

Quantum _OFDM

Step 1: Create circuit for 2 user

Step2: Encode qubits

Step3: Apply QFT

Step4: Introduce Noisy Channel (Hence Kraus Operator , Non-unitary ,

hence no state vector only density matrices and measurement)

Step5: IQFT

Step6: Trans pile &Measurement

Step7: QBER Estimation Based on ^{2025BSZ8209}(Phase Noise= theta &

Q_OFDM.py 9+ X 2_user_QSMA.py 8 QCDMA_Modified.py 8 BER_Estimation.py 2 ●

C: > Users > DELL > Soumen_2025_BSZ_8209 > QKD > Q_OFDM.py > QBER Estimation

```

125 def QBER_Estimation(counts, shots):
152     )
153
154     # Threshold plane
155     ax.plot_surface(
156         Theta, Gamma, QBER_plane,
157         color='red',
158         alpha=0.35
159     )
160
161     ax.set_xlabel("Phase noise θ (radians)")
162     ax.set_ylabel("Amplitude damping γ")
163     ax.set_zlabel("QBER")
164
165     ax.set_title("QBER Surface with QKD Security Threshold")
166
167     fig.colorbar(surf, shrink=0.5, aspect=10, label="QBER")
168
169     plt.show()
170     return qber
171     pass
172
173
174 def main():
175     QBER=[]
176     #for i in range(0,1):
177     qc = QuantumCircuit(2, 2)
178     state_vector= Statevector.from_instruction(qc)
179     show_statevec(state_vector, "Initial State Vector")
180     print(qc.draw('mpl'))
181     plt.show()
182     qc, state_vector_after_encoding=Encoding(qc)
183     qc, state_vector_after_QFT=QFT_After_Encoding(qc)
184     qc, state_vector_after_Noisy_Quantum_Channel=Noisy_QFT(qc)
185     qc=Amplitude_Loss_Non_Unitary_Kraus_operator(qc)
186     qc=IQFT_After_Noisy_Quantum_Channel(qc)
187     counts=Measurement(qc)
188     shots=4096

```

C:\Windows\system32\cmd.e: X +

C:\Users\DELL\Soumen_2025_BSZ_8209\QKD>python Q_OFDM.py

=====

Initial State Vector

=====

|0000000000> (1+0j)

Figure 1

Initial State Vector

2025BSZ8209

2

```

# Threshold plane
ax.plot_surface(
    Theta, Gamma, QBER_plane,
    color='red',
    alpha=0.35
)

ax.set_xlabel("Phase noise θ (radians)")
ax.set_ylabel("Amplitude damping γ")
ax.set_zlabel("QBER")

ax.set_title("QBER Surface with QKD Security Threshold")

fig.colorbar(surf, shrink=0.5, aspect=10, label="QBER")

plt.show()
return qber
pass

main():
QBER=[]
for i in range(0,1):
qc = QuantumCircuit(2, 2)
state_vector= Statevector.from_instruction(qc)
show_statevec(state_vector, "Initial State Vector")
print(qc.draw('mpl'))
plt.show()
qc,state_vector_after_encoding=Encoding(qc)
qc,state_vector_after_QFT=QFT_After_Encoding(qc)
qc,state_vector_after_Noisy_Quantum_Channel=Noisy_QFT(qc)
qc=Amplitude_Loss_Non_Unitary_Kraus_operator(qc)
qc=IQFT_After_Noisy_Quantum_Channel(qc)
counts=Measurement(qc)
shots=4096
qber=QBER_Estimation(counts,shots)
QBER.append(qber)
print(QBER)

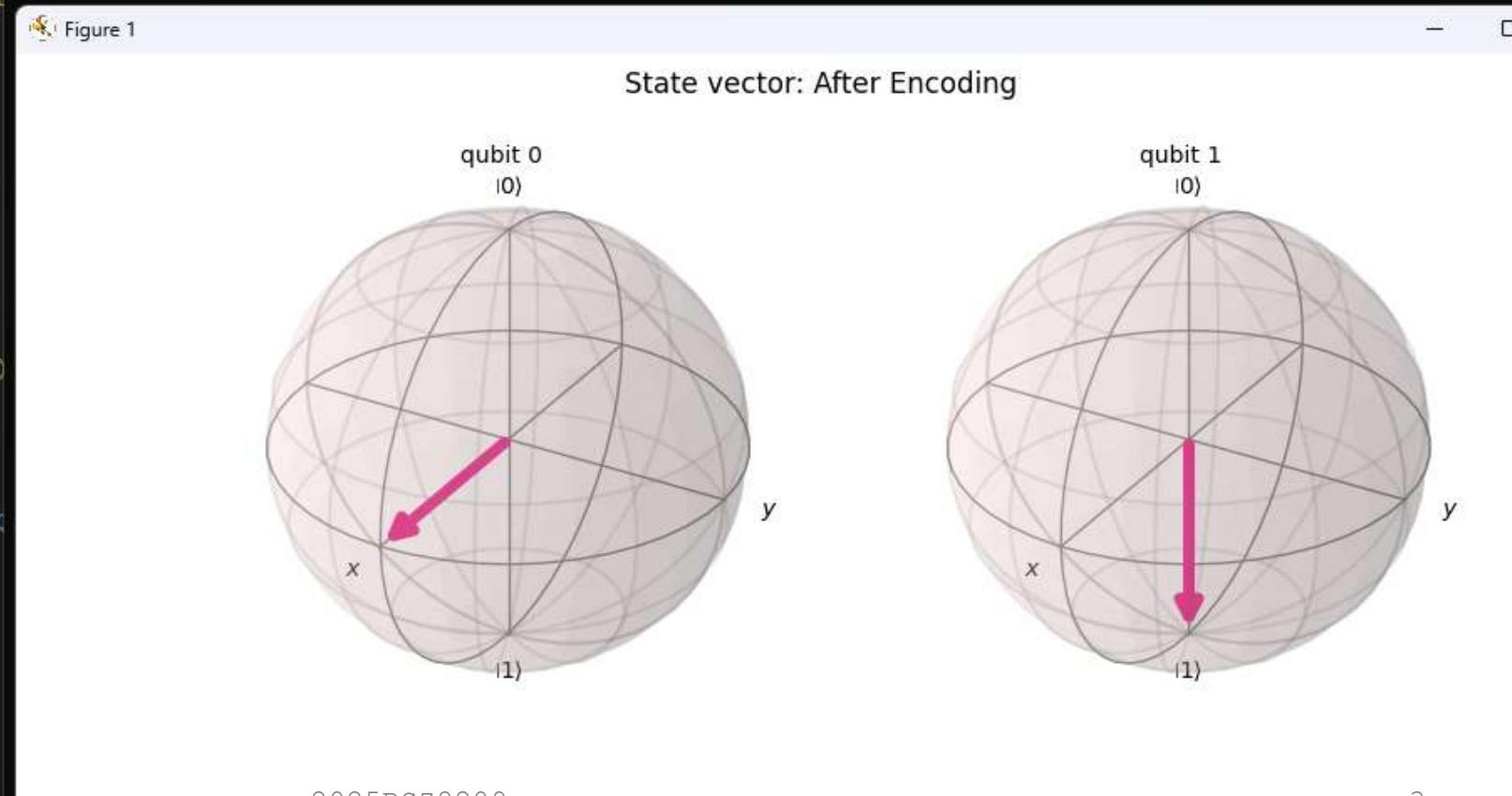
pass

```

```

C:\Users\DELL\Soumen_2025_BSZ_8209\QKD>python Q_OFDM.py
=====
Initial State Vector
=====
|0000000000> (1+0j)
Figure(161.878x284.278)
=====
State vector: After Encoding
=====
|0000000010> (0.7071067811865475+0j)
|0000000011> (0.7071067811865475+0j)

```



```

# Threshold plane
ax.plot_surface(
    Theta, Gamma, QBER_plane,
    color='red',
    alpha=0.35
)

ax.set_xlabel("Phase noise θ (radians)")
ax.set_ylabel("Amplitude damping γ")
ax.set_zlabel("QBER")

ax.set_title("QBER Surface with QKD Security Threshold")

fig.colorbar(surf, shrink=0.5, aspect=10, label="QBER")

plt.show()
return qber
pass

```

```

f main():
QBER=[]
for i in range(0,1):
qc = QuantumCircuit(2, 2)
state_vector= Statevector.from_instruction(qc)
show_statevec(state_vector, "Initial State Vector")
print(qc.draw('mpl'))
plt.show()
qc,state_vector_after_encoding=Encoding(qc)
qc,state_vector_after_QFT=QFT_After_Encoding(qc)
qc,state_vector_after_Noisy_Quantum_Channel=Noisy_QFT(qc)
qc=Amplitude_Loss_Non_Unitary_Kraus_operator(qc)
qc=IQFT_After_Noisy_Quantum_Channel(qc)
counts=Measurement(qc)
shots=4096
qber=QBER_Estimation(counts,shots)
QBER.append(qber)
print(QBER)

pass

```

C:\Users\DELL\Soumen_2025_BSZ_8209\QKD>python Q_OFDM.py

=====

Initial State Vector

=====

|0000000000> (1+0j)

Figure(161.878x284.278)

=====

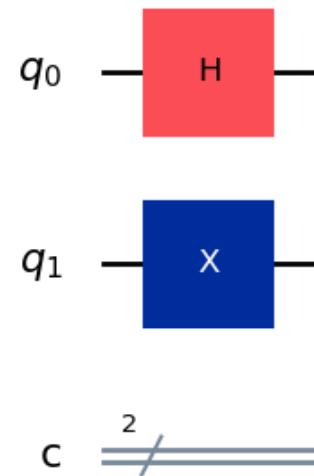
State vector: After Encoding

=====

|0000000010> (0.7071067811865475+0j)

|0000000011> (0.7071067811865475+0j)

Figure(203.683x284.278)



```

ax.set_xlabel("Phase noise θ (radians)")
ax.set_ylabel("Amplitude damping γ")
ax.set_zlabel("QBER")

ax.set_title("QBER Surface with QKD Security Threshold")

fig.colorbar(surf, shrink=0.5, aspect=10, label="QE")

plt.show()
return qber
pass

main():
QBER=[]
for i in range(0,1):
qc = QuantumCircuit(2, 2)
state_vector= Statevector.from_instruction(qc)
show_statevec(state_vector, "Initial State Vector")
print(qc.draw('mpl'))
plt.show()
qc.state_vector_after_encoding=Encoding(qc)
qc.state_vector_after_QFT=QFT_After_Encoding(qc)
qc.state_vector_after_Noisy_Quantum_Channel=Noisy_QFT(qc)
qc=Amplitude_Loss_Non_Unitary_Kraus_operator(qc)
qc=IQFT_After_Noisy_Quantum_Channel(qc)
counts=Measurement(qc)
shots=4096
qber=QBER_Estimation(counts,shots)
QBER.append(qber)
print(QBER)

pass
()

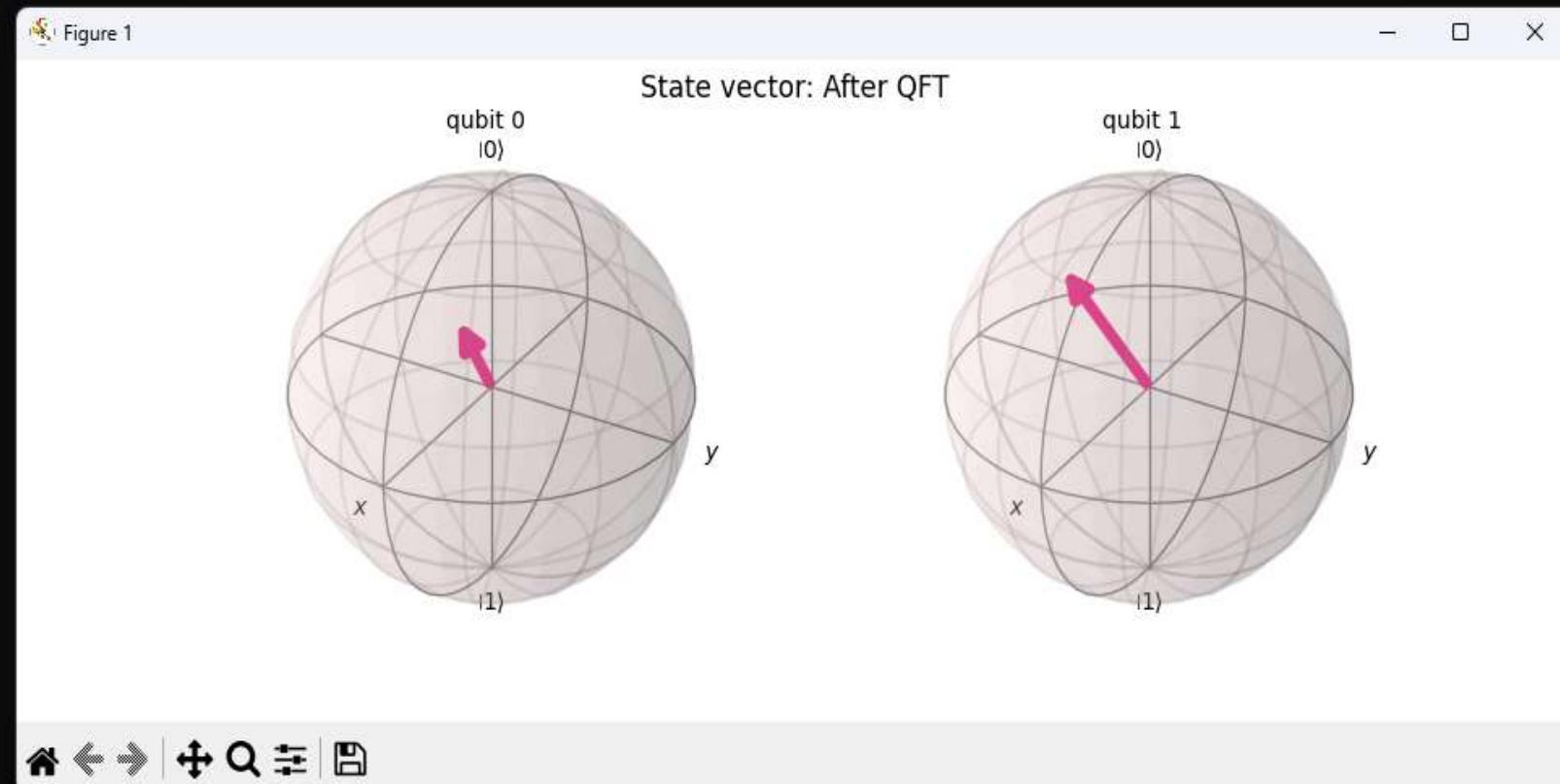
```

```

=====
State vector: After Encoding
=====
|0000000010> (0.7071067811865475+0j)
|0000000011> (0.7071067811865475+0j)
Figure(203.683x284.278)

=====
State vector: After QFT
=====
|0000000000> (0.7071067811865474+0j)
|0000000001> (-0.3535533905932737-0.3535533905932737j)
|0000000011> (-0.3535533905932737+0.3535533905932737j)

```



L > Soumen_2025_BSZ_8209 > QKD > Q_OFDM.py > QBER Estimation

BER_Estimation(counts,shots):

```

Threshold plane
..plot_surface(
    Theta, Gamma, QBER_plane,
    color='red',
    alpha=0.35

.set_xlabel("Phase noise θ (radians)")
.set_ylabel("Amplitude damping γ")
.set_zlabel("QBER")

.set_title("QBER Surface with QKD Security Threshold")
g.colorbar(surf, shrink=0.5, aspect=10, label="QBER")
t.show()
return qber
iss

in():
    ER=[]
for i in range(0,1):
    qc = QuantumCircuit(2, 2)
    state_vector= Statevector.from_instruction(qc)
    show_statevec(state_vector, "Initial State Vector")
    int(qc.draw('mpl'))
    t.show()
    ,state_vector_after_encoding=Encoding(qc)
    ,state_vector_after_QFT=QFT_After_Encoding(qc)
    ,state_vector_after_Noisy_Quantum_Channel=Noisy_QFT(qc)
    :=Amplitude_Loss_Non_Unitary_Kraus_operator(qc)
    :=IQFT_After_Noisy_Quantum_Channel(qc)
    counts=Measurement(qc)
    shots=4096

```

C:\Windows\system32\cmd.e X + |

C:\Users\DELL\Soumen_2025_BSZ_8209\QKD>python Q_OFDM.py

=====

Initial State Vector

=====

$|0000000000\rangle (1+0j)$

Figure(161.878x284.278)

=====

State vector: After Encoding

=====

$|0000000010\rangle (0.7071067811865475+0j)$

$|0000000011\rangle (0.7071067811865475+0j)$

Figure(203.683x284.278)

=====

State vector: After QFT

=====

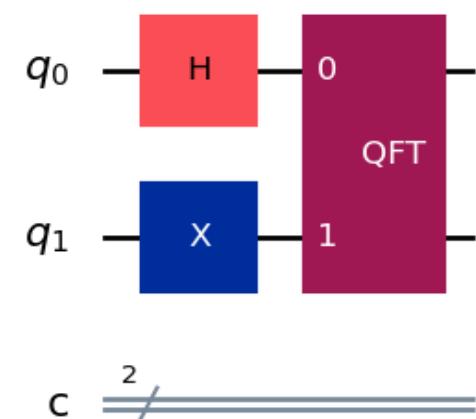
$|0000000000\rangle (0.7071067811865474+0j)$

$|0000000001\rangle (-0.3535533905932737-0.3535533905932737j)$

$|0000000011\rangle (-0.3535533905932737+0.3535533905932737j)$

Figure(287.294x284.278)

Figure 1



File Edit Selection View Go Run Terminal Help

C:\Windows\system32\cmd.e: + v

```

43 def QFT_After_Encoding(qc):
44     qc.append(QFT())
45     return qc.state_vector_after_QFT
46     pass
47
48 #Step 3: Non-ideal quantum channel (a
49 #Implements (k)->i0kik)
50 def Noisy_Quantum_Channel(qc):
51     #theta = [0.0, 0.3, 0.0, -0.3] #
52     theta=float(input("Enter the thet
53     theta = [0.0, theta, 0.0, -theta]
54     phase_unitary = Diagonal([
55         np.exp(1j * theta[0]), # |00>
56         np.exp(1j * theta[1]), # |01>
57         np.exp(1j * theta[2]), # |10>
58         np.exp(1j * theta[3]), # |11>
59     ])
60
61     qc.append(phase_unitary, [0, 1])
62     state_vector_after_Noisy_Quantum
63     show_statevec(state_vector_after_
64     print(qc.draw('mpl'))
65     plt.show()
66     return qc.state_vector_after_Nois
67     pass
68
69 def Amplitude_Loss_Non_Unitary_Kraus
70     #gamma = 0.2 # loss strength (co
71     gamma=float(input("Enter the gamm
72     #Amplitude loss is non-unitary, s
73     if gamma>0:
74         K0 = np.array([[1, 0], [0, np.sqr
75
76         amp_damp = Kraus([K0, K1]) #
77         #Apply independently to both
78         qc.append(amp_damp, [0])
79         qc.append(amp_damp, [1])
80
81
82     #state_vector_after_Amplitude_Los
83     #show_statevec(state_vector_after_
84     print(qc.draw('mpl'))
85     plt.show()
86     rho_after_loss = DensityMatrix.fr
87     print(rho_after_loss)
88     #state_vector_after_Amplitude_Loss = Statevector.from_instruction(qc)
89     #return qc
90     pass
91
92 def TOFT_After_Noisy_Quantum_Channel(qc):
93
94
95
96

```

Initial State Vector

```
|0000000000> (1+0j)
Figure(161.878x284.278)
```

State vector: After Encoding

```
|000000010> (0.7071067811865475+0j)
|000000011> (0.7071067811865475+0j)
Figure(203.683x284.278)
```

State vector: After QFT

```
|0000000000> (0.7071067811865474+0j)
|0000000001> (-0.3535533905932737-0.3535533905932737j)
|0000000011> (-0.3535533905932737+0.3535533905932737j)
Figure(287.294x284.278)
Enter the theta=0.3
```

State vector: Information passes through after Noisy Quantum Channel

qubit 0
|0>
y
|1>
x

qubit 1
|0>
y
|1>
x

state which ins unitary

15°C
Mostly cloudy

2025BSZ8209

166, Col 1 Spaces: 4 UTF-8 CRLF {} Python

ENG IN 21:12 18-12-2025

```

qc.show()
return qc,state_vector_after_QFT
pass

# 3: Non-ideal quantum channel (alpha elements |k>→eiθ|k|k)
Noisy_Quantum_Channel(qc):
#theta = [0.0, 0.3, 0.0, -0.3] #
theta=float(input("Enter the theta"))
theta = [0.0, theta, 0.0, -theta]
phase_unitary = Diagonal([
np.exp(1j * theta[0]), # |00>
np.exp(1j * theta[1]), # |01>
np.exp(1j * theta[2]), # |10>
np.exp(1j * theta[3]), # |11>

qc.append(phase_unitary, [0, 1])
state_vector_after_Noisy_Quantum_
show_statevec(state_vector_after_
print(qc.draw('mpl'))
plt.show()
return qc,state_vector_after_Nois
pass

Amplitude_Loss_Non_Unitary_Kraus_
#gamma = 0.2 # loss strength (co
gamma=float(input("Enter the gamm
#Amplitude loss is non-unitary, s
if gamma>0:
    K0 = np.array([[1, 0], [0, np
    K1 = np.array([[0, np.sqrt(ga
        amp_damp = Kraus([K0, K1]) #
    #Apply independently to both
    qc.append(amp_damp, [0])

```

C:\Windows\system32\cmd.e X + -

Initial State Vector

|0000000000> (1+0j)
Figure(161.878x284.278)

State vector: After Encoding

|000000010> (0.7071067811865475+0j)
|000000011> (0.7071067811865475+0j)
Figure(203.683x284.278)

State vector: After QFT

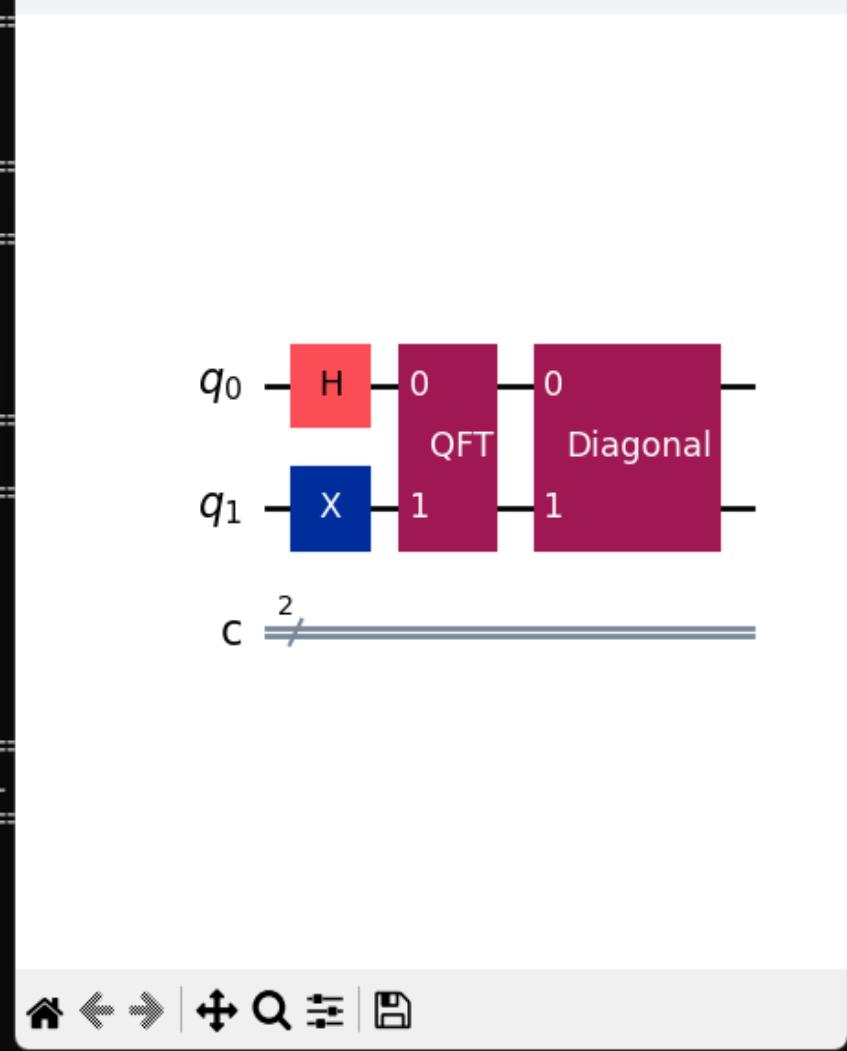
|000000000> (0.7071067811865474+0j)
|0000000001> (-0.3535533905932737-0.3535533905932737j)
|0000000011> (-0.3535533905932737+0.3535533905932737j)
Figure(287.294x284.278)

Enter the theta=0.3

State vector: Information passes through after Noisy Quantum Channel

|0000000000> (0.7071067811865472-1.3877787807814457e-17j)
|0000000001> (-0.23328028383389054-0.4422446259417736j)
|0000000011> (-0.23328028383389054+0.4422446259417736j)
Figure(454.517x284.278)

Figure 1



Q_OFDM.py 9+ X

2_user_QSMA.py 8

QCDMA_Modified.py 8

BER_Estimation.py 2 ●

C: > Users > DELL > Soumen_2025_BSZ_8209 > QKD > Q_OFDM.py > Amplitude_Loss_Non_Unitary_Kraus_operator

```
43 def QFT_After_Encoding(qc):
44     qc.show()
45     return qc, state_vector_after_QFT
46     pass
47
48 #Step 3: Non-ideal quantum channel (a
49 #Implements |k>→eiθk|k>
50 def Noisy_Quantum_Channel(qc):
51     #theta = [0.0, 0.3, 0.0, -0.3] #
52     theta=float(input("Enter the theta"))
53     theta = [0.0, theta, 0.0, -theta]
54     phase_unitary = Diagonal([
55         np.exp(1j * theta[0]), # |00>
56         np.exp(1j * theta[1]), # |01>
57         np.exp(1j * theta[2]), # |10>
58         np.exp(1j * theta[3]), # |11>
59     ])
60     qc.append(phase_unitary, [0, 1])
61     state_vector_after_Noisy_Quantum_
62     show_statevec(state_vector_after_
63     print(qc.draw('mpl'))
64     plt.show()
65     return qc, state_vector_after_Nois_
66     pass
67
68 def Amplitude_Loss_Non_Unitary_Kraus_
69     #gamma = 0.2 # loss strength (co
70     gamma=float(input("Enter the gamm
71     #Amplitude loss is non-unitary, s
72     if gamma>0:
73         K0 = np.array([[1, 0], [0, np
74         K1 = np.array([[0, np.sqrt(ga
75
76         amp_damp = Kraus([K0, K1]) #
77         #Apply independently to both
78         qc.append(amp_damp, [0, 1])
79
80
81
82
83
```

C:\Windows\system32\cmd.e: X + ▾

=====

Initial State Vector

=====

|0000000000> (1+0j) Figure 1

Figure(161.878x284.2)

State vector: After

|0000000010> (0.707)

|0000000011> (0.707)

Figure(203.683x284.2)

State vector: After

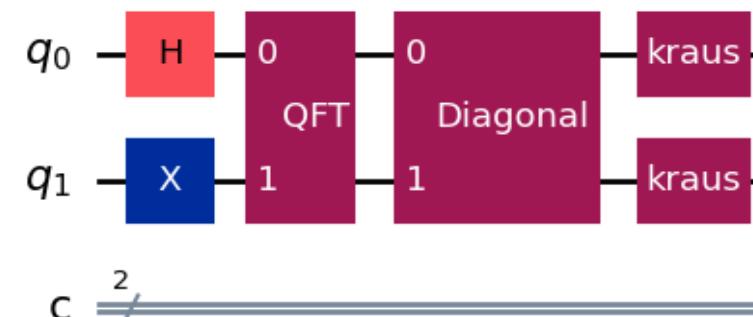
|0000000000> (0.707)

|0000000001> (-0.35)

|0000000011> (-0.35)

Figure(287.294x284.2)

Enter the theta=0.3



State vector: Information passes through after Noisy Quantum Channel

|0000000000> (0.7071067811865472-1.3877787807814457e-17j)

|0000000001> (-0.23328028383389054-0.4422446259417736j)

|0000000011> (-0.23328028383389054+0.4422446259417736j)

Figure(454.517x284.278)

Enter the gamma=0.2

Figure(538.128x284.278) 2025BSZ8209

```

50     return qc, state_vector_after_QFT
51     pass
52
53 #Step 3: Non-ideal quantum channel (a
54 #Implements |k>→eiθ|k⟩
55 def Noisy_Quantum_Channel(qc):
56     theta = [0.0, 0.3, 0.0, -0.3] #
57     theta=float(input("Enter the thet
58     theta = [0.0, theta, 0.0, -theta]
59     phase_unitary = Diagonal([
60         np.exp(1j * theta[0]), # |00>
61         np.exp(1j * theta[1]), # |01>
62         np.exp(1j * theta[2]), # |10>
63         np.exp(1j * theta[3]), # |11>
64     ])
65     qc.append(phase_unitary, [0, 1])
66     state_vector_after_Noisy_Quantum_
67     show_statevec(state_vector_after_
68     print(qc.draw('mpl'))
69     plt.show()
70     return qc, state_vector_after_Nois
71     pass
72
73 def Amplitude_Loss_Non_Unitary_Kraus_
74     #gamma = 0.2 # loss strength (co
75     gamma=float(input("Enter the gamm
76     #Amplitude loss is non-unitary, s
77     if gamma>0:
78         K0 = np.array([[1, 0], [0, np
79         K1 = np.array([[0, np.sqrt(ga
80
81         amp_damp = Kraus([K0, K1]) #
82         #Apply independently to both
83         qc.append(amp_damp, [0])
84         qc.append(amp_damp, [1])
85
86         #state_vector_after_Amplitude_Los
87         #show_statevec(state_vector_after_
88         print(qc.draw('mpl'))
89         plt.show()

```

|0000000000> (1+0j)
Figure(161.878x284.278)

=====

State vector: After Encoding
Figure 1

|0000000010> (0.70710678+0.j)
|0000000011> (0.70710678+0.j)
Figure(203.683x284.278)

=====

State vector: After QFT
Figure 2

|0000000000> (0.70710678+0.j)
|0000000001> (-0.35355339+0.j)
|0000000011> (-0.35355339+0.j)
Figure(287.294x284.278)

Enter the theta=0.3

=====

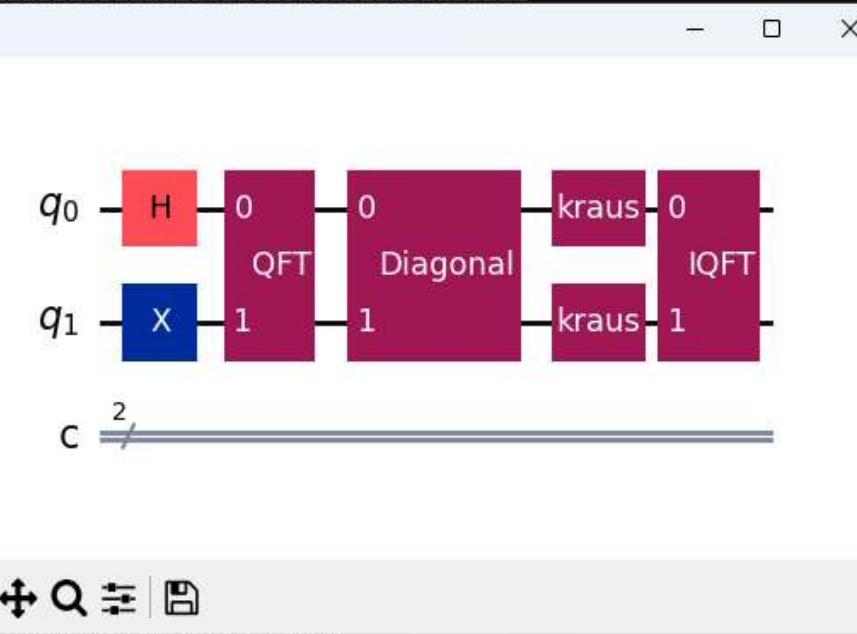
State vector: Information Loss
Figure 3

|0000000000> (0.7071067811865472-1.3877787807814457e-17j)
|0000000001> (-0.23328028383389054-0.4422446259417736j)
|0000000011> (-0.23328028383389054+0.4422446259417736j)
Figure(454.517x284.278)

Enter the gamma=0.2
Figure(538.128x284.278)

DensityMatrix([[0.56 +0.j, -0.14753941+0.27970006j,
-0.02525158+0.03691013j, -0.13196326-0.25017134j],
[-0.14753941-0.27970006j, 0.24 +0.j,
0. +0.j, -0.10100632+0.14764052j],
[-0.02525158-0.03691013j, 0. +0.j
0.04 +0.j, 0. +0.j],
[-0.13196326+0.25017134j, -0.10100632-0.14764052j,
0. +0.j, 0.16 +0.j]],
dims=(2, 2))

Figure(621.739x284.278)



Q_OFDM.py 9+ X

2_user_QSMA.py 8

QCDMA_Modified.py 8

BER_Estimation.py 2

:> Users > DELL > Soumen_2025_BSZ_8209 > QKD > Q_OFDM.py > Amplitude_Loss_Non_Unitary_Kraus_operator

```
43 def QFT_After_Encoding(qc):
44     plt.show()
45     return qc, state_vector_after_QFT
46 pass
47
48 #Step 3: Non-ideal quantum channel ( $\alpha$ )
49 #Implements  $|k\rangle \rightarrow e^{i\theta} k |k\rangle$ 
50 def Noisy_Quantum_Channel(qc):
51     theta = [0.0, 0.3, 0.0, -0.3] #
52     theta=float(input("Enter the theta"))
53     theta = [0.0, theta, 0.0, -theta]
54     phase_unitary = Diagonal([
55         np.exp(1j * theta[0]), # |00>
56         np.exp(1j * theta[1]), # |01>
57         np.exp(1j * theta[2]), # |10>
58         np.exp(1j * theta[3]), # |11>
59     ])
60     qc.append(phase_unitary, [0, 1])
61     state_vector_after_Noisy_Quantum =
62     show_statevec(state_vector_after)
63     print(qc.draw('mpl'))
64     plt.show()
65     return qc, state_vector_after_Noisy_Quantum
66 pass
67
68 def Amplitude_Loss_Non_Unitary_Kraus():
69     gamma = 0.2 # loss strength (coefficient)
70     gamma=float(input("Enter the gamma"))
71     #Amplitude loss is non-unitary, so
72     if gamma>0:
73         K0 = np.array([[1, 0], [0, np.sqrt(gamma)]])
74         K1 = np.array([[0, np.sqrt(gamma)], [0, 1]])
75         amp_damp = Kraus([K0, K1]) #
76         #Apply independently to both
77         qc.append(amp_damp, [0, 1])
78
79
80
81
82
83
```

C:\Windows\system32\cmd.e: X + v

Figure(161.878x284.278)

=====
State vector: After Encoding
=====

|0000000010> (0.7071067811865475+0j)
|0000000011> (0.7071067811865475+0j)

Figure(203.683x284.278)

=====
State vector: After Encoding
=====

|0000000000> (

|0000000001> (

|0000000011> (

Figure(287.294)

Enter the theta:

=====
State vector:
=====

|0000000000> (

|0000000001> (

|0000000011> (

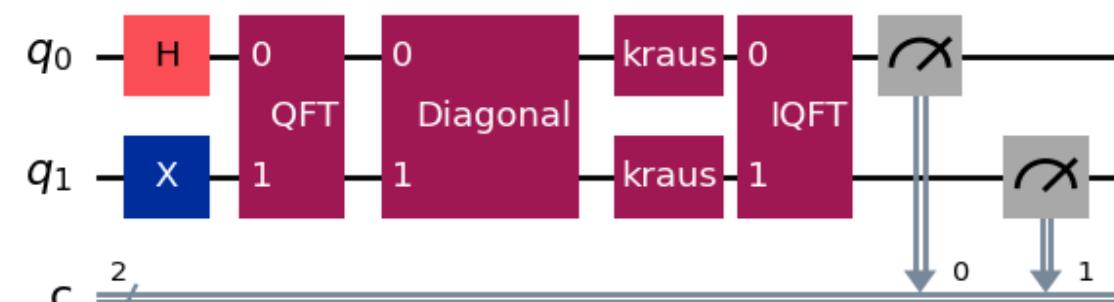
Figure(454.517)

Enter the gamma:

=====
Figure(538.128)



DensityMatrix([[[0.5, 0.0, 0.0, 0.0], [0.0, 0.14753941+0.27970006j, -0.02525158+0.03691013j, -0.13196326-0.25017134j], [-0.14753941-0.27970006j, 0.24, +0.j, 0.0, +0.j], [0.0, +0.j, -0.10100632+0.14764052j, 0.0, +0.j], [-0.02525158+0.03691013j, 0.0, 0.0, +0.j]]])



```

    return qc,state_vector_after_QFT
pass

#Step 3: Non-ideal quantum channel (a
#Implements |k>→eiθ|k>
def Noisy_Quantum_Channel(qc):
    #theta = [0.0, 0.3, 0.0, -0.3] #
    theta=float(input("Enter the thet
    theta = [0.0, theta, 0.0, -theta]
    phase_unitary = Diagonal([
        np.exp(1j * theta[0]), # |00>
        np.exp(1j * theta[1]), # |01>
        np.exp(1j * theta[2]), # |10>
        np.exp(1j * theta[3]), # |11>
    ])
    qc.append(phase_unitary, [0, 1])
    state_vector_after_Noisy_Quantum
    show_statevec(state_vector_after_
    print(qc.draw('mpl'))
    plt.show()
    return qc,state_vector_after_Nois
pass

def Amplitude_Loss_Non_Unitary_Kraus_
    #gamma = 0.2 # loss strength (co
    gamma=float(input("Enter the gamm
    #Amplitude loss is non-unitary, s
    if gamma>0:
        K0 = np.array([[1, 0], [0, np
        K1 = np.array([[0, np.sqrt(ga
            amp_damp = Kraus([K0, K1]) #
            #Apply independently to both
            qc.append(amp_damp, [0])
            qc.append(amp_damp, [1])

    #state_vector_after_Amplitude_Los
    #show_statevec(state_vector_after_
    print(qc.draw('mpl'))
    plt.show()

```

```

|0000000011> (0.7071067811865475+0j)
Figure(203.683x284.278)

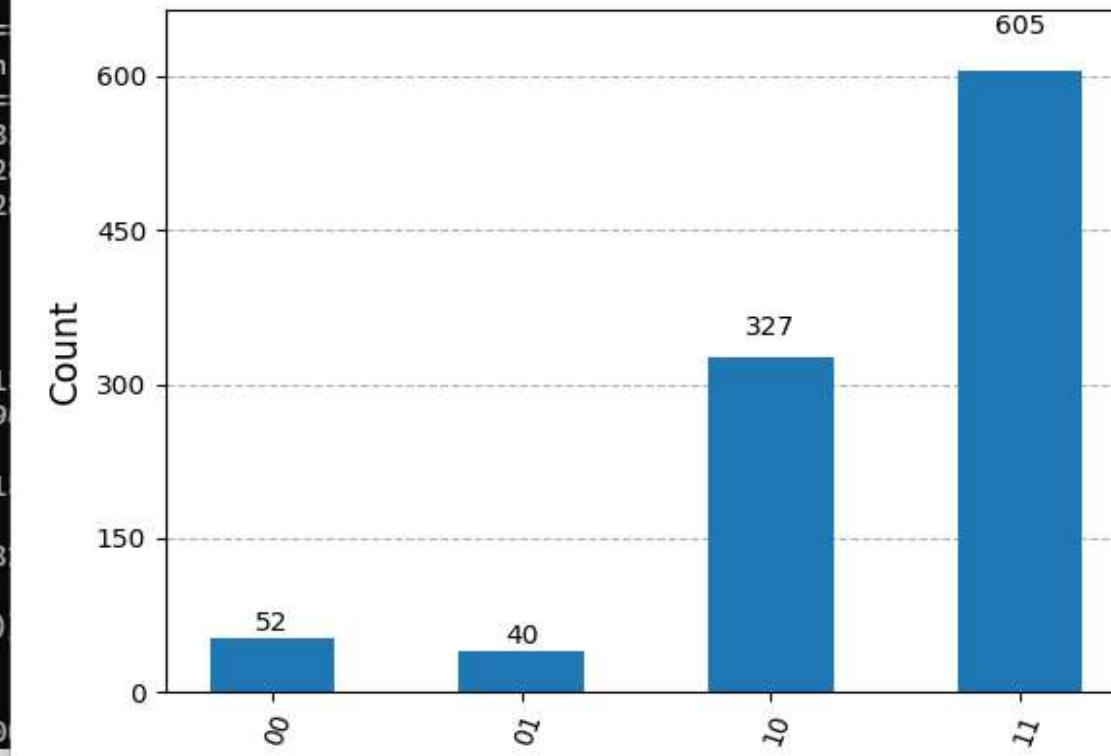
=====
State vector: After QFT
=====

|0000000000> (0.7071067811865474+0j)
|0000000011> (-0.35355332+0j)
|0000000111> (-0.35355333+0j)
Figure 1
Figure(287.294x284.278)
Enter the theta=0.3

=====
State vector: Information
=====

|0000000000> (0.7071067811865474+0j)
|0000000011> (-0.2332802+0j)
|0000000111> (-0.2332802+0j)
Figure(454.517x284.278)
Enter the gamma=0.2
Figure(538.128x284.278)
DensityMatrix([[ 0.56
                  -0.025251
                  [-0.147539
                   0.
                  [-0.025251
                   0.04
                  [-0.131963
                   0.
                  dims=(2, 2)
Figure(621.739x284.278)
Figure(788.961x284.278)
{'10': 327, '11': 605, '0
Measurement results:
10 327
11 605
00 52
01 40

```



```

for outcome, count in counts.items():
    print(outcome, count)
plot_histogram(counts)
plt.show()
return counts
pass

def QBER_Estimation(counts, shots):
    error_counts = sum(v for k, v in
    qber = error_counts / shots
    print("QBER=", qber)
# Parameter ranges
theta = np.linspace(0, np.pi, 120)
gamma = np.linspace(0, 1, 120)

Theta, Gamma = np.meshgrid(theta,
# QBER formula
QBER = (1 - Gamma) * np.sin(Theta)

# Security threshold
QBER_th = 0.11
QBER_plane = QBER_th * np.ones_like(QBER)

# Plot
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
# QBER surface
surf = ax.plot_surface(
    Theta, Gamma, QBER,
    cmap='viridis',
    alpha=0.85,
    edgecolor='none'
)
# Threshold plane
ax.plot_surface(
    Theta, Gamma, QBER_plane,
    color='red',
    alpha=0.35
)
for outcome, count in counts.items():
    print(outcome, count)
plot_histogram(counts)
plt.show()
return counts
pass

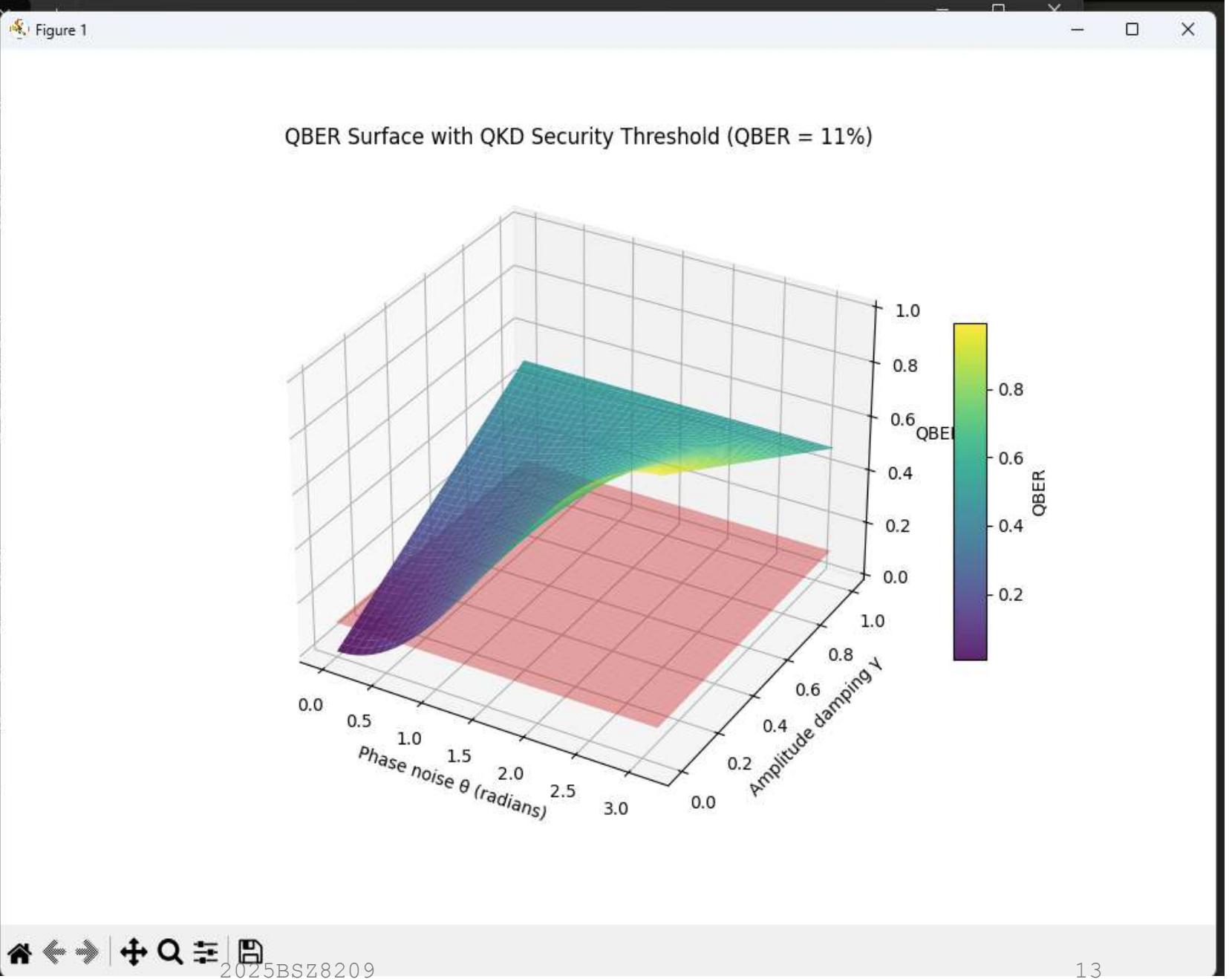
def QBER_Estimation(counts, shots):
    error_counts = sum(v for k, v in
    qber = error_counts / shots
    print("QBER=", qber)
# Parameter ranges
theta = np.linspace(0, np.pi, 120)
gamma = np.linspace(0, 1, 120)

Theta, Gamma = np.meshgrid(theta,
# QBER formula
QBER = (1 - Gamma) * np.sin(Theta)

# Security threshold
QBER_th = 0.11
QBER_plane = QBER_th * np.ones_like(QBER)

# Plot
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
# QBER surface
surf = ax.plot_surface(
    Theta, Gamma, QBER,
    cmap='viridis',
    alpha=0.85,
    edgecolor='none'
)
# Threshold plane
ax.plot_surface(
    Theta, Gamma, QBER_plane,
    color='red',
    alpha=0.35
)

```



Thank You
