

Week 1: Quantum Measurement Dataset

Foundations

Project: Machine Learning for Quantum State Tomography

Task 1 & 2: Environment and Theory

The goal is to build a workflow to reconstruct quantum states. I am using **Pauli Projective Measurements** (X, Y, Z bases) instead of SIC-POVMs because they are easier to implement in Qiskit.

Born Rule: The probability of an outcome k is $p(k) = \text{Tr}(M_k \rho)$. I will measure in 3 bases to reconstruct the density matrix ρ .

```
# Installing the required libraries as per the "Required baseline packages"
!pip install qiskit qiskit-aer pennylane numpy scipy pandas plotly tqdm nbfo
```

Show hidden output

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
from qiskit.quantum_info import DensityMatrix, state_fidelity, random_statevector
import plotly.graph_objects as go

# Defining the Pauli Matrices for reconstruction later
I = np.array([[1, 0], [0, 1]], dtype=complex)
X = np.array([[0, 1], [1, 0]], dtype=complex)
Y = np.array([[0, -1j], [1j, 0]], dtype=complex)
Z = np.array([[1, 0], [0, -1]], dtype=complex)

print("Environment ready. Pauli operators loaded.")
```

Environment ready. Pauli operators loaded.

Visualization Helper

This is the code provided in the assignment brief to plot the density matrices as 3D histograms.

helper functions for density matrix visualization

```
#@title helper functions for density matrix visualization
```

```
import numpy as np
import plotly.graph_objects as go
from fractions import Fraction

_CUBE_FACES = (
    (0, 1, 2), (0, 2, 3), # bottom
    (4, 5, 6), (4, 6, 7), # top
    (0, 1, 5), (0, 5, 4),
    (1, 2, 6), (1, 6, 5),
    (2, 3, 7), (2, 7, 6),
    (3, 0, 4), (3, 4, 7)
)

def _phase_to_pi_string(angle_rad: float) -> str:
    """Format a phase angle as a simplified multiple of π."""
    if np.isclose(angle_rad, 0.0):
        return "0"
    multiple = angle_rad / np.pi
    frac = Fraction(multiple).limit_denominator(16)
    numerator = frac.numerator
    denominator = frac.denominator
    sign = "-" if numerator < 0 else ""
    numerator = abs(numerator)
    if denominator == 1:
        magnitude = f"{numerator}" if numerator != 1 else ""
    else:
        magnitude = f"{numerator}/{denominator}"
    return f"{sign}{magnitude}π" if magnitude else f"{sign}π"

def plot_density_matrix_histogram(rho, basis_labels=None, title="Density mat
    """Render a density matrix as a grid of solid histogram bars with phase
    rho = np.asarray(rho)
    if rho.ndim != 2 or rho.shape[0] != rho.shape[1]:
        raise ValueError("rho must be a square matrix")

    dim = rho.shape[0]
    mags = np.abs(rho)
    phases = np.angle(rho)
    x_vals = np.arange(dim)
    y_vals = np.arange(dim)

    if basis_labels is None:
        basis_labels = [str(i) for i in range(dim)]

    meshes = []
    colorbar_added = False
    for i in range(dim):
        for j in range(dim):
            height = mags[i, j]
            phase = phases[i, j]
            x0, x1 = i - 0.45, i + 0.45
            y0, y1 = j - 0.45, j + 0.45
            vertices = (
                (x0, y0, 0.0), (x1, y0, 0.0), (x1, y1, 0.0), (x0, y1, 0.0),
                (x0, y0, height), (x1, y0, height), (x1, y1, height), (x0, y
```

```
)  
x_coords, y_coords, z_coords = zip(*vertices)  
i_idx, j_idx, k_idx = zip(*_CUBE_FACES)  
phase_pi = _phase_to_pi_string(phase)  
mesh = go.Mesh3d(  
    x=x_coords,  
    y=y_coords,  
    z=z_coords,  
    i=i_idx,  
    j=j_idx,  
    k=k_idx,  
    intensity=[phase] * len(vertices),  
    colorscale="HSV",  
    cmin=-np.pi,  
    cmax=np.pi,  
    showscale=not colorbar_added,  
    colorbar=dict(  
        title="phase ",  
        tickvals=[-np.pi, -np.pi/2, 0, np.pi/2, np.pi],  
        ticktext=["- $\pi$ ", "- $\pi/2$ ", "0", " $\pi/2$ ", " $\pi$ "]  
    ) if not colorbar_added else None,  
    opacity=1.0,  
    flatshading=False,  
    hovertemplate=  
        f"i={i}, j={j}  
        |\rho_{ij}|={height:.3f}  
        arg(\rho_{ij})={phase}  
        lighting=dict(ambient=0.6, diffuse=0.7)  
    ),  
    meshes.append(mesh)  
    colorbar_added = True  
  
fig = go.Figure(data=meshes)  
fig.update_layout(  
    scene=dict(  
        xaxis=dict(  
            title="i",  
            tickmode="array",  
            tickvals=x_vals,  
            ticktext=basis_labels  
        ),  
        yaxis=dict(  
            title="j",  
            tickmode="array",  
            tickvals=y_vals,  
            ticktext=basis_labels  
        ),  
        zaxis=dict(title="|\rho_{ij}|"),  
        aspectratio=dict(x=1, y=1, z=0.7)  
    ),  
    title=title,  
    margin=dict(l=0, r=0, b=0, t=40)  
)  
  
fig.show()
```

Task 3 & 4: Data Generation and Tomography

I am generating a dataset containing:

1. **Reference States:** $|0\rangle, |1\rangle, |+\rangle, |-\rangle, |+i\rangle$.
2. **Random Circuits:** 5 completely random states (as requested in the "Task Roadmap" to extend the pipeline).

For each state, I measure it in X, Y, and Z bases, then use **Linear Inversion** to calculate ρ .

```
from qiskit import transpile

simulator = AerSimulator()
shots = 1024 # Standard shot count
dataset = []

# Helper to rotate basis
def measure_in_basis(qc, basis):
    meas_qc = qc.copy()
    if basis == "X":
        meas_qc.h(0)
    elif basis == "Y":
        meas_qc.sdg(0); meas_qc.h(0)
    meas_qc.measure_all()
    return meas_qc

# 1. Standard States
ref_states = ["0", "1", "+", "-", "+i"]
for name in ref_states:
    qc = QuantumCircuit(1)
    if name == "1": qc.x(0)
    elif name == "+": qc.h(0)
    elif name == "-": qc.x(0); qc.h(0)
    elif name == "+i": qc.h(0); qc.s(0)

    # Get ground truth
    rho_true = DensityMatrix(qc).data
    dataset.append({"name": name, "qc": qc, "rho_true": rho_true})

# 2. Random States
print("Generating random states...")
for i in range(5):
    rand_vec = random_statevector(2)
    qc = QuantumCircuit(1)
    qc.initialize(rand_vec, 0)

    rho_true = DensityMatrix(rand_vec).data
    dataset.append({"name": f"Random_{i+1}", "qc": qc, "rho_true": rho_true})

#Processing & Plotting
print(f"{'State':<10} | {'Fidelity':<10}")
print("-" * 25)
```

```
final_data = []

for item in dataset:
    probs = {}
    # Run simulation
    for basis in ["X", "Y", "Z"]:
        m_qc = measure_in_basis(item["qc"], basis)

        t_qc = transpile(m_qc, simulator)

        res = simulator.run(t_qc, shots=shots).result()
        probs[basis] = res.get_counts().get("0", 0) / shots

    # Reconstruction (Linear Inversion)
    exp_x = 2 * probs["X"] - 1
    exp_y = 2 * probs["Y"] - 1
    exp_z = 2 * probs["Z"] - 1
    rho_recon = 0.5 * (I + exp_x * X + exp_y * Y + exp_z * Z)

    # Validation
    fid = state_fidelity(item["rho_true"], rho_recon, validate=False)

    print(f"{item['name']:<10} | {fid:.4f}")

    # Plotting
    try:
        plot_density_matrix_histogram(rho_recon, title=f'Reconstructed {item["name"]}')
    except:
        pass

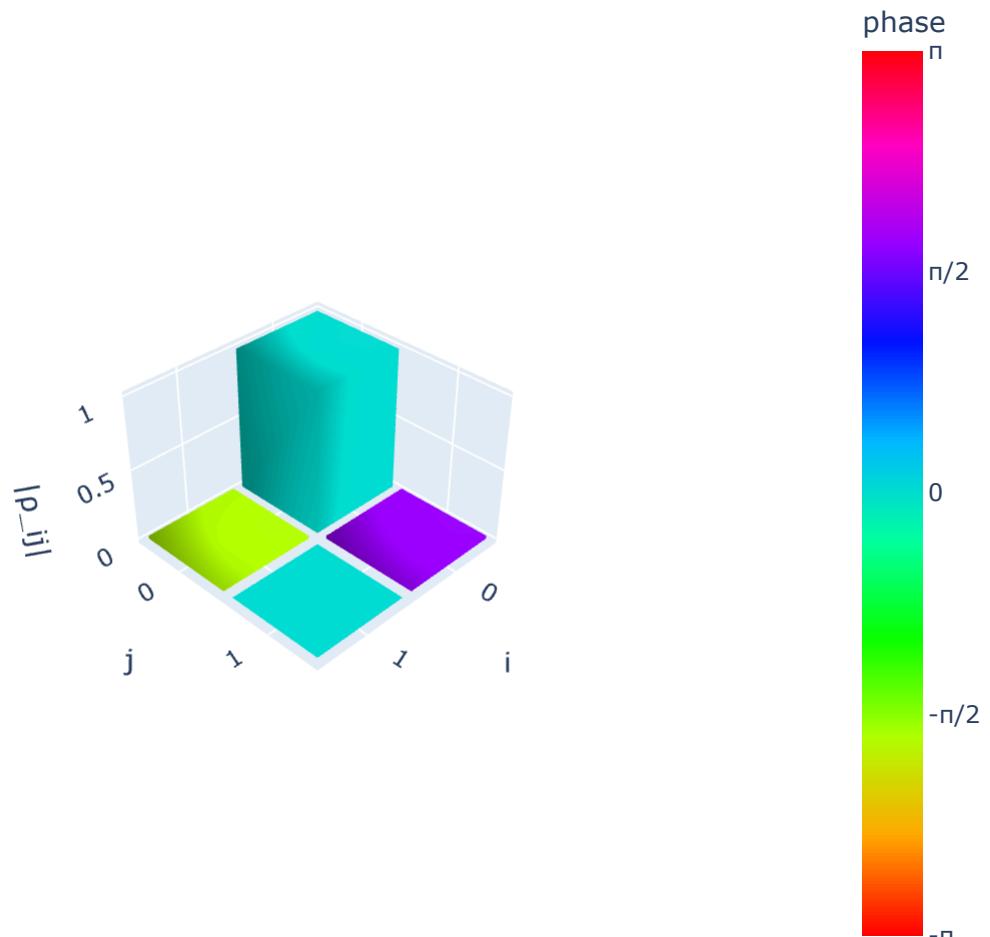
final_data.append({"state": item["name"], "rho_recon": rho_recon, "fidel": fid})
```


Generating random states...

State | Fidelity

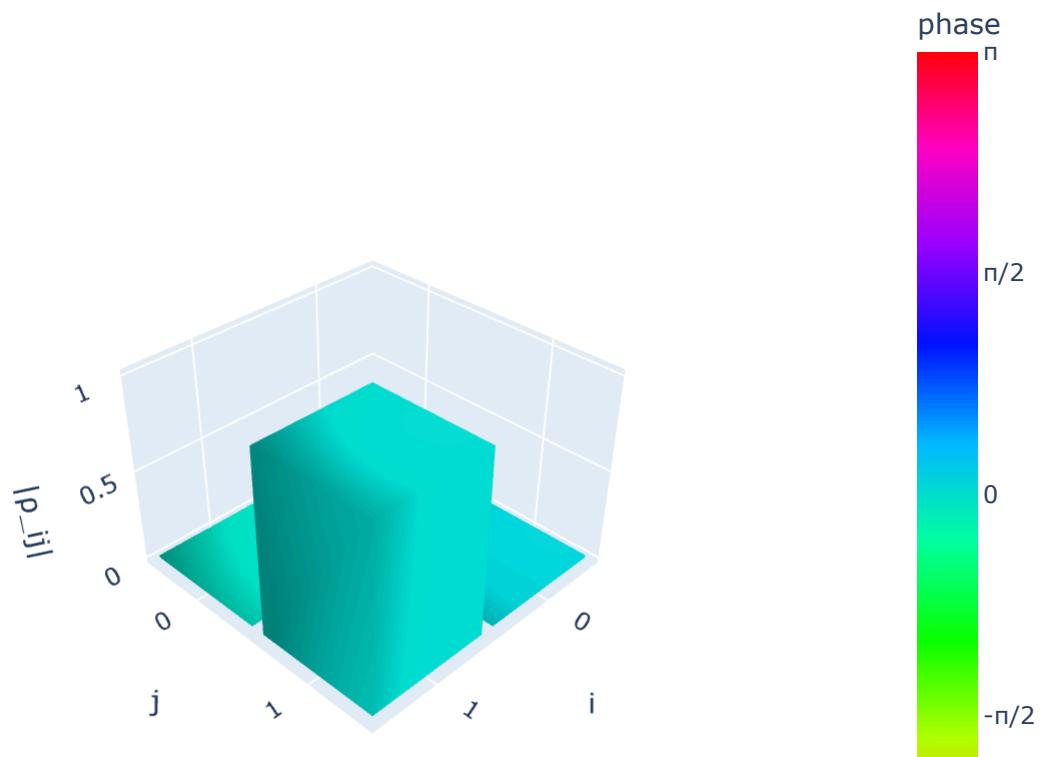
0 | 1.0000

Reconstructed 0 (F=1.000)

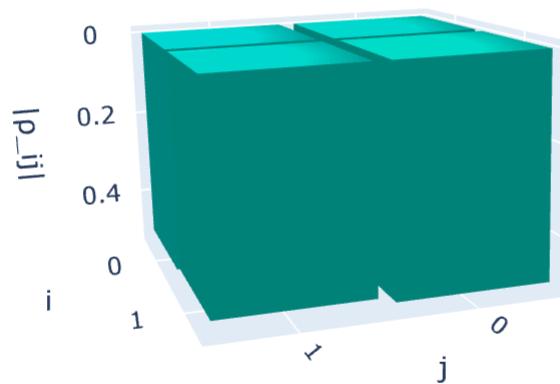


1 | 1.0000

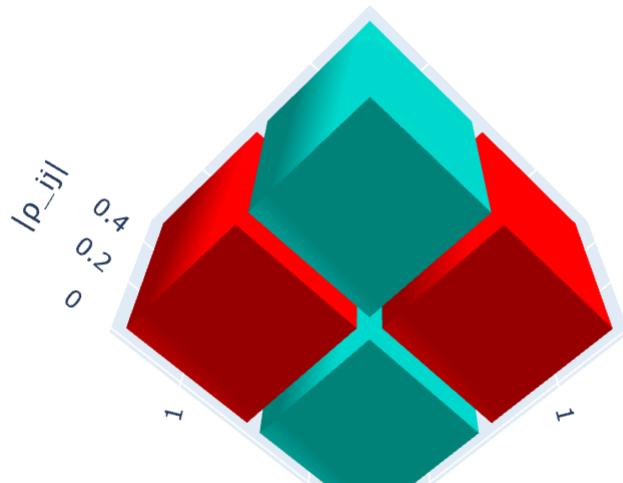
Reconstructed 1 (F=1.000)

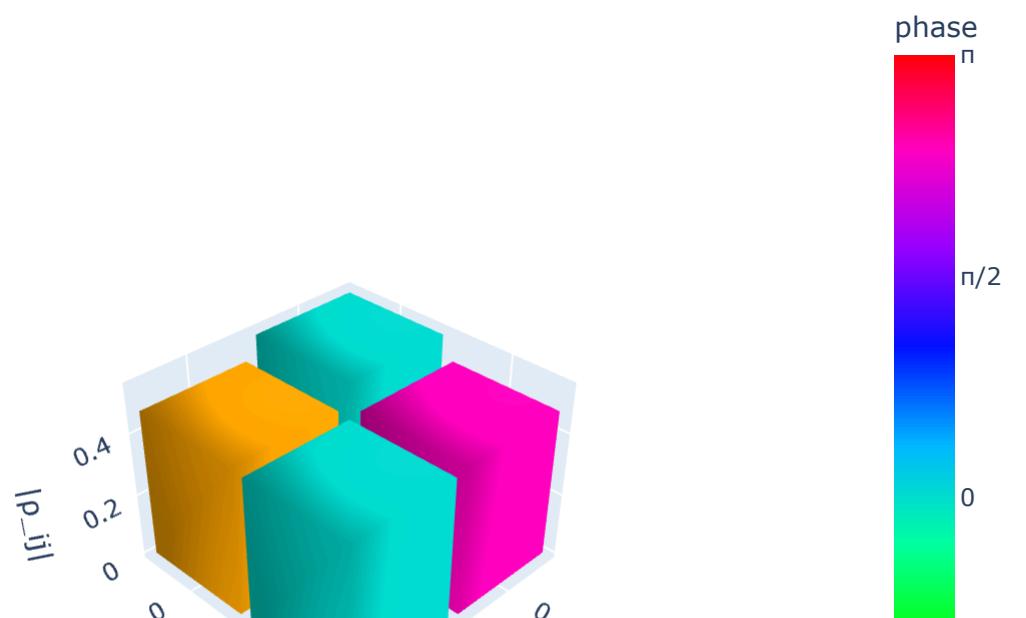
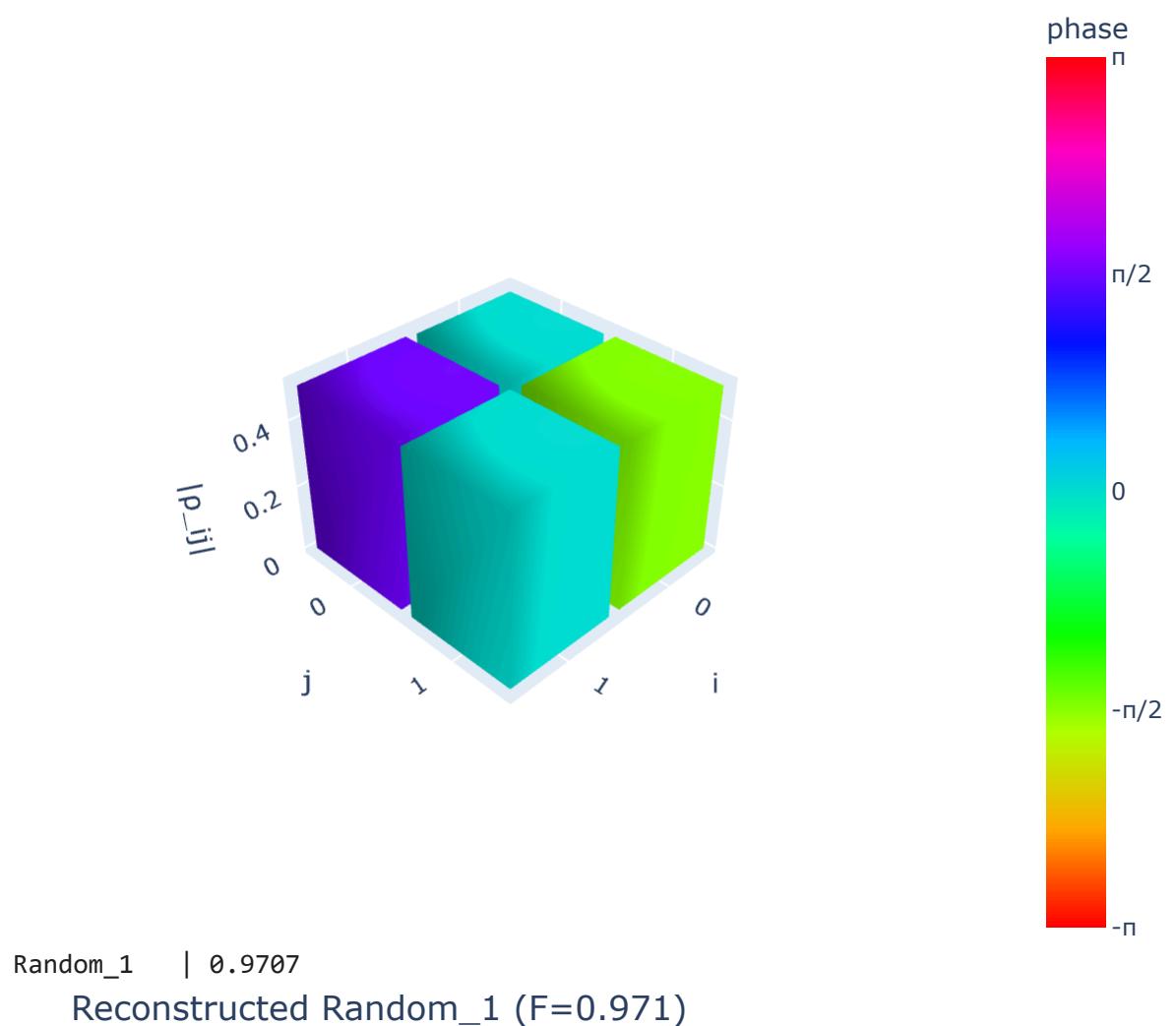


+ | 1.0000
Reconstructed + (F=1.000)



- | 1.0000
Reconstructed - (F=1.000)

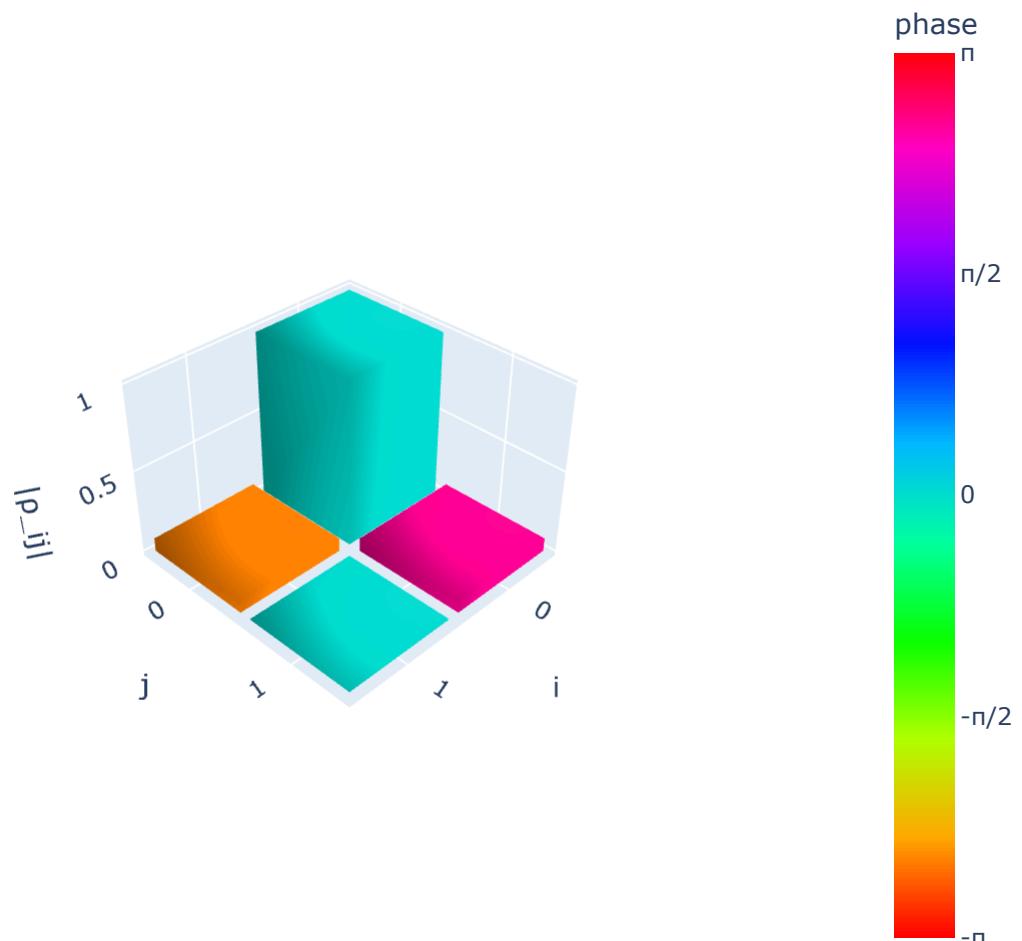






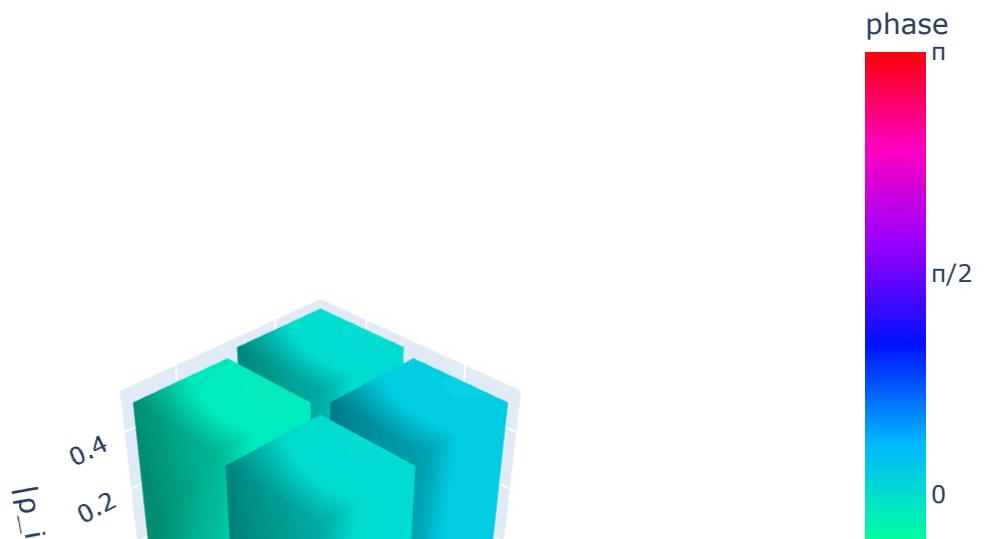
Random_2 | 0.9966

Reconstructed Random_2 (F=0.997)



Random_3 | 1.0005

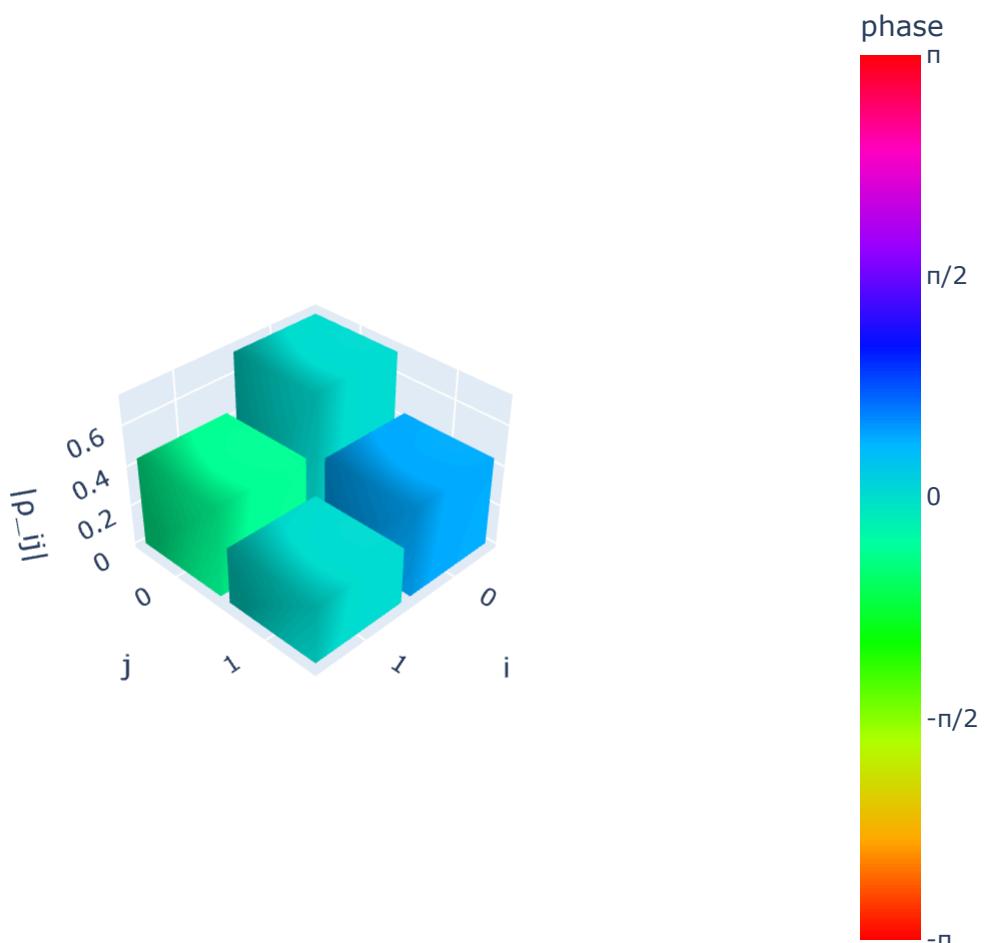
Reconstructed Random_3 (F=1.000)





Random_4 | 0.9931

Reconstructed Random_4 (F=0.993)



Random_5 | 0.9864

Reconstructed Random_5 (F=0.986)



Task 5: Validation and Trends

The assignment asks to "plot trends per shot count". I am running a sweep from 100 to 10,000 shots to see how the error reduces.

```
# Analyzing Shot Noise
shot_levels = [100, 500, 1000, 5000, 10000]
fidelities = []

# Testing on |+i> state
qc_test = QuantumCircuit(1); qc_test.h(0); qc_test.s(0)
rho_target = DensityMatrix(qc_test).data

print("Running shot noise analysis...")
for s in shot_levels:
    probs = {}
    for basis in ["X", "Y", "Z"]:
        qc = measure_in_basis(qc_test, basis)
        res = simulator.run(qc, shots=s).result()
        probs[basis] = res.get_counts().get("0", 0) / s

    exp_x = 2*probs["X"]-1
    exp_y = 2*probs["Y"]-1
    exp_z = 2*probs["Z"]-1
    rho_rec = 0.5 * (I + exp_x*X + exp_y*Y + exp_z*Z)
    fidelities.append(state_fidelity(rho_target, rho_rec, validate=False))

# Plot trend
fig = go.Figure(data=go.Scatter(x=shot_levels, y=fidelities, mode='lines+markers'))
fig.update_layout(title="Fidelity vs Shot Count", xaxis_title="Shots", yaxis_title="Fidelity")
fig.show()
```

Running shot noise analysis...

Fidelity vs Shot Count

