Solving linear systems of equations using HHL and its Qiskit implementation

In this tutorial, we introduce the HHL algorithm, derive the circuit, and implement it using Qiskit. We show how to run the HHL on a simulator and on a five qubit device.

Contents

1. Introduction

We see systems of linear equations in many real-life applications across a wide range of areas. Examples include the solution of Partial Differential Equations, the calibration of financial models, fluid simulation or numerical field calculation. The problem can be defined as, given a matrix $A\in\ \mathbb{C}^{N\times N}$ and a vector $\vec{b} \in \mathbb{C}^N$, find $\vec{x} \in \mathbb{C}^N$ satisfying $A\vec{x} = \vec{b}$

For example, take $N = 2$,

$$A = \begin{pmatrix} 1 & -1/3 \\ -1/3 & 1 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{and} \quad \vec{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Then the problem can also be written as find such that `{latex} x_{1}, x_{2}\in\mathbb{C}`

$$\begin{cases} x_1 - \frac{x_2}{3} = 1 \\ -\frac{x_1}{3} + x_2 = 0 \end{cases}$$

A system of linear equations is called s-sparse if A has at most s non-zero entries per row or column. Solving an s-sparse system of size N with a classical computer requires $\mathcal{O}(Ns\kappa\log(1/\epsilon))$ running time using the conjugate gradient method [1]. Here κ denotes the condition number of the system and ϵ the accuracy of the approximation.

The HHL algorithm estimates a function of the solution with running time complexity of $\mathcal{O}(\log(N)s^2\kappa^2/\epsilon)$[2]. The matrix A must be Hermitian, and we assume we have efficient oracles for loading the data, Hamiltonian simulation, and computing a function of the solution. This is an exponential speed up in the size of the system, with the catch that HHL can only approximate functions of the solution vector, while the classical algorithm returns the full solution.

2. The HHL algorithm

A. Some mathematical background

The first step towards solving a system of linear equations with a quantum computer is to encode the problem in the quantum language. By rescaling the system, we can assume $\vec{b}$ and $\vec{x}$ to be normalised and map them to the respective quantum states $|b\rangle$ and $|x\rangle$. Usually the mapping used is such that i^{th} component of $\vec{b}$ (resp. $\vec{x}$) corresponds to the amplitude of the i^{th} basis state of the quantum state $|b\rangle$ (resp. $|x\rangle$). From now on, we will focus on the rescaled problem

$$A|x\rangle = |b\rangle$$

Since A is Hermitian, it has a spectral decomposition

$$A = \sum_{j=0}^{N-1} \lambda_j |u_j\rangle\langle u_j|, \quad \lambda_j \in \mathbb{R}$$

where $|u_j\rangle$ is the j^{th} eigenvector of A with respective eigenvalue λ_j. Then,

$$A^{-1} = \sum_{j=0}^{N-1} \lambda_j^{-1} |u_j\rangle\langle u_j|$$

and the right hand side of the system can be written in the eigenbasis of A as

$$|b\rangle = \sum_{j=0}^{N-1} b_j |u_j\rangle, \quad b_j \in \mathbb{C}$$

It is useful to keep in mind that the goal of the HHL is to exit the algorithm with the readout register in the state

$$|x\rangle = A^{-1}|b\rangle = \sum_{j=0}^{N-1} \lambda_j^{-1} b_j |u_j\rangle$$

Note that here we already have an implicit normalisation constant since we are talking about a quantum state.

B. Description of the HHL algorithm

The algorithm uses three quantum registers, all set to $|0\rangle$ at the beginning of the algorithm. One register, which we will denote with the subindex n_l, is used to store a binary representation of the eigenvalues of A. A second register, denoted by n_b, contains the vector solution, and from now on $N = 2^{n_b}$. There is an extra register for auxiliary qubits, used for intermediate steps in the computation. We can ignore any auxiliary in the following description as they are $|0\rangle$ at the beginning of each computation, and are restored back to $|0\rangle$ at the end of each individual operation.

The following is an outline of the HHL algorithm with a high-level drawing of the corresponding circuit. For simplicity all computations are assumed to be exact in the ensuing description, and a more detailed explanation of the non-exact case is given in Section 2.D..

1. Load the data $|b\rangle \in \mathbb{C}^N$. That is, perform the transformation

$$|0\rangle_{n_b} \mapsto |b\rangle_{n_b}$$

2. Apply Quantum Phase Estimation (QPE) with

$$U = e^{iAt} := \sum_{j=0}^{N-1} e^{i\lambda_j t} |u_j\rangle\langle u_j|$$

The quantum state of the register expressed in the eigenbasis of A is now

$$\sum_{j=0}^{N-1} b_j |\lambda_j\rangle_{n_l} |u_j\rangle_{n_b}$$

where is the `{latex} |\lambda _ {j }\rangle_{n_{l}}` n_l-bit binary representation of λ_j.

3. Add an auxiliary qubit and apply a rotation conditioned on $|\lambda_j\rangle$,

$$\sum_{j=0}^{N-1} b_j |\lambda_j\rangle_{n_l} |u_j\rangle_{n_b} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_j} |1\rangle \right)$$

where C is a normalisation constant, and, as expressed in the current form above, should be less than the smallest eigenvalue λ_{min} in magnitude, i.e., $|C| < \lambda_{min}$.

4. Apply QPE†. Ignoring possible errors from QPE, this results in

$$\sum_{j=0}^{N-1} b_j |0\rangle_{n_l} |u_j\rangle_{n_b} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_j} |1\rangle \right)$$

5. Measure the auxiliary qubit in the computational basis. If the outcome is 1, the register is in the post-measurement state

$$\left(\sqrt{\frac{1}{\sum_{j=0}^{N-1} |b_j|^2 / |\lambda_j|^2}} \right) \sum_{j=0}^{N-1} \frac{b_j}{\lambda_j} |0\rangle_{n_l} |u_j\rangle_{n_b}$$

which up to a normalisation factor corresponds to the solution.

6. Apply an observable M to calculate $F(x) := \langle x|M|x\rangle$.

C. Quantum Phase Estimation (QPE) within HHL

Quantum Phase Estimation is described in more detail in Chapter 3. However, since this quantum procedure is at the core of the HHL algorithm, we recall here the definition. Roughly speaking, it is a quantum algorithm which, given a unitary U with eigenvector $|\psi\rangle_m$ and eigenvalue $e^{2\pi i\theta}$, finds θ. We can formally define this as follows.

Definition: Let $U \in \ \mathbb{C}^{2^m \times 2^m}$ be unitary and let $|\psi\rangle_m \in \mathbb{C}^{2^m}$ be one of its eigenvectors with respective eigenvalue $e^{2\pi i\theta}$. The **Quantum Phase Estimation** algorithm, abbreviated **QPE**, takes as inputs the unitary gate for U and the state and returns the state . Here `{latex} |0\rangle_{n}|\psi\rangle_{m} {latex} |\tilde{\theta}\rangle_{n}|\psi\rangle_{m}` $\tilde{\theta}$ denotes a binary approximation to $2^n\theta$ and the n subscript denotes it has been truncated to n digits.

$$\mathrm{QPE}(U, |0\rangle_n |\psi\rangle_m) = |\tilde{\theta}\rangle_n |\psi\rangle_m$$

For the HHL we will use QPE with $U = e^{iAt}$, where A is the matrix associated to the system we want to solve. In this case,

$$e^{iAt} = \sum_{j=0}^{N-1} e^{i\lambda_j t} |u_j\rangle\langle u_j|$$

Then, for the eigenvector , which has eigenvalue `{latex} |u_{j}\rangle_{n_{b}}` $e^{i\lambda_j t}$, QPE will output . Where `{latex} |\tilde{\lambda }_ { j }\rangle_{n_{l}}|u_{j}\rangle_{n_{b}}` $\tilde{\lambda}_j$ represents an n_l-bit binary approximation to $2^{n_l}\frac{\lambda_j t}{2\pi}$. Therefore, if each λ_j can be exactly represented with n_l bits,

$$\mathrm{QPE}(e^{iAt}, \sum_{j=0}^{N-1} b_j |0\rangle_{n_l} |u_j\rangle_{n_b}) = \sum_{j=0}^{N-1} b_j |\lambda_j\rangle_{n_l} |u_j\rangle_{n_b}$$

D. Non-exact QPE

In reality, the quantum state of the register after applying QPE to the initial state is

$$\sum_{j=0}^{N-1} b_j \left(\sum_{l=0}^{2^{n_l}-1} \alpha_{l|j} |l\rangle_{n_l} \right) |u_j\rangle_{n_b}$$

where

$$\alpha_{l|j} = \frac{1}{2^{n_l}} \sum_{k=0}^{2^{n_l}-1} \left(e^{2\pi i \left(\frac{\lambda_j t}{2\pi} - \frac{l}{2^{n_l}} \right)} \right)^k$$

Denote by $\tilde{\lambda_j}$ the best n_l-bit approximation to λ_j, $1 \leq \ j \leq \ N$. Then we can relabel the n_l-register so that $\alpha_{l|j}$ denotes the amplitude of . So now, `{latex} |l + \tilde { \lambda } _ { j } \rangle_{n_{l}}`

$$\alpha_{l|j} := \frac{1}{2^{n_l}} \sum_{k=0}^{2^{n_l}-1} \left(e^{2\pi i \left(\frac{\lambda_j t}{2\pi} - \frac{l + \tilde{\lambda}_j}{2^{n_l}} \right)} \right)^k$$

If each $\frac{\lambda_j t}{2\pi}$ can be represented exactly with n_l binary bits, then `{latex} \frac { \lambda _ { j } t } { 2 \pi }=\frac { \tilde { \lambda } _ { j } } { 2 ^ { n_{l} } } \ \forall j`. Therefore in this case $\forall j$, $1 \leq \ j \leq \ N$, it holds that $\alpha_{0|j} = 1$ and $\alpha_{l|j} = 0 \forall l \quad \neq 0$. Only in this case we can write that the state of the register after QPE is

$$\sum_{j=0}^{N-1} b_j |\lambda_j\rangle_{n_l} |u_j\rangle_{n_b}$$

Otherwise, $|\alpha_{l|j}|$ is large if and only if and the state of the register is `{latex} \frac { \lambda _ { j } t } { 2 \pi } \approx \frac { l + \tilde { \lambda } _ { j } } { 2 ^ { n_{l} } }`

$$\sum_{j=0}^{N-1}\sum_{l=0}^{2^{n_l}-1}\alpha_{l|j}b_j|l\rangle_{n_l}|u_j\rangle_{n_b}$$

3. Example: 4-qubit HHL

Let's take the small example from the introduction to illustrate the algorithm. That is,

$$A=\begin{pmatrix}1 & -1/3\\ -1/3 & 1\end{pmatrix}\quad\text{and}|b\rangle\quad=\begin{pmatrix}1\\0\end{pmatrix}$$

We will use $n_b=1$ qubit to represent $|b\rangle$, and later the solution $|x\rangle$, $n_l=2$ qubits to store the binary representation of the eigenvalues and 1 auxiliary qubit to store whether the conditioned rotation, hence the algorithm, was successful.

For the purpose of illustrating the algorithm, we will cheat a bit and calculate the eigenvalues of A to be able to choose t to obtain an exact binary representation of the rescaled eigenvalues in the n_l-register. However, keep in mind that for the HHL algorithm implementation one does not need previous knowledge of the eigenvalues. Having said that, a short calculation will give

$$\lambda_1=2/3\text{and}\qquad\lambda_2=4/3$$

Recall from the previous section that the QPE will output an n_l-bit (2-bit in this case) binary approximation to $\frac{\lambda_j t}{2\pi}$. Therefore, if we set

$$t=2\pi\cdot\frac{3}{8}$$

the QPE will give a 2-bit binary approximation to

$$\frac{\lambda_1 t}{2\pi}=1/4\text{and}\qquad\frac{\lambda_2 t}{2\pi}=1/2$$

which is, respectively,

$$|01\rangle_{n_l}\quad\text{and}|10\quad\rangle_{n_l}$$

The eigenvectors are, respectively,

$$|u_1\rangle=\frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}\quad\text{and}|\quad u_2\rangle=\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}$$

Again, keep in mind that one does not need to compute the eigenvectors for the HHL implementation. In fact, a general Hermitian matrix A of dimension N can have up to N different eigenvalues, therefore calculating them would take $\mathcal{O}(N)$ time and the quantum advantage would be lost.

We can then write $|b\rangle$ in the eigenbasis of A as

$$|b\rangle_{n_b}=\sum_{j=1}^{2}\frac{1}{\sqrt{2}}|u_j\rangle_{n_b}$$

Now we are ready to go through the different steps of the HHL algorithm.

1. State preparation in this example is trivial since $|b\rangle = |0\rangle$.
2. Applying QPE will yield

$$\frac{1}{\sqrt{2}}|01\rangle|u_1\rangle + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle$$

3. Conditioned rotation with $C = 1/8$ that is less than the smallest (rescaled) eigenvalue of $\frac{1}{4}$. Note, the constant C here needs to be chosen such that it is less than the smallest (rescaled) eigenvalue of $\frac{1}{4}$ but as large as possible so that when the auxiliary qubit is measured, the probability of it being in the state $|1>$ is large.

$$\frac{1}{\sqrt{2}}|01\rangle|u_1\rangle\left(\sqrt{1-\frac{(1/8)^2}{(1/4)^2}}|0\rangle + \frac{1/8}{1/4}|1\rangle\right) + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle\left(\sqrt{1-\frac{(1/8)^2}{(1/2)^2}}|0\rangle + \frac{1/8}{1/2}|1\rangle\right)$$

$$= \frac{1}{\sqrt{2}}|01\rangle|u_1\rangle\left(\sqrt{1-\frac{1}{4}}|0\rangle + \frac{1}{2}|1\rangle\right) + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle\left(\sqrt{1-\frac{1}{16}}|0\rangle + \frac{1}{4}|1\rangle\right)$$

4. After applying QPE† the quantum computer is in the state

$$\frac{1}{\sqrt{2}}|00\rangle|u_1\rangle\left(\sqrt{1-\frac{1}{4}}|0\rangle + \frac{1}{2}|1\rangle\right) + \frac{1}{\sqrt{2}}|00\rangle|u_2\rangle\left(\sqrt{1-\frac{1}{16}}|0\rangle + \frac{1}{4}|1\rangle\right)$$

5. On outcome 1 when measuring the auxiliary qubit, the state is

$$\frac{\frac{1}{\sqrt{2}}|00\rangle|u_1\rangle\frac{1}{2}|1\rangle + \frac{1}{\sqrt{2}}|00\rangle|u_2\rangle\frac{1}{4}|1\rangle}{\sqrt{5/32}}$$

   A quick calculation shows that

$$\frac{\frac{1}{2\sqrt{2}}|u_1\rangle + \frac{1}{4\sqrt{2}}|u_2\rangle}{\sqrt{5/32}} = \frac{|x\rangle}{||x||}$$

6. Without using extra gates, we can compute the norm of $|x\rangle$: it is the probability of measuring 1 in the auxiliary qubit from the previous step.

$$P(|1\rangle) = \left(\frac{1}{2\sqrt{2}}\right)^2 + \left(\frac{1}{4\sqrt{2}}\right)^2 = \frac{5}{32} = ||x||^2$$

4. Qiskit Implementation

Now that we have analytically solved the problem from the example we are going to use it to illustrate how to run the HHL on a quantum simulator and on the real hardware. The following uses , a Qiskit-based package which can be found in this repository and installed as described in the corresponding file. For the quantum simulator, already provides an implementation of the HHL algorithm requiring only the matrix `quantum_linear_solvers Readme quantum_linear_solvers` A and $|b\rangle$ as inputs in the simplest example. Although we can give the algorithm a general Hermitian matrix and an arbitrary initial state as NumPy arrays, in these cases the quantum algorithm will not achieve an exponential speedup. This is because the default implementation is exact and therefore exponential in the number of qubits. There is no algorithm polynomial

resources in the number of qubits that can prepare an exact arbitrary quantum state, or that can perform the exact operation e^{iAt} for some general Hermitian matrix A. If we know an efficient implementation for a particular problem, the matrix and/or the vector can be given as objects. Alternatively, there's already an efficient implementation for tridiagonal Toeplitz matrices and in the future there might be more. `QuantumCircuit`

However,at the time of writing the existing quantum computers are noisy and can only run small circuits. Therefore, in Section 4.B. we will see an optimised circuit that can be used for a class of problems to which our example belongs and mention the existing procedures to deal with noise in quantum computers.

A. Running HHL on a simulator: general method

To run the code on this page, you'll need to install the linear solvers package. you can do this through the command:

```
pip install git+https://github.com/anedumla/quantum_linear_solvers
```

The interface for all algorithms to solve the linear system problem is . The problem to be solved is only specified when the method is called: `LinearSolver solve()`

```
LinearSolver(...).solve(matrix, vector)
```

The simplest implementation takes the matrix and the vector as NumPy arrays. Below we also create a (the classical algorithm) to validate our solutions. `NumPyLinearSolver`

```
import numpy as np
# pylint: disable=line-too-long
from linear_solvers import NumPyLinearSolver, HHL
matrix = np.array([[1, -1/3], [-1/3, 1]])
vector = np.array([1, 0])
naive_hhl_solution = HHL().solve(matrix, vector)
```

For the classical solver we need to rescale the right hand side (i.e.) to take into account the renormalisation that occurs once is encoded in a quantum state within HHL. `vector / np.linalg.norm(vector) vector`

```
classical_solution = NumPyLinearSolver().solve(matrix,
                                               vector/np.linalg.norm(vector))
```

The package contains a folder called intended to be a placeholder for efficient implementations of particular types of matrices. At the time of writing the only truly efficient implementation it contains (i.e. complexity scaling polynomially in the number of qubits) is the class. Tridiagonal Toeplitz symmetric real matrices are of the following form `linear_solvers matrices TridiagonalToeplitz`

$$A = \begin{pmatrix} a & b & 0 & 0 \\ b & a & b & 0 \\ 0 & b & a & b \\ 0 & 0 & b & a \end{pmatrix} ,a,b\in \quad \mathbb{R}$$

(note that in this setting we do not consider non symmetric matrices since the HHL algorithm assumes that the input matrix is Hermitian).

Since the matrix A from our example is of this form we can create an instance of and compare the results to solving the system with an array as input. `TridiagonalToeplitz(num_qubits, a, b)`

```
from linear_solvers.matrices.tridiagonal_toeplitz import TridiagonalToeplitz
tridi_matrix = TridiagonalToeplitz(1, 1, -1 / 3)
tridi_solution = HHL().solve(tridi_matrix, vector)
```

Recall that the HHL algorithm can find a solution exponentially faster in the size of the system than their classical counterparts (i.e. logarithmic complexity instead of polynomial). However the cost for this exponential speedup is that we do not obtain the full solution vector.
Instead, we obtain a quantum state representing the vector x and learning all the components of this vector would take a linear time in its dimension, diminishing any speedup obtained by the quantum algorithm.

Therefore, we can only compute functions from x (the so called observables) to learn information about the solution.
This is reflected in the object returned by , which contains the following properties `LinearSolverResult solve()`

- `state` : either the circuit that prepares the solution or the solution as a vector
- `euclidean_norm` : the euclidean norm if the algorithm knows how to calculate it
- `observable` : the (list of) calculated observable(s)
- `circuit_results` : the observable results from the (list of) circuit(s)

Let's ignore and for the time being and check the solutions we obtained before. `observable circuit_results`

First, was the result from a classical algorithm, so if we call it will return an array: `classical_solution .state`

```
print('classical state:', classical_solution.state)
```

```
classical state: [1.125 0.375]
```

Our other two examples were quantum algorithms, hence we can only access to the quantum state. This is achieved by returning the quantum circuit that prepares the solution state:

```
print('naive state:')
print(naive_hhl_solution.state)
print('tridiagonal state:')
print(tridi_solution.state)
```

```
naive state:
     ┌──────────────┐┌─────┐          ┌──────────┐
  q4: ┤ circuit-85 ├┤3    ├──────────┤3         ├
     └──────────────┘│     │          │          │
q5_0: ───────────────┤0    ├─┐      ┌─┤0         ├
                     │  QPE │ │      │ │  QPE_dg  │
q5_1: ───────────────┤1    ├─┤1     ├─┤1         ├
                     │     │ │ 1/x  │ │          │
q5_2: ───────────────┤2    ├─┤0     ├─┤2         ├
                     └─────┘ │      │ └──────────┘
  q6: ───────────────────────┤3     ├─────────────
                             └──────┘
tridiagonal state:
     ┌──────────────┐┌─────┐          ┌──────────┐
  q26: ┤ circuit-298 ├┤3    ├──────────┤3         ├
     └──────────────┘│     │          │          │
q27_0: ──────────────┤0    ├─┐      ┌─┤0         ├
                     │  QPE │ │      │ │  QPE_dg  │
q27_1: ──────────────┤1    ├─┤1     ├─┤1         ├
                     │     │ │ 1/x  │ │          │
q27_2: ──────────────┤2    ├─┤0     ├─┤2         ├
                     └─────┘ │      │ └──────────┘
  q28: ──────────────────────┤3     ├─────────────
                             └──────┘
```

Recall that the Euclidean norm for a vector is defined as `{latex} \mathbf{x}=(x_1,\dots,x_N)` $||\mathbf{x}|| = \sqrt{\sum_{i=1}^{N} x_i^2}$. Therefore, the probability of measuring 1 in the auxiliary qubit from Step 5 in Section B is the squared norm of $\mathbf{x}$. This means that the HHL algorithm can always calculate the euclidean norm of the solution and we can compare the accuracy of the results:

```
print('classical Euclidean norm:', classical_solution.euclidean_norm)
print('naive Euclidean norm:', naive_hhl_solution.euclidean_norm)
print('tridiagonal Euclidean norm:', tridi_solution.euclidean_norm)
```

```
classical Euclidean norm: 1.1858541225631423
naive Euclidean norm: 1.185854122563138
tridiagonal Euclidean norm: 1.1858541225631365
```

Comparing the solution vectors component-wise is more tricky, reflecting again the idea that we cannot obtain the full solution vector from the quantum algorithm. However, for educational purposes we can check that indeed the different solution vectors obtained are a good approximation at the vector component level as well.

To do so first we need to use from the package and extract the right vector components, i.e. those corresponding to the ancillary qubit (bottom in the circuits) being `Statevector quantum_info` 1 and the work qubits (the two middle in the circuits) being 0. Thus, we are interested in the states and , corresponding to the first and second components of the solution vector respectively. `10000 10001`

```
from qiskit.quantum_info import Statevector

naive_sv = Statevector(naive_hhl_solution.state).data
tridi_sv = Statevector(tridi_solution.state).data

# Extract vector components; 10000(bin) == 16 & 10001(bin) == 17
naive_full_vector = np.array([naive_sv[16], naive_sv[17]])
tridi_full_vector = np.array([tridi_sv[16], tridi_sv[17]])

print('naive raw solution vector:', naive_full_vector)
print('tridi raw solution vector:', tridi_full_vector)
```

```
naive raw solution vector: [0.75+3.52055626e-16j 0.25+1.00756137e-16j]
tridi raw solution vector: [0.75-9.01576529e-17j 0.25+3.74736911e-16j]
```

At a first glance it might seem that this is wrong because the components are complex numbers instead of reals. However note that the imaginary part is very small, most likely due to computer accuracy, and can be disregarded in this case (we'll use the array's attribute to get the real part). `.real`

Next, we will divide the vectors by their respective norms to suppress any constants coming from the different parts of the circuits. The full solution vector can then be recovered by multiplying these normalised vectors by the respective Euclidean norms calculated above:

```
def get_solution_vector(solution):
    """Extracts and normalizes simulated state vector
    from LinearSolverResult."""
    solution_vector = Statevector(solution.state).data[16:18].real
    norm = solution.euclidean_norm
    return norm * solution_vector / np.linalg.norm(solution_vector)

print('full naive solution vector:', get_solution_vector(naive_hhl_solution))
print('full tridi solution vector:', get_solution_vector(tridi_solution))
print('classical state:', classical_solution.state)
```

```
full naive solution vector: [1.125 0.375]
full tridi solution vector: [1.125 0.375]
classical state: [1.125 0.375]
```

It should not come as a surprise that is exact because all the default methods used are exact. However, is exact only in the `naive_hhl_solution` `tridi_solution` 2×2 system size case. For larger matrices it will be an approximation, as shown in the slightly larger example below.

```
from scipy.sparse import diags

NUM_QUBITS = 2
MATRIX_SIZE = 2 ** NUM_QUBITS
# entries of the tridiagonal Toeplitz symmetric matrix
# pylint: disable=invalid-name
a = 1
b = -1/3

matrix = diags([b, a, b],
               [-1, 0, 1],
               shape=(MATRIX_SIZE, MATRIX_SIZE)).toarray()

vector = np.array([1] + [0]*(MATRIX_SIZE - 1))
# run the algorithms
classical_solution = NumPyLinearSolver(
                       ).solve(matrix, vector / np.linalg.norm(vector))
naive_hhl_solution = HHL().solve(matrix, vector)
tridi_matrix = TridiagonalToeplitz(NUM_QUBITS, a, b)
tridi_solution = HHL().solve(tridi_matrix, vector)

print('classical euclidean norm:', classical_solution.euclidean_norm)
print('naive euclidean norm:', naive_hhl_solution.euclidean_norm)
print('tridiagonal euclidean norm:', tridi_solution.euclidean_norm)
```

```
classical euclidean norm: 1.237833351044751
naive euclidean norm: 1.209980623111888
tridiagonal euclidean norm: 1.2094577218705271
```

We can also compare the difference in resources from the exact method and the efficient implementation. The $2\times\ 2$ system size is again special in that the exact algorithm requires less resources, but as we increase the system size, we can see that indeed the exact method scales exponentially in the number of qubits while is polynomial. `TridiagonalToeplitz`

```
from qiskit import transpile

MAX_QUBITS = 4
a = 1
b = -1/3

i = 1
# calculate the circuit depths for different number of qubits to compare the use
# of resources (WARNING: This will take a while to execute)
naive_depths = []
tridi_depths = []
for n_qubits in range(1, MAX_QUBITS+1):
    matrix = diags([b, a, b],
                   [-1, 0, 1],
                   shape=(2**n_qubits, 2**n_qubits)).toarray()
    vector = np.array([1] + [0]*(2**n_qubits -1))

    naive_hhl_solution = HHL().solve(matrix, vector)
    tridi_matrix = TridiagonalToeplitz(n_qubits, a, b)
    tridi_solution = HHL().solve(tridi_matrix, vector)

    naive_qc = transpile(naive_hhl_solution.state,
                         basis_gates=['id', 'rz', 'sx', 'x', 'cx'])
    tridi_qc = transpile(tridi_solution.state,
                         basis_gates=['id', 'rz', 'sx', 'x', 'cx'])

    naive_depths.append(naive_qc.depth())
    tridi_depths.append(tridi_qc.depth())
    i +=1


sizes = [f"{2**n_qubits}×{2**n_qubits}"
         for n_qubits in range(1, MAX_QUBITS+1)]
columns = ['size of the system',
           'quantum_solution depth',
           'tridi_solution depth']
data = np.array([sizes, naive_depths, tridi_depths])
ROW_FORMAT ="{:>23}" * (len(columns) + 2)
for team, row in zip(columns, data):
    print(ROW_FORMAT.format(team, *row))
```

```
     size of the system                   2×2                   4×4                   8×8                 16×16
 quantum_solution depth                   334                  2593                 34008                403899
   tridi_solution depth                   565                  5107                 14756                 46552
```

The reason the implementation still seems to need exponential resources is because the current conditioned rotation implementation (step 3 from Section 2.B) is exact (i.e. needs exponential resources in n_l). Instead we can calculate how many more resources the default implementation needs compared to Tridiagonal - since they only differ in how they implement e^{iAt}:

```
print('excess:',
      [naive_depths[i] - tridi_depths[i] for i in range(0, len(naive_depths))])
```

```
excess: [-231, -2514, 19252, 357347]
```

In the near future the plan is to integrate to obtain a polynomial implementation of the conditioned rotation as well. `qiskit.circuit.library.arithmetics.PiecewiseChebyshev`

Now we can return to the topic of observables and find out what the and properties contain. `observable circuit_results`

The way to compute functions of the solution vector $\mathbf{x}$ is through giving the method a as input. There are two types of available which can be given as input: `.solve() LinearSystemObservable LinearSystemObservable`

```
from linear_solvers.observables import AbsoluteAverage, MatrixFunctional
```

For a vector , the observable computes `{latex} \mathbf{x}=(x_1,...,x_N) AbsoluteAverage` $\left|\frac{1}{N}\sum_{i=1}^{N} x_i\right|$.

```
NUM_QUBITS = 1
MATRIX_SIZE = 2 ** NUM_QUBITS
# entries of the tridiagonal Toeplitz symmetric matrix
a = 1
b = -1/3

matrix = diags([b, a, b],
               [-1, 0, 1],
               shape=(MATRIX_SIZE, MATRIX_SIZE)).toarray()
vector = np.array([1] + [0]*(MATRIX_SIZE - 1))
tridi_matrix = TridiagonalToeplitz(1, a, b)

average_solution = HHL().solve(tridi_matrix,
                               vector,
                               AbsoluteAverage())
classical_average = NumPyLinearSolver(
                         ).solve(matrix,
                                 vector / np.linalg.norm(vector),
                                 AbsoluteAverage())

print('quantum average:', average_solution.observable)
print('classical average:', classical_average.observable)
print('quantum circuit results:', average_solution.circuit_results)
```

```
quantum average: 0.7499999999999962
classical average: 0.75
quantum circuit results: (0.4999999999999952+0j)
```

The observable computes `MatrixFunctional` $\mathbf{x}^T B\mathbf{x}$ for a vector $\mathbf{x}$ and a tridiagonal symmetric Toeplitz matrix B. The class takes the main and off diagonal values of the matrix for its constructor method.

```
observable = MatrixFunctional(1, 1 / 2)

functional_solution = HHL().solve(tridi_matrix, vector, observable)
classical_functional = NumPyLinearSolver(
                        ).solve(matrix,
                                vector / np.linalg.norm(vector),
                                observable)

print('quantum functional:', functional_solution.observable)
print('classical functional:', classical_functional.observable)
print('quantum circuit results:', functional_solution.circuit_results)
```

```
quantum functional: 1.8281249999999818
classical functional: 1.828125
quantum circuit results: [(0.6249999999999941+0j), (0.4999999999999952+0j), (0.1249999999999988+0j)]
```

Therefore, contains the final value of the function on `observable` $\mathbf{x}$, while contains the raw values obtained from the circuit and used to process the result of . `circuit_results` `observable`

This 'how to process the result' is better explained by looking at what arguments takes. The method accepts up to five arguments: `.solve()` `solve()`

```
def solve(self, matrix: Union[np.ndarray, QuantumCircuit],
          vector: Union[np.ndarray, QuantumCircuit],
          observable: Optional[Union[LinearSystemObservable, BaseOperator,
                                     List[BaseOperator]]] = None,
          post_rotation: Optional[Union[QuantumCircuit, List[QuantumCircuit]]] = None,
          post_processing: Optional[Callable[[Union[float, List[float]]],
                                             Union[float, List[float]]]] = None) \
        -> LinearSolverResult:
```

The first two are the matrix defining the linear system and the vector right hand side of the equation, which we have already covered. The remaining parameters concern the (list of) observable(s) to be computed out of the solution vector x, and can be specified in two different ways. One option is to give as the third and last parameter a (list of) (s). Alternatively, we can give our own implementations of the , and , where `LinearSystemObservable` `observable` `post_rotation` `post_processing`

- `observable` is the operator to compute the expected value of the observable and can be e.g. a `PauliSumOp`
- `post_rotation` is the circuit to be applied to the solution to extract information if additional gates are needed.
- `post_processing` is the function to compute the value of the observable from the calculated probabilities.

In other words, there will be as many as circuits, and is telling the algorithm how to use the values we see when we print to obtain the value we see when we print
. `circuit_results` `post_rotation` `post_processing` `circuit_results` `observable`

Finally, the class accepts the following parameters in its constructor method: `HHL`

- error tolerance : the accuracy of the approximation of the solution, the default is `1e-2`
- expectation : how the expectation values are evaluated, the default is `PauliExpectation`
- quantum instance: the or backend, the default is a simulation `QuantumInstance` `Statevector`

```
from qiskit import Aer

backend = Aer.get_backend('aer_simulator')
hhl = HHL(1e-3, quantum_instance=backend)

accurate_solution = hhl.solve(matrix, vector)
classical_solution = NumPyLinearSolver(
                    ).solve(matrix,
                            vector / np.linalg.norm(vector))

print(accurate_solution.euclidean_norm)
print(classical_solution.euclidean_norm)
```

```
1.185854122563138
1.1858541225631423
```

B. Running HHL on a real quantum device: optimised example

In the previous section we ran the standard algorithm provided in Qiskit and saw that it uses 7 qubits, has a depth of ~100 gates and requires a total of 54 CNOT gates. These numbers are not feasible for the current available hardware, therefore we need to decrease these quantities. In particular, the goal will be to reduce the number of CNOTs by a factor of 5 since they have worse fidelity than single-qubit gates. Furthermore, we can reduce the number of qubits to 4 as was the original statement of the problem: the Qiskit method was written for a general problem and that is why it requires 3 additional auxiliary qubits.

However, solely decreasing the number of gates and qubits will not give a good approximation to the solution on real hardware. This is because there are two sources of errors: those that occur during the run of the circuit and readout errors.

Qiskit provides a module to mitigate the readout errors by individually preparing and measuring all basis states, a detailed treatment on the topic can be found in the paper by Dewes et al.[3] To mitigate errors, we can use Richardson extrapolation to calculate the error to the zero limit by running the circuit three times, each replacing each CNOT gate by 1, 3 and 5 CNOTs respectively[4]. The idea is that theoretically the three circuits should produce the same result, but in real hardware adding CNOTs means amplifying the error. Since we know we've obtained results with an amplified error, and we can estimate how much the error was amplified by in each case, we can recombine the quantities to get a new result closer to the analytic solution.

Below we give the optimised circuit that can be used for any problem of the form

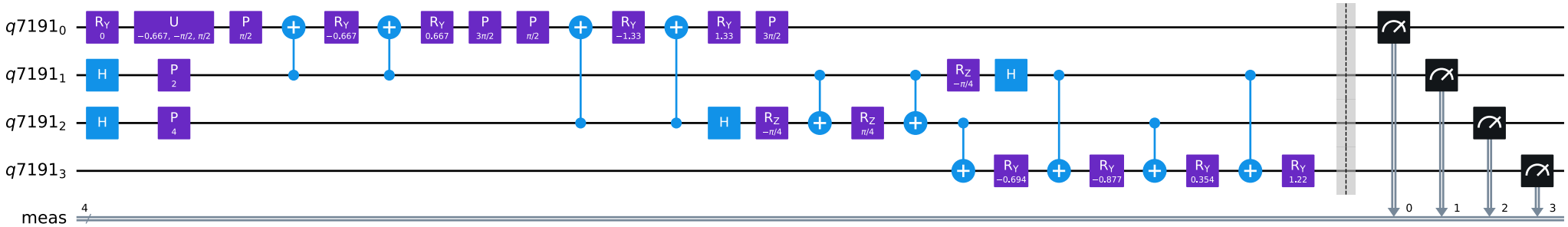
$$A = \begin{pmatrix} a & b \\ b & a \end{pmatrix} \quad \text{and} |b\rangle \quad = \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix}, a, b, \theta \in \quad \mathbb{R}$$

The following optimisation was extracted from a work on the HHL for tridiagonal symmetric matrices[5], this particular circuit was derived with the aid of the UniversalQCompiler software[6].

```
from qiskit import QuantumRegister, QuantumCircuit
import numpy as np

t = 2  # This is not optimal; As an exercise, set this to the
       # value that will get the best results. See section 8 for solution.

NUM_QUBITS = 4  # Total number of qubits
nb = 1  # Number of qubits representing the solution
nl = 2  # Number of qubits representing the eigenvalues

theta = 0  # Angle defining |b>

a = 1  # Matrix diagonal
b = -1/3  # Matrix off-diagonal

# Initialize the quantum and classical registers
qr = QuantumRegister(NUM_QUBITS)

# Create a Quantum Circuit
qc = QuantumCircuit(qr)

qrb = qr[0:nb]
qrl = qr[nb:nb+nl]
qra = qr[nb+nl:nb+nl+1]

# State preparation.
qc.ry(2*theta, qrb[0])

# QPE with e^{iAt}
for qu in qrl:
    qc.h(qu)

qc.p(a*t, qrl[0])
qc.p(a*t*2, qrl[1])

qc.u(b*t, -np.pi/2, np.pi/2, qrb[0])


# Controlled e^{iAt} on \lambda_{1}:
params=b*t

qc.p(np.pi/2,qrb[0])
qc.cx(qrl[0],qrb[0])
qc.ry(params,qrb[0])
qc.cx(qrl[0],qrb[0])
qc.ry(-params,qrb[0])
qc.p(3*np.pi/2,qrb[0])
```

```
# Controlled e^{2iAt} on \lambda_{2}:
params = b*t*2

qc.p(np.pi/2,qrb[0])
qc.cx(qrl[1],qrb[0])
qc.ry(params,qrb[0])
qc.cx(qrl[1],qrb[0])
qc.ry(-params,qrb[0])
qc.p(3*np.pi/2,qrb[0])

# Inverse QFT
qc.h(qrl[1])
qc.rz(-np.pi/4,qrl[1])
qc.cx(qrl[0],qrl[1])
qc.rz(np.pi/4,qrl[1])
qc.cx(qrl[0],qrl[1])
qc.rz(-np.pi/4,qrl[0])
qc.h(qrl[0])

# Eigenvalue rotation
t1=(-np.pi +np.pi/3 - 2*np.arcsin(1/3))/4
t2=(-np.pi -np.pi/3 + 2*np.arcsin(1/3))/4
t3=(np.pi -np.pi/3 - 2*np.arcsin(1/3))/4
t4=(np.pi +np.pi/3 + 2*np.arcsin(1/3))/4

qc.cx(qrl[1],qra[0])
qc.ry(t1,qra[0])
qc.cx(qrl[0],qra[0])
qc.ry(t2,qra[0])
qc.cx(qrl[1],qra[0])
qc.ry(t3,qra[0])
qc.cx(qrl[0],qra[0])
qc.ry(t4,qra[0])
qc.measure_all()

print(f"Depth: {qc.depth()}")
print(f"CNOTS: {qc.count_ops()['cx']}")
qc.draw(fold=-1)
```

```
Depth: 26
CNOTS: 10
```



The code below takes as inputs our circuit, the real hardware backend and the set of qubits we want to use, and returns and instance that can be run on the specified device. Creating the circuits with 3 and 5 CNOTs is the same but calling the transpile method with the right quantum circuit.

Real hardware devices need to be recalibrated regularly, and the fidelity of a specific qubit or gate can change over time. Furthermore, different chips have different qubit connectivity. If we try to run a circuit that performs a two-qubit gate between two qubits that are not connected on the specified device, the transpiler will add SWAP gates. Therefore it is good practice to check with the IBM Quantum Experience webpage[7] before running the following code and choose a set of qubits with the right connectivity and lowest error rates at the given time.

```
from qiskit import IBMQ, transpile
from qiskit.utils.mitigation import complete_meas_cal

provider = IBMQ.load_account()

backend = provider.get_backend('ibmq_quito') # calibrate using real hardware
layout = [2,3,0,4]
chip_qubits = 5

# Transpiled circuit for the real hardware
qc_qa_cx = transpile(qc, backend=backend, initial_layout=layout)
```

The next step is to create the extra circuits used to mitigate the readout errors[3].

```
meas_cals, state_labels = complete_meas_cal(qubit_list=layout,
                                            qr=QuantumRegister(chip_qubits))
qcs = meas_cals + [qc_qa_cx]

job = backend.run(qcs, shots=10)
```

The following plot[5], shows the results from running the circuit above on real hardware for 10 different initial states. The x-axis represents the angle θ defining the initial state in each case. The results where obtained after mitigating the readout error and then extrapolating the errors arising during the run of the circuit from the results with the circuits with 1, 3 and 5 CNOTs.

Compare to the results without error mitigation nor extrapolation from the CNOTs[5].

8. Problems

Real hardware:

1. Set the time parameter for the optimised example.

▶ Solution (Click to expand)

2. Create transpiled circuits for 3 and 5 CNOTs from a given circuit ''. When creating the circuits you will have to add barriers so that these consecutive CNOT gates do not get cancelled when using the function. `qc transpile()`
3. Run your circuits on the real hardware and apply a quadratic fit to the results to obtain the extrapolated value.

9. References

1. J. R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1994.
2. A. W. Harrow, A. Hassidim, and S. Lloyd, “Quantum algorithm for linear systems of equations,” Phys. Rev. Lett. 103.15 (2009), p. 150502.
3. A. Dewes, F. R. Ong, V. Schmitt, R. Lauro, N. Boulant, P. Bertet, D. Vion, and D. Esteve, “Characterization of a two-transmon processor with individual single-shot qubit readout,” Phys. Rev. Lett. 108, 057002 (2012).
4. N. Stamatopoulos, D. J. Egger, Y. Sun, C. Zoufal, R. Iten, N. Shen, and S. Woerner, “Option Pricing using Quantum Computers,” arXiv:1905.02666 .
5. A. Carrera Vazquez, A. Frisch, D. Steenken, H. S. Barowski, R. Hiptmair, and S. Woerner, “Enhancing Quantum Linear System Algorithm by Richardson Extrapolation,” ACM Trans. Quantum Comput. 3 (2022).
6. R. Iten, O. Reardon-Smith, L. Mondada, E. Redmond, R. Singh Kohli, R. Colbeck, “Introduction to UniversalQCompiler,” arXiv:1904.01072 .
7. https://quantum-computing.ibm.com/ .
8. D. Bucher, J. Mueggenburg, G. Kus, I. Haide, S. Deutschle, H. Barowski, D. Steenken, A. Frisch, "Qiskit Aqua: Solving linear systems of equations with the HHL algorithm" https://github.com/Qiskit/qiskit-tutorials/blob/master/legacy_tutorials/aqua/linear_systems_of_equations.ipynb


```
# pylint: disable=unused-import
import qiskit.tools.jupyter
%qiskit_version_table
```

Version Information

Qiskit Software	Version
`qiskit-terra`	0.21.0
`qiskit-aer`	0.10.4
`qiskit-ibmq-provider`	0.19.2
`qiskit`	0.37.2
`qiskit-nature`	0.4.1
`qiskit-finance`	0.3.2

`qiskit-optimization`	0.4.0
`qiskit-machine-learning`	0.4.0
System information	
Python version	3.8.13
Python compiler	Clang 13.1.6 (clang-1316.0.21.2.5)
Python build	default, Aug 29 2022 05:17:23
OS	Darwin
CPUs	8
Memory (Gb)	32.0
Wed Sep 21 15:26:35 2022 BST	