QOSF Screening Task 3

0. Task Description

Please write a simple compiler – program, which translates one quantum circuit into another, using a restricted set of gates.

You need to consider just the basic gates for the input circuit, such as (I, H, X, Y, Z, RX, RY, RZ, CNOT, CZ).

The output circuit should consist only from the following gates: RX, RZ, CZ. In other words, each gate in the original circuit must be replaced by an equivalent combination of gates coming from the restricted set (RX, RZ, CZ) only.

For example, a Hadamard gate after compilation looks like this:

RZ(pi/2) RX(pi/2) RZ(pi/2)

Analyze what's the overhead of the compiled program compared to the original one and propose how to improve it. What we mean by overhead is the following: by replacing all the initial gates with the restricted set of gates given in the problem, you will see that the resulting circuit is much more involved than the original one. This is what we called the overhead, and you may think about how to treat this problem, i.e. you could try to simplify as much as possible the resulting circuit.

1. Gate Identities

Rotational operators are defined as:

$$R_x(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix}$$
$$R_y(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -sin(\frac{\theta}{2}) \\ sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix}$$
$$R_z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$$

It's easy to observe the following identities:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \propto R_x(\pi) \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \propto R_y(\pi) \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \propto R_z(\pi)$$

where $\propto$ denotes "proportional to", or "equivalent up to global phase"

With the famous commutation relations of X, Y and Z, we can also observe that:

$$\sigma_i\sigma_j \propto \sigma_j\sigma_i$$

where $\sigma_i, \sigma_j \in \{I, X, Y, Z\}$

We can also easily express S in terms of $R_z(\theta)$:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \propto R_z(\frac{\pi}{2})$$

Therefore, we have a way to express $R_y(\theta)$ in terms of $R_x(\theta)$ and $R_z(\theta)$ by using the Clifford property of S gate:

$$\begin{aligned} R_y(\theta) &= SR_x(\theta)S^\dagger \\ &\propto R_z(\frac{\pi}{2})R_x(\theta)R_z(-\frac{\pi}{2}) \end{aligned}$$

and also a way to express Y:

$$\begin{aligned} Y &\propto R_y(\pi) \\ &\propto R_z(\frac{\pi}{2})R_x(\pi)R_z(-\frac{\pi}{2}) \end{aligned}$$

We can also express H in terms of $R_x(\theta)$ and $R_z(\theta)$:

$$\begin{aligned} H &= \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ &\propto R_y(\frac{\pi}{2})R_z(\pi) \\ &\propto R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \end{aligned}$$

Therefore, transforming CX to CZ, $R_x(\theta)$ and $R_z(\theta)$ is also realized using the Clifford property of H gate:

$$\begin{aligned} CX_{0,1} &= H_1CZ_{0,1}H_1 \\ &\propto [R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})]_1CZ_{0,1}[R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})]_1 \\ &\propto [R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})]_1CZ_{0,1}[R_z(\pi)R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})]_1 \end{aligned}$$

We can further observe X gate is equvalent to a 180° rotation around X-axis (or simple X gate) sandwiched by two 90° around Z-axis:

$$\begin{aligned} X &= HZH \\ &\propto R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})ZR_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \\ &\propto R_z(\frac{\pi}{2})Rx(\pi)R_z(\frac{\pi}{2}) \\ &\propto R_z(\frac{\pi}{2})XR_z(\frac{\pi}{2}) \end{aligned}$$

By symmetry, we have the other 5 equations:

$$\begin{aligned} X &\propto R_y(\frac{\pi}{2})XR_y(\frac{\pi}{2}) \\ Y &\propto R_x(\frac{\pi}{2})YR_x(\frac{\pi}{2}) \\ Y &\propto R_z(\frac{\pi}{2})YR_z(\frac{\pi}{2}) \\ Z &\propto R_x(\frac{\pi}{2})ZR_x(\frac{\pi}{2}) \\ Z &\propto R_y(\frac{\pi}{2})ZR_y(\frac{\pi}{2}) \end{aligned}$$

Therefore, a new way with one less gate to express Y in terms of $R_x(\theta)$ and $R_z(\theta)$ is:

$$\begin{aligned} Y &\propto R_z(\frac{\pi}{2})R_x(\pi)R_z(-\frac{\pi}{2}) \\ &\propto R_z(\frac{\pi}{2})R_z(\frac{\pi}{2})R_x(\pi)R_z(-\frac{\pi}{2})R_z(\frac{\pi}{2}) \\ &\propto R_z(\pi)R_x(\pi) \end{aligned}$$

$$\begin{aligned} R_x(\theta) &= HR_z(\theta)H \\ &\propto R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})R_z(\theta)R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \\ &= R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\theta+\pi)R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \end{aligned}$$

$$R_z(\alpha+\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\theta+\pi)R_x(\frac{\pi}{2})R_z(\beta+\frac{\pi}{2}) \propto R_z(\alpha)R_x(\theta)R_z(\beta)$$
$$R_x(\alpha+\frac{\pi}{2})R_z(\frac{\pi}{2})R_x(\theta+\pi)R_z(\frac{\pi}{2})R_x(\beta+\frac{\pi}{2}) \propto R_x(\alpha)R_z(\theta)R_x(\beta)$$

In [1]:
```python
#initialization
import qiskit
from qiskit import execute, Aer
from qiskit.circuit.library import IGate, XGate, YGate, ZGate, HGate, SGate, RXGate, RYGate, RZGate, CXGate, CZGate
from qiskit.quantum_info import Statevector
import numpy as np
import random

pi = np.pi

I = IGate().to_matrix()
X = XGate().to_matrix()
Y = YGate().to_matrix()
Z = ZGate().to_matrix()
H = ZGate().to_matrix()
S = ZGate().to_matrix()
RX = lambda t:RXGate(t).to_matrix()
RY = lambda t:RYGate(t).to_matrix()
RZ = lambda t:RZGate(t).to_matrix()
CX = CXGate().to_matrix()
CZ = CZGate().to_matrix()
```

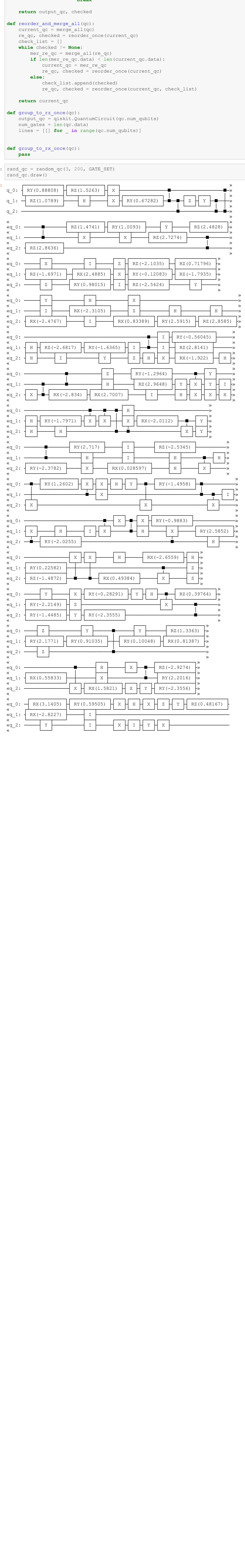
In [2]:
```python
# Helper functions
# ng = no global phase

def get_param(gate):
    return gate.params[0]

def presice_matrix(gates):
    if len(gates) == 0:
        return 1
    return gates[-1].to_matrix().dot(presice_matrix(gates[:-1]))

def remove_global_phase(mat):
    for i in range(len(mat[0])):
        if np.round(mat[0][i], 3) != 0:
            global_phase_angle = np.angle(mat[0][i])
            break
    global_phase = np.e**(global_phase_angle*1j)
    return mat/global_phase

def matrix(gates, precision=3):
    if precision == None:
        return precise_matrix(gates)
    else:
        return np.round(presice_matrix(gates), precision)

def matrix_ng(gates, precision=3):
    if len(gates) == 0:
        return 1
    mat = presice_matrix(gates)
    mat = remove_global_phase(mat)
    if precision == None:
        return mat
    else:
        return np.round(mat, precision)

def equal(gates1, gates2):
    return (matrix(gates1) == matrix(gates2)).all()

def equal_ng(gates1, gates2):
    return (matrix_ng(gates1) == matrix_ng(gates2)).all()

def equal_mat(mat1, mat2):
    return (mat1 == mat2).all()

def equal_mat_ng(mat1, mat2):
    return (remove_global_phase(mat1) == remove_global_phase(mat2)).all()

def commute(gates1, gates2):
    return equal(gates1+gates2, gates2+gates1)

def commute_ng(gates1, gates2):
    return equal_ng(gates1+gates2, gates2+gates1)

def commute_mat(mat1, mat2):
    return equal_mat(mat1.dot(mat2), mat2.dot(mat1))

def commute_mat_ng(mat1, mat2):
    return equal_mat_ng(mat1.dot(mat2), mat2.dot(mat1))

def gate_to_str(gate):
    return gate.name if len(gate.params)==0 else f'{gate.name}({np.round(gate.params[0]/pi, 3)}pi)'

def equal_circuit(qc1, qc2, precision=3):
    backend_sim = Aer.get_backend('unitary_simulator')
    job_sim = execute([qc1, qc2], backend_sim)
    result_sim = job_sim.result()
    unitary1 = result_sim.get_unitary(qc1)
    unitary2 = result_sim.get_unitary(qc2)
    return np.allclose(unitary1, unitary2)

def equal_circuit_ng(qc1, qc2, precision=3):
    s1 = Statevector.from_instruction(qc1)
    s2 = Statevector.from_instruction(qc2)
    return s1.equiv(s2)

def equal_test(gates1, gates2):
    print(f'{[gate_to_str(g) for g in gates1]} vs {[gate_to_str(g) for g in gates2]}')
    print(f'equal: {equal(gates1, gates2)}')
    print(f'equal up to global phase: {equal_ng(gates1, gates2)}')
    print()

def commute_test(gates1, gates2):
    print(f'{[gate_to_str(g) for g in gates1]} vs {[gate_to_str(g) for g in gates2]}')
    print(f'commute: {commute(gates1, gates2)}')
    print(f'commute up to global phase: {commute_ng(gates1, gates2)}')
    print()

def equal_circuit_test(qc1, qc2):
    print(f'{qc1.name} vs {qc2.name}')
    print(f'equal: {equal_circuit(qc1, qc2)}')
    print(f'equal up to global phase: {equal_circuit_ng(qc1, qc2)}')
    print()

def compare_circuits(qcs):
    print('circuit depth:')
    for qc in qcs:
        print(f'{qc.name}: {qc.depth()}')
    print()
    print('number of gates:')
    for qc in qcs:
        print(f'{qc.name}: {len(qc)}')

gates1 = [XGate()]
gates2 = [ZGate()]
commute_test(gates1, gates2)

gates1 = [YGate()]
gates2 = [XGate(), ZGate()]
equal_test(gates1, gates2)

gates1 = [RZGate(-pi/2), RXGate(pi), RZGate(pi/2)]
gates2 = [XGate(), ZGate()]
equal_test(gates1, gates2)

gates1 = [XGate()]
gates2 = [RXGate(pi)]
equal_test(gates1, gates2)

gates1 = [HGate()]
gates2 = [RZGate(pi), RYGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [HGate()]
gates2 = [RZGate(pi/2), RXGate(pi/2), RZGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RYGate(1.2345)]
gates2 = [RZGate(-pi/2), RXGate(1.2345), RZGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RXGate(5.4338)]
gates2 = [RXGate(2.2922), RXGate(pi)]
equal_test(gates1, gates2)

gates1 = [RXGate(1.2345)]
gates2 = [RZGate(pi/2), RXGate(pi/2), RZGate(1.2345+pi), RXGate(pi/2), RZGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RZGate(1.2345)]
gates2 = [RXGate(pi/2), RZGate(pi/2), RXGate(1.2345+pi), RZGate(pi/2), RXGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RYGate(pi/2), XGate(), RYGate(pi/2)]
gates2 = [XGate()]
equal_test(gates1, gates2)

gates1 = [RZGate(pi/2), XGate(), RZGate(pi/2)]
gates2 = [XGate()]
equal_test(gates1, gates2)

gates1 = [RXGate(pi/2), YGate(), RXGate(pi/2)]
gates2 = [YGate()]
equal_test(gates1, gates2)

gates1 = [RZGate(pi/2), YGate(), RZGate(pi/2)]
gates2 = [YGate()]
equal_test(gates1, gates2)

gates1 = [RXGate(pi/2), ZGate(), RXGate(pi/2)]
gates2 = [ZGate()]
equal_test(gates1, gates2)

gates1 = [RYGate(pi/2), ZGate(), RYGate(pi/2)]
gates2 = [ZGate()]
equal_test(gates1, gates2)

print(commute_mat_ng(np.kron(I, RZ(5.4321)), CZ))
print(commute_mat_ng(np.kron(RZ(5.4321), I), CZ))

#print(matrix_ng(gates1))
#print(matrix_ng(gates2))
```

```
['x'] vs ['z']
commute: False
commute up to global phase: True

['y'] vs ['x', 'z']
equal: False
equal up to global phase: True

['rz(-0.5pi)', 'rx(1.0pi)', 'rz(0.5pi)'] vs ['x', 'z']
equal: False
equal up to global phase: True

['x'] vs ['rx(1.0pi)']
equal: False
equal up to global phase: True

['h'] vs ['rz(1.0pi)', 'ry(0.5pi)']
equal: False
equal up to global phase: True

['h'] vs ['rz(0.5pi)', 'rx(0.5pi)', 'rz(0.5pi)']
equal: False
equal up to global phase: True

['ry(0.393pi)'] vs ['rz(-0.5pi)', 'rx(0.393pi)', 'rz(0.5pi)']
equal: True
equal up to global phase: True

['rx(1.73pi)'] vs ['rx(0.73pi)', 'rx(1.0pi)']
equal: True
equal up to global phase: True

['rx(0.393pi)'] vs ['rz(0.5pi)', 'rx(0.5pi)', 'rz(1.393pi)', 'rx(0.5pi)', 'rz(0.5pi)']
equal: False
equal up to global phase: True

['rz(0.393pi)'] vs ['rx(0.5pi)', 'rz(0.5pi)', 'rx(1.393pi)', 'rz(0.5pi)', 'rx(0.5pi)']
equal: False
equal up to global phase: True

['ry(0.5pi)', 'x', 'ry(0.5pi)'] vs ['x']
equal: True
equal up to global phase: True

['rz(0.5pi)', 'x', 'rz(0.5pi)'] vs ['x']
equal: True
equal up to global phase: True

['rx(0.5pi)', 'y', 'rx(0.5pi)'] vs ['y']
equal: True
equal up to global phase: True

['rz(0.5pi)', 'y', 'rz(0.5pi)'] vs ['y']
equal: True
equal up to global phase: True

['rx(0.5pi)', 'z', 'rx(0.5pi)'] vs ['z']
equal: True
equal up to global phase: True

['ry(0.5pi)', 'z', 'ry(0.5pi)'] vs ['z']
equal: True
equal up to global phase: True

True
True
```

In [3]:
```python
GATE_SET = {'i', 'h', 'x', 'y', 'z', 'rx', 'ry', 'rz', 'cx', 'cz'}
BASIS_GATE_SET = {'rx', 'rz', 'cz'}

def random_qc(num_q, num_g, gate_set):
    qc = qiskit.QuantumCircuit(num_q)
    for _ in range(num_g):
        g_name = random.choice(list(gate_set))
        if g_name[0] == 'c':
            c_index, t_index = random.sample(range(num_q), 2)
            getattr(qc, g_name)(c_index, t_index)
        elif g_name[0] == 'r':
            index = random.randint(0, num_q-1)
            param = random.uniform(-pi, pi)
            getattr(qc, g_name)(param, index)
        else:
            index = random.randint(0, num_q-1)
            getattr(qc, g_name)(index)
    return qc

def rewrite(qc):
    output_qc = qiskit.QuantumCircuit(qc.num_qubits)
    for gate_info in qc.data:
        gate = gate_info[0]
        g_name = gate.name
        index_list = gate_info[1]
        if len(index_list) == 1:
            index = index_list[0]
            if g_name == 'i':
                pass
            elif g_name == 'h':
                output_qc.rz(pi/2, index)
                output_qc.rx(pi/2, index)
                output_qc.rz(pi/2, index)
            elif g_name == 'x':
                output_qc.rx(pi, index)
            elif g_name == 'y':
                output_qc.rx(pi, index)
                output_qc.rz(pi, index)
            elif g_name == 'z':
                output_qc.rz(pi, index)
            elif g_name == 'rx':
                output_qc.rx(gate.params[0], index)
            elif g_name == 'ry':
                output_qc.rz(-pi/2, index)
                output_qc.rx(gate.params[0], index)
                output_qc.rz(pi/2, index)
            elif g_name == 'rz':
                output_qc.rz(gate.params[0], index)
        else:
            c_index, t_index = index_list
            if g_name == 'cx':
                output_qc.rz(pi/2, t_index)
                output_qc.rx(pi/2, t_index)
                output_qc.rz(pi/2, t_index)
                output_qc.rz(pi/2, t_index)
                output_qc.cz(c_index, t_index)
                output_qc.rx(pi/2, t_index)
                output_qc.rz(pi/2, t_index)
            elif g_name == 'cz':
                output_qc.cz(c_index, t_index)
    return output_qc

def merge_once(qc):
    output_qc = qiskit.QuantumCircuit(qc.num_qubits)
    num_gates = len(qc.data)
    check_list = []
    for i in range(num_gates):
        if i not in check_list:

            gate_info_i = qc.data[i]
            index_list_i = gate_info_i[1]
            gate_i = gate_info_i[0]
            g_name_i = gate_i.name

            if g_name_i in {'rz', 'rx'}:
                index_i = index_list_i[0]

                for j in range(i+1, num_gates):

                    gate_info_j = qc.data[j]
                    index_list_j = gate_info_j[1]
                    gate_j = gate_info_j[0]
                    g_name_j = gate_j.name

                    if index_i in index_list_j:
                        if g_name_j == 'cz':
                            if g_name_i == 'rz':
                                pass
                            elif g_name_i == 'rx':
                                getattr(output_qc, g_name_i)(gate_i.params[0], index_i)
                                break
                        elif g_name_j == g_name_i:
                            param_sum = gate_i.params[0]+gate_j.params[0]
                            if not np.round(param_sum/(2*pi), 3).is_integer():
                                getattr(output_qc, g_name_i)(param_sum, index_i)
                            check_list.append(j)
                            break
                        else:
                            getattr(output_qc, g_name_i)(gate_i.params[0], index_i)
                            break
                else:
                    getattr(output_qc, g_name_i)(gate_i.params[0], index_i)

            elif g_name_i == 'cz':
                for j in range(i+1, num_gates):

                    gate_info_j = qc.data[j]
                    index_list_j = gate_info_j[1]
                    gate_j = gate_info_j[0]
                    g_name_j = gate_j.name

                    set_i = set(index_list_i)
                    set_j = set(index_list_j)

                    if set_j.intersection(set_i) != set():
                        if set_i == set_j:
                            check_list.append(j)
                            break
                        else:
                            getattr(output_qc, g_name_i)(*index_list_i)
                            break
                else:
                    getattr(output_qc, g_name_i)(*index_list_i)
    return output_qc

def merge_all(qc):
    old_qc = qc
    new_qc = merge_once(old_qc)
    while len(new_qc.data) < len(old_qc.data):
        old_qc = new_qc
        new_qc = merge_once(old_qc)
    return new_qc

def reorder_once(qc, check_list=[]):
    output_qc = qc.copy()
    num_gates = len(qc.data)
    checked = None
    for i in range(num_gates):
        if i not in check_list:
            gate_info_i = output_qc.data[i]
            index_list_i = gate_info_i[1]
            gate_i = gate_info_i[0]
            g_name_i = gate_i.name

            if g_name_i in {'rz', 'rx'}:
                temp = np.round(gate_i.params[0]/pi, 3)
                if temp.is_integer() and int(temp)%2 == 1:
                    index_i = index_list_i[0]
                    for j in range(i+1, num_gates):

                        gate_info_j = output_qc.data[j]
                        index_list_j = gate_info_j[1]
                        gate_j = gate_info_j[0]
                        g_name_j = gate_j.name

                        if index_i in index_list_j:
                            if len(index_list_j)==1 and g_name_i != g_name_j:
                                temp = np.round(gate_j.params[0]/pi, 3)
                                if temp.is_integer() and int(temp)%2 == 1:
                                    output_qc.data[i], output_qc.data[j] = output_qc.data[j], output_qc
.data[i]
                                    checked = i
                            break
                    if checked != None:
                        break

    return output_qc, checked

def reorder_and_merge_all(qc):
    current_qc = merge_all(qc)
    re_qc, checked = reorder_once(current_qc)
    check_list = []
    while checked != None:
        mer_re_qc = merge_all(re_qc)
        if len(mer_re_qc.data) < len(current_qc.data):
            current_qc = mer_re_qc
            re_qc, checked = reorder_once(current_qc)
        else:
            check_list.append(checked)
            re_qc, checked = reorder_once(current_qc, check_list)

    return current_qc

def group_to_rz_once(qc):
    output_qc = qiskit.QuantumCircuit(qc.num_qubits)
    num_gates = len(qc.data)
    lines = [[] for _ in range(qc.num_qubits)]


def group_to_rx_once(qc):
    pass
```

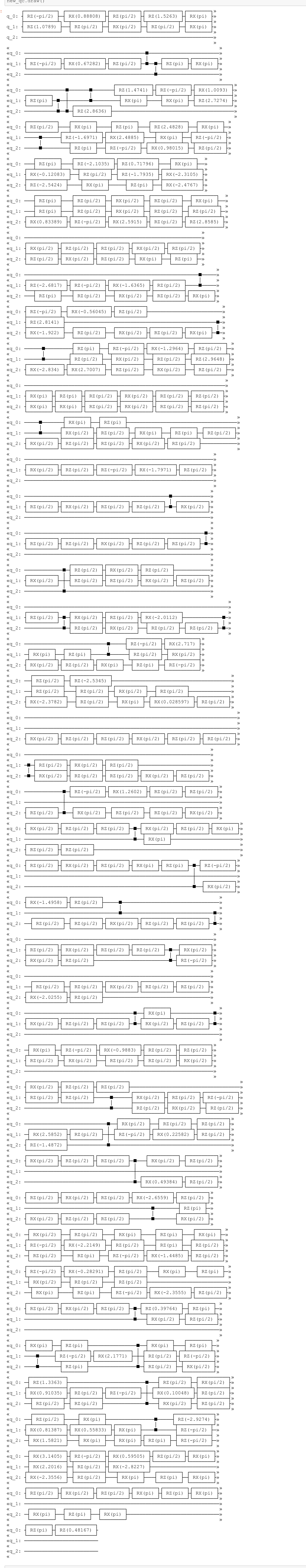
In [4]:
```python
rand_qc = random_qc(3, 200, GATE_SET)
rand_qc.draw()
```

Out[4]:



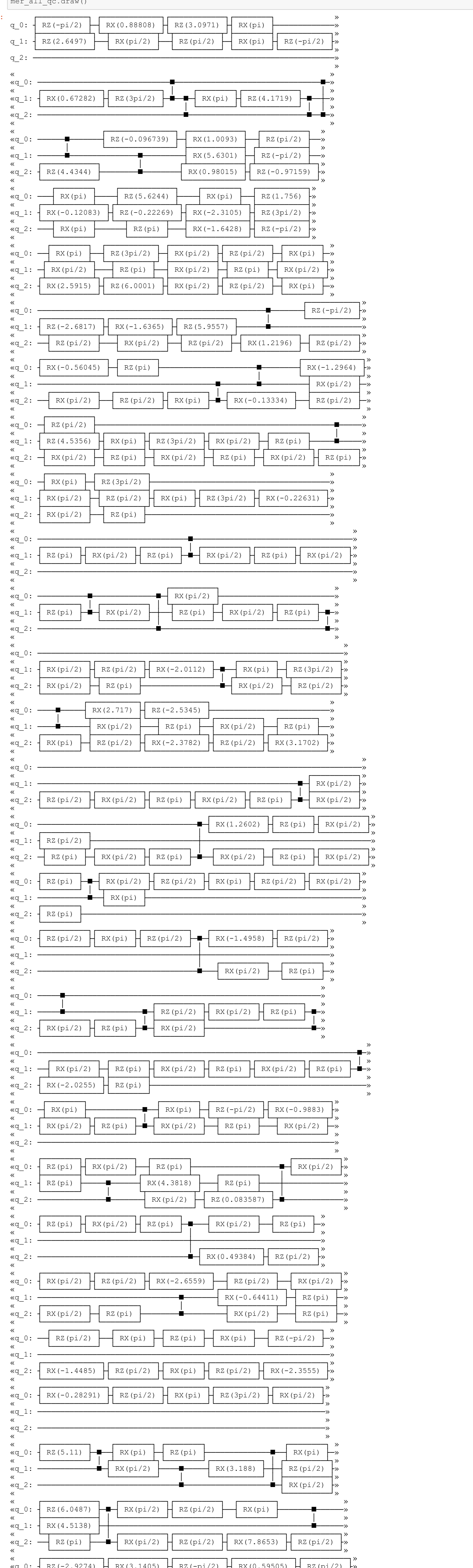
In [5]:
```
new_qc = rewrite(rand_qc)
new_qc.draw()
```

Out[5]:



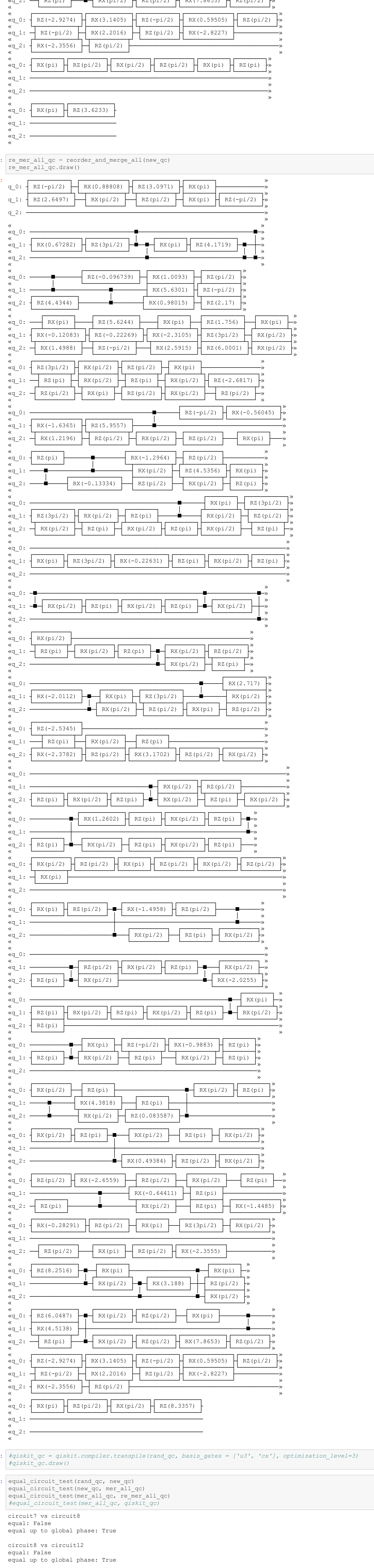
In [6]:
```
mer_all_qc = merge_all(new_qc)
mer_all_qc.draw()
```

Out[6]:



In [7]:
```
re_mer_all_qc = reorder_and_merge_all(new_qc)
re_mer_all_qc.draw()
```

Out[7]:



In [8]:
```
#qiskit_qc = qiskit.compiler.transpile(rand_qc, basis_gates = ['u3', 'cx'], optimization_level=3)
#qiskit_qc.draw()
```

In [9]:
```
equal_circuit_test(rand_qc, new_qc)
equal_circuit_test(new_qc, mer_all_qc)
equal_circuit_test(mer_all_qc, re_mer_all_qc)
#equal_circuit_test(mer_all_qc, qiskit_qc)
```

```
circuit7 vs circuit8
equal: False
equal up to global phase: True

circuit8 vs circuit12
equal: False
equal up to global phase: True

circuit12 vs circuit26
equal: False
equal up to global phase: True
```

In [10]:
```
compare_circuits([rand_qc, new_qc, mer_all_qc, re_mer_all_qc])
```

```
circuit depth:
circuit7: 107
circuit8: 232
circuit12: 172
circuit26: 162

number of gates:
circuit7: 200
circuit8: 423
circuit12: 287
circuit26: 276
```

In [ ]:

In [ ]: