Task3

February 12, 2021

QOSF Mentorship - Screening Task - 3. Basic Quantum Simulator

Requirements: math, numpy, itertools

The Simulator

[1]:
```
import numpy as np
import math
from itertools import product
np.set_printoptions(precision=2)
```

For this task I made a class which works how I like, and I've made separate functions to match the task description (https://github.com/quantastica/qosf-mentorship/blob/master/qosf-simulator-task.ipynb) and parse inputs to work with my Quantum Circuit class.

Starting with the QuantumCircuit which sits behind the main functions:

[2]:
```
class QuantumCircuit():
    '''A basic quantum circuit'''
    def __init__(self, n_qubits, states=None):
        '''Initialises the qubits, given the number of qubits "n_qubits" and␣
↪the list of gates in the circuit. States default to all in the ground state␣
↪unless supplied in "states". '''
        ## First we do the qubits and statevector
        self.n_qubits = n_qubits
        self.q0 = np.array([1.0,0.0], dtype='complex')
        self.q1 = np.array([0.0,1.0], dtype='complex')
        if not isinstance(states, type(None)): # Check if states have been␣
↪provided
            self.statevector = states[0] # Need to treat the first qubit␣
↪differently
            for index in range(1,self.n_qubits):
                self.statevector = np.kron(self.statevector, states[index]) #␣
↪Iteratively build up the statevector
        else: #Defaults to all in |0> state
            self.statevector = self.q0 # Need to treat the first qubit␣
↪differently
            for index in range(1,self.n_qubits):
                self.statevector = np.kron(self.statevector, self.q0) #␣
↪Iteratively build up the statevector
```

```
        ## Initialise a list to hold the gates
        self.gates = []

    def print_statevector(self):
        '''Prints the current statevector for the quantum circuit.'''
        print("Current statevector is: ")
        print(self.statevector)
        print("")

    def get_operator(self):
        '''Calculates the overall unitary operator for the gates currently in the circuit.'''
        operator = self.gates[0]
        for gate in self.gates[1:]:
            operator = operator.dot(gate)
        return operator

    def apply_gates(self):
        '''Applies the gates that have been added to the statevector.'''
        operator = self.get_operator()
        self.statevector = operator.dot(self.statevector)

    def I(self, qubit):
        '''Adds identity gate acting on the given qubit (indexed from 0) to the list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = I
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def x(self, qubit):
        '''Adds Pauli X gate acting on the given qubit (indexed from 0) to the list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        X = np.array([[0.0, 1.0],
                      [1.0, 0.0]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = X
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
```

```python
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def y(self, qubit):
        '''Adds Pauli Y gate acting on the given qubit (indexed from 0) to the␣
↪list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        Y = np.array([[0.0, -1.0j],
                      [1.0j, 0.0]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = Y
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def z(self, qubit):
        '''Adds Pauli Z gate acting on the given qubit (indexed from 0) to the␣
↪list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        Z = np.array([[1.0, 0.0],
                      [0.0, -1.0]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = Z
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def h(self, qubit):
        '''Adds Hadamard gate acting on the given qubit (indexed from 0) to the␣
↪list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        H = 0.5*np.sqrt(2)*np.array([[1.0, 1.0],
                                     [1.0, -1.0]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = H
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)
```

```
    def sqrtx(self, qubit):
        '''Adds sqrt(X) gate (where X is Pauli X) acting on the given qubit (indexed from 0) to the list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        sqrtX = 0.5*np.array([[1.0+1.0j, 1.0-1.0j],
                              [1.0-1.0j, 1.0+1.0j]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = sqrtX
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def s(self, qubit):
        '''Adds phase shift gate S (sqrt(Z)) acting on the given qubit (indexed from 0) to the list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        S = np.array([[1.0, 0.0],
                      [0.0, 1.0j]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = S
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def sdg(self, qubit):
        '''Adds phase shift gate Sdg (S dagger) acting on the given qubit (indexed from 0) to the list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        Sdg = np.array([[1.0, 0.0],
                        [0.0, -1.0j]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = Sdg
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def t(self, qubit):
```

```
        '''Adds phase shift gate T (sqrt(S)) acting on the given qubit (indexed␣
↪from 0) to the list of gates for the circuit'''
        I = np.identity(2, dtype='complex')
        T = np.array([[1.0, 0.0],
                      [0.0, np.exp(0.25*1.0j*np.pi)]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = T
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def rz(self, phi, qubit):
        '''Adds an arbitrary phase shift gate to the list of gates for the␣
↪circuit. Phase shift gate takes an arbitrary phi in [0,2pi), and produces␣
↪the gate [[1,0],[0,e^(i*phi)]]'''
        I = np.identity(2, dtype='complex')
        rz = np.array([[1.0,0.0],
                      [0.0,np.exp(1.0j*phi)]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = rz
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)

    def u3(self, theta, phi, lam, qubit):
        '''Adds an arbitrary single qubit rotation gate to the list of gates␣
↪for the circuit. This gate applies 3 Euler angle rotations RZ( )RX(- /
↪2)RZ( )RX( /2)RZ( )'''
        I = np.identity(2)
        u3 = np.array([[np.cos(0.5*theta),-np.exp(1.0j*lam)*np.sin(0.5*theta)],
                      [np.exp(1.0j*phi)*np.sin(0.5*theta),np.exp(1.
↪0j*(phi+lam))*np.cos(0.5*theta)]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        sub_gates[qubit] = u3
        full_gate = sub_gates[0]
        for index in range(1,len(sub_gates)):
            full_gate = np.kron(full_gate, sub_gates[index])
        self.gates.append(full_gate)
```

```
    def cx(self, control_qubit, target_qubit):
        '''Adds a cx (controlled X) gate to the list of gates for the circuit.
↪'''
        I = np.identity(2, dtype='complex')
        X = np.array([[0.0, 1.0],
                      [1.0, 0.0]], dtype='complex')
        # Define projection operator |0><0|
        P0x0 = np.array([[1.0, 0.0],
                      [0.0, 0.0]], dtype='complex')
        # Define projection operator |1><1|
        P1x1 = np.array([[0.0, 0.0],
                      [0.0, 1.0]], dtype='complex')
        sub_gates = [] # Collecting all the pieces to np.kron into a list
        for qubit in range(self.n_qubits):
            sub_gates.append(I)
        off = np.copy(sub_gates) # Put into off and on, which represent the two␣
↪cases for the control
        on = np.copy(sub_gates)
        off[control_qubit] = P0x0
        on[control_qubit] = P1x1
        on[target_qubit] = X
        full_off = off[0]
        full_on = on[0]
        for index in range(1,len(off)):
            full_off = np.kron(full_off, off[index])
            full_on = np.kron(full_on, on[index])
        full_gate = full_off + full_on # Add the two control cases
        self.gates.append(full_gate)

    def arbitrary_unitary(self, unitary_matrix):
        '''Adds an arbitrary unitary matrix (np.array) to the list of gates for␣
↪the circuit. Must follow the qubit ordering of the given circuit.'''
        rows = unitary_matrix.shape[0]
        columns = unitary_matrix.shape[1]
        m = np.matrix(unitary_matrix) # Cast provided data to matrix, to ensure␣
↪the Hermitian conjugate can be taken
        is_unitary = np.allclose(np.eye(m.shape[0]), m.H * m)
        if rows == self.n_qubits and columns == self.n_qubits and is_unitary:
            self.gates.append(unitary_matrix)
        elif rows != self.n_qubits or columns != self.n_qubits:
            print("Matrix provided does not have the correct shape.")
        else:
            print("Matrix provided is not unitary.")

    def generalised_operator(self, controls, targets, thetas, phis, lams):
```

```
    '''Adds a generalised operator to the list of gates for the circuit.␣
↪User provides the control qubits, target qubits, and the input parameters␣
↪for the u3 gates that apply to the targets.
    Input parameters must each be of length 2^len(controls).'''
    I = np.identity(2, dtype='complex')
    # Define projection operator |0><0|
    P0x0 = np.array([[1.0, 0.0],
                     [0.0, 0.0]], dtype='complex')
    # Define projection operator |1><1|
    P1x1 = np.array([[0.0, 0.0],
                     [0.0, 1.0]], dtype='complex')
    full_gate = np.zeros((pow(2,self.n_qubits), pow(2,self.n_qubits))) #␣
↪Initialise the final operator
    identity_list = [] # Set up list
    for qubit in range(self.n_qubits):
        identity_list.append(I)
    control_combinations = list(product((P0x0, P1x1), repeat =␣
↪len(controls))) # Make a list of all combinations of the projection␣
↪operators for the control qubits
    for combination_index in range(len(control_combinations)): # Looping␣
↪through the elements of the sum
        combination = control_combinations[combination_index]
        sub_gates = np.copy(identity_list) # Collecting all the pieces to␣
↪np.kron into a list, defaulting to I so we can focus on control and target␣
↪qubits
        for control_index in range(len(controls)): # Matching the␣
↪projectors to the correct qubits
            control = controls[control_index]
            sub_gates[control] = combination[control_index]
        for target_index in range(len(targets)): # Matching the target␣
↪gates to the correct qubits
            target = targets[target_index]
            theta = thetas[combination_index][target_index]
            phi = phis[combination_index][target_index]
            lam = lams[combination_index][target_index]
            u3 = np.array([[np.cos(0.5*theta),-np.exp(1j*lam)*np.sin(0.
↪5*theta)],
                    [np.exp(1j*phi)*np.sin(0.5*theta),np.
↪exp(1j*(phi+lam))*np.cos(0.5*theta)]], dtype='complex')
            sub_gates[target] = u3
        sum_element = sub_gates[0]
        for index in range(1,len(sub_gates)):
            sum_element = np.kron(sum_element, sub_gates[index])
        full_gate = full_gate+sum_element # Adding this element of the␣
↪universal operator sum
    self.gates.append(full_gate)
```

```python
    def multi_shot_measurement(self, n_shots):
        '''Performs multi-shot measurement of the resulting quantum state, by sampling using the probabilities of the states (following the Born rule). Returns a dictionary of the number of cases.'''
        norm2 = (np.conjugate(np.copy(self.statevector))*self.statevector).real # Get probabilities from Born rule
        names = [''.join(row) for row in list(product('01', repeat = self.n_qubits))] # Get labels for different measurements
        samples = np.random.choice(names, size=n_shots, p=norm2)
        unique, counts = np.unique(samples, return_counts=True) # Count the number of each result
        results = dict(zip(unique, counts))
        return results
```

and then the wrapper functions which make use of the QuantumCircuit to match the task description.

[3]:

```python
def get_ground_state(n_qubits):
    '''Returns a QuantumCircuit object with n_qubits in the ground state.'''
    qpu = QuantumCircuit(n_qubits)
    return qpu

def run_program(qpu, circuit):
    '''Parses the given circuit dictionary (as defined in the task rules), applies the gates to the given qpu, and returns the final statevector.
    Params should hold the information other than the gate type and target qubits. For complex gates such as the generalised_operator, this should just be another dictionary with the variables (see below).'''
    for op in circuit:
        if op['gate'] == "generalised_operator":
            qpu.generalised_operator(op['params']['controls'], op['target'], op['params']['thetas'], op['params']['phis'], op['params']['lambdas']) # Targets in 'target', others in dictionary in 'params'
        elif op['gate'] == "arbitrary_unitary":
            qpu.arbitrary_unitary(op['params']) # Put whole unitary in 'params'
        elif op['gate'] == 'cx':
            qpu.cx(op['target'][0],op["target"][1]) # Put pulls out control and target bits
        elif op['gate'] == 'u3':
            qpu.u3(op['params']['theta'], op['params']['phi'], op['params']['lambda'], op['target'][0]) # Put variables in params
        elif op['gate'] == 'rz':
            qpu.rz(op['params'], op['target'][0]) # Put phi in 'params'
        else:
            eval('qpu.{}(op["target"][0])'.format(op['gate']))
    qpu.apply_gates()
```

```python
    return(qpu.statevector)

def measure_all(state_vector):
    '''Single measurement. I got lazy and barely changed this from the␣
↪multi-shot measurement, but it could easily be simpler.'''
    norm2 = (np.conjugate(np.copy(state_vector))*state_vector).real # Get␣
↪probabilities from Born rule
    n_qubits = int(math.log2(len(state_vector))) # Find number of qubits
    names = [''.join(row) for row in list(product('01', repeat = n_qubits))] #␣
↪Get labels for different measurements
    samples = np.random.choice(names, size=1, p=norm2)
    unique, counts = np.unique(samples, return_counts=True) # Count the number␣
↪of each result
    results = dict(zip(unique, counts))
    index = names.index(results.keys()[0])
    return index

def get_counts(state_vector, num_shots):
    '''Performs multi-shot measurement of the resulting quantum state, by␣
↪sampling using the probabilities of the states (following the Born rule).␣
↪Returns an array of the number of cases.'''
    norm2 = (np.conjugate(np.copy(state_vector))*state_vector).real # Get␣
↪probabilities from Born rule
    n_qubits = int(math.log2(len(state_vector))) # Find number of qubits
    names = [''.join(row) for row in list(product('01', repeat = n_qubits))] #␣
↪Get labels for different measurements
    samples = np.random.choice(names, size=num_shots, p=norm2)
    unique, counts = np.unique(samples, return_counts=True) # Count the number␣
↪of each result
    results = dict(zip(unique, counts))
    return results
```

Testing the simulator

Can start with the simple example provided in the exercise:

[4]:
```python
my_circuit = [
{ "gate": "h", "target": [0] },
{ "gate": "cx", "target": [0, 1] }
]

# Create "quantum computer" with 2 qubits (this is actually just a vector :) )

my_qpu = get_ground_state(2)



# Run circuit
```

```
final_state = run_program(my_qpu, my_circuit)


# Read results

counts = get_counts(final_state, 1000)

print(counts)
```

```
{'00': 500, '01': 500}
```

Which is operating as required.

Next, we can test parametric gates. Specifically, u3:

[5]:
```
my_circuit = [
              { "gate": "u3", "params": { "theta": 3.1415, "phi": 1.5708,␣
 ↪"lambda": -3.1415 }, "target": [0] }
            ]

my_qpu = get_ground_state(1)

final_state = run_program(my_qpu, my_circuit)

print(my_qpu.gates)
```

```
[array([[ 4.63e-05+0.00e+00j,  1.00e+00+9.27e-05j],
       [-3.67e-06+1.00e+00j,  4.46e-09-4.63e-05j]])]
```

With the rounding error, this matches what we expect: [[0+0j, 1+0j], [0+1j, 0+0j]]

Hopefully I'll get around to coming back to this, and refining the code, adding global variables for VQA and adding more tests.