Task 4 - Random Circuits

March 11, 2023

1 Task 4 - Random Circuits

1.0.1 Problem Statement

Design a function that generates a random quantum circuit by considering as parameters the number of qubits, the number of depths, and the base of gates to be used. You can only use the quantum gates of 1 and 2 qubits. Bonus: use the described order between qubits of a quantum computer [implemented at the end!]

Assumption [1]:
The distribution with which each gate is picked is independently and identically distributed. This means that picking the ith gate is independent of what we picked for the previous i-1 gates.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, transpile, Aer, assemble

print('Process Complete!')
```

```
Process Complete!
```

1.1 Attempt 1: Implementing a basic random circuit using the Pauli Gates

In this attempt we will try to get a 'feel' of the problem and create a random circuit generator that uses the pauli gates: I, X, Y, Z.

[2]:
```
# simple attempt for the Pauli Gates (including Identity)
def random_circuit_1(num_qubits: int, depth: int, basis_gates: list, pd: list =
 ↪['uniform'], include_identity=False):
    '''

    pd = probability distribution according to which each gate in basis_gates
 ↪can be picked
    include_identity = If True then it actually implements the Identity gate if
 ↪it is randomly sampled
                        If False then it ignores the Identity gate if it is
 ↪randomly sampled

    note that the identity gate must be part of basis_gates
```

```
    '''

    num_classical_bits = num_qubits
    qc = QuantumCircuit(num_qubits, num_classical_bits)

    if pd == ['uniform']:
        pd = [1 / len(basis_gates)] * len(basis_gates)

    for i in range(num_qubits):
        for j in range(depth):
            gate = np.random.choice(basis_gates, p=pd)

            if gate == 'X':
                qc.x(i)

            elif gate == 'Y':
                qc.y(i)

            elif gate == 'Z':
                qc.z(i)

            elif gate == 'ID' and include_identity:
                # apply the identity gate
                qc.i(i)

    print('Basis Gates: ', basis_gates)
    print('Probabilities of Occurrence: ', pd)
    print(qc)

print('Process Complete!')
```

```
Process Complete!
```

[3]:

```
# Uniformly pick any of the Pauli Gates
random_circuit_1(3, 5, ['X', 'Y', 'Z', 'ID'], pd=[0.25, 0.25, 0.25, 0.25],␣
 ↪include_identity=True)

# Uniformly pick any of the Pauli Gates (without explicitly applying Identity)
random_circuit_1(3, 5, ['X', 'Y', 'Z', 'ID'], pd=[0.25, 0.25, 0.25, 0.25],␣
 ↪include_identity=False)

# Pick either Sigma Y or Sigma Z with probability 0.5 each
random_circuit_1(3, 5, ['X', 'Y', 'Z', 'ID'], pd=[0, 0.5, 0.5, 0])
```

```
Basis Gates:  ['X', 'Y', 'Z', 'ID']
Probabilities of Occurrence:  [0.25, 0.25, 0.25, 0.25]

q_0:   Y    X    Z    Z    Y
```

```
q_1:   X   Z   X   Y   I

q_2:   Z   I   I   Y   X

c: 3/

Basis Gates:  ['X', 'Y', 'Z', 'ID']
Probabilities of Occurrence:  [0.25, 0.25, 0.25, 0.25]

q_0:   Y   X   Y   X

q_1:   Z   Y

q_2:   Y   Y   Y   Y

c: 3/

Basis Gates:  ['X', 'Y', 'Z', 'ID']
Probabilities of Occurrence:  [0, 0.5, 0.5, 0]

q_0:   Z   Y   Y   Z   Z

q_1:   Y   Z   Z   Y   Y

q_2:   Y   Y   Z   Y   Z

c: 3/
```

1.2 Attempt 2: Adding more flexibility

It would be nice to control which qubit to apply the gate on (probabilistically).
We are changing the mechanism slightly so that first a gate is chosen from the gate distribution and qubit (or multiple qubits) are chosen based on the qubit distribution.

To maintain a limit on the max depth, a check is performed. Another degree of freedom would be to have randomized depth (with certain upper and lower bounds), which has also been implemented.

[8]:
```
# A more powerful way of generating a random circuit

# np.random.seed(0) # for debugging

def check_gate_threshold(gates_on_each_qubit, min_depth):
    '''
    returns False if the number of gates on any qubit is lesser than the depth
    '''
    for num_gates in gates_on_each_qubit:
```

```python
        if num_gates < min_depth:
            return False
    return True

def remove_maxed_out_qubits(gates_on_each_qubit, max_depth):
    global qubits, qubit_distribution

    for q in qubits:

        qubit_index = qubits.index(q)

        if gates_on_each_qubit[q] >= max_depth:
            qubits.remove(q)
            qubit_prob = qubit_distribution[qubit_index]
            qubit_distribution = np.delete(qubit_distribution, qubit_index)
            qubit_distribution = qubit_distribution + qubit_prob/
 ↪len(qubit_distribution)


def pick_qubit(gates_on_each_qubit, max_depth):
    '''
    How re-normalization is handled:
    If the array is [0.1  0.15  0.25  0.25  0.15  0.1]
    And the qubit 2 reaches its maximum gate depth,
    then we distribute the probabilities of qubit 2 to the others,
    which gives us:               [0.15  0.2  0.3  0.2  0.15]
    Where the qubit indices are:  [0     1    3    4    5   ]

    This way our algorithm is guaranteed to halt
    '''

    global qubits, qubit_distribution

    if len(qubits) == 1:
        return_qubit = qubits[0]
        qubit_distribution = np.array([])
        qubits = []
        return return_qubit

    remove_maxed_out_qubits(gates_on_each_qubit, max_depth)
    if len(qubits) == 0:
        return -1

    return np.random.choice(qubits, p=qubit_distribution)

def apply_single_qubit_gate(qc, gates_on_each_qubit, max_depth, gate,␣
 ↪include_identity):
```

```python
    qubit = pick_qubit(gates_on_each_qubit, max_depth)

    if qubit == -1:
        return

    if gate == 'X':
        qc.x(qubit)

    elif gate == 'Y':
        qc.y(qubit)

    elif gate == 'Z':
        qc.z(qubit)

    elif gate == 'ID' and include_identity:
        qc.i(qubit)

    gates_on_each_qubit[qubit] += 1


def random_circuit_2(num_qubits: int, max_depth: int, min_depth: int,␣
 ↪basis_gates: list,
                     include_identity: int = False):
    '''
    include_identity = If True then it actually implements the Identity gate if␣
 ↪it is randomly sampled
                       If False then it ignores the Identity gate if it is␣
 ↪randomly sampled

    note that the identity gate must be part of basis_gates (for effective␣
 ↪circuit generation)
    '''
    global qubits, qubit_distribution, gate_distribution

    two_qubit_gates = ['CX']

    num_classical_bits = num_qubits
    qc = QuantumCircuit(num_qubits, num_classical_bits)

    if np.array_equal(gate_distribution, np.array(['uniform'])):
        gate_distribution = [1 / len(basis_gates)] * len(basis_gates)

    if np.array_equal(qubit_distribution, np.array(['uniform'])):
        qubit_distribution = [1 / num_qubits] * num_qubits

    # Number of gates on each qubit
```

```
    gates_on_each_qubit = [0] * num_qubits

    while not check_gate_threshold(gates_on_each_qubit, min_depth) and len(qubits) != 0:

        gate = np.random.choice(basis_gates, p=gate_distribution)

        gate_type = 'single-qubit'

        if gate in two_qubit_gates:
            gate_type = '2-qubit'

        # Further categories for 'gate_type' will go here (example '3-qubit' for 3 qubit gates)

        if gate_type == 'single-qubit':
            apply_single_qubit_gate(qc, gates_on_each_qubit, max_depth, gate, include_identity)

        if gate_type == '2-qubit':
            if len(qubits) == 1:
                apply_single_qubit_gate(qc, gates_on_each_qubit, max_depth, gate, include_identity)

            if len(qubits) == 2:
                # Tricky case, so we're avoiding pick_qubit()
                qubit1 = np.random.choice(qubits, p=qubit_distribution)
                qubit1_index = np.where(qubits == qubit1)[0]

                if qubit1_index == 0:
                    qubit2 = qubits[1]
                else:
                    qubit2 = qubits[0]

            else:
                qubit1 = pick_qubit(gates_on_each_qubit, max_depth)

                if qubit1 == -1:
                    break

                qubit2 = pick_qubit(gates_on_each_qubit, max_depth)

                if qubit2 == -1:
                    break

                if qubit1 == qubit2 and len(qubits) == 1:
```

```
                    apply_single_qubit_gate(qc, gates_on_each_qubit, max_depth,␣
↪gate, include_identity)
                    continue

                while qubit1 == qubit2:
                    qubit2 = pick_qubit(gates_on_each_qubit, max_depth)
                    if qubit2 == -1:
                        break

            if gate == 'CX':
                qc.cx(qubit1, qubit2)

            gates_on_each_qubit[qubit1] += 1
            gates_on_each_qubit[qubit2] += 1


    print('Number of qubits  : ', num_qubits)
    print('Min Circuit Depth : ', min_depth)
    print('Max Depth         : ', max_depth)
    print('Basis Gates       : ', basis_gates)
    print('Gate Distribution : ', gate_distribution)
    print('Qubit Distribution: ', qubit_distribution)
    print('include_identity  : ', include_identity)
    print(qc)


# The gate distribution is the probability distribution using with which a gate␣
↪is sampled.
# The qubit distribution is the probability distribution using with which a␣
↪qubit is sampled.
num_qubits = 6
qubits = [i for i in range(num_qubits)]
gate_distribution = np.array([0.15, 0.15, 0.15, 0.3, 0.25])
qubit_distribution = np.array([0.1, 0.15, 0.25, 0.25, 0.15, 0.1])

# np.random.seed(309)
random_circuit_2(num_qubits = num_qubits, max_depth = 10, min_depth = 5,
                 basis_gates = ['X', 'Y', 'Z', 'ID', 'CX'],
                 include_identity = False)


'''
# For testing
for i in range(10001):
    print('seed ', i, end=' ')
    np.random.seed(i)
    num_qubits = 6
```

```
    qubits = [i for i in range(num_qubits)]
    gate_distribution = np.array([0.15, 0.15, 0.15, 0.3, 0.25])
    qubit_distribution = np.array([0.1, 0.15, 0.25, 0.25, 0.15, 0.1])

    random_circuit_2(num_qubits = num_qubits, max_depth = 10, min_depth = 5,
                     basis_gates = ['X', 'Y', 'Z', 'ID', 'CX'],
                     include_identity = False)
'''

print('Process Complete!')
```

```
Number of qubits  :  6
Min Circuit Depth :  5
Max Depth         :  10
Basis Gates       :  ['X', 'Y', 'Z', 'ID', 'CX']
Gate Distribution :  [0.15 0.15 0.15 0.3  0.25]
Qubit Distribution:  [0.15 0.2  0.3  0.2  0.15]
include_identity  :  False

q_0:        Y   Y

q_1:   Z      Z

q_2:   X      Z   X   Y   Y      Z      Z

q_3:        X   Z   Y      X

q_4:     X   X   Z

q_5:   Y                      X

c: 6/

Process Complete!
```

1.3 Attempt 3: Refactoring + Architecture Aware Random Circuit Generator (Bonus)

The above implementation is nice, but the code could be better written. We'll create a class called `RandomCircuitGenerator`, which will do the same as above. We'll also provide support for more gates.

Also, the bonus gives us the qubit topology (layout) of a quantum computer. This layout can be represented using the graph data structure (so that the solution is generalized, and not hard coded to a specific layout).

Approach: If there is an edge between any 2 nodes i and j (that is, qubit_i and qubit_j are 'connected'), then the probability of applying a 2 qubit gate between them is larger than if there was no edge.

Assumption [2]:
For now we're taking an assumption that a 2 qubit gate can be applied if the two qubits are connected in the topology. One edge will be picked uniformly at random.

This can be changed later as follows: We will change the probabilities depending on the distance between the qubits. As the path distance between qubit_i and qubit_j increases, the probability of applying a 2 qubit gate decreases (which makes sense, since we want to minimize the number of SWAP operations).

[5]:
```
class RandomCircuitGenerator():

    def __init__(self, num_qubits, single_qubit_gates, two_qubit_gates,
                 single_qubit_gate_distribution, two_qubit_gate_distribution,
                 qubit_distribution_1,
                 topology,
                 min_depth, max_depth,
                 probability_of_two_qubit_gate):

        '''
        qubit_distribution_1 = the distribution with which we will pick a qubit␣
 ↪for a single qubit operation
        qubit_distribution_2 = the distribution with which we will pick a qubit␣
 ↪pair for a 2 qubit gate operation

        topology = graph
        '''

        self.num_qubits = num_qubits
        self.qubits     = [i for i in range(num_qubits)]

        self.single_qubit_gates = single_qubit_gates
        self.two_qubit_gates    = two_qubit_gates

        self.single_qubit_gate_distribution = single_qubit_gate_distribution
        self.two_qubit_gate_distribution    = two_qubit_gate_distribution

        self.qubit_distribution_1 = qubit_distribution_1

        self.qubit_distribution_2 = np.array([np.array([topology['E'][i][0],␣
 ↪topology['E'][i][1], 1/len(topology['E'])]) for i in␣
 ↪range(len(topology['E']))]) # Sampling an edge uniformly at random

        self.min_depth = min_depth
        self.max_depth = max_depth
        self.probability_of_two_qubit_gate = probability_of_two_qubit_gate

        self.gates_on_each_qubit = [0 for i in range(num_qubits)]
```

```
        self.copy_qubit_dist_1 = list(self.qubit_distribution_1)
        self.copy_qubit_dist_2 = self.qubit_distribution_2


    def generate_qubit_distribution_2(self, topology):
        '''
        for when we want to use the distances between qubits to calculate␣
↪qubit_distribution_2
        to remove assumption [2]
        '''

        None

    def check_min_gate_threshold(self):
        '''
        returns False if the number of gates on any qubit is lesser than the␣
↪depth
        '''

        for self.num_gates in self.gates_on_each_qubit:
            if self.num_gates < self.min_depth:
                return False
        return True


    def remove_maxed_out_qubits(self):
        for q in self.qubits:
            qubit_index = self.qubits.index(q)

            if self.gates_on_each_qubit[q] >= self.max_depth:
                self.qubits.remove(q)
                qubit_prob_1 = self.qubit_distribution_1[qubit_index]
                self.qubit_distribution_1 = np.delete(self.
↪qubit_distribution_1, qubit_index)
                self.qubit_distribution_1 = self.qubit_distribution_1 +␣
↪qubit_prob_1 / len(self.qubit_distribution_1)

                new_dist = []

                #print(self.qubit_distribution_2)

                if len(self.qubit_distribution_2) != 0:


                    for e in self.qubit_distribution_2:
                        if len(e) == 0:
                            continue
```

```
                        q0 = e[0]
                        q1 = e[1]
                        prob = e[2]

                        row = []
                        if q0 != q and q1 != q:
                            row.append(q0)
                            row.append(q1)
                            row.append(prob)
                        new_dist.append(np.array(row))

                    self.qubit_distribution_2 = np.array(new_dist)

        if len(self.qubit_distribution_2) == 0:
            return

        updated_probs_for_dist_2 = 1 / len(self.qubit_distribution_2)

        new_dist = []

        if (len(self.qubit_distribution_2)) != 0:
            for e in self.qubit_distribution_2:
                if len(e) == 0:
                    continue
                q0 = e[0]
                q1 = e[1]
                prob = e[2]
                row = np.array([q0, q1, updated_probs_for_dist_2])
                new_dist.append(row)

            self.qubit_distribution_2 = np.array(new_dist)


    def pick_one_qubit(self):
        if len(self.qubits) == 1:
            return_qubit = self.qubits[0]
            self.qubit_distribution_1 = np.array([])
            self.qubits = []
            return return_qubit

        self.remove_maxed_out_qubits()

        if len(qubits) == 0:
            return -1

        return np.random.choice(self.qubits, p=self.qubit_distribution_1)
```

```
    def pick_two_qubits(self):


        self.remove_maxed_out_qubits()

        if len(self.qubits) == 0:
            return -1, -1

        if len(self.qubit_distribution_2) == 0:
            return -1, -1

        index = np.random.choice([i for i in range(len(self.
↪qubit_distribution_2))])
        edge = self.qubit_distribution_2[index]

        # decision of which qubit should be control
        if np.random.choice([0,1]) == 0:
            return edge[0], edge[1]
        return edge[1], edge[0]


    def apply_single_qubit_gate(self, qc, gate, include_identity):

        qubit = self.pick_one_qubit()

        if qubit == -1:
            return

        if gate == 'X':
            qc.x(qubit)

        elif gate == 'Y':
            qc.y(qubit)

        elif gate == 'Z':
            qc.z(qubit)

        elif gate == 'H':
            qc.h(qubit)

        elif gate == 'S':
            qc.p(np.pi / 2, qubit)

        elif gate == 'T':
            qc.p(np.pi / 4, qubit)
```

```
        elif gate == 'RZ':
            phi = (np.pi * 2) * np.random.random()
            qc.rz(phi, qubit)

        elif gate == 'SX':
            qc.sx(qubit)

        elif gate == 'ID' and include_identity:
            qc.i(qubit)

        self.gates_on_each_qubit[qubit] += 1


    def random_circuit(self, include_identity = False):
        '''
        include_identity = If True then it actually implements the Identity␣
↪gate if it is randomly sampled
                           If False then it ignores the Identity gate if it is randomly␣
↪sampled

        note that the identity gate must be part of basis_gates
        '''

        num_classical_bits = num_qubits
        qc = QuantumCircuit(num_qubits, num_classical_bits)

        # Add support for default value: ['uniform']

        while not self.check_min_gate_threshold() and len(self.qubits) != 0:

            # decide whether to apply a single qubit gate or a 2 qubit gate
            if np.random.uniform() < probability_of_two_qubit_gate:
                # apply 2 qubit gate

                if len(self.qubits) == 1:
                    gate = np.random.choice(self.single_qubit_gates, p=self.
↪single_qubit_gate_distribution)
                    self.apply_single_qubit_gate(qc, gate, include_identity)

                else:
                    gate = np.random.choice(self.two_qubit_gates, p=self.
↪two_qubit_gate_distribution)
                    qubit1, qubit2 = self.pick_two_qubits()

                    if qubit1 == -1:
                        break
```

```
                    qubit1, qubit2 = int(qubit1), int(qubit2)

                    if gate == 'CX':
                        qc.cx(qubit1, qubit2)
                    elif gate == 'SWAP':
                        qc.swap(qubit1, qubit2)

                    self.gates_on_each_qubit[qubit1] += 1
                    self.gates_on_each_qubit[qubit2] += 1


            else:
                # apply 1 qubit gate
                gate = np.random.choice(self.single_qubit_gates, p=self.
↪single_qubit_gate_distribution)
                self.apply_single_qubit_gate(qc, gate, include_identity)

        self.print_results(qc, include_identity)


    def print_results(self, qc, include_identity):
        print('Number of qubits  : ', self.num_qubits)
        print('Min Circuit Depth : ', self.min_depth)
        print('Max Depth         : ', self.max_depth)
        print('1 Qubit Gates     : ', self.single_qubit_gates)
        print('2 Qubit Gates     : ', self.two_qubit_gates)

        print('1 Qubit Gate Distribution : ', self.
↪single_qubit_gate_distribution)
        print('2 Qubit Gate Distribution : ', self.two_qubit_gate_distribution)
        print('Qubit Distribution 1: ', self.copy_qubit_dist_1)
        print('Qubit Distribution 2:\n', self.copy_qubit_dist_2)
        print('include_identity  : ', include_identity)
        print(qc)



num_qubits = 6

topology = {
    'V': [i for i in range(num_qubits)],
    'E': [[0,1], [1,4], [4,5], [1,2], [3,4], [2,3]]
}

single_qubit_gates = ['X', 'Y', 'Z', 'H', 'S', 'T', 'ID']
two_qubit_gates    = ['CX', 'SWAP']
```

```
single_qubit_gate_distribution = [0.15, 0.2, 0.1, 0.25, 0.15, 0.05, 0.1]
two_qubit_gate_distribution     = [0.7, 0.3]

qubit_distribution_1 = np.array([0.1, 0.15, 0.25, 0.25, 0.15, 0.1])

probability_of_two_qubit_gate = 0.35

min_depth = 5
max_depth = 10

RCG = RandomCircuitGenerator(num_qubits, single_qubit_gates, two_qubit_gates,
                             single_qubit_gate_distribution,␣
 ↪two_qubit_gate_distribution,
                             qubit_distribution_1, topology, min_depth,␣
 ↪max_depth, probability_of_two_qubit_gate)

RCG.random_circuit()


'''
# For testing
for i in range(10001):
    np.random.seed(i)
    print(i)
    RCG.random_circuit()
'''

print('Process Complete!')
```

```
Number of qubits  :  6
Min Circuit Depth :  5
Max Depth         :  10
1 Qubit Gates     :  ['X', 'Y', 'Z', 'H', 'S', 'T', 'ID']
2 Qubit Gates     :  ['CX', 'SWAP']
1 Qubit Gate Distribution :  [0.15, 0.2, 0.1, 0.25, 0.15, 0.05, 0.1]
2 Qubit Gate Distribution :  [0.7, 0.3]
Qubit Distribution 1:  [0.1, 0.15, 0.25, 0.25, 0.15, 0.1]
Qubit Distribution 2:
 [[0.         1.         0.16666667]
 [1.         4.         0.16666667]
 [4.         5.         0.16666667]
 [1.         2.         0.16666667]
 [3.         4.         0.16666667]
 [2.         3.         0.16666667]]
include_identity  :  False

q_0:  Y   Y             H   X
```

```
q_1:        X   H    X     Y   X

q_2:   X   X     Z   P( /4)        X    X    H

q_3:     X   H   Z           X

q_4:        H   X    X     H

q_5:   Z   X     X        Z

c: 6/

Process Complete!
```

1.4 Testing on IBM Washington's basis gates

[6]:
```python
# Trying our generator on a set of basis gates (same as the basis gates of IBM␣
↪Washington {CX, ID, RZ, SX, X})
num_qubits = 6

topology = {
    'V': [i for i in range(num_qubits)],
    'E': [[0,1], [1,4], [4,5], [1,2], [3,4], [2,3]]
}

single_qubit_gates = ['ID', 'RZ', 'SX', 'X']
two_qubit_gates    = ['CX']

single_qubit_gate_distribution = [0.15, 0.2, 0.35, 0.3]
two_qubit_gate_distribution    = [1]

qubit_distribution_1 = np.array([0.1, 0.15, 0.25, 0.25, 0.15, 0.1])

probability_of_two_qubit_gate = 0.35

min_depth = 5
max_depth = 10

RCG = RandomCircuitGenerator(num_qubits, single_qubit_gates, two_qubit_gates,
                             single_qubit_gate_distribution,␣
↪two_qubit_gate_distribution,
                             qubit_distribution_1, topology, min_depth,␣
↪max_depth, probability_of_two_qubit_gate)

RCG.random_circuit()
```

```
Number of qubits  :  6
```

```
Min Circuit Depth :  5
Max Depth         :  10
1 Qubit Gates     :  ['ID', 'RZ', 'SX', 'X']
2 Qubit Gates     :  ['CX']
1 Qubit Gate Distribution :  [0.15, 0.2, 0.35, 0.3]
2 Qubit Gate Distribution :  [1]
Qubit Distribution 1:  [0.1, 0.15, 0.25, 0.25, 0.15, 0.1]
Qubit Distribution 2:
 [[0.         1.         0.16666667]
 [1.         4.         0.16666667]
 [4.         5.         0.16666667]
 [1.         2.         0.16666667]
 [3.         4.         0.16666667]
 [2.         3.         0.16666667]]
include_identity  :  False
                                                                        »
q_0:                                          »
                                                          »
q_1:     X      √X        √X        X      X   Rz(5.4439)  »
                                                       »
q_2:     X      X      Rz(3.8886)           X          »
                                                    »
q_3:  Rz(4.2238)   √X        √X        √X        Rz(3.218)  »
                                                     »
q_4:     X      X    X             X      X     √X    »
                                                        »
q_5:                  X      Rz(1.8231)            »
                                                                  »
c: 6/                                         »
                                                                        »
«
«q_0:      Rz(0.52968)   √X
«
«q_1:   X       X
«
«q_2:         X      X    √X   √X
«
«q_3:   X            X
«
«q_4:   √X      X
«
«q_5:             √X   X
«
«c: 6/
«

/tmp/ipykernel_239134/3153980321.py:90: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
```

```
tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant
to do this, you must specify 'dtype=object' when creating the ndarray.
  self.qubit_distribution_2 = np.array(new_dist)
```

As we can see, the testing section performs a rigorous test using random (numpy) seeds from 0 to 10000.

1.5 Future Scope

- Can introduce conditional probability to see interesting effects.
  For example, H followed by CNOT gives us an entangled state. So if we made P(CNOT | H applied before) high, then we'll see more entanglement.
- Can be used to model quantum phenomena and understand quantum effects better
- We can account for transpilation and adjust the circuit depth accordingly
- We can account for multi-qubit gates, which operate on more than 2 qubits (like the Toffoli Gate)

[7]:
```
import qiskit.tools.jupyter
%qiskit_version_table
```

```
<IPython.core.display.HTML object>
```