QC Emulator

March 29, 2025

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

0.1 Non-Parameterised Gates

[2]:
```
I = np.array([[1, 0], [0, 1]])
X = np.array([[0, 1], [1, 0]])
Y = np.array([[0, -1j], [1j, 0]])
Z = np.array([[1, 0], [0, -1]])
H = (1 / np.sqrt(2)) * np.array([[1, 1], [1, -1]])
S = np.array([[1, 0], [0, 1j]])
T = np.array([[1, 0], [0, np.exp(1j * np.pi / 4)]])
```

0.2 Parameterised Gates

[3]:
```
def Rx(theta):
    return np.array([
        [np.cos(theta / 2), -1j * np.sin(theta / 2)],
        [-1j * np.sin(theta / 2), np.cos(theta / 2)]
        ], dtype=complex)

def Ry(theta):
    return np.array([
        [np.cos(theta / 2), -np.sin(theta / 2)],
        [np.sin(theta / 2), np.cos(theta / 2)]
        ], dtype=complex)

def R(phi):
    return np.array([
        [1, 0], [0, np.exp(1j * phi)]
        ], dtype=complex)
```

0.3 Quantum Circuit Class

[4]:
```
class Circuit:

    '''Create column state vector'''
    def __init__(self, n):
```

```
        self.n = n
        self.state = np.zeros(2**n, dtype=complex)
        self.state[0] = 1

    '''Apply gates (Kronecker product or bitwise operators)'''
    def apply(self, gate, *qubits):
        if len(qubits) == 1:
            k = qubits[0]

            I_front = np.eye(2**(k - 1))
            I_back = np.eye(2**(self.n - k))
            padded_gate = np.kron(np.kron(I_front, gate), I_back)

            self.state = padded_gate @ self.state

        else:
            control, target = qubits
            matrix = np.eye(2**self.n, dtype=complex)

            for i in range(2**self.n):
                if (i >> (self.n - control)) & 1:

                    if np.allclose(X, gate):
                        j = i ^ (1 << (self.n - target))

                        matrix[i, i], matrix[i, j] = 0, 1
                        matrix[j, i], matrix[j, j] = 1, 0

                    elif np.allclose(H, gate):
                        i0 = i & ~(1 << (self.n - target))
                        i1 = i | (1 << (self.n - target))

                        matrix[i0, i0] = H[0, 0]
                        matrix[i0, i1] = H[0, 1]
                        matrix[i1, i0] = H[1, 0]
                        matrix[i1, i1] = H[1, 1]

            self.state = matrix @ self.state

    '''Measure specific qubits'''
    def measure(self, *qubits):
        for qubit in qubits:

            group_0 = [(i >> (self.n - qubit)) & 1 == 0 for i in range(2**self.
↪n)]
            group_1 = [(i >> (self.n - qubit)) & 1 == 1 for i in range(2**self.
↪n)]
```

```
            prob_0 = np.sum(np.abs(self.state[group_0])**2)
            prob_1 = np.sum(np.abs(self.state[group_1])**2)

            measurement = np.random.choice([0, 1], p=[prob_0, prob_1])

            mask = group_0 if measurement == 0 else group_1
            self.state = np.where(mask, self.state, 0)
            self.state /= np.linalg.norm(self.state)

            return measurement

    '''Measure all qubits'''
    def measure_all(self, collapse=True):
        probabilities = np.abs(self.state) ** 2
        measurement = np.random.choice(len(self.state), p=probabilities)
        bitstring = format(measurement, f'0{self.n}b')

        if collapse:
            self.state = np.zeros_like(self.state)
            self.state[measurement] = 1

        return bitstring

    '''Display state vector in column form'''
    def show(self):
        print(self.state.reshape(-1,1))

    '''Display state vector in Dirac notation'''
    def diracify(self):
        terms = []

        for i, amplitude in enumerate(self.state):

            if not np.isclose(amplitude, 0):
                basis = format(i, f'0{self.n}b')
                amp = f"({amplitude:.3f})"
                terms.append(f"{amp} \033[1m|{basis} \033[0m")

        print(" + ".join(terms))
```

0.4 Example usage with W state

[5]:

```
# create W state
qc = Circuit(3)
qc.apply(Ry(np.arccos(-1/3)), 1)
qc.apply(H, 1, 2)
```

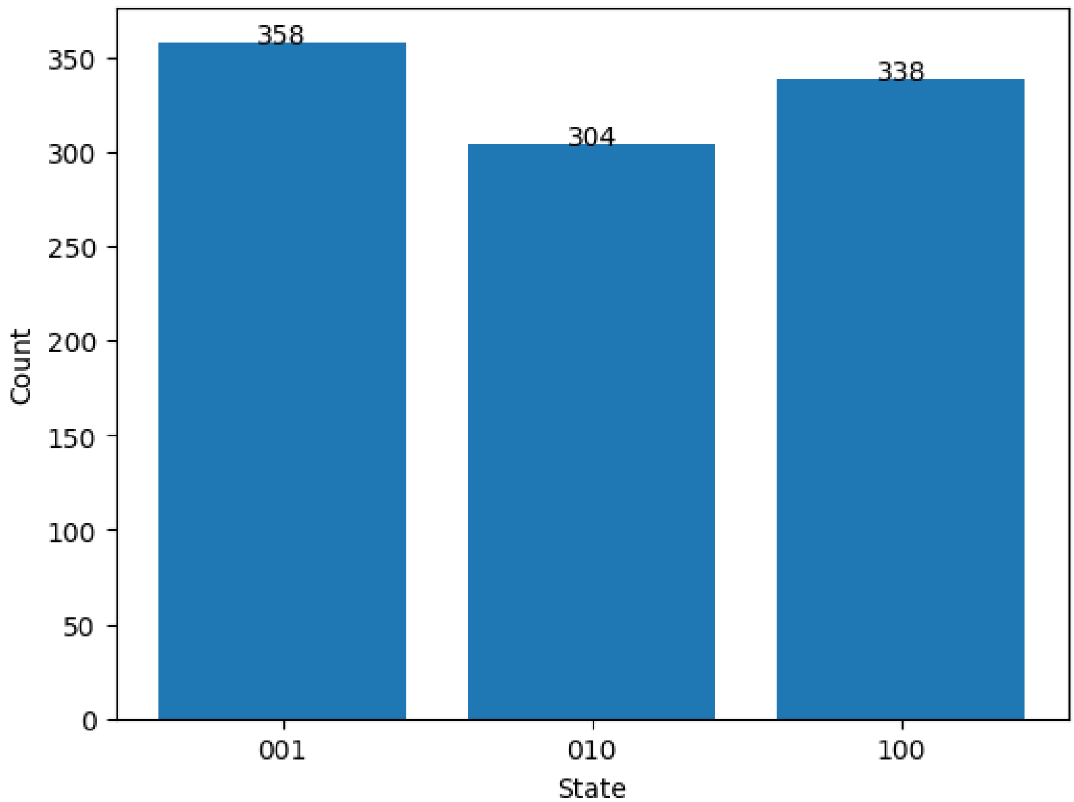
```
qc.apply(X, 2, 3)
qc.apply(X, 1, 2)
qc.apply(X, 1)
qc.show()
qc.diracify()
```

```
[[0.        +0.j]
 [0.57735027+0.j]
 [0.57735027+0.j]
 [0.        +0.j]
 [0.57735027+0.j]
 [0.        +0.j]
 [0.        +0.j]
 [0.        +0.j]]
(0.577+0.000j) |001  + (0.577+0.000j) |010  + (0.577+0.000j)
|100
```

[6]:
```
# measure circuit 1000 times
results = [qc.measure_all(False) for _ in range(1000)]
states, counts = np.unique(results, return_counts=True)
for state, count in zip(states, counts):
    print(f'{state}: {count}')

# plot bar chart of counts
plt.bar(states, counts)
plt.xlabel('State')
plt.ylabel('Count')
for i in range(len(states)):
    plt.text(i, counts[i], counts[i], ha='center')
plt.show()
```

```
001: 358
010: 304
100: 338
```


358
304
338
350
300
250
200
150
100
50
0
Count
001
010
100
State