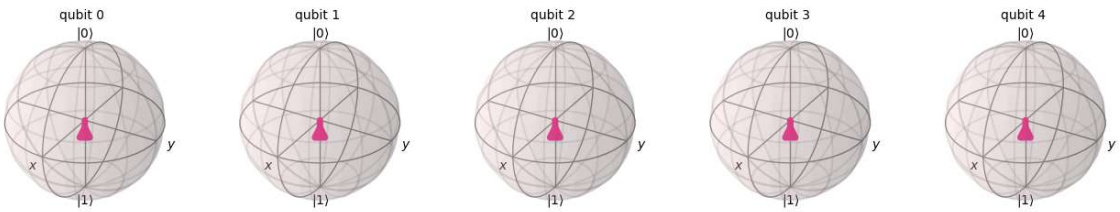

Im[ρ]
0

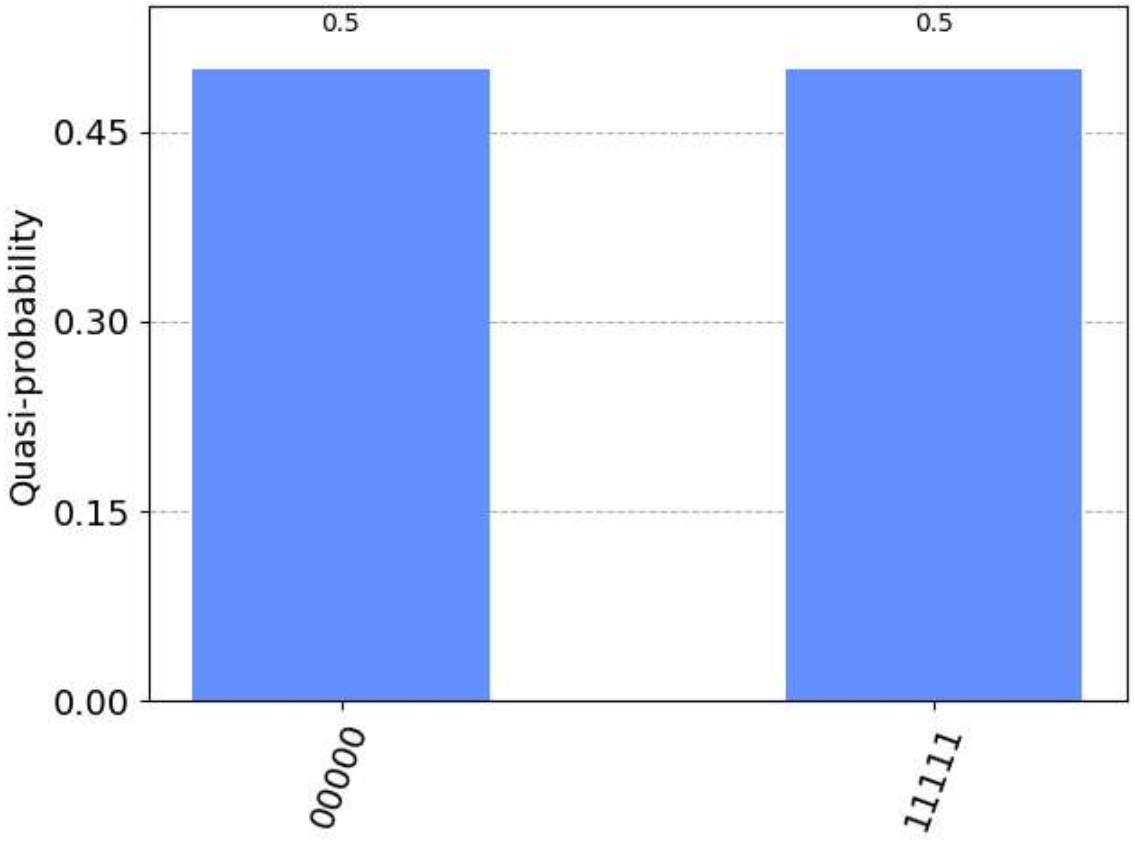

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

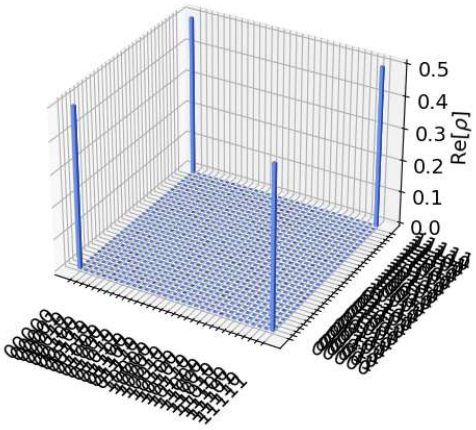

Re[ρ]
Im[ρ]

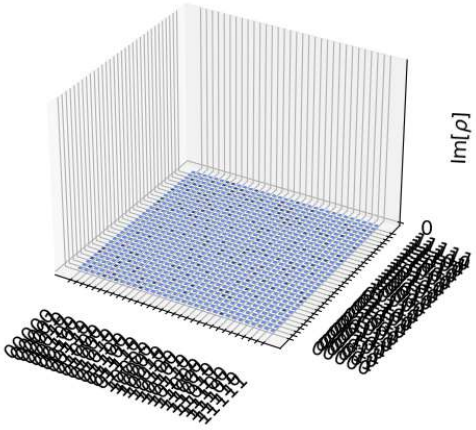

statevector
H
q_0
q_1
q_2
q_3
q_4

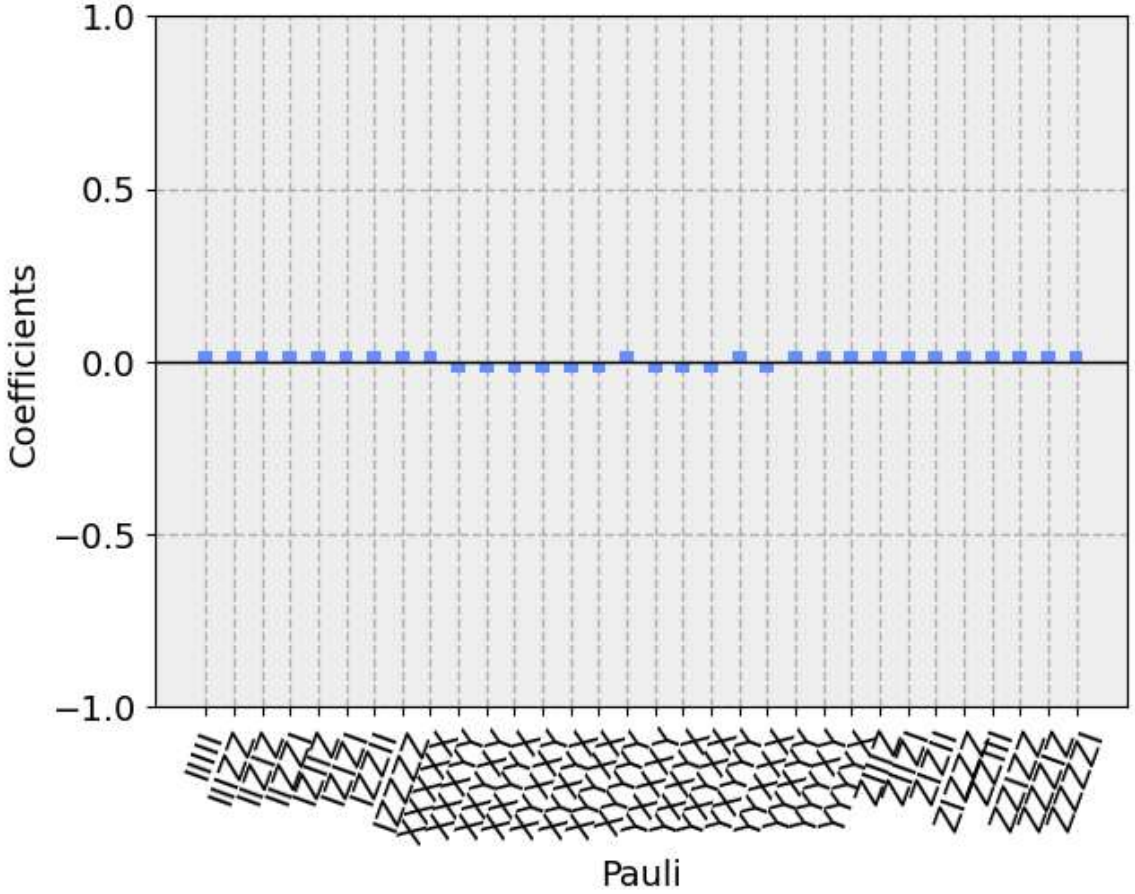

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

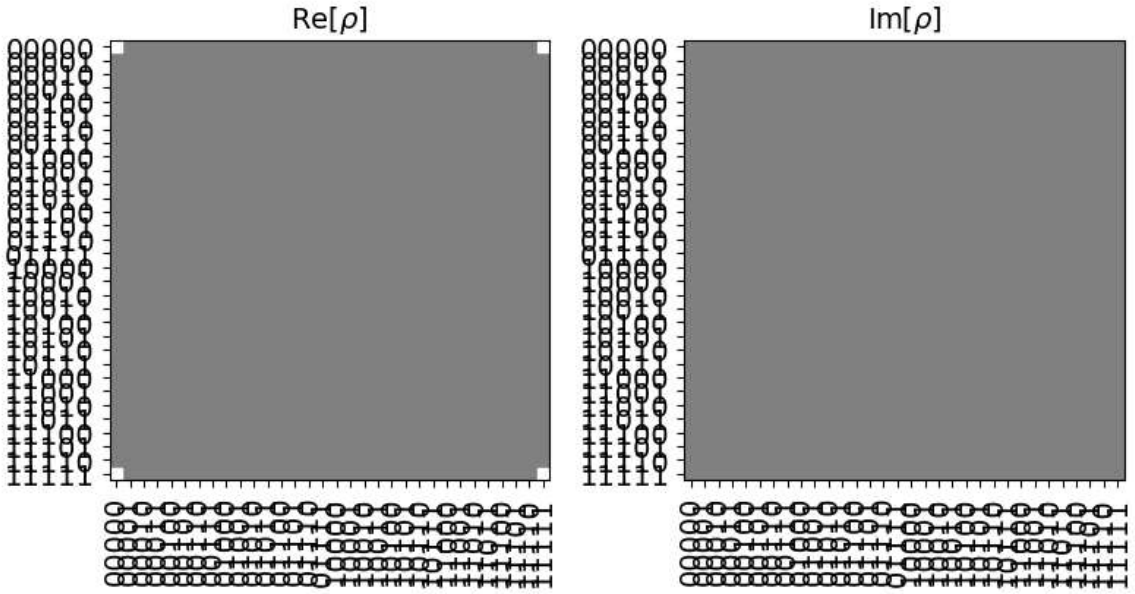

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

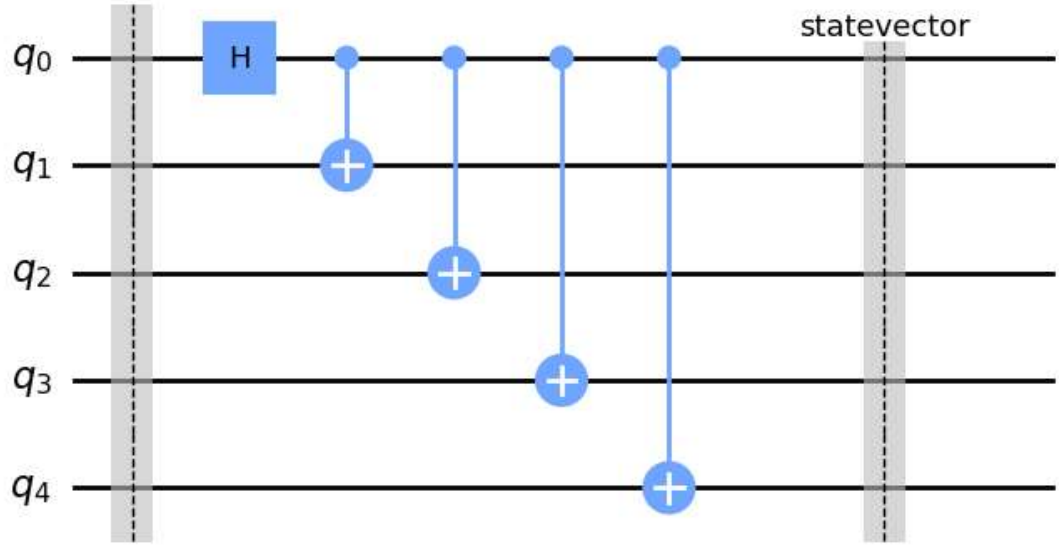

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

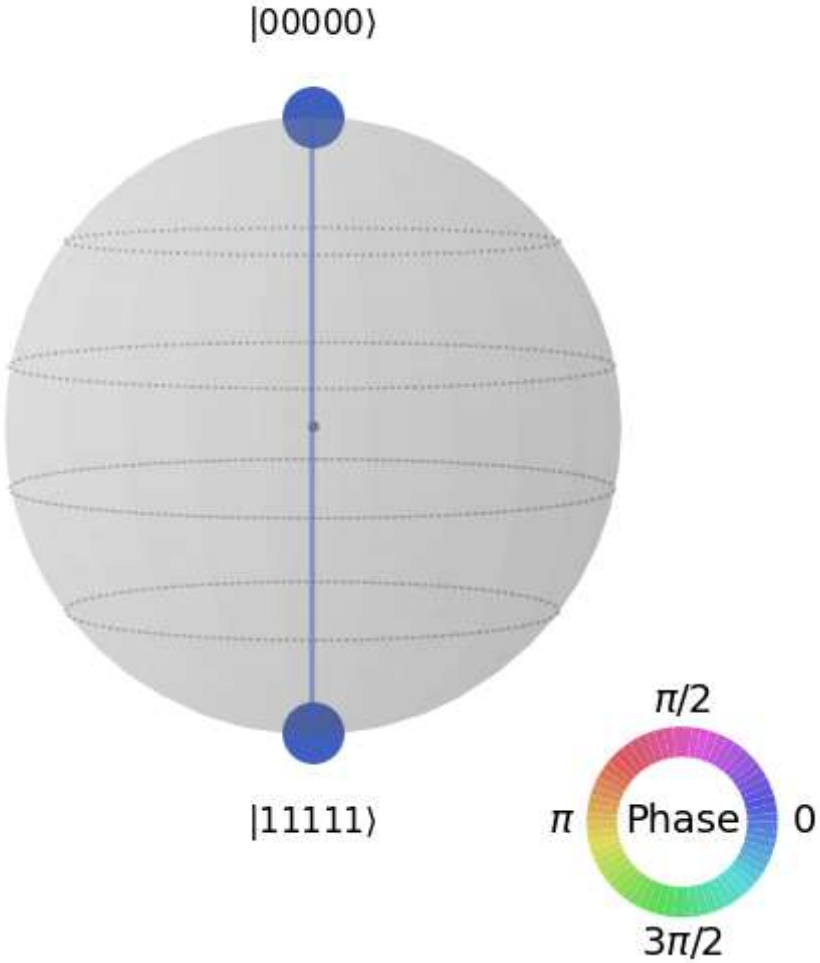

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

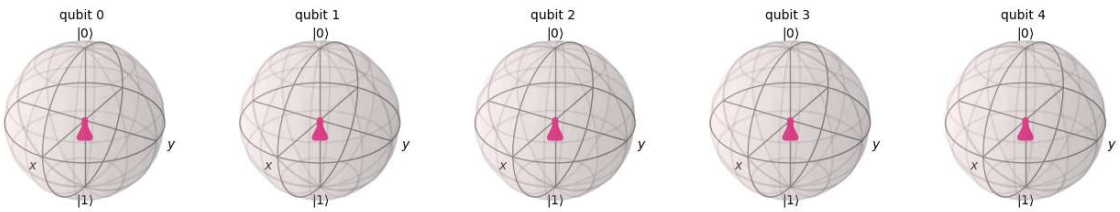

Im[ρ]
0

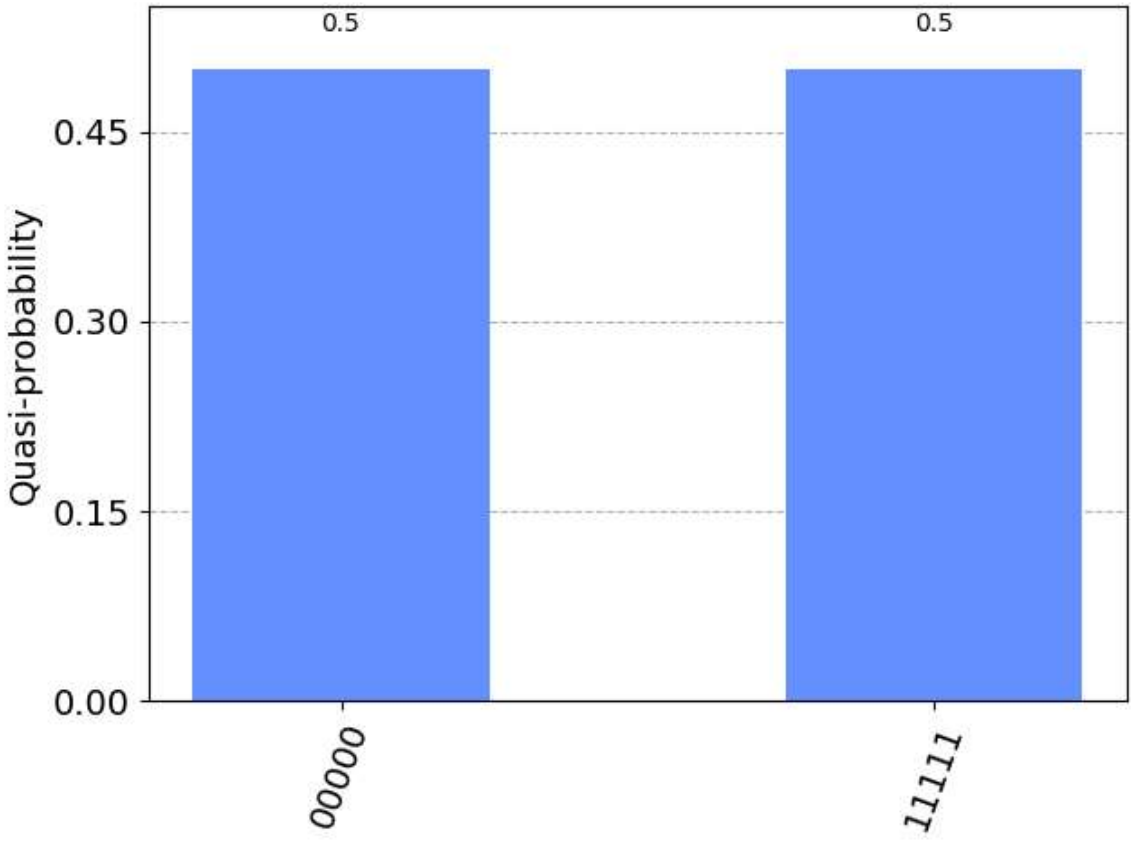

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

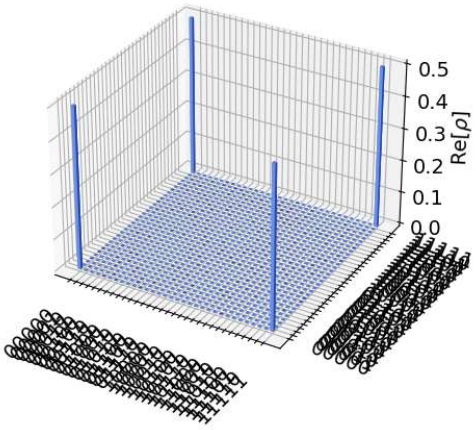

Re[ρ]
Im[ρ]

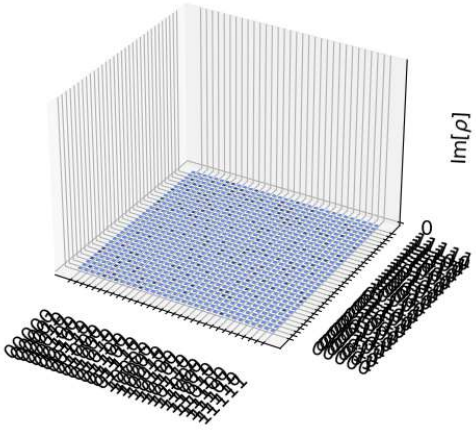

statevector
H
q_0
q_1
q_2
q_3
q_4

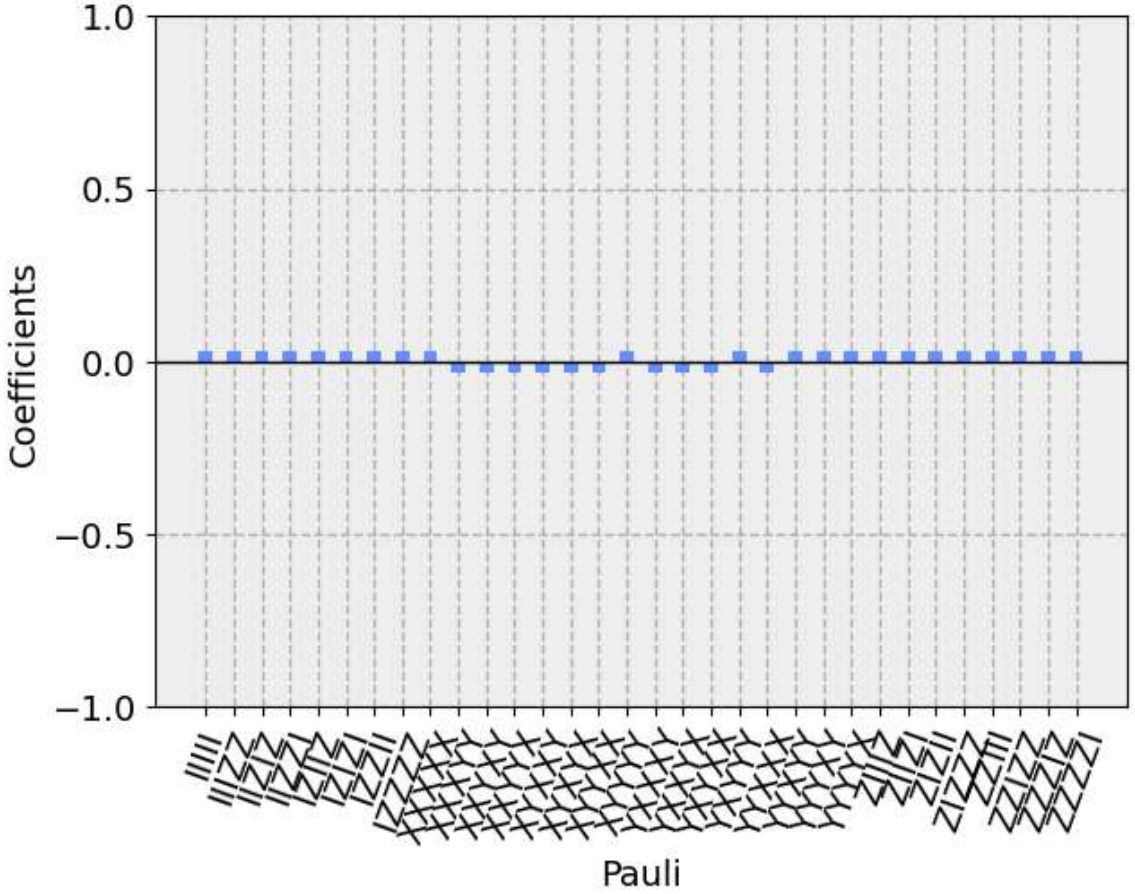

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

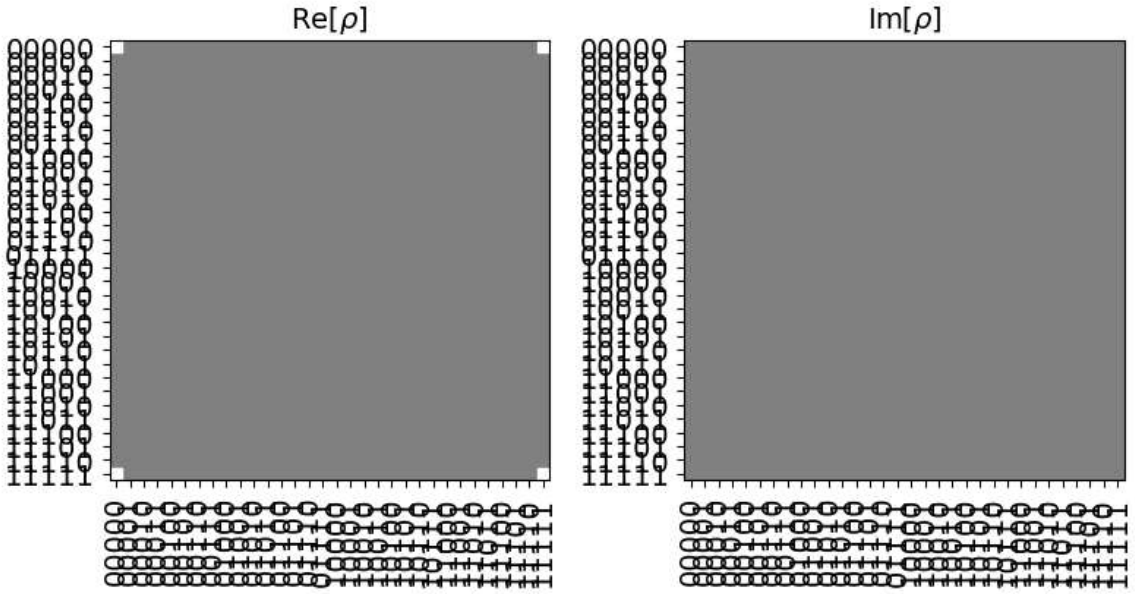

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

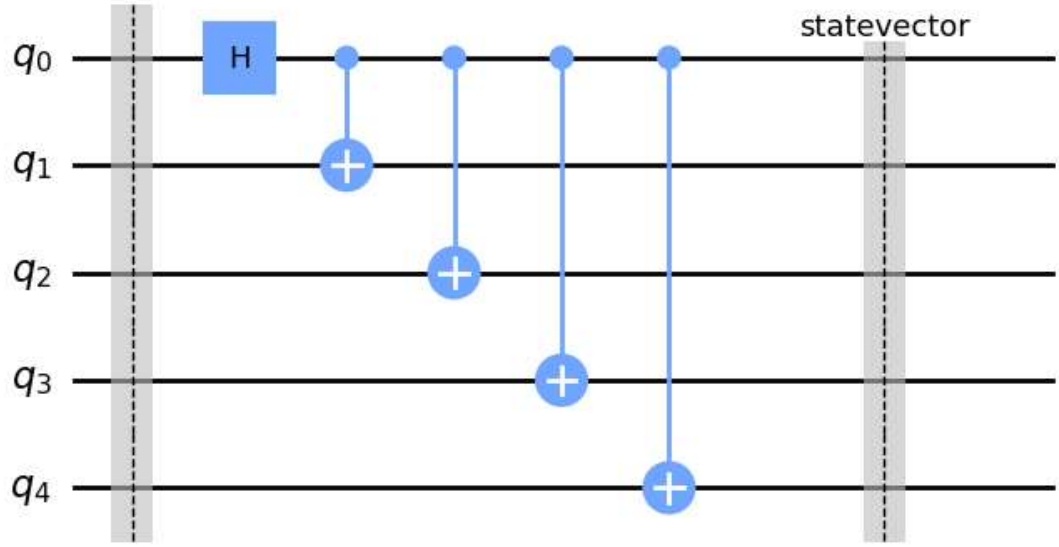

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

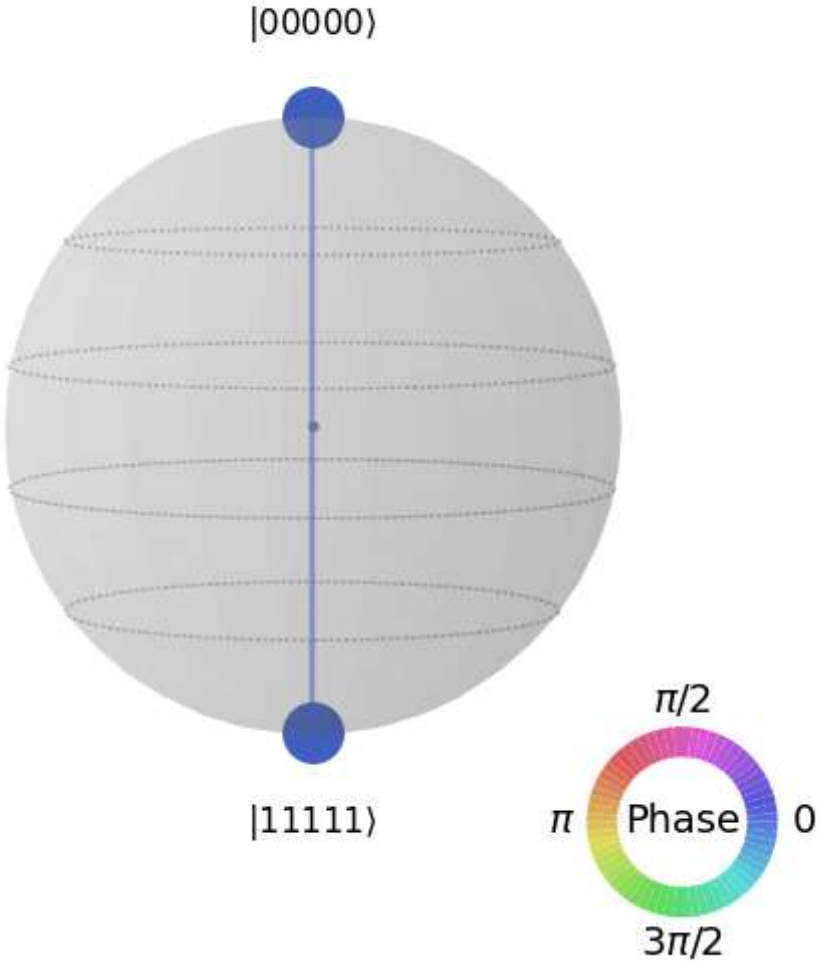

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

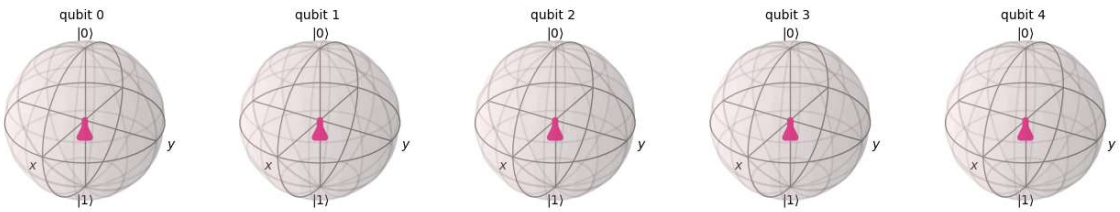

Im[ρ]
0

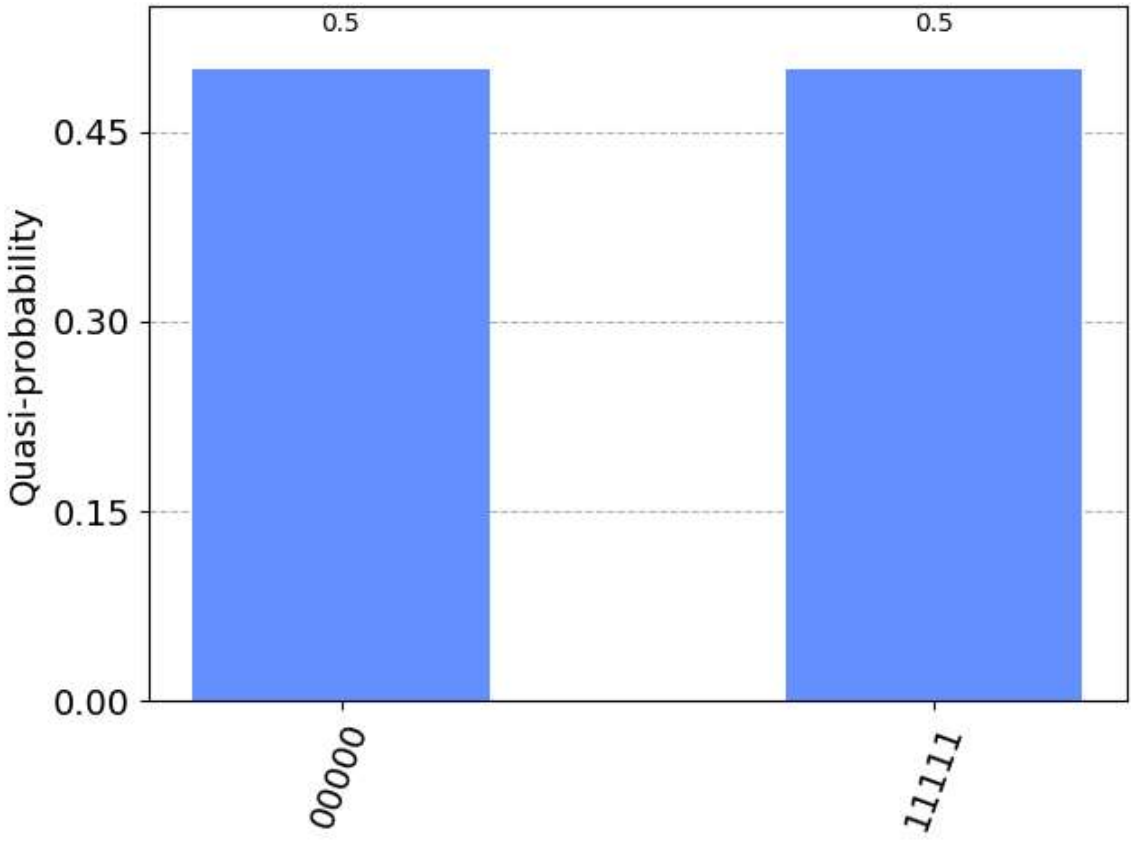

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

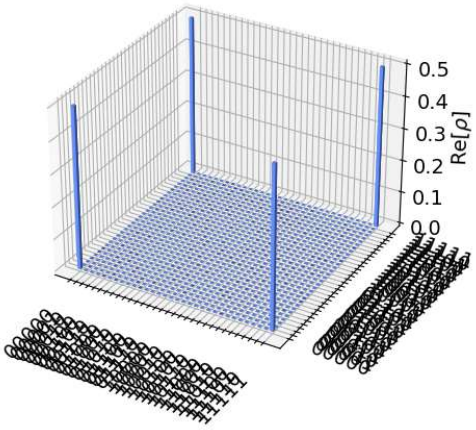

Re[ρ]
Im[ρ]

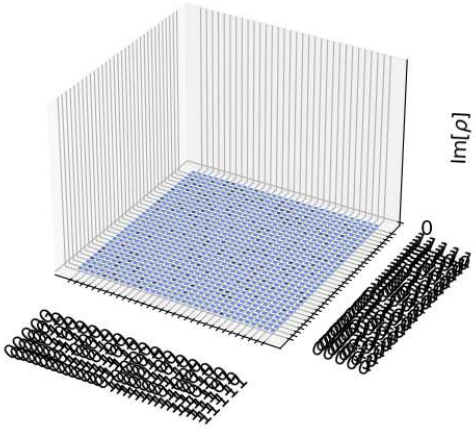

statevector
H
q_0
q_1
q_2
q_3
q_4

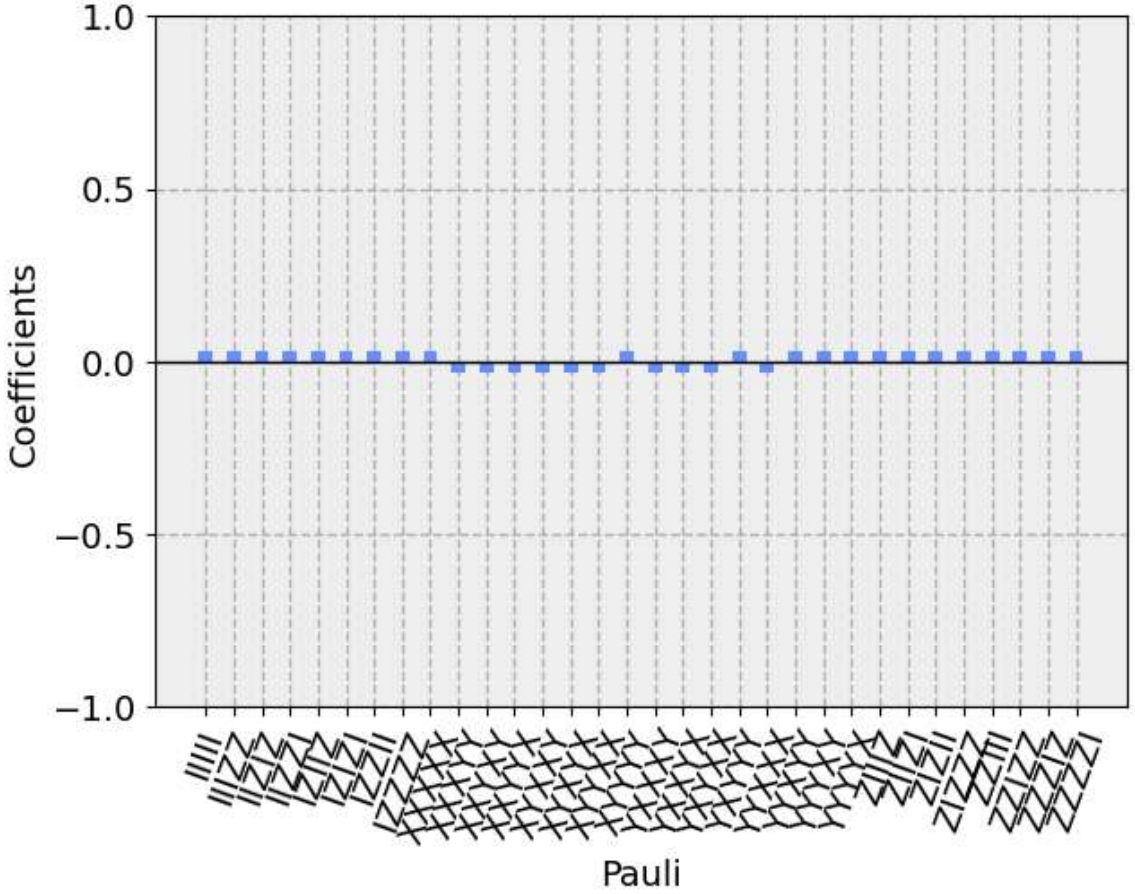

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

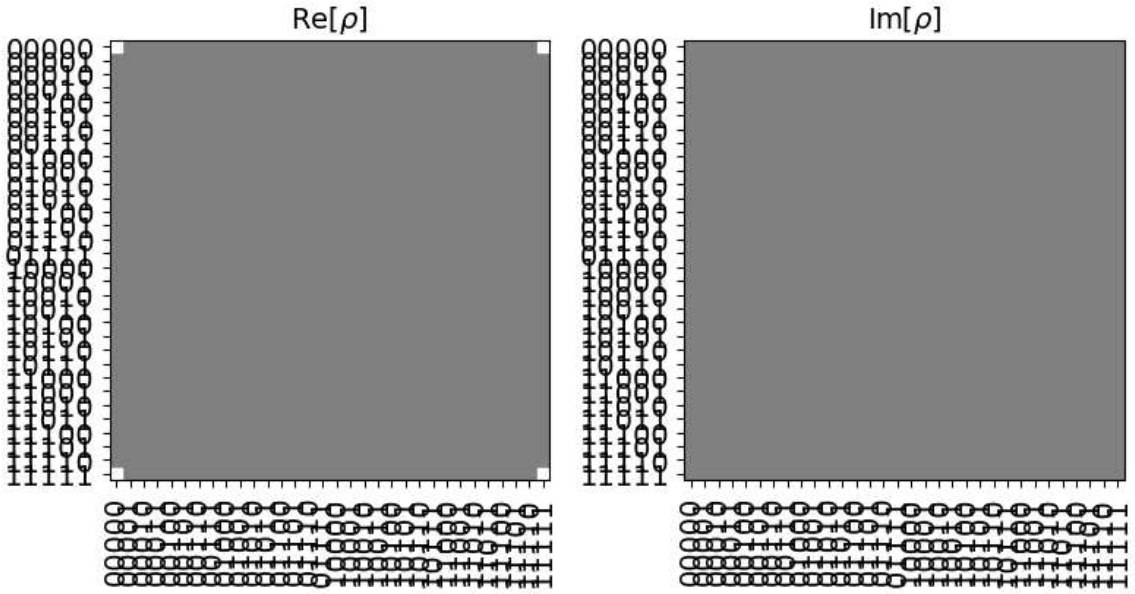

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

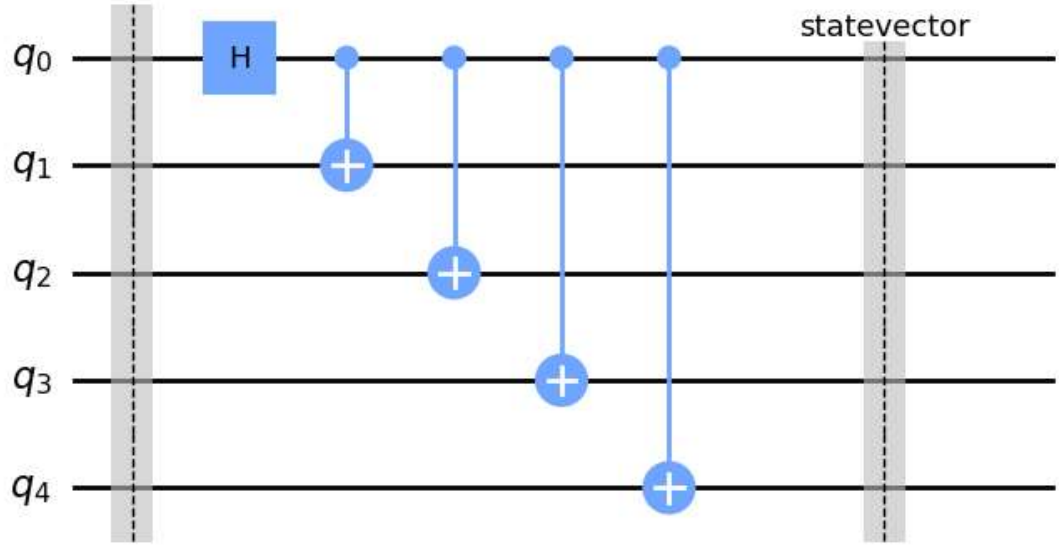

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

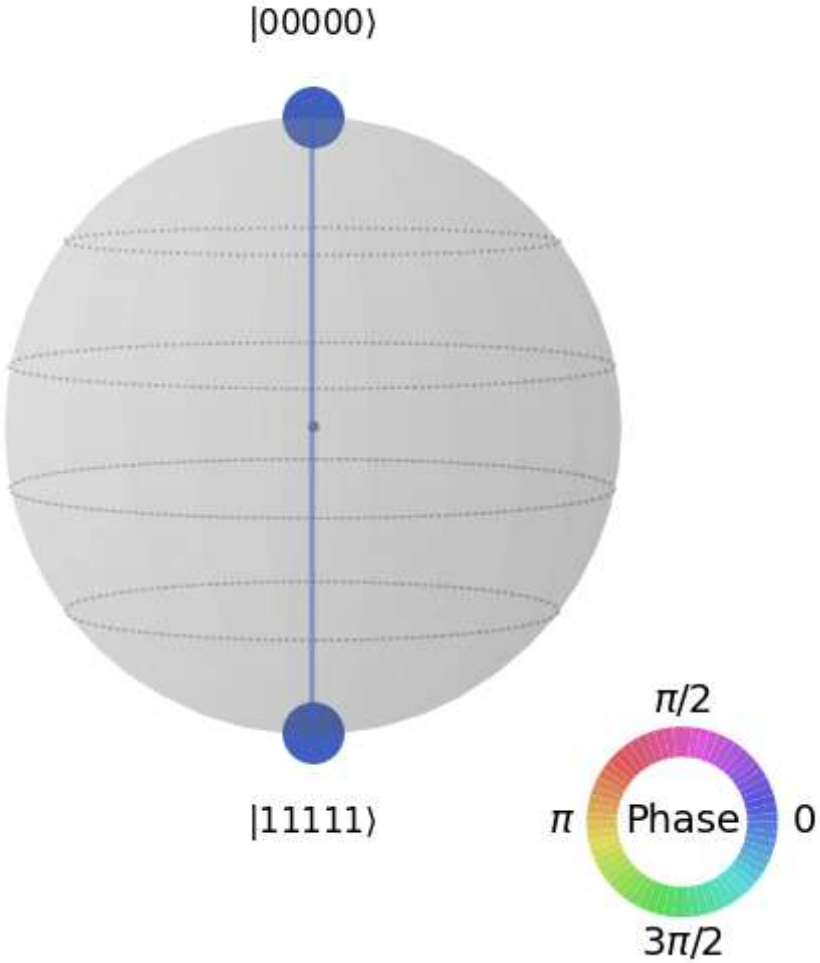

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

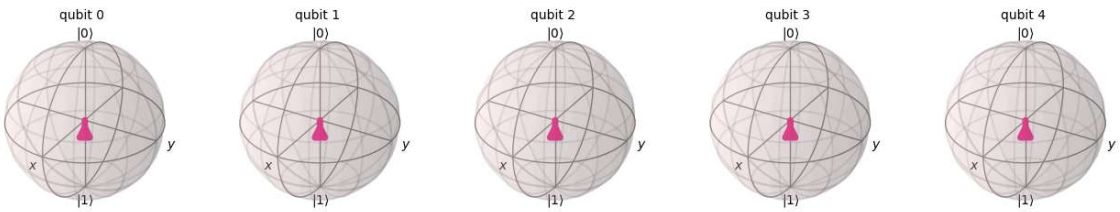

Im[ρ]
0

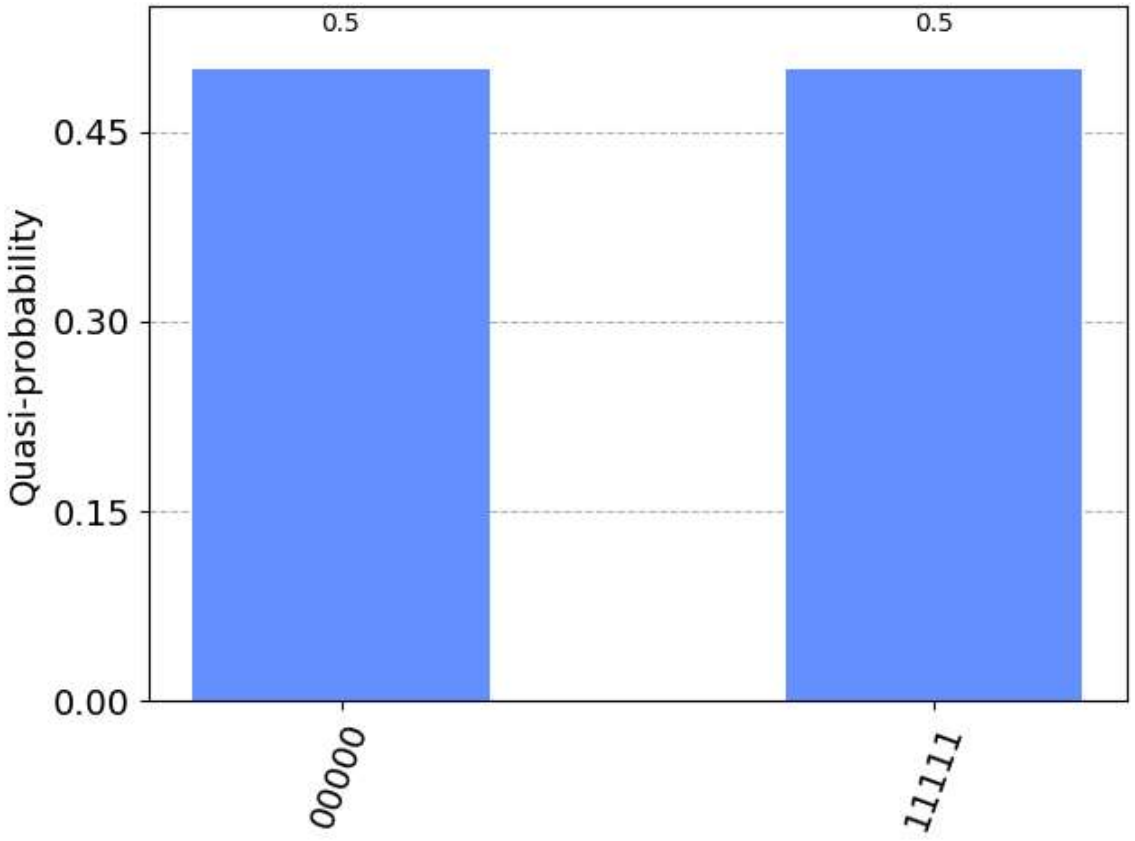

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

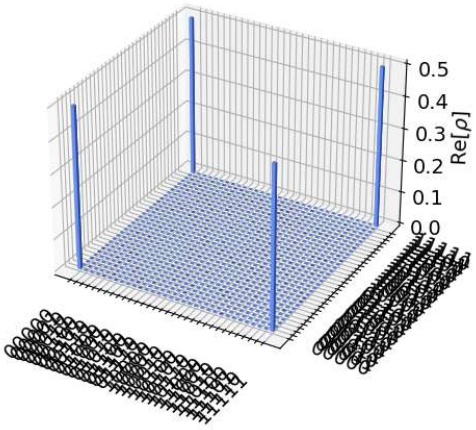

Re[ρ]
Im[ρ]

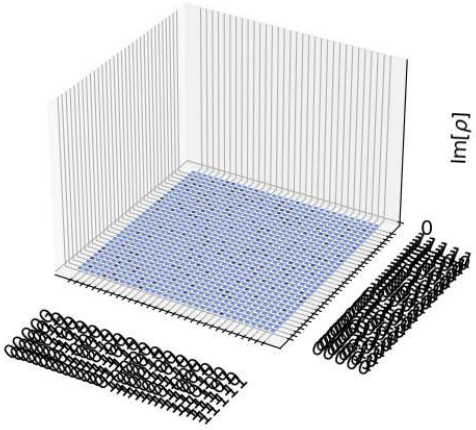

statevector
H
q_0
q_1
q_2
q_3
q_4

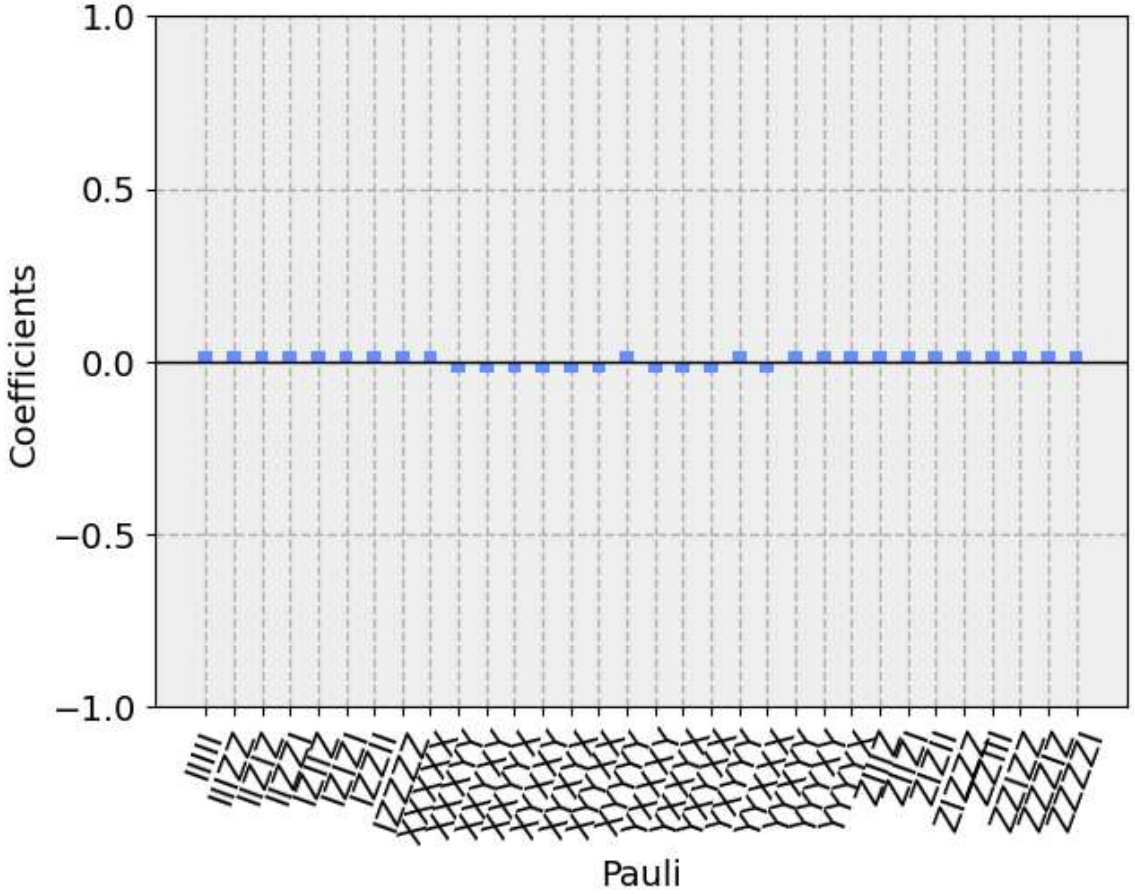

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

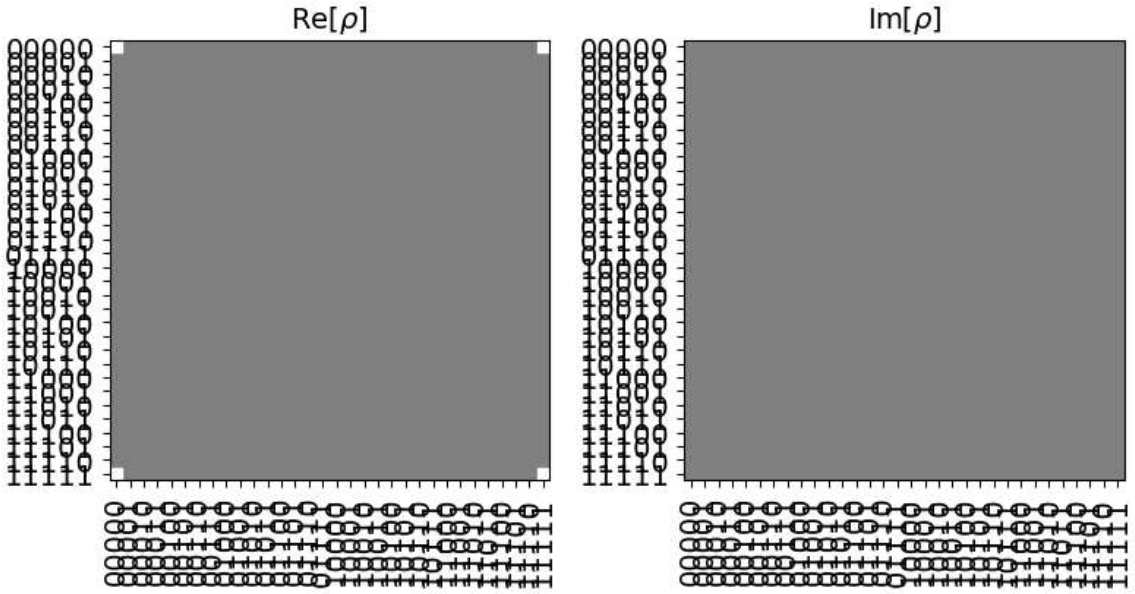

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

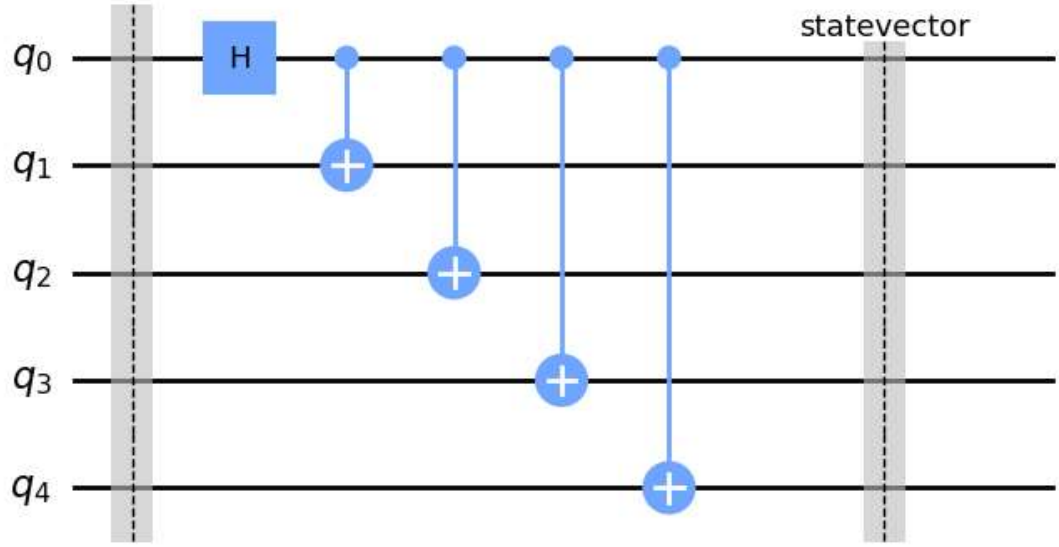

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

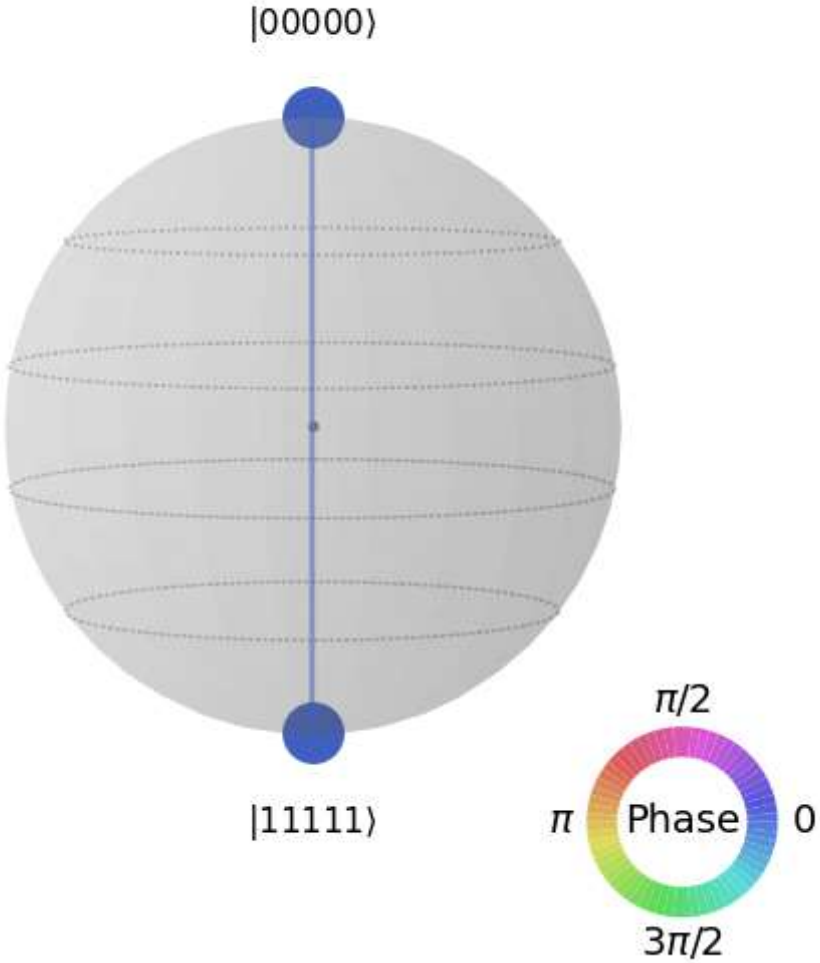

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

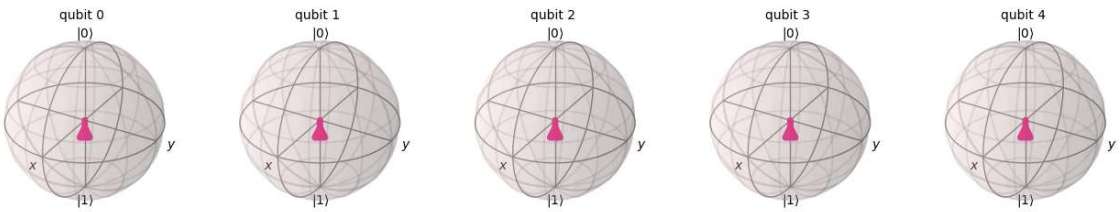

Im[ρ]
0

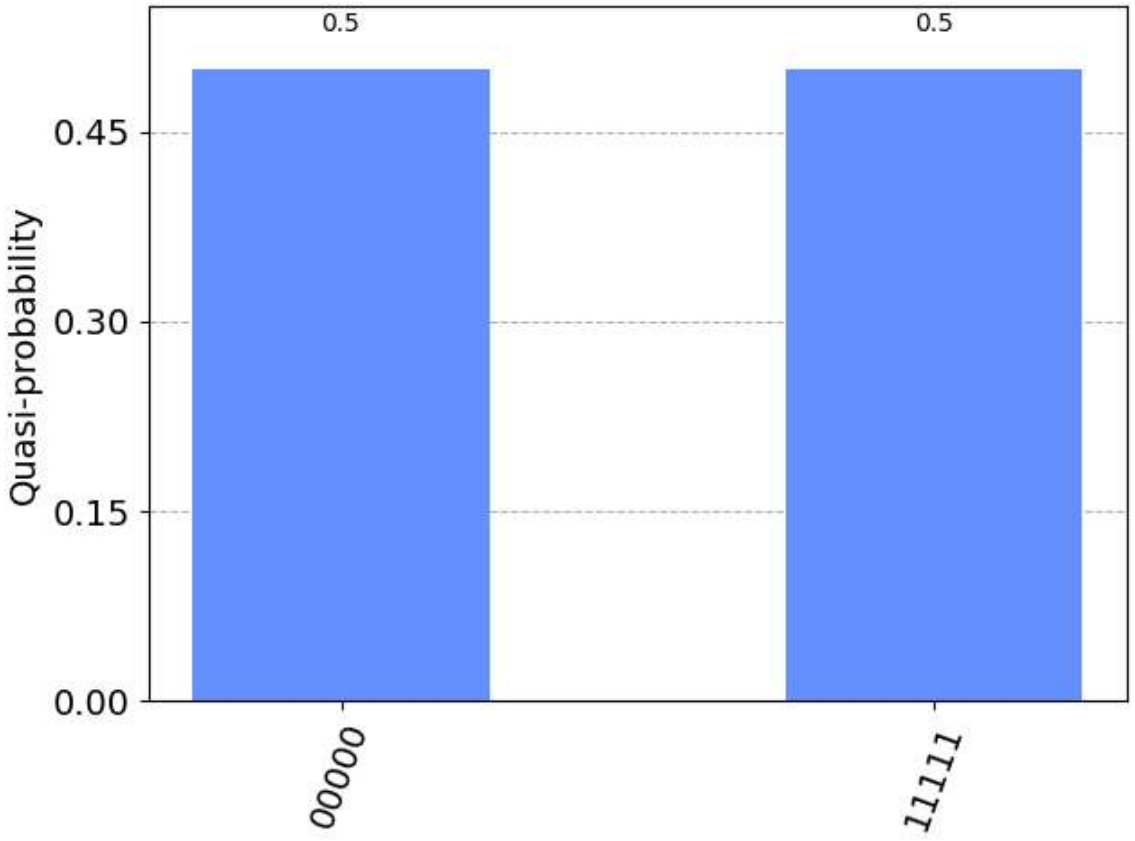

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

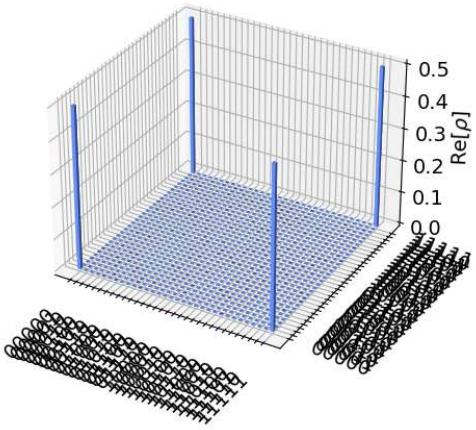

Re[ρ]
Im[ρ]

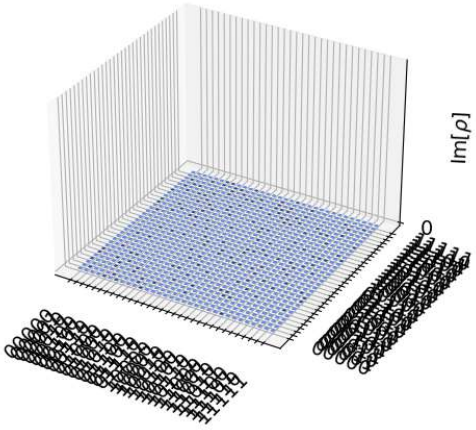

statevector
H
q_0
q_1
q_2
q_3
q_4

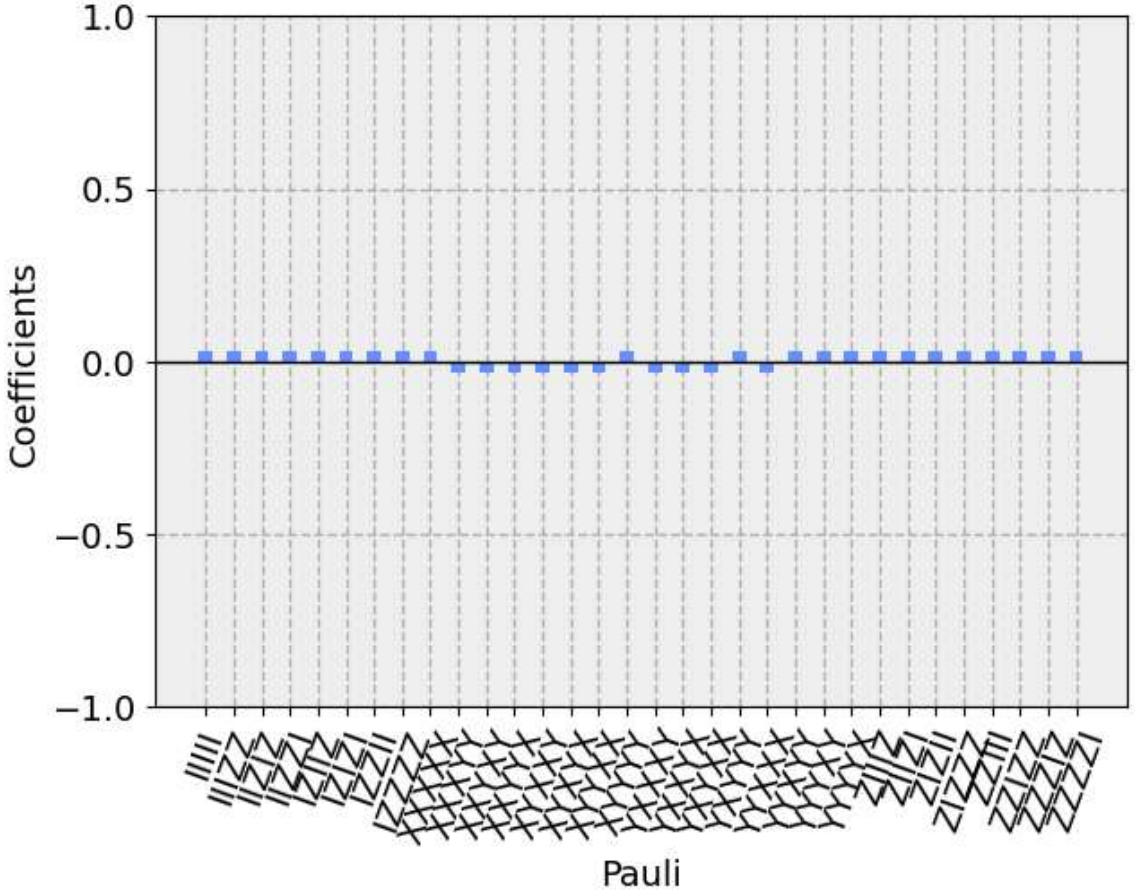

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

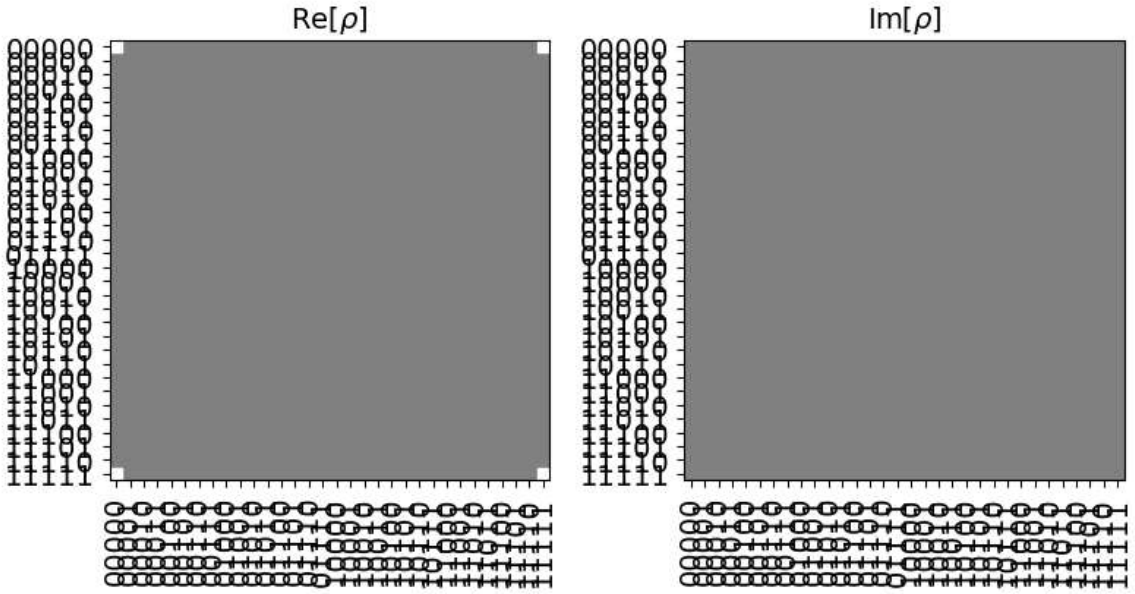

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

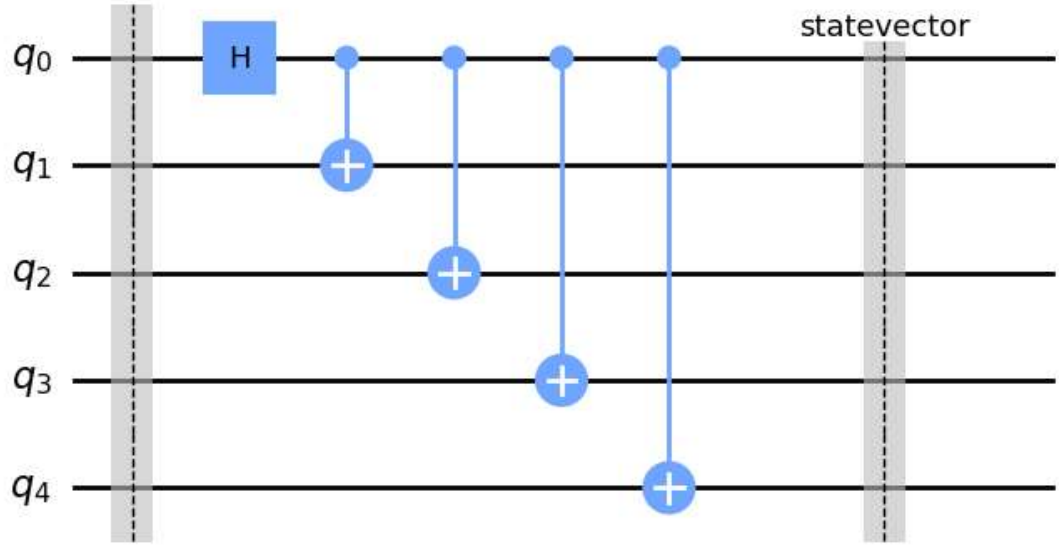

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

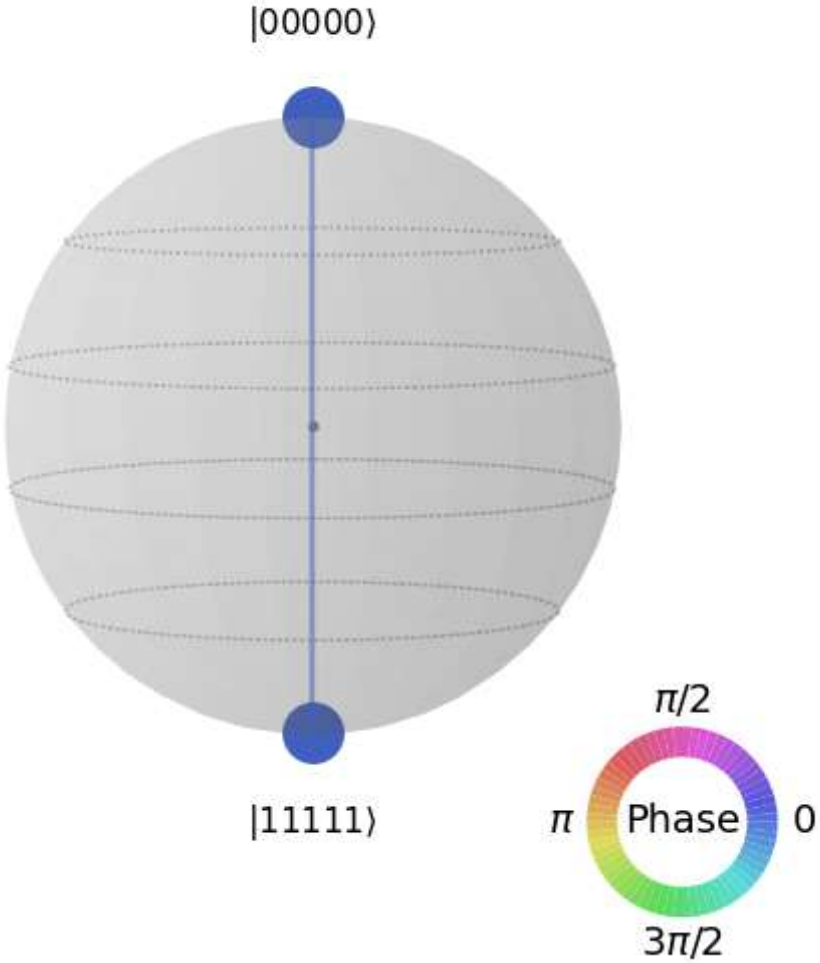

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

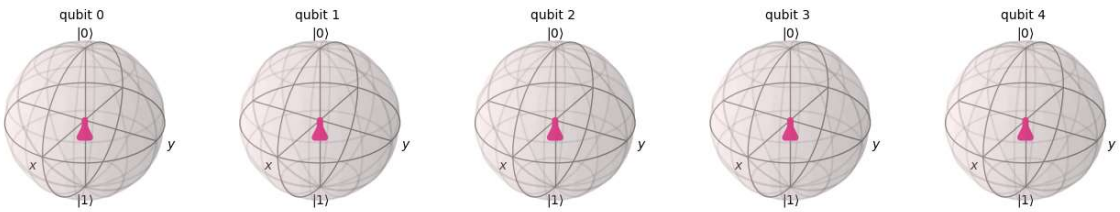

Im[ρ]
0

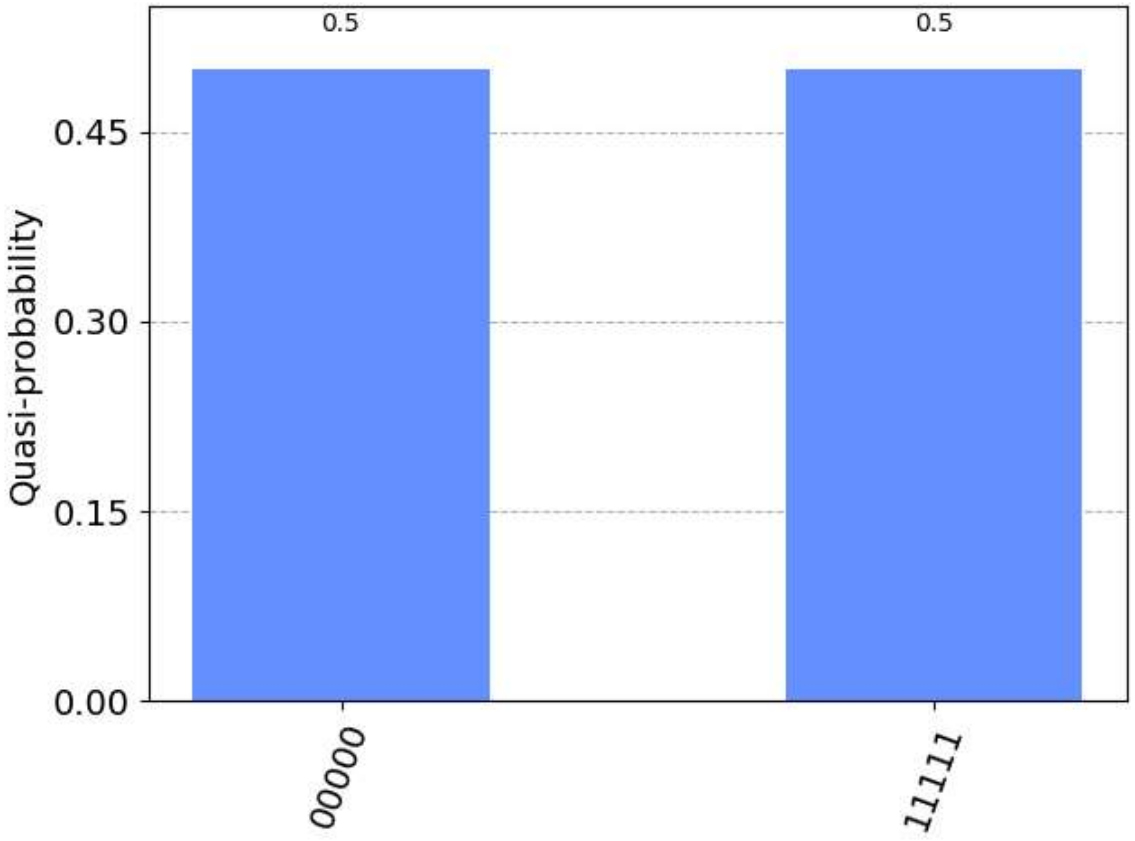

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

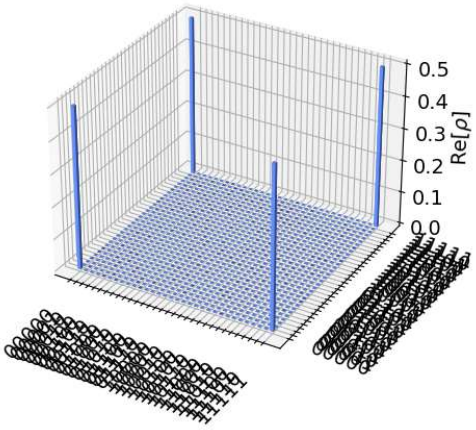

Re[ρ]
Im[ρ]

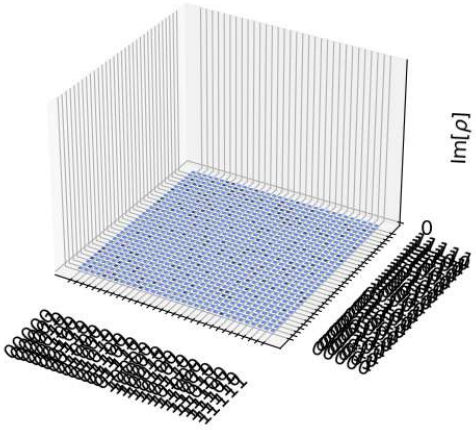

statevector
H
q_0
q_1
q_2
q_3
q_4

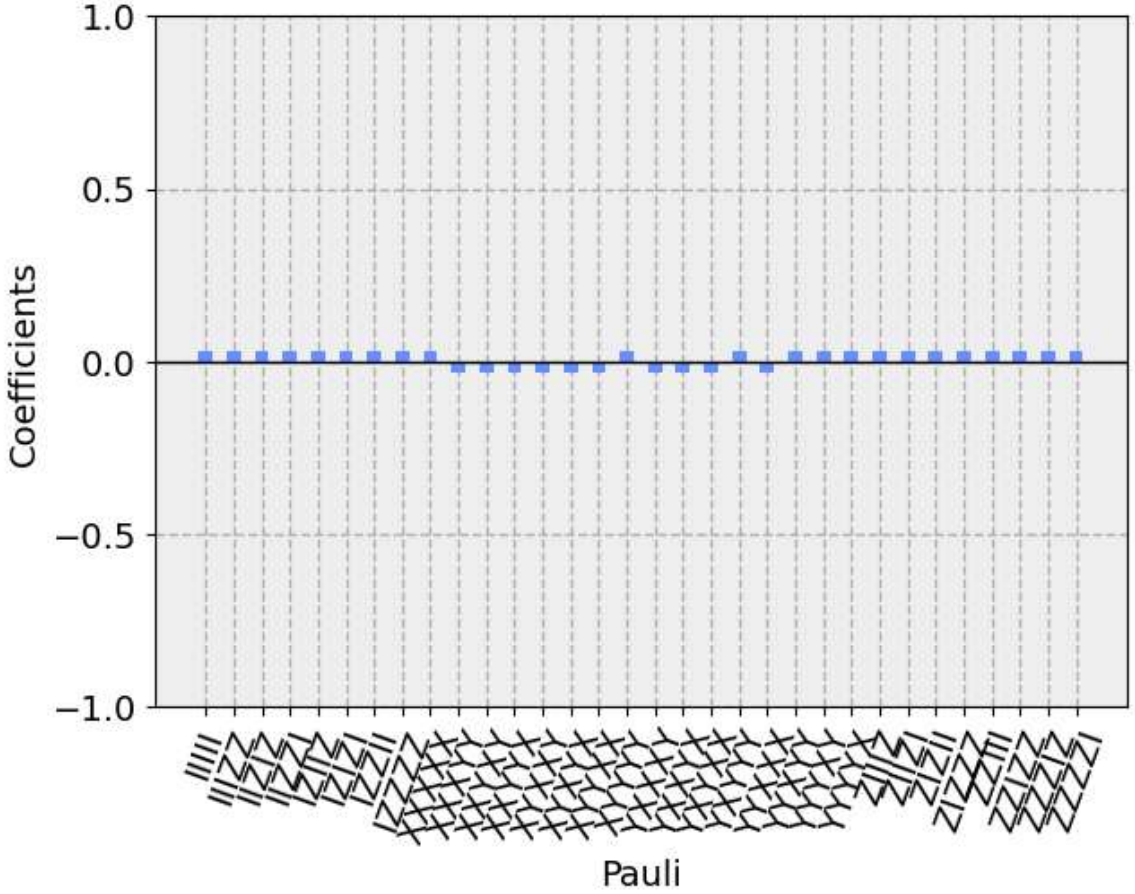

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

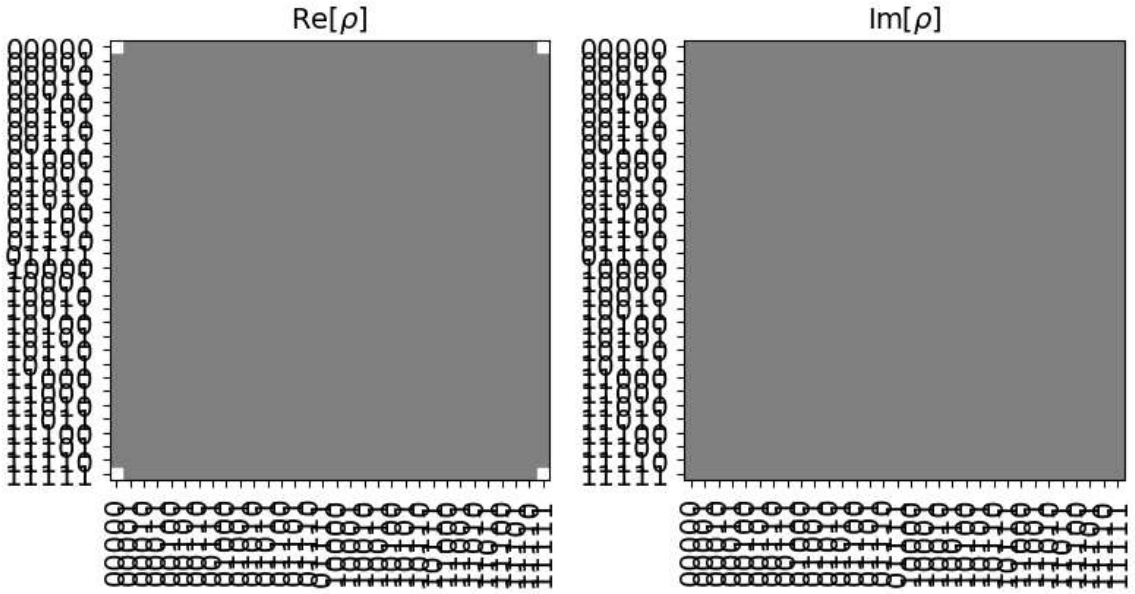

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

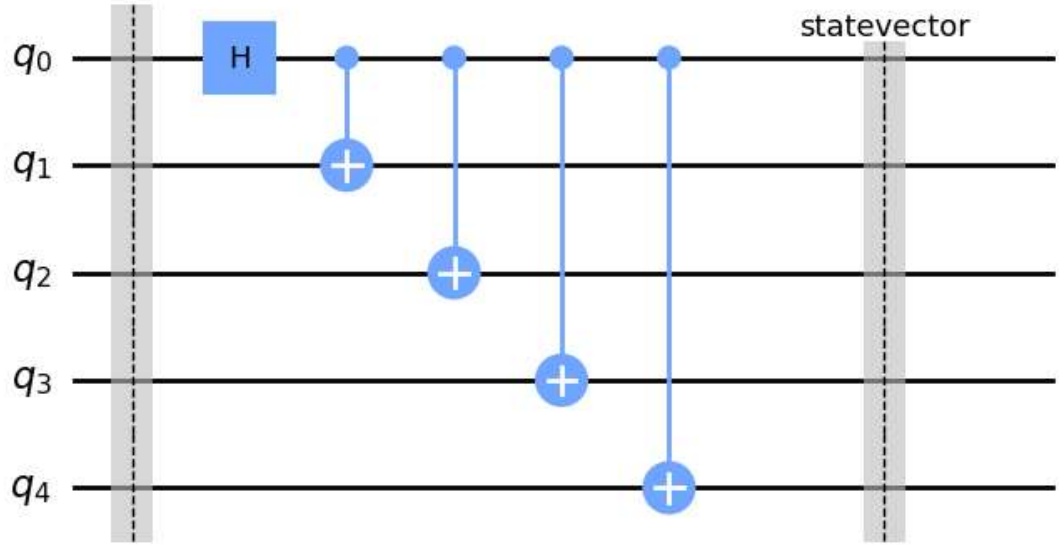

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

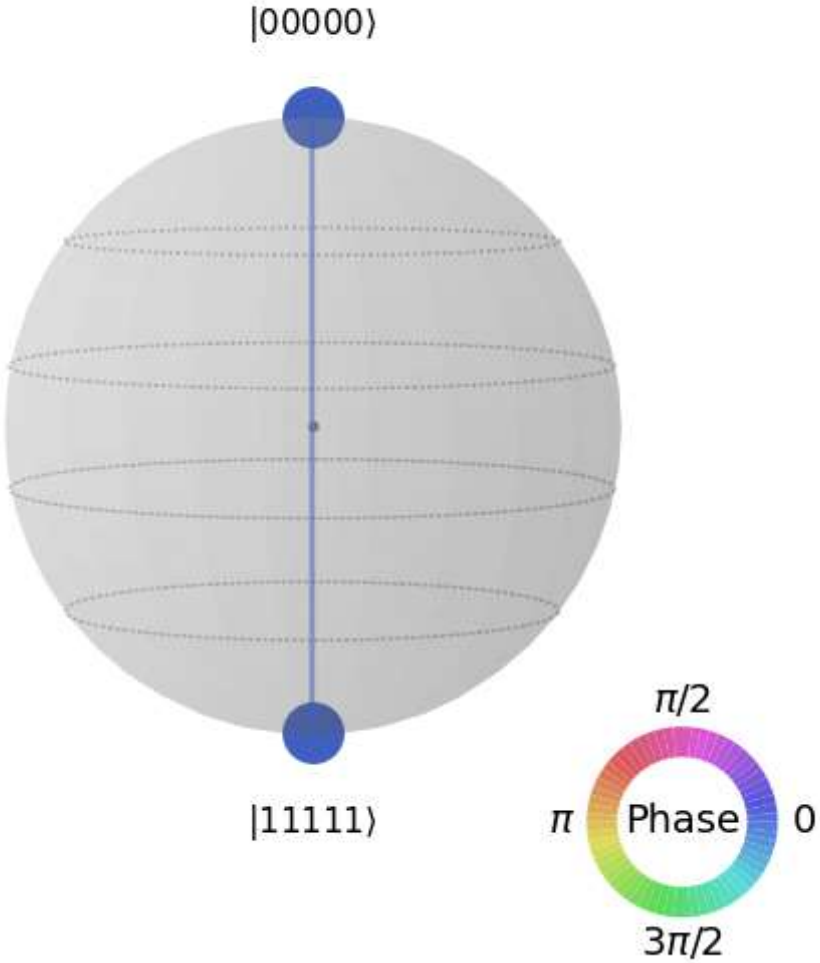

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

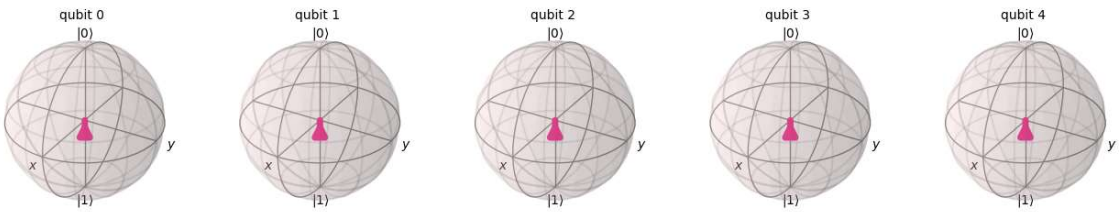

Im[ρ]
0

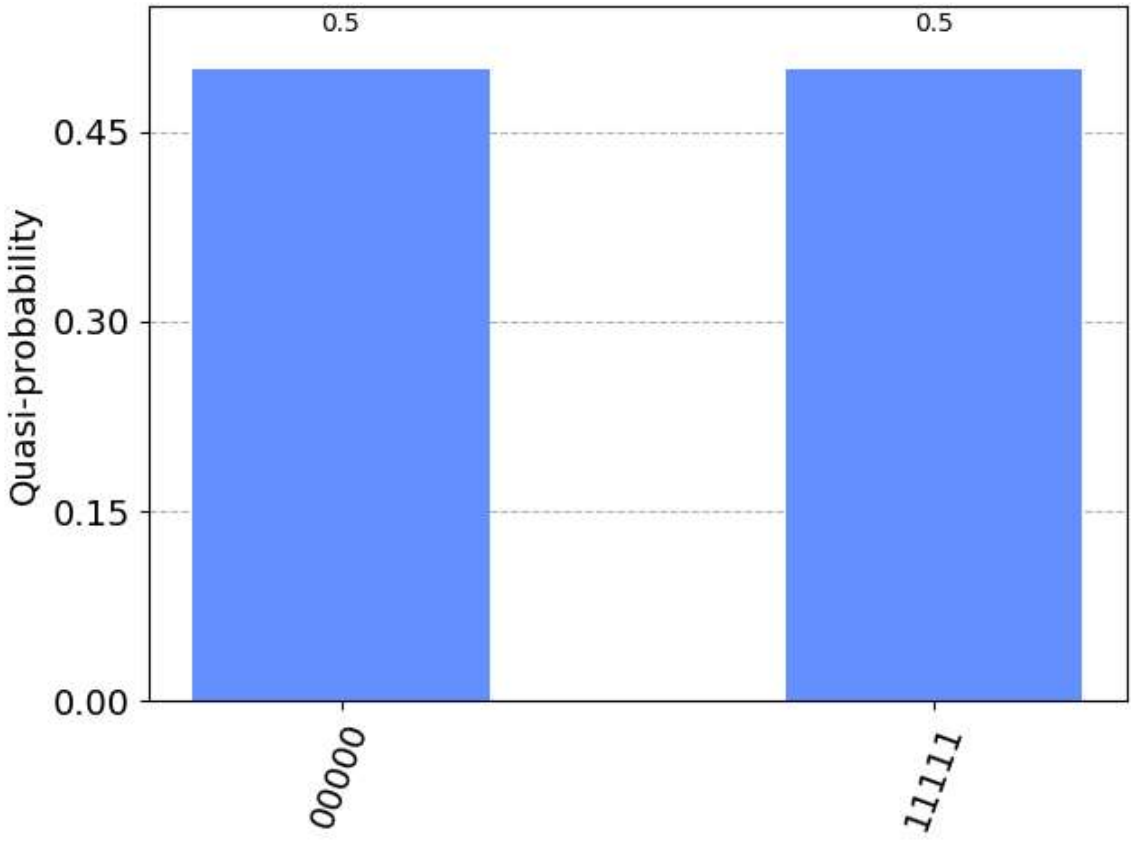

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

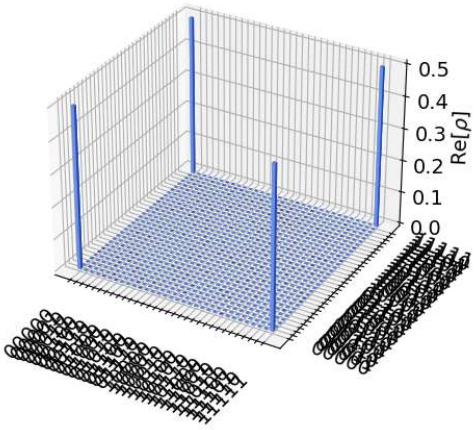

Re[ρ]
Im[ρ]

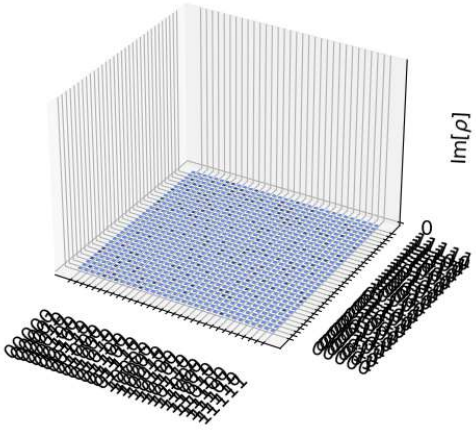

statevector
H
q_0
q_1
q_2
q_3
q_4

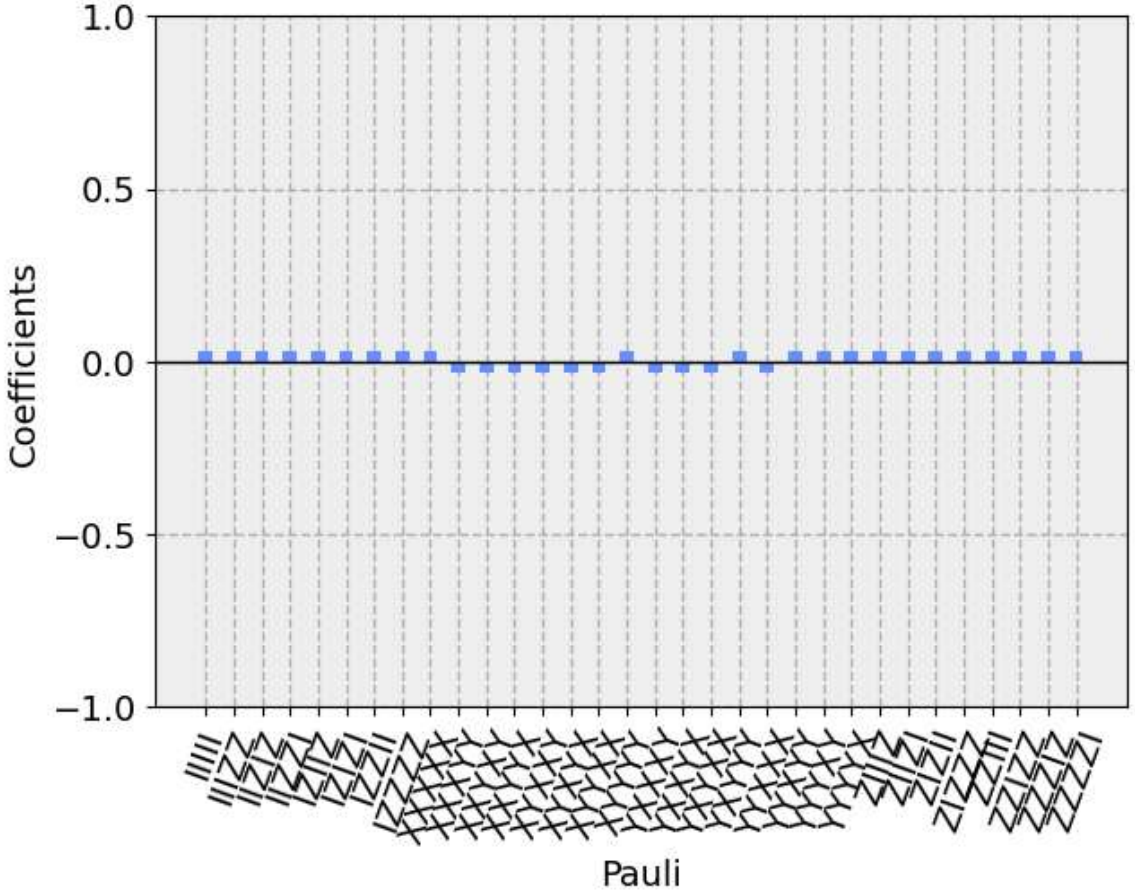

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

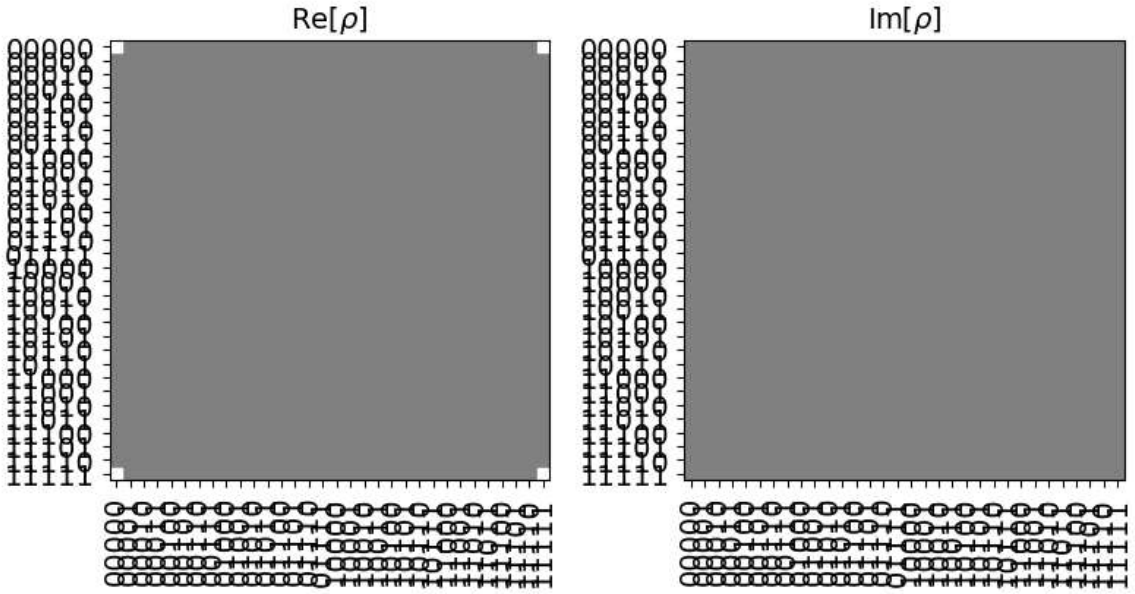

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

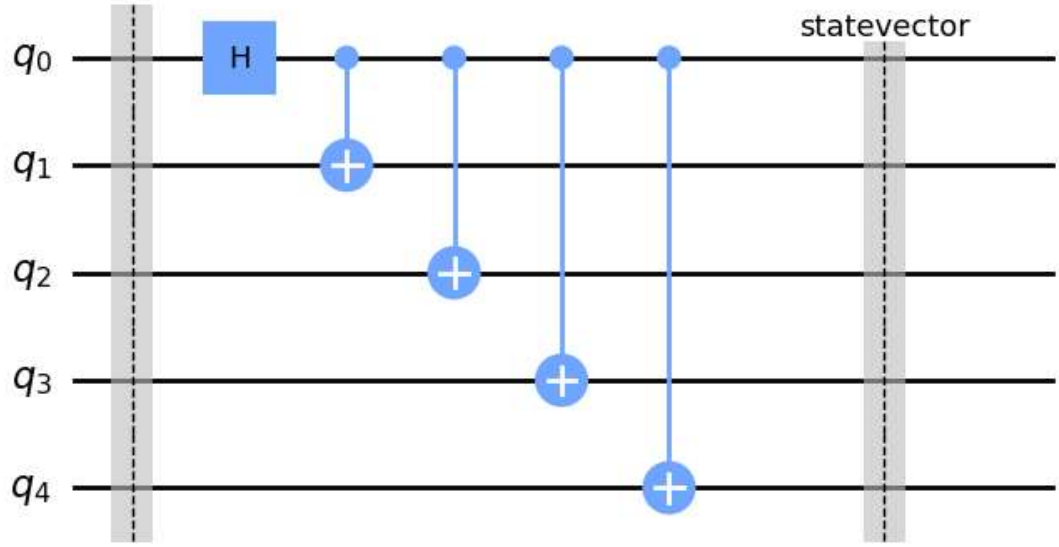

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

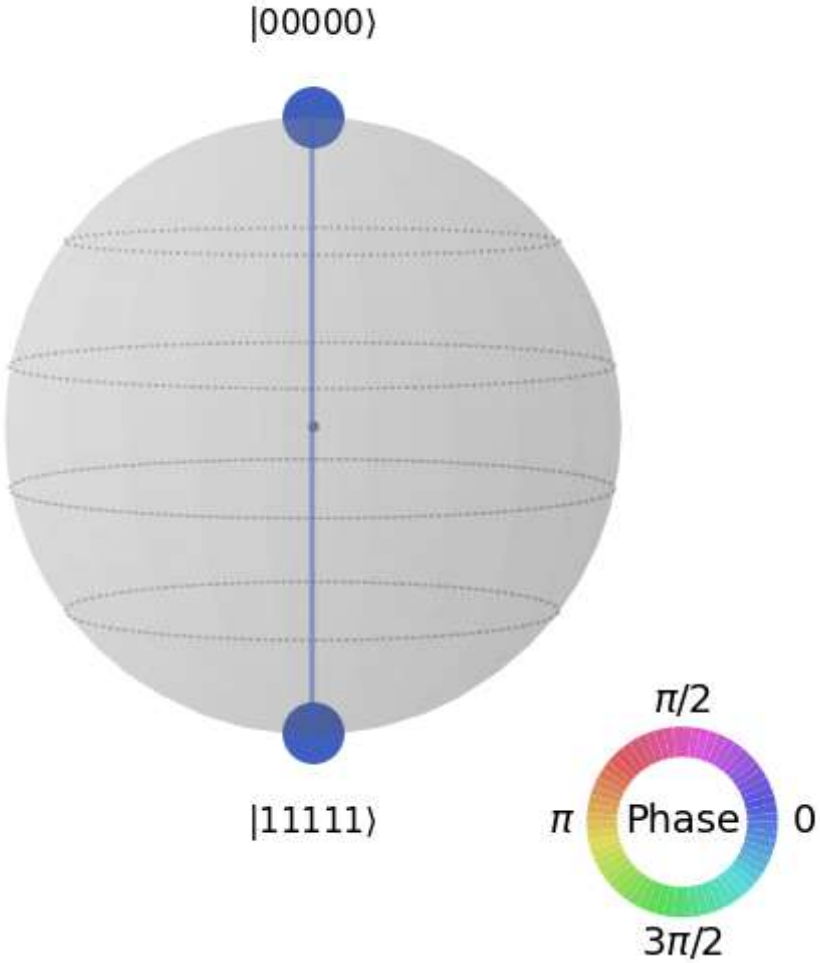

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

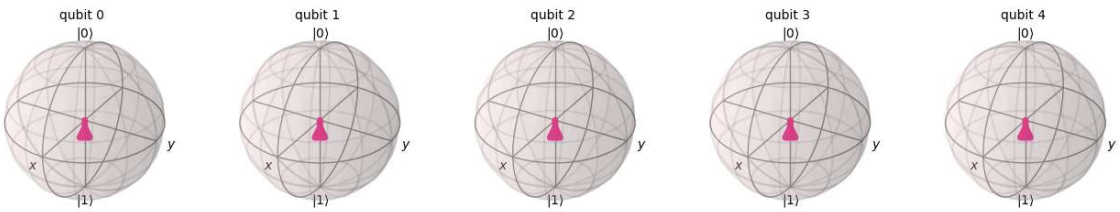

Im[ρ]
0

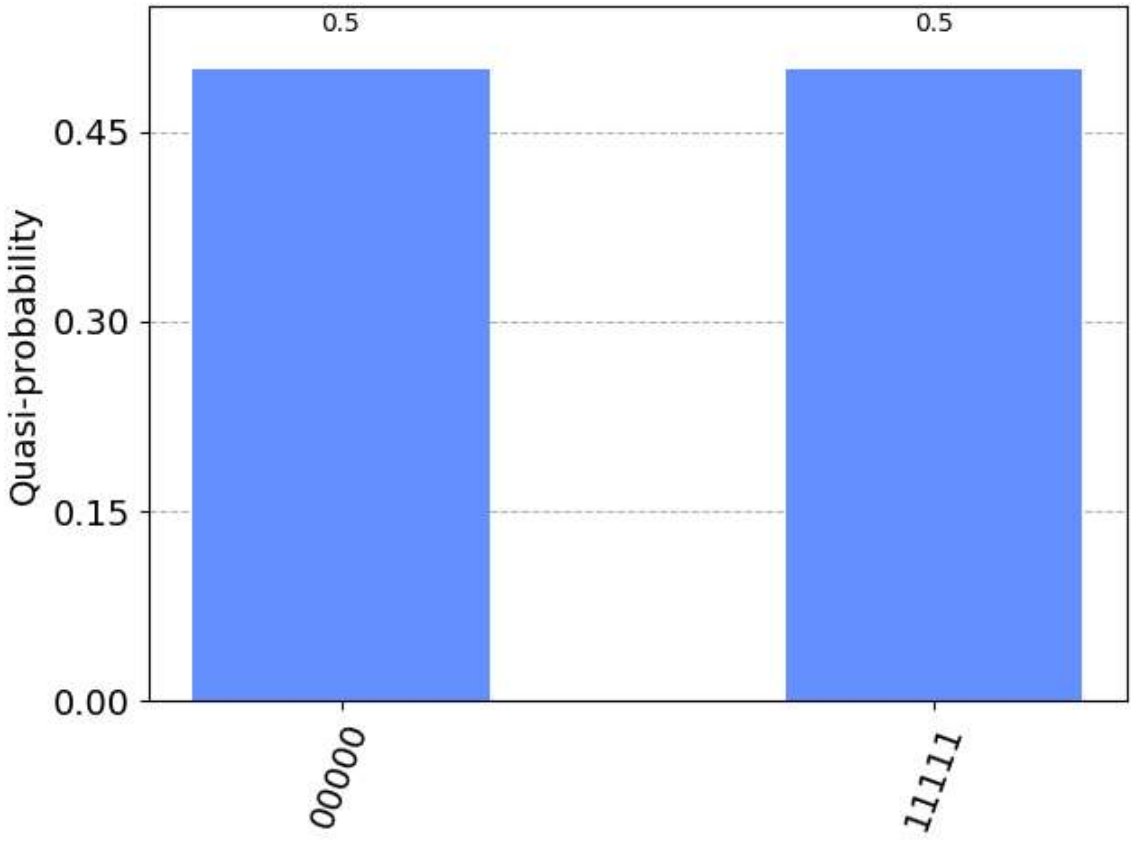

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

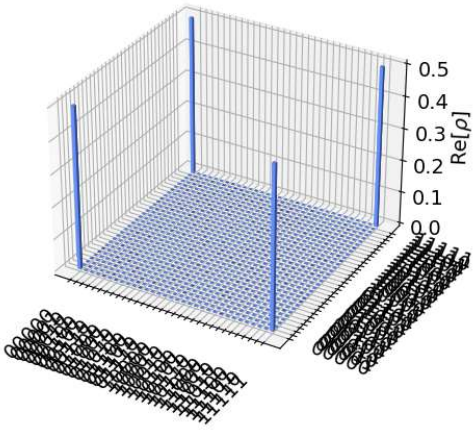

Re[ρ]
Im[ρ]

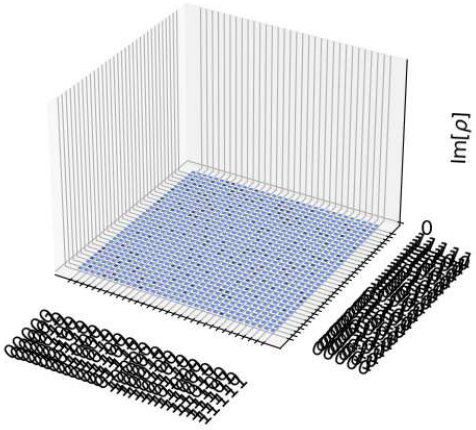

statevector
H
q_0
q_1
q_2
q_3
q_4

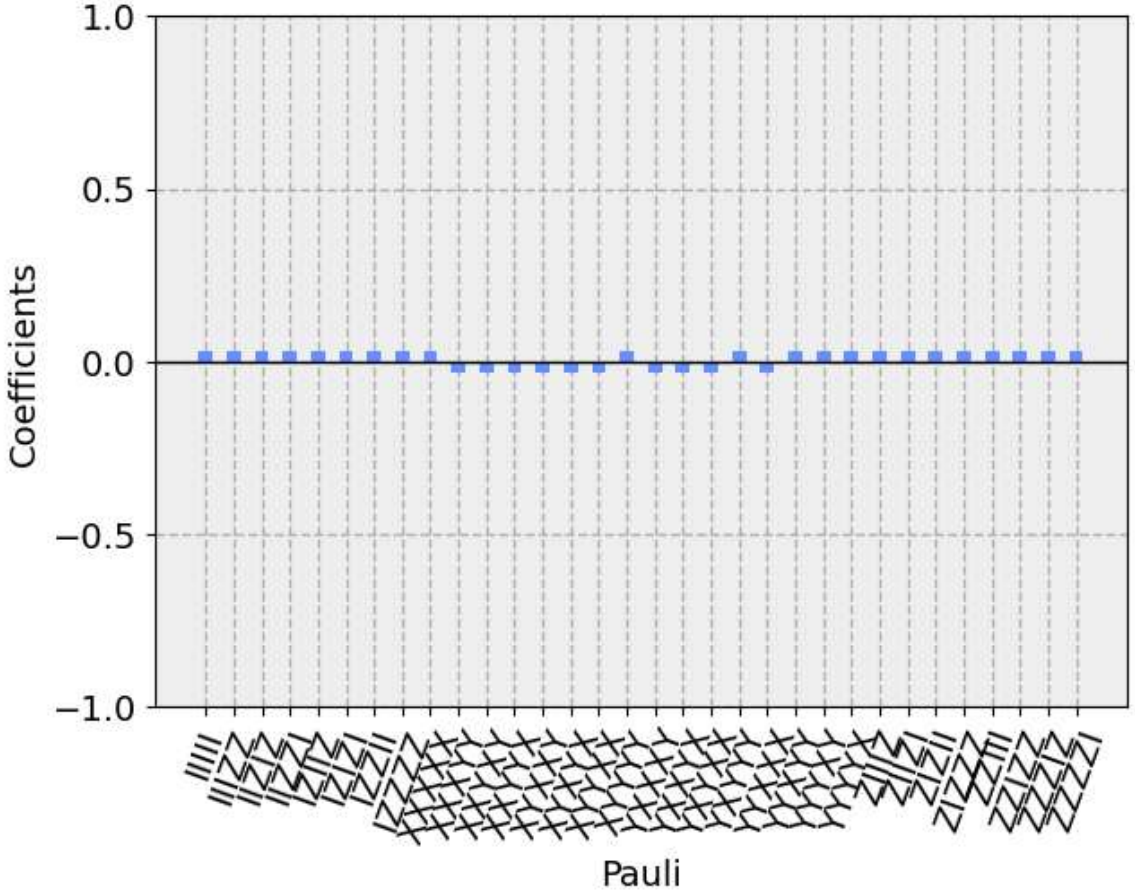

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

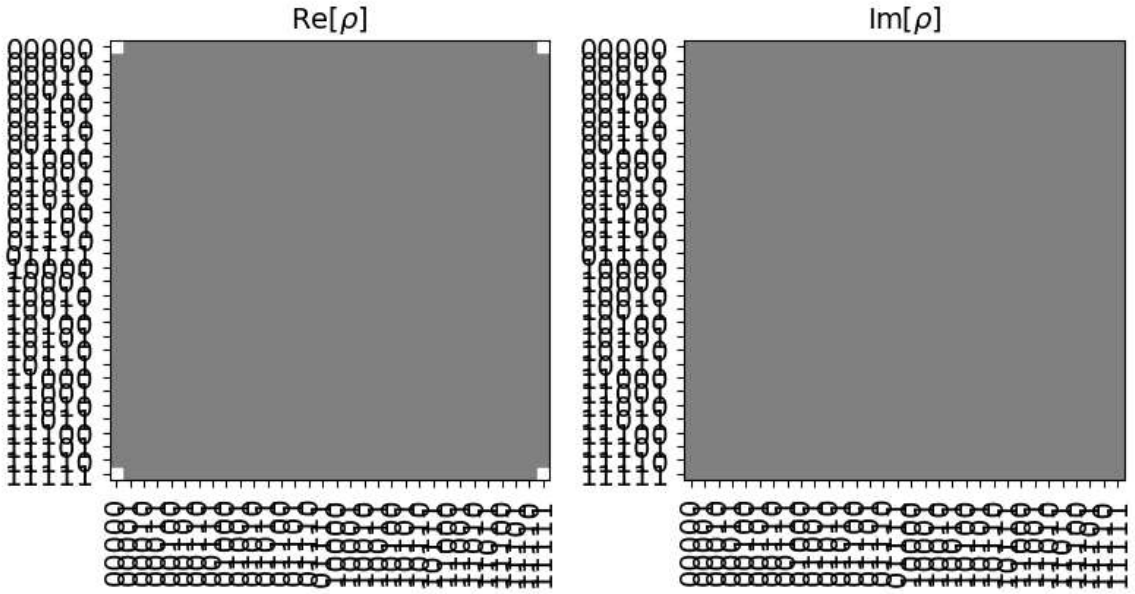

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

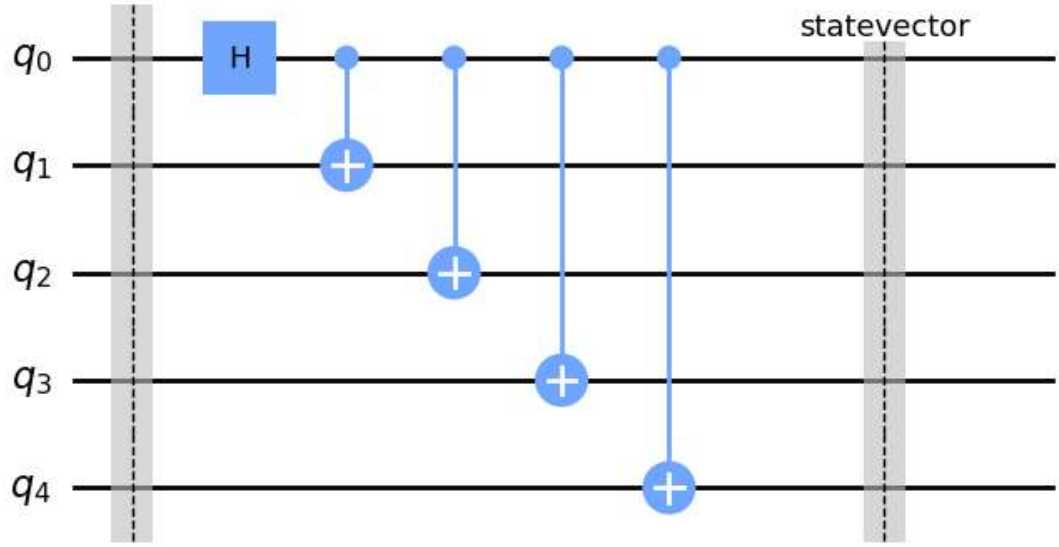

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

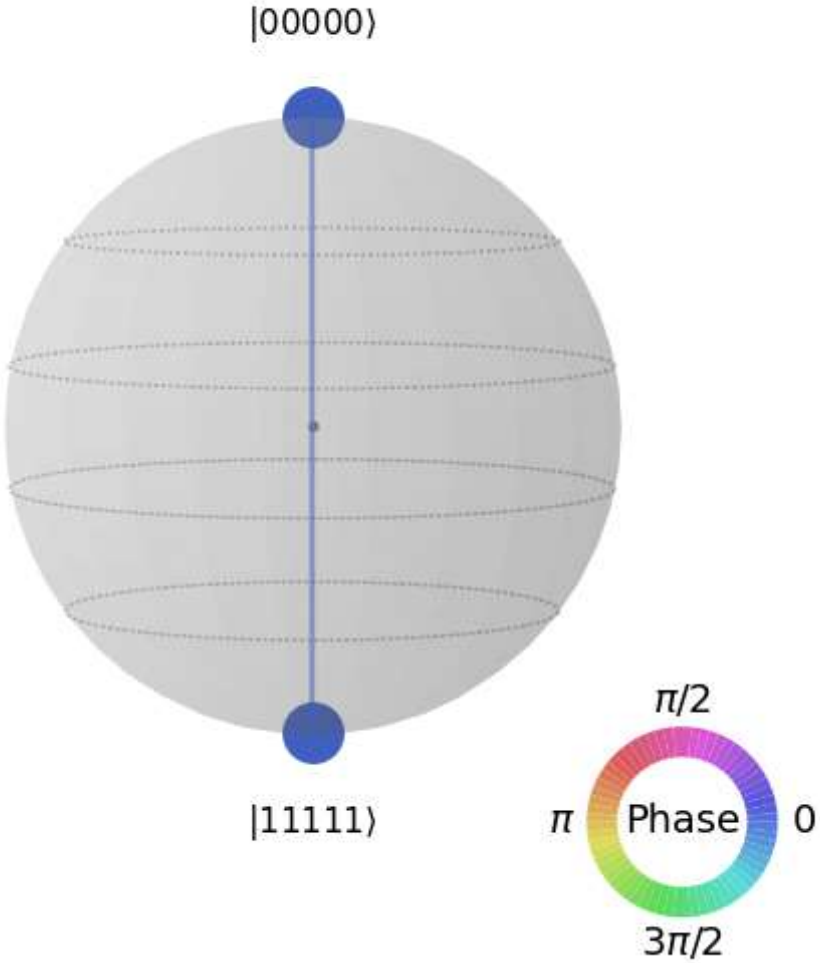

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

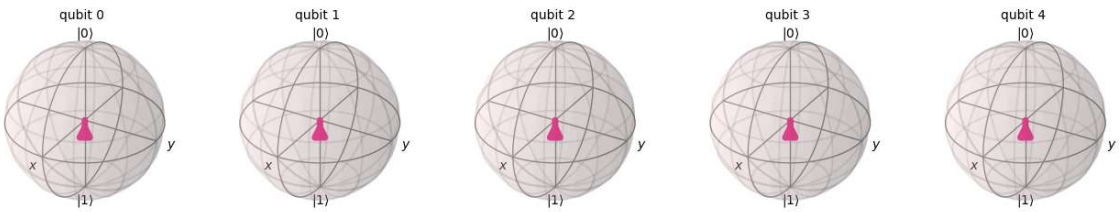

Im[ρ]
0

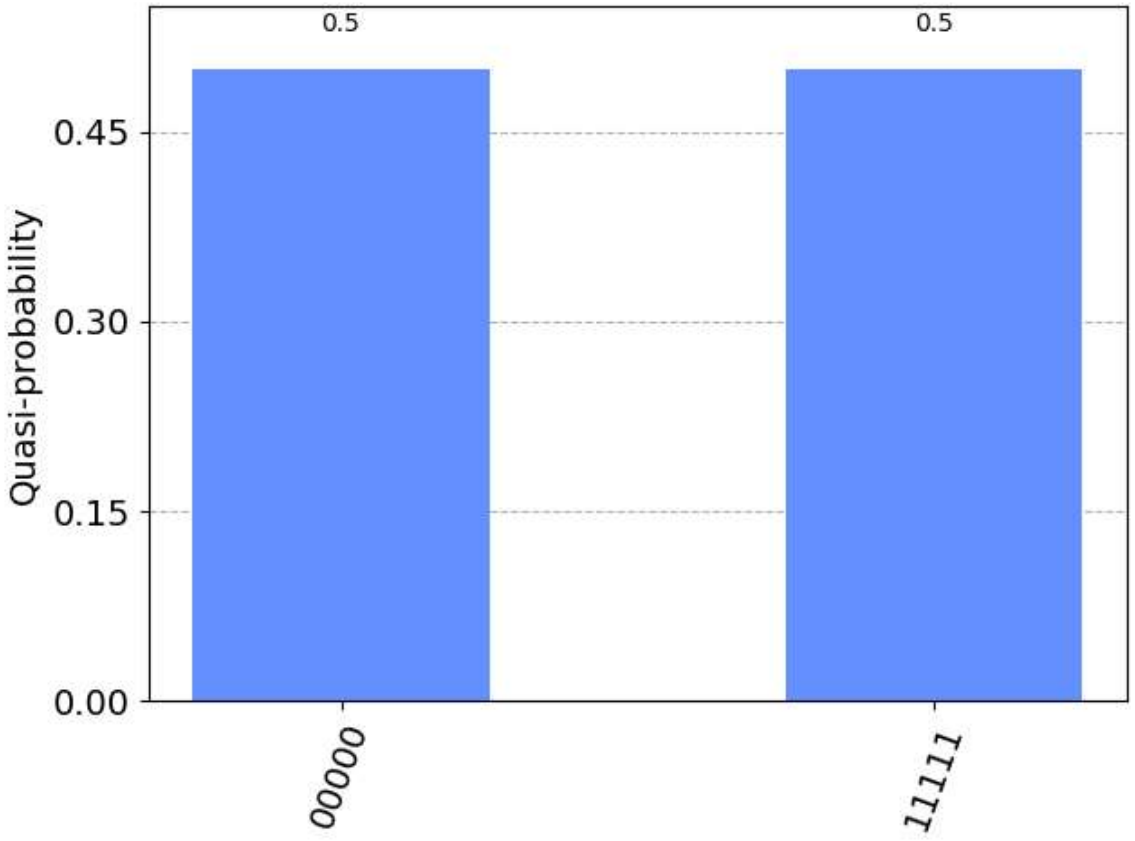

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

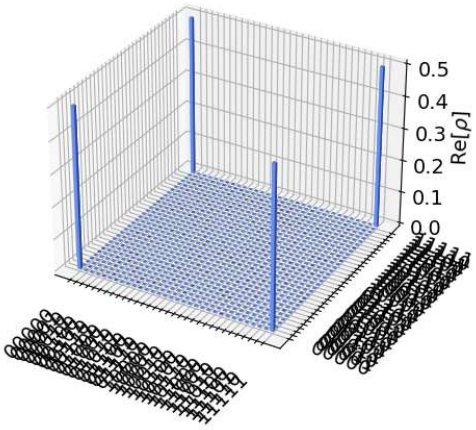

Re[ρ]
Im[ρ]

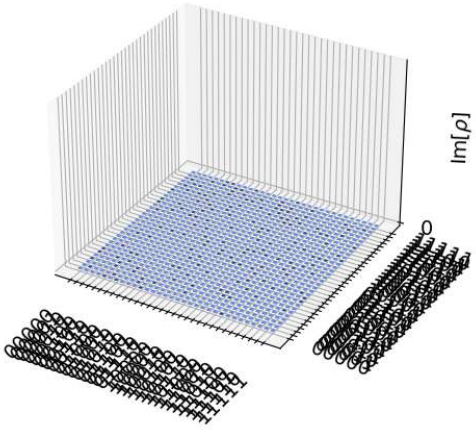

statevector
H
q_0
q_1
q_2
q_3
q_4

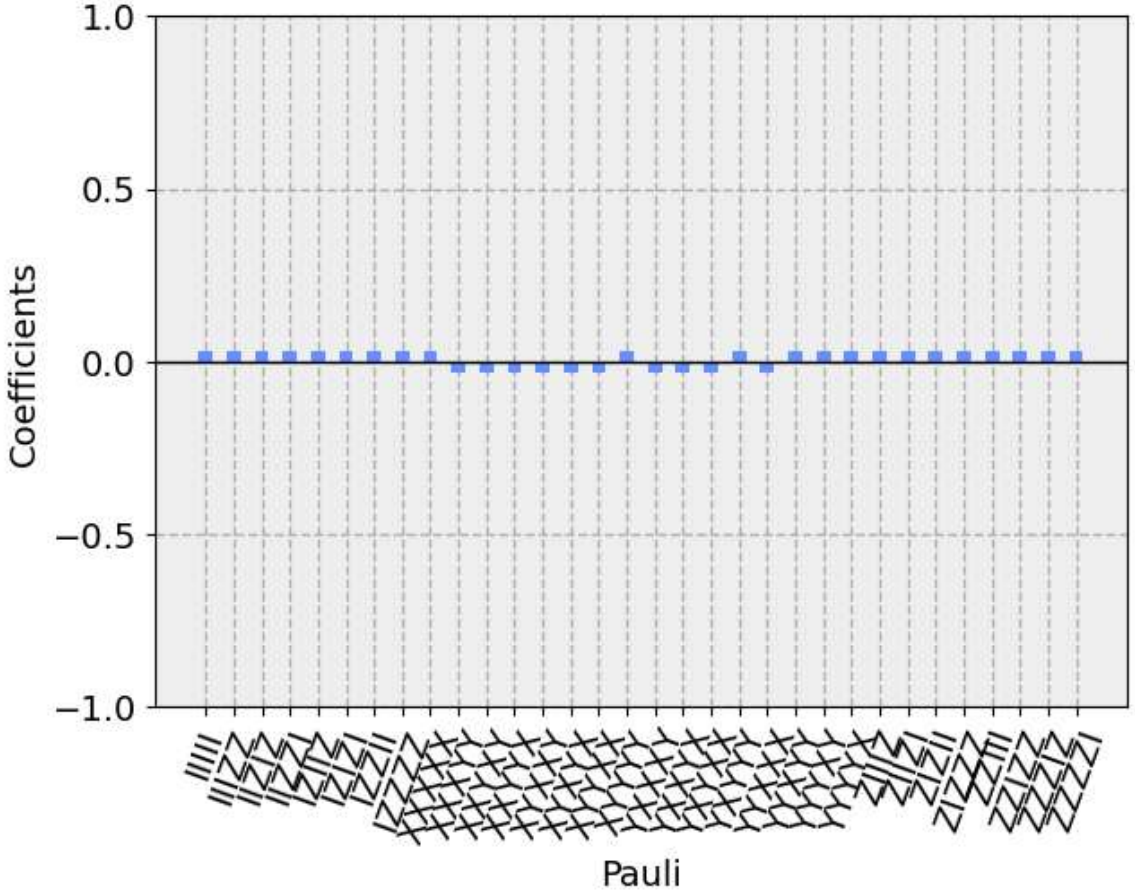

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

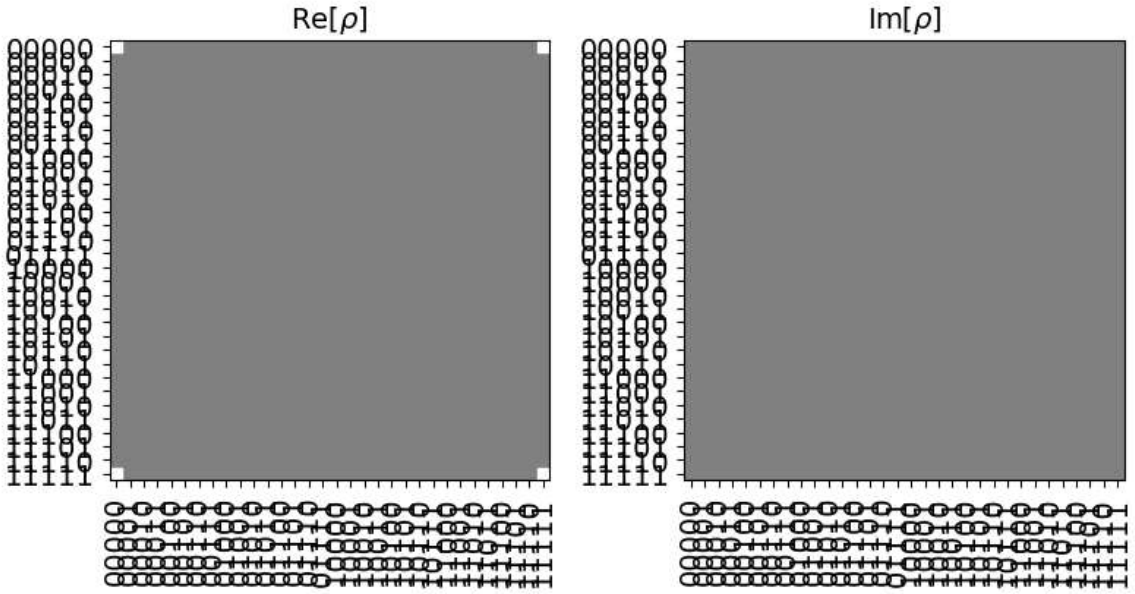

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

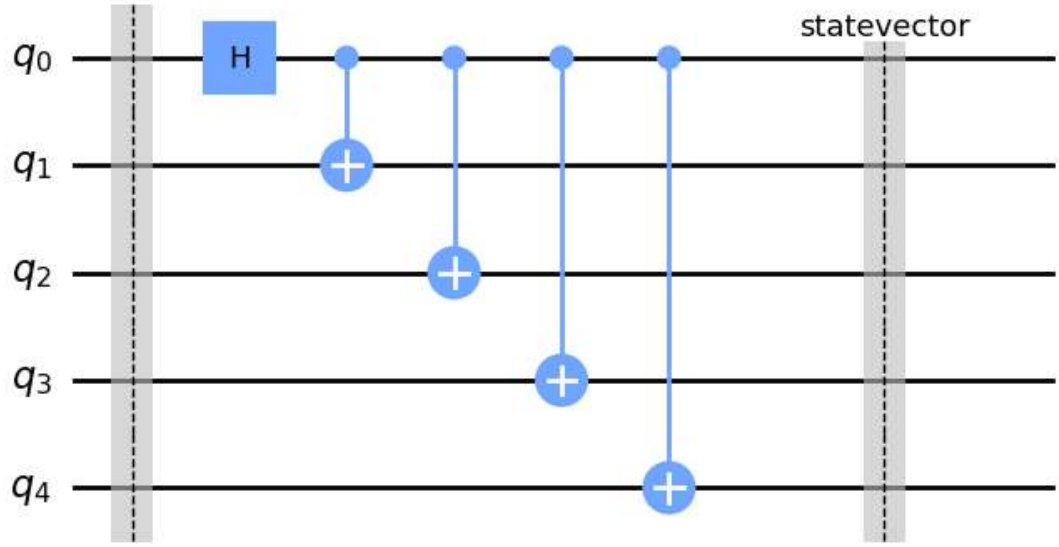

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

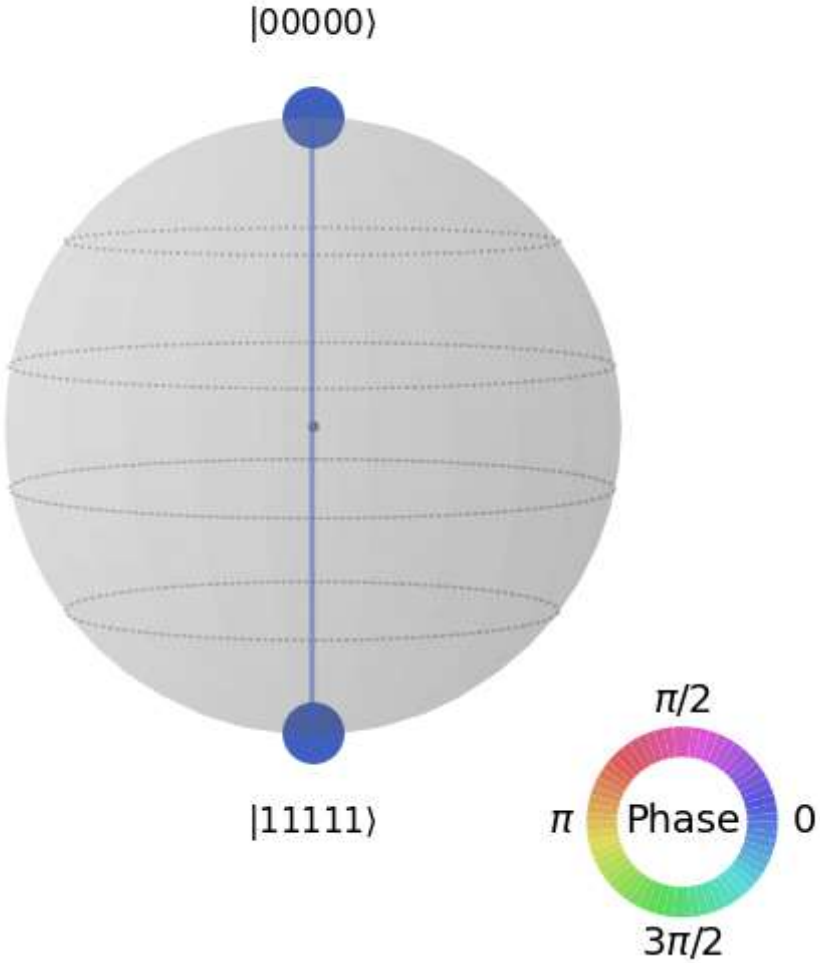

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

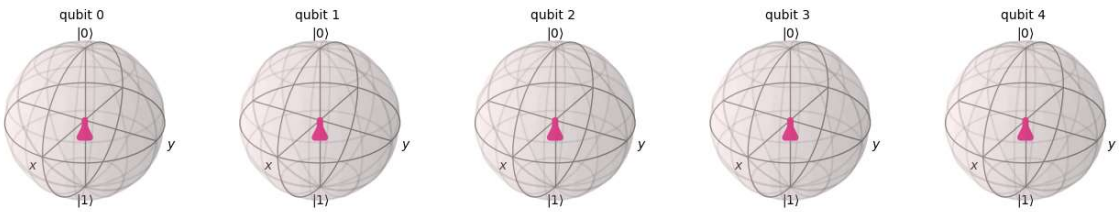

Im[ρ]
0

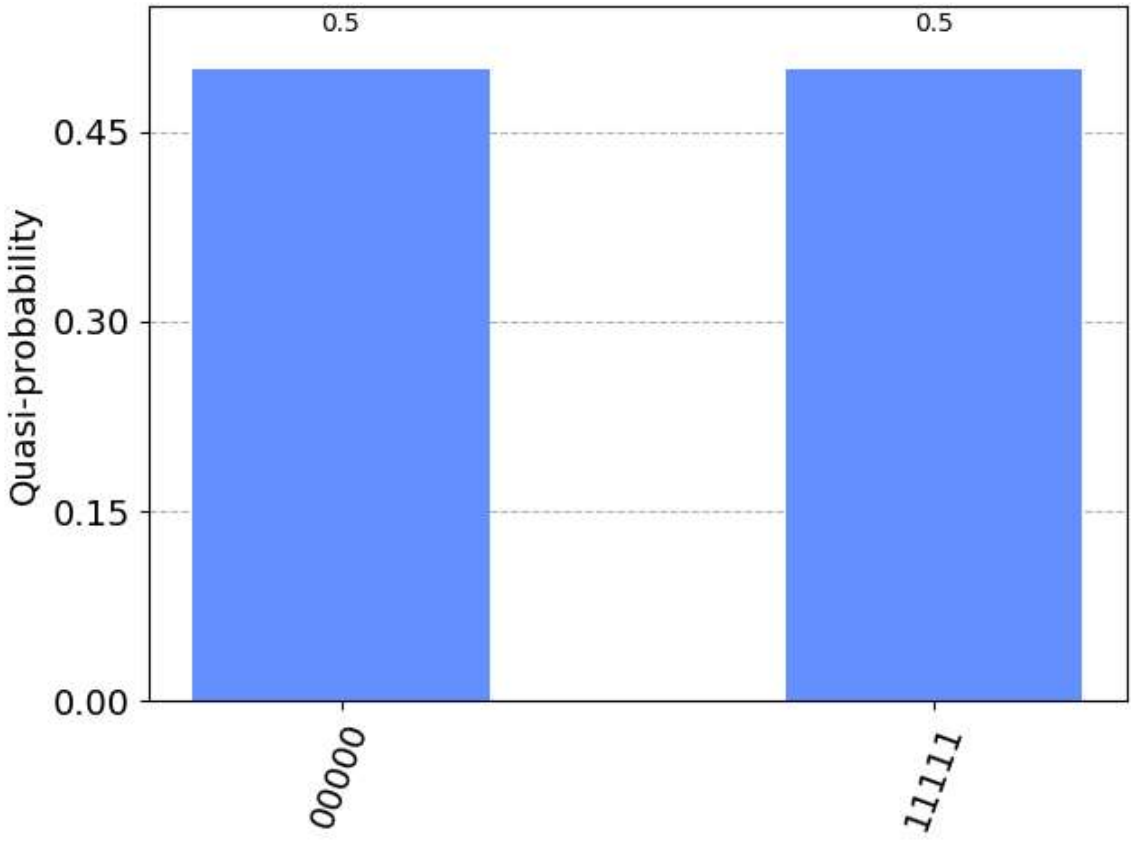

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

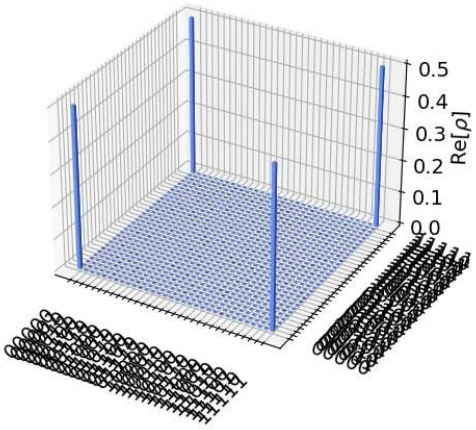

Re[ρ]
Im[ρ]

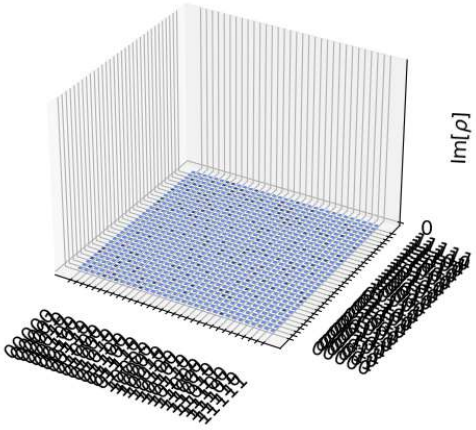

statevector
H
q_0
q_1
q_2
q_3
q_4

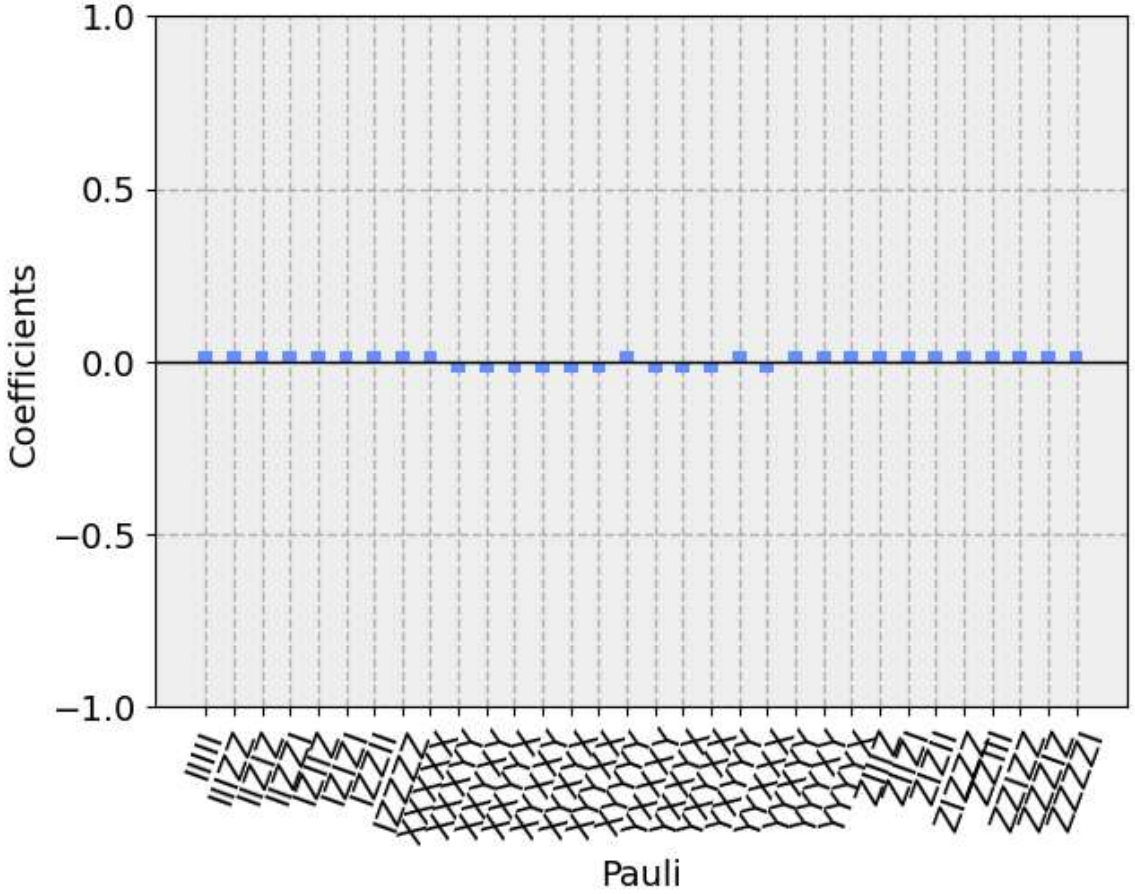

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

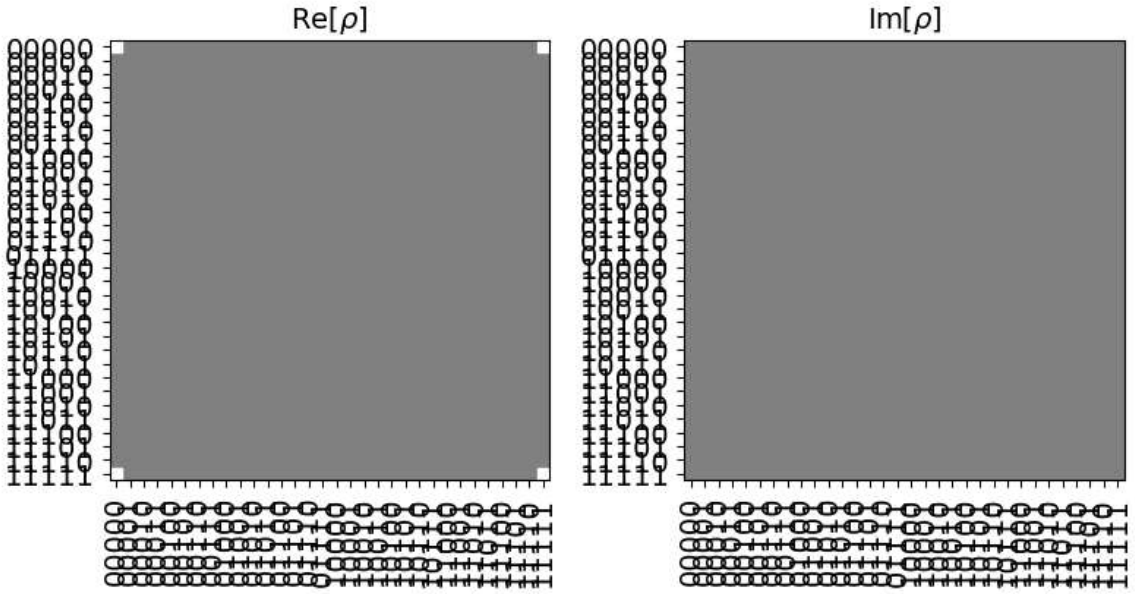

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

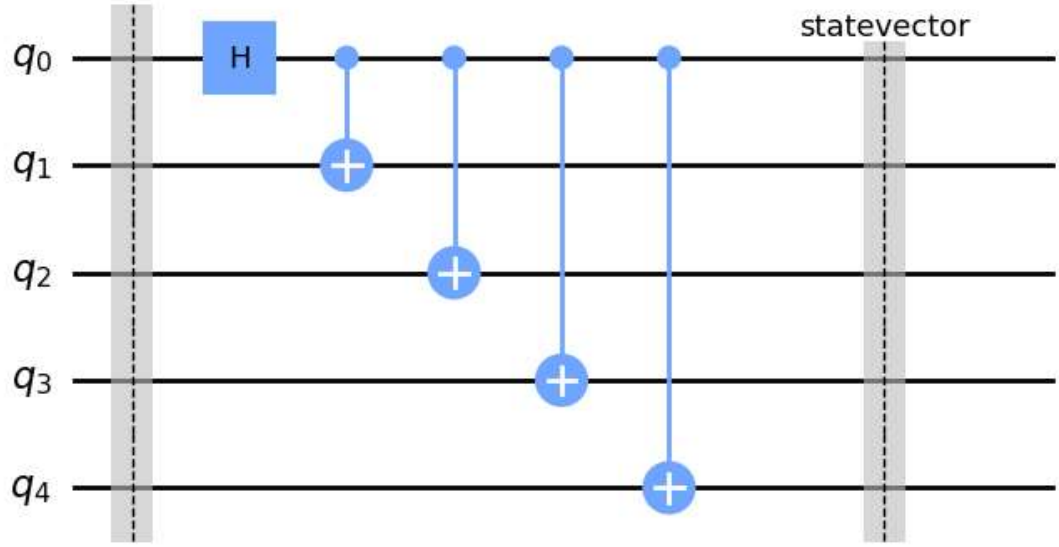

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

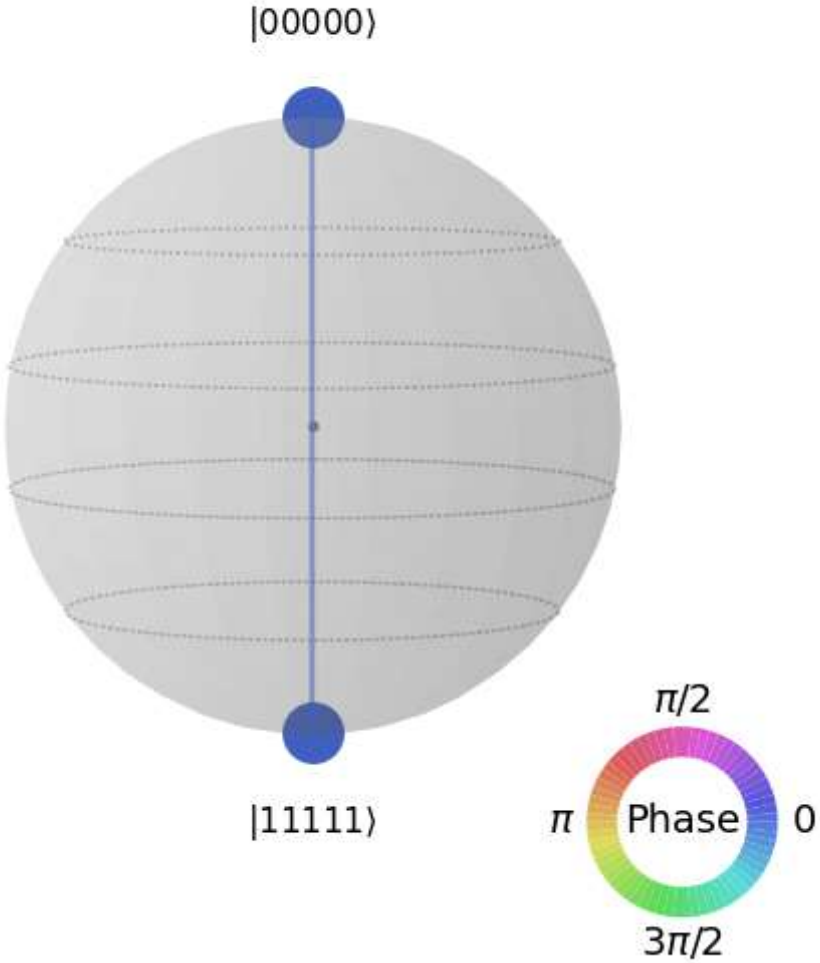

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

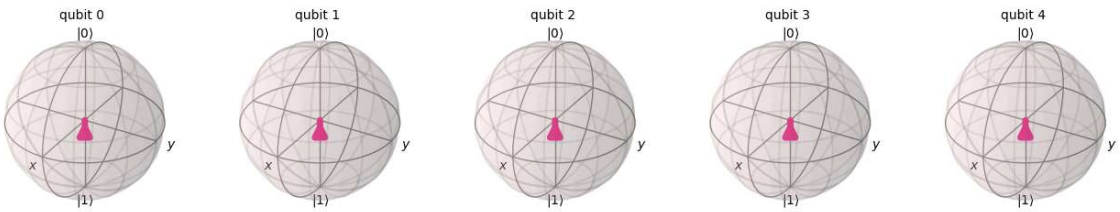

Im[ρ]
0

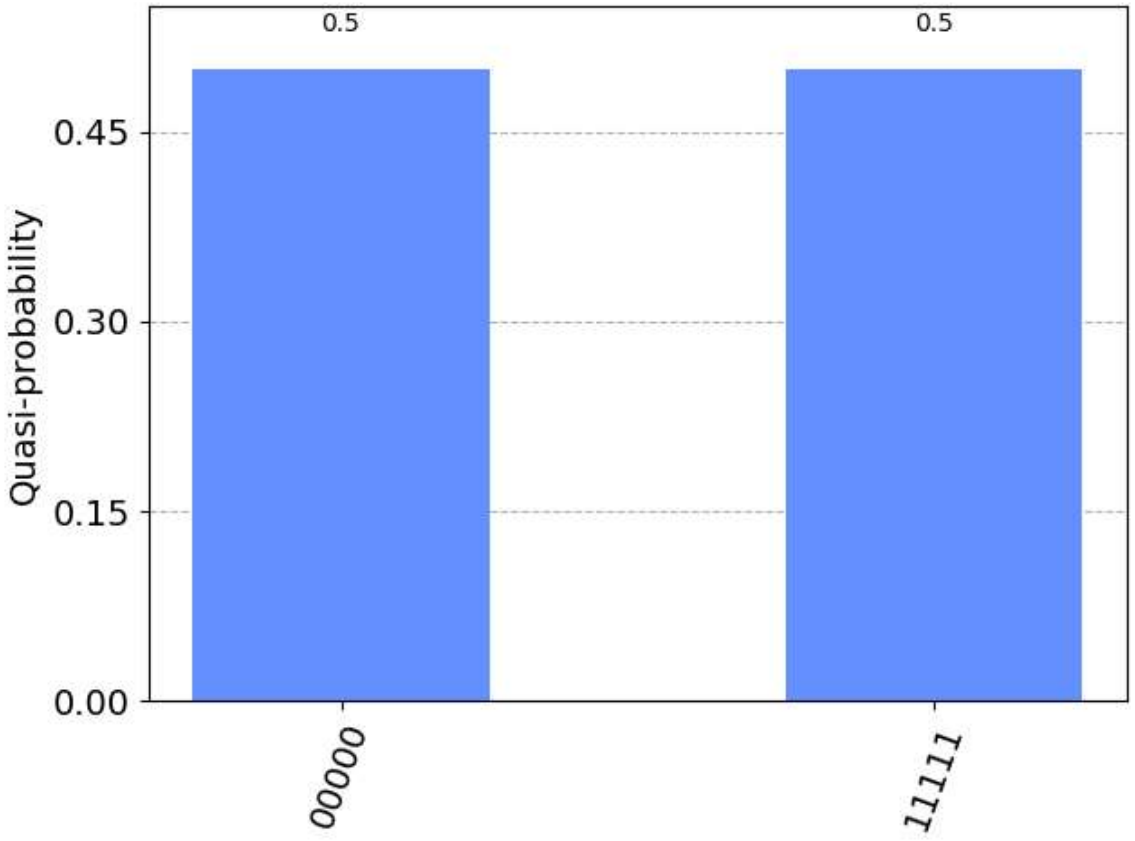

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

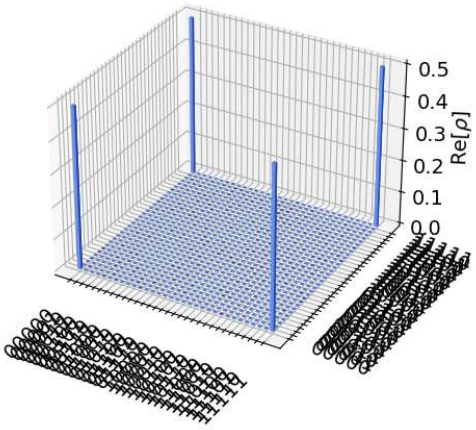

Re[ρ]
Im[ρ]

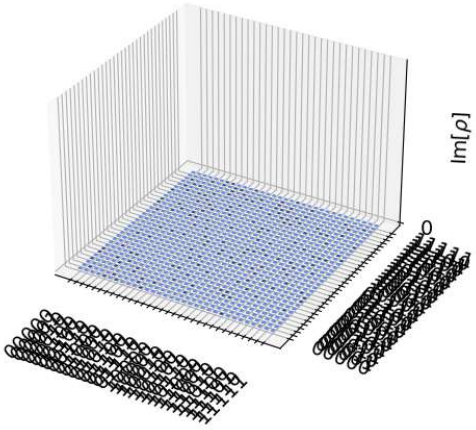

statevector
H
q_0
q_1
q_2
q_3
q_4

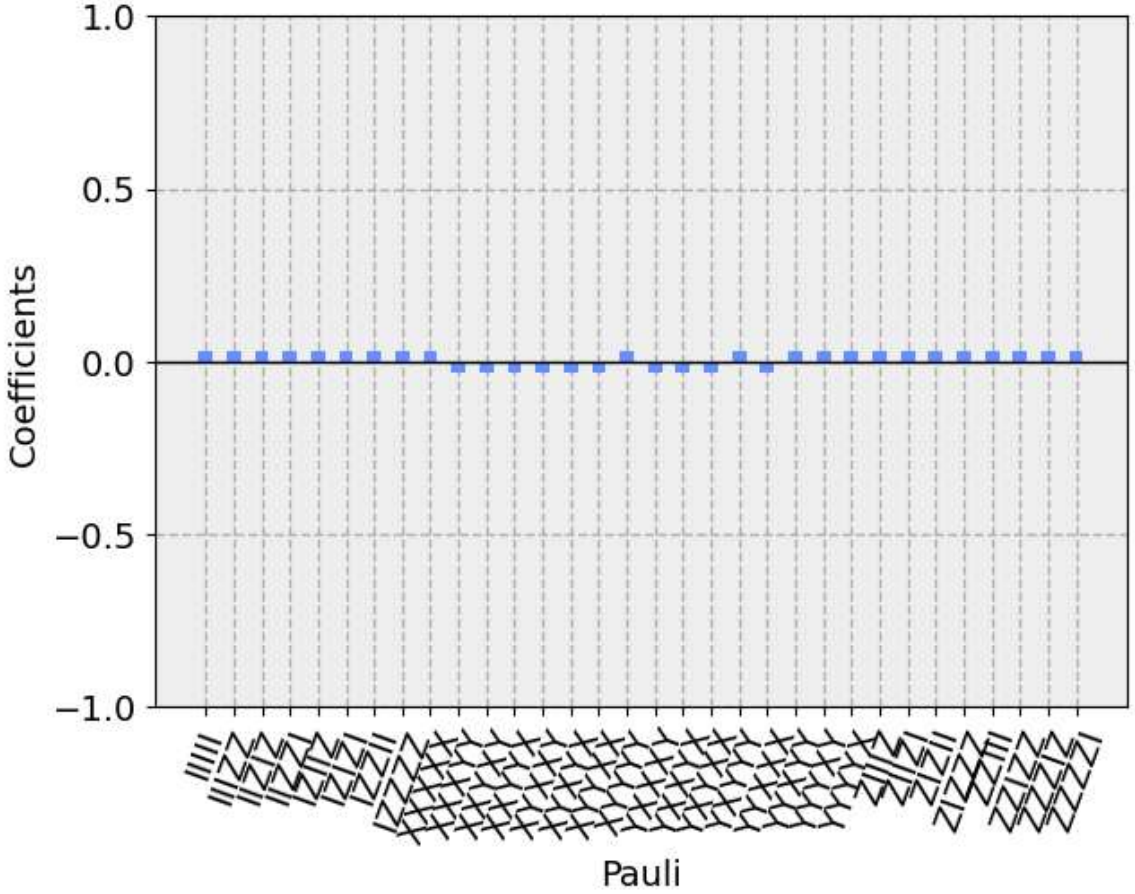

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

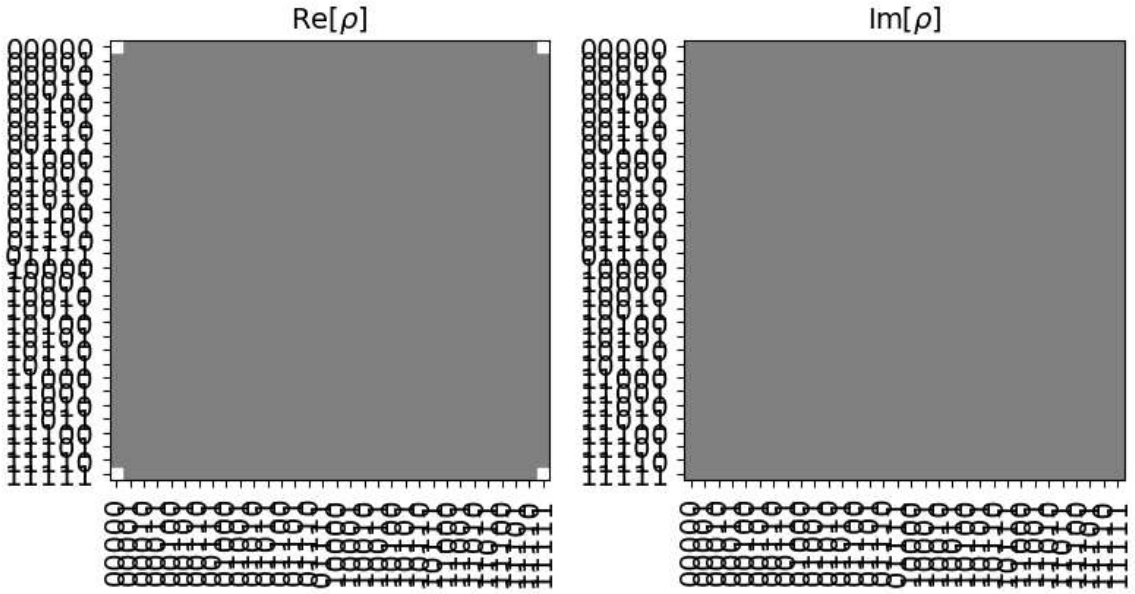

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

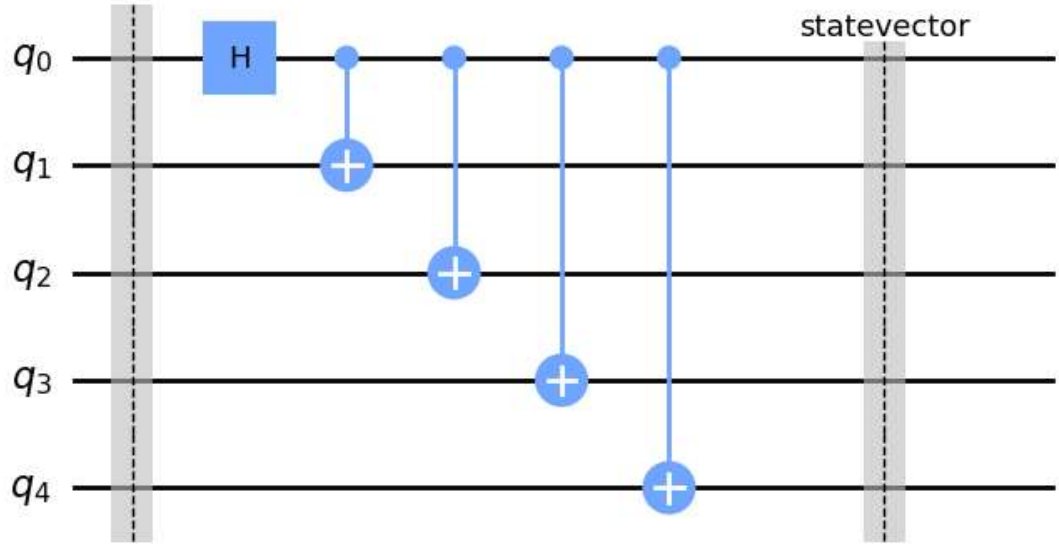

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

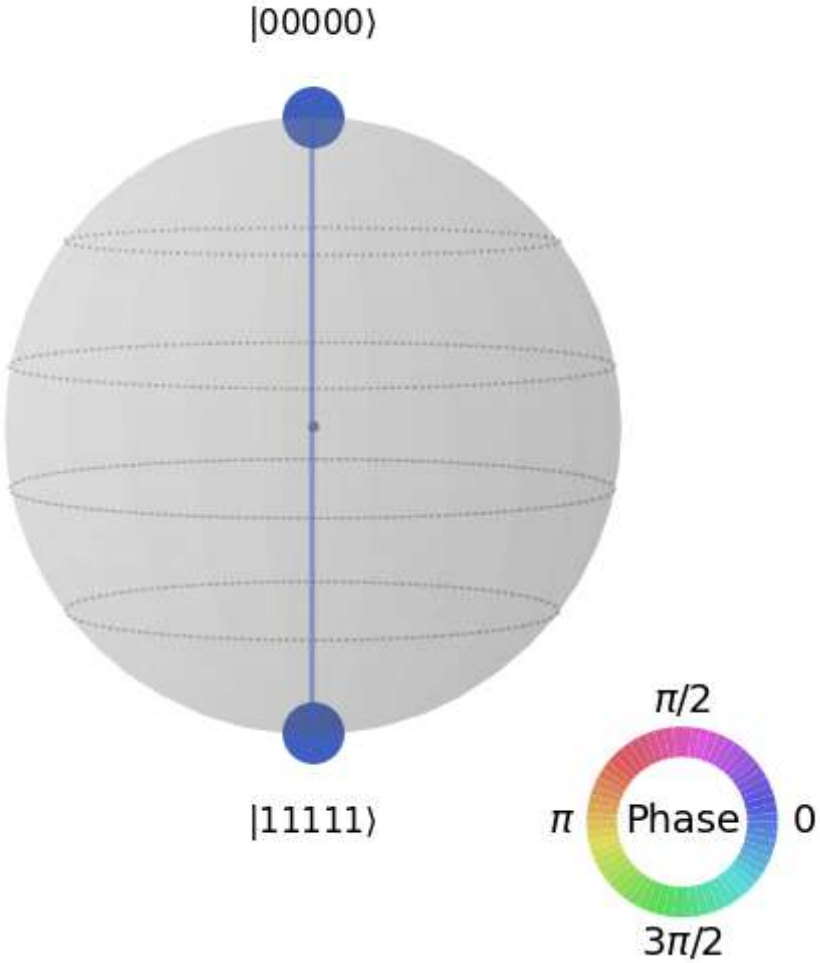

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

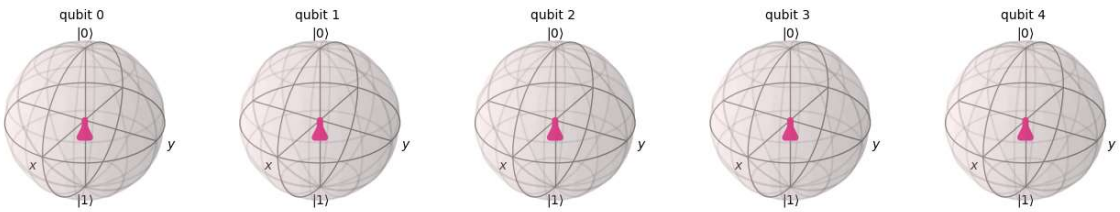

Im[ρ]
0

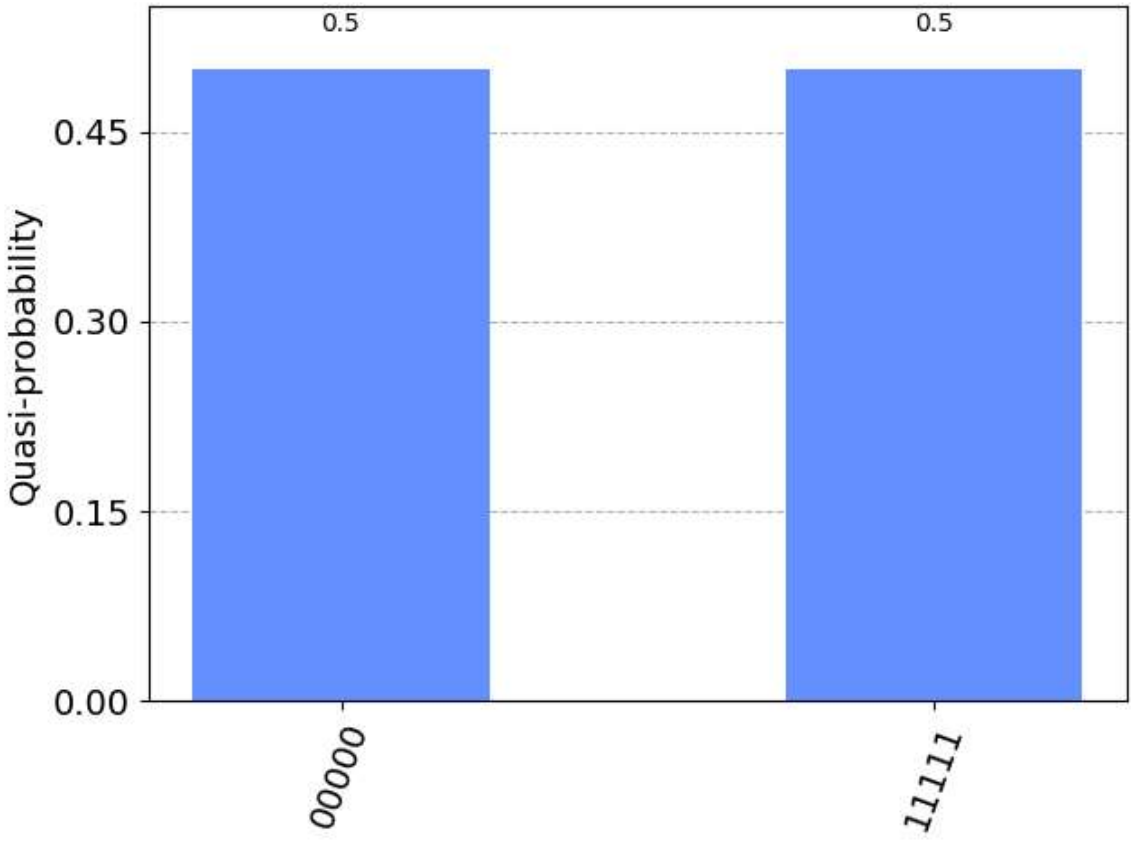

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

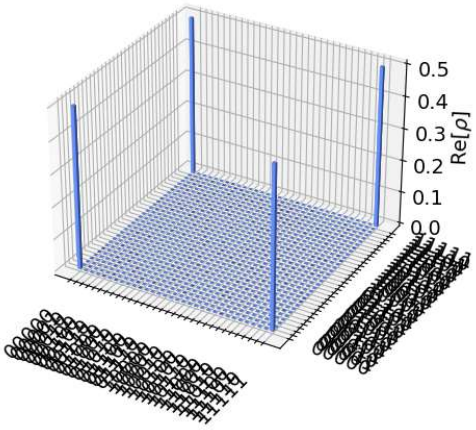

Re[ρ]
Im[ρ]

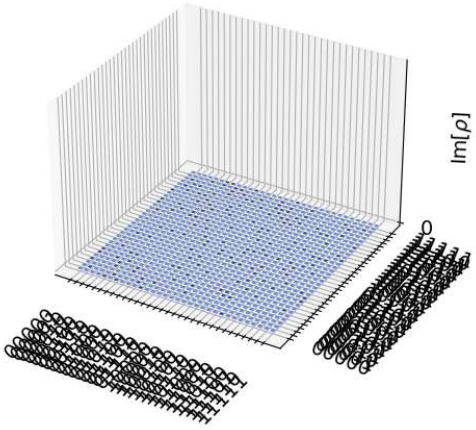

statevector
H
q_0
q_1
q_2
q_3
q_4

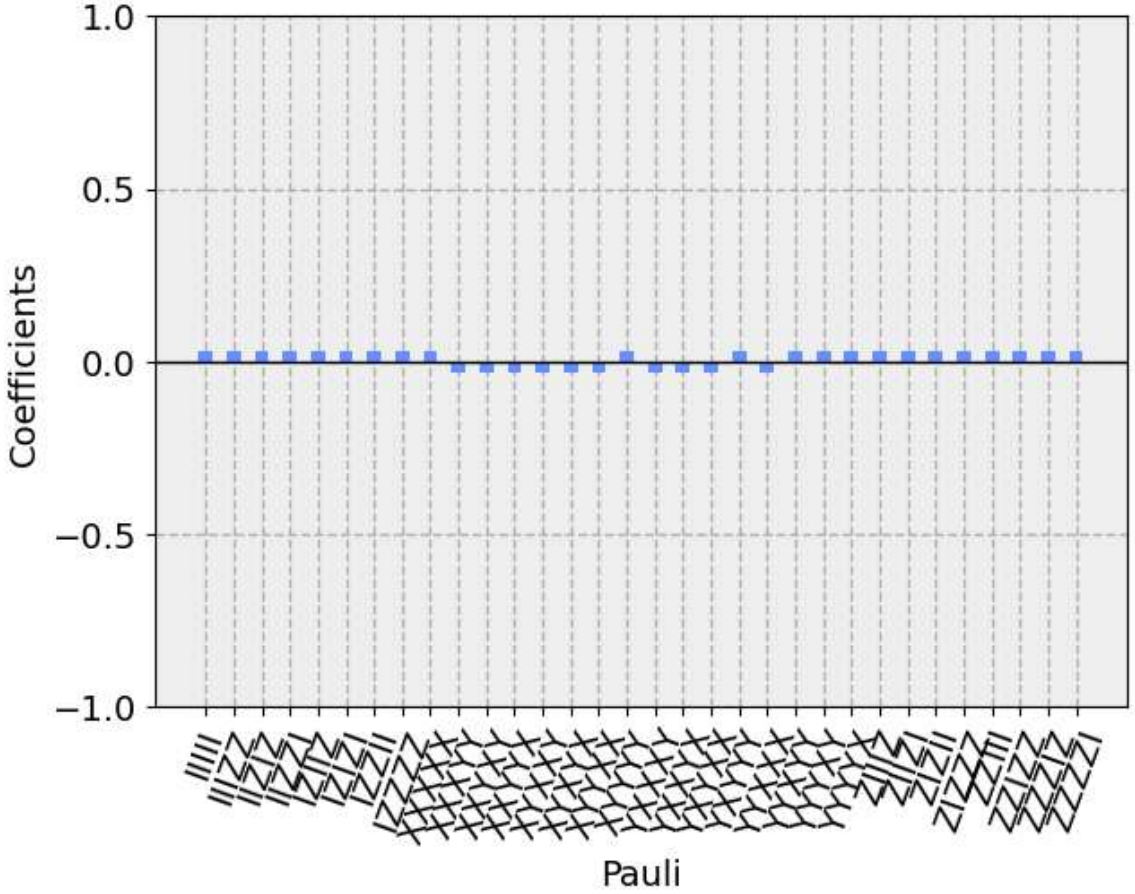

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

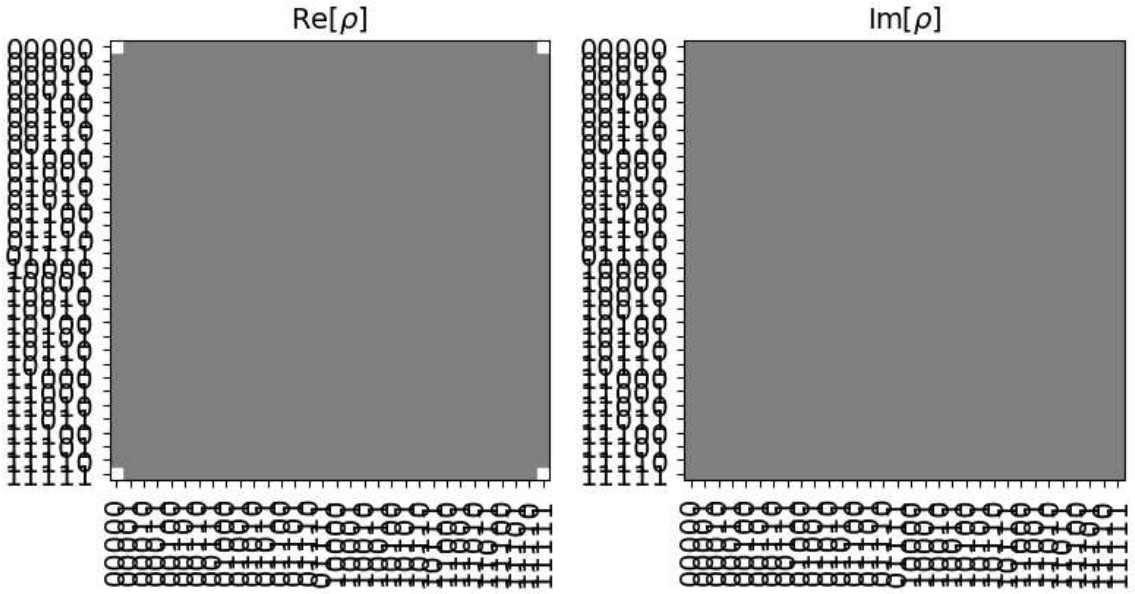

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

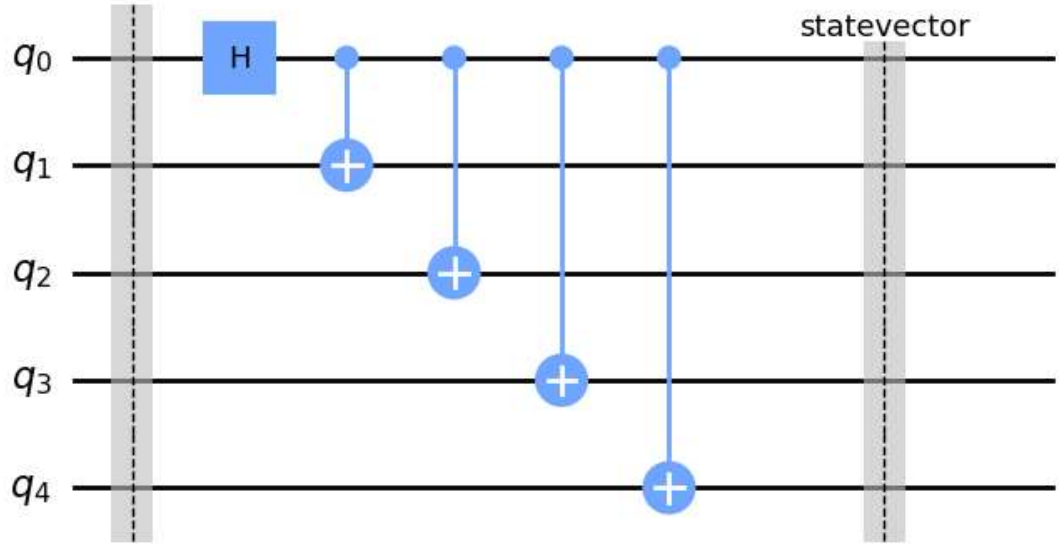

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

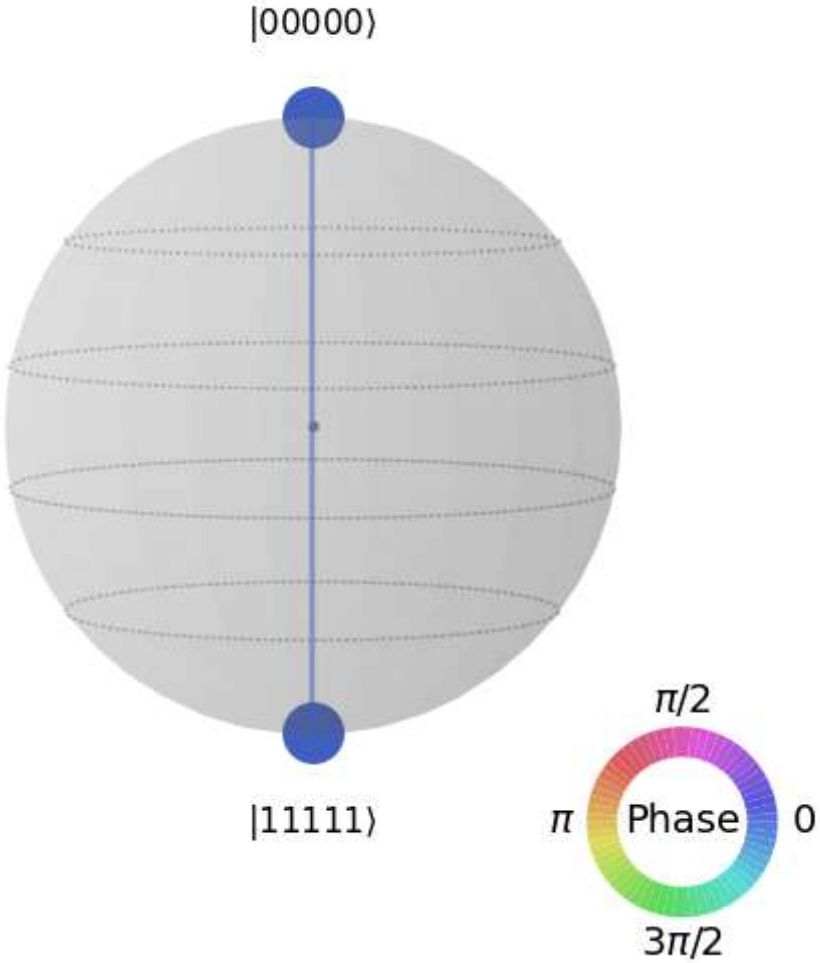

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

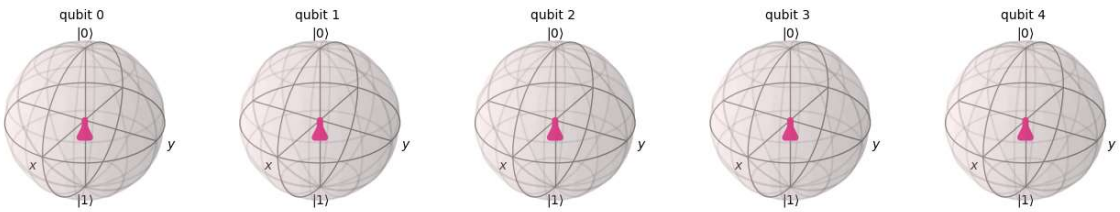

Im[ρ]
0

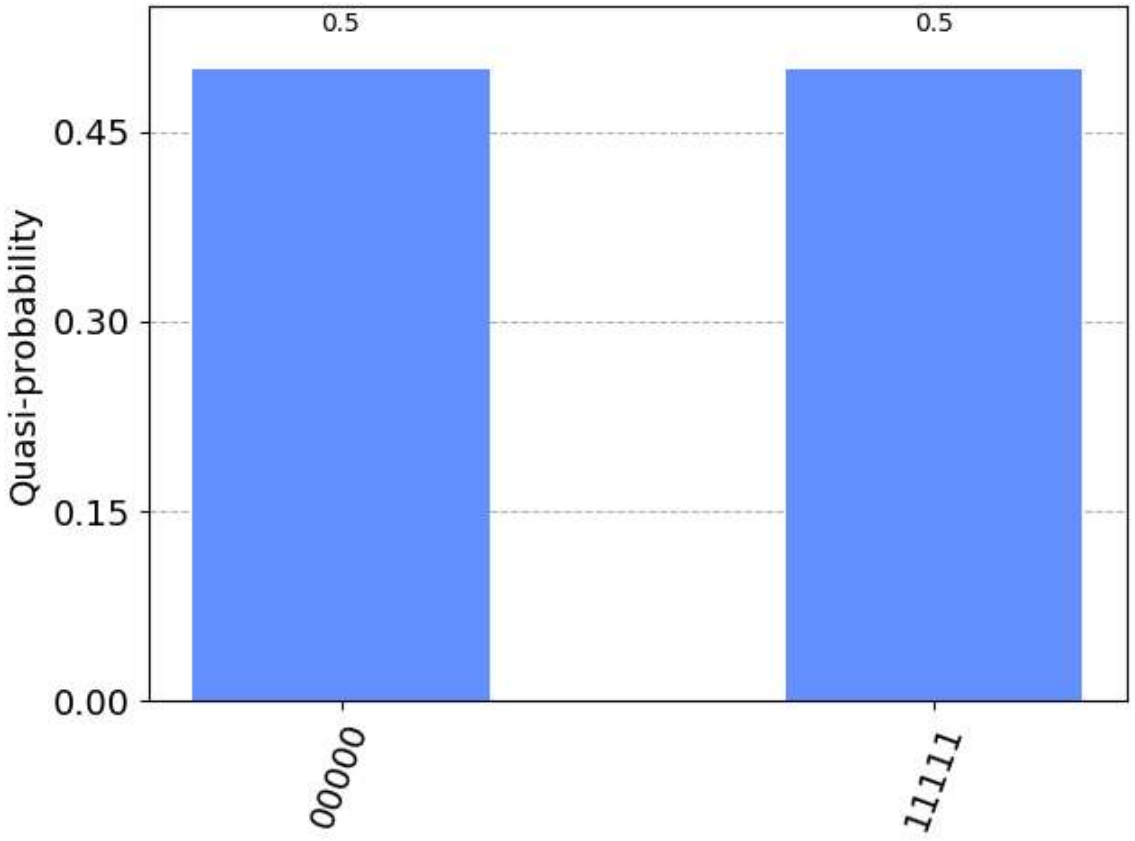

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

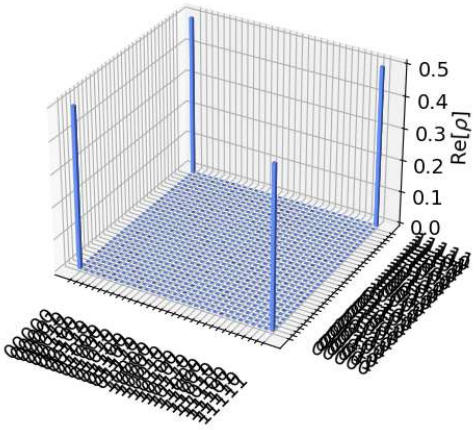

Re[ρ]
Im[ρ]

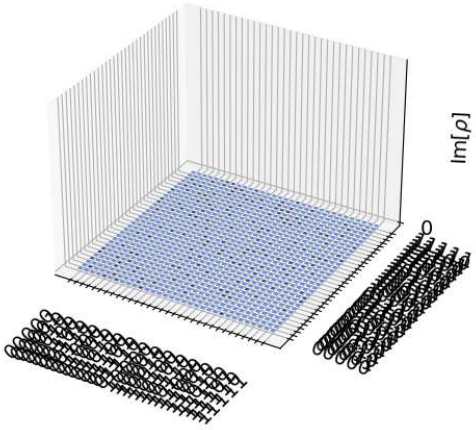

statevector
H
q_0
q_1
q_2
q_3
q_4

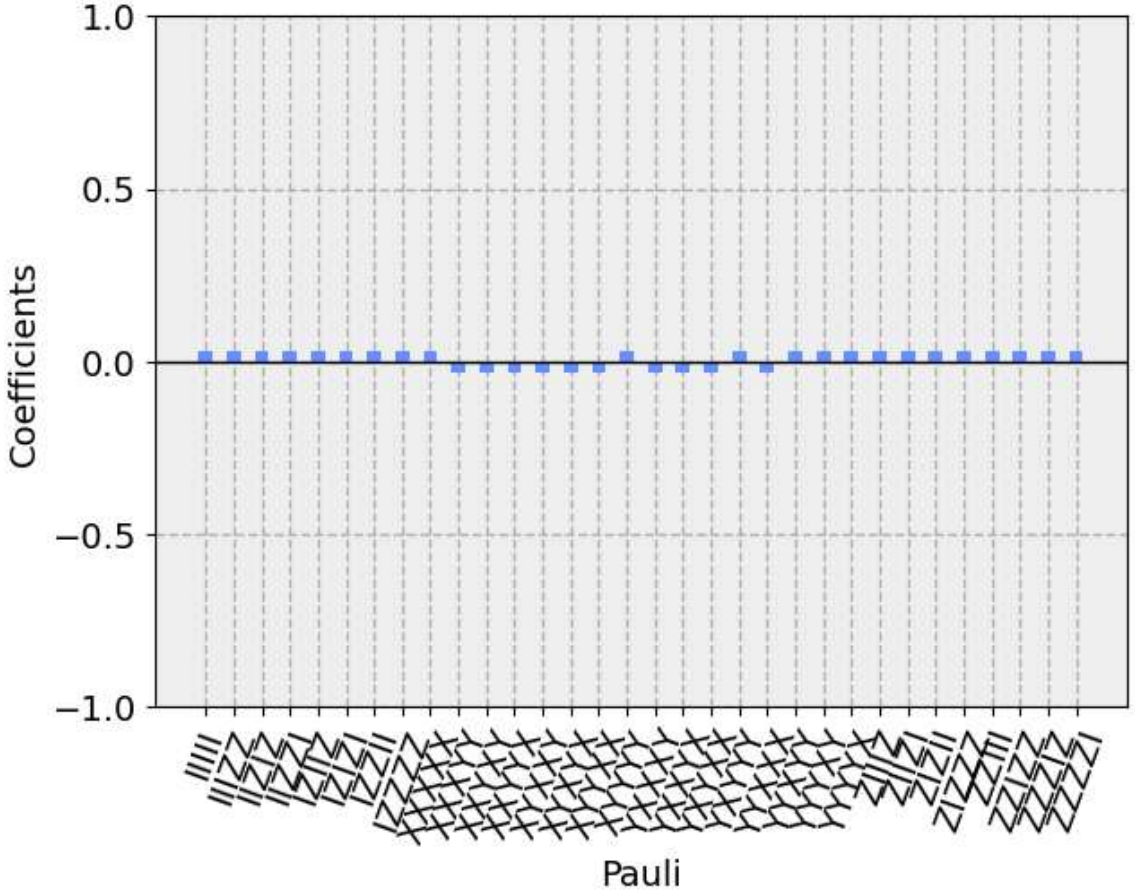

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

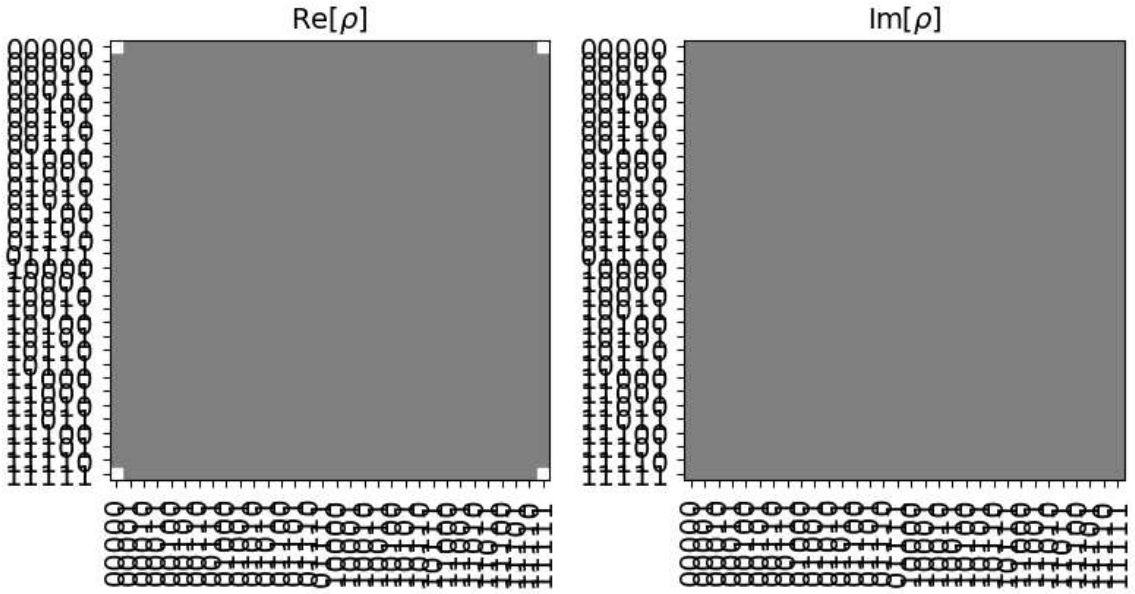

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

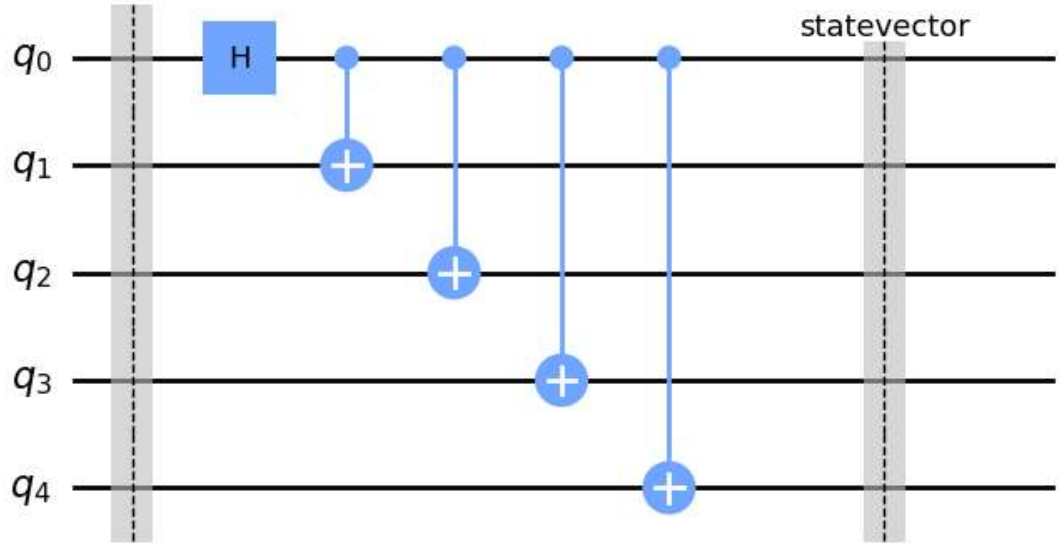

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

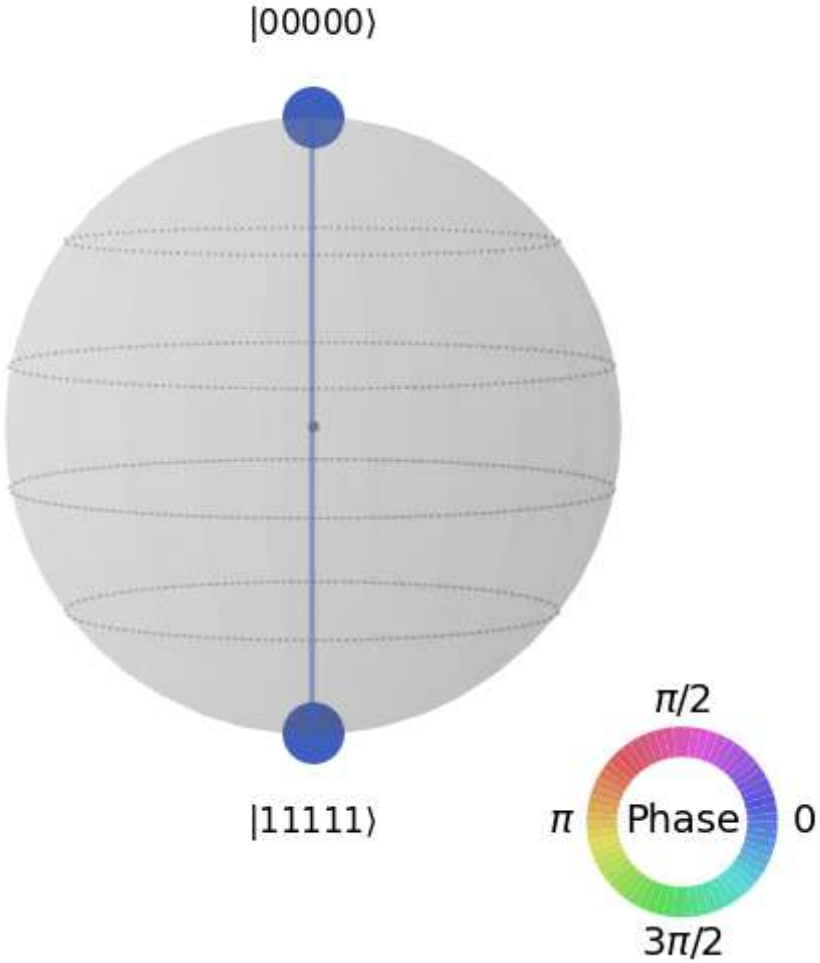

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

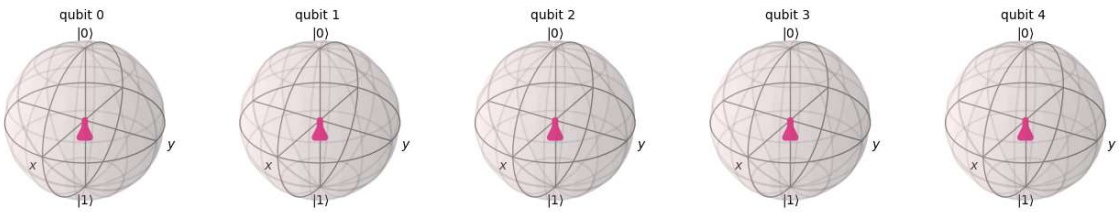

Im[ρ]
0

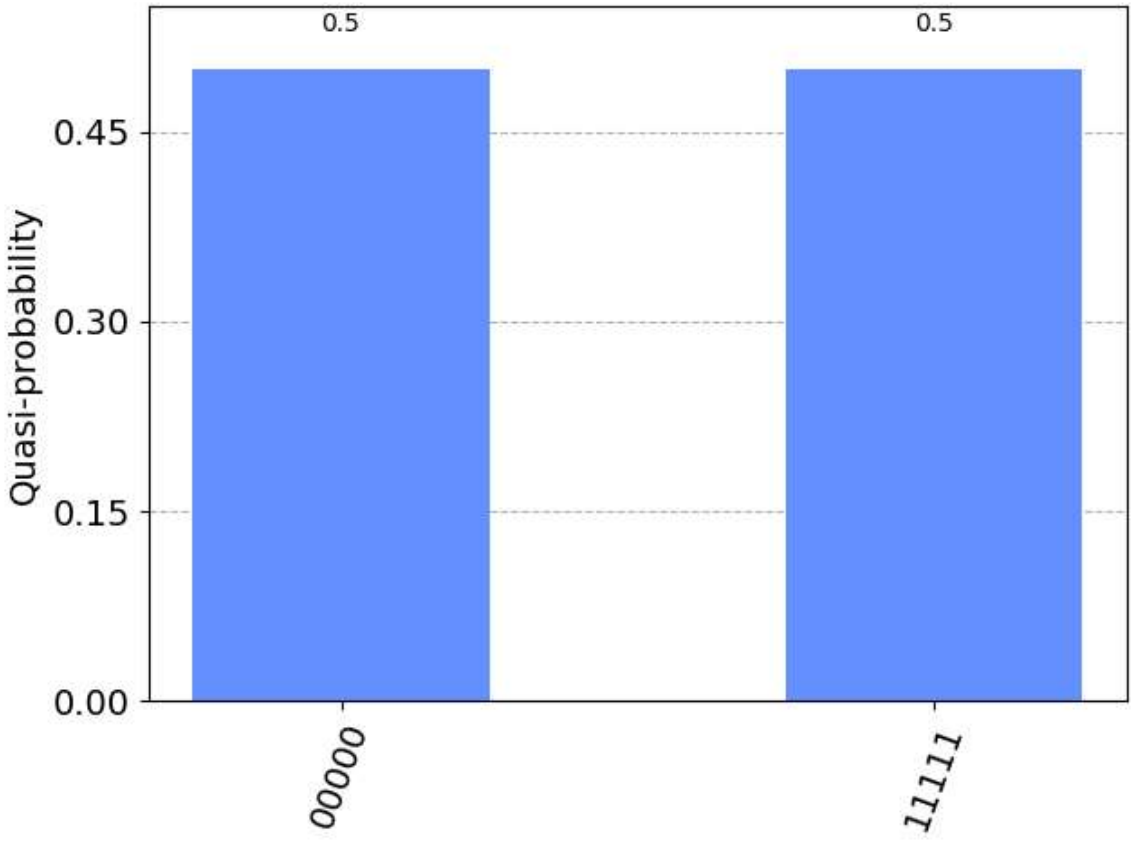

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

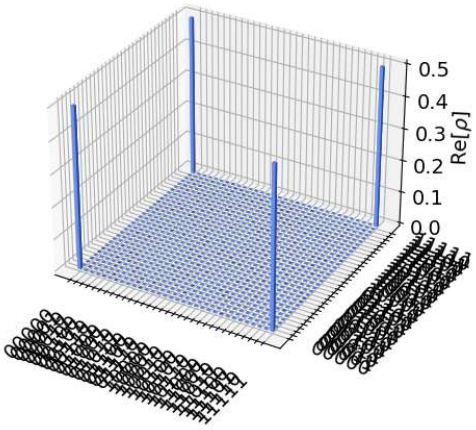

Re[ρ]
Im[ρ]

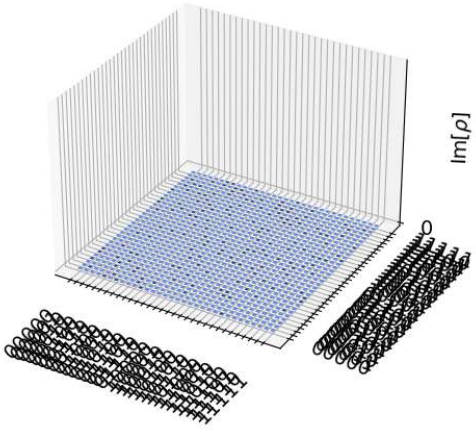

statevector
H
q_0
q_1
q_2
q_3
q_4

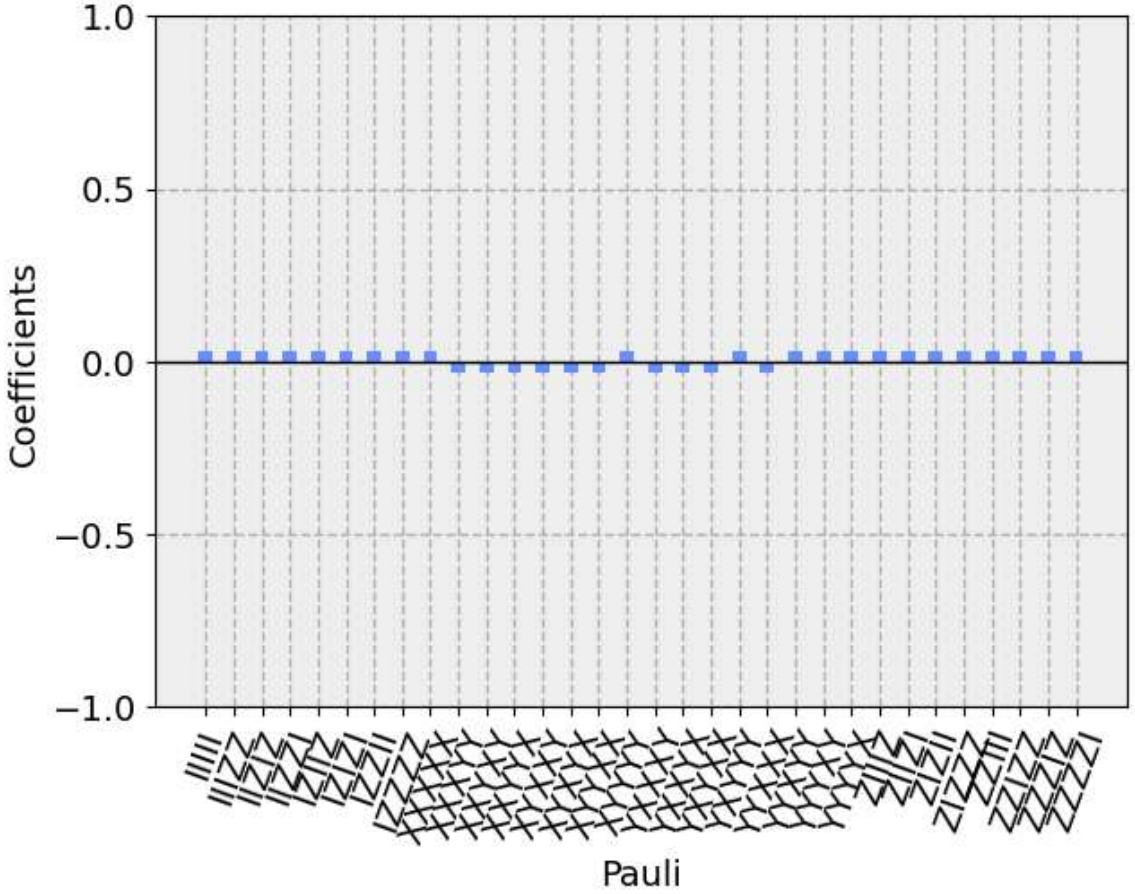

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

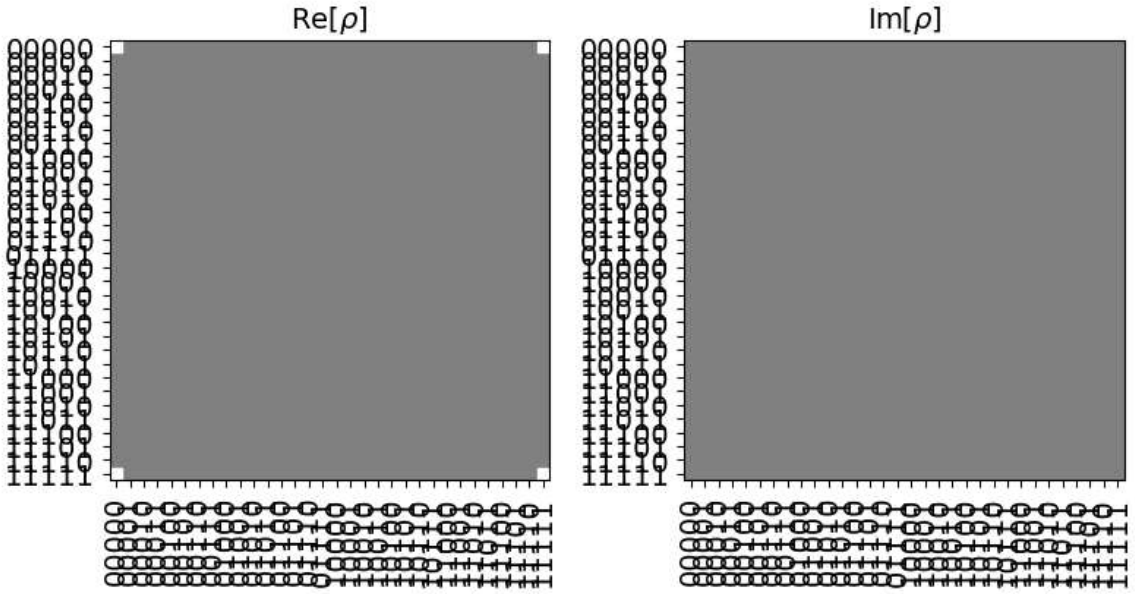

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

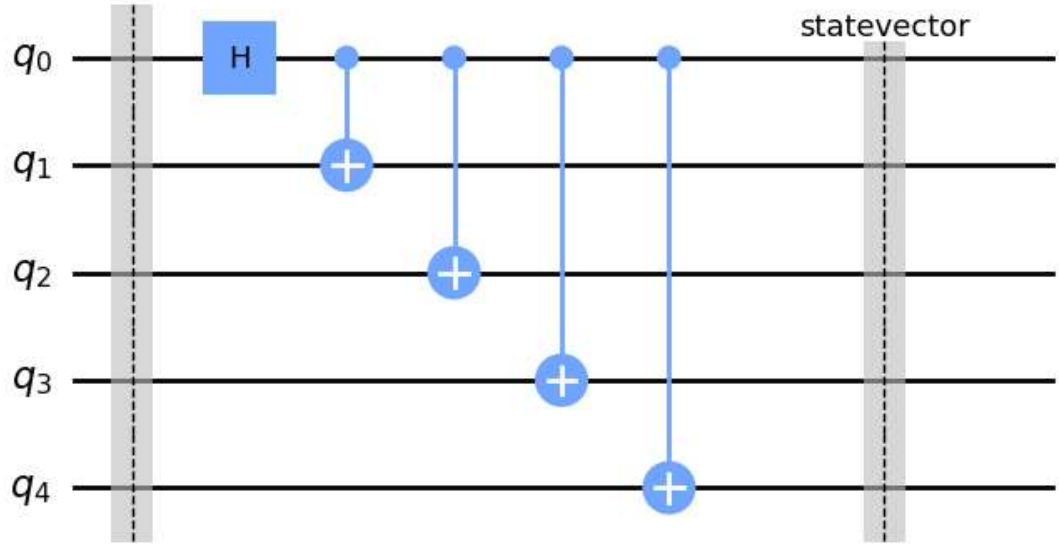

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

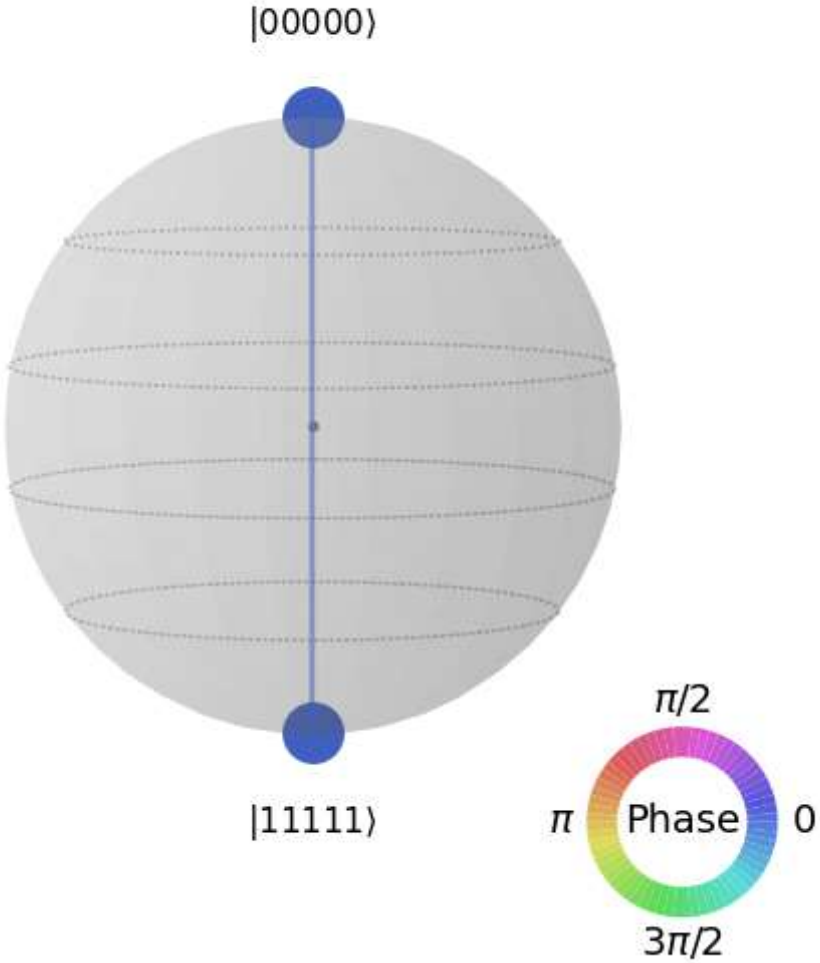

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

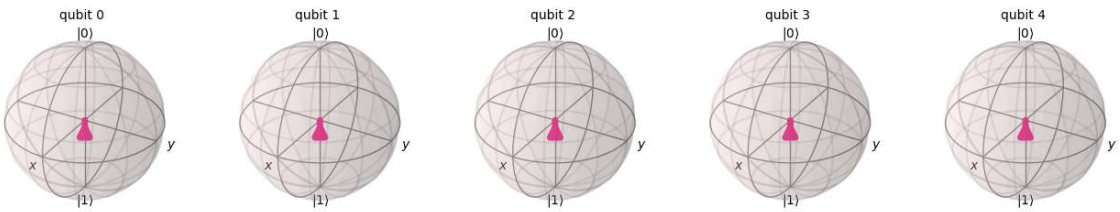

Im[ρ]
0

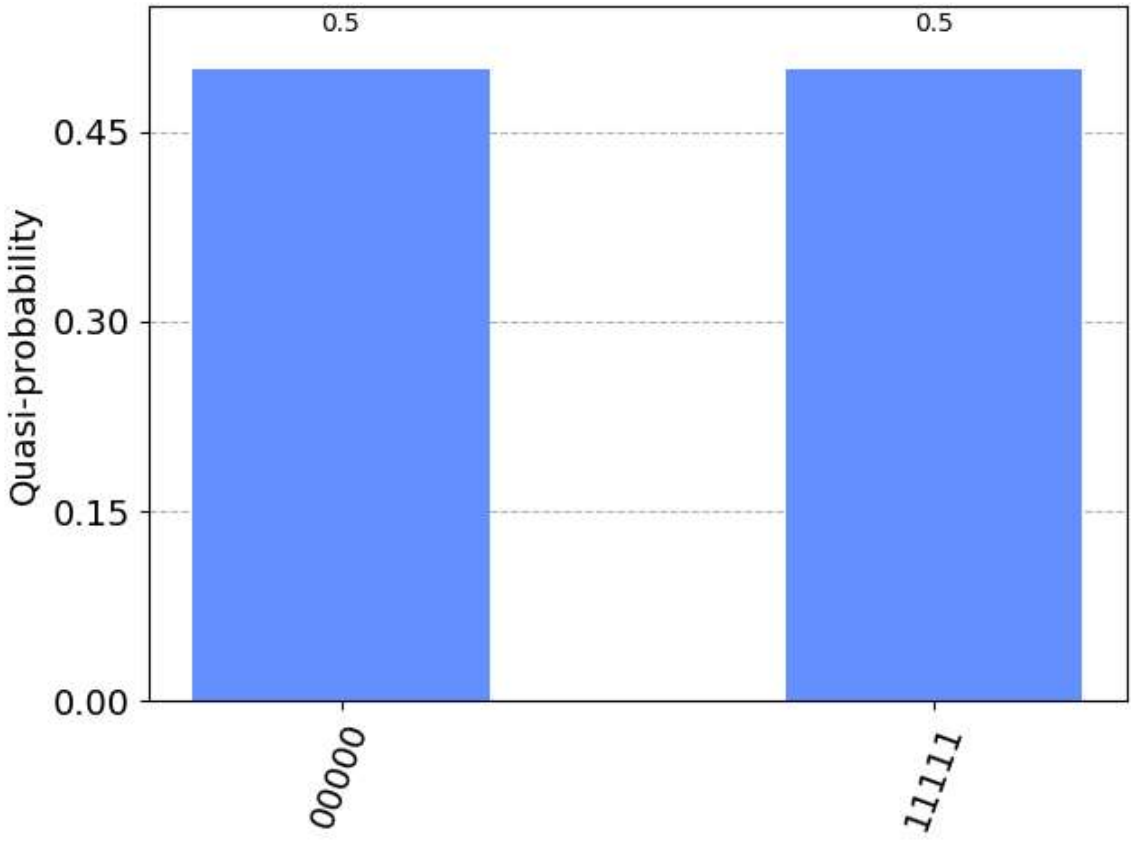

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

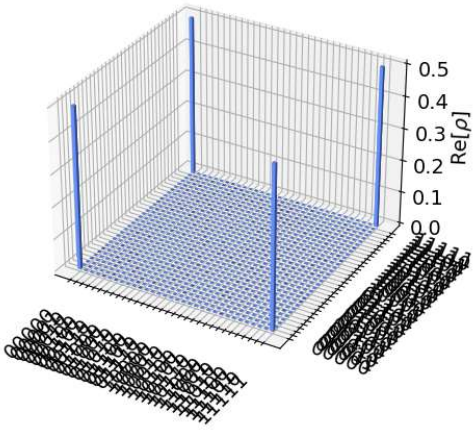

Re[ρ]
Im[ρ]

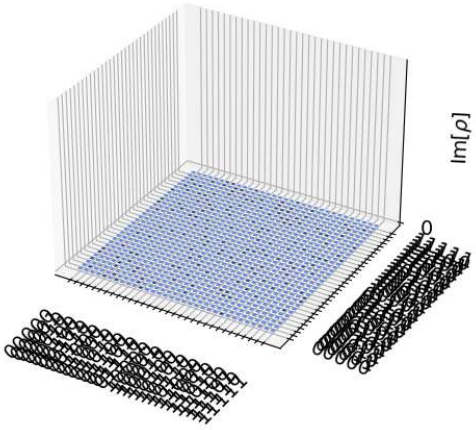

statevector
H
q_0
q_1
q_2
q_3
q_4

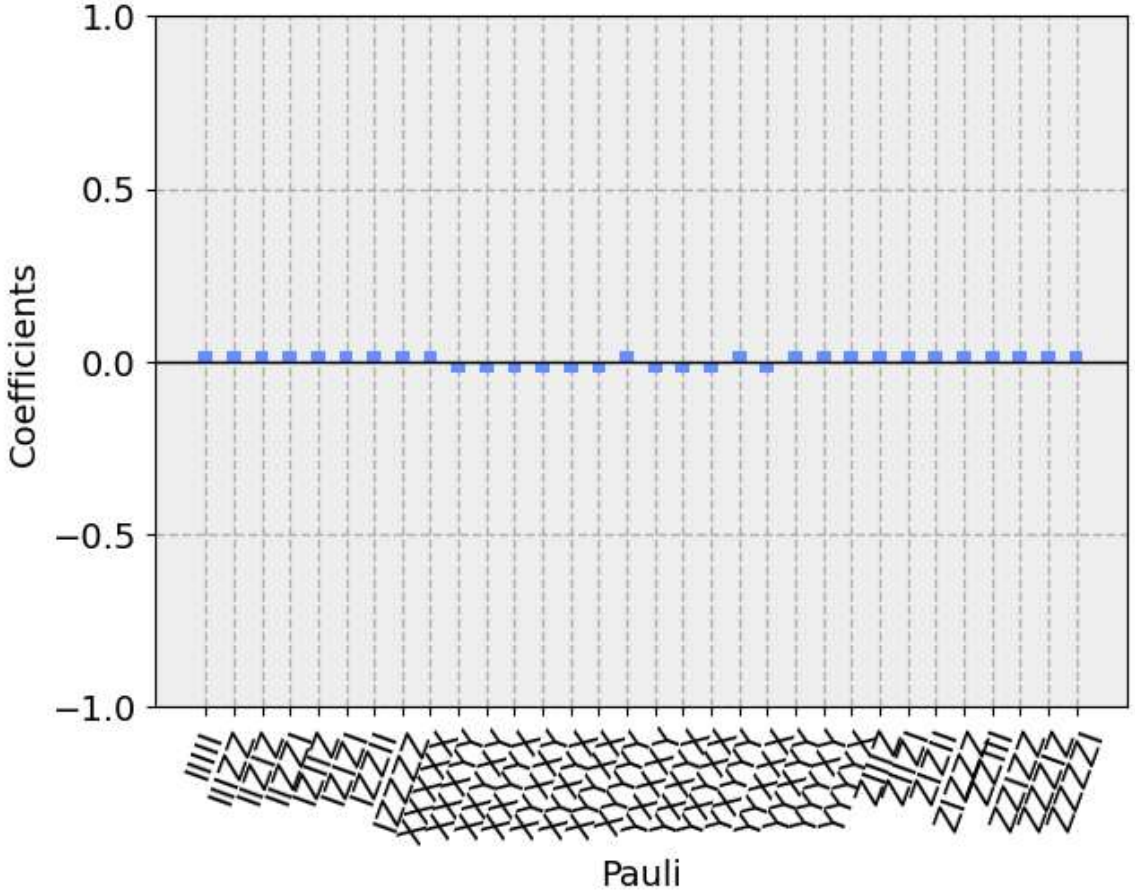

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

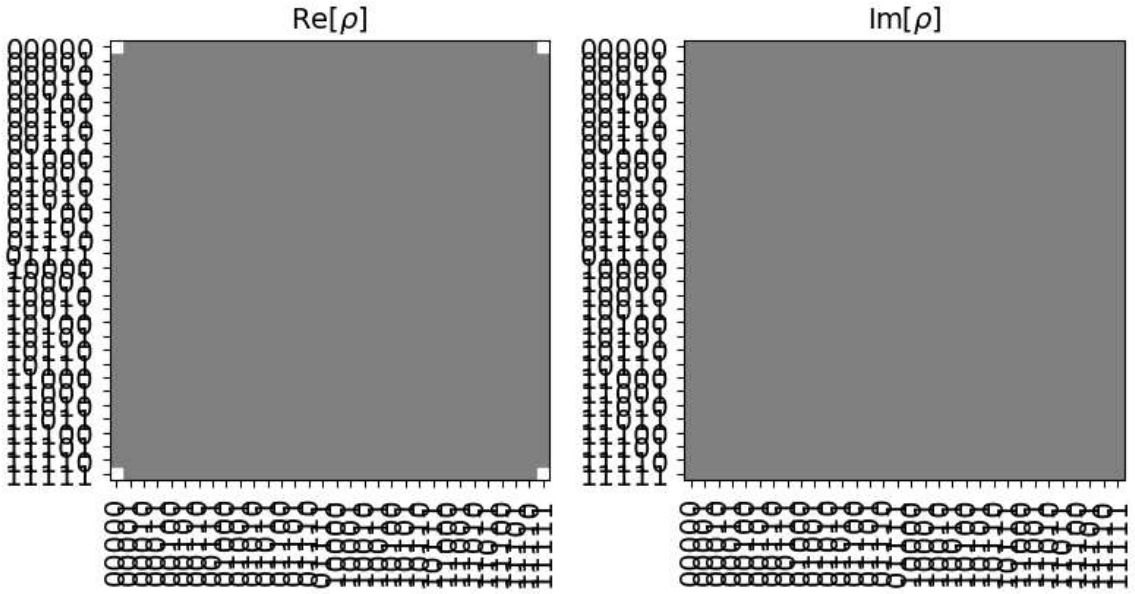

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

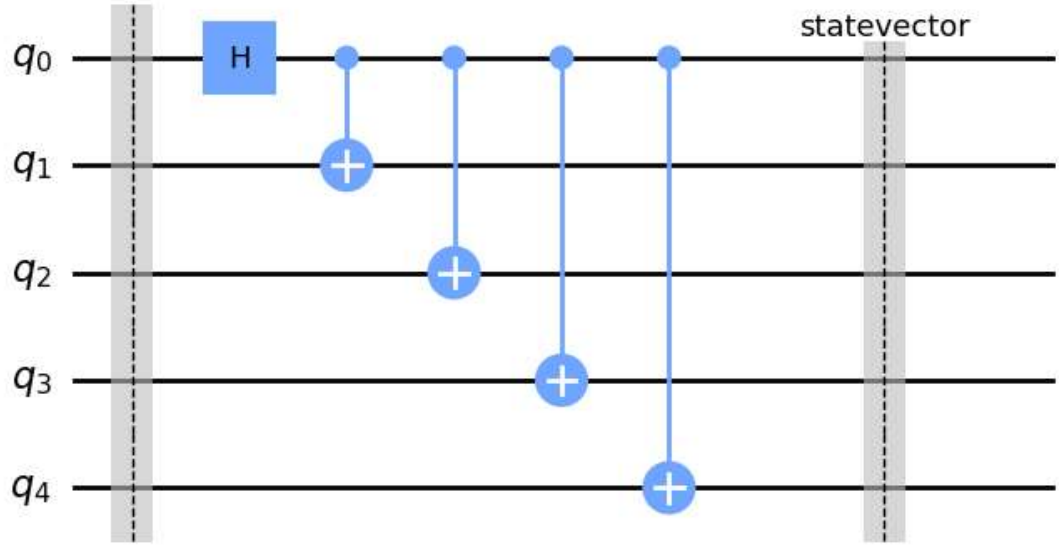

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

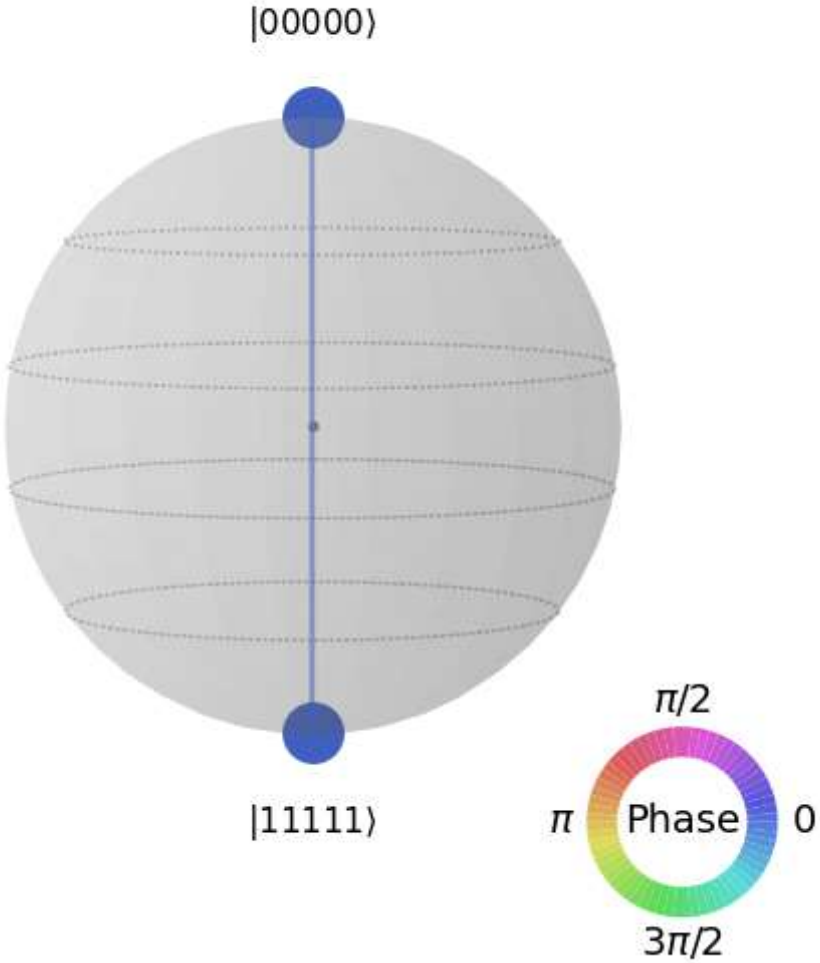

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

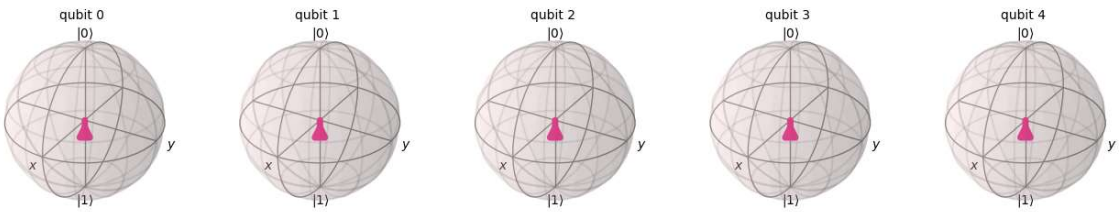

Im[ρ]
0

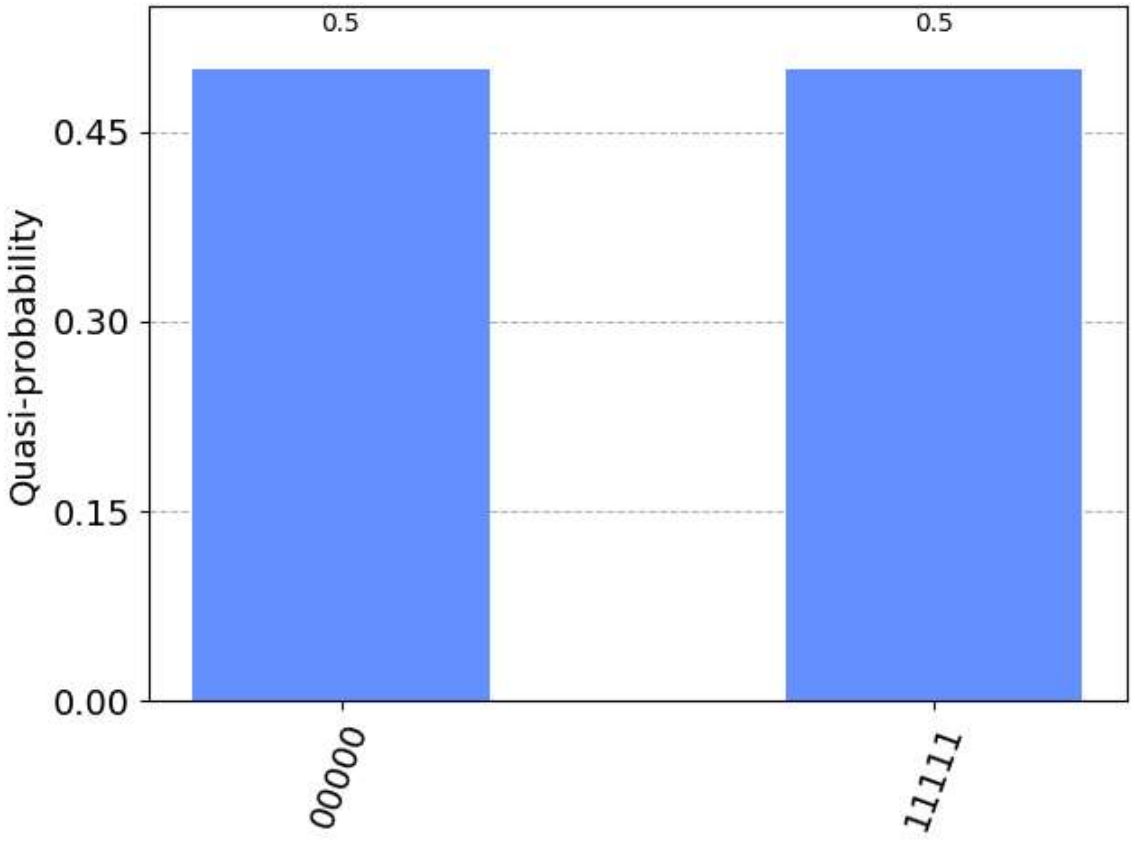

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli

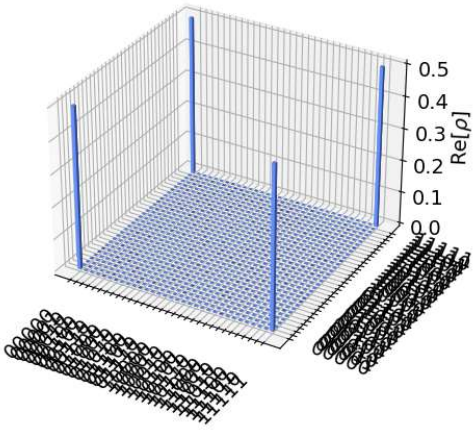

Re[ρ]
Im[ρ]

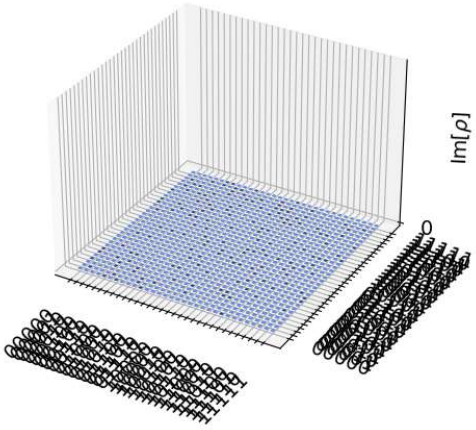

statevector
H
q_0
q_1
q_2
q_3
q_4

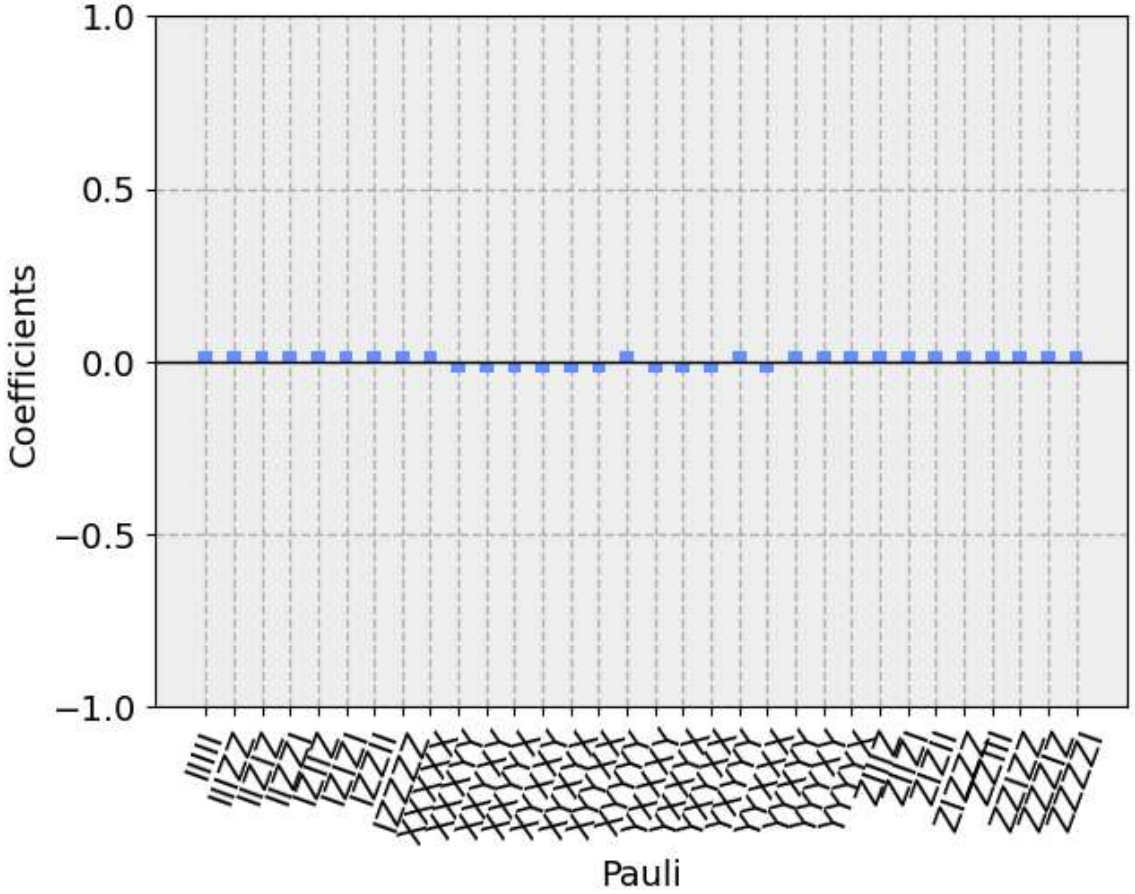

|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2

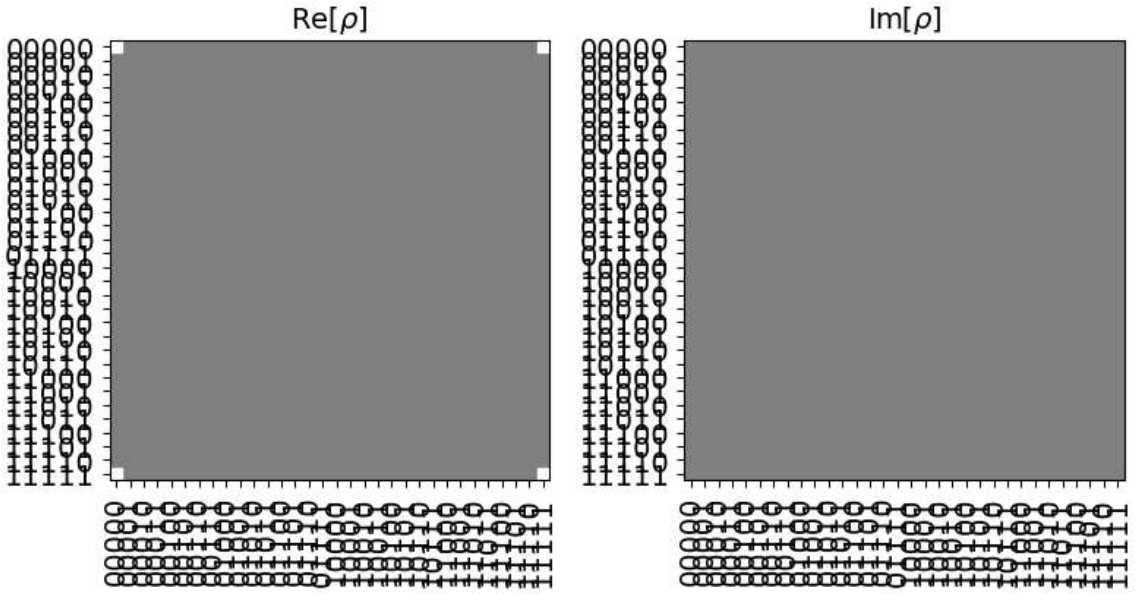

qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y

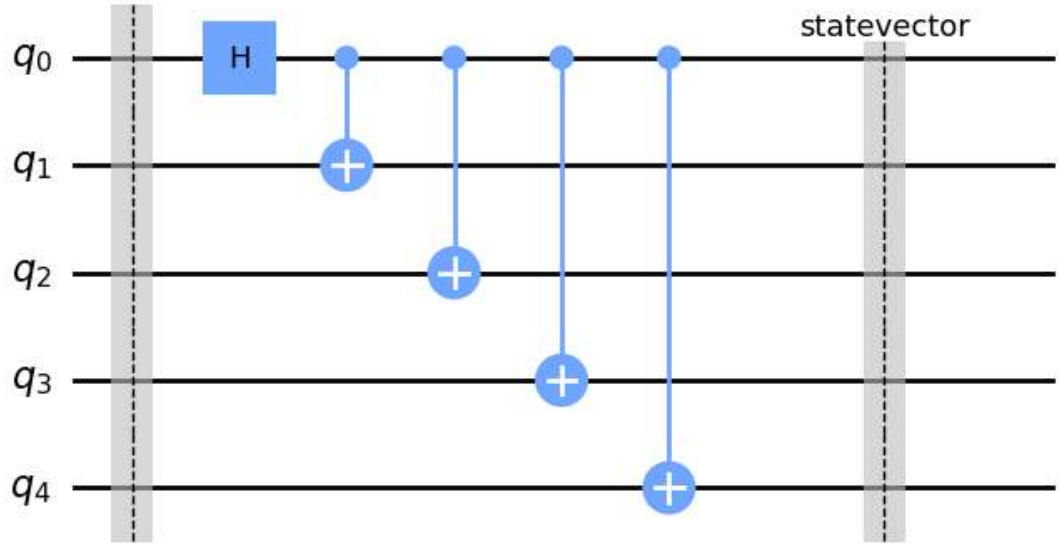

0.5
0.5
0.45
0.30
0.15
0.00
Quasi-probability
00000
11111

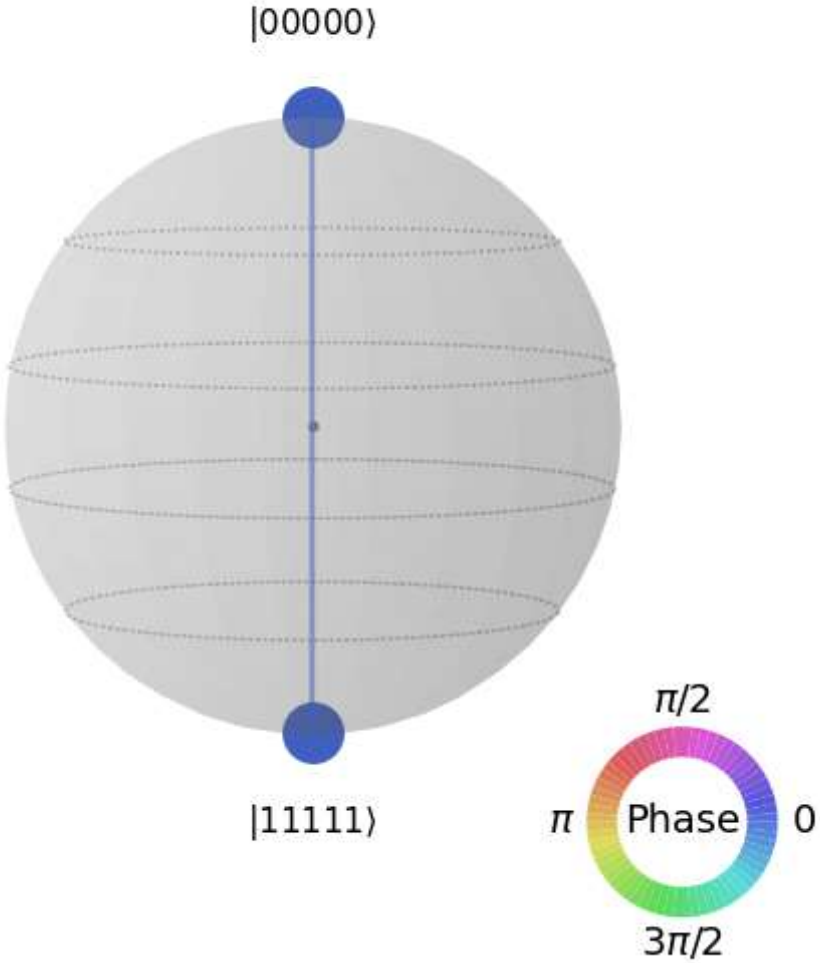

0.5
0.4
0.3
0.2
0.1
0.0
Re[ρ]

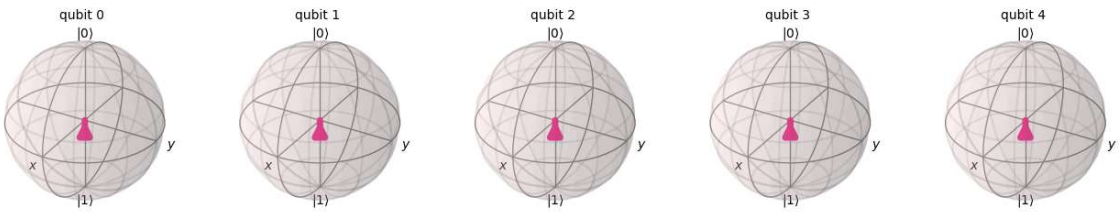

Im[ρ]
0

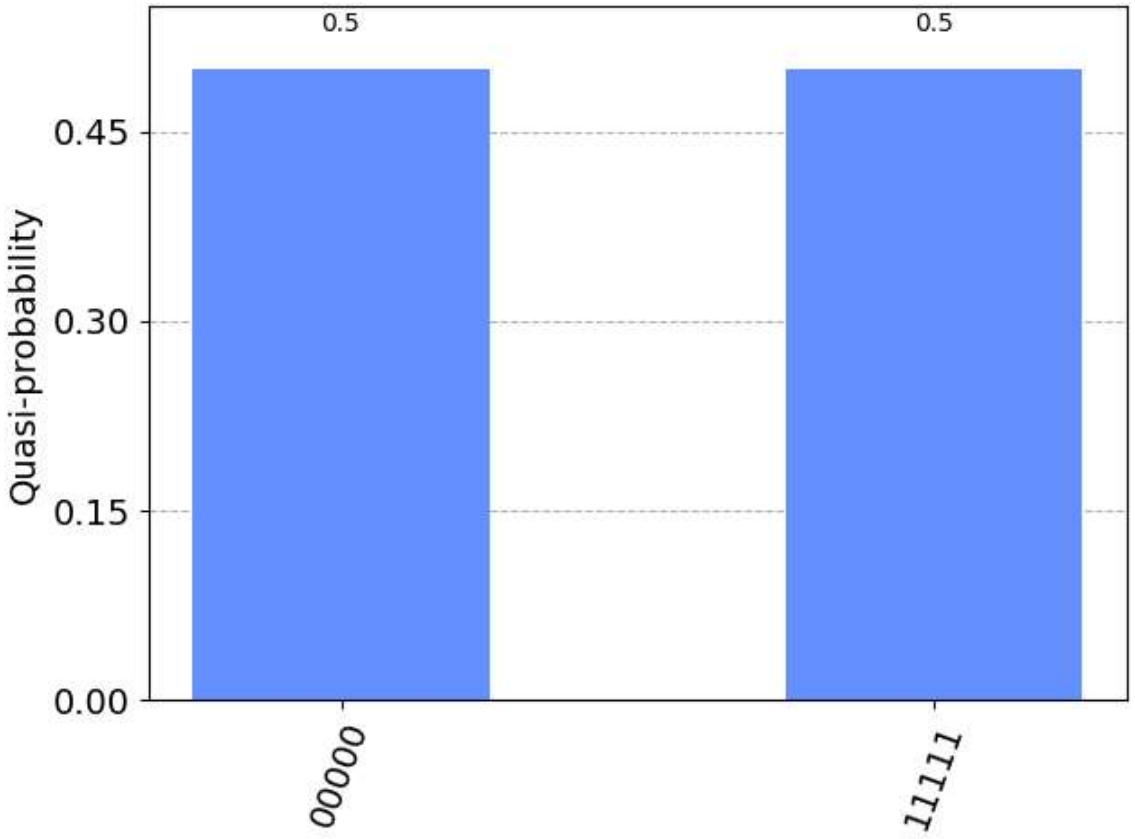

1.0
0.5
0.0
−0.5
−1.0
Coefficients
Pauli


Re[ρ]
Im[ρ]


statevector
H
q_0
q_1
q_2
q_3
q_4


|00000⟩
|11111⟩
π/2
π
Phase
0
3π/2


qubit 0
qubit 1
qubit 2
qubit 3
qubit 4
|0⟩
|1⟩
x
y



In [130…

```
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qub
                                       and not x.configuration().simulator
                                       and x.status().operational==True))
```

In [131…

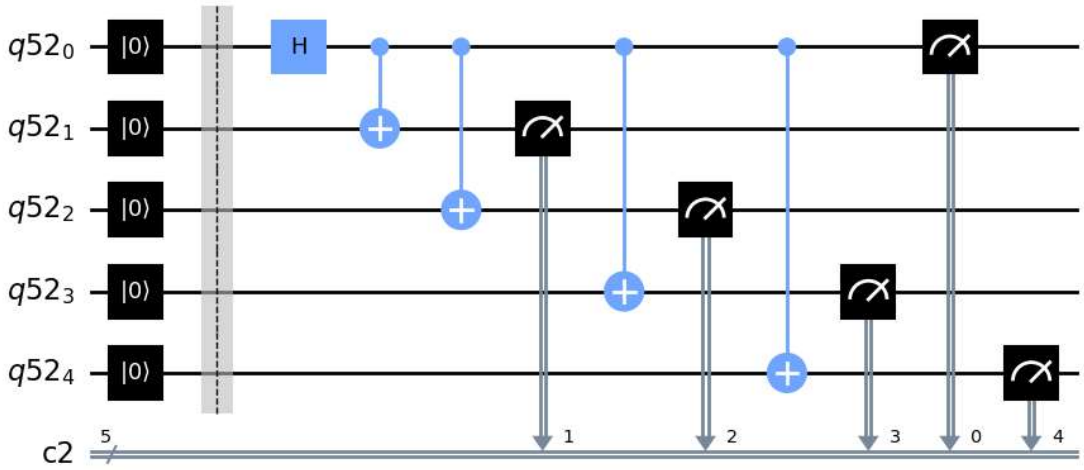
```
def create5QGHZRealDevice(inp1, inp2, inp3, inp4, inp5):
    qr = QuantumRegister(5)
    cr = ClassicalRegister(5)
    qc = QuantumCircuit(qr, cr)
    qc.reset(range(5))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)
    if inp3=='1':
        qc.x(1)
    if inp4=='1':
        qc.x(1)
    if inp5=='1':
        qc.x(1)

    qc.barrier()

    qc.h(0)
    qc.cx(0,1)
    qc.cx(0,2)
    qc.cx(0,3)
    qc.cx(0,4)

    qc.measure(qr, cr)

    job = execute(qc, backend=backend, shots=1000)
    job_monitor(job)
```

```
    result = job.result()

    return qc, result
```

In [132…

```
inp1 = 0
inp2 = 0
inp3 = 0
inp4 = 0
inp5 = 0

#first results
qc, first_result = create5QGHZRealDevice(inp1, inp2, inp3, inp4, inp5)
first_counts = first_result.get_counts()

# Draw the quantum circuit
display(qc.draw())
```

```
Job Status: job has successfully run
```

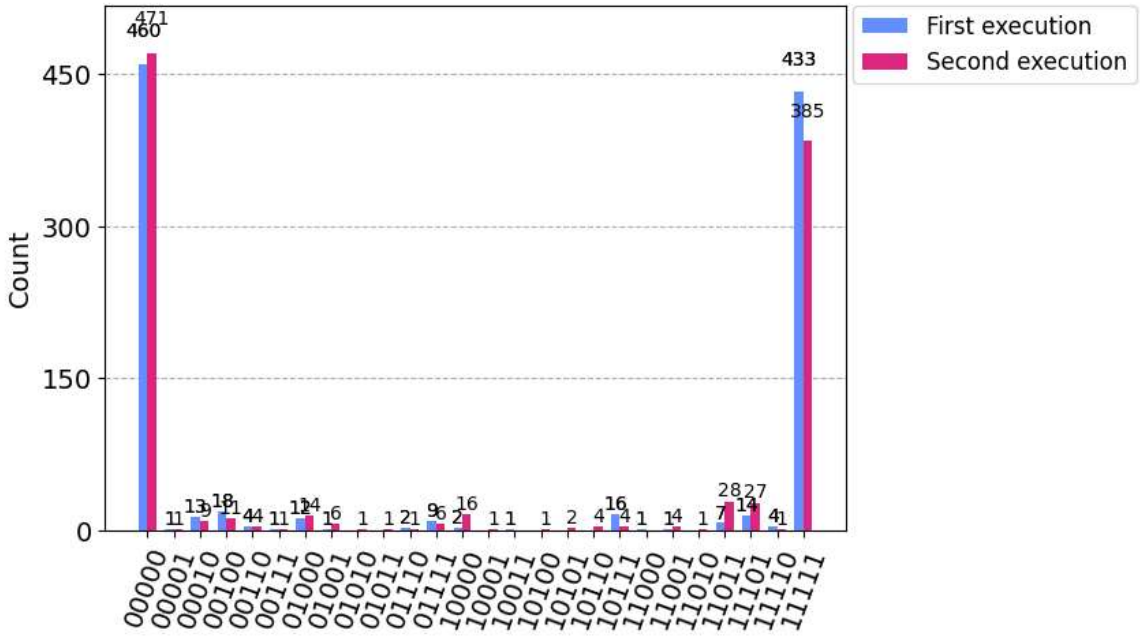


In [133…

```
#second results
qc, second_result = create5QGHZRealDevice(inp1, inp2, inp3, inp4, inp5)
second_counts = second_result.get_counts()

print('For inputs',inp5,inp4,inp3,inp2,inp1,'Representation of GHZ circuit state

# Plot results on histogram with legend
legend = ['First execution', 'Second execution']
plot_histogram([first_counts, second_counts], legend=legend)
```

```
Job Status: job has successfully run
For inputs 0 0 0 0 0 Representation of GHZ circuit states are,
```

Out[133]:



Las simulaciones en hardware real no se llevo a cabo debido a tiempos de espera aun en los sitemas menos ocupados de hasta 10-20 horas

In [26]:
```
QiskitRuntimeService.save_account(channel="ibm_quantum", token="e8fb340b1b0629a1
```

```
Traceback (most recent call last):
  Cell In[26], line 1
    QiskitRuntimeService.save_account(channel="ibm_quantum", token="e8fb340b1b0
629a13d7275b707b7ae53cd45d97daa7d004a75686200b30b76ae8be68b9196af0ad1c3f814c1d5
a0702e3f06ba45c4cff99d80d5d02993c98107")
  File /opt/conda/lib/python3.10/site-packages/qiskit_ibm_runtime/qiskit_runtim
e_service.py:705 in save_account
    AccountManager.save(
  File /opt/conda/lib/python3.10/site-packages/qiskit_ibm_runtime/accounts/mana
gement.py:59 in save
    return save_config(
  File /opt/conda/lib/python3.10/site-packages/qiskit_ibm_runtime/accounts/stor
age.py:34 in save_config
    raise AccountAlreadyExistsError(
AccountAlreadyExistsError: 'Named account (default-ibm-quantum) already exists.
Set overwrite=True to overwrite.'

Use %tb to get the full traceback.
```

Search for solution online