QC Emulator - Version 3.1

August 4, 2025

[ ]:
```
import numpy as np
from numpy import linalg
from functools import reduce
from IPython.display import display, Math
import matplotlib.pyplot as plt
```

[ ]:
```
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)
H = (1 / np.sqrt(2)) * np.array([[1, 1], [1, -1]], dtype=complex)
SX = (1 / 2) * np.array([[1 + 1j, 1 - 1j], [1 - 1j, 1 + 1j]], dtype=complex)
S = np.array([[1, 0], [0, 1j]], dtype=complex)
T = np.array([[1, 0], [0, np.exp(1j * np.pi / 4)]], dtype=complex)
Sdg = np.array([[1, 0], [0, -1j]], dtype=complex)
Tdg = np.array([[1, 0], [0, np.exp( -1j * np.pi / 4)]], dtype=complex)

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 Rz(theta):
    return np.array([
        [np.exp(-1j * theta / 2), 0],
        [0, np.exp(1j * theta / 2)]
    ], dtype=complex)

def P(phi):
    return np.array([
        [1, 0],
```

```
        [0, np.exp(1j * phi)]
    ], dtype=complex)
```

[ ]:
```
class QuantumCircuit:

    '''initialiser method'''
    def __init__(self, num_qubits, structure='', xyz_errors=[0.0, 0.0, 0.0],␣
↪random_error=0.0):

        self.n = num_qubits
        self.dim = 2**num_qubits
        self.structure = 'StateVector' if structure == '' else structure

        if self.structure == 'StateVector':
            self.state = np.zeros(self.dim, dtype=complex)
            self.state[0] = 1
        elif self.structure == 'DensityMatrix':
            self.state = np.zeros((self.dim, self.dim), dtype=complex)
            self.state[0, 0] = 1
        else:
            raise ValueError('Must specify the structure representing the␣
↪quantum system.')

        self.x = xyz_errors[0]
        self.y = xyz_errors[1]
        self.z = xyz_errors[2]
        self.error = random_error

    '''noise methods'''
    def add_gate_errors(self, qubit):

        for error, error_rate in [(X, self.x), (Y, self.y), (Z, self.z)]:
            if np.random.rand() < error_rate:
                error_path = [error if i == qubit else I for i in range(self.n)]

                expanded_error = reduce(np.kron, error_path)
                self.__apply_gate(expanded_error)
                self.state /= np.linalg.norm(self.state)

    def add_noise(self):

        for qubit in range(self.n):
            for error in [X, Y, Z]:
                if np.random.rand() < self.error:
                    error_path = [error if i == qubit else I for i in␣
↪range(self.n)]
```

```
                expanded_error = reduce(np.kron, error_path)
                self.__apply_gate(expanded_error)
                self.state /= np.linalg.norm(self.state)

    '''operation methods'''
    def gate(self, gate, *qubits):

        for qubit in qubits:
            path = [gate if i == qubit else I for i in range(self.n)]
            self.add_gate_errors(qubit)
            self.add_noise()

            expanded_gate = reduce(np.kron, path)
            self.__apply_gate(expanded_gate)
            self.state /= np.linalg.norm(self.state)

    def C(self, gate, control, target):

        proj_0 = np.array([[1, 0], [0, 0]], dtype=complex)
        proj_1 = np.array([[0, 0], [0, 1]], dtype=complex)

        if isinstance(control, int):
            control = [control]

        if target in control:
            raise ValueError('Target qubit cannot be a control qubit.')

        expanded_gate = np.zeros((2**self.n, 2**self.n), dtype=complex)

        for number in range(2**len(control) - 1):
            bitstring = np.binary_repr(number, width=len(control))

            inactive_path = []
            control_index = 0
            for i in range(self.n):
                if i in control:
                    inactive_path.append(proj_0 if bitstring[control_index] == '0' else proj_1)
                    self.add_gate_errors(i)
                    control_index += 1
                else:
                    inactive_path.append(I)

            expanded_gate += reduce(np.kron, inactive_path)

        active_path = []
        for i in range(self.n):
```

```python
            if i in control:
                active_path.append(proj_1)
                self.add_gate_errors(i)
            elif i == target:
                active_path.append(gate)
                self.add_gate_errors(i)
            else:
                active_path.append(I)
        self.add_noise()

        expanded_gate += reduce(np.kron, active_path)
        self.__apply_gate(expanded_gate)
        self.state /= np.linalg.norm(self.state)

    '''channel methods'''
    def dephase(self, *qubits, epsilon=1.0):

        if self.structure == 'StateVector':
            raise ValueError('Structure must be DensityMatrix for channels.')

        for row in range(self.dim):
            for column in range(self.dim):

                if row == column:
                    continue

                binary_row = np.binary_repr(row, width=self.n)
                binary_column = np.binary_repr(column, width=self.n)

                for qubit in qubits:
                    if binary_row[qubit] != binary_column[qubit]:
                        self.state[row, column] *= (1 - epsilon)
                        break

    def depolarise(self, *qubits, epsilon=1.0):

        if self.structure == 'StateVector':
            raise ValueError('Structure must be DensityMatrix for channels.')

        for qubit in qubits:
            path = [I / 2 if i == qubit else I for i in range(self.n)]
            expanded_gate = reduce(np.kron, path)
            self.state = (1 - epsilon) * self.state + epsilon * (expanded_gate @ self.state @ np.conj(expanded_gate).T)

    '''measurement methods'''
    def measure(self, *qubits, collapse=True):
```

```
        results = {}
        for qubit in qubits:

            self.add_gate_errors(qubit)
            self.add_noise()

            list_0 = []
            list_1 = []
            for i in range(2**self.n):
                basis = np.binary_repr(i, width=self.n)

                if basis[qubit] == '0':
                    list_0.append(i)
                else:
                    list_1.append(i)

            if self.structure == 'StateVector':
                prob_0 = np.sum(np.abs(self.state[list_0])**2)
                prob_1 = np.sum(np.abs(self.state[list_1])**2)
            elif self.structure == 'DensityMatrix':
                prob_0 = np.real(np.sum([self.state[i, i] for i in list_0]))
                prob_1 = np.real(np.sum([self.state[i, i] for i in list_1]))

            measurement = np.random.choice([0, 1], p=[prob_0, prob_1])
            results[f'q{qubit}'] = measurement

            if collapse:
                new_state = np.zeros_like(self.state)

                if self.structure == 'StateVector':
                    if measurement == 0:
                        new_state[list_0] = self.state[list_0]
                    elif measurement == 1:
                        new_state[list_1] = self.state[list_1]
                    self.state = new_state / np.linalg.norm(new_state)

                elif self.structure == 'DensityMatrix':
                    indices = list_0 if measurement == 0 else list_1
                    sub_indices = np.ix_(indices, indices)
                    new_state[sub_indices] = self.state[sub_indices]
                    self.state = new_state / np.trace(new_state)

        return results

    def measure_all(self, collapse=True):
```

```
        self.add_noise()

        if self.structure == 'StateVector':
            probabilities = np.abs(self.state) ** 2
            measurement = np.random.choice(len(self.state), p=probabilities)
        elif self.structure == 'DensityMatrix':
            probabilities = np.real(np.diag(self.state))
            measurement = np.random.choice(len(probabilities), p=probabilities)

        basis_measurement = np.binary_repr(measurement, width=self.n)

        if collapse:
            self.state = np.zeros_like(self.state)

            if self.structure == 'StateVector':
                self.state[measurement] = 1
            elif self.structure == 'DensityMatrix':
                self.state[measurement, measurement] = 1

        return basis_measurement

    '''transformation methods'''
    def to_density_matrix(self, permanent=False):

        if self.structure == 'DensityMatrix':
            raise ValueError('Quantum system already a DensityMatrix structure.
↪')

        matrix = np.outer(self.state, np.conj(self.state))

        if permanent:
            self.state = matrix
            self.structure = 'DensityMatrix'
            return self.state
        else:
            return matrix

    def to_state_vector(self, permanent=False):

        if self.structure == 'StateVector':
            raise ValueError('Quantum system already a StateVector structure.')

        eigenvalues, eigenvectors = np.linalg.eigh(self.state)

        for eigenvalue, eigenvector in zip(eigenvalues, eigenvectors.T):
            if np.isclose(eigenvalue, 1):
```

```
                global_phase = np.exp(-1j * np.angle(eigenvector[np.argmax(np.
↪abs(eigenvector))]))
                eigenvector *= global_phase

                if permanent:
                    self.state = eigenvector
                    self.structure = 'StateVector'
                    return self.state
                else:
                    return eigenvector

        raise ValueError('Transformation failed; not a pure state.')

    '''visualisation methods'''
    def show(self, latex=False):

        if self.structure == 'StateVector':
            array = self.state.reshape(-1, 1)
        elif self.structure == 'DensityMatrix':
            array = self.state

        if not latex:
            print(array)

        if latex:
            text = r'\begin{pmatrix}'

            for row in array:
                row_elements = []
                for element in row:
                    value = QuantumCircuit.__clean_format(element)
                    row_elements.append(value)
                text += '&'.join(row_elements) + r'\\' + '\n'

            text += r'\end{pmatrix}'

            display(Math(text))

    def diracify(self):

        if self.structure == 'DensityMatrix':
            raise ValueError('diracify function only valid for StateVector␣
↪structures.')

        terms = []
        for basis, amplitude in enumerate(self.state):
            if not np.isclose(amplitude, 0):
```

```
                amplitude = f'({QuantumCircuit.__clean_format(amplitude)})'
                basis = np.binary_repr(basis, width=self.n)
                terms.append(f'{amplitude} \033[1m|{basis} \033[0m')

        print(" + ".join(terms))

    def plot_probs(self, output='list', dims=[6.4, 4.8], x_rot=0, by_qubit=False):

        if isinstance(output, str):
            output = [output]
        if not all(format in {'list', 'plot'} for format in output):
            raise ValueError('Invalid output format.')

        states = []
        probs = []
        for state in range(self.dim):
            states.append(np.binary_repr(state, width=self.n))

            if self.structure == 'StateVector':
                probs.append(np.abs(self.state[state])**2)
            elif self.structure == 'DensityMatrix':
                probs.append(self.state[state, state].real)

        if by_qubit:
            qubit_probs = {f'q{i}': {0: 0.0, 1: 0.0} for i in range(self.n)}
            for state, prob in zip(states, probs):
                bits = list(map(int, state))
                for i, bit in enumerate(bits):
                    qubit_probs[f'q{i}'][bit] += prob

        if 'list' in output:

            if by_qubit:
                for i in range(self.n):
                    print(f'q{i}: {{0: {qubit_probs[f"q{i}"][0]:.3g}, 1: {qubit_probs[f"q{i}"][1]:.3g}}}')
            else:
                for state, prob in zip(states, probs):
                    print(f'{state}: {prob:.3g}')

        if 'plot' in output:

            if by_qubit:
                qubits = list(qubit_probs.keys())
                prob_0 = [qubit_probs[qubit][0] for qubit in qubits]
                prob_1 = [qubit_probs[qubit][1] for qubit in qubits]
```

```
            x = np.arange(len(qubits))
            width = 0.3

            fig, ax = plt.subplots(figsize=(dims[0], dims[1]))
            bars_0 = ax.bar(x - width/2, prob_0, width, label='0')
            bars_1 = ax.bar(x + width/2, prob_1, width, label='1')

            ax.set_ylim(top=1.05)
            ax.set_xlabel('Qubits')
            ax.set_ylabel('Outcome probability')
            ax.set_xticks(x)
            ax.set_xticklabels(qubits)
            ax.legend()
            plt.xticks(rotation = x_rot)

            y_min, y_max = plt.ylim()
            offset = (y_max - y_min) / 100
            for bar in bars_0 + bars_1:
                height = bar.get_height()
                ax.annotate(f'{height:.3g}',
                xy=(bar.get_x() + bar.get_width() / 2, height + offset),
                ha='center')

            plt.show()

        else:
            plt.figure(figsize=(dims[0], dims[1]))
            plt.bar(states, probs)
            plt.ylim(top=1.05)
            plt.xlabel('Standard basis states')
            plt.ylabel('Outcome probability')
            plt.xticks(rotation = x_rot)

            y_min, y_max = plt.ylim()
            offset = (y_max - y_min) / 100
            for i in range(self.dim):
                plt.text(i, probs[i] + offset, f'{probs[i]:.3g}', ha='center')

            plt.show()

    '''helper methods'''
    def __apply_gate(self, gate):

        self.state = gate @ self.state
        if self.structure == 'DensityMatrix':
```

```
            self.state = self.state @ np.conj(gate).T

    @staticmethod
    def __clean_format(element):

        if isinstance(element, complex):
            re = element.real
            im = element.imag

            if np.isclose(im, 0):
                value = f'{int(re)}' if np.isclose(re, round(re)) else f'{re:.
↪5g}'
            elif np.isclose(re, 0):
                value = f'{int(im)}' if np.isclose(im, round(im)) else f'{im:.
↪5g}i'
            else:
                re_string = f'{int(re)}' if np.isclose(re, round(re)) else␣
↪f'{re:.5g}'
                im_string = f'{int(abs(im))}' if np.isclose(abs(im),␣
↪round(abs(im))) else f'{abs(im):.5g}'
                sign = '+' if im > 0 else '-'
                value = f'{re_string}{sign}{im_string}i'
        else:
            value = f'{int(element)}' if np.isclose(element, round(element))␣
↪else f'{element:.5g}'

        return value
```

[ ]:
```
def run_circuit(circuit, shots=1, output='list', dims=[6.4, 4.8], x_rot=0,␣
↪by_qubit=False):

    if isinstance(output, str):
        output = [output]
    if not all(format in {'list', 'plot'} for format in output):
        raise ValueError('Invalid output format.')

    results = []
    for _ in range(shots):
        qc = circuit()
        results.append(qc.measure_all(False))
    states, counts = np.unique(results, return_counts=True)

    if output == []:
        return states, counts

    num_qubits = len(states[0])
    if by_qubit:
```

```
        qubit_counts = {f'q{i}': {0: 0, 1: 0} for i in range(num_qubits)}
        for state, count in zip(states, counts):
            bits = list(map(int, state))
            for i, bit in enumerate(bits):
                qubit_counts[f'q{i}'][bit] += count

    if 'list' in output:

        if by_qubit:
            for i in range(num_qubits):
                print(f'q{i}: {qubit_counts[f"q{i}"]}')
        else:
            for state, count in zip(states, counts):
                print(f'{state}: {count}')

    if 'plot' in output:

        if by_qubit:
            qubits = list(qubit_counts.keys())
            count_0 = [qubit_counts[qubit][0] for qubit in qubits]
            count_1 = [qubit_counts[qubit][1] for qubit in qubits]

            x = np.arange(len(qubits))
            width = 0.3

            fig, ax = plt.subplots(figsize=(dims[0], dims[1]))
            bars_0 = ax.bar(x - width/2, count_0, width, label='0')
            bars_1 = ax.bar(x + width/2, count_1, width, label='1')

            ax.set_xlabel('Qubits')
            ax.set_ylabel('Counts')
            ax.set_xticks(x)
            ax.set_xticklabels(qubits)
            ax.legend()
            plt.xticks(rotation = x_rot)

            y_min, y_max = plt.ylim()
            offset = (y_max - y_min) / 100
            for bar in bars_0 + bars_1:
                height = bar.get_height()
                ax.annotate(f'{height}',
                xy=(bar.get_x() + bar.get_width() / 2, height + offset),
                ha='center')

            plt.show()

        else:
```

```
    plt.figure(figsize=(dims[0], dims[1]))
    plt.bar(states, counts)
    plt.xlabel('Standard basis states')
    plt.ylabel('Counts')
    plt.xticks(rotation = x_rot)

    y_min, y_max = plt.ylim()
    offset = (y_max - y_min) / 100
    for i in range(len(states)):
        plt.text(i, counts[i] + offset, counts[i], ha='center')

    plt.show()
```