Problem 2: HHL Algorithm

Problem & Analysis

Solving a linear equation `A * x = b` in a quantum computaional manner, where `A` is an square matrix and `b` is an vector with matching dimension. For a simple example:

```
Solving Ax = b where A is
  [   1,     1]
  [1/√2, -1/√2]
and b is
  [ 1/ 2]
  [-1/√2]

The solution is unique:
  [-1/4]
  [ 3/4]
```

From linear algebra basics, we know that the solution `x` might be one from nothing, unique, or an ensemble, depending on the rank of the expanded matrix `[A|b]`. In **any of these cases** noted, we can apply the Gaussian Elimination method or Matrix Inversion to get the accurate answer, but naive implementations cost time of `O(n^2)`, while the most optimized classical algorithm reduces it to `O(N*sqrt(k))`, where `k` describes the cost for inversing the matrix. However, in **case of unique solution**, we have another perspective view and thoughts upon form of the linear equation, where is also probably the HHL Alogrithm derives. 😀

In quantum computation traditions, `U` denotes a quantum gate and `|phi>` denotes a quantum state, when `U` is applied to `|phi>`, it turns to be a new state, say `U |phi> -> |psi>`. Corresponding to the physical process that, a quantum system **evolves** its state from `|phi>` to `|psi>` under the environment influence `U`. Compared with the linear equation `A * x = b`, it's easy to figure out the logical structure in parallel:

```
quantum evolution:  U |phi> -> |psi>
  linear equation:  A   x   =    b

where U and A are square matrices, others are all vectors in computational representation
```

ℹ️ From this view, solving a linear equation is like to **finding the initilal state of a quantum evolution process** where the final state is known to be `|psi>` while the evolution operator is `U`. Due to the **reversible computing** nature of quantum computating, this is not a magic in philosophical sense. 😉

Solution

Now it's clear enough to put an elephant into the refrigerator:

- Encode classical `A` and `b` to quantum `U` and `|psi>`
  - ⚠ tricky: find ways to satisfy the required `unitary & unit vector` condition for `A` and `b`
- Find `|phi>` through matrix inversion of `U`, aka. computing the reversed `|phi> = U†|psi>`
  - ⚠ tricky: do not inverse `U` directly, but instead decompose `|psi>` to `U`'s eigen vectors, then simply inverse the eigen values 🥴
- Decode `|phi>` to get the solution `x` (in a probablistic approximated sense)

Assuming the toy case `dim(A) = rand(A) = 2`, we implement & explain the HHL alogrithm in both mathematic and programmatic view in following sections.

HHL in a mathematic view

Firstly, here's the mathematic formula tour sketch:

```
Our goal is to approximate this state |x>:
A|x> = |b>                    // assume A is an invertibe hermitian (no need be unitary), and |b> is a unit vecto
 |x> = A^(-1) |b>
     = Σi 1/λi |vi><vi| |b>              // eigen decomposition of A^(-1) where λi and |vi> are the i-th eigen pa
     = (Σi 1/λi |vi><vi|) (Σj βj |vj>)   // decompose |b> to A's eigen basis |vi>; i,j ∈ [0, N-1], N = dim(A)
     = ΣiΣj βj/λi |vi><vi|vj>
     = ΣiΣj βj/λi |vi>                   // <vi|vj> == 1 if i==j else 0
     = Σi (Σj βj) 1/λi |vi>              // aggregate inner const coeffs
     = Σi βi/λi |vi>                     // rename, βi only depends on i

Firstly, consider estimating eigvals of A over vector |b>, using two ancilla qubits inited as |00>:
  = QPE_2(A, |00b>)
  = C_02(U^2^1) * C_12(U^2^0) * (H @ H @ I) |00b>     // |q0,q1,q2>, C_ij(U) denotes controlled-unitary U on |qi
  = C_02(U^2) * C_12(U) * |++b>                       // |+> = H|0>, θk for q[k-1]
  = C_02(U^2) * C_12(U) * |+>(|0>+|1>)|b>             // ignore global phase
  = C_02(U^2) * |+>(|0b>+|1>U|b>)
  = C_02(U^2) * (|0>+|1>)(|0b>+|1>U|b>)
  = C_02(U^2) * (|00b> + |01>U|b> + |10b> + |11>U|b>)
  = |00b> + |01>U|b> + |10>U^2|b> + |11>U^3|b>
  = (Σk |k> U^k) |b>                                  // <= how to choose U, then reduce U^k|b> ??

let U = exp(iAt) = Σj exp(i*λj*t*k) |vj><vj| where t is a tunable parameter (eg. 2*pi), now U is unitary since A
  = (Σk |k> exp(iAt)^k) |b>
  = (Σk |k> exp(iAtk)) |b>
  = (Σk |k> exp(iAtk) |vj><vj|) |b>                   // again decompse |b>, U has the same eigvecs with A
  = (Σk |k> exp(iAtk) |vj><vj|) (Σi βi |vi>)          // <vi|vj> == 1 if i==j else 0
  = Σk |k> (Σj βj exp(iAtk) |vj>)
  = Σj βj (Σk |k> exp(iAtk)) |vj>                     // law of commutativity
  = Σj βj (Σk |k> (Σs exp(i*λs*t*k) |vs><vs|)) |vj>
  = Σj βj (Σk |k> exp(i*λj*t*k)) |vj>                 // <vs|vj> == 1 if s==j else 0
  = Σj βj (Σk exp(i*λj*t*k)|k>) |vj>                  // law of commutativity for a scalar
  ≈ Σj βj |λj'> |vj>                                  // FIXME: jump of faith following the standard QPE??
where λj' is the n-qubit binary approximation to 2^n*(λj*t/2*pi), while λj is the real eigvals, i.e.:
  λj' = 2^n * (λj*t) / (2*pi)
see in order to let λj' to approximate the true λj, we will need:
  1 = 2^n * t / (2*pi)
  t = 2*pi / 2^n
when n = 2 in our case, the t (or in hamilton-similation namely t0) should be:
  t = 2*pi / 2^2
    = pi/2

Now, compare what we've got:
  |z> = Σi βi |λi'> |vi>
and what we'd want:
  |x> = Σi βi/λi |vi>
it would be nicely approximated, if we can:
  - move the index value λj' out from the auxiliary register |λj'>
  - turn it to become the proper coefficient 1/λi
Following the thesis, adding an extra ancilla qubit |q0> inited with |1>,
then this |λi> needs an RY rotation of angel θj = -2*arcsin(C/λj), controlled by the ancilla qubit
```

```
where C is a nomalizer constant holding that C <= min(λj):
  = CR(|0,z>, C)
  = C_02(RY(θ1)) * C_01(RY(θ2)) * (X @ I @ I) |0,z>     // |q0,q1,q2,q3>, C_ij(U) denotes controlled-unitary U or
  = C_02(RY(θ1)) * C_01(RY(θ2)) |1> (Σi βi |λi'> |vi>)
  = Σj (sqrt(1-(C/λj)**2)|0> + C/λj|1>) βi |λi'> |vj>
hence when the ancilla bit is measure to be |1>, the |q3> would be in state:
  = Σj (C/λj) βi |vj>
  = Σj C*βi/λj |vj>
when we let C = 1, the state matches what we exactly want :)

For conveniently gathering results, iQPE can be performed to sort the outcoming vector:
  = (I @ iQPE_2(A, |00b>)) * CR(|0,z>, C)
  = (I @ iQPE_2(A, |00b>)) * Σj (sqrt(1-(C/λj)**2)|0> + C/λj|1>) βi |λi'> |vj>
  = Σj (sqrt(1-(C/λj)**2)|0> + C/λj|1>) |00> |vj>
which restores the QPE register |q1q2> to |00>

Finally, amplitude value at 0 and 2**3 are the answer.
We are done. 😆
```

HHL in a programmatic view

And here's the pseudo-code sketch for the procedure framework:

```
def HHL(A:Matrix, b:Vector) -> Vector:
  # Step 1: classical preprocess
  (Ah, b_n), stats = transform(A, b)    # transform to `Ah * y = b_n`

  # Step 2: quantum computing
  cq = HHL_circuit(Ah, b_n)             # build qcircuit
  qstate = qvm.run(cq)                  # run and get the final state vector
  y = project_q3(qstate)                # only need amplitude of |q3>

  # Step 3: classical postprocess
  x = transform_inv(y, *stats)          # transform back to get `x`

  return x
```

Components are explained in following sections.

⚪ Transform the equation

To make the quantum evolution framework work, it requires `A` to be **hermitian** while `b` to be a **normalized vector**. However, `A` and `b` are **arbitarily** given from a linear equation. Follow this to transform an arbitary linear equation to suit valid quantum gate and state:

```
              Ax = b
          (A'A) x = A'*b            # multiply by A' (A.dagger) at both sides, assert A'A is her
(A'A) (x / |A'*b|) = A'*b / |A'*b|  # devide by |A'*b| at both sides, assert right side is unit
          A_h * y = b_n             # rename, now this new equation satisfies all needed conditi
```

It is easy to inverse the transformation back later, in order to get the final answer:

```
x / |A'*b| = y
         x = y * |A'*b|             # the final answer
```

Take the concrete question as example:

```
    Ax = b
=>
    [   1,     1]   = [ 1/ 2]
    [1/√2, -1/√2] x = [-1/√2]
=>
    [1,  1/√2][   1,     1]   = [1,  1/√2][ 1/ 2]
    [1, -1/√2][1/√2, -1/√2] x = [1, -1/√2][-1/√2]
=>
    [3/2, 1/2]   = [0]
    [1/2, 3/2] x = [1]
=>
    A12 x = |1>
```

It turns out to be the equation that many essays have long played with: `A12 x = |1>` where `A12` is a hermitian matrix with eigvals `1` and `2`. 😀

◯ Encode qunatum data

Find the circuits to encode the classical data to quantum facts `U = exp(iAθ)` and `|b> = b0|0> + b1|1>`:

```python
def encode_A(A:Matrix, θ:float) -> QGate:
  ''' encode a hermitian A to unitary exp(iAθ) '''
  assert is_hermitian(A)
  u = scipy.linalg.expm(1j*A*θ)        # matrix expotional, turing to be a unitry for hermitian
  assert is_unitary(u)
  return QOracle(u)                    # make an Oracle gate, or `matrix_decompose()` to U4 gates

def encode_b(b:Vector) -> QCircuit:
  ''' encode a unit vector b onto amplitude of |b> '''
  assert is_unit(b)
  θ = 2 * np.arccos(b[0])
  cq = QCircuit() << RY(θ)             # rotate with RY gate
  if b[1] < 0: cq << Z()               # fix the sign of |1> part
  return cq
```

ℹ Parameters for these gates are manually calculated out accroding to matrix form definition, this is unnatural but on its way...
⚠ The function `encode_A()` shown here is a conceptual demo, we need `U^2^k` rather than simple `U` in QPE, see real code for the details. 🤔

◯ Main circuit routine

Construct the main circuit routine as HHL reqires: QPE, controlled RY rotation and iQPE.

```python
def HHL_circuit(A:Matrix, b:Vector, t0=2*pi, r=4) -> QCircuit:
  # q0: perform RY to inverse λi
  # q1~q2: QPE scatter A => Σ|λi>, λi is integer approx to eigvals of A
  # q3: amplitude encode b => |b>
  q0, q1, q2, q3 = list(qvm.qAlloc_many(4))    # work around of tuple unpack

  # Step 1: prepare |b>, very toyish :(
  enc_b = encode_b(b, q3)

  # Step 2: QPE of A over |b>
  # following the approx formula: λj' = 2^n * (λj*t) / (2*pi)
  # in order to let λj' = λj, we have t = 2*pi / 2**n_qft = pi/2
  t = 2*pi / 2**2
  qpe = QCircuit() \
      << H([q1, q2]) \
      << encode_A(A, q3, t, 0, QOracle).control(q2) \
      << encode_A(A, q3, t, 1, QOracle).control(q1) \
      << H(q1) \
      << CR(q1, q2, pi/2).dagger() \
      << H(q2)

  # Step 3: controled-Y rotate θj = 2*arcsin(C/λi) ≈ 2*C/λi; 2^(1-r)*pi = 2*C
  # NOTE: the rotation angles are accordingly esitimated by eigvals of A
  # note that RY(θ)|1> = [    // the more θ it rotates, the more away from |1> and shifts to |0>
  #   -sin(θ/2),
  #    cos(θ/2),
  # ]
  eigenvals, eigenvecs = npl.eig(A)
  lbd1, lbd2 = sorted(eigenvals)
  C = lbd1        # C is a normalizer being C <= min(λi)
  # we starts ancilla qubit from |1>, so add a sign to rotation angle θj
  cr = QCircuit() \
      << X(q0) \
      << RY(q0, -2*np.arcsin(C/lbd1)).control(q1) \
      << RY(q0, -2*np.arcsin(C/lbd2)).control(q2)

  return enc_b << qpe << cr << qpe.dagger()
```

HHL in a programmatic view (further optimizations and tricks)

However, as an application, this 4-qubits circuit we've just proposed indeed suffers from heavy **precision problems**, due to its very limited resource -- only two qubits -- in the `QPE` process. Raising the consequences:

- It cannot handle input matrix with eigvals that hard to approximate within 2-qubits
- It heavily relies on the `QOracle` gate, when replaced by `matrix_decompose()`, precision drops dramatically

To make somebody happy, we further introduce two cheaty technics responding to the issues above respectively.

- Apply the `reduce` preprocessing rather than that in former section, this will reduces any other arbitary matrix A to our well-known `A12` toy in all essays, assuring no eigval estimation loss during 2-qubit QPE process.
- Since we've fixed the input `A` to be a certain `A12`, we manually do the matrix decomposition to break `QOracle` gate down to `U4` and `X` gates

See these in source code for implementation details~

Source Code

⚠ The whole source code is extremely long (~900 lines), as we've did a lot of fruitless experiments and explorations. Here we only paste the finally working parts. 😂

Tested under `pyqpanda 3.7.12 + Python 3.8.15`

```python
#!/usr/bin/env python3
# Author: Armit
# Create Time: 2023/04/03

from typing import Tuple, List, Callable, Union

from pyqpanda import *
import numpy as np
np.set_printoptions(suppress=True, precision=7)
import numpy.linalg as npl
import scipy.linalg as spl

Matrix = np.ndarray
Vector = np.ndarray

# case of the special: `A12` is the original essay example, `b1` is the test case from Qiskit cc
# NOTE: Aij is required to be hermitian (not needed to be unitary though)
r2 = np.sqrt(2)
pi = np.pi
A12 = np.asarray([      # eigval: 2, 1
  [3, 1],               # eigvec: [0.70710678, 0.70710678], [-0.70710678, 0.70710678]
  [1, 3],
]) / 2
b1 = np.asarray([0, 1]) # |b> = |1>

# case of the target `question1()`
# svd(A) = P * D *Q
#   [-1 0][1    0][-1 -1]
#   [ 0 1][0 1/r2][ 1 -1]
A = np.asarray([     # eigval: 1.34463698, -1.05174376
  [   1,     1],     # eigvec: [0.9454285, 0.32582963], [-0.43812257, 0.89891524]
  [1/r2, -1/r2],
])
b = np.asarray([
    1/2,
  -1/r2,
])


def project_q3(qstate:List[complex], is_first:bool=False) -> List[complex]:
  ''' gather all cases of ancilla qubit to be |1>, then pick out the target |q> only '''

  if is_first:
    # the first |q0> is the target qubit
    return qstate[len(qstate)//2:][:2]
  else:
    # the last |q3> is the target qubit
    return [qstate[0], qstate[len(qstate)//2]]
```

```python
def transform(A:Matrix, b:Vector, meth:str='hermitian', A_r:Matrix=A12) -> Tuple[Tuple[Matrix, V
  ''' Transforming an arbitray Ax = b to equivalent forms '''

  assert meth in ['reduce', 'hermitian']

  if meth == 'reduce':
    '''
      Reduce arbitray Ax = b to Ar * y = b_n where Ar is hand-crafted well-known and b_n is unit
                  A   * x = b
            (Ar * D) * x = b           # left-decompose A by a known matrix Ar, this is NOT effec
      Ar * (D / |b| * x) = b / |b|     # normalize right side b, turining it a unit vector
                  Ar * y = b_n         # rename, suits the form required by HHL_2x2 circuit
    '''
    D = npl.inv(A_r) @ A
    b_norm = npl.norm(b)
    b_n = b / b_norm

    # the validated A and b for new equation, and stats needed for inversion
    return (A_r, b_n), (D, b_norm)

  if meth == 'hermitian':
    '''
      Multiply a certain B on both side to make A a hermitian, then :
                         A   * x = b
                  (A' * A) * x = A' * b                # (if necessary) multiply by A.dagger a
      (A' * A) * (x / |A' * b|) = (A' * b) / |A' * b|    # normalize right side A' * b, turining
                         Ah * y = b_n                  # rename, suits the form required by HH
    '''

    if not np.allclose(A, A.conj().T):    # if A is not hermitian, make it hermitian
      B = A.conj().T                      # A' * A is promised to be hermitian in math
      A_h = B @ A
      b   = B @ b
      assert np.allclose(A_h, A_h.conj().T)
    else:
      A_h = A

    b_norm = npl.norm(b)
    b_n = b / b_norm

    # the validated A and b for new equation, and stats needed for inversion
    return (A_h, b_n), (None, b_norm)

def transform_inv(y:Vector, D:Matrix, b_norm:float, meth:str='hermitian') -> Vector:
  ''' Solve x from transformed answer y '''

  assert meth in ['reduce', 'hermitian']

  if meth == 'reduce':
    '''
      Solve x from y = D / |b| * x, this is NOT effecient in any sense though:
```

```python
        D / |b| * x = y
                  x = (D / |b|)^(-1) * y
                  x = D^(-1) * |b| * y
    '''

    return npl.inv(D) @ (b_norm * y)

  if meth == 'hermitian':
    '''
      Solve x from y = x / |A' * b|:
        x / |A' * b| = y
                   x = |A' * b| * y
    '''

    return y if abs(1.0 - b_norm) < 1e-5 else b_norm * y


def encode_b(b:Vector, q:Qubit) -> QCircuit:
  '''
    Amplitude encode a unit vector b = [b0, b1] to |b> = b0|0> + b1|1> use RY+Z gate,
    to fix the lacking the coeff `e^(i*phi)` compared with U3 gate when b1 < 0
                         RY(theta) |0> = |b>
      [cos(theta/2), -sin(theta/2)][1] = [b0]
      [sin(theta/2),  cos(theta/2)][0] = [b1]
  '''
  if isinstance(b, list): b = np.asarray(b)
  assert isinstance(b, Vector), 'should be a Vector/np.ndarray'
  assert tuple(b.shape) == (2,), 'should be a vector lengthed 2'
  assert (npl.norm(b) - 1.0) < 1e-5, 'should be a unit vector'

  theta = 2 * np.arccos(b[0])
  cq = QCircuit() << RY(q, theta)
  if b[1] < 0: cq << Z(q)             # fix sign of |1> part

  return cq

def encode_A(A:Matrix, q:Qubit, theta:float=2*pi, k:float=0, encoder=QOracle) -> Union[QGate, QC
  ''' Unitary generator encode hermitian A to a unitary up to power of 2: U^2^k = exp(iAθ)^2^k '
  assert isinstance(A, Matrix), 'should be a Matrix/np.ndarray'
  assert tuple(A.shape) == (2, 2), 'should be a 2x2 matrix'
  assert np.allclose(A, A.conj().T), 'should be a hermitian matrix'
  assert encoder in [QOracle, matrix_decompose]

  u = spl.expm(1j* A * theta).astype(np.complex128)   # FIXME: explain why this is a valid unita
  assert np.allclose(u @ u.conj().T, np.eye(len(u)))

  return encoder(q, npl.matrix_power(u, 2**k).flatten())


def HHL_2x2_qpanda_5(A:Matrix=A12, b:Vector=b1, enc:str='oracle') -> QCircuit:
  ''' The guessed implementations from QPanda HHLAlg print(circuit) using 5-qubits '''
```

```
  # q0: |b>
  # q1~3: QFT (leave 1 for sign?? do not know why...)
  # q4: ancilla
  qv = qvm.qAlloc_many(5)
  q0, q1, q2, q3, q4 = [qv[i] for i in range(len(qv))]

  # HHLAlg uses RY to encode
  enc_b = QCircuit() << RY(q0, 2*np.arccos(b[0]))

  # FIXME: when QOracle be replaced with `matrix_decompose()`, precision drops dramatically
  t = 2*pi / (1<<3)     # theta = 2*pi / (1<<n_qft)
  encoder = QOracle if enc == 'oracle' else matrix_decompose
  qpe = QCircuit() \
      << H([q1, q2, q3]) \
      << encode_A(A, q0, t, 0, encoder).control(q3) \
      << encode_A(A, q0, t, 1, encoder).control(q2) \
      << encode_A(A, q0, t, 2, encoder).control(q1) \
      << H(q1) \
      << CR(q1, q2, pi/2).dagger() \
      << CR(q1, q3, pi/4).dagger() \
      << H(q2) \
      << CR(q2, q3, pi/2).dagger() \
      << H(q3)

  cr = QCircuit() \
      << X([q2, q3]) \
      << RY(q4, pi).control([q1, q2, q3]) \
      << X([q1, q2]) \
      << RY(q4, pi/3).control([q1, q2, q3]) \
      << X([q3]) \
      << RY(q4, -pi/3).control([q1, q2, q3]) \
      << X([q1]) \
      << RY(q4, -pi).control([q1, q2, q3])

  return enc_b << qpe << cr << qpe.dagger()

def HHL_2x2_ours(A:Matrix=A12, b:Vector=b1, enc:str='oracle', rot:str='eigval') -> QCircuit:
  '''
    My modified version, looking for best param according to Qiskit:
      - https://qiskit.org/textbook/ch-applications/hhl_tutorial.html
      - https://arxiv.org/pdf/2108.09004.pdf
  '''

  assert np.allclose(A, A.conj().T), 'A should be a hermitian'
  assert (npl.norm(b) - 1.0) < 1e-5, 'b should be a unit vector'
  assert enc in ['oracle', 'decompose', 'hard-coded']
  assert rot in ['eigval', 'approx']

  # q0: perform RY to inverse λi
  # q1~q2: QPE scatter A => Σ|λi>, λi is integer approx to eigvals of A
```

```
# q3: amplitude encode b => |b>
qv = qvm.qAlloc_many(4)
q0, q1, q2, q3 = [qv[i] for i in range(len(qv))]

# Step 1: prepare |b>, very toyish :(
enc_b = encode_b(b, q3)

# Step 2: QPE of A over |b>
# following the approx formula: λj' = 2^n * (λj*t) / (2*pi)
# in order to let λj' = λj, we have t = 2*pi / 2**n_qft = pi/2
t = 2*pi / 2**2

if enc == 'oracle':
  u2 = encode_A(A, q3, t, 0, QOracle)
  u1 = encode_A(A, q3, t, 1, QOracle)
if enc == 'decompose':
  u2 = encode_A(A, q3, t, 0, matrix_decompose)
  u1 = encode_A(A, q3, t, 1, matrix_decompose)
if enc == 'hard-coded':
  # The actual U(iAθ) for A=A12, θ=pi/2 is:
  #   [-1+i -1-i]
  #   [-1-i -1+i] / 2
  # the U(iAθ)^2^0 is the same as above, it can be decomposed as U4 and X
  # and the U(iAθ)^2^1 is actually X gate, nice ;)
  u2 = QCircuit() << X(q3) << U4(-0.75*pi, -pi/2, pi/2, pi/2, q3)
  u1 = X(q3)

qpe = QCircuit() \
    << H([q1, q2]) \
    << u2.control(q2) \
    << u1.control(q1) \
    << H(q1) \
    << CR(q1, q2, pi/2).dagger() \
    << H(q2)

# Step 3: controled-Y rotate θj = 2*arcsin(C/λi) ≈ 2*C/λi; 2^(1-r)*pi = 2*C
# note that RY(θ)|1> = [    // the more θ it rotates, the more away from |1> and shifts to |0>
#   -sin(θ/2),
#    cos(θ/2),
# ]
# we starts ancilla qubit from |1>, so add a sign to rotation angle θj

if rot == 'eigval':
  # the rotation angles are accordingly esitimated by eigvals of A
  eigenvals, eigenvecs = npl.eig(A)
  lbd1, lbd2 = sorted(eigenvals)
  C = lbd1        # C is a normalizer being C <= min(λi)
  r1 = -2*np.arcsin(C/lbd1)
  r2 = -2*np.arcsin(C/lbd2)
if rot == 'approx':
  # the rotation angles are generally approximated
```

```
    # for big endian |q1q2>, |00>=0.0, |10>=0.25, |01>=0.5, |11>=0.75
    r1 = -pi
    r2 = -pi/3

  cr = QCircuit() \
      << X(q0) \
      << RY(q0, r1).control(q1) \
      << RY(q0, r2).control(q2)

  return enc_b << qpe << cr << qpe.dagger()


def HHL(A:Matrix, b:Vector, HHL_cq:Callable, HHL_cq_args:tuple=tuple(), meth:str='hermitian', pr
  '''
    Solving linear system equation `Ax = b` by quantum simulation in poly(logN) steps
      - https://arxiv.org/abs/0811.3171
      - https://arxiv.org/abs/2108.09004
      - https://arxiv.org/abs/1110.2232
      - https://arxiv.org/abs/1302.1210
      - https://arxiv.org/abs/1805.10549
      - https://arxiv.org/abs/1302.4310
      - https://arxiv.org/abs/1302.1946
      - https://en.wikipedia.org/wiki/Quantum_algorithm_for_linear_systems_of_equations
      - https://zhuanlan.zhihu.com/p/164375189
      - https://zhuanlan.zhihu.com/p/426811646
      - https://www.qtumist.com/post/5212
      - https://pyqpanda-toturial.readthedocs.io/zh/latest/HHL.html
    NOTE:
      - Input `A` and `b` for this function are arbitary, they will be auto-transformed for vali
      - Due to resource limit of 4 qubits and explorations of our ancestors, we do NOT solve arb
        but only solve one special prototype case in quantum manner, then **reduce** any other c
        This special case is artificially designed to allow eigenvals of matrix A is stored with
        i.e. we choose a unitary A with eigen values of [1, 2], [1, 3] or [2, 3], so that eigenv
  '''
  assert isinstance(A, Matrix) and tuple(A.shape) == (2, 2)
  assert isinstance(b, Vector) and tuple(b.shape) == (2,)

  # Step 1: transform in a classical manner
  (Ar, b_n), stats = transform(A, b, meth=meth)

  # Step 2: solving Ar * y = b_n in a quantum manner
  global qvm
  qvm = CPUQVM()
  qvm.init_qvm()

  hhl_cq = HHL_cq(Ar, b_n, *HHL_cq_args)

  prog = QProg() << hhl_cq
  qvm.directly_run(prog)
  y = project_q3(qvm.get_qstate(), is_first=HHL_cq is HHL_2x2_qpanda_5)
```

```
  ircode = ''
  if prt_ircode:
    try: ircode = to_originir(prog, qvm)
    except: pass
  #qvm.qFree_all(qvm.get_allocate_qubits())   # FIXME: buggy, stucks to SEGV
  qvm.finalize()

  # Step 3: inv-transform in a classical manner
  y = np.asarray(y, dtype=np.complex128)
  x = transform_inv(y, *stats, meth=meth)
  x = x.real.tolist()

  if prt_ircode:
    return x, ircode
  else:
    return x


def question1() -> Tuple[list, str]:
  return HHL(A, b, HHL_2x2_ours, ('hard-coded', 'approx'), meth='hermitian', prt_ircode=True)


def benchmark(kind:str, eps=1e-2):
  circuits = [
    'ours',
    'qpanda_5',
  ]

  v_errors = [0.0] * len(circuits)    # error of value L2
  n_errors = [0.0] * len(circuits)    # error up to a normalization

  for _ in range(1000):
    if   kind == 'random':
      A_ = np.random.uniform(size=[2, 2], low=-1.0, high=1.0)
      b_ = np.random.uniform(size=[2],    low=-1.0, high=1.0)
    elif kind == 'target':     # around target case
      A_ = A + np.random.uniform(size=[2, 2], low=-1.0, high=1.0) * eps
      b_ = b + np.random.uniform(size=[2],    low=-1.0, high=1.0) * eps
    else: raise ValueError

    try:
      z = npl.solve(A_, b_)
      z_n = z / npl.norm(z)
    except npl.LinAlgError:
      continue

    for i, name in enumerate(circuits):
      x = HHL(A_, b_, globals()['HHL_2x2_' + name], meth='hermitian')
      v_errors[i] += npl.norm(np.abs(z - np.asarray(x)))
      x_n = x / npl.norm(x)
      n_errors[i] += npl.norm(np.abs(z_n - x_n))
```

```
  for i, name in enumerate(circuits):
    print(f'  {name}: {v_errors[i]} / {n_errors[i]}')

def run_compares(A:Matrix, b:Vector, title='case'):
  print(f'{title}')
  print('  truth:          ', npl.solve(A, b))
  print('  ours (r+hc+a): ', HHL(A, b, HHL_2x2_ours, ('hard-coded', 'approx'), meth='reduce'))
  print('  ours (r+hc+e): ', HHL(A, b, HHL_2x2_ours, ('hard-coded', 'eigval'), meth='reduce'))
  print('  ours (r+d+a):  ', HHL(A, b, HHL_2x2_ours, ('decompose', 'approx'),  meth='reduce'))
  print('  ours (r+d+e):  ', HHL(A, b, HHL_2x2_ours, ('decompose', 'eigval'),  meth='reduce'))
  print('  ours (r+o+a):  ', HHL(A, b, HHL_2x2_ours, ('oracle', 'approx'),     meth='reduce'))
  print('  ours (r+o+e):  ', HHL(A, b, HHL_2x2_ours, ('oracle', 'eigval'),     meth='reduce'))
  print('  ours (h+hc+a): ', HHL(A, b, HHL_2x2_ours, ('hard-coded', 'approx'), meth='hermitian')
  print('  ours (h+hc+e): ', HHL(A, b, HHL_2x2_ours, ('hard-coded', 'eigval'), meth='hermitian')
  print('  ours (h+d+a):  ', HHL(A, b, HHL_2x2_ours, ('decompose', 'approx'),  meth='hermitian')
  print('  ours (h+d+e):  ', HHL(A, b, HHL_2x2_ours, ('decompose', 'eigval'),  meth='hermitian')
  print('  ours (h+o+a):  ', HHL(A, b, HHL_2x2_ours, ('oracle', 'approx'),     meth='hermitian')
  print('  ours (h+o+e):  ', HHL(A, b, HHL_2x2_ours, ('oracle', 'eigval'),     meth='hermitian')
  print('  qpanda_5 (h+d):', HHL(A, b, HHL_2x2_qpanda_5, ('decompose',), meth='hermitian'))
  print('  qpanda_5 (h+o):', HHL(A, b, HHL_2x2_qpanda_5, ('oracle',),    meth='hermitian'))
  print('  qpanda_alg:    ', np.asarray(HHL_solve_linear_equations((A.conj().T @ A).astype(np.cc
  print()


if __name__ == '__main__':
  # solve the specified target linear equation
  run_compares(A, b, '[target question]')

  # benchmark error of different circuits
  print('[benchmark random] L1 error / L1 error after norm:')
  benchmark(kind='random')
  print('[benchmark target] L1 error / L1 error after norm:')
  benchmark(kind='target', eps=1e-2)
  print()

  # solve random linear equations
  A = np.random.uniform(size=[2, 2], low=-1.0, high=1.0)
  b = np.random.uniform(size=[2],    low=-1.0, high=1.0)
  run_compares(A, b, '[random question]')
  print()

  # test the question API
  _, ircode = question1()
  print(ircode)
```

Run demo:

```
C:¥Windows¥system32¥cmd.exe

D:\Desktop\Workspace\quantum-computation-playground\contest\第二
-> py P2_partial.py
[target question]
  truth:          [-0.25  0.75]
  ours (r+hc+a):  [-0.2499999999999999, 0.7500000000000003]
  ours (r+hc+e):  [-0.2499999999999999, 0.7500000000000003]
  ours (r+d+a):   [-0.2520000636763246, 0.5407413518849395]
  ours (r+d+e):   [-0.2520000636763246, 0.5407413518849395]
  ours (r+o+a):   [-0.2500000000000001, 0.7500000000000004]
  ours (r+o+e):   [-0.2500000000000001, 0.7500000000000004]
  ours (h+hc+a):  [-0.2500000000000003, 0.7500000000000003]
  ours (h+hc+e):  [-0.2500000000000003, 0.7500000000000003]
  ours (h+d+a):   [-0.02368358637268825, 0.6798946985944283]
  ours (h+d+e):   [-0.02368358637268825, 0.6798946985944283]
  ours (h+o+a):   [-0.25000000000000006, 0.7500000000000004]
  ours (h+o+e):   [-0.25000000000000006, 0.7500000000000004]
  qpanda_5 (h+d): [-0.14295193903749967, -5.151865880056138e-17]
  qpanda_5 (h+o): [-0.25000000000000044, 0.7500000000000008]
  qpanda_alg:     [-0.25  0.75]

[benchmark random] L1 error / L1 error after norm:
  ours: 5745.543476354065 / 925.6926510649588
  qpanda_5: 5818.389604480754 / 1130.4593937177713
[benchmark target] L1 error / L1 error after norm:
  ours: 7.635097241276007 / 0.26757021666853165
  qpanda_5: 7.610073790804623 / 4.67870170333206

[random question]
  truth:          [-0.8085786 -1.5551322]
  ours (r+hc+a):  [-0.8085786174566512, -1.5551321559982882]
  ours (r+hc+e):  [-0.8085786174566512, -1.5551321559982885]
  ours (r+d+a):   [-0.43063826232304275, -1.3670488119830915]
  ours (r+d+e):   [-0.43063826232304286, -1.3670488119830912]
  ours (r+o+a):   [-0.8085786174566512, -1.5551321559982885]
  ours (r+o+e):   [-0.8085786174566512, -1.5551321559982885]
  ours (h+hc+a):  [-0.08395896409687646, -0.2515485929002174]
  ours (h+hc+e):  [-0.05601230589576848, -0.2236019346991094]
  ours (h+d+a):   [-0.023584103296993356, -0.03892546194470571]
  ours (h+d+e):   [-0.02372500974867356, -0.03927605327352016]
  ours (h+o+a):   [-0.07519206922026636, -0.06741264250697718]
  ours (h+o+e):   [-0.07484789586366422, -0.06704342162274564]
  qpanda_5 (h+d): [-0.014263439466356814, -0.002288148948400603]
  qpanda_5 (h+o): [-0.2372447294421276, 0.13505744146328122]
  qpanda_alg:     [-0.8567746 -1.6730434]
```

We even compare our method `ours (configs)` to the guessed implementation of QPanda with 5-qubits `qpanda_5` and the direct QAlg API `HHL_solve_linear_equations` (namely `qpanda_alg` here). As for the configs for our method, `r/h` indicates the transform method `reduce` or `hermitian`, `hc/d/o` indicates the the implementation for QPE unitary is `hard-coded`, `matrix_decompose` or `QOracle`, the final `a/e` indicates angles in the rotaion step is generally `approximated` or use exactly `eigval`.

The summarized conclusions are:

- QPanda's `matrix_decompose()` seems not reliable
- Our `hermitian` method behaves similiar like `qpanda_5`
- Our `reduce` method behaves similiar like `qpanda_alg`, but remeber, `qpanda_alg` requires much more qubits, while `ours (r)` need to inverse auxiliary matrix in classical process, which is cheaty 😦

references

- https://arxiv.org/abs/0811.3171 **(root thesis)**
- https://arxiv.org/abs/1110.2232
- https://arxiv.org/abs/1302.1210
- https://arxiv.org/abs/1805.10549
- https://arxiv.org/abs/1302.4310
- https://arxiv.org/abs/1302.1946
- https://en.wikipedia.org/wiki/Quantum_algorithm_for_linear_systems_of_equations
- https://qiskit.org/textbook/ch-applications/hhl_tutorial.html **(highly recommended)**
- https://zhuanlan.zhihu.com/p/164375189
- https://zhuanlan.zhihu.com/p/426811646
- https://www.qtumist.com/post/5212
- https://pyqpanda-toturial.readthedocs.io/zh/latest/HHL.html
- https://en.wikipedia.org/wiki/Matrix_exponential
- https://en.wikipedia.org/wiki/Sylvester's_formula#special_case

2023/04/03