Term paper: Quantum Algorithms for Solving Linear Equations

tomtuamnuq

25.09.2020

Contents

Figures

Tables

1. Quantum Computation

Quantum computing as a sub-field of quantum information science performs computation using quantum mechanical phenomena. The informational units are called qubits and, unlike bits, can be in a superposition of 0 and 1. Furthermore, a system of more than one qubit can be entangled. This means that the state of a single qubit cannot be expressed isolated. Each qubit depends on the state of the system.
In theory, quantum computers can solve certain problems exponentially faster than classical hardware. One example is to obtain certain properties of a solution to a linear system of equations. This is described in section 2.
Practically, the main obstacle is the problem of *decoherence*. The environment of a quantum computer has a huge impact on the state. Being not perfectly isolated, information will be lost quickly and the error rates rise.
Global players like IBM, Microsoft and Google are creating publicly accessible quantum computers. With *IBM's Qiskit Software*, a python framework is given to manipulate systems at gate level. This paper shows concrete implementations with Qiskit in section 3. Error rates of available devices are demonstrated and the results are compared to a simulation[1].
In the following subsections, the basic math behind the quantum computation is given and IBM's quantum computation framework is described briefly.

1.1. Linear Algebra

The natural language of quantum mechanics is linear algebra. Therefore it is appropriate to give a short introduction.

Notation 1. *Throughout this paper, A, U will be matrices with dimension $n \times n$ over the complex field $\mathbb{C}$ and x, b will be column vectors in a complex vector space $\mathbb{C}^n$ with dimension $n \in \mathbb{N}$. Let γ denote a real scalar. $\mathbb{I}_n$ denotes the identity matrix.*

A useful way to describe complex numbers is by Euler's Formula:

Theorem 1. *Given the imaginary unit i and the base of the natural logarithm e,* Euler's Formula *states that $e^{i\gamma} = \cos(\gamma) + i\sin(\gamma)$.*

To describe the math behind quantum algorithms we basically need two special types of matrices.

Definition 1.1. *A matrix U is called* unitary *if the inverse is the complex conjugate transpose:*

$$U^\dagger U = UU^\dagger = \mathbb{I}_n$$

Theorem 2. *Unitary matrices are invertible and diagonalizable with eigenvalues having norm 1.*
*[***LineareAlgebra.2009***]*

[1] The author acknowledges the use of IBM Quantum services for this work. The views expressed are those of the author, and do not reflect the official policy or position of IBM or the IBM Quantum team.

Definition 1.2. *A is called* hermitian*, if $A = A^\dagger$, which means that A is equivalent to its complex conjugate transpose.*

Theorem 3. *If A is hermitian, A is diagonalizable and there exist n linearly independent eigenvectors $u \in \mathbb{C}^n$ so that $Au = \lambda u$ with eigenvalues $\lambda \in \mathbb{R}$* *[**LineareAlgebra.2009**]*

The last property will be very useful to describe the algorithm for solving linear equations mathematically. Therefore we denote:

Notation 2. *Let the n different eigenvectors of A be $u_0, ..., u_{n-1}$ and the n (not necessarily different) eigenvalues $\lambda_0, ..., \lambda_{n-1}$, respectively.*

The eigenvalues will become valuable to outline complexity. To do this, we use the condition number.

Definition 1.3. *The* condition κ *of an invertible matrix A is defined as:*

$$\kappa = \max\{\frac{|\lambda_i|}{|\lambda_j|} \mid 0 \leq i, j \leq n-1\}$$

The *spectral theorem* for hermitian matrices says that there exists an orthonormal basis of $\mathbb{C}^n$ consisting of eigenvectors of A. Eigenvectors of A can be found by using the *Characteristic polynomial* and form a basis of $\mathbb{C}^n$, if A is invertible. The *Gram Schmidt Process* orthonormalizes a given basis [**LineareAlgebra.2009**].

Theorem 4. *Assume a hermitian matrix A. There exists a set of n orthonormal eigenvectors of A. Let P be the matrix with such vectors as columns. P is unitary and $AP = P\Lambda$ with the diagonal matrix Λ having the eigenvalues of A. All eigenvalues of A are real, so $\Lambda \in \mathbb{R}^{n \times n}$ [**LineareAlgebra.2009**].*

In quantum computation one often sees transformations of the form $e^{i\gamma A}$. This is done by using matrix exponentiation, which is described by Taylor expansion.

Definition 1.4. *The* matrix exponentiation *of A is $U = e^{i\gamma A}$.*

If A is hermitian, U is unitary and has the same eigenvectors as A with eigenvalues $e^{i\gamma\lambda}$ [**HectorAbraham.2019**].
Given the basis every element of the vectorspace has a unique representation.
Therefore exist $x_0, ..., x_{n-1}, b_0, ..., b_{n-1} \in \mathbb{C}$ such that:

$$x = \sum_{j=0}^{n-1} x_j u_j \text{ and } b = \sum_{j=0}^{n-1} b_j u_j$$

For complexity reasons it is helpful to know the number of non zero entries in a matrix.

Definition 1.5. *A matrix is called* sparse*, if most entries are zero. The* sparsity *s is defined as the maximum number of non zero entries per row or column. The opposite is a dense matrix.*

1.2. Quantum States

In quantum mechanics it is common to use the bra–ket notation, introduced by famous quantum physicist Paul Dirac [**Gieres.2000**]. A ket denotes a column vector, which represents a quantum state, and a bra a row vector. In particular, given the vector x, the ket is $|x\rangle$ and the bra is $\langle x| = x^\dagger$.
The product of a bra and a ket is the complex inner product and the product of a ket with a bra is an outer product and therefore a matrix. In this notation, one can write the matrix A in spectral decomposition as product of ket's and bra's:

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

A single qubit is represented as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix}\alpha & \beta\end{pmatrix}^T \in \mathbb{C}^2$ under the probability condition $|\alpha|^2 + |\beta|^2 = 1$.
A classical bit is either in the state 0 or 1. A qubits state is a linear combination of both. This is known as *superposition*. The state $|+\rangle$ is a superposition of 0 and 1, denoted as the following complex vectors:

Notation 3.

$$|0\rangle = \begin{pmatrix}1 & 0\end{pmatrix}^T, |1\rangle = \begin{pmatrix}0 & 1\end{pmatrix}^T, |+\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\end{pmatrix}^T$$

An *Observable* is represented by a hermitian matrix M. An Observation is the application of a hermitian operator. Popular observables are the Pauli operators:

Notation 4.

$$X = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}, Y = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}, Z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

Measuring in the computational basis means to observe the qubits state in Z direction. The *computational basis* is formed by the eigenvectors $|0\rangle$ and $|1\rangle$ of the Pauli Z operator.
The squared complex norm of the coefficients, $|\alpha|^2$ and $|\beta|^2$, are the probabilities of measuring the qubit as $|0\rangle$ or $|1\rangle$.
An important fact is that due to the restrictions of probabilities in quantum mechanics, statevectors like x and b must be normalized such that $\langle b|b\rangle = 1 = \langle x|x\rangle$. This is done by rescaling b as $|b\rangle = \frac{b}{\sqrt{\langle b|b\rangle}}$.
A measurement shows an eigenvalue of M and the resulting state is a corresponding eigenvector of M [**Yanofsky.2013**].

Definition 1.6. *The* expectation value *of a hermitian matrix M and a vector $|\psi\rangle$ is* $\langle M\rangle_\psi = \langle\psi \mid M\psi\rangle$.

$\langle M\rangle_\psi$ is the expected value of observing M repeatedly on the same state ψ [**Yanofsky.2013**].

A quantum system of more than one state is described by the *tensorproduct* of the individual states. This means, given $|\psi\rangle \in \mathbb{C}^n, |\phi\rangle \in \mathbb{C}^m$, the resulting state is $|\psi\rangle \otimes |\phi\rangle \in \mathbb{C}^{nm}$ [**Yanofsky.2013**].
For example, if we have three qubits, a measurement of the joint state will show one of eight possible outcomes. Let the three qubits be:
$|a\rangle = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, |b\rangle = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}, |c\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \in \mathbb{C}^2$
The *statevector* of the system is:

$$\begin{aligned} |abc\rangle = |a\rangle \otimes |b\rangle \otimes |c\rangle &= \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \otimes \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \otimes |c\rangle \\ &= \begin{pmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{pmatrix} \otimes |c\rangle = \begin{pmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{pmatrix} \otimes \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \\ &= \begin{pmatrix} a_0 b_0 c_0 \\ a_0 b_0 c_1 \\ a_0 b_1 c_0 \\ a_0 b_1 c_1 \\ a_1 b_0 c_0 \\ a_1 b_0 c_1 \\ a_1 b_1 c_0 \\ a_1 b_1 c_1 \end{pmatrix} \end{aligned}$$

The probability of measuring the system in the state
$|000\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T$ is $|a_0 b_0 c_0|^2$.
For a system with n qubits the state space has dimension 2^n. Each of the entries in the statevector is a complex number.
For reading purposes the $\otimes$ is omitted.
Imagine a system with n classical bits. The state space has also dimension 2^n.
Contrary to a quantum system the classical version is always in one of the 2^n states.
The vector space is over the field $\mathbb{F}_2$. Exactly one entry of the statevector is a one and all the other entries are zero.
A quantum system has the feature of being in a state that cannot be described with one of the 2^n states alone. There is no assignment of individual states that would fit.
In terms of the tensorproduct this means one is not able to write the system separated.
Such states are called *entangled*.
An alternative description of a qubit uses two real variables ϕ, θ. With Euler's Formula the state can be written as:

$$|\psi\rangle = \cos(\frac{\theta}{2}) |0\rangle + e^{i\phi} \sin(\frac{\theta}{2}) |1\rangle \tag{2}$$

The global phase of a quantum state is not detectable. Such terms vanish in the process of observation. This means the state $|\psi\rangle$ is equal to $e^{i\gamma} |\psi\rangle$.

Only the difference between $|0\rangle$ and $|1\rangle$ is observable. Its called the *relative phase* [**HectorAbraham.2019**].
An intuitive way to imagine the state of a qubit is the *Bloch-sphere*, as shown in figure 1. Using equation (2) with ϕ, θ as spherical coordinates, the qubits state is a point on the surface of the unit sphere $\mathbb{R}^3$. Note that the sphere cannot represent an entangled state. Details can be found in the book "Quantum Computing for Computer Scientists" [**Yanofsky.2013**].

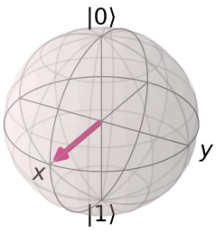

Figure 1: A qubit in the state $|+\rangle$

1.3. Programming a Quantum Computer

1.3.1. Gates and Routines

Quantum states are changed by applying gates. Mathematically, this means to multiply a unitary matrix to the current statevector.
The Hadamard gate $H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$ changes the basis representation of one qubit from computational Z to X and vice versa. After performing H on a qubit in state $|0\rangle$, it is in superposition $H|0\rangle = |+\rangle$.
$|+\rangle$ is an eigenvector of Pauli X. X flips the states $|0\rangle$ and $|1\rangle$. It can therefore be seen as the quantum version of a classical NOT Gate.
The unitary S gate is the root of the Pauli Z and therefore $\begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}$. S adds a phase of $\frac{\pi}{2}$. By applying S a qubit does a $\frac{\pi}{2}$ radian rotation around the Z-axis.
The inverse is $S^\dagger = \overline{S^T} = \begin{pmatrix}1 & 0\\ 0 & -i\end{pmatrix}$.
Rotations around the Y-axis of the Bloch-Sphere are performed by:

$$R_y(\theta) = e^{-i\theta\frac{Y}{2}} = \cos(\frac{\theta}{2})\mathbb{I}_2 - i\sin(\frac{\theta}{2})Y = \begin{pmatrix}\cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2})\\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2})\end{pmatrix}$$

A controlled unitary operation C-U applies U on the target qubit only if the corresponding control qubit is in the state $|1\rangle$. For example, the C-R_y operation is:

$$C\text{-}R_y(\theta) = \begin{pmatrix}\mathbb{I}_2 & 0\\ 0 & R_y(\theta)\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2})\\ 0 & 0 & \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2})\end{pmatrix}$$

Using controlled operations has often the effect of adding relative phase to a control qubit. This is called *phase kickback* [**HectorAbraham.2019**].
The *Hamiltonian simulation* of a hermitian matrix A is done by applying the unitary matrix e^{itA} for a specific time t. This is an outcome of the time dependent Schrödinger equation [**Yanofsky.2013**].
Quantum computers implement reversible logic. The number of in,- and output qubits on a gate must be equal. Therefore it is often necessary to introduce extra qubits. Those are called ancillary qubits or *ancillas*.

A quantum algorithm usually has three basic steps. At first, data needs to be encoded into qubits of one or more registers. Secondly, a gate-sequence is applied to change the qubits state. This step often contains subroutines. Lastly, one or more qubits are measured. This results in a classically interpretable outcome.
The *quantum Fourier transformation (QFT)* is the well known Fourier transformation, working on quantum registers. Therefore the inverse, denoted as $QFT^\dagger$, transforms Fourier basis states to Z-basis states. A general implementation can be found in the Qiskit textbook [**HectorAbraham.2019**].
The *quantum phase estimation (QPE)* gives an approximation for the eigenvalues of a unitary matrix U. Given $U|\psi\rangle = e^{i2\pi\theta}|\psi\rangle, \theta \in \mathbb{R}$, the algorithm applies multiple controlled unitary operations with U and uses the phase kickback in the Fourier basis states. This means, as a result of the application of U on the target qubit, relative phase with respect to the eigenvalues is added to a control qubit.
After performing $QFT^\dagger$ on the control qubits, the eigenvalues can be found in computational basis in the state of the control qubits. With a register of n control qubits as precision, initialized as $|0\rangle^{\otimes n}$, the algorithm ends with the state $|2^n\theta\rangle$ as a n bit binary approximation in the control register [**HectorAbraham.2019**].

1.3.2. Qiskit Software

Qiskit is an open-source quantum programming framework that enables developers to get started with quantum programming in an intuitive way [**HectorAbraham.2019**].
The foundational roots of the software are provided by Qiskit *Terra*. The central unit is the quantum circuit. The gates as basic building blocks are added to the circuit in a straightforward manner. One is able to visualize such circuits. Figure 2 shows a code example[2] with output.
The U_3 gate is Qiskits universal single-qubit rotation gate with three angles. It is physically implemented and every qubit operation is compiled down to combinations involving this gate. Therefore Qiskit automatically replaces the R_y rotation gate with two U_3 gates and adds C-X to implement a controlled version. The U_1 and U_2 gates are special cases of the U_3 gate with fixed angles.

[2]The implementations in this paper were done with Qiskit meta package version 0.18.3.

```
from qiskit import QuantumCircuit
from math import pi
qc=QuantumCircuit(2) #initialize 2 qubits
qc.h(0) #add an hadamard gate to qubit 0
#rotate qubit 1 around the y axis with an angle of pi/2:
qc.cry(pi/2,0,1)
qc.draw('mpl')
```

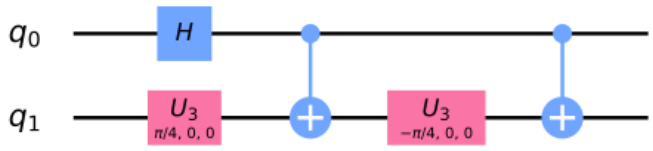

Figure 2: Qiskit code example of a quantum circuit and the outcome of the draw method

Notation 5.

$$U_3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i\lambda+i\phi}\cos(\frac{\theta}{2}) \end{pmatrix}$$

$$U_2(\phi,\lambda) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i\lambda+i\phi} \end{pmatrix} = U_3(\frac{\pi}{2},\phi,\lambda)$$

$$U_1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix} = U_3(0,0,\lambda)$$

Qiskit *Aer* provides a framework to simulate quantum devices. The *StatevectorSimulator* calculates the exact state of a quantum system after one circuit execution. The *QasmSimulator* enables noisy or noise free multi shot execution of a circuit.
In the *qiskit.aqua* package a library of quantum algorithms and components is already available. For example, the class QSVM in qiskit.aqua.algorithms provides a quantum algorithm for data classification.

1.3.3. IBM Quantum systems

The IBM Quantum systems are using *superconducting transmon qubits* as basis for electrically controlled solid-state quantum computers. Electromagnetic microwave pulses with a particular phase, frequency and duration change the qubits state and physically perform the gates. A *dilution refrigerator* cools the system down to 0.0015 Kelvin.
A set of quantum processors is publicly available via the online platform *IBM Q Experience*. The site provides a forum, a set of tutorials and lots of useful

tools. The machines can be accessed over cloud by using the *AccountProviders* class in the package *qiskit.providers.ibmq* [**HectorAbraham.2019**]. The *IBMQBackend* class provides technical information about quantum devices.
One of the backends is the *IBM Q Rome*. The basic gates $U_1, U_2, U_3, C\text{-}X, id$ are implemented. The system provides five qubits. The connectivity and error rates are shown in figure 3.

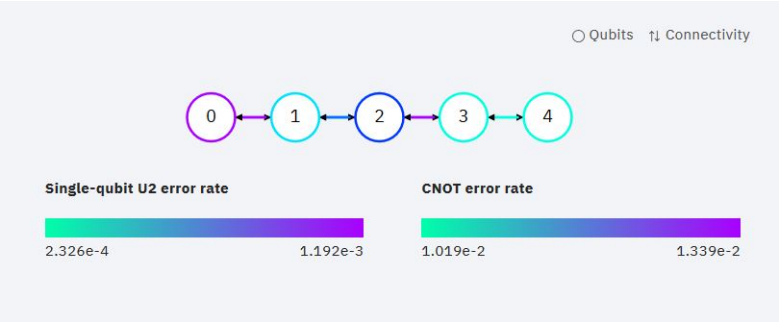

Figure 3: Connectivity and error rates of the IBM Q Rome qubits [**IBMQuantumteam.202010.06.2020**]

The noise model of the IBM Q Rome is available by using the from_backend method in the class *NoiseModel* in *qiskit.providers.aer.noise*.
Experiments on real devices and the simulator can be started by *qiskit.execute*.
Figure 4 shows an example.

```
from qiskit import QuantumCircuit, Aer, execute, IBMQ
from qiskit.providers.aer.noise import NoiseModel
qc=QuantumCircuit(2)
IBMQ.load_account()
provider=IBMQ.get_provider(hub='ibm-q')
backend=provider.get_backend("ibmq_rome")
noise_model = NoiseModel.from_backend(backend)
result=execute(qc,Aer.
 ↪get_backend('qasm_simulator'),noise_model=noise_model,shots=4096).
 ↪result()
```

Figure 4: Qiskit code for 4096 simulated measurements of a circuit on the IBM Q Rome

2. A Quantum Algorithm for Solving Linear Equations

Quantum computing could become extremely valuable in any situations where we're dealing with a huge amount of data. Examples are financial models, fluid simulation

and machine learning. In big data, the problem of solving linear equations arises very often.
Dating 2009, Harrow, Hassidim and Lloyd proposed an algorithm for solving linear systems of equations with a quantum computer. The algorithm shows an exponential speed-up over all known classical approaches. Complexity issues are discussed in subsection 2.4. It holds out hope quantum computers could help to cope with increasing amount of data in real world applications.
The algorithm already had a profound impact in the young research area of quantum computation. Barz et al. used a simplified circuit with one state and one eigenvalue qubit to demonstrate their "two-qubit photonic quantum processor" [**Barz.2014**]. Zhao et al. described an algorithm for Quantum Bayesian training of neural networks. The algorithm to solve linear equations is used as part of the Quantum Gaussian process algorithm [**Zhao.2019**] as well as a subroutine for matrix multiplication and inversion in Quantum Data-Fitting [**Wiebe.2012**]. The ideas have been used to create an algorithm for solving non-linear differential equations [**Leyton.12232008**].
The efficient implementation is still an important research focus. There exists an approach to solve a crucial linear equation in a quantum support vector machine [**J.4102018**]. In 2019, a Hybrid quantum linear equation algorithm was proposed to overcome practical disadvantages of the original algorithm [**Lee.2019**].
In the following subsections, the algorithm as proposed by Harrow, Hassidim and Lloyd is described step by step with respect to subsequent research.

2.1. Linear Equations

The classical problem: Given an invertible matrix A and a vector b, find a vector x such that $Ax = b$.

The quantum solution is: Given a quantum state $|b\rangle$ and an invertible, hermitian matrix A, find a quantum state $|x\rangle$ such that $A|x\rangle = |b\rangle$.
This is basically done by using spectral decomposition $A = P^\dagger \Gamma P$, see equation (1). Using theorem 2 and theorem 4, the inverse of A is given as:

$$A^{-1} = (P^\dagger \Gamma P)^{-1} = P^\dagger \Gamma^{-1} P = \sum_{j=0}^{n-1} \lambda_j^{-1} |u_j\rangle \langle u_j| \tag{3}$$

Therefore the non-unit solution is:

$$x = A^{-1} b = \sum_{j=0}^{n-1} \lambda_j^{-1} b_j u_j \tag{4}$$

Holding the probability condition the resulting state is:

$$|x\rangle = \frac{A^{-1}|b\rangle}{\sqrt{b^\dagger (A^{-1})^2 b}} \tag{5}$$

If the given matrix is invertible, but not hermitian, one can solve the system $\begin{pmatrix} 0 & A \\ A^\dagger & 0 \end{pmatrix} \begin{pmatrix} 0 \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$ with the hermitian matrix $\begin{pmatrix} 0 & A \\ A^\dagger & 0 \end{pmatrix}$ [**Harrow.2009**].

2.2. The Quantum Algorithm

We can imagine the algorithm in a nutshell as transformations of equations [**J.4102018**]:

$$P^\dagger \Gamma P |x\rangle = |b\rangle \overset{step\ 2}{\rightarrow} \Gamma P |x\rangle = P |b\rangle \overset{step\ 3}{\rightarrow} P |x\rangle = \Gamma^{-1} P |b\rangle \overset{step\ 4}{\rightarrow} |x\rangle = P^\dagger \Gamma^{-1} P |b\rangle \quad (6)$$

The algorithm uses three quantum registers. The first, denoted as a, simply holds one ancilla. The second,r , holds enough qubits to store the n eigenvalues of A in binary format. Therefore n would be best. The last register,m , stores the state $|b\rangle$. Therefore it needs $\lceil \log(n) \rceil$ qubits. The circuit is given in subsection 2.3.

Step 1: State preparation The preparation of the state $|b\rangle$ is done by mapping the j-th entry to the amplitude of the corresponding computational basis state $|j\rangle$ in the m register. Grover and Rudolph showed how this can be done efficiently for most cases [**Grover.2002**]. Clader et al. developed a state preparation routine that can initialize generic states [**Clader.2013**].
If $|b\rangle$ is in a different basis, for example the Z-Basis, the j-th amplitude with respect to the eigenvectors of A is given by $b_j = \langle u_j | b \rangle$ because the eigenvectors of A form an orthonormal basis [**LineareAlgebra.2009**].
After step 1 the state of the quantum system is $|0\rangle_a |0\rangle_r^{\otimes n} |b\rangle_m$. The leftmost ket is the ancilla and the $^{\otimes n}$ means that the r register starts with n qubits in state $|0\rangle$. The state of the system is the tensorproduct of the states of the three registers.

Step 2: Quantum phase estimation In this step the QPE is applied with the unitary $U := e^{iAt}$. In easy terms this can be seen as multiplying P from the left in equation (6). With conditional Hamiltonian simulation for a specific time t_0 in U, where the r register is used as control and m as target, the phase kickback in the Fourier basis $|k\rangle$ will be found in r. After the inverse Fourier transformation the two registers r and m are in superposition with the state:

$$\sum_{j=0}^{n-1} \sum_{k=0}^{T-1} \alpha_{k|j} b_j |k\rangle_r |u_j\rangle_m \qquad [\textbf{Harrow.2009}] \quad (7)$$

Here T denotes the available dimension in the complex vector space given $\log(T)$ control qubits as precision. Given n qubits in the eigenvalue register this becomes 2^n. The amplitudes $\alpha_{k|j}$ of the Fourier basis states are concentrated on k values satisfying $\lambda_j \approx \frac{2\pi k}{t}$. The exact state (given enough precision) of the system would be proportional to:

$$\sum_{j=0}^{n-1} |0\rangle_a |\lambda_j\rangle_r b_j |u_j\rangle_m \qquad [\textbf{Pan.2014}] \quad (8)$$

Step 3: Eigenvalue inversion The eigenvalue inversion of A is applied as follows. Conditioned on each λ_j, the ancilla will be rotated around the Y-axis to the state:

$$\sqrt{1-\frac{C^2}{\lambda_j^2}}\,|0\rangle + \frac{C}{\lambda_j}\,|1\rangle \qquad \textbf{[J.4102018]} \tag{9}$$

The rotation angle for the j-th eigenvalue is $\theta_j = 2arcsin(\frac{C}{\lambda_j})$, so controlled $R_y(\theta_j)$ is performed with the ancilla as target.
C is a non-zero normalization constant, bounded by $\mathcal{O}(\frac{1}{\kappa})$.
Keep in mind that we need to calculate the reciprocals of the eigenvalues to get the right angles. Cao et al. introduced a subroutine to approximate the state $|\theta_j\rangle$ as control qubits [**Cao.2012**]. Lee et al. used a classical computer to solve this problem. This is further described in subsection 3.2.
After step 3 the system state is:

$$\sum_{j=0}^{n-1}\Big(\sqrt{1-\frac{C^2}{\lambda_j^2}}\,|0\rangle_a + \frac{C}{\lambda_j}\,|1\rangle_a\Big)|\lambda_j\rangle_r\, b_j\,|u_j\rangle_m \tag{10}$$

Step 4: Inverse quantum phase estimation In this step, $P^\dagger$ is multiplied from the left in equation (6).
To do this in terms of the quantum state, the inverse of the phase estimation subroutine is applied. If the phase estimation and its inverse were perfect, the coefficients $\alpha_{k|j}$ would have 1 if $\lambda_j = \frac{2\pi k}{t}$ and 0 otherwise [**Harrow.2009**]. This will disentangle the eigenvalue register and brings the system to the state:

$$\sum_{j=0}^{n-1}\Big(\sqrt{1-\frac{C^2}{\lambda_j^2}}\,|0\rangle_a + \frac{C}{\lambda_j}\,|1\rangle_a\Big)|0\rangle_r\, b_j\,|u_j\rangle_m \qquad \textbf{[J.4102018]} \tag{11}$$

Step 5: Projective measurement on the ancilla If the ancilla is measured in the state $|1\rangle$, the state of the system collapses to:

$$\sum_{j=0}^{n-1}\frac{C}{\lambda_j}\,|1\rangle_a\,|0\rangle_r\, b_j\,|u_j\rangle_m \tag{12}$$

Therefore the state of the m register is:

$$\sum_{j=0}^{n-1} b_j\frac{C}{\lambda_j}\,|u_j\rangle = C\sum_{j=0}^{n-1}\frac{b_j}{\lambda_j}\,|u_j\rangle = CA^{-1}\,|b\rangle = |x\rangle \tag{13}$$

The statevector $|x\rangle$ in the computational basis is proportional to the solution x. The probability of collapsing into this state is $|C|^2\sum_{j=1}^{n-1}|\frac{b_j}{\lambda_j}|^2$ [**Cao.2012**]. It is necessary to repeat the steps 2 to 5 until the condition of the ancilla is met. The probability is at least $\Omega(\frac{1}{\kappa^2})$ [**Harrow.2009**]

Step 6: Measurement To gain statistical information about the solution state, we can measure the expectation values $\langle X \rangle_x, \langle Y \rangle_x, \langle Z \rangle_x$ with respect to the Pauli operators or other observables [**J.4102018**].

2.3. Circuit Design

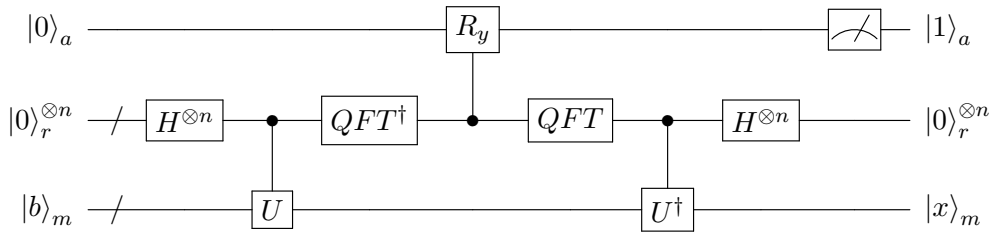

Figure 5: This is the abstract circuit of steps 2 to 5, created with the LaTeX package *qcircuit* [**Eastin.2004**]. The QPE(step 2) is the left part with the Hadamard, C-U and $QFT^\dagger$. In the center one can see step 3, the controlled rotations on the ancilla. This is followed by $QPE^\dagger$(step 4). Step 5 is the projective measurement of the ancilla in state $|1\rangle$.

Figure 5 shows the general circuit design. Qiskit Aqua (qiskit.aqua.algorithms.hhl) provides a general solution for hermitian matrices and arbitrary input states. Currently the output circuits are too large to be tested on real hardware. In the Qiskit textbook an implementation for special 2×2 systems is described using the recent research of HHL for tridiagonal symmetric matrices [**HectorAbraham.2019**].

2.4. Complexity

In this subsection, we use the condition κ (def. 1.3) and sparsity (def. 1.5) of the matrix A. The condition number is associated with the inaccuracy of finding the solution x. Let ϵ be the total error in the output solution x. We assume that the input state $|b\rangle$ is given.
A classical algorithm gives a full solution. The HHL algorithm estimates a function to the solution. This is an important difference.
Every classical algorithm would need at least $\Omega(n)$ to read or write the complete solution vector. The quantum version gives the possibility to encode the solution in a quantum state in $\mathcal{O}(\log(n))$. This limits to the accessibility of certain features of the solution, e.g. the expectation value of an operator [**Cao.2012**].
Since the original HHL work only covers the sparse case in terms of runtime completely, we need to distinguish between the sparse and dense case of the given matrix A. Theoretically, the best known complexity for a sparse matrix is classically $\mathcal{O}(nsk \log(\frac{1}{\epsilon}))$ [**Shewchuk.1994**] using the conjugate gradient method. The HHL algorithm has complexity $\mathcal{O}(\log(n)\frac{s^2\kappa^2}{\epsilon})$ for a sparse matrix, which shows an exponential speed-up in some cases. A test for similar stable states in two different

stochastic processes is described in the original paper [**Harrow.2009**]. Andris Ambainis applied variable time amplitude amplification to the HHL algorithm to improve the dependency on κ [**Ambainis.10212010**]. Childs et al. proposed methods to improve the dependency on ϵ [**Childs.2017**].
For a dense matrix on a classical computer, the best complexity is $\mathcal{O}(n^{2.376})$, using the algorithm by Don Coppersmith and Shmuel Winograd [**Coppersmith.1990**].
Applying the original HHL algorithm to a dense matrix, the exponential advantage is lost. A quantum approach for dense matrices, maintaining the runtime advantage, was given by Wossnig et al. [**Wossnig.2018**].

3. Implementations of Quantum Linear Solvers

Section 3 shows circuit implementations in Qiskit and experiments on the simulator and quantum hardware. Three initial states $|b\rangle$ were tested on the simulator and measured in different bases. Three more states and two matrices were tested on a real quantum computer.
The states $|0\rangle, |1\rangle$ and $|+\rangle$ were given in notation 3. Additional, nontrivial states of the inputs $|b_0\rangle, |b_1\rangle, |b_2\rangle$ are displayed in figure 6.

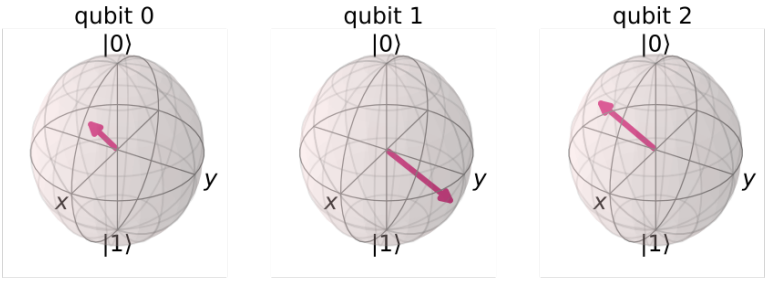

Figure 6: Input states $|b_0\rangle, |b_1\rangle, |b_2\rangle$ for tests of the algorithm

3.1. The first implemented Matrix

Cao et al. proposed a circuit to solve the matrix $A = \frac{1}{2}\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}$ in 2012 [**Cao.2012**].
Shortly after, Pan et al. tested different inputs $|b\rangle$ with this circuit using a "4-qubit nuclear magnetic resonance quantum information processor". The experiment was the first implementation of the HHL algorithm and obtained solutions with over 96 % fidelity [**Pan.2014**]. In 2017, Zheng et al. tested a "four-qubit superconducting quantum processor" using the same matrix but a different circuit design [**Zheng.2017**].

3.1.1. The 2012 Circuit in Qiskit

The matrix A has the eigenvalues $\lambda_0 = 1$ and $\lambda_1 = 2$. Therefore two qubits are enough to encode them precisely as $|01\rangle$ and $|10\rangle$.

A Qiskit implementation of the circuit proposed by Cao et al. [**Cao.2012**] can be seen in figure 7. The initial state $|1\rangle$ is set up by a Pauli X.

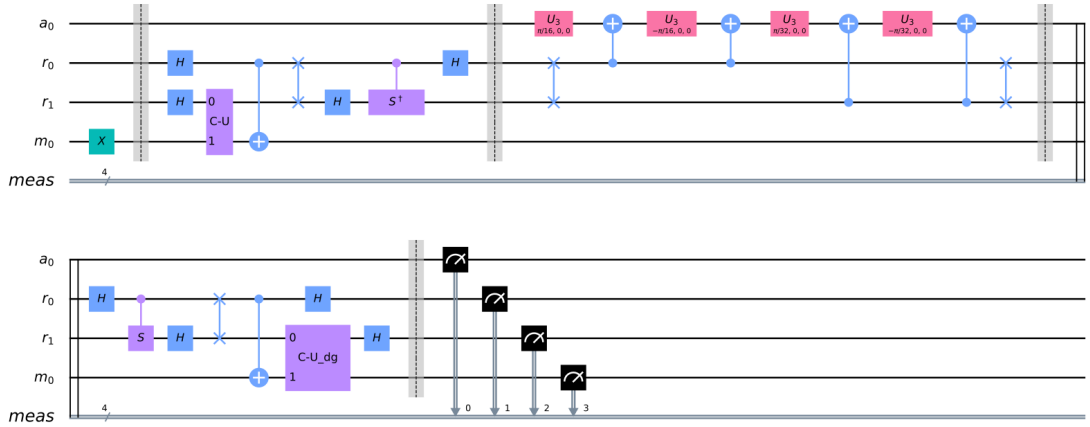

Figure 7: Qiskit circuit implementation of the HHL algorithm for a 2x2 matrix and input state $|1\rangle$

The barriers relate to the steps from section 2.2. The eigenvalue register is $|r_0 r_1\rangle$. After applying the Hadamard gates, r is in superposition. The Hamiltonian simulation of A is done by applying

$$U = e^{iA\frac{\pi}{2}} = \frac{1}{2}\begin{pmatrix} -1+i & -1-i \\ -1-i & -1+i \end{pmatrix} \qquad [\textbf{J.4102018}] \tag{14}$$

conditioned on r, multiple times on m. U is implemented with the *qiskit.extensions.UnitaryGate* class. The first time U is applied if r_1 is in state $|1\rangle$. If r_0 is in state $|1\rangle$, U is applied two times. This gives $U^2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, which is the Pauli X.
The gate-sequence of a $SWAP$, two Hadamards and the $S^\dagger$ gate implements $QFT^\dagger$. The eigenvalue inversion in the register r is done by a simple $SWAP$ gate between r_0 and r_1, which can be seen after the second barrier. Afterwards the controlled $R_y(\tilde{\theta}_j)$ rotations are performed with an approximated angle $\tilde{\theta}_j = \frac{C}{\lambda_j} \approx 2\arcsin(\frac{C}{\lambda_j})$ and $C = \frac{\pi}{2^4}$. See Cao et al.'s work for the details of these parameters [**Cao.2012**]. Qiskit transforms the controlled rotation gates into the equivalent gate-sequence of U_3 and C-X gates.
From now on the implementation was added automatically by the inverse method of the class *QuantumCircuit*. The $SWAP$ prior to the third barrier makes the eigenvalue inversion undone. The lower part of the circuit shows $QPE^\dagger$. This uncomputes the r register.
At the end of the circuit, all qubits are measured in computational basis. Without any decoherence, the r register should always be measured in the state $|00\rangle$. Due to

entanglement, the state of the m register is only the solution if a is measured in the state $|1\rangle$.
Figure 8 shows an example of the outcome for 20000 shots using a simulator.
The m register is the leftmost number in the string on the x-Axis and holds the solution $|x\rangle$ only if the rightmost number is 1. This happened $158 + 345 = 503$ times.
So the probability of getting the solution is around 2.5 %.
This shows that only the minority of data contributes to the solution x. In the appendix A.1.2 one can find the python code to simulate the circuit and draw the output histogram.

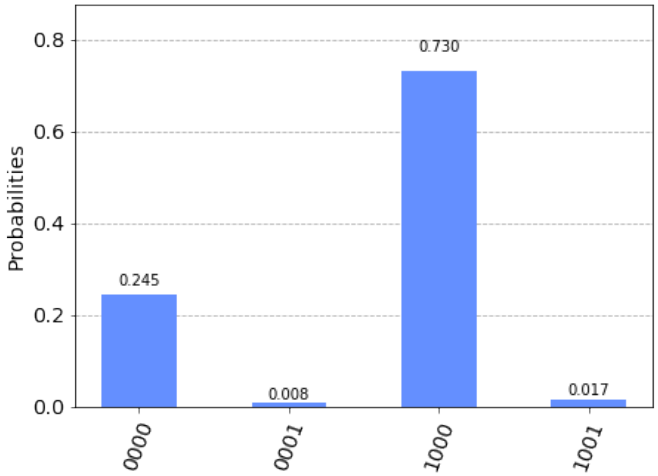

Figure 8: 20000 measurements of the circuit from figure 7 for $\langle Z \rangle_x$ and $|b_1\rangle$

Unfortunately, tests on real devices did not show the solution state. Due to decoherence the circuit results too often in an arbitrary state.
Possible reasons are:

- The custom gate, which was introduced to implement the Hamiltonian Simulation, produces deviation. Qiskit transpiles C-U down to a gate-sequence of C-X and rotation gates. The combination of controlled gates has a high error rate.
- The angles in step 3 are an estimation.
- The connectivity of the qubits in the system is limited. Two qubit gates are noisy.

3.1.2. Results with the QasmSimulator

Table 1 contains the expectation values for the Pauli operators analytically and as an outcome of the repeated simulation of the circuit. Results by the simulator are quite close to the exact calculations. No noise model was added.

The eigenvalues of the Pauli matrices are 1 and -1. The expectation value (def. 1.6) is the average of times we observe $|x\rangle$ in the two eigenstates, multiplied by the corresponding eigenvalue. For example, if we measure $|x\rangle$ with input $|b_1\rangle$ 158 times in the state $|0\rangle$ and 345 times in the state $|1\rangle$, the expectation value is approximately:

$$\langle Z\rangle_x \approx \frac{158 \cdot 1 + 345 \cdot (-1)}{158 + 345} = -0.372$$

Apparently the approximation gets better by using more shots.
The input vectors and solutions are shown in table 2. The theoretical expectation value is the inner product of the solution vector and the matrix product of the observable and the solution vector. Therefore $\langle Z\rangle_x$ for $|b_1\rangle$ is:

$$\langle Z\rangle_x = \langle x|Zx\rangle = x^\dagger Zx = \begin{pmatrix} 0.500 & -0.146 - 0.854i \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0.500 \\ -0.146 + 0.854i \end{pmatrix} = -0.368$$

Qiskit allows measurement in the computational basis only. To observe Pauli X or Y, we need to transform the basis representation of the solution state.
To measure $\langle X\rangle_x$, which means to observe x in X basis, one only needs to add an Hadamard gate at the end of the circuit. To measure $\langle Y\rangle_x$, an $S^\dagger$ gate, followed by a Hadamard, changes the basis representation from Z to Y.
The calculations were done with Pythons *NumPy* package and can be found in the Appendix A.1.1.

Table 1: Expectation values in the 2012 circuit simulation

Observable	$\langle X\rangle_x$		$\langle Y\rangle_x$		$\langle Z\rangle_x$	
Input state $\|b\rangle$	Theory	Simulation	Theory	Simulation	Theory	Simulation
$\|1\rangle$	-0.600	-0.535	0.000	-0.037	-0.800	-0.815
$\|b_0\rangle$	0.186	0.225	0.000	0.033	0.983	0.968
$\|b_1\rangle$	-0.686	-0.659	0.628	0.645	-0.368	-0.399

Table 2: Statevectors and solutions for inputs $|1\rangle$, $|b_0\rangle$ and $|b_1\rangle$

$\|b\rangle$	vector b^T	solution x^T
$\|1\rangle$	$(0, 1)$	$(-0.316, 0.949)$
$\|b_0\rangle$	$(0.924, 0.383)$	$(0.996, 0.093)$
$\|b_1\rangle$	$(0.500, -0.146 + 0.854i)$	$(0.499 - 0.259i, -0.285 + 0.776i)$

3.2. Parametrized Design

Lee et al. introduced a *Hybrid HHL algorithm* to overcome the practical disadvantage of setting up the rotation angles in the eigenvalue inversion step of the original HHL

algorithm. To physically implement the gates, it is necessary to have some information about the matrix A. Lee et al. gain this information by splitting up the circuit. At first, the QPE is applied and the eigenvalues of A are measured. Having such information, a classical computer calculates the rotation angles. Now a "Reduced HHL" algorithm is applied to finally get the solution state [**Lee.2019**].
In this section, Lee et al.'s circuit design is used without the implementation of eigenvalue measurement [3]. The rotation angles are precalculated, proposing the information is already available. Subsection 3.2.1 focuses on the circuit of the QPE and eigenvalue inversion part. Tests of the complete circuit on real quantum hardware are subject of subsection 3.2.2.

3.2.1. Tailored Circuit

The tested matrices are $A_\lambda = \begin{pmatrix} \frac{1}{2} & \lambda - \frac{1}{2} \\ \lambda - \frac{1}{2} & \frac{1}{2} \end{pmatrix}$ for $\lambda \in \{\frac{1}{4}, \frac{1}{2}\}$ [**Lee.2019**].
Notice that $A_{\frac{1}{2}} = \frac{1}{2}\mathbb{I}_2$.
The parameter λ alters the rotation angles in the Hamiltonian simulation of the QPE and the eigenvalue inversion part.
The Qiskit implementation of the QPE can be seen in figure 9. The applied unitary is $U_\lambda = e^{2\pi i A_\lambda}$. It starts with the creation of a superposition in the eigenvalue register. For the controlled Hamiltonian simulation $C\text{-}U_\lambda^f$ is applied with f being 1 for control qubit r_1 and 2 for r_0.
Both of the controlled gates are implemented as the following gate-sequence:

$$H \quad R_z(2\pi\lambda f) \quad C\text{-}X \quad R_z(2\pi\lambda f) \quad C\text{-}X \quad R_z(-4\pi\lambda f) \quad H$$

The parameters λ and f determine the rotation angles. Such gates can be implemented on a real quantum device.

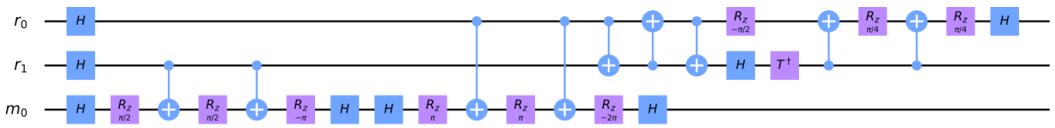

Figure 9: QPE circuit for $\lambda = \frac{1}{4}$ [**Lee.2019**]

In the original algorithm one needs to know the normalization constant C to setup the rotation angles in step 3. Analytically the constant is $C_\lambda = \frac{1}{\sqrt{b^\dagger (A_\lambda^{-1})^2 b}}$.
The first part of the *Hybrid HHL algorithm* gives an approximation for this constant. Here, we assume that the value C_λ is already known.

[3] The author acknowledges the work of Yonghae Lee, Jaewoo Joo & Soojoon Lee under the Creative Commons License: http://creativecommons.org/licenses/by/4.0/

Figure 10 display's the eigenvalue inversion part . Since only the reduced part of the Hybrid algorithm is used, r_1 has not to be connected to the ancilla. It is sufficient to apply a single qubit gate $R_y(\theta_1)$ and a controlled rotation gate $R_y(\theta_3 - \theta_1)$, conditioned on r_0. This increases the accuracy on a real device. The angles are precalculated by the formula:

$$\theta_n = 2\arccos(\sqrt{1 - \frac{C_\lambda^2}{n^2}}) \qquad \textbf{[Lee.2019]} \tag{15}$$

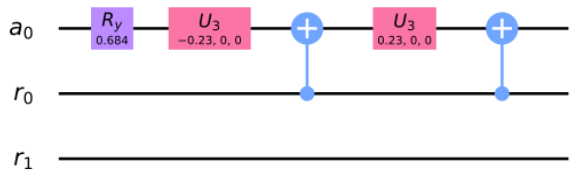

Figure 10: Eigenvalue inversion circuit for $\lambda = \frac{1}{4}$ **[Lee.2019]**

3.2.2. Results by the IBM Q Rome

Lee et al. tested their circuit on the *IBM Q X4* with input state $|0\rangle$ **[Lee.2019]**. In the following, results of tests on the IBM Q Rome are presented. Input vectors and the corresponding solution states (in computational basis) for $\lambda = \frac{1}{4}$ are given in table 3. Since $A_{\frac{1}{2}}$ is a multiple of the identity matrix, the solution states are always the input vectors for $\lambda = \frac{1}{2}$.

Table 3: Statevectors and solutions for inputs $|0\rangle, |+\rangle$ and $|b_2\rangle$

λ	$\|b\rangle$	vector b^T	solution x^T
$\frac{1}{4}$	$\|0\rangle$	$(1, 0)$	$(0.894, 0.447)$
	$\|+\rangle$	$\frac{1}{\sqrt{2}}(1, 1)$	$\frac{1}{\sqrt{2}}(1, 1)$
	$\|b_2\rangle$	$(0.924, 0.271 - 0.271i)$	$(0.801 - 0.102i, 0.554 - 0.205i)$

Table 4 shows the expectation values of X. Without any noise the simulation is close to the true values. The IBM Q Rome has a variance in all cases. Half of the times the expectation value gets close to the true value.
Internally, Qiskit transpiles the circuit to fit the qubit connectivity and physically implemented gates on the IBM Q Rome. The complete transpiled circuit is appended A.2.2. Some additional $SWAP$ gates were added automatically to implement missing connections between the qubits.

Table 4: Results by the QasmSimulator without noise and by the IBM Q Rome

λ	$\|b\rangle$	$\langle X\rangle_x$	$\langle X\rangle_x$ QasmSimulator	$\langle X\rangle_x$ IBM Q Rome
$\frac{1}{4}$	$\|0\rangle$	0.800	0.788	0.567
	$\|+\rangle$	1.000	1.000	0.960
	$\|b_2\rangle$	0.929	0.936	0.763
$\frac{1}{2}$	$\|0\rangle$	0.000	-0.020	-0.062
	$\|+\rangle$	1.000	1.000	0.840
	$\|b_2\rangle$	0.500	0.523	0.546

The QasmSimulator has the feature to simulate noise in a quantum system. Results of the simulation with the noise model of the IBM Q Rome can be seen in figure 11. The simulation matches the outcome of the quantum hardware well.
Figure 12 illustrates the noise in the system. Due to decoherence, r is not always $|00\rangle$ at the end of the circuit. Such outcomes and the cases with the ancilla being 0 do not contribute to the solution x.

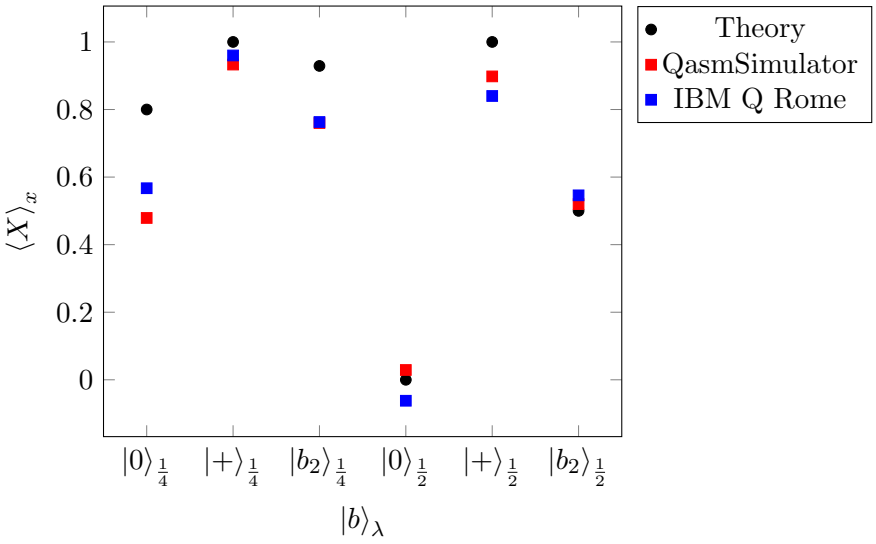

Figure 11: Expectation values by measuring the circuit outcome for different input states with the IBMQRome and the simulator with the noise model of the IBMQRome

3.3. Conclusion

In the last two sections we saw concrete implementations of a quantum algorithm for solving linear equations. Tests with the QasmSimulator have illustrated that the algorithm works for small dimensions. The tested matrices are quite easy and could be inverted by hand. However, given more qubits, the circuit design is scalable.

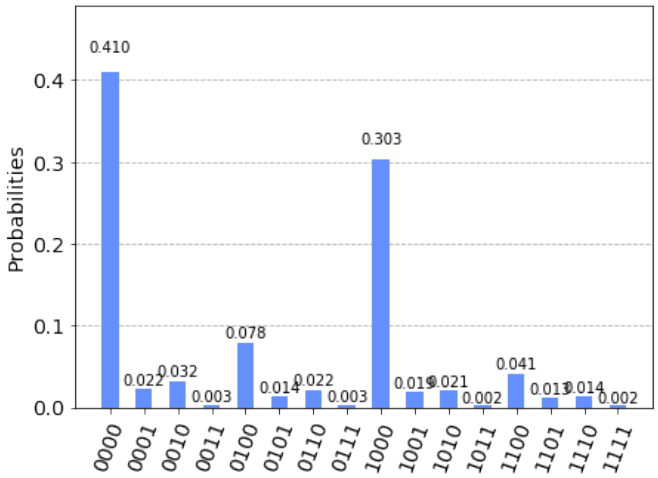

Figure 12: Measurement results for $\lambda = \frac{1}{2}$, input $|0\rangle$ and $\langle X \rangle_x$ by using 4096 shots on the IBM Q Rome. The leftmost number is the measurement result of m. The eigenvalue register r is represented by the two numbers in the middle of the string. The ancilla is the rightmost number.

The probability distributions reveal the probabilistic nature of quantum computation. Resulting states are not always the same. Measurements must be repeated to get the desired outcome.
A general approach to deal with arbitrary matrices is indicated by the 2012 circuit design. Nevertheless, Hamiltonian simulation and eigenvalue inversion still needs to be implemented efficiently.
The parametrized design works for A_λ both on the simulator and a real quantum computer. Due to a high dependency on λ a generalization would be difficult.
IBM Q Rome has five qubits of which four were used for the circuit. The connectivity between the qubits is limited but enough to provide some useful results. In every test the expectation value lies in the vicinity of the theoretical value. In half of the cases even close. Figure 12 shows decoherence occurs quite often during the circuit.
Even though the quantum hardware has a high error rate, the software is able to simulate noise. One could be able to correct errors.
It is still a long way for quantum computers to deal with real life applications. Due to decoherence it is still unclear if this can ever be achieved efficiently. However, if, linear equations should not be an obstacle.
IBM gives everyone the opportunity to get familiar with quantum computation. Making the machinery available to the public accelerates innovations and the education of young scientists.
With Qiskit the implementation of a circuit for a given algorithm is intuitive and straightforward. The framework is well described and under high development.

A. Appendix

A.1. 2012 Circuit

A.1.1. NumPy Calculations

[1]:
```
import numpy as np
pauli_X=np.array([[complex(0,0),complex(1,0)],
        [complex(1,0),complex(0,0)]])
pauli_Y=np.array([[complex(0,0),complex(0,-1)],
        [complex(0,1),complex(0,0)]])
pauli_Z=np.array([[complex(1,0),complex(0,0)],
        [complex(0,0),complex(-1,0)]])
operators={"X":pauli_X,"Y":pauli_Y,"Z":pauli_Z}
```

[2]:
```
def getAnalyticalObservation(A,b,M=""):
        A_inv=np.linalg.inv(A)
        x=A_inv.dot(b)
        x=x/np.linalg.norm(x,2)
        if M in operators.keys():
                operator=operators[M]
                return np.vdot(operator.dot(x),x)
        return x
```

[3]:
```
A=np.array([[3/2,1/2],[1/2,3/2]])
# Get input states b:
inputs=[]
inputs.append(('1',np.array([0, 1])))
inputs.append(('b0',np.array([0.92388,0.382683])))
inputs.append(('b1',np.array([0.5,complex(-0.146447,0.853553)])))

results={}
for c,b in inputs:
        x=getAnalyticalObservation(A,b) #solution in computational␣
 ↪basis
for M in ["X","Y","Z"]:
        M_X=getAnalyticalObservation(A,b,M)
        results[(c,M)]=(x,M_X)
```

[4]:
```
#print analytical results:
for c,M in results.keys():
        x,M_X= results[(c,M)]
        print("Base ",M," with input ",c," has expectation value ",'{:
 ↪f}'.format(M_X.real)," and solution ",x)
```

A.1.2. QasmSimulator Qiskit Script

[1]:
```
from qiskit import QuantumCircuit, QuantumRegister, execute, Aer
from qiskit.extensions import UnitaryGate, SdgGate
from qiskit.visualization import plot_histogram
import numpy as np
from math import pi
# created with qiskit meta-package version 0.18.3
```

[2]:
```
def createStatePreparationCircuit(qb,c):
        statePreparationCircuit=QuantumCircuit(qb)
        if c=="1":
                statePreparationCircuit.x(0)
        elif c=="b0":
                statePreparationCircuit.ry(pi/4,0)
        elif c=="b1":
                statePreparationCircuit.rx(-pi/4,0)
                statePreparationCircuit.ry(-3*pi/2+pi/4,0)
                statePreparationCircuit.rz(pi/4,0)
        return statePreparationCircuit
```

[3]:
```
def createCircuit(r,c,M):
        qa=QuantumRegister(1,name="a") #store ancilla
        qev=QuantumRegister(2,name="r") #store both eigenvalues
        qb=QuantumRegister(1,name="m") #store |b>
        qc=QuantumCircuit(qa,qev,qb)
        qc=qc+createStatePreparationCircuit(qb,c)
        qc.barrier();

        phaseEstimationCircuit=QuantumCircuit(qev,qb)
        phaseEstimationCircuit.h(qev[0])
        phaseEstimationCircuit.h(qev[1])
        # add Hamiltonian Simulation

        #e^{i A pi/2} with control qev[1] and target qb[0]
        hamiltonianOp=1/2*np.array([[complex(-1,1),complex(-1,-1)],
                [complex(-1,-1),complex(-1,1)]])
        hamiltonianGate=UnitaryGate(hamiltonianOp)
        phaseEstimationCircuit.append(hamiltonianGate.
↪control(1),[qev[1],qb[0]])
        #e^{i pi A } with control qev[0] and target qb[0]
        phaseEstimationCircuit.cx(qev[0],qb[0]) #simply an X Gate

        phaseEstimationCircuit.swap(qev[0],qev[1])
```

```
        phaseEstimationCircuit.h(qev[1])
        phaseEstimationCircuit.append(SdgGate().control(1),qev)
        phaseEstimationCircuit.h(qev[0])
        qc=qc+phaseEstimationCircuit

        qc.barrier();
        qc.swap(1,2)
        qc.cry(2*pi/2**r,1,0)
        qc.cry(pi/2**r,2,0);

        #invert circuit before Ry
        qc.swap(1,2)
        qc.barrier();
        qc=qc+phaseEstimationCircuit.inverse()
        if M=="X":
                qc.h(qb[0])
        elif M=="Y":
                qc.sdg(qb[0])
                qc.h(qb[0])
        qc.measure_all()
        return qc
```

[4]:
```
def measureExpectationValue(qc):
        counts=execute(qc,Aer.
 ↪get_backend('qasm_simulator'),shots=20000).result().get_counts()
        if "0001" not in counts:
                a_square=0
        else:
                a_square=counts["0001"]
        if "1001" not in counts:
                b_square=0
        else:
                b_square=counts["1001"]
        # eigenvalues are 1 and -1
        expectationValue=(1*a_square+(-1)*b_square)/␣
 ↪(a_square+b_square)
        return expectationValue
```

[5]:
```
#setting parameter r as described in Figure 3 of the 2012 paper by Cao␣
 ↪et al.:
r=4
qc=createCircuit(r,"b1","Z")
```

```
counts=execute(qc,Aer.get_backend('qasm_simulator'),shots=20000).
 ↪result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0001': 158, '1000': 14600, '1001': 345, '0000': 4897}
```

[6]:
```
results={}
for M in ["X","Y","Z"]:
        for c in ["1","b0","b1"]:
                qc=createCircuit(r,c,M)
                results[(c,M)]=measureExpectationValue(qc)

for c,M in results.keys():
        M_X= results[(c,M)]
        print("Observing ",M," with input ",c," has expectation value␣
 ↪",'{:f}'.format(M_X.real))
```

A.2. Parametrized Design

A.2.1. NumPy Calculations

[1]:
```
import numpy as np
from math import sqrt
hadamard=np.array([[complex(1/sqrt(2),0),complex(1/sqrt(2),0)],
        [complex(1/sqrt(2),0),complex(-1/sqrt(2),0)]])
pauli_X=np.array([[complex(0,0),complex(1,0)],
        [complex(1,0),complex(0,0)]])
```

[2]:
```
# Get input states b:
inputs=[]
inputs.append(('0',np.array([1, 0])))
inputs.append(('+',hadamard.dot(np.array([1, 0]))))
inputs.append(('b2',np.array([0.92388,complex(0.270598,-0.270598)])))
results={}
for lamb in [1/4,1/2]:
        A=np.array([[1/2,lamb-1/2],[lamb-1/2,1/2]])
        A_inv=np.linalg.inv(A)
        for c, b in inputs:
                x=A_inv.dot(b)
                x=x/np.linalg.norm(x,2)
                M_X=np.vdot(pauli_X.dot(x),x)
                # expectation value of x in X basis
```

```
                results[(lamb,c)]=(x,M_X)
```

[3]:
```
#print analytical results:
for l,b in results.keys():
        x,M_X= results[(l,b)]
        print("lambda ",l," with input ",b," has expectation value␣
 ↪",'{:f}'.format(M_X.real)," and solution ",x)
```

A.2.2. IBM Q Rome Qiskit Script

[1]:
```
from qiskit import QuantumCircuit, IBMQ, execute, QuantumRegister,␣
 ↪transpile
from qiskit.visualization import plot_histogram
from qiskit.tools.monitor import job_monitor
import numpy as np
from math import sqrt, pi
# created with qiskit meta-package version 0.18.3
hadamard=np.array([[complex(1/sqrt(2),0),complex(1/sqrt(2),0)],
        [complex(1/sqrt(2),0),complex(-1/sqrt(2),0)]])
pauli_X=np.array([[complex(0,0),complex(1,0)],
        [complex(1,0),complex(0,0)]])
```

[2]:
```
def preCalculation(lamb,b):
        A=np.array([[1/2,lamb-1/2],[lamb-1/2,1/2]])
        A_inv=np.linalg.inv(A)
        x_unnorm=A_inv.dot(b)
        norm=np.linalg.norm(x_unnorm,2)
        c_lamb=1/norm
        return c_lamb
```

[3]:
```
def createStatePreparationCircuit(qb,c):
        statePreparationCircuit=QuantumCircuit(qb)
        if c=="0":
                statePreparationCircuit.id(0)
        elif c=="+":
                statePreparationCircuit.h(0)
        elif c=="b2":
                statePreparationCircuit.ry(pi/4,0)
                statePreparationCircuit.rz(-pi/4,0)
        return statePreparationCircuit
```

[4]:
```
def createPhaseEstimationCircuit(qb,qev,lamb):
        phaseEstimationCircuit=QuantumCircuit(qev,qb)
```

```
        #superposition of eigenvalue register
        phaseEstimationCircuit.h(qev[0])
        phaseEstimationCircuit.h(qev[1])
        #apply Controlled Hamiltonian U _lamb  with qev[1] as control
        m=1
        phaseEstimationCircuit.h(qb[0])
        phaseEstimationCircuit.rz(2*pi*lamb*m,qb[0])
        phaseEstimationCircuit.cx(qev[1],qb[0])
        phaseEstimationCircuit.rz(2*pi*lamb*m,qb[0])
        phaseEstimationCircuit.cx(qev[1],qb[0])
        phaseEstimationCircuit.rz(-4*pi*lamb*m,qb[0])
        phaseEstimationCircuit.h(qb[0])
        #apply Controlled Hamiltonian U _lamb ^2 with qev[0] as control
        m=2
        phaseEstimationCircuit.h(qb[0]) #inverse to step before
        phaseEstimationCircuit.rz(2*pi*lamb*m,qb[0])
        phaseEstimationCircuit.cx(qev[0],qb[0])
        phaseEstimationCircuit.rz(2*pi*lamb*m,qb[0])
        phaseEstimationCircuit.cx(qev[0],qb[0])
        phaseEstimationCircuit.rz(-4*pi*lamb*m,qb[0])
        phaseEstimationCircuit.h(qb[0])
        #apply inverse Fourier transform
        phaseEstimationCircuit.cx(qev[0],qev[1])
        phaseEstimationCircuit.cx(qev[1],qev[0]) #this was a swap
        phaseEstimationCircuit.cx(qev[0],qev[1])
        phaseEstimationCircuit.rz(-pi/2,qev[0])
        phaseEstimationCircuit.h(qev[1])
        phaseEstimationCircuit.tdg(qev[1])
        phaseEstimationCircuit.cx(qev[1],qev[0])
        phaseEstimationCircuit.rz(pi/4,qev[0])
        phaseEstimationCircuit.cx(qev[1],qev[0])
        phaseEstimationCircuit.rz(pi/4,qev[0])
        phaseEstimationCircuit.h(qev[0])
        return phaseEstimationCircuit
```

[5]:
```
def createEigenvalueInversionCircuit(qa,qev,c_lamb):
        theta_1=2*np.arccos(sqrt(1-(c_lamb/1)**2))
        theta_2=2*np.arccos(sqrt(1-(c_lamb/2)**2))
        theta_3=2*np.arccos(sqrt(1-(c_lamb/3)**2))
        aqeCircuit=QuantumCircuit(qa,qev)
        aqeCircuit.ry(theta_1,qa[0])
        aqeCircuit.cry(theta_3-theta_1,qev[0],qa[0])
        return aqeCircuit
```

[6]:
```
def testOnRealDevice(qc,backend):
        job = execute(qc, backend=backend,␣
 ↪shots=4096,optimization_level=3)
        job_monitor(job, interval = 2)
        result= job.result()
        counts = result.get_counts()
        if "0001" not in counts:
                c_plus_square=0
        else:
                c_plus_square=counts["0001"]
        if "1001" not in counts:
                c_minus_square=0
        else:
                c_minus_square=counts["1001"]
        m=c_plus_square+c_minus_square
        average=(c_plus_square-c_minus_square)/m
        print("<x|X|x>=",average)
        return counts
```

[7]:
```
# Get input states |b>:
inputs={'0':np.array([1, 0]),'+':hadamard.dot(np.array([1, 0])),'b2':
 ↪np.array([0.92388,complex(0.270598,-0.270598)])}
# Setting up parameters
lamb=1/2
c='0'
b=inputs[c]
c_lamb=preCalculation(lamb,b)
backendName="ibmq_rome"
IBMQ.load_account()
provider=IBMQ.get_provider(hub='ibm-q')
backend=provider.get_backend(backendName)
```

[8]:
```
qa=QuantumRegister(1,name="a") #store ancilla
qev=QuantumRegister(2,name="r") #store both eigenvalues
qb=QuantumRegister(1,name="m") #store |b>
qc=QuantumCircuit(qa,qev,qb)
qc=qc+createStatePreparationCircuit(qb,c)
qc.barrier()
phaseEstimationCircuit=createPhaseEstimationCircuit(qb,qev,lamb)
qc=qc+phaseEstimationCircuit
qc.barrier();
aqeCircuit=createEigenvalueInversionCircuit(qa,qev,c_lamb)
qc=qc+aqeCircuit
```

```
qc.barrier();
qc=qc+phaseEstimationCircuit.inverse()
qc.barrier();
#measure |x> in X basis
qc.h(qb[0])
qc.measure_all()
```

[9]:
```
counts = testOnRealDevice(qc,backend)
plot_histogram(counts)
```

```
Job Status: job has successfully run
<x|X|x>= 0.08235294117647059
```

[10]:
```
# this is the circuit that was actually used:
qc_trans=transpile(qc,backend)
qc_trans.draw('mpl')
```

[10]: