Implementations of Quantum Algorithms for Solving Linear Systems

Sigurdur Ag. Sigurdsson

MSc Thesis

TU Delft

TU Berlin

Implementations of Quantum Algorithms for Solving Linear Systems

by

Sigurdur Ag. Sigurdsson

To obtain the degree of Master of Science
at TU Delft & TU Berlin as part of the COSSE program,
to be defended publicly on January 26th, 2021.

Student number TU Delft:	5162599	
Matrikelnummer TU Berlin:	405725	
Project duration:	September 1st 2019 – January 5th 2021	
Thesis committee:	Dr. M. Möller,	TU Delft, supervisor
	Prof. dr. ir. C. Vuik,	TU Delft, responsible professor
	Dr. D. de Laat	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

TU berlin

Contents

List of Figures

Preface

I am not a physicist nor am I a computer scientist, I studied mechanical engineering at Reykjavik University and worked in mechanical design before starting the master's program that led to this thesis. Therefore my interest in the topic of quantum computers does not come from theorizing better algorithms, but from getting better ways to solve practical problems. For the introduction of this thesis, I will assume that the reader has a good understanding of linear algebra and the basics of computer science. That is why details of the topics of quantum physics and quantum computation are out of the scope of this thesis. However, if the reader is interested in these topics, I recommend the lecture series from Richard Feynman, published online by the California Institute of Technology [14], as well as the book Quantum Computing: a gentle introduction by Eleanor Rieffel and Wolfgang Polak [33].
Front cover image courtesy of IBM image gallery [7].

In this thesis, I will make a comparison of two quantum algorithms for solving systems of linear equations. The two approaches are tested using both simulations of quantum processes and simulations of noisy intermediary-scale quantum computers, i.e., NISQ [31]. Both methods are tested on the same problems to compare sensitivity and runtime efficiency, varying the tests along with different levels of condition numbers, sparsity and regularity.

The first of the two methods compared here is a method introduced by Harrow, Hassim and Lloyd referred to as the HHL method, after the author's initials [17]. The second method is by a research team at Los Alamos National Laboratory called the variational quantum linear solver, or VQLS for short [5]. The first method, HHL, was the original discovery of solving linear systems using quantum computing methods and so has served as the backdrop and benchmark for other algorithms doing the same thing. The HHL method showed that an advantage can be had when using quantum computing to solve linear systems, in fact they show that this quantum method can have exponential speed up over the commonly used classical method.

Quantum algorithms themselves are rather new and as the reader may not be very familiar with them I offer a brief introduction in the next chapter along with the physics behind it and the notation, which will be used throughout this work. In short, quantum algorithms seek to use the peculiar physics and mathematics tied to the behavior of quantum particles. A quantum algorithm does this by leveraging the extra information a quantum superposition provides and by using this information it can in certain problems arrive at an answer in fewer steps than it is possible by classical methods.

These novel algorithms are then realized using a quantum device known as a quantum computer. A quantum computer is simply a computer that runs on the principles of quantum mechanics. These machines have been a long time in the making as it is notoriously tricky to manipulate elements on the quantum scale. Today, we have reached a point where these machines are starting to be possible, but the current generation of hardware only glimpses at what is feasible and does not reach the ideals of the theory. This is why the current generation of quantum computers have been called NISQ devices as they exhibit noisy behavior in the measurement of the computation and are small in scale compared to modern classical computers.

In this thesis, I want to explore the functionality of the quantum linear solvers, specifically HHL and VQLS, using the IBM Qiskit platform. I will test these implementations in both perfect simulations and noisy conditions. The experiments will focus on matrices that test varying sparsity and condition numbers. The general aim is to answer the following questions.

1. Which approaches exist to solve linear systems on quantum computers?
2. How do the implementations of these approaches scale with problems?
3. Can these approaches get to quantum advantage in the near term?
4. Can one use these approaches within a larger algorithm?

Sigurdur Ag. Sigurdsson
Oxford, January 2021

1 Introduction

This chapter gives an overview of the background necessary for understanding the implementations and algorithms covered. We will go over the shift in perspective needed to approach quantum computation and how information as a core physical concept is key to that shift. Then we explore what is needed to make any computation through the theorem of universal computers and how a quantum system can achieve this universality. Before we look at the quantum system we will take a brief look at the mathematics behind quantum mechanics to understand the underlying parts of a quantum computer. We conclude with an example of a quantum algorithm to demonstrate the concepts explored previously and show how a quantum algorithm can be computationally faster than a classical one.

1.1. Computational Perspective

In 1883 Antoni Gaudì was designing his masterpiece La Sagrada Familia. During the design process, Gaudì envisioned arches and columns so complex that there was no simple way of calculating the load of the structure. Today we can set this up on a computer and analyze complex structures with, e.g. the Finite Element method, then compute the model and quickly resolve any issues in the design. As Gaudì could not wait for the computer to be invented he had to devise another way of calculating his dream structure.

So how did Gaudì solve this? He was using an old building style where the entire structure is kept under compression, therefore there is no need for steel for structural support, and only his chosen material of Catalonian sandstone was used. He devised a way of hanging ropes from the ceiling and connecting them in such a way that they formed the building he wanted. He then hung weights on the ropes where they would be supporting the roof of the building. As he hung the ropes and weights the calculations for the archways and columns were instantly done as the rope itself solves the ideal arch, this was shown by Robert Hooke [20]. This tension model using ropes and weights under gravity becomes a model of the building under compression when turned upside down. So this completes the structural calculations needed for the building. This way Gaudì was able to measure the archways on his model and scale them up to the real size of the model. A replica of his model stands in the Sagrada Familia today, cf. Figure 1.1, [2].

This story is to illustrate the way an analog computer functions and to set the mindset that a complex computation does not necessarily need the machines which we so often use today. In a more abstract notion of computation, the information contained in numbers is like currency, i.e., numbers are the medium we use to exchange information between systems in the same way we exchange currency for goods and services. In Gaudì's system a hanging rope was used to represent an archway and a sandbag the weight it would need to support, then gravity did the calculations for him. With measurements of the model he then turned it into numbers and scaled them properly to represent the real version. There is something canonical in the model and how we move it into the real world, this we call *information*.

Figure 1.1: A replica of Gaudì's rope and sandbag model from La Sagrada Familia [22].

1.2. Information

Information as a fundamental part of Information theory came about in the late 1940s. It was first introduced by Claude Shannon [35], mainly in relation to communication and computation. It has now grown from its origin into a fundamental part of physics as we can say that every microscopic state of a physical system can be characterized by how much information is needed to define that microscopic state. Perhaps it was this definition that led the physicist John Wheeler say "Information is the most fundamental building block of reality" [21].

Shannon's **information theory** focuses on information as an abstraction and a distinction between possible alternative messages, not the meaning of the message. This means that a message represents one out of many alternative possible messages and the information of the message is to be able to reliably distinguish between the alternatives. Shannon theorized the smallest possible level of distinction of information was a binary one, 0 and 1, and he called this a **bit**. To then capture the information in a message we can try to think of it as how many yes or no questions you would need to ask to determine the message, this is also known as the message entropy. With bits, we can build and send any message by stringing them together. We can also create new messages by transforming the information and performing calculations.

Classical computers use bits as the basis for their calculations and are representing information by strings of bits. Calculations on the bits themselves are performed with logic gates. The informational value of an 8-bit number is straightforward, it is simply 8 bits. This is not true for quantum information. As we will discuss in later chapters, quantum information is represented by a quantum bit, i.e., a qubit, and each qubit can represent more than one bit of information at a time using the natural effects of quantum mechanics. As opposed to the classical case where an 8-bit string can have 8 bits of information an 8-qubit string can contain 2^8 bits of information.

1.3. Universal Computers

Alan Turing, one of the fathers of modern computing, reasoned that all sufficiently sophisticated computers are equivalent. That is once a computer is able to simulate another computer all computations done on the second one can be done on the first. In this way, Turing tried to capture the concept of a universal computer, i.e., that is a computer that is equivalent to all others. Turing set to design the simplest possible computer with this property and this came to be known as the Turing machine.

Turing's model consists of a machine that had infinite tape to read from and write to, a head that executes reading and writing. The Turing machine has finitely many states (memory), and a transition function δ. The transition function determines what the machine will do at time t, given the current state it is in and what is read at time t. The output of the transition function δ is a triple consisting of what will be written on the tape, what direction the head will move, and what will be the state of the machine

at time $t + 1$. It is a common belief that such a simple computer represents a universal computer, this is phrased as the Church-Turing thesis [39].

Church-Turing Thesis [2] A Turing machine can compute any function computable by a reasonable physical device.

The thesis then says that all complete computers are the same. However, this only means that they can do the same calculations, but the important factor when doing computation is time, i.e., how long does it take to finish the calculation. This is called the time complexity of an algorithm and we categorize problems according to how the computation time scales with the input size of the problem. In computational theory, this is often analyzed as the number of operations a computer must perform to do the calculation. Simple problems such as addition and multiplication have polynomial complexity, i.e., they belong to the class P. A problem is in the class P if there exists an algorithm which takes at most a polynomial number of steps to solve the problem with respect to the input size. But for quite some problems we do not know whether a polynomial algorithm exists. Problems for which, given an answer, we can verify whether the given answer is a solution in polynomial time, belong to the nondeterministic polynomial class, i.e., the class NP. For researchers, the latter class is the most interesting as even though a problem is classed as NP now it does not prove that there can not be a polynomial algorithm just that we do not have one yet.

Quantum computers have been shown to be able to solve some of these class NP problems within a (deterministic) polynomial time and so have broken a barrier thought by many to be impregnable. They do so by utilizing quantum information and by forming a universal computer out of quantum systems. To be clear, a quantum computer can not calculate anything more than classical computers can. That is, any Turing machine can calculate *any* problem and they are just Turing machines like any classical computer. Their benefit is in re-framing problems so that they can be solved faster than using classical physics. Like Gaudì's ropes, they are just better suited to some types of problems than others. Before we go into more details about how quantum computers work we need to familiarize ourselves with the notations and conventions of quantum mechanics.

1.4. Quantum Mechanics

Quantum mechanics is a theory of physics that was set out in the early 20th century to explain the behavior of matter on the smallest scales. Out of these developments came the essential mathematical framework used to describe this quantum world. Although this branch of physics has a notoriety for being incomprehensible, that perception applies more to the structure of the fundamental particles that make up the quantum world and not the mathematics behind it. Luckily we only require the mathematic side to understand the topics covered in this thesis. These concepts mainly depend on a good understating of linear algebra and calculus, nothing more.

Even though our main focus is on the mathematic side of quantum mechanics let us review some of the basic postulates of quantum mechanics. This is a good way to grasp the notation and vocabulary used with quantum mechanics. All this can be found in any good textbook on quantum mechanics or quantum computing, e.g. *Quantum Computation and Quantum Information* by Nielsen and Chuang [27].
Starting with the postulate of the vector spaces of quantum mechanics;

Postulate I [27] Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbert space which is also a complete space) known as the state space of the system. The system is completely described by its state vector, which is a unit vector in the system's state space.

Postulate one is why linear algebra is so prevalent in quantum, as each quantum state can be completely described by a vector and so the common rules of linear algebra apply when we start manipulating that vector,i.e. that state. Next, we look at how our quantum system evolves in time, as we

look at the second postulate.

Postulate II [27] The evolution of a closed quantum system is described by a unitary transformation. That is, the state $|\phi\rangle$ of the system at time t_1 is related to the $|\phi\rangle'$ state of the system at time t_2 by a unitary operator U which depends only on the times t_1 and t_2,

$$|\psi\rangle' = U\,|\psi\rangle\ .$$

Secondly, the time evolution of a closed quantum system is described by the Schrödinger equation

$$i\hbar\frac{d}{dt}\,|\psi(t)\rangle = \mathcal{H}\,|\psi(t)\rangle\ . \tag{1.1}$$

In this equation, $\hbar$ is a physical constant known as Planck's constant whose value must be experimentally determined. The exact value is not important to us. In practice, it is common to absorb the factor $\hbar$ into $\mathcal{H}$, effectively setting $\hbar = 1$. $\mathcal{H}$ is a fixed Hermitian operator known as the Hamiltonian of the closed system.

With this, we know that any unitary operator U applied to a quantum state can describe a new quantum state, but that the natural behavior of a quantum system has to follow the Schrödinger equation. In the next postulate, we look at measurements, i.e. the interaction in which we observe the quantum state and in which we break open the system, meaning that afterward we can no longer consider the system a closed quantum system.

Postulate III [27] Quantum measurement is a collection of measurement operators M_m, where m is the measurement outcome that may occur; these operators act on the state space of the system being measured. The probability of measurement outcome m is defined as follows,

$$\mathbb{P}(m) = \langle\psi|M_m^* M_m\,|\psi\rangle\,. \tag{1.2}$$

This in conjunction with the completeness equation,

$$\sum_m M_m^* M_m = I, \tag{1.3}$$

forces the measurement operators to sum up to one, and so do the probabilities then. Then the state of the system after measurement can be describe thus,

$$\frac{M_m\,|\psi\rangle}{\sqrt{\langle\psi|M_m^* M_m\,|\psi\rangle}}. \tag{1.4}$$

Linear algebra is the mathematical language used to describe quantum systems. A base set of vectors is any set of vectors such that any vector in the span of the set can be written as a linear combination of the base, so with a base $\{b_0, b_1\}$ any vector $b = \alpha_0 b_0 + \alpha_1 b_1$, $\alpha_0, \alpha_1 \in \mathbb{C}$. For a quantum base set we want to span a Hilbert space and so we describe each state using a two dimensional vector of complex numbers in a Hilbert space, $|\theta\rangle \in \mathbb{C}^2$, the notation $|\theta\rangle$ is called a ket of θ and stands for

$$|\theta\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}, \quad \langle\theta| = \begin{bmatrix}\alpha^* & \beta^*\end{bmatrix},$$

where α^* is the conjugate of α, and similarly for β^*. This notation is commonly used in quantum physics and is the one we will use for the remainder of this work. It is a compact notation that makes it easy to write common operations such as inner product $\langle\phi|\psi\rangle$. A common base to choose is $\{|0\rangle, |1\rangle\}$ where,

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}.$$

An important operation using the bra-ket notation is the inner product. Here we take two vectors as

input and produce a complex number as an output. The notation for the inner product of a state vector with itself is $\langle\phi|\phi\rangle = \phi^* \cdot \phi$. Together with regular vector multiplication and inner products, we also use the operation of tensor products on quantum systems. With it we connect two qubits in an operation, we will explain more on this in the next chapter. An example of a tensor product on two qubit vectors would be

$$|\theta\rangle \otimes |\psi\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix} \otimes \begin{bmatrix}\gamma\\ \delta\end{bmatrix} = \begin{bmatrix}\alpha\gamma\\ \alpha\delta\\ \beta\gamma\\ \beta\delta\end{bmatrix}.$$

As a short-hand notation we often write $|\phi\psi\rangle$ as the tensor product of $|\phi\rangle$ and $|\psi\rangle$. These operations are needed for the last postulate.

Postulate IV The state space of a composite quantum system is the tensor product of the state spaces of the component physical systems.

Further common concepts from linear algebra which are used are unitary matrices, linear independence, and bases of vector spaces. These use a common notation and so should be familiar to anyone who has studied them in any other context. These concepts will not be explored in depth here but we will come back to them when we introduce quantum algorithms and how they function.

1.5. Quantum Computers

As discussed earlier, quantum computers can solve some classically hard problems faster, which is due to the way quantum computers represent information, i.e. in the form of a qubit. In a quantum system information is stored in a fundamental property of the particle, this is the qubit and it can be represented e.g. by the spin of an electron. The spin can only be measured in two values, spin-up or spin-down, the information stored in the electrons states are not the same as in the classical, 0 or 1 states. These two state quantum systems come from the properties of the electron, i.e. it can be in a superposition of both states. That is the key information we can use for calculations beyond what classical mechanics can do.

If we take an example of a two qubit system, we can describe it as a tensor product of two spaces, which will then span a Hilbert space $\mathbb{C}^2 \otimes \mathbb{C}^2$. This space is spanned by the computational basis,

$$\begin{aligned}|00\rangle &:= |0\rangle \otimes |0\rangle\,,\\ |01\rangle &:= |0\rangle \otimes |1\rangle\,,\\ |10\rangle &:= |1\rangle \otimes |0\rangle\,,\\ |11\rangle &:= |1\rangle \otimes |1\rangle\,.\end{aligned}$$

In a sentence, a quantum computer is simply a computational device based around a quantum process. Its benefits are inherent in these quantum processes, which can store more information than any typical (classical) computer and thus solve more complicated problems with fewer processes. This allows quantum computers to solve some problems faster than any classical computers.

To perform calculations on qubits we evolve our quantum system according to an operator U such that $U|q_1q_2\rangle = |f(q_1,q_2)\rangle$, where f is the function we want to compute. We then look for a Hamiltonian $\mathcal{H}$ which generates this desired evolution U according to the Schrödinger's equation, (1.1). Such a Hamiltonian exists as long as the linear operator U is unitary [34].

To then convert our known problems to the quantum computer we need a way of using the extra information and so we design special algorithms, the quantum algorithms. They are similar to classical algorithms since they represent a series of steps that compute a solution to a given setup. A classical computer uses bits while a quantum computer stores its data and programs in qubits. These qubits can be put in a superposition, where n qubits have information on both possible states giving a total information capacity of 2^n bits. A quantum algorithm is an algorithm that preserves these quantum states and can use them to simultaneously do the calculation on all of its possibilities, giving it the ability to

e.g. run a program with both 0 and 1 as input and evaluate both options.

As was talked about in Section 1.4 we can describe qubits using two dimensional vectors over the field of complex numbers, $\mathbb{C}$. Further we can write those vectors as a linear combination of some base vectors $\alpha_0 |0\rangle + \alpha_1 |1\rangle$. Here, we are using $|0\rangle$, $|1\rangle$ as our base where,

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \text{and} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Any linear combination, $\alpha_0 |0\rangle + \alpha_1 |1\rangle$, $\alpha_0, \alpha_1 \in \mathbb{C}$, of these basis vectors is called a superposition if $|\alpha_0|^2 + |\alpha_1|^2 = 1$. To draw this complex vector we use a three dimensional shell of a sphere, called a Bloch sphere, where the third dimension comes from the complex numbers, in Figure 1.2 we see a demonstration of this using a arbitrary state $|\psi\rangle$.

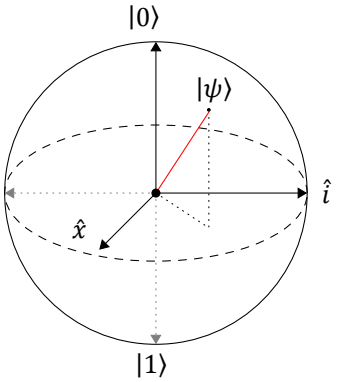


Figure 1.2: Bloch sphere with the bases $|0\rangle$ and $|1\rangle$ drawn on top and bottom respectively.

While the qubit remains unobserved it is in any state on the surface of the Bloch sphere, but as soon as it is measured it collapses to either $|0\rangle$ or $|1\rangle$, (or to another chosen measurement basis). Even though this state is a real physical state of a quantum system it is hard to picture, therefore one often only looks at the mathematical properties, which have been established during the last century. This is because the mechanics in the quantum realm can be unintuitive and clashes with our regular experiencing of reality. Let us look at an example of a qubit in a state, $|\psi\rangle$, where it has a probability one half of being measured in $|0\rangle$ and a probability one half of being measured in $|1\rangle$, i.e. $|\psi\rangle = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle$. See Figure 1.3 for and illustration of such $|\psi\rangle$ on a Bloch sphere.

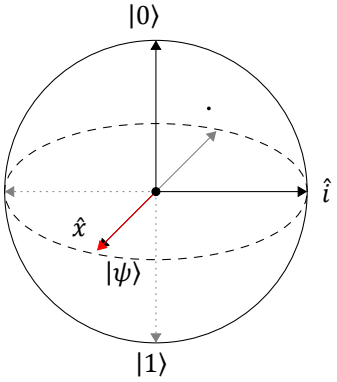


Figure 1.3: Bloch sphere with a half state vector ψ, where $|\psi\rangle = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle$.

The basis chosen in the previous paragraph is often referred to as the Z-basis and the state $|\psi\rangle$ is one of the other standard basis, the X-basis. More precisely,

$$\text{Z-basis} : \{|0\rangle, |1\rangle\},$$
$$\text{X-basis} : \left\{|+\rangle := \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle, |-\rangle := \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right\}.$$

There is also a lesser used Y-basis, $\{|i\rangle, |-i\rangle\}$. Note that there is no *measurable* difference between $|+\rangle$ and $|-\rangle$ if one measures a state in the Z-basis, they do however behave differently in quantum algorithms and this is crucial [33].

Quantum algorithms are often represented in the form of circuits. Through the circuits we can easily explain what is going on with each qubit, since it displays how gates are applied to particular qubit(s). Each gate represents a matrix transformation on a single or multiple qubits. There are several standard gates which frequently appear in quantum algorithms; a summary of the ones appearing in this thesis can be found in Appendix B and an example of a circuit can be found in Figure 1.4.

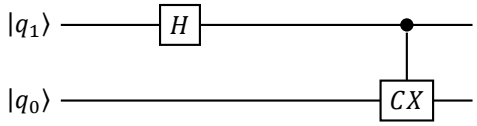


Figure 1.4: A simple circuit with a Hadamard gate, H, on $|q_1\rangle$ and a conditional not gate, CX, on $|q_0\rangle$, conditioned on $|q_1\rangle$. This circuit and its purpose are further explained in Appendix A.

1.6. Example of a Quantum Algorithm

By now we have covered how and why quantum algorithms work and the tools and notations we need to understand quantum computing. But when is a quantum algorithm faster? The first example of such an algorithm was Deutsch's algorithm. That algorithm was later expanded to the Deutsch-Jozsa algorithm, which we will exemplify here. The problem the Deutsch-Jozsa algorithm solves is defined as follows.

Deutsch-Jozsa algorithm problem
Given a function

$$f : \{0,1\}^n \to \{0,1\},$$

which is promised to be either constant or balanced, find out whether f is a constant or a balanced function [10].

Here, balanced means that f returns 0 for exactly one half of the input values and 1 for the other half; constant means that f either returns 0 for all inputs or it returns 1 for all inputs.

A classical algorithm can solve this problem in $2^{n-1} + 1$ steps, since in the worst case we have to check half of all possible inputs and then one more to determine whether f is balanced or constant. The quantum algorithm proposed to solve this, by David Deutsch and Richard Jozsa in 1992, can do it in one single step [12].

The explanation of this method is as follows. We assume all registers are in the $|0\rangle$ position, we need that the register *a* is in the $|1\rangle$ position so we flip it with a not gate (X). Next we switch our basis from Z to X by using the Hadamard gate (H) on all registers. Then our states are,

$$|q_i\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}},\ i = 1, \dots, n,$$
$$|a\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

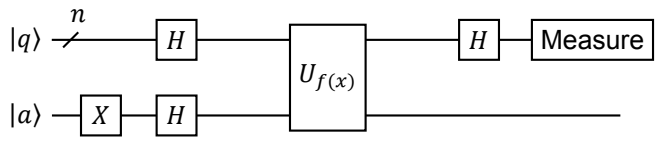


Figure 1.5: Deutsch-Jozsa circuit. The first register $|q\rangle$ is of size n and is the input into the function. The second register is $|a\rangle$ which is the ancilla register and will represent the output. Here H is the Hadamard gate where $H|0\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $H|1\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$, X is the not gate, where $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$, and U is the unitary that represents the constant or balanced function f $U_f|q\rangle|a\rangle = |q\rangle|f(q) \oplus a\rangle$, cf. Appendix B.

In this state we apply the unitary operator that represents the *oracle*, i.e. a black box function that knows our answer. We implement it separately by connecting a cnot gate where appropriate from the values in the register q and to the qubit a. This formulation allows for the final state of the q to tell if the function encoding the unitary is balanced or constant. This is because with a control in the Z basis and with a target bit in $|1\rangle$, phase kickback gives us the state

$$\frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{xy}\right]|y\rangle\,.$$

We convert the basis back by using the inverse of the Hadamard which is just the Hadamard itself and measure the register $|q\rangle$. If it comes out 1, the function is constant or else balanced. So this result is achieved in a single step when a classical method needs $2^{n-1}+1$ steps. This full process is illustrated in Figure 1.5. It should be pointed out that for this specific problem the advantage is by design; it was the specific intent of Deutsch to find a problem that would have an advantage when formulated as a quantum system. This advantage, going from $O(n)$ to $O(1)$, is not restricted to such problems and it is usually what researchers look for when developing new quantum algorithms. Such potential to lower the complexity of a problem has been proved in a multitude of other practical problems, from factoring numbers and optimization [16] to solving linear systems[17]. The algorithms that are of interest in this work are a quantum solution to the problem of solving linear systems. These algorithms have been shown to have better efficiency than any classical ones. We will explore some of them further in the subsequent chapter.

2

Quantum Linear Solvers

In this chapter, we will briefly go over the classical definition of the problem of solving a linear system before moving on to the quantum formulation of this same problem. We will then discuss the two quantum linear solvers which are the focus of this thesis, the direct quantum method and the hybrid quantum/classical method. For both methods, we will go in detail through the algorithmic process and look at their algorithmic complexity. Finally, we will briefly look at experimental realizations that use these methods and are already in use, together with their implications.

2.1. Solving Linear Systems

The problem of solving a linear system is a collection of problems where multiple equations involve the same set of variable and the goal is to solve for those variables. A linear system commonly shows up in most branches of science, such as design problems, signal processing, and homework. The general definition of this problem is as follows.

Solving linear system
Given an invertible matrix $A \in \mathbb{C}^{N \times N}$, and a vector $b \in \mathbb{C}^N$ find a vector $x \in \mathbb{C}^N$ such that

$$Ax = b. \tag{2.1}$$

This problem is solved by finding, or approximating, an inverse of the matrix A such that $x = A^{-1}b$. Many algorithms exist to solve this problem. An example of an efficient method that is widely used is the method of Conjugate Gradients. It can solve this in $O(N\sqrt{\kappa})$ time, but CG has the condition that A is a Hermitian, positive-definite matrix.

For solving the linear system in a quantum setting a re-formulation is needed to reach the full benefits of a quantum speedup. The quantum formulation is:

Quantum Linear Solver (QLS) [17]
Let A be an $N \times N$ Hermitian matrix with a unit determinant. Let b and x be N-dimensional vectors such that $x := A^{-1}b$. Let the quantum state on $\lceil \log(N) \rceil$ qubits $|b\rangle$ be given by

$$\frac{\sum_i b_i |i\rangle}{|| \sum_i b_i |i\rangle ||_2},$$

and for $|x\rangle$ by

$$\frac{\sum_i x_i |i\rangle}{|| \sum_i x_i |i\rangle ||_2},$$

where b_i, and x_i are the i-th components of b and x respectively. Given A and $|b\rangle$, output a state $|\tilde{x}\rangle$ such that $||(|\tilde{x}\rangle - |x\rangle)||_2 \leq \epsilon$, with some probability larger than $\frac{1}{2}$ that the following holds

$$|x\rangle = A^{-1} |b\rangle . \tag{2.2}$$

The output $|x\rangle$ is a quantum state that represents x and to read out all N values of x would take $O(N)$ runs of the algorithm. This is because even though $|x\rangle$ contains all the information of x we can measure only one value of it each time, so to use this solution we must remain in the quantum state. To use this we do not simply look at a $Ax = b$ problem but embed this as a subroutine in another problem. Then for example one can calculate $x^T M x$, where M is some linear operator, you can map M to a quantum operator and get $\langle x|M\,|x\rangle$[24].

2.2. Quantum Algorithm for Linear Systems of Equations

In 2009 Harrow, Hassim, and Lloyd introduced a quantum algorithm to solve the quantum variation of the linear system $Ax = b$ [17]. Their solution is built from previously established quantum algorithms. The three main steps of the quantum algorithm for solving a linear system referred to as HHL after the author's initials, are the phase estimation, ancilla rotation and then uncomputing of the registers B and C. These steps are shown in in the circuit model of the algorithm in Figure 2.1.

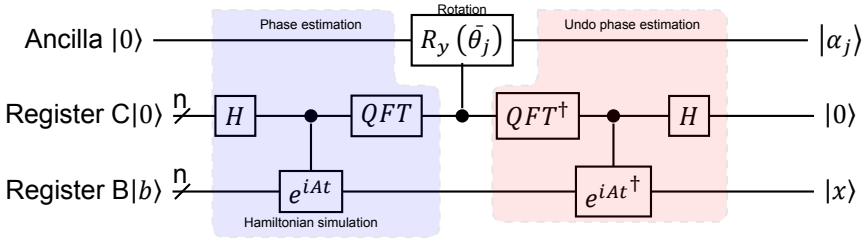


Figure 2.1: Circuit model of the algorithm proposed by Harrow, Hassim and Lloyd [28].

In their paper Harrow, Hassim, and Lloyd use a detailed eight step procedure to describe their algorithm (cf. Appendix B):

1. Convert A into a unitary operator e^{iAt}.
2. Prepare the B register $|b\rangle$.
3. Compose $|b\rangle$ into the eigenvector basis of e^{iAt} using phase estimation.
4. Apply the conditional Hamiltonian evolution, conditioned on register C in Figure 2.1.
5. Apply the Quantum Fourier Transform (QFT) on register C.

6. Use an additional ancilla, α_j, with a rotation conditioned on register C.

7. Undo the phase estimation to uncompute everything except the last ancilla bit

8. Measure α_j, if it is 1 the computation was successful if not repeat.

Phase estimation

The first part of the algorithm is the phase estimation, i.e. QPE. This part estimates the eigenvalues of A by taking the operator eigenstate of A and estimating its phase θ. Since our eigenvalues are complex numbers we can write $\lambda = e^{i2\pi\theta}$, so our eigenvalue estimation in this form is $U|\phi\rangle = e^{i2\pi\theta}$. The method of how to do this in a quantum circuit is broken into two major subroutines the, Hamiltonian simulation and the quantum Fourier transform, describe below.

Hamiltonian Simulation

Converting A into a unitary operator e^{iAt} is only possible when A is Hermitian and unitary. This has been shown to be efficient through multiple different techniques, such as Trotter-Suzuki [36], graph-coloring, and Szegedy's quantum walk. All these techniques show that a Hamiltonian that acts on n qubits can be efficiently simulated by a quantum circuit U_H if for $O(poly(n, t, 1/\epsilon))$-number of gates it holds that $||U_H - e^{iHt}|| < \epsilon$. This however adds a dependency on time t, which is crucial since this time factor can be used to show that any such simulation requires at least $O(t)$ time to run and so always acts as a lower bound.

Quantum Fourier Transform

In a classical sense, the Fourier transform is a method that allows us to change into frequency space and analyze a signal on its base frequencies. Given a square invertible matrix F its discrete Fourier transform is $\mathcal{F}_{nm} = \frac{1}{\sqrt{N}} e^{(\frac{i2\pi}{N})(nm)}$. The columns of the new matrix are orthonormal so they can be used as a basis known as the Fourier basis of the matrix F. Similarly to the classical Fourier transform the quantum Fourier transform, QFT, takes a quantum state $|x\rangle$ to a new state f_x thus,

$$|x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{N-1}^{k=0} e^{\frac{i2\pi}{N} xk} |k\rangle . \tag{2.3}$$

Rotation

The purpose of the ancilla qubit is to check if the estimation of the eigenvalues was successful in the phase estimation. Therefore we only need to measure the ancilla qubit at the end to know if we should run everything again or whether we can continue with $|x\rangle$ into the next step. This check is done using amplitude amplification, meaning we rotate the ancilla qubit based on the outcome of the phase estimation while it is in the Fourier basis to amplify the correct amplitudes so that if the phase estimation was successful it will rotate from $|0\rangle$ position to $|1\rangle$. This technique is based on Grover's search algorithm and lowers the error rate of the algorithm that is based on probabilistic success to be arbitrarily close to 0 with $O(1/\sqrt{s})$ repetitions, where s is the success rate of the algorithm [16].

Uncomputing

The uncomputing step in its entirety is not strictly necessary for the mathematics of this method to work out. This is because we only want the $|x\rangle$ register so we could focus only on getting that and leave the rest be. If however we would want to continue the circuit forward into some new problem where we use $|x\rangle$, we need to return register C to $|0\rangle$. We do this by simply undoing the operations we performed on it. More precisely, since gates in quantum circuits are matrix operations we invert the operations already done by taking the inverse of those matrices; for the Hadamard gate this is just Hadamard itself and for the QFT we use the inverse $\text{QFT}^\dagger$.

Completion
At completion, the HHL algorithm should have the ancilla qubit in state $|1\rangle$, C register in state $|0\rangle$, and the B register in state $|x\rangle$. We only measure the ancilla qubit and it determines whether the algorithm worked or not, however as the condition on the ancilla qubit is set before the undoing the phase any failures in this step would not be detected by this method.

Improvements
After the work of Harrow, Hassim, and Lloyd improvements have been made both in the efficiency of the algorithm and by relaxing the restrictions. In 2010, Andris Ambainis improved the efficiency of the amplitude amplification and so reduced the overall runtime of HHL [3]. Ambainis generalize amplitude amplification and used that to generalization to shorten the HHL runtime. In 2017, Childs et al. further reduced the runtime of the original algorithm by bypassing the quantum phase estimation using a technique for implementing operators suitable for Fourier series representation [9]. Lastly, Wossnig et al. reduced the restrictions on sparsity by building on the singular value estimation subroutine, given that a dense matrix A has a spectral norm bounded by a constant [41]. The development of the HHL runtime is depicted in Figure 2.2.

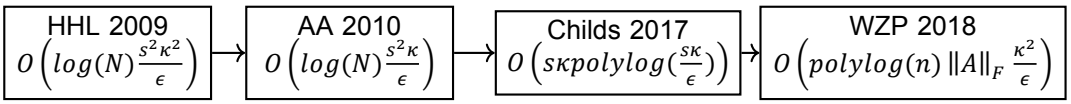


Figure 2.2: Chronological improvements of the HHL, starting with Harrow, Hassim, and Lloyd in 2009 [17] then Andris Ambainis work in 2010 [3], followed by Childs et al. in 2017 [9], and lastly the work of Wossnig, Zhao, and Prakash [41].

Experimental realization of HHL
In 2011 Pan, Cao, et al. ran a proof of concept experiment using a 4-qubit nuclear magnetic resonance quantum information processor to solve a 2×2 linear system using the HHL algorithm [28]. Their experiment was successful and results showed a 96% fidelity. This was the first realization of the HHL algorithm.

Similarly, in an article from 2013 by Cai, Weedbrook, et al. the simplest meaningful instance of the HHL algorithm was tested on 2×2 matrices again in an experiment using a linear optical network with four photon base qubits [6].

2.3. Variational Quantum Linear Solver
VQLS is a hybrid solution using both classical and quantum computing methods to solve the quantum system of equations. This new algorithm, published in 2019, is not an iteration on the HHL but a proposed intermediary solution to HHLs high demand of qubits and high quality of computation. This method is designed to work on so called noisy intermediate scale quantum computers (NISQ) by reducing the depth of the quantum circuit needed to solve the problem. It does this essentially by moving parts of the algorithm back to a classical computer. This frees up qubits and creates stability as they do not need to be in a quantum state for too long, therefore reducing the likelihood of de-coherence. The authors of the article experimented with this new algorithm on hardware from Rigetti using a problem size of 32×32 with successful results. This is a heuristic algorithmic approach and as such it is hard to make rigorous complexity analysis but numerical simulations show that it is efficient in the condition number κ and in the error $1/\epsilon$, but it can never have as good of a complexity as the HHL is supposed to have on an ideal quantum computer [5].

The VQLS algorithm defines the cost function in terms of the overlap between the quantum states $|b\rangle$ and $\frac{A|x\rangle}{\sqrt{\langle x|A^\dagger A|x\rangle}}$. To estimate this cost we use an efficient quantum circuit. The α parameters are determined classically and fed to the quantum computer. The quantum computer then prepares the state $|x(\alpha)\rangle$ and with it efficiently estimates a cost function $C(\alpha)$ which is then returned to the classical computer. After this the classical computer minimizes the parameter α over a cost function using a

classical optimization algorithm, e.g. COBYLA [30] or a gradient based optimization. The new α is then fed again into the quantum computer which again prepares the state $|x(\alpha)\rangle$. This loop repeats itself until the desired minimal cost is reached, i.e. $C(\alpha) \leq \gamma$. The system then outputs the optimal α which can be used to prepare a state $|x\rangle$. This process is illustrated in Figure 2.3.

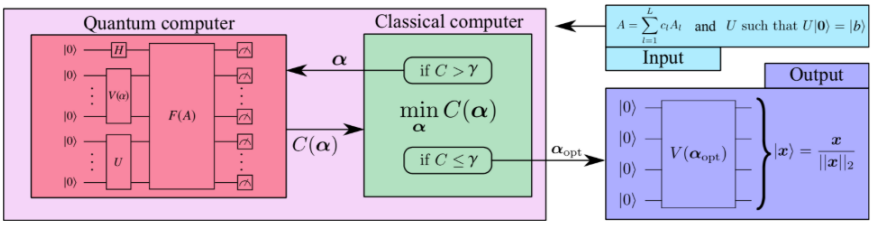


Figure 2.3: A diagram representing the VQLS method, from Bravo et al. [5]. H is the Hadamard gate, $V(\alpha)$ is the ansatz function, U is the unitary that prepares $|b\rangle$ from $|0\rangle$, $F(A)$ is the cost function, $C(\alpha)$ is the cost given the parameter α, which is the optimization parameter. To obtain the output the ansatz $V(\alpha)$ is repeated with the now optimal α which yields $|x\rangle$, cf. Appendix B.

Input

The system takes as input a decomposition of the matrix A into a linear combination of unitaries with complex coefficients, $A = \sum_n c_n A_n$, and a unitary matrix U that prepares the state $|b\rangle$ from $|0\rangle$. To assume an A can be given in such a form is the same as to assume that a Hamiltonian can be given as a linear combination of Pauli operators $H = \sum_l^L c_l \sigma_l$, see variational quantum eigensolver [8]. There the assumption is that L is polynomial in the number of qubits, n; in addition we need to assume that the norm of A is bounded by one, $||A|| \leq 1$, and that the condition number is not infinite, $\kappa < \infty$.

Ansatz $V(\alpha)$

The ansatz prepares a potential solution $|x(\alpha)\rangle = V(\alpha)|0\rangle$, where $V(\alpha)$ is a trainable gate sequence. $V(\alpha)$ can be expressed as sequence of L gates $V(\alpha) = G_{k_L}(\theta_L) \dots G_{k_1}(\theta_1)$, where $G_k(\theta)$ is the k^{th} gate with input θ_k. Therefore, α encompasses both k and θ, $\alpha = (k, \theta)$, where k is a discrete parameter determining the circuit layout, and θ are continuous parameters such as rotational angles of the gates in question. There are many ways to prepare an ansatz but no way to know what is best in each case. For example, we can choose to only vary one of the parameters, k or θ and then have a variable-structure ansatz, or a fixed-structure ansatz, respectively. These sorts of methods can be used to save on computation as fewer parameters are needed to solve. In the classical optimization, the general rule is only that your ansatz needs to be able to prepare your desired state of $|x\rangle$.

Cost function $F(A)$

Like with the ansatz, a cost function can be defined in many ways. A simple cost function comprises the overlap between a projector $|\psi\rangle\langle\psi|$ and the subspace orthogonal to $|b\rangle$, where $|\psi\rangle = A|x\rangle$. This can be stated as follows,

$$\hat{C}_G = \mathrm{Tr}(|\psi\rangle\langle\psi|(\mathrm{I} - |b\rangle\langle b|)) = \langle x|H_G|x\rangle,$$

where H_G is the Hamiltonian, $H_G = A^\dagger(\mathrm{I} - |b\rangle\langle b|)A$. This cost function is small if $|\psi\rangle$ is proportional to $|b\rangle$ or if the norm of $|\psi\rangle$ is small. To avoid the second situation, as it does not represent a proper solution, we can normalize $\hat{C}_G$ using the norm of $|\psi\rangle$, thus

$$C_G = \frac{\hat{C}_G}{\langle\psi|\psi\rangle} = 1 - |\langle b|\Psi\rangle|^2, \quad |\Psi\rangle = \frac{|\psi\rangle}{\sqrt{\langle\psi|\psi\rangle}}. \tag{2.4}$$

Assuming that the condition number of A is smaller than infinity and so the norm $\langle\psi|\psi\rangle \neq 0$ this function vanishes under the condition that $|\psi\rangle \propto |b\rangle$, which is exactly the case when our QLSP is solved.

Classical optimization

Classical optimization is a well studied field and there are multiple optimizers that can be used to train the gate sequence $V(\alpha)$. The choice of optimizer depends on the choice of ansatz as each ansatz has different variables that need to be optimized and their connection to the classical optimization is an ongoing research. For example, gradient based methods have been shown to be beneficial in variational methods as the first-order gradient information can be directly accessed by measurement of the quantum circuit [23].

Experiments

The original paper proposing VQLS conducted experiments with a real quantum computer. Their results indicated that VQLS scales efficiently in both the condition number, κ and the error, $1/\epsilon$. The tests they ran using Rigetti's quantum machine were implemented on problems of sizes 2×2, 4×4, 8×8, and 32×32. This first experiment is already larger in size than any previous HHL experiment, in terms of problem size, i.e. matrix dimensions.

As this article with VQLS was published in September 2019 it has been a short time to publish experimental research, but as the algorithm has been implemented on IBM's, Rigetti's, and Pennylane's platforms more will undoubtedly follow.

2.4. Summary of Quantum Linear Solvers

Even though the quantum formulation of the linear systems is not the exact same as the classical linear systems it has value and promise in advancing the computation. Both, the HHL algorithm and the VQLS algorithm provide a better alternative to their classical counterparts in regards to their computational complexity. In the next chapters we will look at the methods used in implementing these algorithms on a modern platform.

3

Method

Implementations of a QLS have come far since Harrow, Hassidim, and Lloyd first published their algorithm. Experiments have been performed and improvements have been made to the versions they envisioned. In addition to algorithmic improvements, quantum computers themselves have become more robust and readily available. Quantum computing experiments are no longer bound to laboratories of large institutions as several companies have created online interfaces for their growing number of quantum computers. One such is IBM; they have built a platform that can connect anyone to nine of their working quantum computers. Coding for a quantum computer has been brought to a more accessible level through the use of high-level language such as Python. IBM has created a Python package called Qiskit that allows a user to write gate and circuit instructions directly in Python. This Python code is then compiled into the machine code needed for IBM's quantum computers by IBM's system and can be executed either using an online interface or directly from a Python console. In this Python package, there is also a system to run quantum simulations locally or on the IBM cloud, the simulators are used to test implementations on both a theoretical perfect quantum computer and a noisy model of a real quantum computer.

The version of Qiskit and its packages I used for producing the results shown in this thesis can be seen in Listing 3.1,

Listing 3.1: Qiskit versions used in this work.

```
'qiskit-terra': 0.16.1,
'qiskit-aer': 0.7.1,
'qiskit-ignis': 0.5.1,
'qiskit-ibmq-provider': 0.11.1,
'qiskit-aqua': 0.8.1,
'qiskit': 0.23.1
```

In this Chapter we introduce IBM's quantum system by looking into the hardware and the software of it, and how they are used in the implementation of the quantum linear solvers.

3.1. IBM's Quantum Network

As the race to build a quantum computer is ongoing so is the race to create the software on which quantum computers will run on. IBM has made it far on both fronts as they have built working quantum computers and brought them online along with building a software platform based on the Open Quantum Assembly Language (OpenQASM) [11]. Building on the OpenQASM, IBM has made an open-source framework for developing and testing quantum algorithms, called the Quantum Information Science Kit (Qiskit) and the cloud computing platform IBM Q Experience. Qiskit is a Python package that allows a high-level interface with the OpenQASM language and it can be run on and set up locally on any machine, just like any other Python package. Qiskit lets one run quantum simulations locally as well as provides a backend to IBM's quantum-simulators and -computers [4]. As Qiskit is an open platform

many are developing algorithms with it and so both HHL and VQLS have already been implemented. This implementation will be the basis of the next phase of experiments in this work. The main part of this project uses both the simulators and hardware available through the IBM Q Experience, and is developed on the Qiskit module.

The Qiskit framework has four main components:

- Terra is the foundational element which all others take from e.g. the circuit structure;
- Ignis handles noise and error correction for better performance and analysis;
- Aqua is a library of quantum algorithms;
- Aer provides the simulation backend allowing local simulation of quantum processes.

A demonstration algorithm could make use of all these packages by creating a circuit in Terra, ties in an existing algorithm from Aqua into the circuit then optimize it using Ignis and finally simulate it in Aer. This process is graphed in Figure 3.1 including the core scientific domains that already have the first steps of quantum advantage ready. Furthermore, IBM provides an excellent documentation on the whole framework on the Qiskit website [4].

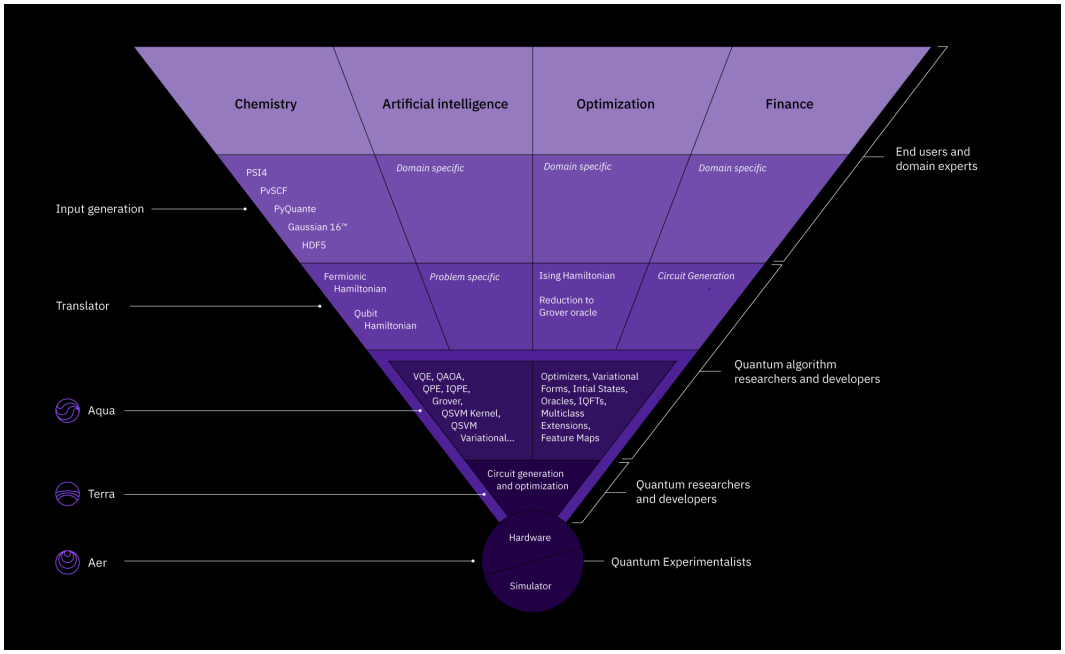


Figure 3.1: The structure of IBM's elemental components [29].

3.2. Simulators

While researchers are working on more reliable hardware the best way to test quantum algorithms is by using classical simulations. As stated by the Church-Turing thesis all Turing machines can compute the same functions so any classical computer, which is Turing complete, can simulate a quantum computer. The purpose of any scientific simulation is to reproduce the actual function of a target event inside of the given test conditions, and so a simulation of a quantum computer by a classical computer mathematically reproduces the conditions of a quantum system. This reproduction can be used to test algorithms without having to set up and program a full quantum computer. Importantly, these classical simulations of quantum computation are not efficient and so often takes much longer to simulate a quantum algorithm than it would take to run on a quantum computer. It is also worth noting that these computations are *perfect* and thus represent the theoretical upper bound of what a quantum computer can do. In the near term, we have NISQ computers which are far from this kind of perfection, however

every year brings improvements in this field.

Qiskit module Aer gives a classical computer the ability to run quantum simulations, allowing the user to quickly run and test quantum algorithms. As a part of this local simulator, it is also possible to run *noisy* tests, that is trying to simulate the real quantum hardware which is in use today. This will be important in the continuation of the development of quantum algorithms such as the VQLS as they are designed for noisy hardware and should be tested on such. This implementation could also prove useful if hardware tests are proving unsuccessful as the noise in the simulations can be calibrated, unlike in the hardware devices. Simulations also offer a greater size than a quantum device, e.g., IBM's online simulator can do a 32-qubit simulation, which is double the number of qubits from the hardware option. Note that simulators do not scale beyond 40–50 qubits since their memory consumption grows exponentially with increasing qubit numbers.

The Aer component of Qiskit provides three main methods of simulations, the State vector method, the Unitary method, and the Qasm method. Each method takes a quantum circuit as an input and outputs a state-vector representing the final values of the experiment.

- State-vector method, is a single shot perfect simulation of a quantum circuit and it returns a vector with the final state of each qubit of the circuit.
- Unitary method, turns a quantum circuit into a unitary matrix. This matrix is then applied to a starting vector, returning the product as a new state-vector.
- Qasm method is designed to simulate a real device. It can simulate the noise and help to create more noise resistant circuits. It can import real data from IBM's quantum computers and use it to simulate the noise present in each of the imported systems basis gates as well as the effects of its topology. The simulation can also take into account the measurement error of each qubit. This gives a quick way of seeing if an algorithm will function on a real device.

The three methods provided by Qiskit can be separated into two distinct categories, perfect simulation, and noisy simulation. The State-vector and Unitary method are perfect simulations where a single run of the circuit yields the desired results by mathematically perfect methods. However, the Qasm method is a noisy simulation. An ideal circuit is simulated but controlled noise is added to the calculation and the model is run multiple times to achieve a statistical result. This is done to simulate the kind of quantum computers we have today more accurately. The Qasm method can control the noise on multiple levels, such as gate noise or measurement noise. Noise models can also be actively fetched to represent a specific IBM quantum computer to accurately see how an algorithm would perform on that device.

Aqua

The intent behind Aqua is to ease the access of researchers who want to experiment with quantum computation in their work. The algorithms provided are core to much of the quantum advantage that has been discovered so far; such as the HHL algorithm or variational methods. With this package, a researcher can call on these methods, either as a single algorithm or as a part of a larger experiment, and use them directly without having to write up all of the quantum circuits. This means that research with quantum computers is not bound to a quantum computer scientist, but can be used e.g. in chemistry or optimization research. To further accommodate such research, that is any research seeking quantum advantage, specific domains have been identified where the quantum advantage is clear and special methods have been added to allow researchers to use existing tools and techniques in conjunction with Aqua without having to learn quantum specific skills [29]. Thus, the Aqua package allows convenient use of existing quantum algorithms by providing a ready-built solution with a standard interface for many researchers.

Aqua functions by giving the user direct access to the open-source software library of quantum algorithms without any intermediary layer. This gives a user also the full power of classical computing by giving him the ability to, e.g., create loops and logic checks whilst utilizing quantum components. This is often much easier than accessing blocks of code through API or other indirect methods. The

openness of the platform has also allowed errors to be quickly corrected and for users to submit their work into the library, this increases the reliability and the scale of the whole system.

Hardware

Quantum computers are not *really* here yet. Even though the effort to develop them is growing and new publications show improvements in this field every month, the truth remains we are not working with perfect qubits. The qubits that we are working with are noisy and prone to errors, and also very small scale systems, on the order of 10 qubits (which may soon become on the order of 100 qubits). These systems we call noisy intermediary scale quantum computers or NISQ for short. NISQ systems are good for testing small scale circuits that take a short time to run but are not taking over from classical computers yet. However, if better techniques are developed it may very well be that NISQ quantum computers are all we need to reach quantum advantage in some cases [31].

IBM uses superconducting qubits as the basis for all their quantum computers. Each computer needs a complex cryogenic system to keep the quantum states for as long as possible and so that the qubits remain undisturbed by heat. The control mechanism is a microwave pulse system that gives the user precise control over their experiments. In total 28 machines have been put online for use eight of which are in open access to anyone who signs up for IBMQ network, these open access machines vary from a single qubit computer to a 15 qubit computer [26].

Complexities are a part of working with NISQ devices. When working with quantum computers you need to not only consider noise in the circuits but also coherency time, i.e. time a quantum state can be held, and connectivity of the qubits themselves, e.g. Figure 3.2. All of these factors can make a difference in how your algorithm runs. Because of these factors and the complications around real quantum computers this thesis is bound only to simulations not using real quantum hardware.

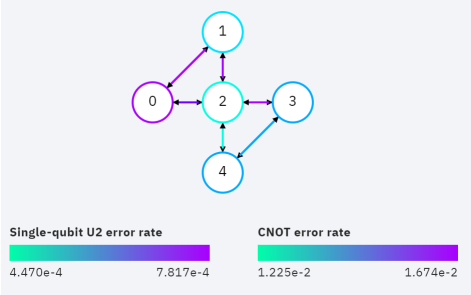


Figure 3.2: The topology of the ibmqx2 5 qubit machine, one of the machines available through the IBM Q experience.

To summarize, the IBM system offers open access to quantum computing simulators and hardware through a Python interface. The platform offers varying methods for conduction experiments and eases usage further with a pre-built quantum algorithm inside the core module Aqua. With these elements, we build the experimental procedure and get our results in the following Chapter.

4

Experiments and Results

In this chapter, we will cover the procedure of each experiment and give their results. The procedure of both HHL and VQLS followed the construction given in the IBM Qiskit textbook [4]; with augmentation for the decomposition of A and matrix size variations for VQLS, and noise accommodation with code refactoring for HHL. For HHL, a single function was produced that handled all test runs; this function can be found in Appendix C.1. In VQLS a separate function handled each size and noise configuration, i.e., six functions in total; they can be found in Appendix C.2.

For experiments, both HHL and VQLS were tested with a perfect simulation and a noisy simulation, and each of those tests was run on three sizes of matrices 2×2, 4×4, and 8×8. In selecting matrices for testing I kept a similar structure of sparsity and condition numbers throughout varying sizes to be able to compare results better. Note that due to the limited capabilities of the simulators larger and more complex matrices were not able to be tested. All test matrices can be found in Appendix D.

The chapter is structured such that the procedure of each experimental setup proceeds the results of those experiments. The procedure represents an implementation of the algorithms described in chapter 2 and is based on the IBM Qiskit textbook [4]. The outputs of the experiments are mainly circuit size, runtime data, and fidelity of the solution vector. After each section, we do a quick analysis but reserve an in-depth discussion for Chapter 5.

4.1. Procedure of HHL

The procedure for the HHL algorithm used in the following experiments uses the Aqua implementation of the HHL function. The input is comprised of an eigenvalue circuit, circuit size (including ancilla), a matrix A, a vector b, and the size of the matrix, n. Since this implementation requires us to input the circuit that handles the eigenvalues, i.e., the quantum phase estimation from Chapter 2.2, separately, we first use a special Aqua function, *EigsQPE*. This extra input is needed as the solver has not been fully implemented yet.

The *EigsQPE* function is a numerical approach to solving the quantum phase estimation, i.e., QPE, meaning it is not an exact method. It uses an expansion method, either the method of Lloyd [19] or Suzuki [36] and takes as input the order of expansion and the number of time slices. Experiments varying the number of ancilla and order expansion can be found later in this chapter, cf. 4.7. The expansion methods are both based on the Trotter expansion formula,

$$e^{A+B} = \lim_{n\to\infty} \left(e^{A/n} e^{B/n}\right)^n, \tag{4.1}$$

where A and B are operators, Equation (4.1) corresponds to formula (3.10) in [36]. In Lloyd's method it is used directly whilst in Suzuki the generalized form, called the Suzuki-Trotter formula, is used. This adds the expansion order, p, where $p = 2$ corresponds to the Trotter expansion,

$$\lim_{n\to\infty} f_{n,1}(A_j) = exp\left(\sum_{j=1}^{p} A_j\right), \tag{4.2}$$

where again A_j is a bounded operator and p is a positive integer, cf. formula (3.9) in [36].

To perform QPE we need a Hermitian matrix. If our matrix A is not Hermitian, we expand it in the following way.

$$H = \begin{bmatrix} 0 & A \\ A^H & 0 \end{bmatrix},$$

and continue with H in place of A. Evidently, this doubles the size of the matrix, which is quite costly. Therefore, for experimental purposes, this situation was avoided.

After having checked that A is valid and having the QPE circuit we can run the HHL algorithm with the desired simulator, chosen by the user. This part has a timing loop around to measure how long the run takes. When we have the results of the HHL experiment we run a classical linear solver to compare the fidelity of the HHL solution. Fidelity is calculated simply as the state fidelity between the normalized HHL solution and the normalized classical solution,

$$\tilde{x}_{norm} = \frac{\tilde{x}}{||\tilde{x}||_2},$$
$$x_{norm} = \frac{x}{||x||_2},$$
$$f = \frac{\tilde{x}_{norm}}{x_{norm}},$$

where $\tilde{x}$ is the experimental solution, x is the reference solution, and f is fidelity. The function finishes by writing all the relevant test data to a designated file.

4.2. Perfect Simulation of HHL

For the experiments with a state vector simulator, a perfect simulation, the Suzuki method was used for the expansion, 3 ancilla, and a default vector of $b = \mathbb{1}_n$, Lloyd's method was tested for reference see further in Appendix E.2. For the Suzuki method the parameters were, order 1, and number of time slices 50. Each test matrix was run three times and the results of time and fidelity were averaged over those three runs. The runtime was measured over only the linear solver part of the script. In Table 4.1 we see the circuit size of matrices tested in this simulation. The circuit size did not vary for the different test matrices in each size category as the expansion order remained the same for all tests.

Matrix size	2×2	4×4	8×8
Qubits	7	8	9
Depth	101	104	111

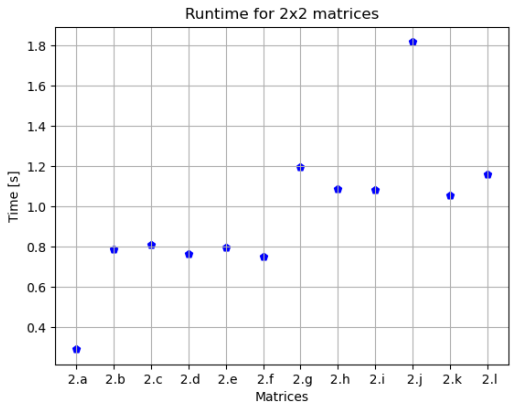
Table 4.1: Circuit sizes in HHL state vector experiments. Qubits represent the maximum amount qubits the circuit uses, and depth is the maximum amount of gates any qubit passes through before conclusion.

Experiments of size 2×2

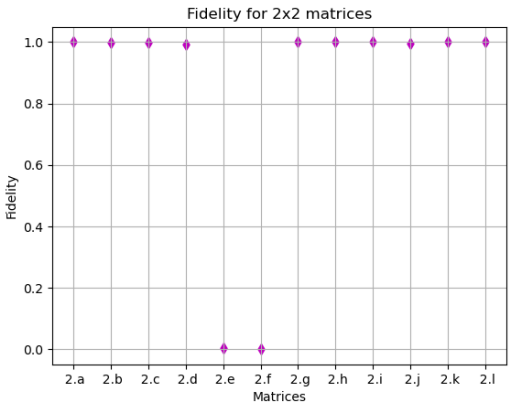
The matrices prefixed with 2 in Appendix D were all tested for the perfect simulation of HHL. In Figure 4.1a we can see the runtimes for each test matrix. The first matrix, $2.a$, is the identity and it took a considerably shorter time to run. The general runtime was between 0.8 seconds and 1.2 seconds. Fidelity of the 2×2 state vector simulation can be found in Figure 4.1b, there we see that the fidelity was near 1, i.e., near perfect, except in two outlying cases, $2.e$ and $2.f$.

Experiments of size 4×4

The matrices used in the 4×4 experiments can be found in Appendix D prefixed with 4. Looking at Figure 4.2a we can see that again the matrix $4.a$ is an outlier; again this is the identity matrix. The results are generally steady up to $4.g$, with the matrices coming close to a runtime of 2.5 seconds. The matrices from $4.g$ take longer going from 2.5 seconds up to 10 seconds, these matrices increase the



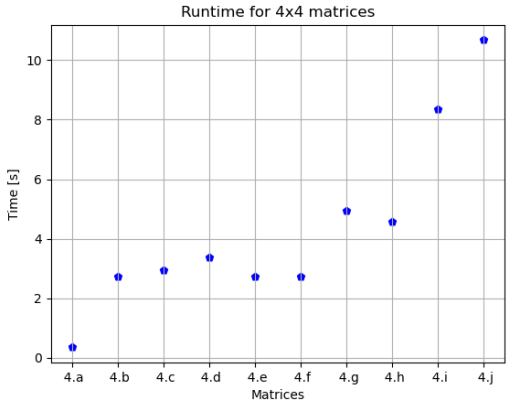
(a) Runtimes for 2×2 matrices.



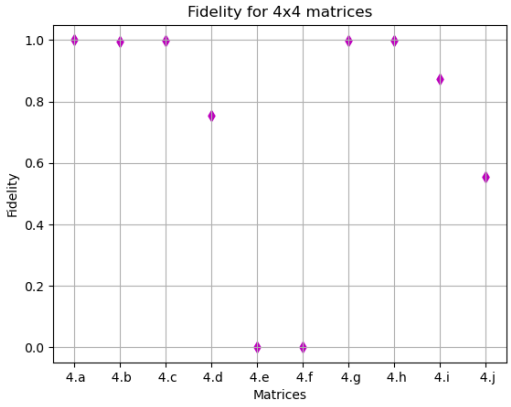
(b) Fidelity for 2×2 matrices.

Figure 4.1: Experiments with 2×2 matrices, state vector simulations.

density from the first matrices which where all diagonal matrices. For fidelity we see that the solution conforms to the classical solution nicely for most matrices, see Figure 4.2b. For the first three cases the results are near perfect, but again $4.e$ and $4.f$ are outliers showing very poor fidelity, these matrices where selected as they are ill-conditioned and so that undoubtedly plays a part in their poor fidelity.



(a) Runtimes for 4×4 matrices.



(b) Fidelity for 4×4 matrices.

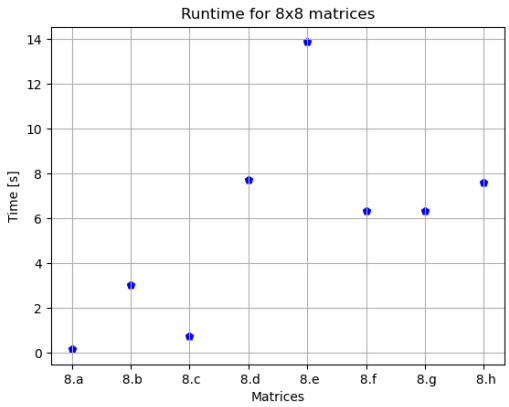
Figure 4.2: Experiments with 4×4 matrices, state vector simulations.

Experiments of size 8×8

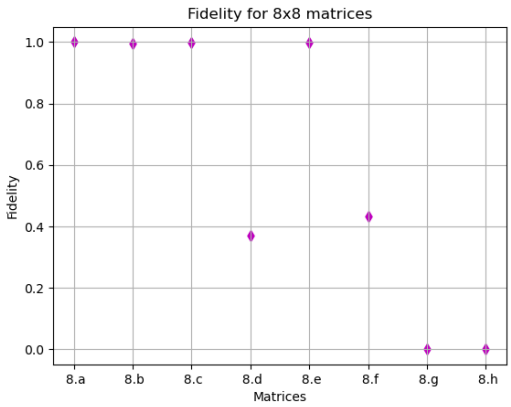
Test matrices of size 8×8 for the state vector simulation of HHL can be found in Appendix D with prefix 8. This was the toughest to run and only eight matrices ran successfully. In Figure 4.3a we see that the identity matrix, $8.a$ again has a shorter runtime similar to that of the identity of 2×2 and 4×4. The next two, have a runtime of 3.9 and 1.1 seconds for $8.b$ and $8.c$, respectively. From $8.d$ onward the runtime is closer to 7 seconds but peaks in $8.e$ at 14 seconds, correspondingly $8.e$ has the worst sparsity. The fidelity of these solutions is near perfect for the first runs; $8.a$ is 1.0, $8.b$ is 0.996, and $8.c$ is at 0.999. The $8.h$, $8.g$, and $8.h$ represent the ill-conditioned matrices and again they show quite bad fidelity from other results.

State Vector Simulation of HHL Analysis

In the state vector simulations of HHL we immediately saw that the quantum algorithm can produce accurate results in a short test time frame. The circuits that come out of the runs were all in an expected range but larger than the HHL theory sets it out; there qubit size for a 2×2 matrix is 5 not 7 [17]. The



(a) Runtimes for 8×8 matrices.



(b) Fidelity for 8×8 matrices.

Figure 4.3: Experiments with 8×8 matrices, state vector simulations.

impression from these results point to the sparsity number of the matrices increase the runtime of the algorithm while the condition number decreases the fidelity. Lastly, note a quick runtime for the identity matrix, which was almost constant, i.e., independent of size. However, this may be due to some short circuit in the simulator.

4.3. Noise Simulation of HHL

For experiments with a noisy simulation the same expansion and parameters were used as in the state vector simulation. For a noisy simulation, a noise model must be chosen. There are a variety of parameters to alter for testing certain aspects of a circuit and its sensitivity to noise. As the goal here is to show how these algorithms would perform on a real NISQ quantum computer. For this the best option is to generate a noise model based on an existing machine.

Therefore we import data of the 15 qubits Melbourne computer of the IBM cloud services, the model data can be found in Appendix C.1. The Melbourne computer has five basis gates, the Identity (I), the Not gate (X), Conditional not gate (CX), Rotation on Z axis (RZ), and square root not gate (SX). The computer connects its 16 qubits using the topology shown in Figure 4.4. This specific model was chosen as it is the only real quantum machine available with enough qubits for our purposes, i.e., more than 9. The test matrices used in the noisy experiments are the same as the ones used in the state vector simulations, they are listed in Appendix D.

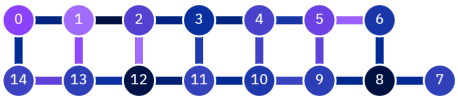


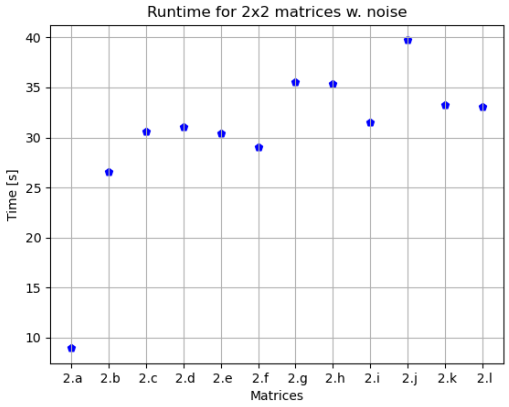
Figure 4.4: The topology of the Melbourne 15 qubit computer. Nodes represent qubits and edges represent qubits' ability to entangle.

The circuit sizes for all runs relative to the matrix input size is the same as it was in the state vector simulation, see Table 4.1. One could try to mitigate the noise with additional changes to the circuit but the point here is to test how the algorithm would perform as is on a NISQ device.

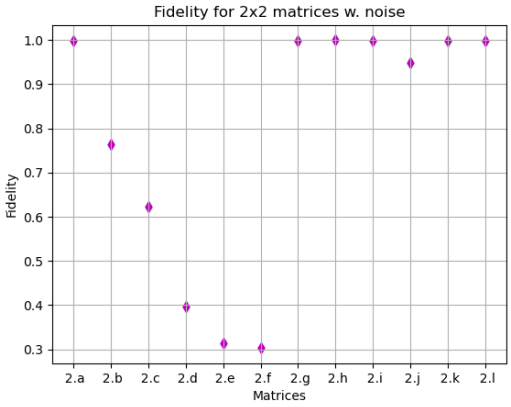
Experiments of size 2×2

In Figure 4.5a, we see the runtime results of the 2×2 test matrices. That the time it takes to execute each run is considerably larger than in the state vector simulations; the average of the three runs is now around 32 seconds, excluding $2.a$, the identity. In Figure 4.5b, the fidelity of the same test matrices

is presented. In comparison to Figure 4.1b, the no-noise run of the same size, we can see that the spread is considerably wider and a similar low fidelity results can be found for $2.e$ and $2.f$.



(a) Runtimes for 2×2 matrices with noise.

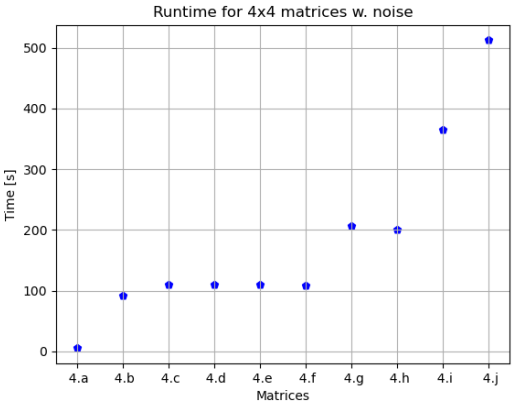


(b) Fidelity for 2×2 matrices with noise.

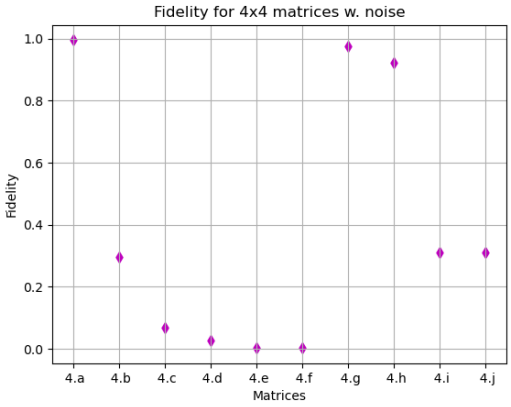
Figure 4.5: Experiments with 2×2 matrices, noisy simulations.

Experiments of size 4×4

Figure 4.6a presents the results of all matrices of size 4×4 tested in a noisy simulation of HHL. We see that the general runtime for the diagonal matrices is around 110 seconds, except for the identity, $4.a$. The runtime again increases from $4.g$ on and peaks at 500 seconds, so a 5 fold increase going from the general diagonal matrices to the dense random matrices. For fidelity of the solutions of each test, depicted in Figure 4.6b, there is a slow decline in solution fidelity as we go trough the matrices until $4.g$ where we jump back up. Again the ill-conditioned matrices perform near zero while a dense random matrix, $4.j$, is at 0.3.



(a) Runtimes for 4×4 matrices noise.



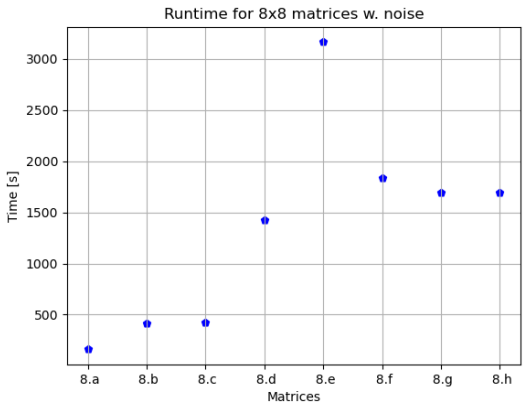
(b) Fidelity for 4×4 matrices noise.

Figure 4.6: Experiments with 4×4 matrices, noisy simulations.

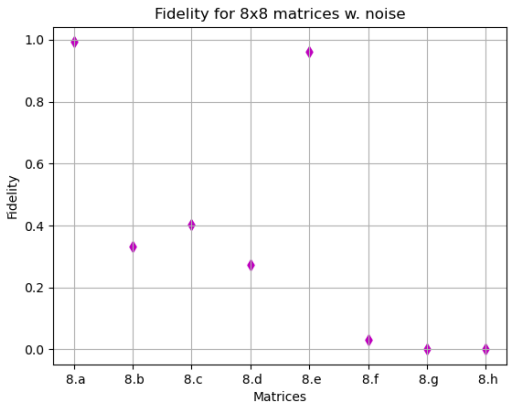
Experiments of size 8×8

In Figure 4.7a we see the runtime of the tested matrices in the 8×8 category. Here we see the longest runtime, $8.e$, took over 3.000 seconds and the runtime of $8.f$,$8.g$,$8.h$ was around 1.700 seconds, all being more than three times that of any 4×4 run. Again the identity stands out with a very short runtime, around 10 seconds. This could be due to some optimization in the simulator. Fidelity, seen in Figure 4.7b, is considerably weaker then without noise but shows similar results to the 4×4 case. Two cases stand out with perfect fidelity, the identity $8.a$ and $8.e$ which is a tridiagonal matrix. The

ill-conditioned matrices, the last three, all perform near zero.



(a) Runtimes for 8×8 matrices with noise simulation.



(b) Fidelity for 8×8 matrices with noise simulation.

Figure 4.7: Experiments with 8×8 matrices, noisy simulations.

Noise Simulation of HHL Analysis

The noise experiments overall took a significantly increased runtime compared to the state vector simulation. They also suffer from a loss in fidelity in almost all results. This is to be expected with no change in the circuit for error correction as the noise is of course there to disturb the normal function of the simulation. It is however clear that results can be achieved under certain conditions and that if they had the capacity NISQ computers could produce passable results under these experimental conditions.

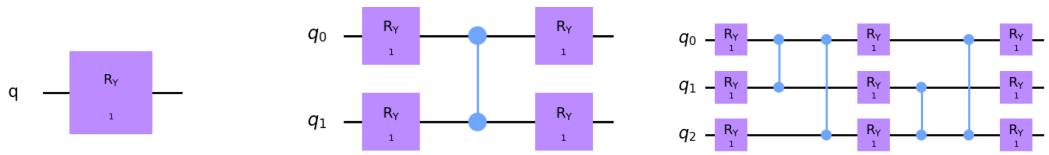
4.4. Procedure of VQLS

The procedure of the VQLS follows the steps laid out in Chapter 2.3. Three different procedures were used to run each of the different sizes of matrices as the function setup could not be generalized. These three different procedures differ in the ansatz and the control tests as these are tailored to matrix sizes.

The VQLS algorithm takes in a matrix A, which needs to be decomposed into a linear combination of unitaries $A = \sum_n c_n A_n$, where c_n are complex coefficients. Secondly, it needs a unitary matrix U such that it takes a state $|0\rangle$ to the state $|b\rangle$. After minimizing the cost function it outputs a normalized form of the state $|x\rangle$. As a simplification, vector b was chosen such that $U = H_n$, that is a tensor product of Hadamard gates.

The Ansatz of VQLS is a trainable gate sequence, $V(\alpha)$, which prepares the state $|x\rangle$ from $|0\rangle$. There are many ways to make such a circuit but its implementation depends on its function. One can vary both, the structure of the whole circuit, or the continuous parameters on rotational gates. It is hard to say what is the best ansatz, but starting with an example of what worked for other researchers is often a good practice. In the VQLS experiments, we chose a fixed-structure ansatz, that is it only varies the continuous parameters of the rotational gates and not the structure of the circuit itself. For each of the three sizes a different ansatz is used as the ansatz structure depends on the circuit structure. The ansätze chosen can be seen in Figure 4.8 for details about gates see Appendix B. These particular ansätze were chosen from the main sources of these algorithms [4, 5] as they proved successful in their experiments.

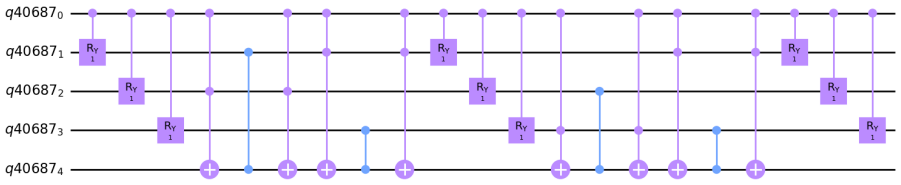
At each iteration the rotation of the gates changes and this allows us to search the state space of A. The rotational values in their corresponding gates can be found in Figure 4.8. This sort of implementation can be further optimized by a so-called Hardware-Efficient Ansatz [18], where gates are chosen to match the gates set of the quantum computer itself; this technique was not employed in the experiments performed for this project.



(a) Ansatz for a 2×2 matrix. (b) Ansatz for a 4×4 matrix. (c) Ansatz for a 8×8 matrix.

Figure 4.8: Ansatz circuit for the three experimental sizes used.

Figure 4.9: Control ansatz for an 8×8 matrix.



Next, the procedure needs to perform a Hadamard test and a controlled test of the ansatz. In the work of Bravo-Prieto et al. [5] this is done using a novel technique called the *Hadamard Overlap Experiments*, which reduces the number of gates used at the cost of an extra ancilla qubit. Following the example of the Qiskit textbook, this was not done, as the overlap test is hard to implement; however, a more general VQLS function should include this. Instead of the overlap test we are using a control Hadamard test. The control Hadamard test uses a control sequence on the gates of the decomposition of A, followed by a control test of the ansatz where an ancilla qubit is used to control selected qubits. This depends on the original ansatz and the first ancilla. We can see an illustration of this in Figure 4.9. The control Hadamard test presents a problem when generalizing the procedure as the control-control-not gates need an added control sequence for each additional qubit. Therefore a Hadamard-Overlap test would be preferable.

Lastly, the procedure needs the main cost function, i.e., the function that returns $C(\alpha)$ and is fed into the classical training algorithm. The cost function creates the circuits of the ansatz and U, and runs the simulation on them to output $\frac{|\langle b|\psi\rangle|^2}{\langle\psi|\psi\rangle}$, as defined in (2.4). For all experiments, the COBYLA method was used as a classical minimizer algorithm. To get out the vector x we then need to run our ansatz one more time with the input as the optimized parameters of the main loop. We do this by building a quick new circuit with just the ansatz, no extra ancilla, and using the parameters recovered from the output of the minimizer as the input. Finally, we simulate this circuit and measure the state to get the solution vector x.

4.5. Perfect Simulation of VQLS

For experimenting with the state vector simulation we must run the state vector simulation in three different locations, twice in the cost function and once after the optimization loop to determine x. The size of the circuit is considerably smaller than in the HHL case as we can see in Table 4.2. Each circuit uses fewer qubits and has much fewer gates. This however is fair in direct comparison to HHL as this circuit requires multiple loops and samplings to complete the calculations, but it does provide better usability on smaller scale quantum machines.

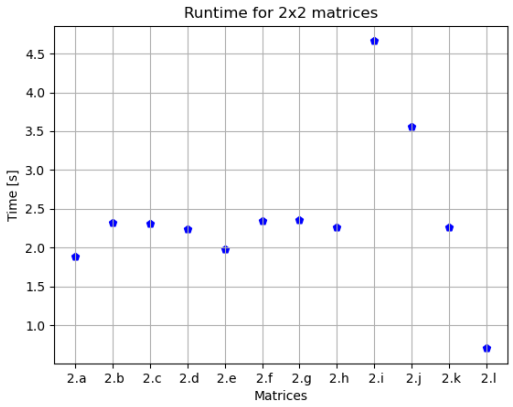
Matrix size	2×2	4×4	8×8
Qubits	3	4	5
Max Depth	8	10	26

Table 4.2: Circuit sizes in VQLS state vector experiments.

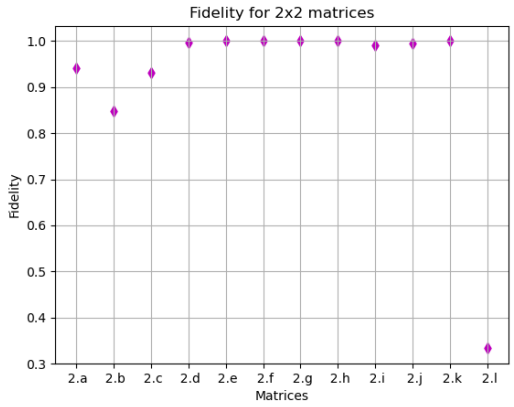
Experiments with the VQLS simulations use the same matrices as the HHL experiments, cf. Appendix D.

Experiments of size 2×2

The results of all 2×2 matrix tests can be found in Figure 4.10a. The runtime was mostly around 2.5 seconds with only two exceptions, $2.j$ and $2.i$ taking longer and one taking shorter, $2.l$. Fidelity results can be seen in Figure 4.10b, the results show a near perfect fidelity for all matrices, with the exception of $2.l$ which is very low.



(a) Runtimes for 2×2 matrices.

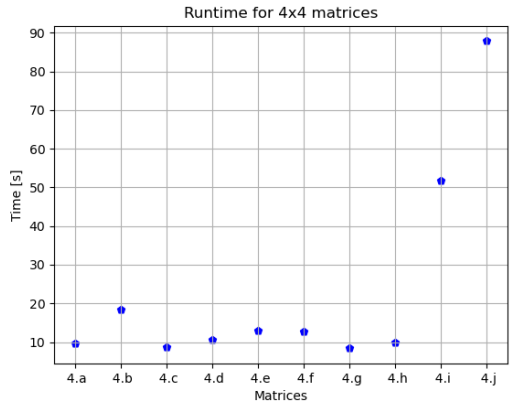


(b) Fidelity for 2×2 matrices.

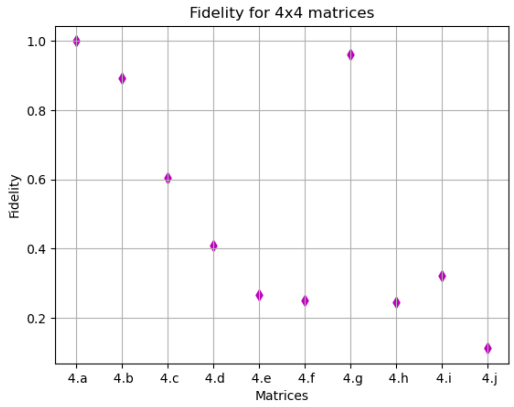
Figure 4.10: Experiments with 2×2 matrices, state vector simulations.

Experiments of size 4×4

In experiments with the 4×4 matrices we have varied time results, as can be seen in the graph in Figure 4.11a. Timings where between 10 and 20 seconds for most tests but for the dense random matrices $4.i$ and $4.j$ the timing when up to 52 and 81 seconds respectively. The fidelity, as shown in Figure 4.11b,



(a) Runtimes for 4×4 matrices.



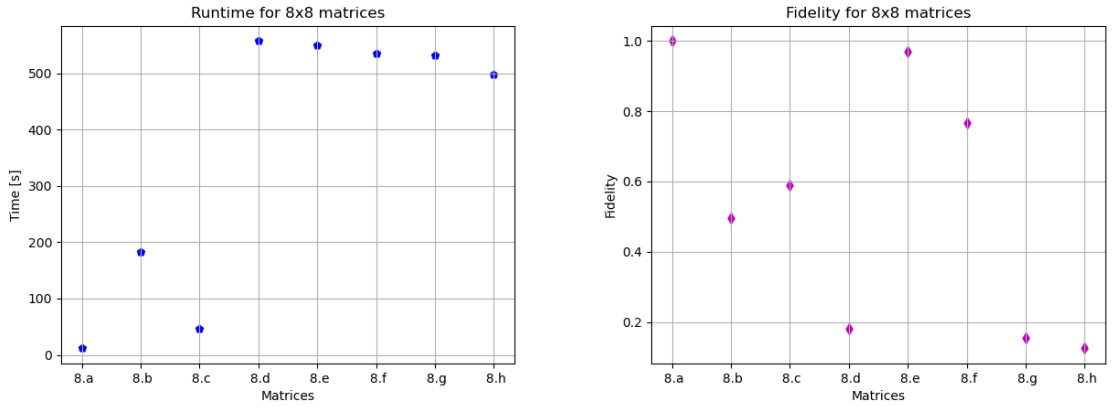
(b) Fidelity for 4×4 matrices.

Figure 4.11: Experiments with 4×4 matrices, state vector simulations.

Experiments of size 8×8

The VQLS simulation can handle more complex 8×8 matrices but as we mainly wanted to compare the results to HHL it seemed redundant to test more matrices just for VQLS. The timing results are seen in Figure 4.12a and the fidelity in Figure 4.12b. Timings are considerably longer than for the 4×4 case with most runs being over 500 seconds long. The fidelity varies considerably but like in HHL $8.a$ and

$8.e$ perform near perfect, but $8.d$, $8.g$, and $8.h$ are near zero. Those last three matrices listed all have a poor condition number, all though $8.d$ was not designed to be ill-conditioned it has $\kappa = 18.4$



(a) Runtimes for 8×8 matrices.

(b) Fidelity for 8×8 matrices.

Figure 4.12: Experiments with 8×8 matrices, state vector simulations.

State Vector Simulation of VQLS Analysis

Shortly, the state vector simulations of VQLS were a success for smaller matrices. They show a much leaner circuit compared to HHL but take a longer time to show results under these perfect conditions. The fidelity is good in the smallest matrices but is not favorable in the subsequent tests. Because of the poorer fidelity it is hard to draw a strong conclusion but there again seems to be a link between sparsity and runtime and also a link between condition number and fidelity

4.6. Noise Simulation of VQLS

We need to change the procedure for the noisy experiment of VQLS. In particular in the cost function, where the circuits are sampled. Using the same noise model as in HHL, detailed in Appendix C.1, we ran multiple shots of the quantum simulation to get a clear result, using the average of those runs as our real result. For our experiment we use 10.000 shots, this number was determined by experiment, see Appendix E.1. This is a rather high number of shots but to reach convergence with the classical optimizer it is necessary to get more accurate results of the quantum simulation [4]. Again the noisy simulation presents the same circuit results as the state vector one as the circuit does not change, see Table 4.2.

Experiments of size 2×2

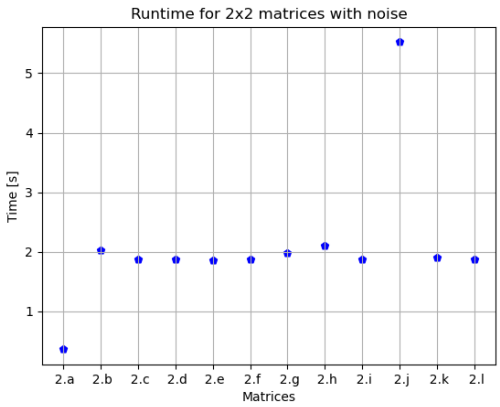
On the whole, the noisy configuration runtime, see Figure 4.13a takes a similar time as its non-noisy counterpart, see Figure 4.10a. Two outliers are the identity and the matrix $2.j$, for which the entries were chosen with a random number generator. The fidelity, however is poor, see Figure 4.13b; with a lone value at 0.7 and the rest somewhere below 0.4.

Experiments of size 4×4

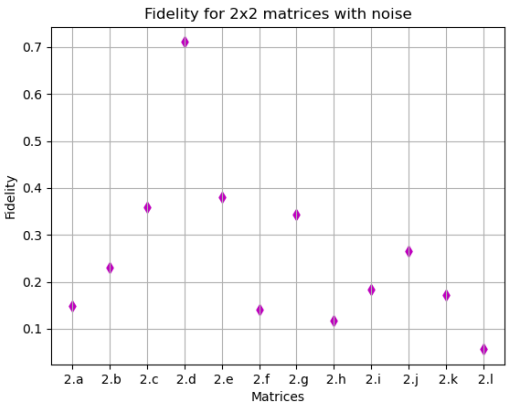
The Figure 4.14a shows us the runtimes of the 4×4 experiments, in a noisy configuration of VQLS. We see, that the average time is mostly steady around 300 seconds for each test matrix, however like in the state vector simulation the $4.i$ and $4.j$ take considerably longer. The $4.i$ and $4.j$ are the dense matrices and they two take a longer runtime then the rest combined. Fidelity, seen in Figure 4.14b, has a steady downwards trend with each tested matrix and is nowhere good; maximum value is only 0.7 and the lowest under 0.1.

Experiments of size 8×8

Unusually, the runtime of these 8×8 experiments was around similar values to the 4×4 cases and very similar to the state vector simulation of 8×8. We can see from Figure 4.15a that the maximum

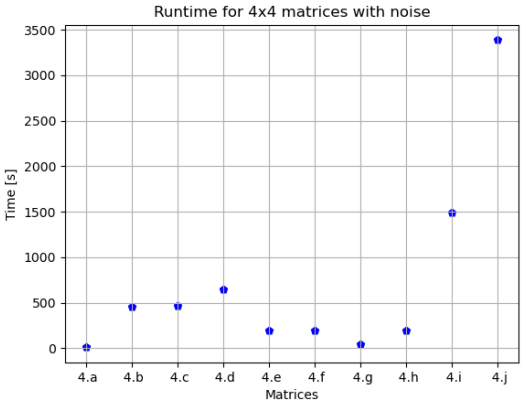


(a) Runtimes for 2×2 matrices with noise.

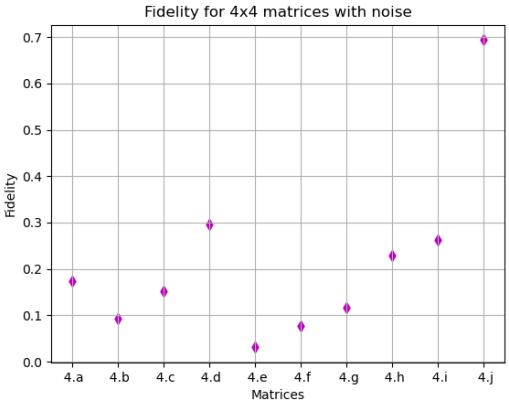


(b) Fidelity for 2×2 matrices with noise.

Figure 4.13: Experiments with 2×2 matrices, noisy simulations.



(a) Runtimes for 4×4 matrices with noise.



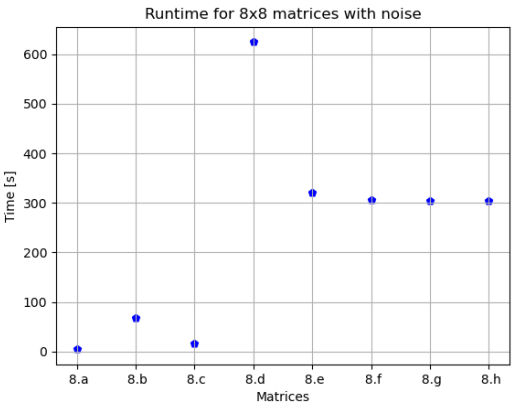
(b) Fidelity for 4×4 matrices with noise.

Figure 4.14: Experiments with 4×4 matrices, noisy simulations.

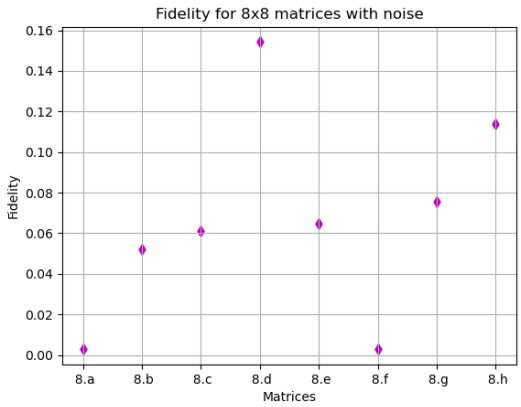
runtime was around 600 seconds, while most cases where around 300 seconds. Fidelity was worse still than in the smaller sizes though, as the highest value doesn't reach 0.16, see Figure 4.15b. These fidelity results are on such a small scale as well that they display no structure or are counter intuitive. For example from $8.f$ to $8.h$ there is an increase in fidelity, while the condition number increases from 10 to 1000, with sparsity unchanged.

Noisy Simulation of VQLS Analysis

The runtime of the VQLS noisy simulations showed results in the 4×4 to take the longest. This could relate to the ansatz or it may be due to the high number of shots needed to converge. In general, the fidelity results for all experiments was poor, with not even the identity matrix showing a fidelity of 1 in any test. It is hard to improve these results by tuning parameters as unlike HHL, VQLS does not rely on approximations like the QPE. We will look at parameter tuning in the next chapter.



(a) Runtimes for 8×8 matrices with noise.



(b) Fidelity for 8×8 matrices with noise.

Figure 4.15: Experiments with 8×8 matrices, noisy simulations.

4.7. Additional Tests for HHL

The HHL algorithm has several variables to calibrate when running it. These variables include the number of ancillae used in measurement and the expansion order of the expansion method. In the following sections, we experiment with varying these two numbers, first the ancilla then the expansion order. These additional tests were conducted using only the 4×4 matrices and perfect simulation conditions.

Varying Ancilla

The number of ancillae determines the measurement accuracy of the final values. Increasing the number of ancillae is similar to increasing the number of bits in a *float* value. The general tests used 3 qubits as ancilla, this value was taken from previous experiments [4]. To further test the effects of the ancilla on fidelity and runtime we performed the tests using 4, 5, and 6 ancillae in the HHL simulation.

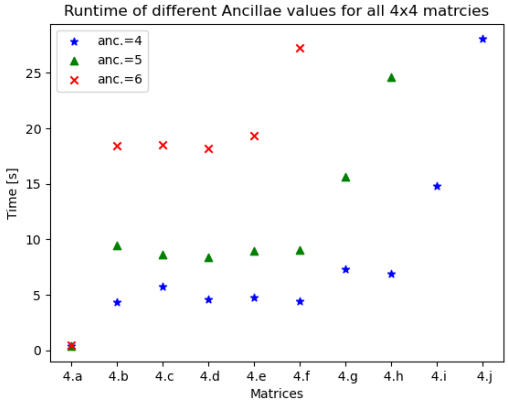
Increasing the amount of ancilla naturally increases the circuit width used in the simulation. Furthermore, the added qubit needs to fully connect to the rest of the circuit to work in measurement and so these added ancilla qubits double the depth of the circuit in each step, see Table 4.3. This linear growth in width and depth size increases the size of each circuit such that the simulations start to fail in the most difficult cases. In the case of 5 ancilla, $4.i$ and $4.j$ cause a segmentation fault and when using 6 ancilla $4.h$ and $4.g$ fail in addition to $4.i$ and $4.j$. This simulation failure is representative of the current capabilities and the limitations of HHL.

Ancillae	3	4	5	6
Qubits	8	9	10	11
Max Depth	104	208	402	802

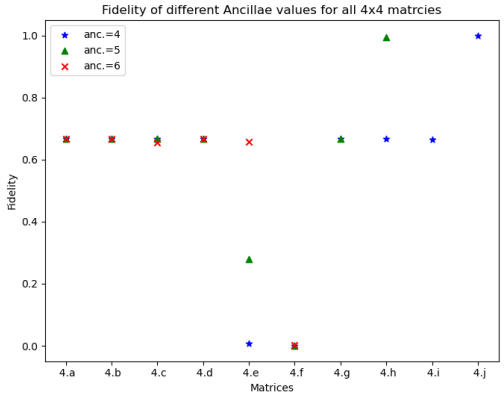
Table 4.3: Circuit sizes in HHL state vector experiments with increased number of ancilla.

In Figure 4.16b we can see that there is not much to be gained from adding the extra ancilla for fidelity in general. Only $4.e$ and $4.h$ show growth in fidelity with an addition of ancilla. The case $4.e$ was chosen as an example of an ill-conditioned matrix with $\kappa = 100$. It can be observed that this kind of an ill-conditioned matrix benefits from the extra ancilla. But we can see from the case $4.f$ that the effects of extra ancilla are not enough to bring up the fidelity there. The matrix $4.f$ is worse conditioned than $4.e$ with $\kappa = 1000$. So the case of an ill-conditioned matrix may benefit from the added ancilla but the effect is quickly lost if the condition number is too large. The price in calculation time is also high. As we see in Figure 4.16a, each added ancilla doubles the runtime. This is not surprising as we already saw that the depth of the circuit doubles. The added time in the calculation for additional ancilla is not rewarded with much greater fidelity. However, the results do show that a particular ill-conditioned case

does have great benefits, with fidelity going from near 0 to 0.7. This could indicate that matrices around that particular condition number should use additional ancilla but this would have to be explored further, perhaps in conjunction with preconditioning.



(a) Runtimes for 4×4 matrices with varying Ancilla.

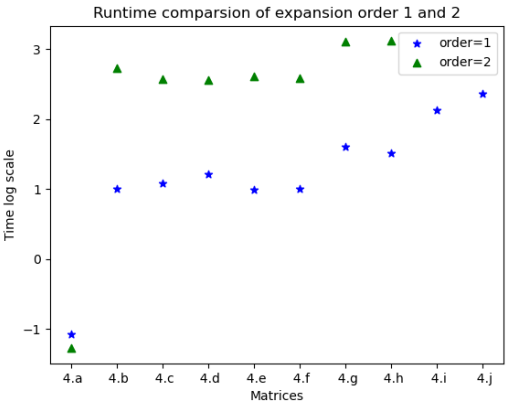


(b) Fidelity for 4×4 matrices with varying Ancilla.

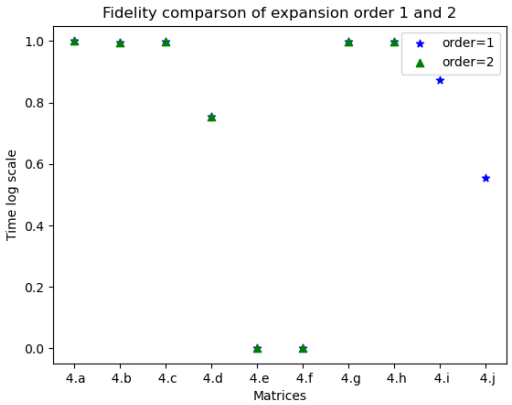
Figure 4.16: Experiments with 4×4 matrices using varying number of Ancilla.

Varying Expansion Order

Expansion order in the Suzuki-Trotter equation, (4.2), can increase the accuracy of the solution by approximating the phase estimation better. This accuracy comes at a cost of computational complexity, even though neither the width nor the depth of the circuit increases. This increase in computational complexity is visible in the runtime for expansion order 1 and 2 in Figure 4.17a, where all 4×4 matrices where attempted. It is also apparent in the case of matrices $4.i$ and $4.j$, since they cause a segmentation fault in the run of the simulation with expansion order 2. The segmentation fault also happened for all 4×4 matrices, except $4.a$, for trials with expansion orders 3 and 4.



(a) Runtimes for 4×4 matrices with varying expansion order.



(b) Fidelity for 4×4 with varying expansion order.

Figure 4.17: Experiments with 4×4 matrices in state vector simulation of HHL with varying expansion order.

5 Discussion

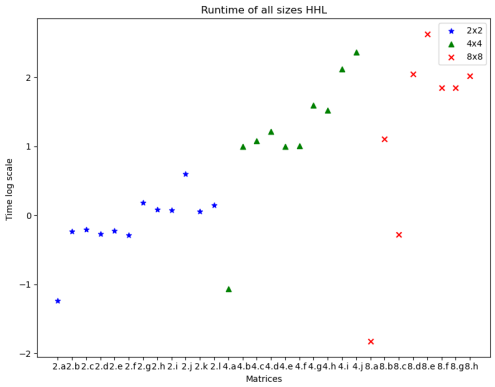
To review the results we will compare the data of VQLS and HHL as well as look at the effects that noise had on the experiments. The noise experiment acts as a stand-in for real experiments ran on quantum computers, since the quantum computers which were accessible at the time could not handle the HHL algorithm. This is already a point in the favor of VQLS, which with its hybrid approach has much smaller quantum circuits. However, VQLS needs to use those circuits more often and is still relying on the robustness of the calculations. Looking at the data of the circuits for HHL and VQLS, cf. Table 4.1 and 4.2, respectively, we see that they are similar in width. VQLS remains smaller but optimization could get HHL to the same level. On the other hand, the depth is on a different scale; the VQLS experiments have a maximum depth of 26 while the HHL algorithm starts at 101 and goes up to 111. This is especially important if one considers that the depth and width multiplied together to form the complete gate count, which contributes to the error rate in noisy conditions immensely, as each gate has some error.

5.1. Comparison

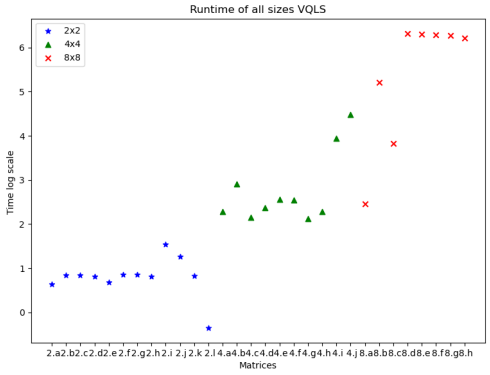
To compare the effects matrix size has on the runtime of each algorithm (under perfect simulation), let us consider Figure 5.1, which shows all runtimes of HHL and VQLS on a logarithmic time scale. The data seems to indicate exponential dependence with size because it appears to be linear under the logarithmic scale. Notably, HHL has three clear outliers that perform under the average in every test, these are the identity matrices. As was pointed out in the analysis of these results in Chapter 4, this may be due to a short circuit or any other optimization in the simulation.

If we then compare HHL and VQLS together on runtime, cf. Figure 5.2, the HHL algorithm performed faster in most cases both with and without noise. The only exceptions to this are the 2×2 cases in a noisy setting which performed better on VQLS. This is not surprising as the theory of VQLS has shown that it is not meant to perform faster than HHL but to be able to bring quantum advantage to linear solvers sooner with smaller circuits, which as previously discussed, it does. It is then important to note that the noise results in VQLS seem to indicate worse scaling than in HHL, but the limited number of results restricts the conjectures possible from this data.

Let us now turn our attention to fidelity, cf. Figure 5.3. We can see there that the matrices $*.d$, $*.e$, and $*.f$ were specifically chosen to see what the effect the condition number of a matrix has on different experiments. These matrices are not present in the 8×8 experiments as any deviation of this kind failed to run in the HHL setup. If we view the fidelity of these solutions we see that the cases $\kappa = 100$ and $\kappa = 1000$, i.e. $*.e$, and $*.f$, respectively, are the single outliers in the state vector simulation of HHL performing near 0 in both cases. For HHL this could be improved by increasing the expansion order in the QPE, but this would come at a cost of circuit complexity. On the same matrices, the 2×2 VQLS state vector model performed well in fidelity but lost it in a noisy scenario. Therefore we can conclude that the condition number affects both algorithms to a high degree.

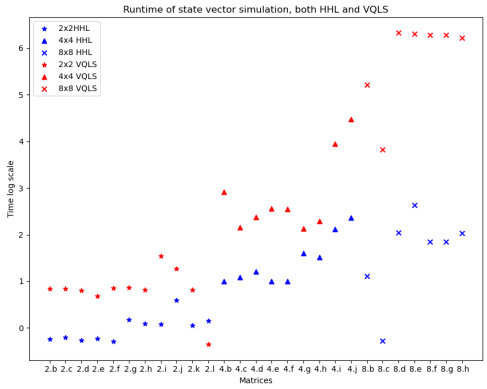


(a) All runtimes of HHL with no noise, time in log scale.

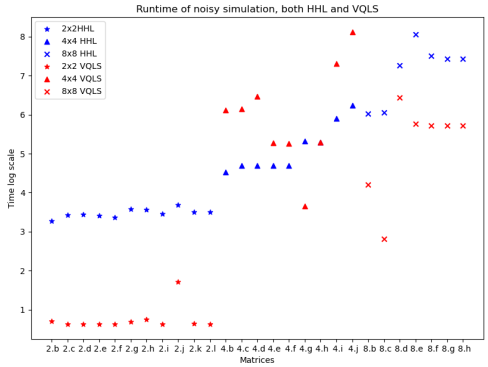


(b) All runtimes of VQLS with no noise, time in log scale.

Figure 5.1: Runtime comparison as a factor of size compared in the state vector simulations of HHL and VQLS, time (y-axis) is in logarithmic scale.



(a) Runtime comparison of HHL and VQLS, without noise.



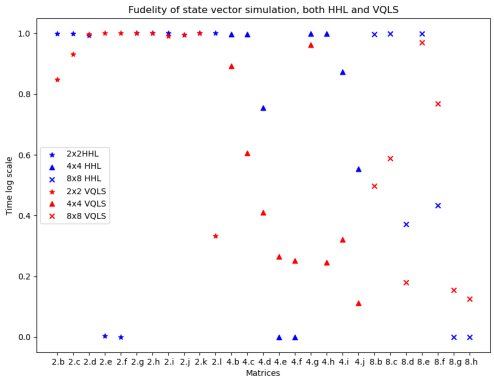
(b) Runtime comparison of HHL and VQLS, with noise.

Figure 5.2: Runtime comparison of HHL and VQLS. Time is set in logarithmic scale and the identity matrices have been taken out.

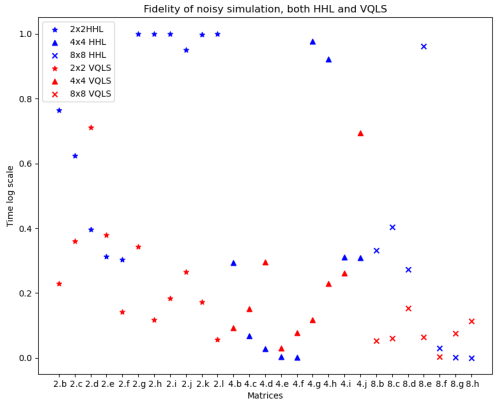
Notably, the results of the experiments show no direct correlation between runtime and fidelity. When we plot these together the fidelity seems independent of runtime, see Figure 5.4, where time is again in logarithmic scale on the y-axis and fidelity on the x-axis.

When viewing the data of noisy simulations and comparing it to their no-noise counterparts, it is clear that adding noise to the system changes the outcomes greatly, especially its runtime. The increase in runtime can go from seconds to minutes, in both HHL and VQLS, see an example for the 4×4 cases of HHL in Figure 5.5. The added time to the simulation is not a direct indication of poor performance on a quantum computer, since a quantum computer naturally has such conditions, and the increase in difficulty comes from the increased circuit size. However, the increased runtime of VQLS in a noisy setting is not only due to increased simulation difficulty but also due to the fidelity of the cost function. As VQLS runs an optimization loop, errors can cause the loop to run far more times than it did under perfect conditions.

Viewing the fidelity in the same light, we see a considerable downgrade in terms of the accuracy of the solution provided. This pushes the reality of using the current quantum computers for solving linear systems a little further out since it is difficult to use these outputs in a real setting.

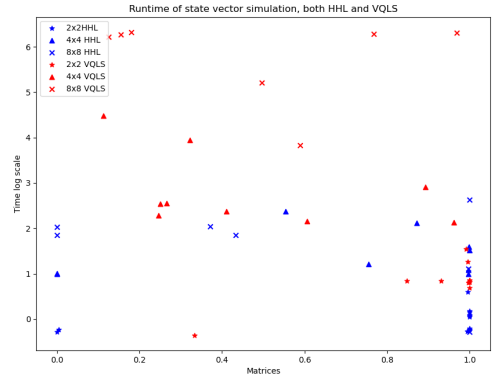


(a) Fidelity comparison of HHL and VQLS, without noise.

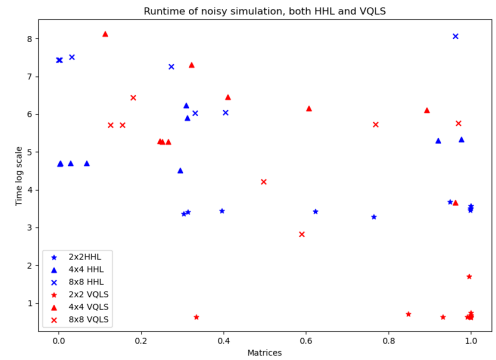


(b) Fidelity comparison of HHL and VQLS, with noise.

Figure 5.3: Fidelity comparison of HHL and VQLS.



(a) Runtime relation to fidelity in a perfect simulation.



(b) Runtime relation to fidelity in a noisy simulation.

Figure 5.4: Runtime in relation to fidelity, comparison of HHL and VQLS in both a perfect and a noisy setting.

5.2. Future Work

From an experimental point of view, there are multiple avenues for future work; from improving the algorithms themselves to using the algorithms in conjunction with other work. It is clear, that the groundwork for better linear solvers has been laid out and that if we had perfect and large quantum computers we could do much more with these algorithms.

Moreover, more and different experiments can be done with the algorithms provided. There are unanswered questions in what causes difficulty in the noisy runs of VQLS and in the HHL additional tests. This could be improved by different approaches to the circuit design or simply with better simulators. A good example would be to run the implementation of real hardware. Real hardware experiments have been performed with both examples [5] [28] [40] [6] but doing them on different implementations, e.g. varying the ansatz for VQLS or the QPE for HHL, could yield good improvements in the scientific knowledge needed before we can claim quantum advantage using QLS. In the line of algorithmic improvements, a full generalization of VQLS would be very desirable, especially if it includes the Hadamard-Overlap test and a variable structure ansatz.

With improved algorithms designing a practical problem is often the next natural step. For example, setting up a larger problem that uses the quantum linear solver. This would involve taking some known problem that uses a linear solver, then converting it to a quantum setting so that $|x\rangle$ need not

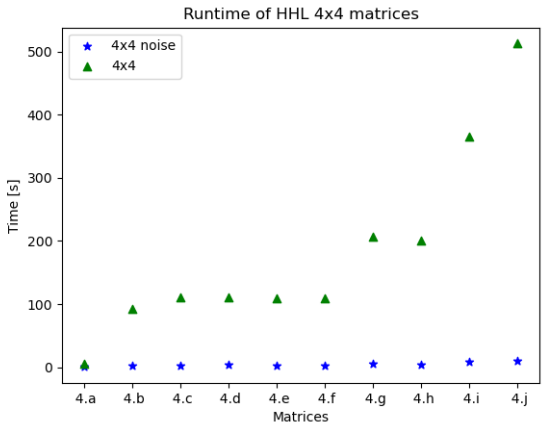


Figure 5.5: All 4×4 HHL runtimes.

be read classically but is used immediately in its quantum state, as already proposed in [24]. This sort of implementation has already gained some traction in the field of machine learning, see [32] and [13].

5.3. Conclusion

In the preface of the thesis these four questions where proposed.

1. Which approaches exist to solve linear systems on quantum computers?
2. How do the implementations of these approaches scale with problems?
3. Can these approaches get to quantum advantage in the near term?
4. Can one use these approaches within a larger algorithm?

This thesis is developed around answering these questions and here is a summary of the answers.

1. There are two quantum algorithms that can solve a reframed problem of linear systems, the HHL method and VQLS method. HHL is a full quantum solution while VQLS takes a hybrid approach using both classical optimizers and quantum solutions. Both methods are a part of very active ongoing research in the field of quantum computing.

2. HHL has a theoretical scaling of $O(s\kappa \mathrm{polylog}(s^2\kappa^2/\epsilon))$ while VQLS has heuristic scaling which has been shown experimentally to be efficient in the condition number and error. Experiments showed that simulations of the quantum implementations of these algorithms grow exponentially with the size of the input matrix. This increase is likely a cause of complications experienced in experiments where simulators were unable to complete varying calculations in larger test cases. The experiments seem to show a correlation between the sparsity and the runtime and a correlation between fidelity and the condition number. However, neither HHL nor VQLS showed dependency between the fidelity of the solution and the size of the input. Neither was their dependency shown between runtime and fidelity.

3. For these approaches to reach quantum advantage in the near term they both require the problem of quantum memory to be solved efficiently. Meaning, they need to store matrix A and vector b efficiently on quantum computers to use them as inputs to algorithms. HHL requires a circuit of $2n + 1$ qubits and has a deep gate structure. This gate structure requires HHL to have robust quantum computers which are expected to be a reality in the near term. VQLS has the same qubit size as HHL, $2n + 1$, but a much shallower gate structure. This gate structure makes it

much more feasible to run VQLS on near term hardware. However, noisy experiments of VQLS showed that VQLS still relies on robustness as the cost function output needs accuracy for the minimization loop.

4. Embedding these linear solvers into a larger algorithm is feasible. Taking any classical problem that requires a linear solver one can replace that classical solver with either of the quantum approaches. This method restricts the choice of problems to ones where A is a square Hermitian matrix but this is a common restriction and is adhered to in many physical problems, such as boundary value problems. This switch has been tested in a finite element solver by Montanaro and Pallister, where they found that the improved HHL algorithm from Childs et al. [9] can only achieve a polynomial speed up but not exponential as previous work had stated [25]. The inherent problem with any such experiment is that it does not capitalize on the full power of the quantum linear solver as one reads out the solution vector x which is rather inefficient. A better way to design such a problem is to have a larger quantum algorithm that produces A and $|b\rangle$ then runs a quantum linear solver as a subroutine. It should then use the output solution x in a quantum setting before finishing its calculations. This kind of solution has been demonstrated in support vector machines by Rebentrost et al. using the HHL algorithm [32].

There are still multiple routes left to explore when it comes to Quantum Linear Solvers. This work has shown two approaches that can solve the quantum linear systems problem, and demonstrated them in a variety of experiments spanning different sizes and complexity. It is hard to conclude that a clear quantum advantage can be gained in the near-term use of these methods, but if sufficient improvements in hardware are made either solution could prove to be a good algorithm of choice for linear solvers in quantum environments.

A

Quantum Hello World

Hello World is often the first exercise a programmer will do when learning a new programming language, it gets its name from its function of printing out the phrase “Hello World”. The compatible test in the quantum computing world is testing a two qubit system with a Hadamard gate and a C-not gate so that they produce either $|00\rangle$ or $|11\rangle$ with a half probability. A circuit representation of such a program can be found in Figure A.1.

The a walk through of this introduction into quantum programming with the IBM Qiskit package can be found as a Jupyter notebook in the Qiskit textbook [4].

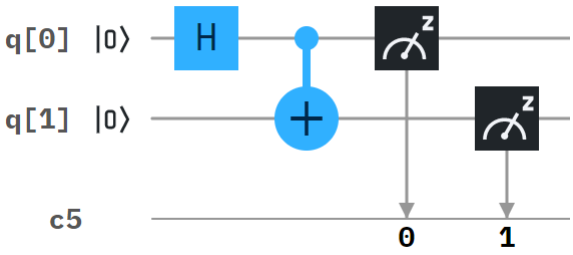


Figure A.1: A circuit of two qubits and one classical bit. q[0] first has a Hadamard gate giving it the position $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, then q[1] is flipped conditioned on q[0]; after these two gates both qubits are measured and the results put into the classical bit.

B

Gate Glossary

Below is the summary of the standard gates used in the circuits in the thesis. For more details see Qiskit documentation [37].

- **The NOT-Gate (X-Gate)**: negates the input,

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

- **The Hadamard-Gate (H-Gate)**: changes the basis from X to Z and vice versa,

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

- **The Rotation Gate** $R_x(\theta)$: represents rotation around the x-axis for θ,

$$R_x(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}.$$

- **The Rotation Gate** $R_y(\theta)$: represents rotation around the y-axis for θ,

$$R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}.$$

- **The Rotation Gate** $R_z(\theta)$: represents rotation around the z-axis for θ,

$$R_z(\theta) = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}.$$

- **The Controled-Not-Gate (CNOT-Gate, CX-Gate)**: represents controled negation, i.e. it flips the target qubit if the control qubit is $|1\rangle$,

$$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

C

Implementations of HHL and VQLS

Initialization and Functions

Listing C.1: Imports and basic functions

```
from qiskit import Aer
from qiskit.circuit.library import QFT
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.quantum_info import state_fidelity
from qiskit.aqua.algorithms import HHL, NumPyLSsolver
from qiskit.aqua.components.eigs import EigsQPE
from qiskit.aqua.components.reciprocals import LookupRotation
from qiskit.aqua.operators import MatrixOperator
from qiskit.aqua.components.initial_states import Custom
import numpy as np
import time

# Data collection loop
def WritetoFile(f, data):
   listToStr = ' '.join([str(elem) for elem in data])
   file = open(f, "a+")
   file.write(listToStr)
   file.close()

def fidelity(hhl, ref):
   solution_hhl_normed = hhl / np.linalg.norm(hhl)
   solution_ref_normed = ref / np.linalg.norm(ref)
   fidelity = state_fidelity(solution_hhl_normed, solution_ref_normed)
   return fidelity
```

C.1. HHL

Running HHL

Listing C.2: Function to run HHL on any A

```
def HHLRUN(orig_size, matrixname, Amatrix, bvector, FileName,
        noise = False,
        Nmodel = [],
        Bgates = [],
        num_ancillae = 3,
        negative_evals = False,
        expansion_mode = 'suzuki',
        num_time_slices = 50,
```

```
     order = 1):
'''
Parameters
----------
orig_size : Size of matrix
matrixname : Name of matrix
Amatrix : Matrix A
bvector : Vector b
FileName : Filename to output too
noise : Boolian for running noisy model, optional, the default is False.
num_ancillae : Number of Ancillae bits, optional, the default is 3.
negative_evals : negative evaluations, optional The default is False.
expansion_mode : Expansiton method, optional, the default is 'suzuki'.
num_time_slices : Timeslices in expansion method, optional, default is 50.
order : Order of expansion, optional, the default is 1.

Returns
-------
Text document with information on algorithm runtime and fidelity

'''
# Check if matrix is Hermitian, resize it if neccisary
Amatrix, vector, truncate_powerdim, truncate_hermitian =
    HHL.matrix_resize(Amatrix, bvector)

# Initialize eigenvalue finding module
# Check for negative evaluation and add to the number of ancillae if there are
    any
ne_qfts = [None, None]
if negative_evals:
   num_ancillae += 1
   ne_qfts = [QFT(num_ancillae - 1), QFT(num_ancillae - 1).inverse()]

eigs = EigsQPE(MatrixOperator(matrix=Amatrix),
               QFT(num_ancillae).inverse(),
               num_time_slices=num_time_slices,
               num_ancillae=num_ancillae,
               expansion_mode='suzuki',
               expansion_order = order,
               evo_time=None,
               negative_evals=negative_evals,
               ne_qfts=ne_qfts)

# Get total number of qubtis and ancillae
num_q, num_a = eigs.get_register_sizes()

# Initialize initial state module
init_state = Custom(num_q, state_vector=bvector)

# Initialize reciprocal rotation module
reciprocal = LookupRotation(negative_evals=eigs._negative_evals,
    evo_time=eigs._evo_time)

tic = time.perf_counter()
HHL_Setup = HHL(Amatrix, bvector, truncate_powerdim, truncate_hermitian,
            eigs, init_state, reciprocal, num_q, num_a, orig_size)
if (noise):
   HHL_Setup.set_backend(Aer.get_backend('qasm_simulator'),
       basis_gates=Bgates, noise_model=Nmodel)
else:
   HHL_Setup.set_backend(Aer.get_backend('statevector_simulator'))
```

```
HHL_output = HHL_Setup.run()
toc = time.perf_counter()
# Calculate classical results and fidelity with quantum solution
result_classical = NumPyLSsolver(Amatrix, vector).run()
Fi = fidelity(HHL_output['solution'], result_classical['solution'])

# Export results as a single string
INFO = ["\n -------------- \n n size,", orig_size, "\n",
      "A Matrix, ", matrixname, "\n",
      "solution,", np.round(HHL_output['solution'], 5).tolist(), "\n",
      "classical solution, ", np.round(result_classical['solution'], 5), "\n",
      "Fidelity,", Fi ,"\n",
      "Time to process,", toc - tic, "\n",
      "circuit_width,", HHL_output['circuit_info']['width'], "\n"
      "circuit_depth,", HHL_output['circuit_info']['depth']]

# Write the information into a file
WritetoFile(FileName, INFO)
```

Noise Model

For making the noise version of this same code one needs to only define a noise model and set feed it to the Qasm simulator; here I import the model from IBMQ of the Melbourne 16 qubit machine.

```
from qiskit.providers.aer.noise import NoiseModel
import qiskit.providers.aer.noise as noise
provider = IBMQ.load_account()
backend = provider.get_backend('ibmq_16_melbourne')
noise_model = NoiseModel.from_backend(backend)

# Set coupling map from backend
coupling_map = backend.configuration().coupling_map

# Set basis gates from noise model
basis_gates = noise_model.basis_gates
```

The resulting data from the noise model at the time of the experiments, 20.12.20, is listed in C.3

Listing C.3: Noise model

```
NoiseModel:
Basis gates: ['cx', 'id', 'sx', 'u3', 'x']
Instructions with noise: ['sx', 'x', 'id', 'measure', 'cx']
Qubits with noise: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Specific qubit errors:
[('id', [0]), ('id', [1]), ('id', [2]), ('id', [3]), ('id', [4]), ('id', [5]),
    ('id', [6]), ('id', [7]), ('id', [8]), ('id', [9]), ('id', [10]), ('id',
    [11]), ('id', [12]), ('id', [13]), ('id', [14]), ('sx', [0]), ('sx', [1]),
    ('sx', [2]), ('sx', [3]), ('sx', [4]), ('sx', [5]), ('sx', [6]), ('sx',
    [7]), ('sx', [8]), ('sx', [9]), ('sx', [10]), ('sx', [11]), ('sx', [12]),
    ('sx', [13]), ('sx', [14]), ('x', [0]), ('x', [1]), ('x', [2]), ('x', [3]),
    ('x', [4]), ('x', [5]), ('x', [6]), ('x', [7]), ('x', [8]), ('x', [9]),
    ('x', [10]), ('x', [11]), ('x', [12]), ('x', [13]), ('x', [14]), ('cx',
    [14, 0]), ('cx', [0, 14]), ('cx', [14, 13]), ('cx', [13, 14]), ('cx', [6,
    8]), ('cx', [8, 6]), ('cx', [5, 9]), ('cx', [9, 5]), ('cx', [4, 10]),
    ('cx', [10, 4]), ('cx', [11, 3]), ('cx', [3, 11]), ('cx', [12, 2]), ('cx',
    [2, 12]), ('cx', [13, 1]), ('cx', [1, 13]), ('cx', [13, 12]), ('cx', [12,
    13]), ('cx', [10, 11]), ('cx', [11, 10]), ('cx', [9, 10]), ('cx', [10, 9]),
    ('cx', [9, 8]), ('cx', [8, 9]), ('cx', [7, 8]), ('cx', [8, 7]), ('cx', [5,
    6]), ('cx', [6, 5]), ('cx', [5, 4]), ('cx', [4, 5]), ('cx', [4, 3]), ('cx',
```

```
        [3, 4]), ('cx', [2, 3]), ('cx', [3, 2]), ('cx', [1, 2]), ('cx', [2, 1]),
        ('cx', [1, 0]), ('cx', [0, 1]), ('cx', [11, 12]), ('cx', [12, 11]),
        ('measure', [0]), ('measure', [1]), ('measure', [2]), ('measure', [3]),
        ('measure', [4]), ('measure', [5]), ('measure', [6]), ('measure', [7]),
        ('measure', [8]), ('measure', [9]), ('measure', [10]), ('measure', [11]),
        ('measure', [12]), ('measure', [13]), ('measure', [14])]
```

C.2. VQLS

Decomposing A into Pauli gates [15].

Listing C.4: decomposition into Pauli gates

```
def HS(M1, M2):
    """Hilbert-Schmidt-Product of two matrices M1, M2"""
    return (np.dot(M1.conjugate().transpose(), M2)).trace()

def c2s(c):
    """Return a string representation of a complex number c"""
    if c == 0.0:
        return "0"
    if c.imag == 0:
        return "%g" % c.real
    elif c.real == 0:
        return "%gj" % c.imag
    else:
        return "%g+%gj" % (c.real, c.imag)
```

Listing C.5: 2x2 matrix decomposition into Pauli gates

```
def decompose2(H,qbit):
    """Decompose Hermitian 2x2 matrix H into Pauli Gates"""
    if type(H) is np.ndarray:
        W = H
    else:
        W = np.array(H)
    size = W.shape[1]

    gates = []
    coef = []
    # Setup basis gates
    X = np.array([[0, 1], [ 1, 0]], dtype=np.complex128)
    Y = np.array([[0, -1j],[1j, 0]], dtype=np.complex128)
    Z = np.array([[1, 0], [0, -1]], dtype=np.complex128)
    I = np.array([[1, 0], [ 0, 1]], dtype=np.complex128)
    S = [I, Z, X, Y,]
    labels = ['I', 'Z', 'X', 'Y' ]

    for i in range(4):
        label = labels[i]
        a_i = 0.5 * HS(S[i], W)
        if a_i != 0:
            coef.append(a_i)
            gates.append([i])

    return coef,gates
```

Listing C.6: 4x4 matrix decomposition into Pauli gates

```
def decompose4(H):
    """Decompose Hermitian 4x4 matrix H into Pauli matrices"""

    if type(H) is np.ndarray:
        W = H
    else:
        W = np.array(H)
    size = W.shape[1]
    gates = []
    coef = []
    # Setup basis gates
    X = np.array([[0, 1], [ 1, 0]], dtype=np.complex128)
    Y = np.array([[0, -1j],[1j, 0]], dtype=np.complex128)
    Z = np.array([[1, 0], [0, -1]], dtype=np.complex128)
    I = np.array([[1, 0], [ 0, 1]], dtype=np.complex128)
    S = [I, Z, X, Y,]
    #labels = ['I', 'Z', 'X', 'Y' ]
    for i in range(4):
        for j in range(4):
            a_ij = 0.25 * HS(np.kron(S[i],S[j]), W)
            if a_ij != 0:
                coef.append(a_ij)
                gates.append([i,j])

    return coef,gates
```

Listing C.7: 8x8 matrix decomposition into Pauli gates

```
def decompose8(H):
    """Decompose Hermitian 8x8 matrix H into Pauli matrices"""
    if type(H) is np.ndarray:
        W = H
    else:
        W = np.array(H)

    gates = []
    coef = []
    # Setup basis gates
    X = np.array([[0, 1], [ 1, 0]], dtype=np.complex128)
    Y = np.array([[0, -1j],[1j, 0]], dtype=np.complex128)
    Z = np.array([[1, 0], [0, -1]], dtype=np.complex128)
    I = np.array([[1, 0], [ 0, 1]], dtype=np.complex128)
    S = [I, Z, X, Y,]
    labels = ['I', 'Z', 'X', 'Y' ]
    for i in range(4):
        for j in range(4):
            for k in range(4):
                label = labels[i]+labels[j]+labels[k]
                a_ij = 1/8 * HS(np.kron(np.kron(S[i],S[j]),S[k]), W)
                if a_ij != 0:
                    coef.append(a_ij)
                    gates.append([i,j,k])
    return coef,gates
```

Vector b

Making the unitary U from the vector b using a conditional Hadamard gate; this function is the same for all sizes of A.

Listing C.8: Control function for making b

```
def control_b(ancilla, qubits):
    for ia in qubits:
        circ.ch(ancilla, ia)
```

Ansatz

A fixed hardware ansatz was used for all cases. Each size category needed a tailored ansatz, cf. Chapter 4.4.

Listing C.9: Ansatz for 2x2 cases.

```
def apply_fixed_ansatz2(circ, qubits, parameters):
    circ.ry(parameters[0][0], qubits[0])
```

Listing C.10: Ansatz for 4x4 cases.

```
def apply_fixed_ansatz4(circ, qubits, parameters):
    #print(parameters)
    for iz in range(len(qubits)):
        circ.ry(parameters[0][iz], qubits[iz])

    circ.cz(qubits[0], qubits[1])

    for iz in range(len(qubits)):
        circ.ry(parameters[1][iz], qubits[iz])
```

Listing C.11: Ansatz for 8x8 cases.

```
def apply_fixed_ansatz8(circ,qubits, parameters):

    for iz in range(0, len(qubits)):
        circ.ry(parameters[0][iz], qubits[iz])

    circ.cz(qubits[0], qubits[1])
    circ.cz(qubits[2], qubits[0])

    for iz in range(0, len(qubits)):
        circ.ry(parameters[1][iz], qubits[iz])

    circ.cz(qubits[1], qubits[2])
    circ.cz(qubits[2], qubits[0])

    for iz in range(0, len(qubits)):
        circ.ry(parameters[2][iz], qubits[iz])
```

Control ansatz

Listing C.12: Control Ansatz for 2x2 cases.

```
def control_fixed_ansatz2(circ, qubits, parameters, ancilla, reg):
    nrqubits = len(qubits)
    circ.cry(parameters[0][0], qiskit.circuit.Qubit(reg, ancilla),
        qiskit.circuit.Qubit(reg, qubits[0]))
    circ.ccx(ancilla, qubits[0], nrqubits+1)
    circ.cz(qubits[0], nrqubits+1)
    circ.ccx(ancilla, qubits[0], nrqubits+1)
```

Listing C.13: Control Ansatz for 4x4 cases.

```
def control_fixed_ansatz4(circ, qubits, parameters, ancilla):
   nrqubits = len(qubits)
   for i in range(nrqubits):
      circ.cry(parameters[0][i], ancilla, qubits[i])
   circ.ccx(ancilla, qubits[1], nrqubits+1)
   circ.cz(qubits[0], nrqubits+1)
   circ.ccx(ancilla, qubits[1], nrqubits+1)
   for i in range(nrqubits):
      circ.cry(parameters[1][i], ancilla, qubits[i])
```

Listing C.14: Control Ansatz for 8x8 cases.

```
def control_fixed_ansatz8(circ,qubits, parameters, ancilla, reg):

   for i in range(0, len(qubits)):
      circ.cry(parameters[0][i], qiskit.circuit.Qubit(reg, ancilla),
             qiskit.circuit.Qubit(reg, qubits[i]))
   circ.ccx(ancilla, qubits[1], 4)
   circ.cz(qubits[0], 4)
   circ.ccx(ancilla, qubits[1], 4)
   circ.ccx(ancilla, qubits[0], 4)
   circ.cz(qubits[2], 4)
   circ.ccx(ancilla, qubits[0], 4)
   for i in range(0, len(qubits)):
      circ.cry(parameters[1][i], qiskit.circuit.Qubit(reg, ancilla),
             qiskit.circuit.Qubit(reg, qubits[i]))
   circ.ccx(ancilla, qubits[2], 4)
   circ.cz(qubits[1], 4)
   circ.ccx(ancilla, qubits[2], 4)
   circ.ccx(ancilla, qubits[0], 4)
   circ.cz(qubits[2], 4)
   circ.ccx(ancilla, qubits[0], 4)
   for i in range(0, len(qubits)):
      circ.cry(parameters[2][i], qiskit.circuit.Qubit(reg, ancilla),
             qiskit.circuit.Qubit(reg, qubits[i]))
```

Hadamard tests

Hadamard tests and the control Hadamard test, which uses a control sequence on the ansatz, are not size depended except for needing the correct fixed ansatz functions to work correctly for each size.

Listing C.15: Hadamard tests for all cases.

```
def had_test(circ,gate_type, qubits, ancilla_index, parameters):
   circ.h(ancilla_index)
   apply_fixed_ansatz(circ, qubits, parameters)
   for ie in range (0, len(gate_type[0])):
      if (gate_type[0][ie] == 1):
         circ.cz(ancilla_index, qubits[ie])
   for ie in range (0, len(gate_type[1])):
      if (gate_type[1][ie] == 1):
         circ.cz(ancilla_index, qubits[ie])
   circ.h(ancilla_index)

def special_had_test(circ, gate_type, qubits, ancilla_index, parameters, reg):
   circ.h(ancilla_index)
   control_fixed_ansatz(circ,qubits, parameters, ancilla_index, reg)
   for ty in range (0, len(gate_type)):
      if (gate_type[ty] == 1):
```

```
            circ.cz(ancilla_index, qubits[ty])
    control_b(circ,ancilla_index, qubits)
    circ.h(ancilla_index)
```

Cost function

The cost function is the function that the classical minimizer calls in order to run a loop. Each different size model needs a different function because we built the circuits differently and therefore require a different number of input parameters for each different ansatz.

Listing C.16: Cost function for 2x2 cases.

```
    global opt
    overall_sum_1 = 0
    parameters = [inparameters[0:2]]
    for i in range(0, len(gate_set)):
        for j in range(0, len(gate_set)):
            global circ
            qctl = QuantumRegister(3)
            qc = ClassicalRegister(3)
            circ = QuantumCircuit(qctl, qc)
            backend = Aer.get_backend('statevector_simulator')

            multiply = coefficient_set[i]*coefficient_set[j]
            had_test2(circ,[gate_set[i], gate_set[j]], [1], 0, parameters)
            job = execute(circ, backend)
            result = job.result()
            outputstate = np.real(result.get_statevector(circ, decimals=100))
            o = outputstate
            m_sum = 0
            for l in range (0, len(o)):
                if (l%2 == 1):
                    n = o[l]**2
                    m_sum+=n
            overall_sum_1+=multiply*(1-(2*m_sum))
    overall_sum_2 = 0

    for i in range(0, len(gate_set)):
        for j in range(0, len(gate_set)):
            multiply = coefficient_set[i]*coefficient_set[j]
            mult = 1

            for extra in range(0, len(gate_set)):
                qctl = QuantumRegister(3)
                qc = ClassicalRegister(3)
                circ = QuantumCircuit(qctl, qc)
                backend = Aer.get_backend('statevector_simulator')

                if (extra == 0):
                    special_had_test2(circ, gate_set[i], [1], 0, parameters, qctl)
                if (extra == 1):
                    special_had_test2(circ, gate_set[j], [1], 0, parameters, qctl)

                job = execute(circ, backend)
                result = job.result()
                outputstate = np.real(result.get_statevector(circ, decimals=100))
                o = outputstate

                m_sum = 0
                for l in range (0, len(o)):
                    if (l%2 == 1):
```

```
                n = o[l]**2
                m_sum+=n
            mult = mult*(1-(2*m_sum))
        overall_sum_2+=multiply*mult

    return 1-float(overall_sum_2/overall_sum_1)
```

Listing C.17: Cost function for 4x4 cases.

```
def calculate_cost_function4(inparameters, gate_set, coefficient_set):
    psi_psi = 0
    parameters = [inparameters[0:2], inparameters[2:4]]

    for i in range(0,len(gate_set)):
        for j in range(0,len(gate_set)):
            global circ
            qctl = QuantumRegister(4)
            qc = ClassicalRegister(4)
            circ = QuantumCircuit(qctl, qc)
            backend = Aer.get_backend('statevector_simulator')
            multiply = coefficient_set[i]*coefficient_set[j]
            mult = 1
            had_test4(circ, [gate_set[i], gate_set[j]], [1, 2], 0, parameters)
            job = execute(circ, backend)
            result = job.result()
            output = np.real(result.get_statevector(circ, decimals=100))
            out_beta_ll = output
            m_sum = 0
            for l in range(len(out_beta_ll)):
                if (l % 2 == 1):
                    n = out_beta_ll[l]**2
                    m_sum += n
            mult = mult*(1-(2*m_sum))
            psi_psi += multiply*mult
    b_psi_qure = 0

    for i in range(len(gate_set)):
        for j in range(len(gate_set)):
            multiply = coefficient_set[i]*coefficient_set[j]
            mult = 1
            for extra in range(0, len(gate_set)):
                qctl = QuantumRegister(4)
                qc = ClassicalRegister(4)
                circ = QuantumCircuit(qctl, qc)
                backend = Aer.get_backend('statevector_simulator')

                if (extra == 0):
                    special_had_test4(circ, gate_set[i], [1, 2], 0, parameters)

                if (extra == 1):
                    special_had_test4(circ, gate_set[j], [1, 2], 0, parameters)

                job = execute(circ, backend)
                result = job.result()
                output = np.real(result.get_statevector(circ, decimals=100))
                out_gamma_ll = output

                m_sum = 0
                for l in range(len(out_gamma_ll)):
                    if (l % 2 == 1):
                        n = out_gamma_ll[l]**2
```

```
                m_sum+=n
            mult = mult*(1-(2*m_sum))
        b_psi_qure += multiply*mult
    return 1-float(b_psi_qure/psi_psi)
```

Listing C.18: Cost function for 8x8 cases.

```
def calculate_cost_function8(parameters,gate_set, coefficient_set):
    global opt
    overall_sum_1 = 0
    parameters = [parameters[0:3], parameters[3:6], parameters[6:9]]
    for i in range(len(gate_set)):
        for j in range(len(gate_set)):
            global circ
            qctl = QuantumRegister(5)
            qc = ClassicalRegister(5)
            circ = QuantumCircuit(qctl, qc)
            backend = Aer.get_backend('statevector_simulator')
            multiply = coefficient_set[i]*coefficient_set[j]
            had_test8(circ,[gate_set[i], gate_set[j]], [1, 2, 3], 0, parameters)
            job = execute(circ, backend)
            result = job.result()
            outputstate = np.real(result.get_statevector(circ, decimals=100))
            o = outputstate
            m_sum = 0
            for l in range(0, len(o)):
                if (l % 2 == 1):
                    n = o[l]**2
                    m_sum += n

            overall_sum_1 += multiply*(1-(2*m_sum))
    overall_sum_2 = 0

    for i in range(len(gate_set)):
        for j in range(len(gate_set)):
            multiply = coefficient_set[i]*coefficient_set[j]
            mult = 1
            for extra in range(0,2):
                qctl = QuantumRegister(5)
                qc = ClassicalRegister(5)
                circ = QuantumCircuit(qctl, qc)
                backend = Aer.get_backend('statevector_simulator')
                if (extra == 0):
                    special_had_test8(circ,gate_set[i], [1, 2, 3], 0,
                                parameters, qctl)
                if (extra == 1):
                    special_had_test8(circ,gate_set[j], [1, 2, 3], 0,
                                parameters, qctl)
                job = execute(circ, backend)
                result = job.result()
                outputstate = np.real(result.get_statevector(circ, decimals=100))
                o = outputstate
                m_sum = 0
                for l in range(0,len(o)):
                    if (l % 2 == 1):
                        n = o[l]**2
                        m_sum += n
                mult = mult*(1-(2*m_sum))
            overall_sum_2 += multiply*mult
    return 1-float(overall_sum_2/overall_sum_1)
```

Cost function in a noisy setting

As the cost function sets up the quantum circuits it is the only function we need to change to apply a noise simulation. In Listing C.19 we see an example of such a change for the 2×2 case, each case requires the same change to its cost function. This mainly involves the sampling of multiple test shots and averaging the results.

Listing C.19: Cost function for 8x8 cases.

```
def calculate_cost_function2(inparameters, gate_set, coefficient_set,Nmodel,
    Bgates):
   global opt
   overall_sum_1 = 0
   parameters = [inparameters[0:2]]
   backend = Aer.get_backend('qasm_simulator', basis_gates=Bgates,
       noise_model=Nmodel)
   for i in range(0, len(gate_set)):
      for j in range(0, len(gate_set)):
         global circ
         qctl = QuantumRegister(3)
         qc = ClassicalRegister(3)
         circ = QuantumCircuit(qctl, qc)
         multiply = coefficient_set[i]*coefficient_set[j]
         had_test2(circ,[gate_set[i], gate_set[j]], [1], 0, parameters)
         circ.measure(0, 0)
         job = execute(circ, backend, shots=10000)
         result = job.result()
         outputstate = result.get_counts(circ)
         m_sum = 0
         if ('1' in outputstate.keys()):
            m_sum = float(outputstate["1"])/100000
         else:
            m_sum = 0

         overall_sum_1+=multiply*(1-(2*m_sum))

   overall_sum_2 = 0
   for i in range(0, len(gate_set)):
      for j in range(0, len(gate_set)):
         multiply = coefficient_set[i]*coefficient_set[j]
         mult = 1
         for extra in range(0, len(gate_set)):
            qctl = QuantumRegister(3)
            qc = ClassicalRegister(3)
            circ = QuantumCircuit(qctl, qc)
            if (extra == 0):
               special_had_test2(circ, gate_set[i], [1], 0, parameters, qctl)
            if (extra == 1):
               special_had_test2(circ, gate_set[j], [1], 0, parameters, qctl)
            circ.measure(0,0)
            job = execute(circ, backend, shots=10000)
            result = job.result()
            outputstate = result.get_counts(circ)
            if ('1' in outputstate.keys()):
               m_sum = float(outputstate["1"])/10000
            else:
               m_sum = 0
            mult = mult*(1-(2*m_sum))
         overall_sum_2+=multiply*mult
   return 1-float(overall_sum_2/overall_sum_1)
```

Running VQLS

To run the full VQLS algorithm, a function is used that handles calling the minimizer, COBYLA, and exporting time and resulting data to a text file.

Listing C.20: A function which runs the full VQLS algorithm on a given matrix A and vector b.

```
def VQLSfun_8(inputmatrix, matrixname, b, FileName):
    # Set the input matrix as an np array
    Amatrix = np.array(inputmatrix)

    #Decompose A into Pauli gates
    coefficients,gates = decompose8(Amatrix)

    # Run the classical minimizer with the cost function circuit as an input
        function

    tic = time.perf_counter()
    out = minimize(calculate_cost_function8,
                x0=[float(random.randint(0, 3000))/1000 for i in range(9)],
                args=(gates, coefficients), method="COBYLA", options={'maxiter':
                    200})
    toc = time.perf_counter()

    out_f = [out['x'][0:3], out['x'][3:6], out['x'][6:9]]
    circ2 = QuantumCircuit(3, 3)
    apply_fixed_ansatz8(circ2,[0, 1, 2], out_f)
    backend = Aer.get_backend('statevector_simulator')
    job = execute(circ2, backend)
    result = job.result()
    o = result.get_statevector(circ2, decimals=10)

    result_classical = NumPyLSsolver(Amatrix, b).run()
    fid = fidelity(o, result_classical['solution'])

    # Export results as a single string
    INFO = ["\n -------------- \n n size,", size, "\n",
          "A Matrix, ", matrixname, "\n",
          "solution,", np.round(o, 5).tolist(), "\n",
          "classical solution, ", np.round(result_classical['solution'], 5), "\n",
          "Fidelity,", fid ,"\n",
          "Time to process,", toc - tic, "\n"]

    # Write the information into a file
    WritetoFile(FileName, INFO)
```

D

Test Matrices

Here all test matrices used in experiments are listed together with their condition numbers and sparsity. For each test matrix the condition number, κ, is calculated as the ratio of the largest and the smallest eigenvalue, $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$. Sparsity can be defined in a plethora of ways, here we go by the definition used in the HHL paper where the sparsity, s, equals the maximum number of nonzero entries per row [17].

The test matrices were chosen such that they represent a variety of condition numbers and sparsity numbers for each size category. The structure generally goes from less complex to more, starting with the identity matrix. All sizes include a tri-diagonal matrix as these are common in practical problems involving discretization. They also all include three matrices that are *ill-conditioned*, e.g. $4.d$,$4.e$, and, $4.f$. The matrices $8.d$, $4.c$ are diagonal matrices with randomly generated values while $2.j$, $4.i$, $4.j$ are randomly generated dense Hermitian matrices. The random values for these matrices were determined using the Online Math Tools random matrix generator [1].

Experimental matrices of size 2×2

$$2.a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \kappa = 1.0, \qquad s = 1$$

$$2.b = \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix}, \qquad \kappa = 2.0, \qquad s = 1$$

$$2.c = \begin{bmatrix} 1.5 & 0 \\ 0 & 0.5 \end{bmatrix}, \qquad \kappa = 3.0, \qquad s = 1$$

$$2.d = \begin{bmatrix} 1 & 0 \\ 0 & 0.1 \end{bmatrix}, \qquad \kappa = 10.0, \qquad s = 1$$

$$2.e = \begin{bmatrix} 1 & 0 \\ 0 & 0.01 \end{bmatrix}, \qquad \kappa = 100.0, \qquad s = 1$$

$$2.f = \begin{bmatrix} 1 & 0 \\ 0 & 0.001 \end{bmatrix}, \qquad \kappa = 1000.0, \qquad s = 1$$

$$2.g = \begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}, \qquad \kappa = 3.0, \qquad s = 2$$

$$2.h = \begin{bmatrix} 1 & 0.333 \\ 0.333 & 1 \end{bmatrix}, \qquad \kappa = 2.0, \qquad s = 2$$

$$2.i = \begin{bmatrix} 1 & 0.2 \\ 0.2 & 1 \end{bmatrix}, \qquad \kappa = 1.5, \qquad s = 2$$

$$2.j = \begin{bmatrix} 0.69 & 0.49 \\ 0.49 & 0.92 \end{bmatrix}, \qquad \kappa = 4.337, \qquad s = 2$$

$$2.k = \begin{bmatrix} 0.5 & 1 \\ 1 & 0.5 \end{bmatrix}, \qquad \kappa = 3.0, \qquad s = 2$$

$$2.l = \begin{bmatrix} 0.75 & 0.5 \\ 0.5 & 0.75 \end{bmatrix}, \qquad \kappa = 5.0, \qquad s = 2$$

Experimental matrices of size 4×4

$$4.a = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \kappa = 1.0, \qquad s = 1$$

$$4.b = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix}, \qquad \kappa = 2.0, \qquad s = 1$$

$$4.c = \begin{bmatrix} 0.97 & 0 & 0 & 0 \\ 0 & 0.81 & 0 & 0 \\ 0 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0.13 \end{bmatrix}, \qquad \kappa = 7.462, \qquad s = 1$$

$$4.d = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix}, \qquad \kappa = 10.0, \qquad s = 1$$

$$4.e = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.01 \end{bmatrix}, \qquad \kappa = 100.0, \qquad s = 1$$

$$4.f = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.001 \end{bmatrix}, \qquad \kappa = 1000.0, \qquad s = 1$$

$$4.g = \begin{bmatrix} 1 & -0.5 & 0 & 0 \\ -0.5 & 1 & -0.5 & 0 \\ 0 & -0.5 & 1 & -0.5 \\ 0 & 0 & -0.5 & 1 \end{bmatrix}, \qquad \kappa = 9.472, \qquad s = 3$$

$$4.h = \begin{bmatrix} 1 & 0.333 & 0 & 0 \\ 0.333 & 1 & 0.333 & 0 \\ 0 & 0.333 & 1 & 0.333 \\ 0 & 0 & 0.333 & 1 \end{bmatrix}, \qquad \kappa = 3.342, \qquad s = 3$$

$$4.i = \begin{bmatrix} 0.34 & 0 & 0.63 & 0 \\ 0 & 0.63 & 0.57 & 0.16 \\ 0.63 & 0.57 & 0.2 & 0 \\ 0 & 0.16 & 0 & 0.2 \end{bmatrix}, \qquad \kappa = 7.695, \qquad s = 3$$

$$4.j = \begin{bmatrix} 0.99 & 0.37 & 0.31 & 0 \\ 0.37 & 0.32 & 0.15 & 0.15 \\ 0.31 & 0.15 & 0.6 & 0.21 \\ 0 & 0.15 & 0.21 & 0.73 \end{bmatrix}, \qquad \kappa = 11.157, \qquad s = 4$$

Experimental matrices of size 8×8

$$8.a = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \kappa = 1.0, \qquad s = 1$$

$$8.b = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 \end{bmatrix}, \qquad \kappa = 2.0, \qquad s = 1$$

$$8.c = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 \end{bmatrix}, \qquad \kappa = 2.0, \qquad s = 1$$

$$8.d = \begin{bmatrix} 0.09 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.79 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.75 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.05 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.09 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.21 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.53 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.92 \end{bmatrix}, \qquad \kappa = 18.4, \qquad s = 1$$

$$8.e = \begin{bmatrix} 1 & 0.333 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.333 & 1 & 0.333 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.333 & 1 & 0.333 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.333 & 1 & 0.333 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.333 & 1 & 0.333 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.333 & 1 & 0.333 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.333 & 1 & 0.333 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.333 & 1 \end{bmatrix}, \qquad \kappa = 4.354, \qquad s = 3$$

$$8.f = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 \end{bmatrix}, \qquad \kappa = 10.0, \qquad s = 1$$

$$8.g = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 \end{bmatrix}, \qquad \kappa = 100.0, \qquad s = 1$$

$$8.h = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.001 \end{bmatrix}, \qquad \kappa = 1000.0, \qquad s = 1$$

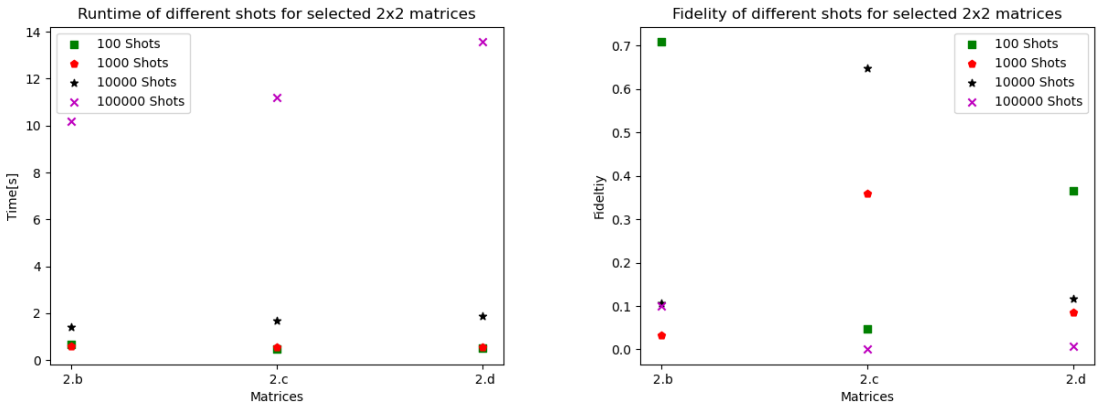
E

Trials for Experimental Setup

The following are trials run to support the chosen experimental setup.

E.1. Varying Number of Shots

For completing the VQLS noisy algorithm we use a number of shots in each run of the quantum circuits. The number of shots used in the general tests was determined by the experiment to be 10.000. The experiment tested, 100, 1.000, 10.000, and 100.000 shots for matrices $2.b$,$2.c$, and $2.d$ and the results can be found in Figure E.1. Viewing the runtime we can see that the increase from 10.000 steps to 100.000 increases the runtime from under 2 seconds to over 10 seconds. This fivefold increase in time does not marginally increase fidelity so the 10.000 shots case is considered the best option.

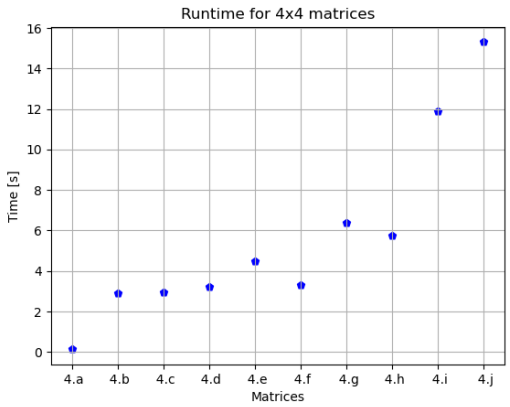


(a) Runtimes for selected 2×2 matrices with varying shot numbers. (b) Fidelity for selected 2×2 matrices with varying shot numbers .

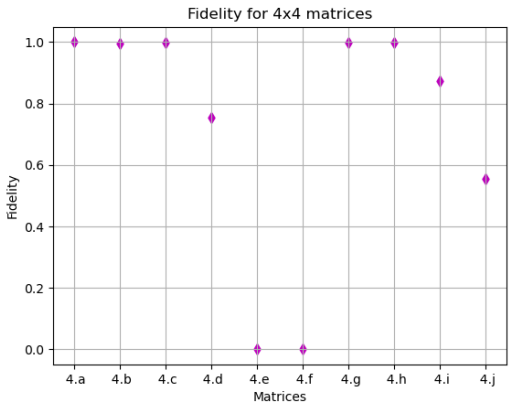
Figure E.1: Experiments with 2×2 matrices, state vector simulations.

E.2. Lloyd's Method Trial

For HHL there are two expansion methods set up in the Qiskit implementation of solving QPE. These are the method of Lloyd which is based on the Trotter expansion formula [38], and the method of Suzuki which is based on the generalized Trotter expansion called the Suzuki-Trotter expansion [36]. As explained in Chapter 4.1 the Suzuki-Trotter expansion corresponds to the Trotter expansion in the first expansion order. So all experiments were generally run under the same conditions as Lloyd's method just using the Suzuki setting. This can be verified by comparing Figure E.2 to Figure 4.2.



(a) Runtimes for 4×4 matrices with Lloyd's method.



(b) Fidelity for 4×4 matrices with Lloyd's method.

Figure E.2: Lloyd's method experiments using 4×4 matrices in state vector simulations.

Bibliography

[1] Random matrix generator. URL https://onlinemathtools.com/generate-random-matrix.

[2] Dorit Aharonov. Quantum computation. *Annual Reviews of Computational Physics VI*, page 259–346, Mar 1999. doi: 10.1142/9789812815569_0007. URL http://dx.doi.org/10.1142/9789812815569_0007.

[3] Andris Ambainis. Variable time amplitude amplification and a faster quantum algorithm for solving systems of linear equations, 2010.

[4] Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Nicholas Bronn, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Richard Chen, Albert Frisch, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, Francis Harkins, Takashi Imamichi, David McKay, Antonio Mezzacapo, Zlatko Minev, Ramis Movassagh, Giacomo Nannicni, Paul Nation, Anna Phan, Marco Pistoia, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, John Stenger, Kristan Temme, Madeleine Tod, Stephen Wood, and James Wootton. Learn quantum computation using qiskit, 2020. URL http://community.qiskit.org/textbook.

[5] Carlos Bravo-Prieto, Ryan LaRose, M. Cerezo, Yigit Subasi, Lukasz Cincio, and Patrick J. Coles. Variational Quantum Linear Solver: A Hybrid Algorithm for Linear Systems. sep 2019. URL http://arxiv.org/abs/1909.05820.

[6] X. D. Cai, Christian Weedbrook, Z. E. Su, M. C. Chen, Mile Gu, M. J. Zhu, L. Li, N. L. Liu, Chao-Yang Lu, and Jian-Wei Pan. Experimental Quantum Computing to Solve Systems of Linear Equations. feb 2013. doi: 10.1103/PhysRevLett.110.230501. URL http://arxiv.org/abs/1302.4310http://dx.doi.org/10.1103/PhysRevLett.110.230501.

[7] Graham Carlow. Quantum computers and accelerated discovery, 2018. URL https://newsroom.ibm.com/image-gallery?l=100&keywords=quantum#gallery_gallery_0:21747.

[8] Marco Cerezo, Kunal Sharma, Andrew Arrasmith, and Patrick J. Coles. Variational quantum state eigensolver, 2020.

[9] Andrew M. Childs, Robin Kothari, and Rolando D. Somma. Quantum algorithm for systems of linear equations with exponentially improved dependence on precision. *SIAM Journal on Computing*, 46(6):1920–1950, Jan 2017. ISSN 1095-7111. doi: 10.1137/16m1087072. URL http://dx.doi.org/10.1137/16M1087072.

[10] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017. doi: 10.1017/9781316219317.

[11] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open Quantum Assembly Language 1 Background. Technical report, 2017.

[12] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 439(1907): 553–558, dec 1992. ISSN 1364-5021. doi: 10.1098/rspa.1992.0167.

[13] Aïmeur Esma, Brassard Gilles, and Gambs Sébastien. Machine learning in a quantum world, 2006. URL https://link.springer.com/chapter/10.1007/11766247_37.

[14] Richard Feynman. The Feynman Lectures on Physics, 1963. URL https://www.feynmanlectures.caltech.edu/.

[15] Michael Goerz. Decomposing two-qubit hamiltonians into pauli matrices. URL https://michaelgoerz.net/notes/decomposing-two-qubit-hamiltonians-into-pauli-matrices.html.

[16] Lov K. Grover. A fast quantum mechanical algorithm for database search. 1996.

[17] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for solving linear systems of equations. nov 2009. doi: 10.1103/PhysRevLett.103.150502. URL http://arxiv.org/abs/0811.3171http://dx.doi.org/10.1103/PhysRevLett.103.150502.

[18] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M. Chow, and Jay M. Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, Sep 2017. ISSN 1476-4687. doi: 10.1038/nature23879. URL http://dx.doi.org/10.1038/nature23879.

[19] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum principal component analysis. *Nature Physics*, 10, 2014. ISSN 1745-2481. doi: 110.1038/nphys3029.

[20] Philippe Lock, Matt DeJong, and John Ochsendorf. As hangs the flexible line: Equilibrium of masonry arches. 8(2):p. 13–24, October 2006. URL http://web.mit.edu/masonry/papers/block_dejong_ochs_NNJ.pdf.

[21] Dirk Meijer. The universe as a cyclic organized information system. john wheeler's world revisited. *NeuroQuantology*, vol 13:pp 57–78„ 03 2015.

[22] Memetician. A different kind of string theory: Antoni Gaudi - memetician — LiveJournal, 2007. URL https://memetician.livejournal.com/201202.html.

[23] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum circuit learning. *Physical Review A*, 98(3), Sep 2018. ISSN 2469-9934. doi: 10.1103/physreva.98.032309. URL http://dx.doi.org/10.1103/PhysRevA.98.032309.

[24] Matthias Möller and Cornelis Vuik. A conceptual framework for quantum accelerated automated design optimization. *Microprocessors and Microsystems*, 66:67 – 71, 2019. ISSN 0141-9331. doi: https://doi.org/10.1016/j.micpro.2019.02.009. URL http://www.sciencedirect.com/science/article/pii/S0141933118303223.

[25] Ashley Montanaro and Sam Pallister. Quantum algorithms and the finite element method. *Physical Review A*, 93(3), Mar 2016. ISSN 2469-9934. doi: 10.1103/physreva.93.032324. URL http://dx.doi.org/10.1103/PhysRevA.93.032324.

[26] Chris Nay. Ibm delivers its highest quantum volume to date, expanding the computational power of its ibm cloud-accessible quantum computers. URL https://newsroom.ibm.com/.

[27] Michael Nielsen and Isaac Chuang. *Quantum Computation and Quantum Information*, volume 49. 2010. ISBN 978110700217.

[28] Jian Pan, Yudong Cao, Xiwei Yao, Zhaokai Li, Chenyong Ju, Xinhua Peng, Sabre Kais, and Jiangfeng Du. Experimental realization of quantum algorithm for solving linear systems of equations. feb 2011. doi: 10.1103/PhysRevA.89.022313. URL http://arxiv.org/abs/1302.1946http://dx.doi.org/10.1103/PhysRevA.89.022313.

[29] Marco Pistoia and Jay Gambetta. Qiskit aqua a library of quantum algorithms and applications, 12 2018. URL https://medium.com/qiskit/qiskit-aqua-a-library-of-quantum-algorithms-and-applications-33ecf3b36008.

[30] M. J. D. Powell. Direct search algorithms for optimization calculations. *Acta Numerica*, 7:287–336, 1998. doi: 10.1017/S0962492900002841.

[31] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 08 2018. ISSN 2521-327X. doi: 10.22331/q-2018-08-06-79. URL https://doi.org/10.22331/q-2018-08-06-79.

[32] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. Quantum support vector machine for big data classification. *Physical Review Letters*, 113(13), Sep 2014. ISSN 1079-7114. doi: 10.1103/physrevlett.113.130503. URL http://dx.doi.org/10.1103/PhysRevLett.113.130503.

[33] Eleanor Rieffel and Wolfgang Polak. *Quantum computing: a gentle introduction*, volume 49. 2011. ISBN 9780262015066. doi: 10.5860/choice.49-0911.

[34] Eleanor Rieffel and Wolfgang Polak. *Quantum computing: a gentle introduction*, volume 49. Massachusetts Institute of Technology, 2011. ISBN 9780262015066. doi: 10.5860/choice.49-0911.

[35] Benjamin Schumacher. *The Science of Information From Language to Black Holes*. The Great Courses, 2015. ISBN 9780198520115.

[36] Masuo Suzuki. Generalized trotter's formula and systematic approximants of exponential operators and inner derivations with applications to many-body problems. *Communications in Mathematical Physics*, 51, 1976. doi: 10.1007/BF01609348. URL https://doi.org/10.1007/BF01609348.

[37] Qiskit Development Team. Summary of quantum operations, 2020. URL https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html.

[38] H. F. Trotter. On the product of semi-groups of operators. *Proceedings American Mathematical Society*, 10:545–551, 1959. doi: https://doi.org/10.1090/S0002-9939-1959-0108732-6.

[39] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: https://doi.org/10.1112/plms/s2-42.1.230. URL https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

[40] Jingwei Wen, Xiangyu Kong, Shijie Wei, Bixue Wang, Tao Xin, and Guilu Long. Experimental realization of quantum algorithms for a linear system inspired by adiabatic quantum computing. *Physical Review A*, 99(1), jan 2019. ISSN 24699934. doi: 10.1103/PhysRevA.99.012320.

[41] Leonard Wossnig, Zhikuan Zhao, and Anupam Prakash. Quantum linear system algorithm for dense matrices. *Physical Review Letters*, 120(5), Jan 2018. ISSN 1079-7114. doi: 10.1103/physrevlett.120.050502. URL http://dx.doi.org/10.1103/PhysRevLett.120.050502.