Swapping

April 16, 2020

1 Entanglement Swapping

This script swaps entanglement between two spatially seperated qbits that do not come into direct contact with each other. Two qbit tomography and readout calibration are used to find the resulting state, before calculating the fidelity of the output to the expected output.

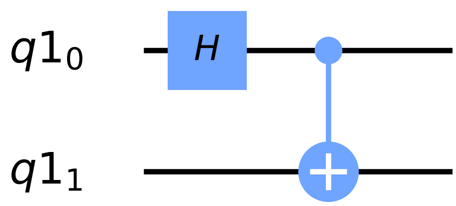
[1]:
```
#importing necessary modules
import numpy as np
import pandas as pd
import random
%config InlineBackend.figure_format = 'svg' # Saves plots in svg format

# qiskit modules
import qiskit
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer
from qiskit.tools.visualization import plot_state_city, plot_state_paulivec
from qiskit.tools.monitor import job_monitor
from qiskit.quantum_info import state_fidelity, concurrence, DensityMatrix

# tomography functions
from qiskit.ignis.verification.tomography import state_tomography_circuits,␣
↪StateTomographyFitter

# readout calibration
from qiskit.ignis.mitigation.measurement import complete_meas_cal,␣
↪CompleteMeasFitter
```

[2]:
```
# importing IBMQ backends
from qiskit import IBMQ
ibmq_provider = IBMQ.load_account() # credentials stored on disk
device = ibmq_provider.get_backend('ibmq_16_melbourne')

# necessary imports for noise modelling
from qiskit.providers.aer.noise import NoiseModel
noise_model = NoiseModel.from_backend(device)
coupling_map = device.configuration().coupling_map
basis_gates = noise_model.basis_gates
```

1.1 Defining the expected state

[3]:
```
# creating a random initial state for 0th qbit
R = []
for i in range(3):
    R.append(10*random.random())

qreg = QuantumRegister(4)
qreg_exp = QuantumRegister(2)
qc_expected = QuantumCircuit(qreg_exp)
# qc_expected.rx((np.pi*R[0])/10,0)
# qc_expected.ry((np.pi*R[1])/10,0)
# qc_expected.rz((np.pi*R[2])/10,0)
qc_expected.h(0)
qc_expected.cx(0,1)
qc_expected.draw(output='mpl')
```

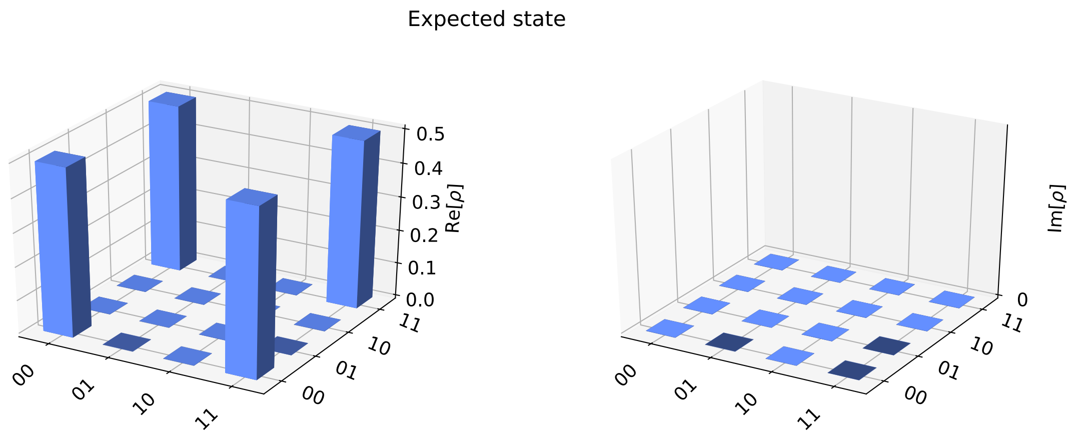
[3]:



[4]:
```
job = qiskit.execute(qc_expected, Aer.get_backend('statevector_simulator'))
psi_expected = job.result().get_statevector(qc_expected)
print('The expected state is:', psi_expected)
plot_state_city(psi_expected, title='Expected state')
```

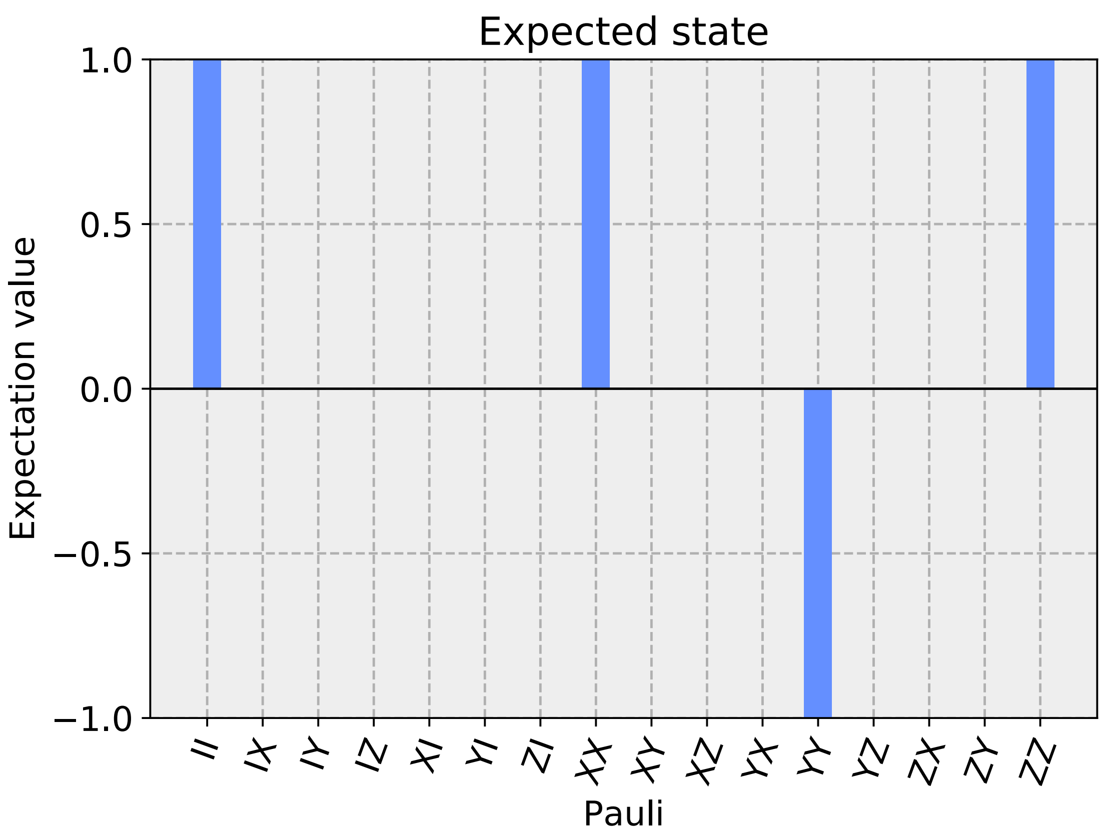
```
The expected state is: [0.70710678+0.j 0.        +0.j 0.        +0.j
 0.70710678+0.j]
```

[4]:



[23]:
```
plot_state_paulivec(psi_expected, title='Expected state')
```

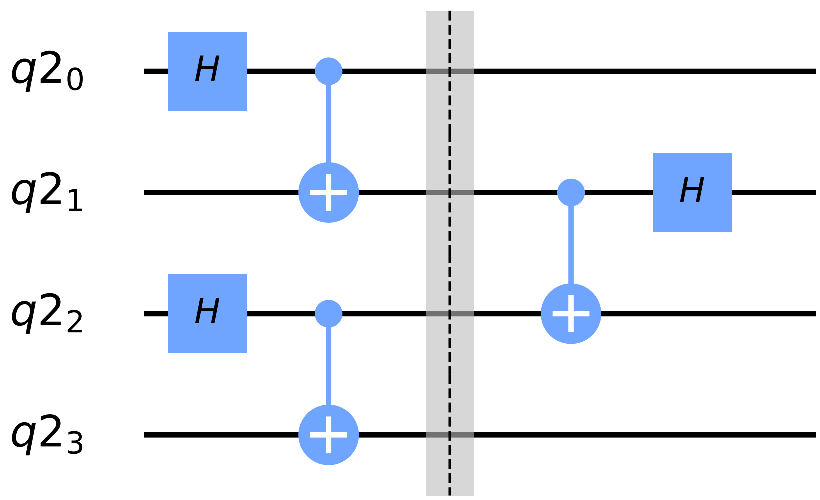
[23]:



1.2 Defining the circuit

[5]:
```
qr = QuantumRegister(4)
cr = ClassicalRegister(4)
qc = QuantumCircuit(qr)
# qc.rx((np.pi*R[0])/10,0)
# qc.ry((np.pi*R[1])/10,0)
# qc.rz((np.pi*R[2])/10,0)
qc.h(0)
qc.h(2)
qc.cx(0,1)
qc.cx(2,3)
qc.barrier()
qc.cx(1,2)
qc.h(1)
qc.draw(output='mpl')
```

[5]:



[6]:
```
# what does this circuit look like on the device?
transpile_qc = qiskit.compiler.transpile(qc,device)
# transpile_qc.draw(output='mpl')
```

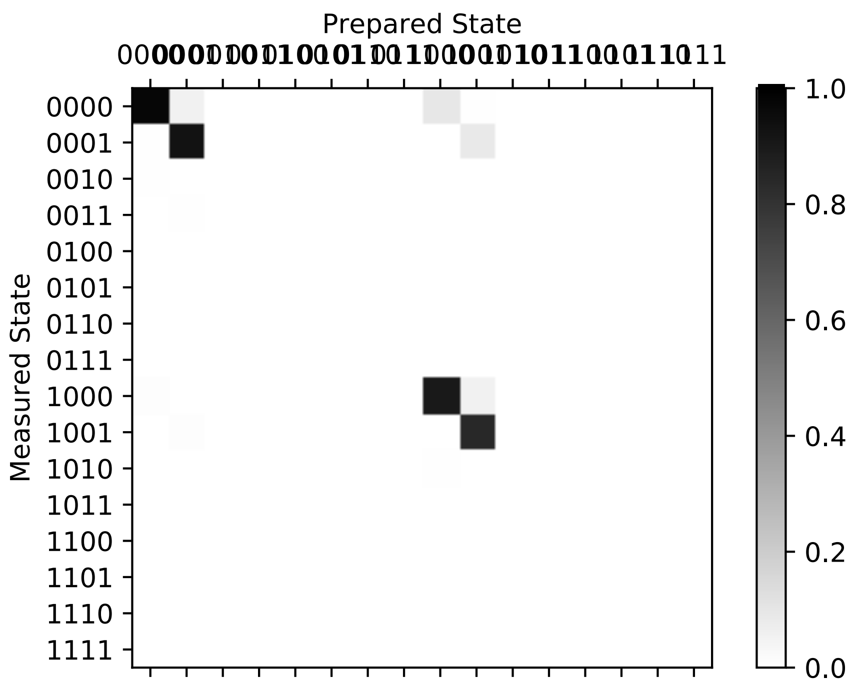
1.3 Readout correction

Calibration is performed on the first and last qbit of the circuit only.

[7]:
```
#readout calibration measurements
clz, state_labels = complete_meas_cal(qr = qc.qregs[0], circlabel =␣
 ↪'measerrormitcal')
```

```python
# doing calibration only for qbit 0 and 3, for 00, 10, 01, 11
cal_circuit = [clz[0], clz[8], clz[1], clz[9]]
cal_job = qiskit.execute(cal_circuit, backend = device, shots = 8192,
    optimization_level = 0)
job_monitor(cal_job)
calres = cal_job.result()
meas_fitter = CompleteMeasFitter(calres, state_labels)
meas_fitter.plot_calibration()
```

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



[8]:
```python
bits = ['0000', '1000', '0001', '1001',
        '0010', '1010', '0011', '1011',
        '0100', '1100', '0101', '1101',
        '0110', '1110', '0111', '1111']

counts = [0,0,0,0]
for i in range(4):
    counts[i] = calres.get_counts(cal_circuit[i])
    for bit in bits:
        if bit not in counts[i]:
            counts[i][bit] = 0

m_IZ = [0,0,0,0]
```

```
m_ZI = [0,0,0,0]
m_ZZ = [0,0,0,0]
for i in range(4):
    for num, bit in enumerate(bits):
        if num % 4 == 0 or num % 4 == 1:
            m_IZ[i] += float(counts[i][bit]) / 8192
        if num % 4 == 2 or num % 4 == 3:
            m_IZ[i] -= float(counts[i][bit]) / 8192
        if num % 4 == 0 or num % 4 == 2:
            m_ZI[i] += float(counts[i][bit]) / 8192
        if num % 4 == 1 or num % 4 == 3:
            m_ZI[i] -= float(counts[i][bit]) / 8192
        if num % 4 == 0 or num % 4 == 3:
            m_ZZ[i] += float(counts[i][bit]) / 8192
        if num % 4 == 1 or num % 4 == 2:
            m_ZZ[i] -= float(counts[i][bit]) / 8192

M = [[1,  1,  1,  1],
     [1,  1, -1, -1],
     [1, -1,  1, -1],
     [1, -1, -1,  1]]
Minv = np.linalg.inv(M)

beta_IZ = np.dot(Minv, m_IZ)
beta_ZI = np.dot(Minv, m_ZI)
beta_ZZ = np.dot(Minv, m_ZZ)

b_vec = [beta_IZ[0], beta_ZI[0], beta_ZZ[0]]
B = [beta_IZ[1:], beta_ZI[1:], beta_ZZ[1:]]
Binv = np.linalg.inv(B)
```

1.4 State Tomography

In order to reconstruct the output state of the circuit, it is necessary to perform 2 qbit tomography to build an arbitrary state from the outcomes of a series of measurements that together give the full state. For 2 qbits, that is 9 (3^n) circuits.

[9]:
```
# defining tomography circuits
qcz = state_tomography_circuits(qc,qr)
# qiskit creates all 81 tomography circuits for 4 qbits, we want just a subset␣
↪of that:
# the nine circuits are in the order [XZZX, XZZY, XZZZ, YZZX, YZZY, YZZZ, ZZZX,␣
↪ZZZY, ZZZZ]
qst_circuit =␣
↪[qcz[24],qcz[25],qcz[26],qcz[51],qcz[52],qcz[53],qcz[78],qcz[79],qcz[80]]
```

[10]:
```
backends = ['simulator', 'simulator_noise', device]

paulis = [('X','Z','Z','X'),('Y','Z','Z','X'),('Z','Z','Z','X'),
          ('X','Z','Z','Y'),('Y','Z','Z','Y'),('Z','Z','Z','Y'),
          ('X','Z','Z','Z'),('Y','Z','Z','Z'),('Z','Z','Z','Z')]

basis = ['XX','YX','ZX',
         'XY','YY','ZY',
         'XZ','YZ','ZZ']

for num, backend in enumerate(backends):
    state ={}
    for bas in basis:
        state.update({bas: {'00': 0, '01': 0, '10': 0, '11': 0}})
    state_df = pd.DataFrame(data=[state,state,state,state])

    if num == 0:
        job_sim = qiskit.execute(qst_circuit, Aer.
↪get_backend('qasm_simulator'), shots=8192)
        tom = StateTomographyFitter(job_sim.result(), qst_circuit)
    if num == 1:
        job_sim = qiskit.execute(qst_circuit, Aer.
↪get_backend('qasm_simulator'), shots=8192,
                                 noise_model=noise_model,
                                 coupling_map=coupling_map,
                                 basis_gates=basis_gates)
        tom = StateTomographyFitter(job_sim.result(), qst_circuit)
    if num == 2:
        job_dev = qiskit.execute(qst_circuit, backend, shots=8192)
        job_monitor(job_dev)
        tom = StateTomographyFitter(job_dev.result(), qst_circuit)

    for pauli in paulis:
        for bit in bits:
            if bit not in tom.data[pauli]:
                tom.data[pauli][bit] = 0

    # 4 possible outcomes of teleport protocol, where the final state is either
    # psi, IX @ psi, IZ @ psi, IZ @ IX @ psi. We store all 4 seperately, to␣
↪then transform
    # the resulting density matrices appropriately before finding fidelity
    for i in range(4):
        for j in range(9):
            state_df.loc[i][basis[j]] = {'00': tom.data[(paulis[j])][bits[4*i]],
                                         '01': tom.
↪data[(paulis[j])][bits[4*i+2]],
```

```
                                                '10': tom.
↪data[(paulis[j])][bits[4*i+1]],
                                                '11': tom.
↪data[(paulis[j])][bits[4*i+3]]}

    if num == 0:
        state_sim = state_df
        print('Simulator done.')
    if num == 1:
        state_sim_noise = state_df
        print('Noisy simulator done.')
    if num == 2:
        state_dev = state_df
        print('Device done.')
```

```
Simulator done.
Noisy simulator done.
Job Status: job has successfully run
Device done.
```

[11]:

```
# defining the matrices to construct the density matrices
I_matrix = np.matrix('1, 0; 0, 1')
X_matrix = np.matrix('0, 1; 1, 0')
Y_matrix = np.matrix('0, 0-1j; 0+1j, 0')
Z_matrix = np.matrix('1, 0; 0, -1')

Iden = np.kron(I_matrix,I_matrix)

IXPaul = np.kron(I_matrix,X_matrix)
IYPaul = np.kron(I_matrix,Y_matrix)
IZPaul = np.kron(I_matrix,Z_matrix)
XIPaul = np.kron(X_matrix,I_matrix)
YIPaul = np.kron(Y_matrix,I_matrix)
ZIPaul = np.kron(Z_matrix,I_matrix)

XXPaul = np.kron(X_matrix,X_matrix)
XYPaul = np.kron(X_matrix,Y_matrix)
XZPaul = np.kron(X_matrix,Z_matrix)

YXPaul = np.kron(Y_matrix,X_matrix)
YYPaul = np.kron(Y_matrix,Y_matrix)
YZPaul = np.kron(Y_matrix,Z_matrix)

ZXPaul = np.kron(Z_matrix,X_matrix)
ZYPaul = np.kron(Z_matrix,Y_matrix)
ZZPaul = np.kron(Z_matrix,Z_matrix)
```

[12]:

```
# there are 4 density matrices to calculate for each of sim, sim_noise and dev

# XX, XY, XZ etc are 9 total r_xx etc terms to calculate
r_exp = np.zeros((4, 9))
# there are three IX type terms and each can be calculated three ways (from XX,␣
↪YX, ZX for IX)
r_IP = np.zeros((4, 9))
r_PI = np.zeros((4, 9))
r_IPavg = np.zeros((4, 3))
r_PIavg = np.zeros((4, 3))

m = np.zeros((4,9,3))
r_corr = np.zeros((4,9,3))
r_IPavg_corr = np.zeros((4, 3))
r_PIavg_corr = np.zeros((4, 3))

state_tot=[state_sim,state_sim_noise,state_dev]
rho=[0,0,0,0]

for stat, states in enumerate(state_tot):
    rhof = [0,0,0,0]
    for i in range(4):
        for j in range(9):
            r_exp[i,j] = (states[basis[j]][i]['00']*1 +␣
↪states[basis[j]][i]['10']*(-1) + states[basis[j]][i]['01']*(-1) +␣
↪states[basis[j]][i]['11']*1)/(states[basis[j]][i]['00']*1 +␣
↪states[basis[j]][i]['10']*(1) + states[basis[j]][i]['01']*(1) +␣
↪states[basis[j]][i]['11']*1)
            r_IP[i,j] = (states[basis[j]][i]['00']*1 +␣
↪states[basis[j]][i]['10']*(1) + states[basis[j]][i]['01']*(-1) +␣
↪states[basis[j]][i]['11']*(-1))/(states[basis[j]][i]['00']*1 +␣
↪states[basis[j]][i]['10']*(1) + states[basis[j]][i]['01']*(1) +␣
↪states[basis[j]][i]['11']*1)
            r_PI[i,j] = (states[basis[j]][i]['00']*1 +␣
↪states[basis[j]][i]['10']*(-1) + states[basis[j]][i]['01']*(1) +␣
↪states[basis[j]][i]['11']*(-1))/(states[basis[j]][i]['00']*1 +␣
↪states[basis[j]][i]['10']*(1) + states[basis[j]][i]['01']*(1) +␣
↪states[basis[j]][i]['11']*1)

    for i in range(4):
        for j in range(3):
                r_IPavg[i,j]  = (r_IP[i,j] + r_IP[i,j+3] + r_IP[i,j+6])/3
                r_PIavg[i,j] = (r_PI[i,3*j] + r_PI[i,3*j+1] + r_PI[i,3*j+2])/3
    for i in range(4):
        rhof[i] = (1/4)*(Iden + r_exp[i,0]*XXPaul + r_exp[i,1]*XYPaul +␣
↪r_exp[i,2]*XZPaul
```

```python
                              + r_exp[i,3]*YXPaul + r_exp[i,4]*YYPaul + r_exp[i,5]*YZPaul
                              + r_exp[i,6]*ZXPaul + r_exp[i,7]*ZYPaul + r_exp[i,8]*ZZPaul
                              + r_IPavg[i,0]*IXPaul + r_PIavg[i,0]*XIPaul
                              + r_IPavg[i,1]*IYPaul + r_PIavg[i,1]*YIPaul
                              + r_IPavg[i,2]*IZPaul + r_PIavg[i,2]*ZIPaul)
        rho[stat] = rhof

# performing corrections for readout error and storing result in another density matrix
for i in range(4):
    for num, bas in enumerate(basis):
        m[i,num,:] = [r_IP[i,num], r_PI[i, num], r_exp[i,num]]
        r_corr[i,num] = np.dot(Binv, np.subtract(m[i,num,:],b_vec))

rho_corr = [0,0,0,0]
for i in range(4):
    for j in range(3):
            r_IPavg_corr[i,j] = (r_corr[i,j][0] + r_corr[i,j+3][0] + r_corr[i,j+6][0])/3
            r_PIavg_corr[i,j] = (r_corr[i,3*j][1] + r_corr[i,3*j+1][1] + r_corr[i,3*j+2][1])/3

    rho_corr[i] = (1/4)*(Iden + r_corr[i,0][2]*XXPaul + r_corr[i,1][2]*XYPaul + r_corr[i,2][2]*XZPaul
                              + r_corr[i,3][2]*YXPaul + r_corr[i,4][2]*YYPaul + r_corr[i,5][2]*YZPaul
                              + r_corr[i,6][2]*ZXPaul + r_corr[i,7][2]*ZYPaul + r_corr[i,8][2]*ZZPaul
                              + r_IPavg_corr[i,0]*IXPaul + r_PIavg_corr[i,0]*XIPaul
                              + r_IPavg_corr[i,1]*IYPaul + r_PIavg_corr[i,1]*YIPaul
                              + r_IPavg_corr[i,2]*IZPaul + r_PIavg_corr[i,2]*ZIPaul)

rho[3] = rho_corr

# applying the corrections for the 3 outcomes for which an X, Z or both gates are required
for ind in range(4):
    rho[ind][1] = ZIPaul @ rho[ind][1] @ ZIPaul
    rho[ind][2] = XIPaul @ rho[ind][2] @ XIPaul
    rho[ind][3] = XIPaul @ IZPaul @ rho[ind][3] @ IZPaul @ XIPaul
```

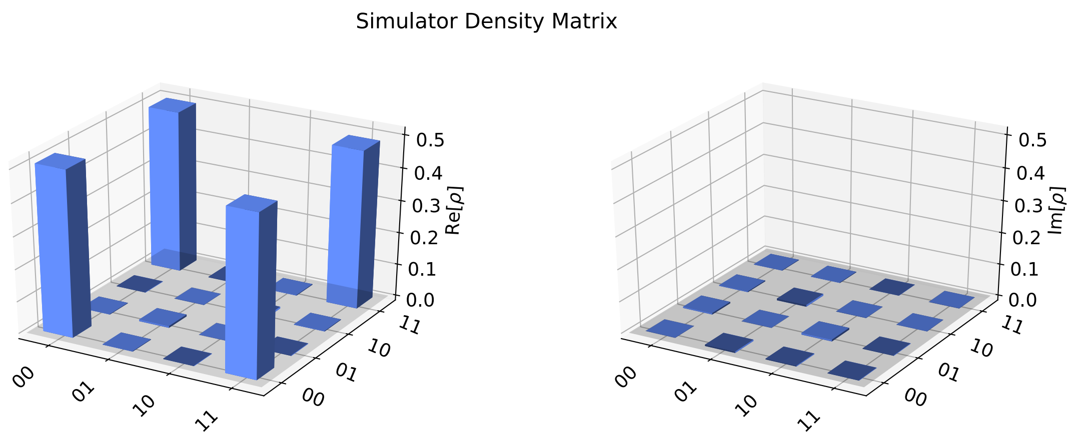
1.5 Calculating Fidelities

[13]:
```
Fidelity = np.zeros((4,4))
for i in range(4):
    for j in range(4):
        Fidelity[i,j] = state_fidelity(psi_expected, rho[i][j], validate=False)

F = np.mean(Fidelity,axis=1)
print('Simulator Fidelity:', F[0])
print('Noisy Simulator Fidelity:', F[1])
print('Device Fidelity:', F[2])
print('Calibrated Device Fidelity', F[3])
```

```
Simulator Fidelity: 1.0
Noisy Simulator Fidelity: 0.8035585002009553
Device Fidelity: 0.7627657648218437
Calibrated Device Fidelity 0.8656129082241617
```

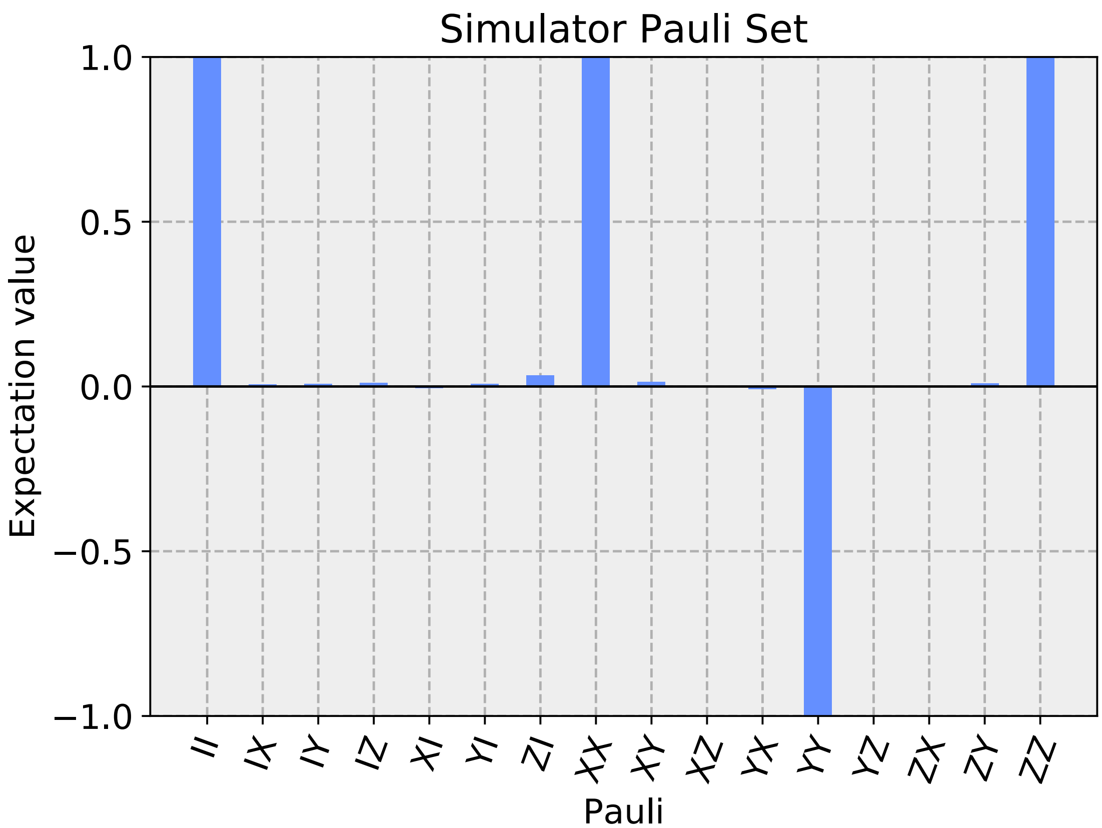
[14]:
```
plot_state_city(rho[0][0], title='Simulator Density Matrix')
```

[14]:



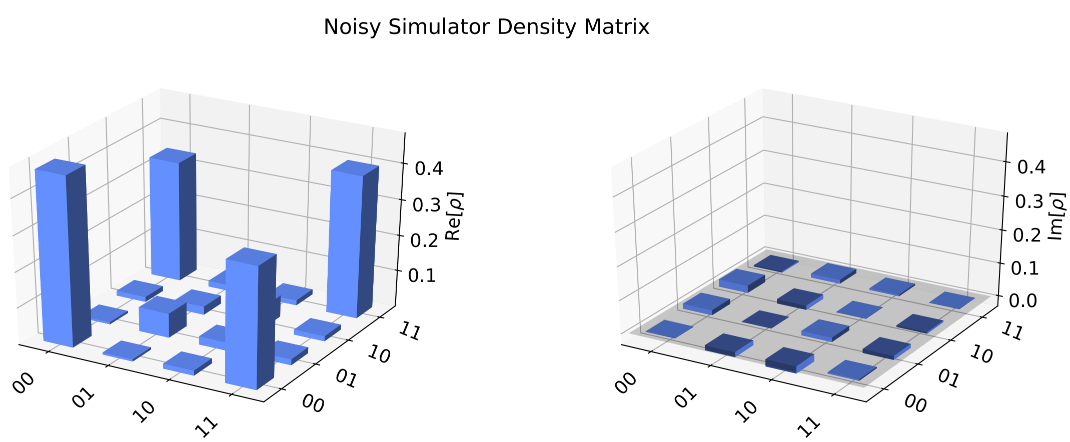
[15]:
```
plot_state_paulivec(rho[0][0], title='Simulator Pauli Set')
```

[15]:



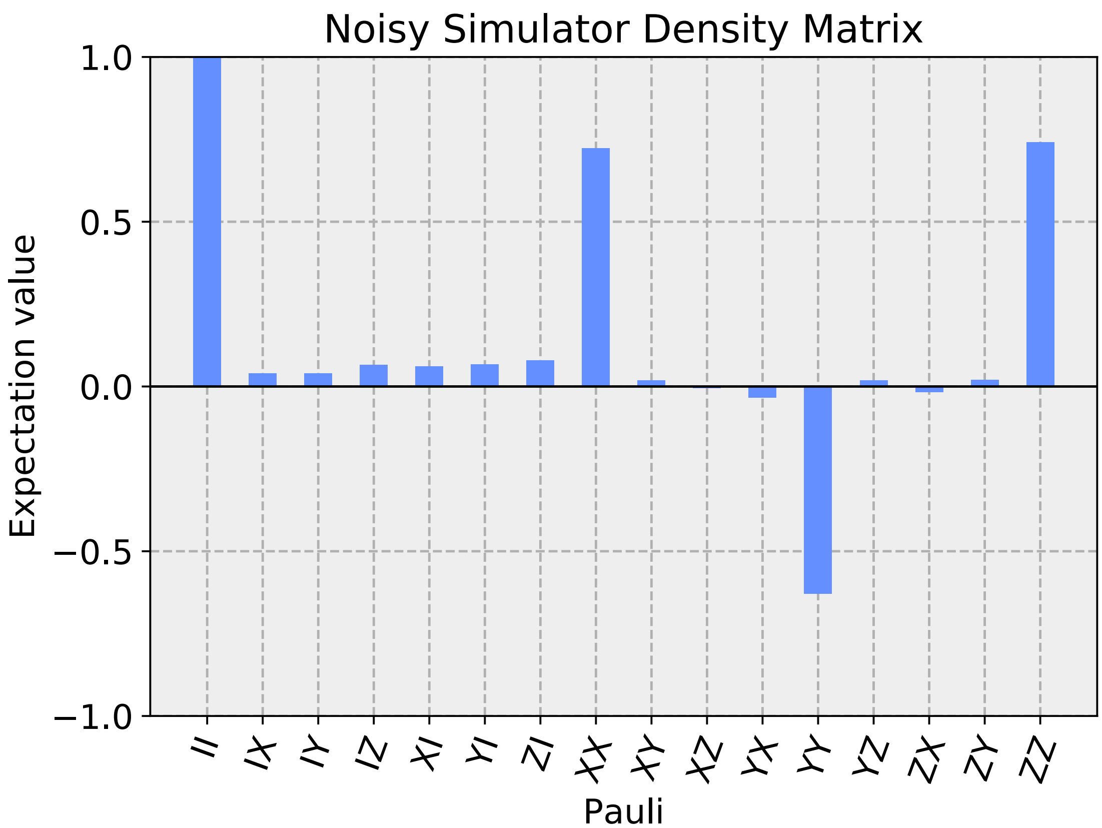

```
[16]: plot_state_city(rho[1][0], title='Noisy Simulator Density Matrix')
```

[16]:



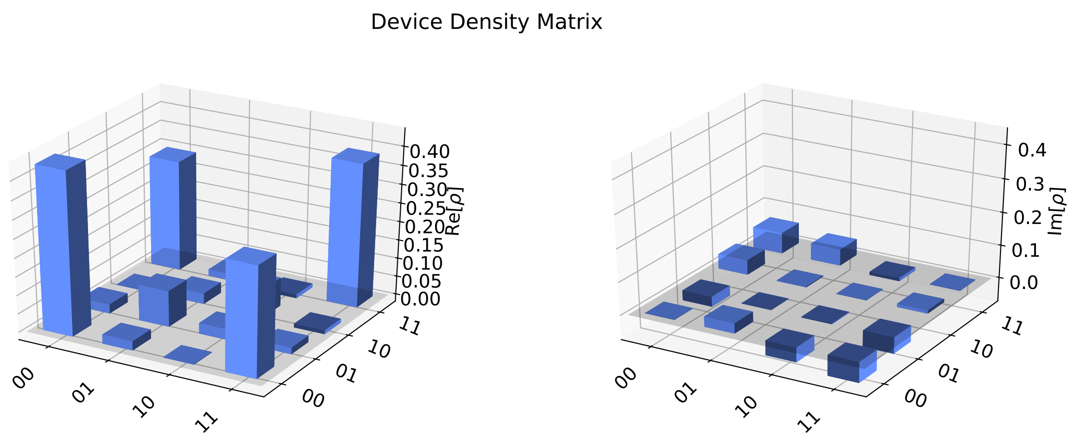

```
[17]: plot_state_paulivec(rho[1][0], title='Noisy Simulator Density Matrix')
```

[17]:



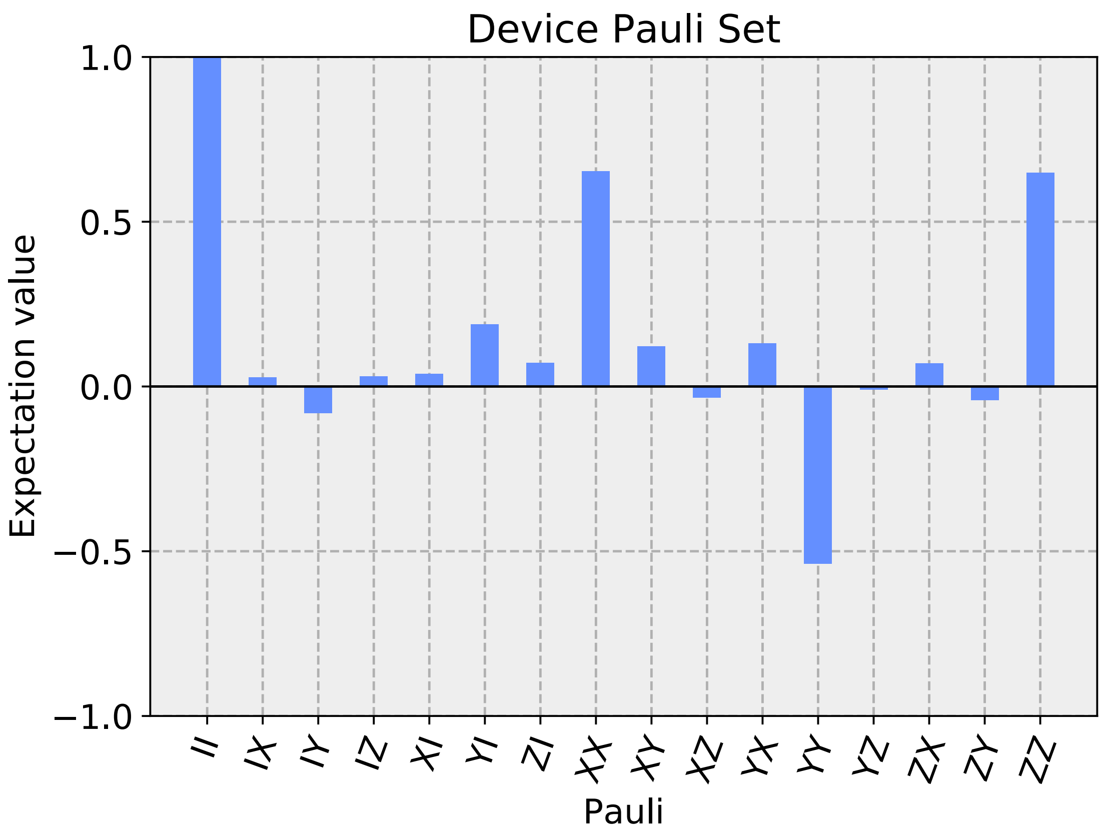
[18]:
```
plot_state_city(rho[2][0], title='Device Density Matrix')
```

[18]:



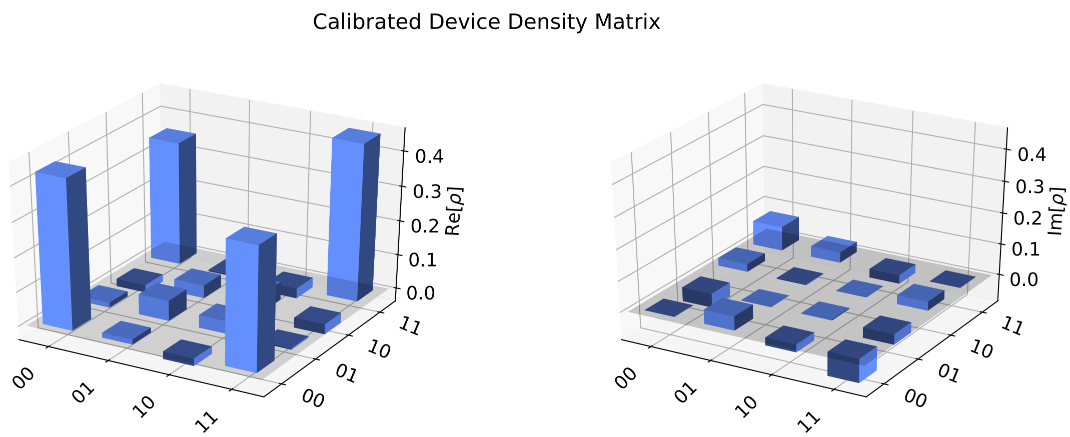
[19]:
```
plot_state_paulivec(rho[2][0], title='Device Pauli Set')
```

[19]:



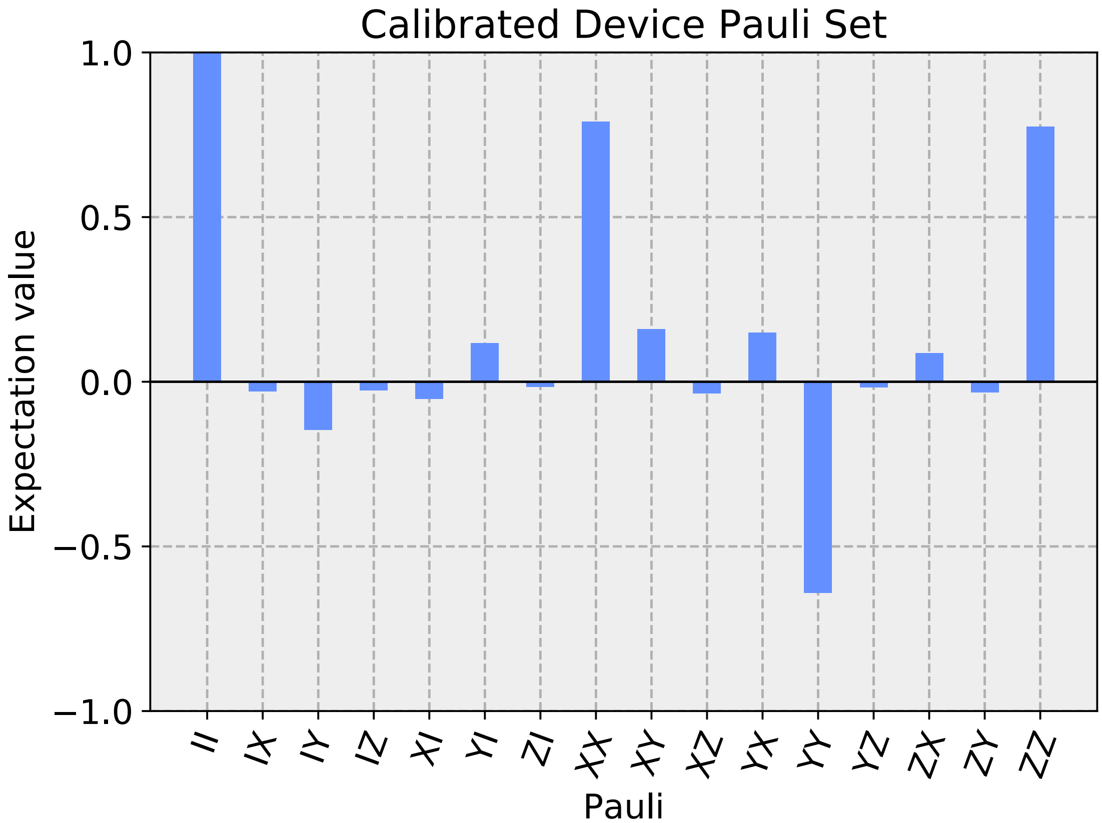
[20]:
```
plot_state_city(rho[3][0], title='Calibrated Device Density Matrix')
```

[20]:



[21]:
```
plot_state_paulivec(rho[3][0], title='Calibrated Device Pauli Set')
```

[21]:



Calibrated Device Pauli Set
1.0
0.5
0.0
−0.5
−1.0
Expectation value
II
IX
IY
IZ
XI
YI
ZI
XX
XY
XZ
YX
YY
YZ
ZX
ZY
ZZ
Pauli