Final_QC

December 19, 2023

1 Final Project of Quantum Computation

20195048 Hyunseong Kim

1.1 Problem 1

Solve the following linear system with HHL algorithm

$$\begin{bmatrix} 1 & -1/3 \\ -1/3 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

1.2 Problem 2

Extend the problem 1 result and solve following equation

$$\frac{1}{4}\begin{bmatrix} 15 & 9 & 5 & -3 \\ 9 & 15 & 3 & -5 \\ 5 & 3 & 15 & -9 \\ -3 & -5 & -9 & 15 \end{bmatrix} \mathbf{x} = \frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

1.3 Answer

Since the matrix already a Hermit matrix, the calculation does not require a conversion. Before we implement HHL. Let us see the exact solutions of the each system.

1.3.1 Exact solutions

[92]:
```
from IPython.display import display, Math
```

[93]:
```
import sympy as sp
import numpy as np
from sympy.printing.mathml import mathml
```

[94]:
```
scale1 = 3
A1 = sp.Matrix([[3, -1], [-1, 3]])/scale1
b1 = sp.Matrix([[0], [1]])
scale2 = 4
A2 = sp.Matrix(
    [
```

```
        [15, 9, 5, -3],
        [9, 15, 3, -5],
        [5, 3, 15, -9],
        [-3, -5, -9, 15]])/scale2
b2 = sp.Matrix(4*[[2]])/scale2

A1_np = np.array(A1).astype(complex)
A2_np = np.array(A2).astype(complex)
```

[95]:
```
# matrix , determinant and exact solutions
display(Math(sp.latex(A1)))
display(Math(f"$det(A_1): {A1.det()}$",))
display(Math("$\mathbf{x}: " + sp.latex(A1.solve(b1).T)))
display(Math(sp.latex(A2)))
display(Math(f"$det(A_2): {A2.det()}$",))
display(Math("$\mathbf{x}: " + sp.latex(A2.solve(b2).T)))
```

$$\left[\begin{matrix}1 & -\frac{1}{3}\\-\frac{1}{3} & 1\end{matrix}\right]$$

$$det(A_1): 8/9$$

$$\mathbf{x}: \left[\begin{matrix}\frac{3}{8} & \frac{9}{8}\end{matrix}\right]$$

$$\left[\begin{matrix}\frac{15}{4} & \frac{9}{4} & \frac{5}{4} & -\frac{3}{4}\\\frac{9}{4} & \frac{15}{4} & \frac{3}{4} & -\frac{5}{4}\\\frac{5}{4} & \frac{3}{4} & \frac{15}{4} & -\frac{9}{4}\\-\frac{3}{4} & -\frac{5}{4} & -\frac{9}{4} & \frac{15}{4}\end{matrix}\right]$$

$$det(A_2): 64$$

$$\mathbf{x}: \left[\begin{matrix}-\frac{1}{32} & \frac{7}{32} & \frac{11}{32} & \frac{13}{32}\end{matrix}\right]$$

In the HHL the solution vector would be encoded as an amplitude of the state vector. That means the measurement result would show square ratio of the solution vectors.

In problem 1; $\mathbf{x} = [1, 3]\frac{3}{8}$, the solution from the HHL would be $[1, 9]$. In problem 2; $\mathbf{x} = [-1, 7, 11, 13]\frac{1}{32}$, the solution would be $[1, 49, 121, 169]$. Now, the problem 1 configuration is well shown in the reference. The below is a material to get a problem 2 configuration.

[96]:
```
# A2 eigenvalues and vectors
display(A2.eigenvals())
display(Math(sp.latex(A2.eigenvects())))
```

```
{8: 1, 4: 1, 2: 1, 1: 1}
```

$$\left[\left(1, \ 1, \ \left[\left[\begin{matrix}-1\\1\\1\\1\end{matrix}\right]\right]\right), \ \left(2, \ 1, \ \left[\left[\begin{matrix}1\\-1\\1\\1\end{matrix}\right]\right]\right), \ \left(4, \ 1, \ \left[\left[\begin{matrix}1\\1\\-1\\1\end{matrix}\right]\right]\right), \ \left(8, \ 1, \ \left[\left[\begin{matrix}-1\\-1\\-1\\1\end{matrix}\right]\right]\right)\right]$$

[166]:
```
evlist = (e1, e2, e3 ,e4) = [np.array(c[0]).astype(complex) for (a, b, c) in A2.
  ↪eigenvects()]
eval2 = list(A2.eigenvals().keys())
(eval2).reverse()
print("Eigenvalues of A2: ", eval2)
```

```
Eigenvalues of A2:  [1, 2, 4, 8]
```

1.4 System configuration

Eigenvalues and eigenvectos of each systems

- $\{\lambda_i\}_{A_1} = \{\frac{2}{3}, \frac{4}{3}\}$
- $\{\lambda_i\}_{A_2} = \{1, 2, 4, 8\}$

First system requires 2 qubits for representing the eigenvalues 01, 10, and the second system requires 4 qubit registers, as 0001, 0010, 0100, 1000. Therefore, $N_1 = 2^2, N_2 = 2^4$.
Consequently, the desired scaled eigenvalues and the scale factors are

$$\tilde{\lambda} = N\frac{\lambda}{2\pi}t$$

- $\{\tilde{\lambda}_i\}_{A_1} = \{1, 2\}$
- $\{\tilde{\lambda}_i\}_{A_2} = \{1, 2, 4, 8\}$
- $t_1 = \frac{3}{4}\pi$
- $t_2 = \frac{1}{8}\pi$

Lastly, RY rotation angle is

$$\theta_i = 2\arcsin(\frac{1}{2^i})\text{for } i \in [0, n_{clock})$$

therefore,

- $[\pi, \pi/3]$
- $[\pi, \pi/3, 2\arcsin(2^{-2}), 2\arcsin(2^{-3})]$

[98]:
```
t1 = 3/4 * np.pi
t2 = 1/8 * np.pi
```

1.4.1 Implementation of HHL algorithm

Major steps of HHL algorithm

3 registers required: ancilla, main-register, b-register.

1. State preparation: Encoding $|b\rangle$ vector to the register, amplitude encoding(b).
2. Quantum Phase estimation(main, b).
3. Ancilla bit rotation(main, ancilla).
4. Inverse Quantum Phase estimation(main, b).
5. Measurement.

1.4.2 Decomposition of Pauli matrix

In the HHL algorithm there is a evolution circuit $U = \exp(iHt)$ gate of the given A matrix. Unfortunately, to implement on the gate circuit, we have to encode the given Hamiltonian as summation of unitary and hermit matrices. That means we needs a Pauli-polynomial of the given matrix.

Since, we can construct general unitary gate in qiskit. However, if we want to make user-defined evolution circuit. It would be useful.

[99]:
```
import numpy as np
from itertools import combinations, combinations_with_replacement as re_combi,␣
↪product
from functools import reduce

I = np.eye(2)
X = np.array([[0, 1], [1, 0]])
Y = complex(0, 1)*np.array([[0, -1], [1, 0]])
Z = np.array([[1, 0], [0, -1]])
```

[167]:
```
def krons(oper_list): # Operator Kronecker delta
    return reduce(np.kron, oper_list)
def get_pauli_xz_family_n_qubit(n, fam="Z"): # Get X-Z families of Pauli-groups
    G = Z if fam=="Z" else X
    return list(map(krons, product([I, G], repeat=int(n)))), list(map(lambda x:␣
↪"".join(x), product(f"I{fam}", repeat=int(n))))
def get_pauli_familiy(n): # Get total Pauli elements of the given n number of␣
↪qubit system.
    p_xs, p_xs_str = get_pauli_xz_family_n_qubit(n, fam="X")
    p_zs, p_zs_str = get_pauli_xz_family_n_qubit(n, fam="Z")

    p_g = []
    p_g_str =[]
    for x_i, x_str in zip(p_xs, p_xs_str):
        for z_j, z_str in zip(p_zs, p_zs_str):
            g = x_i@z_j

            g_coef, g_str = get_coef(x_str, z_str)

            p_g.append(g_coef*g)
            p_g_str.append(g_str)
    return p_g, p_g_str

def get_coef(x_str, z_str): # i coefficient in construction of general␣
↪pauli-element from XZ elements.
    n = len(x_str)
    x_str = x_str.replace("X", "1")
    x_str = x_str.replace("I", "0")
```

```
    z_str = z_str.replace("Z", "1")
    z_str = z_str.replace("I", "0")

    x_int = int(x_str, 2)
    z_int = int(z_str, 2)

    y_pos = format(x_int&z_int, f"0{n}b")
    z_pos = format((x_int|z_int) - x_int, f"0{n}b")
    x_pos = format((x_int|z_int) - z_int, f"0{n}b")

    g_str = []
    for x,y,z in zip(x_pos, y_pos, z_pos):
        if x==y and y==z:
            g_str.append("I")
        elif x== "1":
            g_str.append("X")
        elif y == "1":
            g_str.append("Y")
        else:
            g_str.append("Z")
    return 1j**y_pos.count("1"), "".join(g_str)

def frobenius_inner(A, B): # Frobenius inner product.
    n, n2 = A.shape
    return np.trace((A.conj().T)@B)/(n)

def get_pauli_coefficient(A): # Get Pauli coefficients of the given Hermit␣
 ↪matrix A.
    k, k2 = A.shape
    n = int(np.log2(k))
    p_fam, p_str = get_pauli_familiy(n)

    coef = {}
    for p_m, p_m_str in zip(p_fam, p_str):
        coef[p_m_str] = frobenius_inner(p_m, A)
    return coef
```

[101]:
```
coef_A1 = get_pauli_coefficient(A1_np)
coef_A2 = get_pauli_coefficient(A2_np)
```

[102]:
```
coef_A1
```

[102]:
```
{'I': (1+0j), 'Z': 0j, 'X': (-0.3333333333333333+0j), 'Y': 0j}
```

[103]:
```
coef_A2
```

[103]:
```
{'II': (3.75+0j),
 'IZ': 0j,
 'ZI': 0j,
 'ZZ': 0j,
 'IX': 0j,
 'IY': 0j,
 'ZX': (2.25+0j),
 'ZY': 0j,
 'XI': 0j,
 'XZ': (1.25+0j),
 'YI': 0j,
 'YZ': 0j,
 'XX': 0j,
 'XY': 0j,
 'YX': 0j,
 'YY': (0.75+0j)}
```

The A_1 and A_2 become

$$A_1 = I - \frac{1}{3}X$$
$$A_2 = 3.75II + 1.25XZ + 2.25ZX + 0.75YY$$

About the commuting relationship, A_1 components are tirvially commuting each other. For A_2 elements, we can use a method suggested by Reggio et al (2023).

`Ben Reggio, Nouman Butt, Andrew Lytle, and Patrick Draper. Fast Partitioning of Pauli Strings into Commuting Families for Optimal Expectation Value Measurements of Dense Operators, June 2023. arXiv:2305.11847 [hep-lat, physics:hep-ph, physics:quant-ph].`

Theorem > For two pauli strings, there are X, Z family decomposition such as $P_1 = x_1 * z_1$ and $P_2 = x_2 * z_2$. The given Pauli strings are commuting if and only if $[x_1, z_2]$ and $[x_2, z_1]$ are either commute or anti-commute.

Now see below,

Element	X-family	Z_family
XZ	XI	IZ
ZX	IX	ZI
YY	XX	ZZ

All of them are anticommute, in cross line elements of X, Z family columns. Therefore, we don't have to worry about the anticommute relationship on implementation.

[104]:
```python
# Build evolution unitary matrices.
# U1
I =sp.eye(2)
```

```
X = sp.Matrix([[0,1] ,[1,0] ])
ai = 1*t1
ax = sp.Number(1/3)*t1
U1 = np.array(sp.exp(1j * ai * I) @ sp.exp(1j * ax * X)).astype(complex) # You␣
↪can use trotterization circuit but it becomes too complicated.
U1_inv = np.linalg.inv(U1)
U1
```

[104]:
```
array([[-0.5+0.5j, -0.5-0.5j],
       [-0.5-0.5j, -0.5+0.5j]])
```

[105]:
```
# U2 construction from Pauli polynomial
II = sp.kronecker_product(sp.eye(2), sp.eye(2))
XZ = sp.kronecker_product(sp.Matrix(X), sp.Matrix(Z))
ZX = sp.kronecker_product(sp.Matrix(Z), sp.Matrix(X))
YY = sp.kronecker_product(sp.Matrix(Y), sp.Matrix(Y))
aii = 3.75*t2
axz = 1.25*t2
azx = 2.25*t2
ayy = 0.75*t2
U2 = np.array(sp.exp(1j * aii * II) @ sp.exp(1j * azx * ZX) @ sp.exp(1j * axz *␣
↪XZ) @sp.exp(1j * ayy * YY)).astype(complex)
U2_inv = np.linalg.inv(U2)
U2
```

[105]:
```
array([[ 0.15774658+0.52244755j, -0.65774658-0.02244755j,
        -0.30419319-0.16889416j,  0.19580681+0.33110584j],
       [-0.65774658-0.02244755j,  0.15774658+0.52244755j,
        -0.19580681-0.33110584j,  0.30419319+0.16889416j],
       [-0.30419319-0.16889416j, -0.19580681-0.33110584j,
         0.15774658+0.52244755j,  0.65774658+0.02244755j],
       [ 0.19580681+0.33110584j,  0.30419319+0.16889416j,
         0.65774658+0.02244755j,  0.15774658+0.52244755j]])
```

For convinence, the below code used `UnitaryGate` routine from Qiskit extension. It is a grammar sugar of qiskit. Since, we get a decomposed representation of the A matrices of each problems, let us find a corresponding `CNOT` and single gate combination.

$$U = \exp(i\hat{n} \cdot \sigma\theta) = \cos(\theta)\hat{I} + i\sin(\theta)\hat{n} \cdot (\sigma)$$

The X, Y, Z pauli gates are well known, however, what about the II rotation? Is it worth to mark it? The answer is YES. In the n-qubit Hamiltonian, $\mathrm{I}^{\otimes n}$ affects nothing to the whole system. It just yields a global phase difference. However, with `CNOT` and the other controll gates in n+1 system, it becomes local relative phase difference of the system. For example, think about Ahrnov-Bohm effect in splitted two beam and combine at the end of the optical path. The difference only arise to a phase of one of the beam the effect is well observed in final measurement.

Mathematically, the answer is how we can find a unitary matrix combination of the next large matrix being combined with tensor product.

$$
\text{C-}\exp(ia\hat{\text{I}}\text{I}\theta) = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \exp(ia\theta) & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \exp(ia\theta) & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \exp(ia\theta) & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \exp(ia\theta)
\end{pmatrix}
$$

Looks complicated, but the answer is simple.

$$P(a\theta) \otimes I \otimes I$$

---

[518]:
```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.circuit.library import QFT
from qiskit import QuantumCircuit, Aer, execute
from qiskit.quantum_info import partial_trace, Operator
from qiskit.visualization import plot_histogram
#from qiskit.quantum_info.operators import Operator
from qiskit.extensions import UnitaryGate
```

1.5 Problem 1

[538]:
```
from qiskit.extensions import UnitaryGate, CSGate
```

[539]:
```
y_theta1 = [np.pi, np.pi/3]
U1_1 = U1
U1_2 = U1@U1
U1_1_inv = U1_inv
U1_2_inv = U1_inv@U1_inv
```

[540]:
```
cu11     = UnitaryGate(U1_1, label="U11").control()
cu12     = UnitaryGate(U1_2, label="U12").control()
cu11_inv = UnitaryGate(U1_1_inv, label="U11_inv").control()
cu12_inv = UnitaryGate(U1_2_inv, label="U12_inv").control()
```

[541]:
```
num_ancilla_register = 1
num_clock_register = 2
num_vector_register = 1
```

[542]:
```python
qr_ancilla = QuantumRegister(num_ancilla_register, name="ancilla")
qr_clock = QuantumRegister(num_clock_register, name="clock")
qr_vector = QuantumRegister(num_vector_register, name="vector")
cr = ClassicalRegister(2, name = "classic")
```

[543]:
```python
qc = QuantumCircuit(qr_ancilla, qr_clock, qr_vector, cr, name="HHL")
```

[544]:
```python
qc.x(qr_vector)
qc.barrier() # QPE initiate
qc.h(qr_clock[:])

qc.append(cu11, [qr_clock[0],  qr_vector[0]])
qc.append(cu12, [qr_clock[1],  qr_vector[0]])

qc.append(QFT(2).inverse(), qr_clock[:])

for i, angle in zip(range(0, len(qr_clock)), y_theta1):
    qc.cry(angle, qr_clock[i], qr_ancilla)

qc.measure(qr_ancilla, cr[0])

qc.append(QFT(2), qr_clock[:])

qc.append(cu12_inv, [qr_clock[1],  qr_vector[0]])
qc.append(cu11_inv, [qr_clock[0],  qr_vector[0]])

qc.h(qr_clock[:])
qc.barrier()
```

[544]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x1679cb3da20>
```

[545]:
```python
qc.draw()
```

[545]:
```
                                                                  »
  ancilla:                            Ry( )    Ry( /3)  M      »
                                                                »
  clock_0:       H          0                  0       »
                                       IQFT                             QFT  »
  clock_1:       H          1                  1       »
                                                                   »
   vector:   X        U11    U12                        »
                                                                       »
classic: 2/                                       »
                                                                       0         »
«
«  ancilla:
«
```

```
«  clock_0:              H
«
«  clock_1:              H
«
«   vector:   U12_inv   U11_inv
«
«classic: 2/
«
```

1.5.1 Result 1

[546]:
```
# Statevector
state_vector_sim = Aer.get_backend('statevector_simulator')
result_state1 = execute(qc, state_vector_sim).result()
state_vec_1 = result_state1.get_statevector()
state_vec_1.data
```

[546]:
```
array([ 0.00000000e+00+0.00000000e+00j, -3.16227766e-01+2.80088034e-16j,
       -0.00000000e+00+0.00000000e+00j, -2.19473543e-18-1.62451342e-16j,
        0.00000000e+00+0.00000000e+00j, -1.18879543e-17+2.40907438e-15j,
        0.00000000e+00-0.00000000e+00j, -1.15635141e-15-8.64160897e-17j,
        0.00000000e+00+0.00000000e+00j,  9.48683298e-01-4.51609925e-16j,
        0.00000000e+00-0.00000000e+00j, -2.01902859e-16-1.86602849e-15j,
       -0.00000000e+00+0.00000000e+00j, -1.79005560e-16+4.08789170e-15j,
       -0.00000000e+00+0.00000000e+00j,  5.27025844e-16-4.81763963e-17j])
```

[547]:
```
x0 = state_vec_1.data[1] # 01
x1 = state_vec_1.data[9] # 11
x0_n = x0.conj()*x0
x1_n = x1.conj()*x1

x0_n, x1_n
```

[547]:
```
((0.09999999999999994+0j), (0.9000000000000001+0j))
```

[548]:
```
x1_n/x0_n
```

[548]:
```
(9.000000000000007+0j)
```

Which was predicted at first $\mathbf{x} = [\frac{3}{8}, \frac{9}{8}]^T$

[549]:
```
# Measurement
qc.measure(qr_ancilla, cr[0])
qc.measure(qr_vector , cr[1])
```

[549]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x1679b991480>
```

[550]:
```
qc.draw()
```

[550]:

```
                                                                    »
  ancilla:                    Ry( )   Ry( /3)  M     »
                                                       »
  clock_0:      H         0                   0      »
                                    IQFT                         QFT  »
  clock_1:      H         1                   1      »
                                                        »
   vector:   X        U11   U12                    »
                                                                    »
classic: 2/                              »
                                                                    0        »
«
«  ancilla:                M
«
«  clock_0:             H
«
«  clock_1:             H
«
«   vector:   U12_inv   U11_inv        M
«
«classic: 2/
«                                         0   1
```

[551]:

```
shots =10000
simulator = Aer.get_backend('qasm_simulator')
```

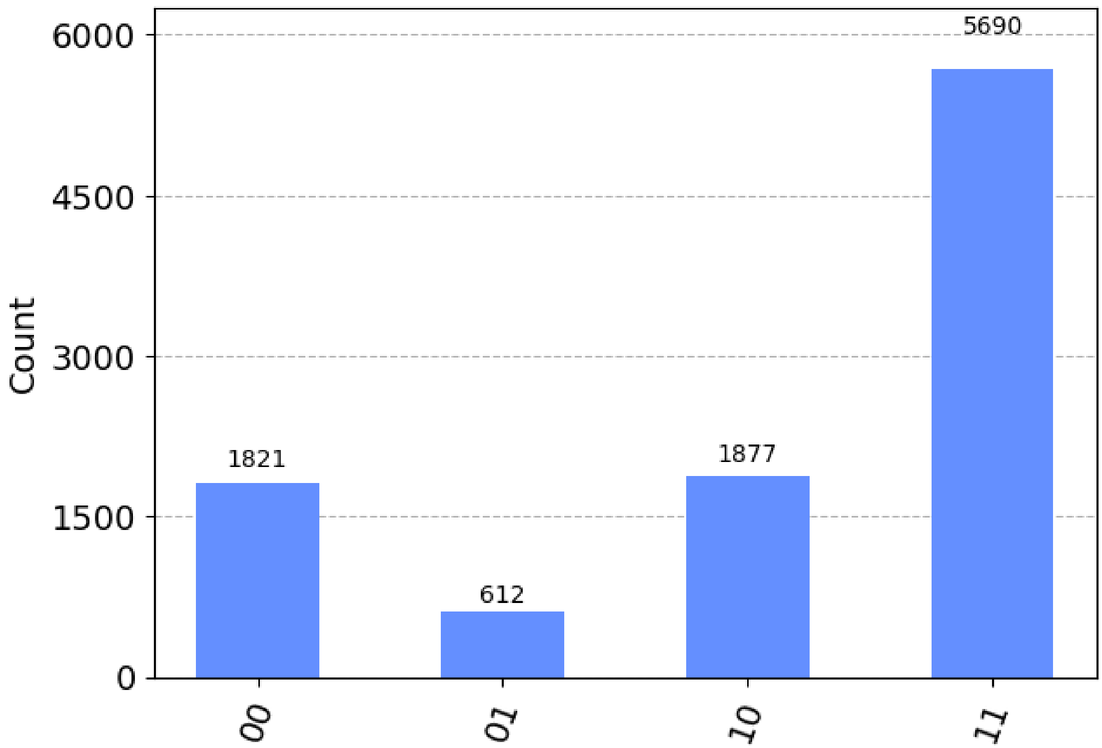
[552]:

```
result = execute(qc, simulator, shots=shots).result()
counts = result.get_counts(qc)
```

[553]:

```
plot_histogram(counts)
```

[553]:



[554]:
```
counts
```

[554]: {'01': 612, '10': 1877, '00': 1821, '11': 5690}

[555]:
```
x0_result = np.array((counts["01"]/ shots,counts["11"]/shots))
x0_result /= x0_result[0]

print("HHL result: ", np.sqrt(x0_result)*3/8)
display(Math("$\mathbf{x}: " + sp.latex(A1.solve(b1).T)))
```

HHL result: [0.375   1.14343555]

$\mathbf{x}: \left[\begin{matrix}\frac{3}{8} & \frac{9}{8}\end{matrix}\right]$

□

1.6 Problem 2

[134]:
```
from qiskit.circuit.library import CPhaseGate
```

[135]:
```
y_theta2 = [2*np.arcsin(1/(2**i)) for i in range(0,4)]
U2_1 = U2
U2_2 = U2@U2
U2_3 = U2_2@U2_2
```

```
U2_4 = U2_3@U2_3
U2_1_inv = U2_inv
U2_2_inv = U2_1_inv@U2_1_inv
U2_3_inv = U2_2_inv@U2_2_inv
U2_4_inv = U2_3_inv@U2_3_inv
```

[165]:
```
U2
```

[165]:
```
array([[ 0.15774658+0.52244755j, -0.65774658-0.02244755j,
         -0.30419319-0.16889416j,  0.19580681+0.33110584j],
        [-0.65774658-0.02244755j,  0.15774658+0.52244755j,
         -0.19580681-0.33110584j,  0.30419319+0.16889416j],
        [-0.30419319-0.16889416j, -0.19580681-0.33110584j,
          0.15774658+0.52244755j,  0.65774658+0.02244755j],
        [ 0.19580681+0.33110584j,  0.30419319+0.16889416j,
          0.65774658+0.02244755j,  0.15774658+0.52244755j]])
```

[137]:
```
y_theta2
```

[137]:
```
[3.141592653589793, 1.0471975511965976, 0.5053605102841573, 0.2506556623361308]
```

[158]:
```
UnitaryGate(np.linalg.matrix_power(U2_1, 4)).control()
```

[158]:
```
Instruction(name='c-unitary', num_qubits=3, num_clbits=0, params=[array([[
0.25+0.25j,  0.75-0.25j, -0.25-0.25j, -0.25-0.25j],
        [ 0.75-0.25j,  0.25+0.25j,  0.25+0.25j,  0.25+0.25j],
        [-0.25-0.25j,  0.25+0.25j,  0.25+0.25j, -0.75+0.25j],
        [-0.25-0.25j,  0.25+0.25j, -0.75+0.25j,  0.25+0.25j]])])
```

[138]:
```
cu21     = UnitaryGate(U2_1, label="U21").control()
cu22     = UnitaryGate(U2_2, label="U22").control()
cu23     = UnitaryGate(U2_3, label="U23").control()
cu24     = UnitaryGate(U2_4, label="U24").control()
cu21_inv = UnitaryGate(U2_1_inv, label="U21_inv").control()
cu22_inv = UnitaryGate(U2_2_inv, label="U22_inv").control()
cu23_inv = UnitaryGate(U2_3_inv, label="U23_inv").control()
cu24_inv = UnitaryGate(U2_4_inv, label="U24_inv").control()
```

```
/Users/hyunseongkim/Documents/   /.conda/lib/python3.11/site-
packages/numpy/linalg/linalg.py:2180: RuntimeWarning: divide by zero encountered
in det
  r = _umath_linalg.det(a, signature=signature)
/Users/hyunseongkim/Documents/   /.conda/lib/python3.11/site-
packages/numpy/linalg/linalg.py:2180: RuntimeWarning: invalid value encountered
in det
  r = _umath_linalg.det(a, signature=signature)
```

[139]:
```python
# System Configuration
num_ancilla_register = 1
num_clock_register = 4
num_vector_register = 2
```

[140]:
```python
# Quantum circuit register defintion
qr_ancilla2 = QuantumRegister(num_ancilla_register, name="ancilla")
qr_clock2 = QuantumRegister(num_clock_register, name="clock")
qr_vector2 = QuantumRegister(num_vector_register, name="vector")
cr2 = ClassicalRegister(3, name = "classic")
```

[141]:
```python
qc2 = QuantumCircuit(qr_ancilla2, qr_clock2, qr_vector2, cr2, name="HHL_2")
```

[142]:
```python
# Circuit configuration
qc2.h(qr_vector2) # Initiate b vector

qc2.barrier() # QPE initiate
qc2.h(qr_clock2)

qc2.append(cu21, [qr_clock2[0],  qr_vector2[0], qr_vector2[1]])
qc2.append(cu22, [qr_clock2[1],  qr_vector2[0], qr_vector2[1]])
qc2.append(cu23, [qr_clock2[2],  qr_vector2[0], qr_vector2[1]])
qc2.append(cu24, [qr_clock2[3],  qr_vector2[0], qr_vector2[1]])

qc2.append(QFT(4).inverse(), qr_clock2[:])

qc2.barrier()
for i, angle in zip(range(0, len(qr_clock2)), y_theta2):
    qc2.cry(angle, qr_clock2[i], qr_ancilla2)

qc2.measure(qr_ancilla2, cr2[0])
qc2.barrier()
# QFT
qc2.append(QFT(4), qr_clock2[:])
#
qc2.append(cu24_inv, [qr_clock2[3],  qr_vector2[0], qr_vector2[1]])
qc2.append(cu23_inv, [qr_clock2[2],  qr_vector2[0], qr_vector2[1]])
qc2.append(cu22_inv_2, [qr_clock2[1],  qr_vector2[0], qr_vector2[1]])
qc2.append(cu21_inv, [qr_clock2[0],  qr_vector2[0], qr_vector2[1]])
qc2.barrier()
qc2.h(qr_clock2[:])
qc2.barrier()
```

[142]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x28cd4f6a0>
```

[143]:
```python
qc2.draw()
```

[143]:

```
                                                                        »
  ancilla:                                Ry( )  »
                                                                  »
  clock_0:     H                 0           »
                                                                        »
  clock_1:     H                 1           »
                                                   IQFT                 »
  clock_2:     H                 2           »
                                                                       »
  clock_3:     H                 3           »
                                              »
 vector_0:   H      0      0      0      0              »
                         U21    U22    U23    U24                     »
 vector_1:   H      1      1      1      1              »
                                                        »
classic: 3/                               »
                                                                            »
«                                                          »
«  ancilla:   Ry( /3)   Ry(0.50536)   Ry(0.25066)   M           »
«                                                       »
«  clock_0:                            0           »
«                                                                          »
«  clock_1:                            1           »
«                                                          QFT              »
«  clock_2:                            2           »
«                                                                           »
«  clock_3:                            3           »
«                                                                     »
« vector_0:                                0          »
«                                                                 U24_inv  »
« vector_1:                                1          »
«                                                                     »
«classic: 3/                               »
«                                                     0                     »
«
«  ancilla:
«
«  clock_0:                        H
«                          Test
«  clock_1:                        H
«
«  clock_2:                        H
«
«  clock_3:                        H
«
« vector_0:  0          0          0
«               U23_inv    Unitary    U21_inv
```

```
« vector_1:  1          1          1
«
«classic: 3/
«
```

[124]:
```
state_vector_sim = Aer.get_backend('statevector_simulator')
simulator = Aer.get_backend('qasm_simulator')
```

[565]:
```
while True:
    result = execute(qc2, state_vector_sim, shots=1).result()
    counts = result.get_counts(qc2)
    if '001' in counts:
        break
```

[566]:
```
state_vec_2 = result.get_statevector(decimals=3)
```

[567]:
```
values = state_vec_2.data
```

[568]:
```
state_vec_2
```

```
Statevector([ 0.   +0.j, -0.054-0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j,
             -0.   -0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j, -0.   -0.j,
              0.   +0.j,  0.   +0.j,  0.   +0.j, -0.   +0.j,  0.   +0.j,
              0.   -0.j,  0.   +0.j,  0.   -0.j,  0.   +0.j,  0.   +0.j,
              0.   +0.j, -0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,
              0.   -0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j, -0.   +0.j,
              0.   +0.j, -0.   +0.j,  0.   +0.j,  0.38 +0.j,  0.   +0.j,
              0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j, -0.   -0.j,
              0.   +0.j,  0.   -0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j,
             -0.   -0.j,  0.   +0.j,  0.   -0.j,  0.   +0.j,  0.   -0.j,
              0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,
              0.   +0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j,  0.   -0.j,
              0.   +0.j,  0.   -0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j,
              0.597+0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   -0.j,
              0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,
              0.   -0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j, -0.   +0.j,
              0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,
             -0.   -0.j,  0.   +0.j,  0.   -0.j,  0.   +0.j, -0.   +0.j,
              0.   +0.j,  0.   -0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j,
              0.   -0.j,  0.   +0.j,  0.705+0.j,  0.   +0.j,  0.   -0.j,
              0.   +0.j,  0.   -0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j,
              0.   +0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j, -0.   -0.j,
              0.   +0.j,  0.   -0.j,  0.   +0.j,  0.   +0.j,  0.   +0.j,
             -0.   -0.j,  0.   +0.j, -0.   +0.j,  0.   +0.j,  0.   -0.j,
              0.   +0.j, -0.   +0.j,  0.   +0.j, -0.   -0.j,  0.   +0.j,
              0.   -0.j,  0.   +0.j, -0.   +0.j],
            dims=(2, 2, 2, 2, 2, 2, 2))
```

[572]:
```python
# Get Probability of the states
state_amplitude = state_vec_2.data.conj()*state_vec_2.data
measure = state_amplitude/state_amplitude[1]
print("Measure probability from state vector")
measure[measure>0]
```

```
Measure probability from state vector
```

[572]:
```
array([  1.        +0.j,  49.51989026+0.j, 122.22530864+0.j,
       170.44753086+0.j])
```

[571]:
```python
# Get solution
result_values = np.array([values[1], values[33], values[65], values[97]])
result_vector = -result_values/result_values.min()
print("Solution from state vector")
result_vector
```

```
Solution from state vector
```

[571]:
```
array([-1.        -0.j,  7.03703704-0.j, 11.05555556-0.j, 13.05555556-0.j])
```

[573]:
```python
qc2.measure(qr_ancilla2, cr2[0])
qc2.measure(qr_vector2[0], cr2[1])
qc2.measure(qr_vector2[1], cr2[2])
```

[573]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x1679daf4f40>
```

[574]:
```python
qc2.draw()
```

[574]:
```
                                                                      »
   ancilla:                                 Ry( )  »
                                                                »
   clock_0:      H                  0              »
                                                                      »
   clock_1:      H                  1              »
                                                      IQFT            »
   clock_2:      H                  2              »
                                                                     »
   clock_3:      H                  3              »
                                                    »
  vector_0:   H      0      0      0      0                 »
                        U21    U22    U23    U24                    »
  vector_1:   H      1      1      1      1                 »
                                                            »
classic: 3/                                   »
                                                                           »
«                                                        »
«  ancilla:   Ry( /3)   Ry(0.50536)   Ry(0.25066)  M           »
«                                                     »
```

```
«  clock_0:                    0           »
«                                                                        »
«  clock_1:                    1           »
«                                                     QFT                »
«  clock_2:                    2           »
«                                                                        »
«  clock_3:                    3           »
«                                                                 »
« vector_0:                        0          »
«                                                            U24_inv  »
« vector_1:                        1          »
«                                                                    »
«classic: 3/                             »
«                                                 0                     »
«
«  ancilla:                    M
«
«  clock_0:                 H
«
«  clock_1:                 H
«
«  clock_2:                 H
«
«  clock_3:                 H
«
« vector_0:  0          0          0                 M
«              U23_inv    U22_inv    U21_inv
« vector_1:  1          1          1                  M
«
«classic: 3/
«                                                      0  1  2
```

[575]: `qc2.draw()`

[575]:

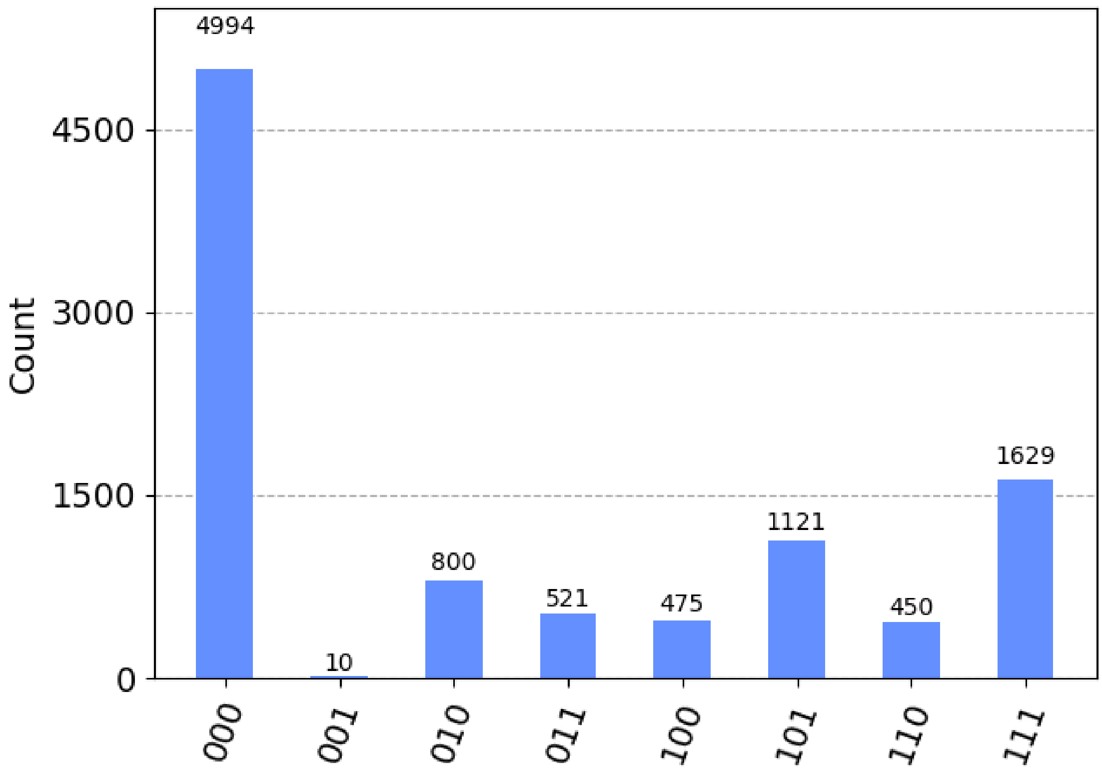
```
                                                                      »
 ancilla:                                     Ry( )  »
                                                                »
 clock_0:       H                   0           »
                                                                      »
 clock_1:       H                   1           »
                                                   IQFT               »
 clock_2:       H                   2           »
                                                                     »
 clock_3:       H                   3           »
                                                      »
vector_0:   H      0      0      0      0                 »
                      U21    U22    U23    U24                       »
```

```
 vector_1:   H       1       1       1       1                 »
                                                               »
classic: 3/                                       »
                                                                                  »
«                                                            »
«  ancilla:   Ry( /3)    Ry(0.50536)    Ry(0.25066)   M             »
«                                                        »
«  clock_0:                          0             »
«                                                                                  »
«  clock_1:                          1             »
«                                                            QFT                »
«  clock_2:                          2             »
«                                                                                  »
«  clock_3:                          3             »
«                                                                          »
« vector_0:                              0          »
«                                                                   U24_inv  »
« vector_1:                              1          »
«                                                                          »
«classic: 3/                                      »
«                                                     0                          »
«
«  ancilla:                          M
«
«  clock_0:                      H
«
«  clock_1:                      H
«
«  clock_2:                      H
«
«  clock_3:                      H
«
« vector_0:  0           0           0                  M
«              U23_inv     U22_inv     U21_inv
« vector_1:  1           1           1                   M
«
«classic: 3/
«                                                             0   1   2
```

[576]:

```
result = execute(qc2, state_vector_sim, shots=10000).result()
counts = result.get_counts(qc2)
plot_histogram(counts)
```

[576]:



The proper values are 001, 011, 101, 111.

[578]:
```
keys = ["001", "011", "101", "111"]
values = np.array([counts[key] for key in keys])
```

[579]:
```
display(np.sqrt(values/values.min())/32)
```

```
array([0.03125    , 0.22556353, 0.33086652, 0.39885089])
```

The above result from HHL measurement only show the amplitude of the solution vector, the true value is $[-1, 7, 11, 13]/32$, the sign of the each position elements requires additional process. □