Gradu Amaierako Lana/Trabajo Fin de Grado

Fisikako Gradua/Grado en Física

Digital-Analog Quantum Computing

Asier Galicia Martínez

Director:
Prof. Iñigo L. Egusquiza
Codirector:
Dr. Mikel Sanz

Department of Theoretical Physics and History of Science
Faculty of Science and Technology
University of the Basque Country UPV/EHU

Leioa, June 2019

Contents

Introduction and objectives

Computers have undoubtedly become a key tool during the past years. Nobody would argue against their utility or their big impact on our daily and scientific life. Fortunately, their computational power has sustainably increased along the decades, allowing us to tackle more and more difficult problems.

This exponential increment in the computational power is incorporated in Moore's law, which states that the number of transistors in a chip doubles every year. However, we can foresee a limit for this trend. The reason is that we are reaching to the atomic level and shrinking transistors has become more and more difficult due to overheating and the emergence of unwanted quantum effects.

Different new technologies have been proposed to overcome this difficulty, and quantum computing stands out among them. Until now, we used quantum mechanics mainly to understand, for instance, the behaviour of these transistors, among other technologies. Indeed, we have developed technologies such as lasers, transistors or superconductors, which can not be explained using classical physics. This event is currently known as the first quantum revolution. Nowadays, however, the challenge is to develop a new generation of technology able manipulate the quantum states of matter and employ quantum effects such as superposition and entanglement as a resource. This is the so called second quantum revolution, to which quantum computers belong.

Quantum computers aim at enhancing information processing using mainly the superposition principle, a purely quantum phenomena. Even though it is a blossomy technology, quantum computing is in its infancy. As a reference, the largest processor provided by IBM comprises 20 qubits.

Mimicking classical computation, digital quantum computers have become the holy Grail of quantum computation. The main advantage over other paradigms, such as analog quantum computing, is that they already have quantum error correction techniques, necessary for the scalability of quantum computing. However, these error correction techniques require high fidelities of the gates and a high amount of physical qubits to encode each logical qubit.

Taking into account how far we are from fulfilling these criteria, it seems appropriate in the short term to adapt the quantum computing paradigm to our current resources. This is what the digital-analog quantum computing (DAQC) paradigm tries to achieve. Even though we do not know yet whether DAQC paradigm admits quantum error correction techniques, it has the advantage of maximizing the use of the resources of current quantum computers.

This TFG focuses on the design of an algorithm that efficiently simulates the evolution of an arbitrary all-to-all (ATA) Ising Hamiltonian employing as a resource an homogeneous nearest-neighbour (NN) Ising Hamiltonian and single qubit rotations. From a physical point of view, we show how to enhance connections, by using only local external control (single qubit rotations) in such a way that it behaves as if all qubits were connected.

This work is divided in four chapters, where the first one is this introduction. We will continue in the second chapter reviewing some basic concepts of both classical and quantum computers. More precisely, we will understand why quantum computers are universal and how to implement the same functions that a classical computers can implement. In the third chapter, we will review two algorithms which clearly outperform their classical counterparts.

These algorithms are Shor's algorithm for factorizing numbers, which has an exponential speed-up over its classical counterpart, and Grover's algorithm, which can find an element in a list with a quadratic speed-up. The last chapter is devoted to DACQ. We explain its fundamentals and we also show the algorithm developed to simulate efficiently an Ising Hamiltonian using as main resource a nearest-neighbour Ising Hamiltonian under the DAQC paradigm.

Chapter 1

From classical to quantum computing

In this section we introduce the fundamentals of quantum computing, showing the main differences and advantages when compared against classical computing. This chapter is a review of the chapters 2 and 3 of Ref. [1].

We will start by reviewing some basic concepts of classical computing aiming at introducing the concept of classical reversible computation. After reviewing these concepts, we will introduce quantum computing and its main build blocks: qubits and unitary gates. We will prove the universality of quantum computing using as reference the exercise 4.3 of Ref. [1]. Finally, we will discuss the challenges that quantum state tomography imposes.

1.1 Classical irreversible computing

By classical computing we mean the computation done with classical resources. These are the bit, which is the basic unit of information, and the classical gates, which perform some predefined operations into a set of bits.

Through these work, we will not focus on the physical implementations of the resources aforementioned. Everything we need to consider is that the bit can only be in a clearly defined state (0 or 1) and that a gate is an operation done to a set of bits in order to transform them. We will think of classical gates as Boolean functions $f : \{0,1\}^n \to \{0,1\}^m$. Some of the most relevant gates are shown in Fig. 1.1. It is noteworthy to mention, that unless $n = m$, these gates are not reversible (the functions are not a bijection).

One question that rises when analysing these gates is whether it is possible to decompose any Boolean function f using a discrete set of gates. The answer is yes. In fact, the AND and

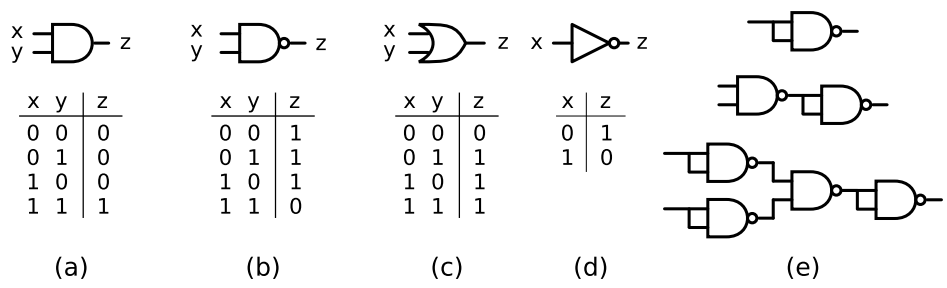
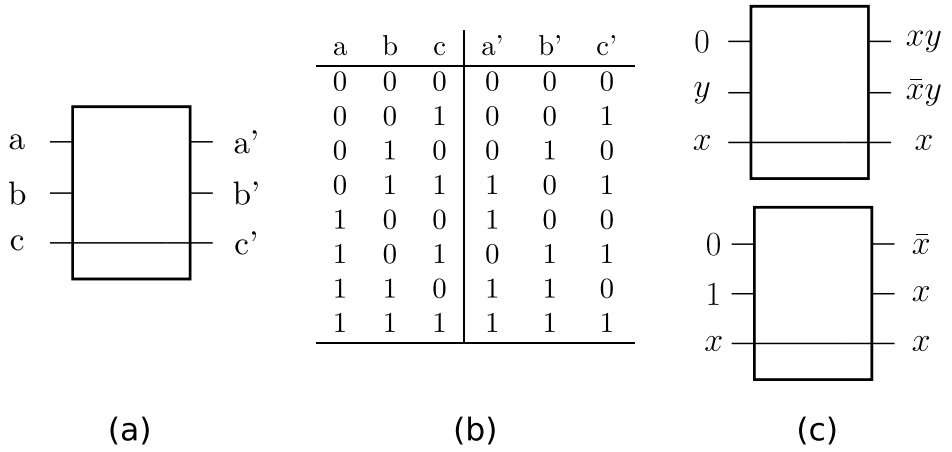


Figure 1.1: Circuit diagram of classical gates along with their truth table. The gates shown are the AND (a), NAND (b), OR (c) and the NOT (d). The last set of circuits shows the configuration of NAND gates to simulate, form top to bottom, the NOT, AND and OR gates.



a	b	c	a'	b'	c'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	1
1	0	0	1	0	0
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	1	1	1

Figure 1.2: Fredkin gate (a) with its truth table (b). In (c) we show the configuration of the Fredkin gate that simulates the AND (top) and NOT (bottom) irreversible gates.

NOT gates are, for instance, sufficient to generate any Boolean function. This is of utmost importance when designing circuits, it states that we only need to focus on combining those two gates and that a better performance of the circuit is founded upon a better performance of those elementary gates.

These two gates are by no means unique. For example, the NAND gate is itself able to simulate the gates NOT, AND and OR, as shown in the figure 1.1 (e)[1]. Taking this into account, the NAND gate could be used to implement any desired Boolean function, which means that the NAND gate is also universal.

1.2 Classical reversible computing

Quantum mechanics describes the evolution of a close system using reversible unitary operations and the measurement postulate. This unitary evolution of a physical system will be introduced later as a way of implementing a quantum gate, but the characteristic reversible evolution of quantum systems seems to encourage a deeper study of classical computation in terms of reversible gates.

In this section, we analyse classical computation using only reversible gates. New key concepts such as ancillary bits and "garbage" outputs will emerge. This will allow us to come up with a general approach to simulate any desired irreversible function using classical reversible computation.

An example of a classical reversible gate is depicted in the Fig. 1.2. Since Fredkin gate can simulate the AND and NOT gates, see Fig. 1.2 (c), it is an universal gate. In order to simulate the AND gate, we require to input a 0 bit in the first entry of the Fredkin gate. That predefined input value is called ancillary bit and it has the only purpose of making possible to the Fredkin gate to simulate an irreversible gate.

The simulation of the AND gate using the Fredkin gate produces two extra unwanted bits. We could completely ignore those extra bits in our computation. However, it is noteworthy to mention that within quantum computing paradigm, these bits could become a problem due to entanglement. We will call these extra output bits "garbage" bits and they usually appear when simulating a classical irreversible gate using only reversible gates.

In order to get rid of those garbage bits in an elegant way, we will show an algorithm that appears in Ref. [1]. The circuit of the algorithm is shown in Fig. 1.3. After applying

[1]Here and in the rest of the work, when we say that a set of gates A simulates another set of gates B, we will mean that gates of A can be combined in some way in order to perform the same function that the gates of B perform.

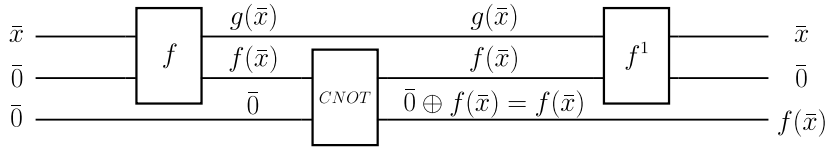


Figure 1.3: Classical reversible circuit employed to clean the garbage bits when simulating irreversible gates. $\bar{0}$ stands for a register containing only 0 bits while $\bar{x}$ holds a given string of bits. We start applying the reversible version of the desired function, we continue copying the data in a third register and we finally clean the registers using the inverse function.

all the gates, we end up having in the first two registers the same initial data. Besides, the last of our registers holds the desired output of the classical irreversible function $f(x)$. The core idea is to use the CNOT[2] gate to copy the desired output in an extra register. Then, we use the inverse of the reversible gate, which always exists, to undo the operation done in the first two registers.

To sum up, we have an algorithm that can implement a clean reversible version of any irreversible gate, for example, the NAND gate[3]. Hence, we are able to implement any irreversible gate. Additionally, classical reversible gates can simulate efficiently classical irreversible gates. By efficiency, we mean that we only need a polynomial number of reversible gates in order to simulate any irreversible function in a reversible manner. For example, the AND gate requires two Fredkin gates and a CNOT gate, plus some fixed number of ancillary bits. This idea of efficient simulation is important since it tell us how the number of reversible gates scales with the number of irreversible gates.

Now that we have seen that it is possible to use classical reversible gates to make universal computation, we are in a good position to introduce quantum computing.

1.3 Quantum computing

Quantum computing is a new framework for computation that aims to outperform classical computation, by exploiting the quantum properties of nature. In this paradigm, it is necessary to introduce the qubit, which is the basic unit of information.

1.3.1 The qubit

A qubit is the quantum analog of a classical bit. It is the representation of a physical system described by a Hilbert space of dimension 2, that is, a physical system that can be in two different states, generally denoted by $|0\rangle$ or $|1\rangle$. It can also be in a superposition of those two states $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and it is actually this superposition possibility, with some extra ingredients, which plays an important role in the development of quantum algorithms outperforming classical algorithms. A simple, thought not really practical, physical device that could be used as a qubit is an electron spin.

Any qubit can be represented by two real numbers in the form of,

$$|\Phi\rangle = \cos(\theta)\,|0\rangle + \sin(\theta)e^{i\phi}\,|1\rangle\,. \tag{1.1}$$

It may seem that the qubit can store an infinite amount of information due to its real values dependence. However, we need to measure the qubit in order to retrieve only partial

[2]It should be noted that we are allowing ourselves to use the CNOT gate in a perfect implementation in the sense that this gate will generate no garbage.

[3]This basically proves the universality of the reversible computation. For any desired irreversible function, take its decomposition in NAND gates and substitute all these gates with their reversible version, obtaining in that way the reversible version of the entire function.

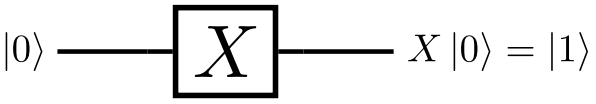


Figure 1.4: Example of quantum circuit where we apply a X gate to the state $|0\rangle$. The initial state is $|0\rangle$, the final is $|1\rangle$ and the wire stands for no evolution in time. The gate X is defined in Eq. (1.3).

information from the system. This forbids storing an infinite amount of information in a qubit.

1.3.2 Gates in quantum computing

The time evolution of these qubits is specified by a unitary operator acting on the state of the qubit. This unitary operator plays the role of a gate in quantum computation. In Fig. 1.4 we show a quantum circuit, the quantum analog of classical circuits. This scheme is read as follows: first, the qubit starts in the state $|0\rangle$. The wire stands for a trivial evolution in time, which means that the state of the qubit does not change. Afterwards, gate X acts on the state, changing it to $|1\rangle$.

In general, these gates are generated via some Hamiltonian that makes the system to evolve in time as $|\Phi(t)\rangle = e^{-itH}|\Phi(0)\rangle$. This is the analog approach and it has the disadvantage that both time and the Hamiltonian need to be specified. We generally only care about the unitary matrix, but there may be more than one way of generating it via different Hamiltonians and times. Furthermore, these implementations vary between different quantum computer architectures. That is why we will use the digital quantum computing paradigm (DQC). In this paradigm we only need to specify the unitary matrix we are using to evolve the qubits. However, we will later introduce digital-analog quantum computing (DAQC), which relies in the natural evolution of a system to specify analog multi-qubit gates. Meanwhile, it uses digital blocks to specify single qubit gates.

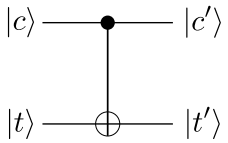
Let us now focus on the DQC paradigm, which is the most straightforward manner of introducing quantum computing.

Single Qubit Rotations

Let us start by reviewing the different operations that a quantum computer can perform to change the state of the qubits. In classical computing, the number of independent operations which we can make to a classical bit are limited: we are only allowed to apply the NOT and the trivial gate (the gate which does not change the classical bit). However, there is a wider variety of operations that can be preformed to a qubit, called single qubit rotations (SQR). Indeed, a qubit can be transformed using any 2×2 unitary matrix, which can be parametrized by only 4 real parameters, as depicted in Eq. (1.2),

$$U(\alpha,\beta,\gamma,\theta) = e^{i\alpha}\begin{pmatrix} e^{-i(\frac{\beta}{2}+\frac{\gamma}{2})}\cos\left(\frac{\theta}{2}\right) & -e^{-i(\frac{\beta}{2}-\frac{\gamma}{2})}\sin\left(\frac{\theta}{2}\right) \\ e^{i(\frac{\beta}{2}-\frac{\gamma}{2})}\sin\left(\frac{\theta}{2}\right) & e^{i(\frac{\beta}{2}+\frac{\gamma}{2})}\cos\left(\frac{\theta}{2}\right) \end{pmatrix} = e^{i\alpha}R_z(\beta)R_y(\theta)R_z(\gamma). \quad (1.2)$$

These operations can be interpreted as rotations in the Bloch sphere. In this scheme, each point of the surface of the Bloch sphere represents a different state of the qubit. In Eq. (1.2), we show that a general rotation could be regarded as three different rotations along the z and y axis of the Bloch sphere. More specifically, these rotations are defined in



(a)

c	t	c'	t'
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

(b)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(c)

Figure 1.5: Figure (a) shows a circuit diagram for the CNOT gate. $|c\rangle$ stands for control qubit and $|t\rangle$ stands for target qubit. Figure (b) shows the truth table and figure (c) depicts the unitary matrix representation of this gate.

Eq. (1.3), where some other useful SQR are also provided,

$$\begin{aligned} R_x(\theta) = e^{-i\theta\frac{X}{2}} &= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}, \quad X = \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \\ R_y(\theta) = e^{-i\theta\frac{Y}{2}} &= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}, \quad Y = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \\ R_z(\theta) = e^{-i\theta\frac{Z}{2}} &= \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}, \quad Z = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}. \end{aligned} \tag{1.3}$$

Multi qubit gates

A system comprising L qubits is associated to a 2^L-dimensional complex Hilbert space. In this case, apart from SQR, there are also more general $2^L \times 2^L$ gates.

Gates acting on to more than one qubit, that can not be decomposed as a product of SQR, are called multi qubit gates. A historically relevant multi-qubit gate is the CNOT gate, whose representation is shown in Fig. 1.5. This gate has two inputs, one of them denoted as the control qubit and the other one as the target qubit. If the control qubit is in the state $|0\rangle$, the target qubit does not change. However, if the control qubit is in the state $|1\rangle$, an X gate is applied to the target qubit.

In the computational basis, where the Z gate is diagonal, the matrix representation of the CNOT gate has a single one in each column and row. This group of matrices are called permutation matrices because they map each input to only one output. This is the same property that classical reversible functions have and actually, each permutation matrix can be mapped to a classical reversible function. We already proved in the previous section that the Fredkin gate is universal from a reversible computation perspective. This means that being able to implement it in a quantum computer, allows us to simulate any classical reversible function. Besides, using the algorithm shown in Fig. 1.3 we can simulate any irreversible function.

However, this does not prove the universality of the Fredkin and CNOT gates in quantum computation. In this paradigm, the universality principle requires to simulate any unitary matrix, not only permutation matrices. We will prove later that the CNOT gate, along with SQR, make up a universal set of gates.

In the following sections, when we refer to a unitary gate that implements a classical function f, we mean a unitary gate implementing the classical reversible version of the function f, according to what we saw in the previous section.

1.3.3 Universality principle

In this section, I discuss the universality principle, which states that any unitary gate can be simulated with arbitrary accuracy using only CNOT gates and SQR. It is worth noting that,

even though SQR are a continuous set of unitary gates, they can be approximates using a discrete set of SQR. However, the explanation of this is beyond the scope of this work, it suffices to say that the SQR H and T can simulate approximately any SQR.

Another vital property of the quantum universality principle is that some unitary matrices require an exponential amount of CNOT gates and SQR to be approximated. Hence, it may not always be possible to simulate any desired unitary matrix.

Generator of a unitary gate

Let us start proving the universality of the CNOT gate together with the SQR by noting that any unitary gate is determined by its Hamiltonian generator, $H_g = -i\log(U)$.

It also important to remember that any Hermitian matrix can be expanded as a sum of Pauli matrices. More precisely, it holds $H_g = \sum_j g_j h_j$, with $g_j = \bigotimes_{k=1}^{L} \sigma_{c(j)}^{(k)}$, where the sum is over all the possible g_j, a L-fold tensor product of Pauli matrices, and L is the number of qubits of the system. Additionally, $c(j) \in \{0,1,2,3\}$, which represents the Hermitian matrices $\{I, X, Y, Z\}$ and h_j is just a real number. The most important thing to notice from this sum is that it has 4^L summands.

Trotter decomposition

As not all the g_j commute, $U = e^{i\sum_j g_j h_j} \neq \prod_j e^{i g_j h_j}$. However, we can expand U as the product of more simple exponentials, up to a certain degree of accuracy. We will use a modification of Trotter decomposition,

$$\lim_{n\to\infty} \left(e^{A/n} e^{B/n}\right)^n = e^{A+B}, \tag{1.4}$$

where A and B are Hermitian matrices.

We will start by noting that

$$U = e^{iH_g} = \left(e^{iH_g \frac{1}{k}}\right)^k = \left(e^{iH_g \Delta}\right)^k, \tag{1.5}$$

which holds for any integer value k. We also defined $\Delta = \dfrac{1}{k}$ for simplicity.

The decomposition we are looking for is

$$e^{iH_g\Delta} = e^{i\sum_j g_j h_j \Delta} = \mathbb{I} + i\sum_j g_j h_j \Delta + \mathcal{O}(\Delta^2) =$$
$$\prod_j (\mathbb{I} + i g_j h_j \Delta) + \mathcal{O}(4^L\Delta^2) = \prod_j e^{i g_j h_j \Delta} + \mathcal{O}(4^L\Delta^2), \tag{1.6}$$

which in the limit of $\Delta \to 0$ becomes a generalization of the Trotter formula. With Eq. (1.6) we approximated the exponential of a sum by the product of exponentials, within an accuracy of $\mathcal{O}(4^L\Delta^2)$.

Then, using the results of Eq. (1.5) and Eq. (1.6) we get

$$U = \left(\prod_j e^{i g_j h_j \Delta} + \mathcal{O}(4^L\Delta^2)\right)^k = \left(\prod_j e^{i g_j h_j \Delta}\right)^k + \mathcal{O}(4^L\Delta), \tag{1.7}$$

where we used $k\Delta = 1$.

To sum up, we have proven that any unitary matrix can be approximated to an arbitrary accuracy of $\mathcal{O}(4^L\Delta^2)$ by the product of more simple unitary matrices. If we want an accuracy of ϵ, we need to set $k = \dfrac{4^L}{\epsilon}$. The next step is to exactly obtain every unitary matrix $e^{i g_j h_j \Delta}$ using only CNOT gates and SQR.

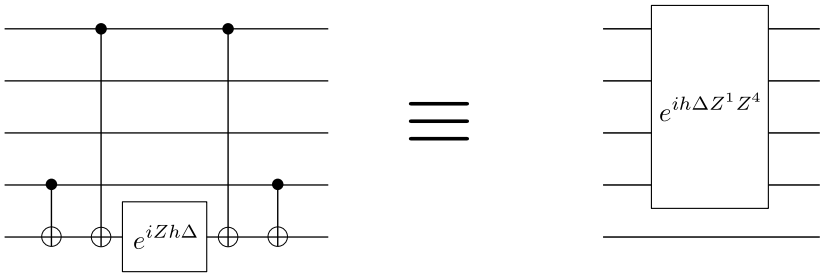


Figure 1.6: A quantum circuit that simulates the evolution of the $e^{iZ^1Z^4h\Delta}$ matrix using an ancillary qubit, CNOT gates and a SQR.

Z interactions

Let us firstly exemplify how to get all the "Z-interactions", that is, all the $e^{ig_jh_j\Delta}$ where g_j only holds Z and I matrices in its L-fold tensor products.

In Fig. 1.6, we show how to simulate the evolution of $e^{igh\Delta}$ with $g = Z^1 \otimes I^2 \otimes I^3 \otimes Z^4$. In order to understand why the circuit of Fig. 1.6 really simulates the required evolution, it suffices to see how the circuit behaves for each vector of the computational basis.

The first set of CNOT gates perform the XOR operation in the ancillary qubit, while the last set of CNOT gates bring back this qubit to its original state. Before the SQR is applied, the state of the ancillary qubit would transform to $|1\rangle$ if an odd number of control qubits were in the state $|1\rangle$, whereas it would transform to $|0\rangle$ if an even number of control qubits were in the state $|1\rangle$. Therefore, the qubits which do not participate in the CNOT operation, qubits 2 and 3, have no effect in the final evolution. If the state of the qubits 1 and 4 is $|01\rangle$ or $|10\rangle$, the last qubit will be set to the state $|1\rangle$ and the SQR applies a phase of $e^{-ih\Delta}$. This is the same phase that the gate $e^{iZ^1Z^4h\Delta}$ sets to those states. If however qubits 1 and 4 are in the state $|00\rangle$ or $|11\rangle$, the ancillary qubit will be set to the state $|0\rangle$, and the SQR would apply a phase of $e^{-iZ^1Z^4h\Delta}$. This is again the same phase as the gate $e^{-iZ^1Z^4h\Delta}$ would add to the whole system.

All in all, the circuit shown in Fig. 1.6, applies the same phase factor that the corresponding Z-interaction matrix does. If we want to simulate the Z-interaction among a certain number of qubits, it suffices to adapt the circuit of the figure Fig. 1.6, placing the CNOT gates only where we want a Z interaction.

XYZ interactions

The rest of the terms are obtained by applying SQR to the Z interaction. For example, if we want to simulate the evolution $e^{iX^1Z^2Y^3Z^4}$, it suffices to realize that

$$e^{iX^1Z^2Y^3Z^4} = S^3H^3H^1e^{iZ^1Z^2Z^3Z^4}H^1H^3S^{\dagger 3}, \tag{1.8}$$

where we used the property $Ue^{iH_g}U^\dagger = e^{iUH_gU^\dagger}$.

Therefore, by sandwiching the Z evolution with Hadamard gates we generate a X interaction, while using the SH SQR let us generate a Y interaction.

This basically completes the universality proof. After calculating the generator of a certain unitary, we just need to use the rules discussed in this section, along with the Eq. (1.7), to simulate the evolution of any unitary gate using only CNOT gates, SQR and an extra ancillary qubit.

Counting the number of CNOT gates

Let us count the total number of CNOT gates needed to simulate a unitary matrix. Recall that the number of SQR required is proportional to the number of CNOT gates. For each

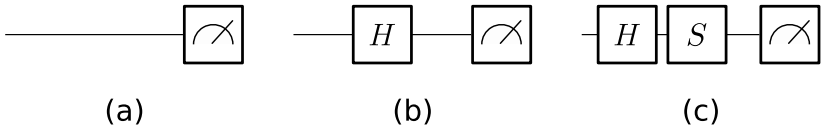


Figure 1.7: Example of circuits to measure in the Z (a), X (b) and Y (c) basis. The last block of each circuit refers to the measure process.

$e^{ig_j h\Delta}$ unitary gate, we need at most $2L$ CNOT gates, that is, $\mathcal{O}(L)$ CNOT gates, where L is the number of qubits in the system. We need to implement at most 4^L $e^{ig_j h\Delta}$ terms to perform the product operation shown in Eq. (1.7). Thus, the total number of CNOT gates will be $\mathcal{O}(4^L L)$. Now, if we want an accuracy ϵ, we need to set $k = \dfrac{4^L}{\epsilon}$. Since k is the number of evolutions required, we need a total of $\mathcal{O}\left(k4^L L\right) = \mathcal{O}\left(16^L \dfrac{L}{\epsilon}\right)$ CNOT gates.

This means that there are some unitary matrices which require an exponential amount of CNOT gates to be simulated. However, most of the interesting problems do not need so many CNOT gates.

1.3.4 Measurements

The last fundamental operation we can perform on qubits is measurement process. When we measure in a quantum computer, we measure the register containing all the qubits, so that at the end of the process, the state of the register has collapsed to one of the states of the computational basis. This is the last step performed in every algorithm.

In order to measure in a basis different from the computational one, we need to previously perform a unitary evolution changing the basis. For example, in Fig. 1.7 the Hadamard gate H is used to measure in the X basis.

This is where the challenging part of quantum computation happens. As we will see later, there are algorithms which clearly outperforms classical ones. However, when we introduce the necessity of making a measurement to retrieve the information from the registers, we will see that those quantum algorithms are no longer faster than classical ones. The manner quantum algorithms do outperform classical algorithms is by making an intelligent manipulation of the qubits in order to retrieve the information needed.

There is a method to retrieve the information of a quantum state which is called quantum state tomography. It is based on the fact that the state of a system can be represented by a density matrix ρ. This matrix can be expanded in the basis of Pauli matrices. An example for a three qubit system is

$$\rho = \sum_{i,j,k\leq 3} a_{ijk}\sigma_i \otimes \sigma_j \otimes \sigma_k. \tag{1.9}$$

It is straightforward to see that, due to the orthogonality of Pauli matrices, the values $a_{ijk} = \frac{1}{2^3}\operatorname{tr}(\rho\sigma_i \otimes \sigma_j \otimes \sigma_k) = \frac{1}{2^3}\langle\sigma_i \otimes \sigma_j \otimes \sigma_k\rangle$. This means that, by performing multiple times the same algorithm and measuring the state after each run of the it, we can infer the state of the system. However, the number of coefficients a used in Eq. (1.9) increases exponentially, having to calculate 4^L coefficients, where L is the number of qubits in the system. Notice that in each run we can only obtain information for one of the coefficients, depending on which basis we choose to measure. Therefore, even if we require to approximate each mean value with 10 measurements, we end up requiring 10×4^L measurements and hence, we need to repeat the algorithm 10×4^L times. This exponential behaviour makes unfeasible to extract information of large quantum systems using quantum state tomography, which is one of the biggest drawbacks of quantum computing.

Chapter 2

Relevant algorithms in quantum computing

This chapter aims at explaining the relevance of some quantum algorithms that have driven the scientific community to investigate deeper in this field. With that in mind, we will introduce some algorithms that outperform their classical counterparts. More specifically, we review the quantum Fourier transform (QFT) algorithm, Deutsch's algorithm, Shor's algorithm and Grover's algorithm. The implementation of the QFT algorithms has been obtained from the book Ref. [1]. The other three algorithms have been reviewed from Ref. [2], where the improvement made to Grover's algorithm is stated as an exercise. However, it is useful to start reviewing some concepts of probabilistic algorithms and computational complexity classes.

2.1 Computational complexity classes and probabilistic algorithms

The review shown in this section is obtained mainly form the book Ref. [1]. We will discuss how to characterize an algorithm in order to understand why quantum algorithms outperform classical ones. We will also introduce the notion of probabilistic algorithm, which is essential to understand some of the algorithms reviewed.

2.1.1 Computational complexity classes

The main idea behind the complexity classes is to characterize how difficult is to solve a given problem. This is done by counting the resources that are required as a function of the size of the input.

We suppose that a problem has some input size that characterized it, i.e, the number of bits that the input requires. In general, the limiting resource to solve a problem is time. However, the amount of time need to solve a problem is proportional to the amount of 'simple' steps[1] or gates[2]. We will use this last resource, the number of gates, as our limiting resource.

With these definitions for the resources and size of a problem, we can classify it according the best algorithm known to solve it. If the amount of resources required to solve the problem increases polynomially, it is said to be efficiently solved and belongs to the complexity class P. However, if we need an exponential amount of resources, it is said to be inefficiently solved. However, it may be possible to check efficiently whether a given answer is a solution. If we

[1] By simple we mean steps that require some fixed or bounded amount of time.

[2] In this case, gates need some bounded operation time so there is a direct connection between these two resources.

have a possible solution w, also called witness, and we can efficiently check whether it is a correct solution, the problem belongs to the complexity class NP.

An historically important example of NP problem would be the factorization problem. Given a number N of L bits long, we wish to find p_1 and p_2 such that $N = p_1 p_2$. In this case, L would be the size of the problem. If we want to know whether a particular number a is a solution of the problem, we just need to divide the two numbers, which requires a polynomial amount of resources. However, finding a possible value a using a simple classical algorithm requires to compute all the divisions for $i \in [2, \sqrt{N}]$ until we find a solution to the problem. That is, we need to search the solution from a set of possible solutions. However, there are approximately $2^{\frac{L}{2}}$ possibilities. That would require to perform and exponential amount of checks and, hence, it requires an exponential amount of resources even if the division operation is computed efficiently.

It is noteworthy that we do not know yet if P $\neq$ NP. A simple way to understand this is that it could exist an unknown mathematical theorem that could be used to efficiently solve a problem believed to belong to the complexity class NP. Proving if P $\neq$ NP is extremely difficult and it is famous for being one of the Millennium Prize Problems.

2.1.2 Probabilistic algorithms

There are algorithms that can achieve better results solving a problem when randomness is allowed in the output of the algorithm. At this point, it is convenient to differentiate this kind of algorithms from the deterministic ones.

A deterministic algorithm is an algorithm defined by two characteristics. The first one is that it always outputs the same result for the same input. The second one is that the output is always known to be correct.

On the other hand, a probabilistic algorithm is mainly characterized by the fact that it outputs a correct answer with a probability p. Its output can also vary for the same input when randomness is allowed on any of its steps.

We are interested in the set of probabilistic algorithms that output an answer which can be checked efficiently. If the succeeding probability of the algorithm is $p(\text{success}) = \epsilon$, we repeat the algorithm k times. The probability of getting a wrong answer in all of them is

$$p(\text{failing k times}) = (1-\epsilon)^k \leq e^{-\epsilon k}, \tag{2.1}$$

meaning that the probability of failing decreases exponentially with the number of repetitions of the algorithm. In order to characterize this algorithms, we are interested in the number of expected run time, which is in this case $\frac{1}{\epsilon}$. If it is polynomial in the size of the problem, the algorithm belongs to the complexity class ZPP (Zero-error Probability Polynomial time). However, notice that worst case scenario may require to run the algorithm an infinite amount of times.

2.2 Quantum Fourier transform

The quantum Fourier transform (QFT) is a key algorithm to understand the algorithms that outperform their classical counterparts. This algorithm is a change of basis with incredible useful properties that can be exploited for information processing. For example, when dealing with a periodic function, as it is the case for the period finding algorithm, the QFT will be able to efficiently extract information about the period.

Let us start by reviewing some concepts of the Fourier transform. The Fourier transform of a complex vector x_j has as output another complex vector of components y_k that are

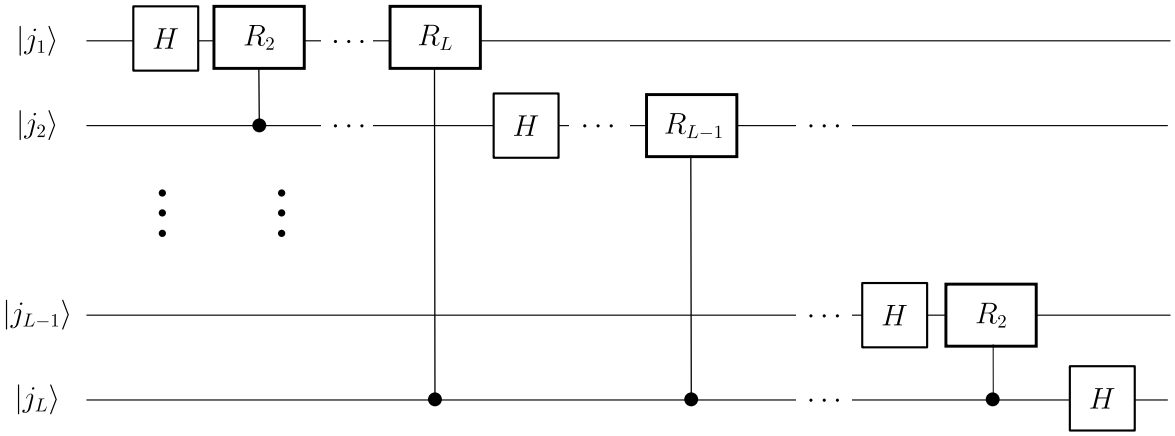


Figure 2.1: Quantum circuit that performs the quantum Fourier transform. The gates R^n are controlled SQR defined in Eq. (2.3). The H gate is the Hadamard gate defined in Eq. (1.3).

related by

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{i2\pi \frac{jk}{N}}$$
$$x_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} y_k e^{-i2\pi \frac{jk}{N}}. \tag{2.2}$$

We will only be interested in the case in which these vectors have dimension $N = 2^L$, being L the number of working bits.

The QFT performs the Fourier transform of the amplitudes of a state of L qubits. If the initial state of the system is $|\phi\rangle = \sum_{j=0}^{N-1} x_j |j\rangle$, the state obtained after applying the QFT gate is $\mathrm{QFT}\,|\phi\rangle = \sum_{k=0}^{N-1} y_k |k\rangle$. The circuit that performs the QFT operations is depicted in Fig. 2.1. In this figure, the control R_n gate applies the following SQR

$$R_n = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{2\pi}{2^n}}. \end{pmatrix} \tag{2.3}$$

This SQR is applied to the target qubit only when the control qubit is in the state $|1\rangle$. The implementation for the QFT takes $\mathcal{O}(L^2)$ controlled R_n gates, where L is the number of qubits. This is due to the fact that the number of controlled R_n gates applied to the qubit $|j_k\rangle$ is equal to $L-k$. Therefore, the total number of gates is $\sum_{k=1}^{L} L-k = L^2 - \frac{L(L+1)}{2} = \frac{L(L-1)}{2}$. This proves that the QFT is an efficient algorithm.

It is also known that the best classical algorithms to implement this transformation requires $\mathcal{O}(L2^L)$ operations. It seems then that it could be possible to use the QFT to replace the classical version. However, that is far from being the truth. Even if we are able to set correctly the initial state, retrieving the information from the QFT transformation is by no means a trivial task. It requires to perform quantum state tomography, which is an operation that repeats the QFT algorithm an exponential amount of times. Therefore, we recover the exponential factor we had for the classical algorithm. For that reason, the QFT algorithm shines best when employed as part of some other algorithm.

The are some properties that make the QFT really useful. First of all, the value $y_{k=0}$ is the mean value of the values x_j,

$$y_0 = \frac{\sqrt{N}}{N} \sum_{j=0}^{N-1} x_j = \frac{1}{\sqrt{N}} \langle x \rangle. \tag{2.4}$$

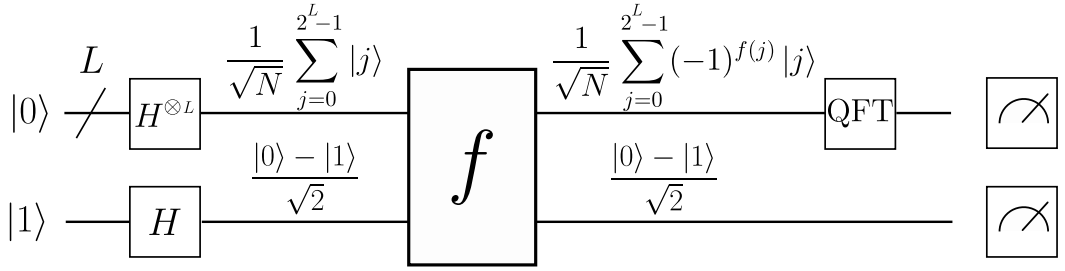


Figure 2.2: Circuit employed to solve the Deutch's problem. The first register holds L qubits initialized in the state $|0\rangle$. N stands for $N = 2^L$. The H gate is the Hadamard gate defined in Eq. (1.3). The gate f refers to the unknown constant or balanced function. The QFT gate performs the quantum Fourier transform operation discussed in section 2.2. The last blocks refers to a measurement in both registers.

It is also useful to remember that the QFT of the state $|0\rangle$ is a superposition of all the states

$$QFT\,|0\rangle = \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1}|k\rangle \tag{2.5}$$

This last result is the case $j = 0$ of the more general transformation,

$$QFT\,|j\rangle = \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1}e^{i2\pi\frac{kj}{N}}\,|k\rangle\,. \tag{2.6}$$

We will use these properties in the next section to explain the transformation that the QFT gate applies to the state.

2.3 Deutsch's algorithm

Thought Deutch's algorithm has no practical application, it provides insight about the operations that a quantum computer can perform to speed-up algorithms. Deutch's algorithm solves the following problem: given a function $f : \{0,1\}^L \to \{0,1\}$, we need to know whether it is constant or balanced using the least amount of queries. A constant function outputs only 1 or 0 for all its inputs, whereas a balanced one, outputs 1 for half its inputs and 0 for the other half.

In this problem our resource is characterized by the number of queries to the function, whereas the size of the problem is characterized by L, the number of bits that the input allows. We use the queries as resource because an efficient number of them implies an efficient number of gates, as long as the function is implemented efficiently. Indeed, using the results of section 1.2, we know that if an efficient classical implementation of a function exists, and efficient quantum one must also exist.

From a classical perspective, the worst case scenario occurs when the function is constant, since we need to check the output of $2^{L-1}+1$ different inputs in order to prove it. Thus, classically the problem grows exponentially.

To solve this problem in a quantum computer we employ the circuit depicted in Fig. 2.2. In this circuit we use L qubits grouped in the first wire as a register for the input. The second wire represents a register employed to hold the output. The first step in the circuit is to apply the Hadamard gate on the initial qubits to obtain a superposition of all the possible inputs in the first register. Notice that the gate H, defined in Eq. (1.3), performs the following transformation in the computational basis,

$$H\,|0\rangle = \frac{|0\rangle+|1\rangle}{2}, H\,|1\rangle = \frac{|0\rangle-|1\rangle}{2}.$$

Then, the gate f performs the following operation

$$f\left|x\right\rangle \frac{\left|0\right\rangle -\left|1\right\rangle }{\sqrt{2}}=\left|x\right\rangle \frac{\left|0\oplus f(x)\right\rangle -\left|1\oplus f(x)\right\rangle }{\sqrt{2}}=\left|x\right\rangle (-1)^{f(x)}\frac{\left|0\right\rangle -\left|1\right\rangle }{\sqrt{2}}. \tag{2.7}$$

Now, we can think of the input for the QFT algorithm as a vector $x_j = \frac{1}{\sqrt{N}}(-1)^{f(j)}$, where $N = 2^L$. Since the QFT computes the Fourier transform of x_j, the output of the QFT will be the state $\sum_{k=0}^{k=N-1} y_k \left|k\right\rangle$, being y_k the Fourier transform of x_j. Taking into account that y_0 is the mean value of x_j, two cases arise:

- **The function f is constant:** In this case, the value of $y_0 = 1$. By normalization $y_k = 0$ for $k \neq 0$.
- **The function f is balanced:** In this case, the value of $y_0 = 0$.

The last step consists in measuring the first register. If we obtain a 0 in the measurement the function is constant, otherwise, it is balanced.

In conclusion, the quantum algorithm is able to output the correct answer with only one query to the function. This is an exponential speed-up over the classical algorithm. The main feature of this algorithm is that it evaluates the function in every input possible at the same time, which will also be an operation performed in the rest of the algorithms. However, we needed to use a clever approach, for example the QFT, to recover the required information.

2.4 Shor's algorithm

Before digging into Shor's algorithm for factoring numbers, we shall emphasize the relevance of this algorithm. There is no classical algorithm known, probabilistic nor deterministic, that can solve efficiently the problem of factoring numbers. Consequently modern cryptography relies on this fact to create a secure protocol of communication. Solving the factorization problem efficiently would then mean to break modern cryptography. However, even though quantum computers promise to achieve this remarkable fact, they are still far from tackling big numbers. For instance, the largest quantum processor provided by IBM, comprises 20 qubits, meaning that it could only factorize numbers as large as $2^{20} = 1048576$.

The core of factoring prime numbers resides in the order finding problem. This section will be devoted to show the connection between this two problems and the next section will focus on solving efficiently the order finding problem. However, we start introducing briefly some concepts about the great common divisor operation and modular arithmetic.

2.4.1 Great common divisor and modular arithmetic

The great common divisor operation, $\gcd(a, b)$, returns the greatest of the common factors between the natural numbers a and b. This is an operation that can be efficiently implemented in a classical computer employing Euclid's algorithm, which relies on the property that $\gcd(a, b) = \gcd(a, b - a)$, with $b > a$. If two numbers share no common factor, they are said to be co-prime and $\gcd(a, b) = 1$.

I will now review some concepts of modular arithmetic. Computing $a \mod N$, is an operation that can also be performed efficiently by classical computers. It only requires to subtract at most L times L-bit long numbers.

A relevant result of modular arithmetic is that if $x \neq \pm 1 \mod N$ and $x^2 = 1 \mod N$, then,

$$\begin{aligned} x^2 &\equiv 1 \mod N \\ x^2 - 1 &\equiv 0 \mod N \\ (x-1)(x+1) &\equiv 0 \mod N \\ (x-1)(x+1) &= kN, \end{aligned} \tag{2.8}$$

where we chose $x \in [2, N-2]$. Since $1 \leq x-1 \leq x+1 \leq N$, $x-1$ or $x+1$ must share a common factor with N, which can be efficiently obtained computing $\gcd(x-1, N)$ and $\gcd(x+1, N)$.

2.4.2 The core of the factorization algorithm

We now introduce the algorithm to factorize a number N. We will simplify the problem by supposing that this number only has two prime factors, p_1 and p_2. In addition to that, we will suppose that $p_1 \neq p_2$ and that both primer numbers are different form 2. This two cases can be checked efficiently using classical algorithms.

We now choose a random number $x \in [2, N-2]$ and compute $s = \gcd(x, N)$. If $s \neq 1$ we already found a factor of N. However, if x is co-prime to N, we need to find the smallest value r such that $x^r = 1 \mod N$. This problem is called order-finding and r is known as the order. We need r to be odd in addition to satisfy $x^{\frac{r}{2}} \neq -1 \mod N$, which fulfils with a probability greater than $\frac{1}{2}$. Proving this is not a trivial task and it is beyond the scope of this work. However, notice that if we do not get an order r that fulfils these criteria, we can repeat the order finding algorithm until we obtain the desired output. The expected number of repetitions is then less than 2.

With these conditions, we can apply the results of Eq. (2.8). That means that either $x^{\frac{r}{2}} - 1$ or $x^{\frac{r}{2}} + 1$ shares a factor with N. Computing their great common divisor with N we obtain the desired factors.

The only step that can not be performed efficiently in a classical machine is the order-finding problem. However, this is a problem efficiently solvable by a quantum computer. The next section will be devoted to show the quantum circuit that performs this task.

2.4.3 Order-finding algorithm

The first thing to discuss is the implementation of the function

$$f(a) = x^a \mod N \tag{2.9}$$

using modular exponentiation. This is a periodic function of period r, $f(a) = f(a+r)$, which is equal to the order r we are looking for. I will show that it is possible to implement efficiently this function classically, which means that, due to our discussion in the previous chapter, an efficient quantum version of this function also exist.

Modular exponentiation

In order to implement in a classical computer the function shown in Eq. (2.9), we need to note that a natural number a is written in its binary form as $a = \sum_{i=0}^{L} a_i 2^i$, where $a_i \in \{0, 1\}$ and L is the number of bits needed to represent a. Since,

$$x^a \mod N = x^{\sum_{i=0}^{L} a_i 2^i} \mod N = \prod_{i=0} a_i g_i \mod N,$$

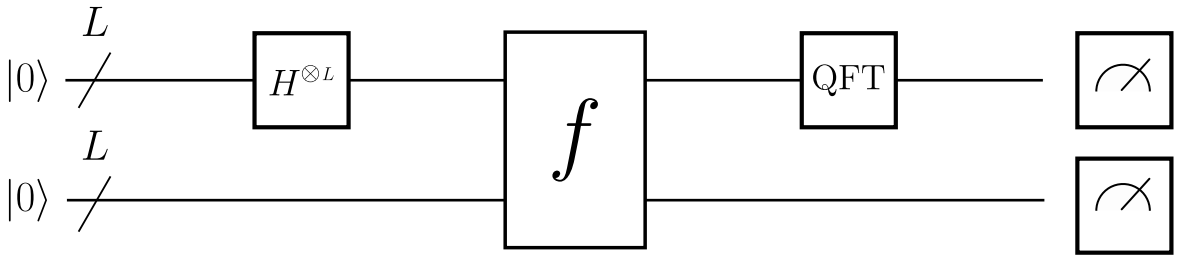


Figure 2.3: Circuit that solves the order finding problem. It has two registers as input, each of them holding L qubits. The H gate is the Hadamard gate defined in Eq. (1.3). The gate f performs the quantum reversible version of the operation defined in Eq. (2.9). The QFT gate performs the quantum Fourier transform operation discussed in section 2.2. The last blocks refers to a measurement in both registers.

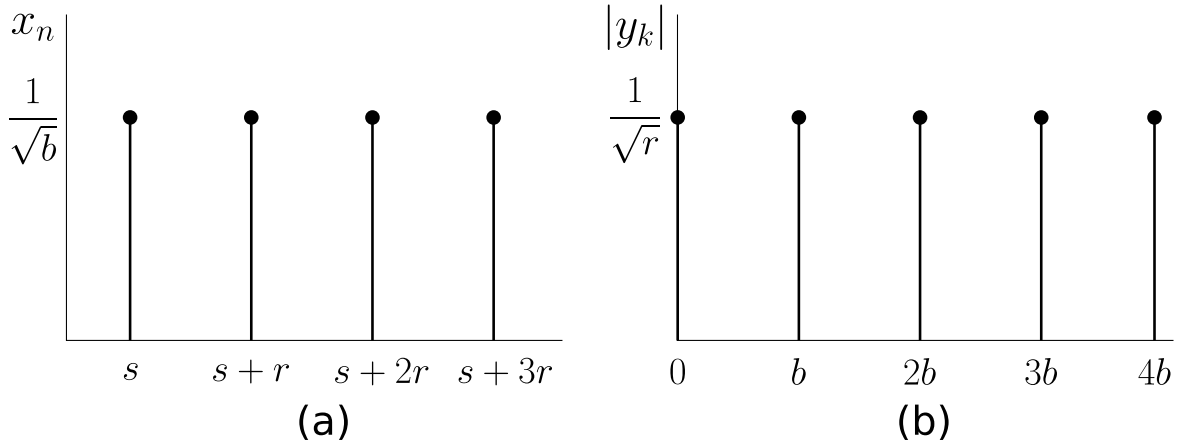


Figure 2.4: Figure (a) shows an example of vector x_n that refers to the amplitudes of a state in the order finding algorithm before applying the QFT operation. Figure (b) shows the amplitude of the state after applying the QFT operation.

where $g_i = x^{2^i} \mod N$. We seek to implement efficiently the g_i factors and finally compute $\prod_{i=0} a_i g_i \mod N$.

We already discussed that the operation $a^2 \mod N$ can be computed efficiently. Then, in order to get all the g_i factors, we use the fact that $g_{i+1} \mod N = g_i^2 \mod N$, so the number of times we need to perform the squaring operation is $\mathcal{O}(L)$. This means that we can efficiently compute all the g_i factors. At the same time, we need to perform L modular multiplications of the g_i factors to get the desired result, which again can be done efficiently.

All in all, we can compute the function 2.9 efficiently employing the algorithm discussed, which is known as modular exponentiation. We continue now reviewing the quantum circuit to solve the order-finding problem, supposing that we already implemented the periodic function f as a quantum gate.

The quantum algorithm

Now that we know how to implement the function of Eq. (2.9), we are ready to explain the quantum order-finding algorithm. For the sake of simplicity, I will assume that $b = \frac{N}{r}$ is a natural number, where N is the number whose factors we are looking for and r is the order of the operation shown in Eq. (2.9).

The order finding quantum circuit is shown in Fig. 2.3. In this circuit we have two registers that holds L qubits each. One of them will be used as input to the function Eq. (2.9) and the other one to hold the output of function. We start the algorithm by

making a superposition of all the possible inputs to f, using the Hadamard gates. We continue applying the periodic function to get the state

$$f|\phi\rangle = \frac{1}{\sqrt{2^L}} \sum_{j=0}^{2^L-1} f|j\rangle |0\rangle^{\otimes L} = \frac{1}{\sqrt{2^L}} \sum_{j=0}^{2^L-1} |j\rangle |f(j)\rangle$$

The next step is to measure the second register, which holds the output of the function. This makes the second register to collapse into a state $|f(s)\rangle$ for some $s < r$. At the same time, the first register collapses in a superposition of all the inputs whose output is $f(s)$. From now on we only need to focus on the first register whose state is

$$|\psi\rangle = \frac{1}{\sqrt{b}} \sum_{n=0}^{b-1} |nr + s\rangle$$

The amplitudes of this state should be regarded as a periodic function of period r, which is represented, along with its Fourier transform, in Fig. 2.4. The function obtained from this Fourier transform represents the amplitudes of the states obtained when the QFT is applied to the state $|\psi\rangle$. This means that the state, after applying the QFT is, up to a global phase factor,

$$QFT|\psi\rangle = \frac{1}{\sqrt{r}} \sum_{l=0}^{r-1} |lb\rangle . \tag{2.10}$$

The last step is to measure the first register in order to get a value k proportional to b. That is, we get $k = lb$ for some l. Rearranging this last equation we get

$$\frac{k}{N} = \frac{l}{r} \tag{2.11}$$

for some value l.

In order to get the value r from $\frac{k}{N}$, we need to apply the continued fraction algorithm. This is a classical algorithm that successes retrieving the factor r only if l and r have no common factors, that is, $\gcd(l, r) = 1$. If they happen to have a common factor $s = \gcd(l, r)$, then the value $\frac{r}{s} < r$ would be the output of the continued fractions algorithm. By checking if $x^r \mod N = 1$, which can be done efficiently, we can ensure that we retrieved the correct value of r. If not, we just need to repeat the algorithm until we do get a value k whose l is co-prime with r. From the discussion in the section 3.1.1, we know that we will retrieve with high probability a correct value k within a few repetitions.

2.5 Grover's algorithm

Grover's algorithm is another of the most relevant algorithms for quantum computing that seem to clearly improve their classical counterparts. However, this time the improvement is not exponential, unlike Shor's algorithm. Nevertheless, it is still a noticeable improvement.

In its most fundamental statement, Grover's algorithm is used to speed up the searching process. We will start this section talking about how the function for searching is defined and then we will dig into the core of the algorithm. Finally, we will show an improvement that makes Grover's algorithm shine better.

It should be noted that during the explanation of Grover's algorithm we will suppose that we already know the number of solutions to the searching problem. If this is not the case, some modifications could be made to the algorithm, but we will omit them for the sake of simplicity.

The searching function

The searching process is modelled as a function $f : \{0,1\}^L \to \{0,1\}$ whose input is a L-bit address and whose output is 1 when the value stored in the address matches our desired value. We will suppose that if a classical version of this function can be built, then, using the algorithm shown in the previous chapter, we can also build a quantum gate F that can perform the following operation,

$$F\left|x\right\rangle\left|0\right\rangle = \left|x\right\rangle\left|0 \oplus f(x)\right\rangle = \left|x\right\rangle\left|f(x)\right\rangle. \tag{2.12}$$

As we can always simulate efficiently F, we will compare the number of queries to this gate for the classical and quantum version of the algorithm. We will obtain that the Grover's algorithm has a square speed-up when comparing to the best classical algorithm.

We are actually more interested in having the information of the value $f(x)$ as a phase. That is way we will define a new gate $O = FZF^{-1}$, called oracle, where the Z operation is performed only in the qubit that holds the value $f(x)$. The whole operation is then just $O\left|x\right\rangle\left|0\right\rangle = (-1)^{f(x)}\left|x\right\rangle\left|0\right\rangle$, making it possible to omit the last qubit from now on. That is, the gate O performs the operation $O\left|x\right\rangle = (-1)^{f(x)}\left|x\right\rangle$.

This searching process can be used to speed up some problems that rely in an algorithm that searches the solution from a given set of possible solutions. For example, the naive way of finding the the prime factors of a number that we explained in section 2.1.2 relays in this search process. This naive algorithm searches the answer by repeating the division for different input values. In this case, all those possible inputs would play the role of the input of our function f. The function f would then perform the division for a given input value, and it would return 1 only if the remainder of the division is 0. Since a classical efficient version of this division-and-check algorithm exists, it is also possible to build a gate F that performs the desired transformation.

The algorithm

This algorithm has a really nice geometrical interpretation but it requires first to introduce some definitions. We will define the "good" state $\left|G\right\rangle$ as a superposition of all the states whose output to the function $f(x)$ is one. That is, all the states we are trying to find. On the other hand, the "bad" state $\left|B\right\rangle$ will be an equal superposition of all the states whose output to function f is 0. This means that, if t is the number of "good" solutions, the initial state as an equal superposition of all the possible inputs to the function f, is well described by the following equation,

$$\left|U\right\rangle = \frac{1}{\sqrt{N}}\sum_{j=0}^{N-1}\left|j\right\rangle = \sqrt{\frac{t}{N}}\left|G\right\rangle + \sqrt{\frac{N-t}{N}}\left|B\right\rangle = \sin(\theta)\left|G\right\rangle + \cos(\theta)\left|B\right\rangle \tag{2.13}$$

where $N = 2^L$ and L is the input size in bits of the function f. We also defined $\theta = \arcsin\left(\sqrt{\frac{t}{N}}\right)$. This last state can be depicted as a vector in a two dimensional space whose basis are the vectors $\left|G\right\rangle$ and $\left|B\right\rangle$. An example of this representation is shown in Fig. 2.5 (a).

I will now show two different operations that we will be able to perform in this vector $\left|U\right\rangle$. First, applying the oracle to the vector $\left|U\right\rangle$, changes the sign of the good state, $O\left|U\right\rangle = -\sin(\theta)\left|G\right\rangle + \cos(\theta)\left|B\right\rangle$, because only the good states will output one to the function f. In the graphical representation of Fig. 2.5 (a) this operations translates into a reflection with the vector $\left|B\right\rangle$.

Now we introduce the Grover gate $\zeta = \mathrm{H}^{\otimes L}\left(2\left|0\right\rangle\left\langle 0\right| - I\right)\mathrm{H}^{\otimes L}$, where I stands for the identity matrix of dimension $2^L \times 2^L$ and $\left|0\right\rangle\left\langle 0\right|$ stands for the projection into the state where all the qubits are 0. The implementation of this gate is shown in Fig. 2.5 (a). This gate

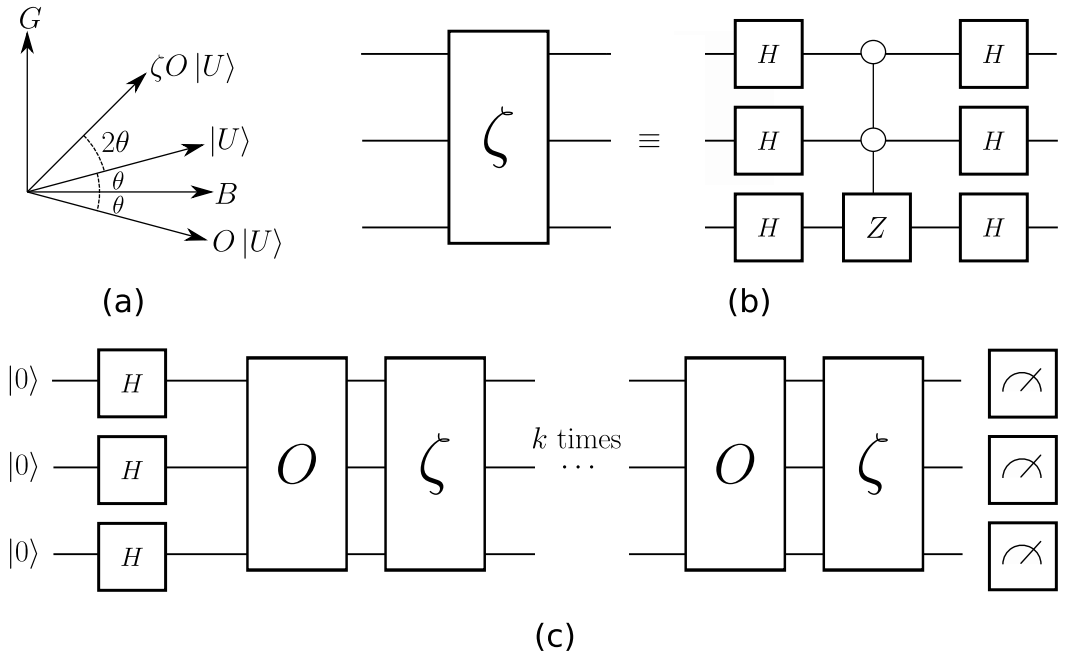


Figure 2.5: Figure (a) shows the geometrical interpretation of Grover's algorithm. We represented the initial vector $|U\rangle$, the vector obtained after the oracle operation, $O|U\rangle$, and the one obtained after both oracle and Grover operation $\zeta O|U\rangle$. G and B represent the 'good' and 'bad' states. Figure (b) shows the implementation of the Grover gate for 3 qubits using the gate H defined in Eq. (1.3) and the controled-Z gate, which applies the Z gate when every control qubit is in the sate $|0\rangle$. Figure (c) shows an example of how Grover's algorithm is implemented for three qubits, where both Grover ζ and oracle O gates are applied k times, with k defined in Eq. (2.15).

can also be written as $\zeta = 2|U\rangle\langle U| - I$ which makes clear that it is a reflection using the initial state vector $|U\rangle$, since the gate applies a relative phase of -1 to the vector $|U\rangle$ and its orthogonal state. After applying this gate to the previous state, we have the new state $\zeta O|U\rangle = \sin(3\theta)|G\rangle + \cos(3\theta)|B\rangle$ shown in Fig. 2.5 (a), which is just the reflection of the state $O|U\rangle$ through the vector $|U\rangle$.

What we actually achieve after these two reflections is to amplify the amplitude of the good state from $\sin^2(\theta)$ to $\sin^2(3\theta)$, meaning that it is know more probable to get one of the good states, when measuring the registers.

If we continue applying this two inversions k times, as it is shown in Fig. 2.5 (c), the final state will be

$$|\Psi_k\rangle = \sin((2k+1)\theta)|G\rangle + \cos((2k+1)\theta)|B\rangle\,. \tag{2.14}$$

This can be easily proven by realizing that any vector whose angle is ϕ will become a vector of angel $\phi + 2\theta$ after the two inversions. If we want the final state to be $|G\rangle$, we need k to be

$$k = \frac{\pi}{4\theta} - \frac{1}{2} = \frac{\pi}{4\arcsin\left(\sqrt{\frac{t}{N}}\right)} - \frac{1}{2}. \tag{2.15}$$

However, since k, the total number of the aforementioned two inversions, must be a natural number, only some values of $\frac{t}{N}$ will make it possible to get the correct state with certainty. Nevertheless, if the best natural number approximation to k is taken, the algorithm will output with high probability a correct answer. Furthermore, we can efficiently check whether an output is correct and if it is not, we can repeat the algorithm until we get a correct answer.

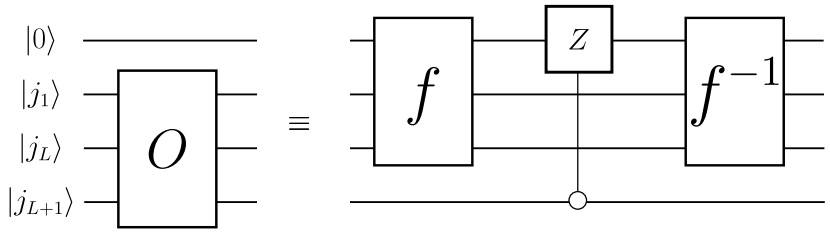


Figure 2.6: The new version of the oracle with an extra qubit as input that can be used to exactly solve the searching problem. It requires the same amount of ancillary bits as the previous version. The gates employed are the gate f, which applies the quantum reversible version of the searching function, and the gate f^{-1}, which is the inverse of the previous gate. The gate in the middle is a control-Z operation which only applies the Z gate if the extra qubit $|j_{L+1}\rangle$ is in the state $|0\rangle$.

In the end, when we measure the register after applying the inversions k times, we will get one of the "good" states. If we now change the searching function f so it does not include our previous answer, we can perform again the searching process to get a new good state. If we continue with this procedure t times, we will get all the possible "good" states, hence solving the searching problem.

Improving Grovers algorithm

A slight improvement can be achieved in Grover's algorithm in order to output with certainty a correct answer. This modification consist in adding an extra qubit to the input and modifying the oracle in such a way that it performs the same operation only if the extra qubit added is in the $|0\rangle$ state. This means that the new solutions to the search problem will be all the previous solutions only if the extra qubit added is in the $|0\rangle$ state. The modification required in the oracle is shown in Fig. 2.6 where the extra qubit is denoted as $|j_{L+1}\rangle$.

It is also necessary to change the initial state. In this case, we will use the initial state

$$|U\rangle = (\sin(\theta)\,|G\rangle + \cos(\theta)\,|B\rangle)\,(\cos(\gamma)\,|0\rangle + \sin(\gamma)\,|1\rangle)\,, \tag{2.16}$$

where the state of the new qubit is obtained performing an $R_y(2\gamma)$ SQR.

Since the new oracle sees as "good" state the state $|G\rangle\,|0\rangle$ (this is the state whose phase will be changed by the new oracle), we can define our new "good" state as being $\sin(\theta')\,|G'\rangle = \sin(\theta)\cos(\gamma)\,|G\rangle\,|0\rangle$, which in turn, let us define the new angle $\sin(\theta') = \sin(\theta)\cos(\gamma)$.

At this point, we just need to remember that the number of inversions we need to perform to end exactly in a good state is given by Eq. (2.15) with $\theta = \theta'$, due to the fact that θ' is the angle of the new "good" state. We can now tune θ' varying the value γ until k becomes an integer number.

Another change that must be done in order to have everything tied up is to modify the ζ gate. The new ζ gate, which performs the reflections through the new vector $|U\rangle$, is $\zeta = \mathrm{H}^{\otimes L} \otimes R_y(2\gamma)\,(2\,|0\rangle\,\langle 0| - I)\,\mathrm{H}^{\otimes L} \otimes R_y(-2\gamma)$, with R_y the gate defined in Eq. (1.3). With this modification we get an algorithm that success retrieving exactly a correct answer to the searching problem.

Grover's algorithm performance

The total number of queries that must be done to the function f is k, the number of times we use the oracle gate. An upper bound for this value is

$$k = \frac{\pi}{4\theta} - \frac{1}{2} < \frac{\pi}{4\theta} \leq \frac{\pi}{4\sin(\theta)} = \frac{\pi}{4}\sqrt{\frac{N}{t}} \tag{2.17}$$

,where we used $\theta \geq \sin(\theta)$. This proves that Grover's algorithm requires $\mathcal{O}\left(\sqrt{\frac{N}{t}}\right)$ queries to f.

The best algorithm that outputs with certainty the answer to the search problem requires $\Theta(N)$ queries, that is why Grover's algorithm gives us a really important speed up. However, the searching problem is still exponential in the number of qubits ($L = \log(N)$), even if we use Grover's algorithm.

Chapter 3

Digital-Analog Quantum Computing

Digital-analog quantum computing (DAQC) is a recently developed quantum computing paradigm, which aims at improving the performance of current quantum computers by exploiting the natural dynamics of a quantum chip. It keeps SQR as digital blocks while it uses analog evolutions as multi-qubit gates.

We start this section with a brief description of some quantum platforms used for quantum computers. We continue proving the universality of the homogeneous Ising Hamiltonian under the DAQC paradigm. Finally, we show the algorithm we developed to efficiently simulate the evolution of an all-to-all (ATA) Ising Hamiltonian using as resource the nearest-neighbour (NN) Ising Hamiltonian.

3.1 Motivation behind DAQC paradigm

Physics describing interactions between qubits in the most relevant quantum platforms are often described by the Ising Hamiltonian. To perform SQR in these technologies, external control comprising lasers or microwave pulses are employed.

An example of this kind of behaviour is shown in in section 7.7 of Ref. [1] , in which it is described a quantum computer based on Nuclear Magnetic Resonance (NMR). More recent approaches are related to superconductive circuits. The physics involved in the interaction between qubits of these technologies is also modelled with an Ising Hamiltonian.

Taking this into account, it is reasonable to use the Ising Hamiltonian as the main resource to perform quantum computation. However, techniques such as refocusing in NMR are used to turn off the interaction between qubits. This makes it easier to implement the CNOT gate as the main building block.

DAQC paradigm aims at avoiding the turning off of internal interactions. It uses these same natural interactions, along with SQR, to perform more efficient computation. This leads to fewer errors and higher fidelities in gates. In Ref. [3], it was shown the viability of this computational model using as main resource the ATA homogeneous Ising Hamiltonian. They describe how to perform universal computation under this paradigm using this Hamiltonian.

In this work, we develop an algorithm that uses NN homogeneous Ising Hamiltonian as resource to efficiently simulate any ATA interaction between qubits. This is physically equivalent to enhance the connectivity of the chip in the software level, i.e without changing the architecture of the chip. The main advantage comes from the expectation of having negligible iteration between qubits far apart. This means that the NN Ising Hamiltonian is physically easier to obtain in a quantum chip.

3.2 Universality of the DAQC paradigm

We start this section by reviewing some definitions which are used in the chapter. We continue showing an algorithm based on the algorithm developed in Ref. [3], which simulates the evolution of an inhomogeneous NN Ising Hamiltonian using as resource the homogeneous one. Finally, we show how to extend it to perform universal computation.

Ising Hamiltonians

The inhomogeneous NN Ising Hamiltonian is,

$$H_{NN}(g_j) = \sum_{j=1}^{L-1} g_j \sigma_z^j \sigma_z^{j+1}, \tag{3.1}$$

where L is the number of qubits in the system. We will use the notation $H_{NN}(g_j)$ when we refer to it. The homogeneous version is obtained when $g_j = g,\ \forall j$. We will refer to it as $H_{NN}(g)$ for the sake of simplicity.

On the other hand, the ATA inhomogeneous Ising Hamiltonian is

$$H_{ATA}(g_{ij}) = \sum_{j<i}^{L-1} g_{ij} \sigma_z^i \sigma_z^j, \tag{3.2}$$

with L, again, the number of qubits in the system. We obtain the homogeneous version setting $g_{ij} = g,\ \forall i, j$. We will denote it by $H_{ATA}(g)$ for simplicity. Similarly, we use $H_{ATA}(g_{ij})$, when we are denoting to the inhomogeneous Hamiltonian.

3.2.1 Transforming $H_{NN}(g)$ into $H_{NN}(g_j)$

In order to prove later the universality using $H_{NN}(g)$, it will be necessary to simulate the evolution of an arbitrary $H_{NN}(g_j)$. This subsection is devoted to show how to achieve this evolution employing a simpler version of the algorithm developed in Ref. [3].

If we perform an X SQR in the first qubit between the evolution of the $H_{NN}(g)$ Hamiltonian, we obtain

$$X^1 e^{itH_{NN}(g)} X^1 = e^{it\sum_{j=1}^{L-1} gX^1\sigma_z^j\sigma_z^{j+1}X^1} = e^{it\sum_{j=1}^{L-1} g(-1)^{\delta_{j1}}g\sigma_z^j\sigma_z^{j+1}}, \tag{3.3}$$

where we used the relation $Ue^{iH}U^\dagger = e^{iUHU^\dagger}$. The result shown in Eq. (3.3) is basically the evolution of a $H_{NN}(g_j)$ with $g_j = (-1)^{\delta_{j1}}g$.

We now combine different $H_{NN}(g_j)$ obtained from this procedure with different time evolutions. We will take advantage of their commuting property in order to get any desired $H_{NN}(g_j)$. The operations we perform are,

$$\prod_{k=1}^{L-1} X^k e^{it_k H_{NN}(g)} X^k = e^{i\sum_{k=1}^{L-1} t_k X^k H_{NN}(g) X^k}.$$

We now focus on the exponent to obtain,

$$\begin{aligned}\sum_{j,k=1}^{L-1} t_k X^k g\sigma_z^j\sigma_z^{j+1}X^k &= t_f \sum_{j=1}^{L-1}\left(\sum_{k=1}^{L-1}\frac{t_k}{t_f}(-1)^{\delta_{jk}+\delta_{j+1k}}g\right)\sigma_z^j\sigma_z^{j+1} \\ &= t_f\sum_{j=1}^{L-1} g_j\sigma_z^j\sigma_z^{j+1} = t_f H_{NN}(g_j)\end{aligned} \tag{3.4}$$

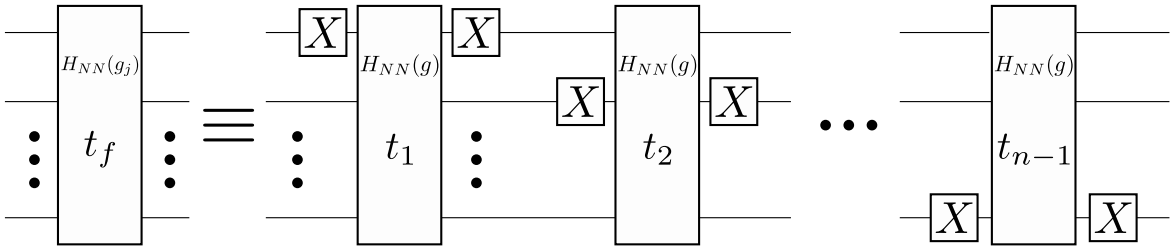


Figure 3.1: A circuit that shows how to implement the protocol to simulate an arbitrary inhomogeneous NN Ising Hamiltonian employing $L-1$ analog blocks of homogeneous NN Ising Hamiltonians, each one with different time. The homogeneous NN Hamiltonians are sandwiched by X SQR defined in Eq. (1.3).

where we used the following definition,

$$g_j = \sum_{k=1}^{L-1} \frac{t_k}{t_f} (-1)^{\delta_{jk}+\delta_{j+1k}} g = \sum_{k=1}^{L-1} M_{jk} \frac{t_k}{t_f}. \tag{3.5}$$

We defined in the last equation the matrix coefficients $M_{jk} = (-1)^{\delta_{jk}+\delta_{j+1k}} g$. Equation (3.5) shows the relation between a vector $\mathbf{g}$ and a vector $\mathbf{t}$ through a matrix M of dimension $(L-1)\times(L-1)$. The values of $\mathbf{t}$ needed for a given $\mathbf{g}$ are obtained from the inverse of M,

$$\mathbf{t} = M^{-1}\mathbf{g}, \tag{3.6}$$

where we set $t_f = 1$ for simplicity.

In Appendix A.1, we prove that the matrix M has determinant different from 0 when $L \neq 4, 5$. Thus, if $L \geq 6$, we can efficiently simulate any desired $H_{NN}(g_j)$ Hamiltonian evolving $\mathcal{O}(L-1)$ Hamiltonians $H_{NN}(g)$ for a time determined in Eq. (3.6). An example of the circuit model that implements this algorithm is shown in Fig. 3.1.

Negative times

In order to simulate the evolution of $H_{NN}(g_j)$, we sometimes require to evolve a $H_{NN}(g)$ Hamiltonian with negative time t. However, this is the same as a positive time evolution with $H_{NN}(-g)$. In order get this Hamiltonian, it suffices to sandwich the $H_{NN}(g)$ evolution with X gates in every odd qubit,

$$\prod_{k\ \mathrm{odd}} X^k \left(\sum_{j=1}^{L-1} g\sigma_z^j\sigma_z^{j+1} \right) X^k = \sum_{j=1}^{L-1} g(-1)^{\sum_{k\ \mathrm{odd}} \delta_{jk}+\delta_{j+1k}} \sigma_z^j\sigma_z^{j+1} = H_{NN}(-g). \tag{3.7}$$

Using an inhomogeneous NN Hamiltonian as resource

If instead of a homogeneous NN Hamiltonian an inhomogeneous one is available, we can still simulate an arbitrary $H_{NN}(g_j)$. However, this is only possible if the Hamiltonian used as resource has a non-zero coupling between every qubit.

In this case, we need to modify the operations performed in Eq. (3.4),

$$\begin{aligned} \sum_{j,k=1}^{L-1} t_k X^k g_j \sigma_z^j\sigma_z^{j+1} X^k &= t_f \sum_{j=1}^{L-1} \left(\sum_{k=1}^{L-1} \frac{t_k}{t_f} (-1)^{\delta_{jk}+\delta_{j+1k}} \right) g_j \sigma_z^j\sigma_z^{j+1} \\ &= t_f \sum_{j=1}^{L-1} g'_j \sigma_z^j\sigma_z^{j+1} = t_f H_{NN}(g'_j), \end{aligned} \tag{3.8}$$

where we used the following definition,

$$g'_j = \sum_{k=1}^{L-1} \frac{t_k}{t_f} (-1)^{\delta_{jk}+\delta_{j+1k}} g_j = g_j \sum_{k=1}^{L-1} M_{jk} \frac{t_k}{t_f}. \tag{3.9}$$

We continue defining $g''_j = \frac{g'_j}{g_j}$. Notice here the requirement for $g_j \neq 0$, $\forall j$, which means that every coupling in the starting Hamiltonian must be different from zero. With this last definition, we obtain a similar equation to Eq. (3.6),

$$\mathbf{t} = M^{-1}\mathbf{g''}. \tag{3.10}$$

3.2.2 Universality demonstration

In order to show the universality of the DAQC paradigm under the evolution of a $H_{NN}(g)$ Hamiltonian, we just need to show the relation between the CNOT gate and $H_{NN}(g_j)$.

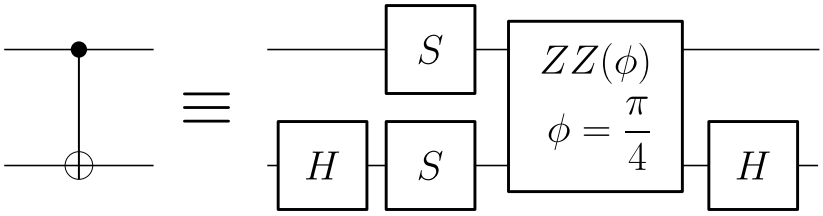


Figure 3.2: Decomposition of a CNOT gate as a $ZZ(\phi = \frac{\pi}{4})$ gate, defined in 3.11, and SQR. The definition of these SQR is shown in Eq. (1.3).

In Fig. 3.2 we show the decomposition of the CNOT gate in terms of a $ZZ(\phi)$ gate

$$ZZ\left(\phi\right) = e^{i\phi\sigma_z^1\sigma_z^2} = e^{iH_{NN}(g_j=\phi\delta_{j1})}. \tag{3.11}$$

In a similar manner, we simulate the CNOT gate between any two adjacent qubits, k and $k+1$, using the Hamiltonian $H_{NN}\left(g_j = \phi\delta_{jk}\right)$. Since the set of CNOT gates between any two adjacent qubits is a universal set of gates, the $H_{NN}\left(g\right)$ Hamiltonian is also universal.

In order to get a circuit in the DAQC paradigm we can replace each CNOT gate for its equivalent Hamiltonian evolution. However, this is by no means the most efficient solution. We need a more direct approach in order to harness the advantages of the DAQC paradigm. For example, grouping CNOT gates acting in different qubits requires to simulate only one $H_{NN}\left(g_j\right)$ Hamiltonian evolution, obtaining consequently advantage over other computational models. Another example is shown in Ref. [4], where they decomposed the quantum Fourier transform in a ATA Hamiltonian, which may be implemented using the algorithm we show in the following section.

3.3 Enhancing connectivity in quantum processors

Sometimes unitary matrices are more efficiently generated using an ATA Hamiltonian but the platform is restricted to a NN connectivity. In this section we develop an algorithm that efficiently transforms a NN Ising Hamiltonian into an ATA Ising Hamiltonian. The strategy consists in using SWAP-like gates to change the interaction between two qubits.

We start this section showing a family of gates performing a SWAP operation on an Ising Hamiltonian. We continue showing the graph representation of an Ising Hamiltonian. We finally connect all these ideas to create the algorithm that efficiently transforms a set of homogeneous $H_{NN}(g)$ Hamiltonian evolutions into an arbitrary inhomogeneous ATA Ising Hamiltonian evolution.

3.3.1 The swapping operation

In the following section, we use a gate U that performs the operation

$$\begin{aligned} U(\sigma_z \otimes I)U^\dagger &= I \otimes \sigma_z, \\ U(I \otimes \sigma_z)U^\dagger &= \sigma_z \otimes I. \end{aligned} \tag{3.12}$$

That is, it changes the Z interaction from one qubit to another. We seek a gate whose decomposition is optima in the resources used in the DAQC paradigm, that is, the number of $H_{NN}(g_j)$ Hamiltonian evolutions required.

In Appendix A.2, we prove that the unitary matrix

$$U(\alpha,\beta,\gamma) = R_z^1\left[\pi\left(\frac{\gamma-\alpha}{2}+\frac{1}{2}+\beta\right)\right] e^{i\frac{\pi}{2}(X^1X^2+Y^1Y^2+(\gamma+\alpha)Z^1Z^2)} R_z^1\left[\pi\left(\frac{\gamma-\alpha}{2}-\frac{1}{2}-\beta\right)\right] \tag{3.13}$$

is, up to a global phase factor, the three parametric family of gates that fulfils Eq. (3.12). It is noteworthy that both the SWAP and iSWAP gates belong to this family of gates. The optima decomposition of $U(\alpha,\beta,\gamma)$ is obtained when $\alpha=\gamma=0$ and $\beta=-\frac{1}{2}$. Notice that since $[X^1X^2, Y^1Y^2]=0$, we only require two $H_{NN}(g_j)$ analog blocks. It is noteworthy to mention that the gate obtained with these conditions is the iSWAP gate.

However, it is not straightforward to obtain the iSWAP operation between any two non-adjacent qubits using only the adjacent iSWAP gate, which is the gate we can still generate with the $H_{NN}(g)$ Hamiltonian. In the following section we will assume that we can perform the iSWAP gates between non-adjacent qubits, for the sake of simplicity. We will be back to this point later, showing how to build the iSWAP gates we need, using only adjacent iSWAP gates.

3.3.2 Graph interpretation of the Ising Hamiltonian

The Ising Hamiltonian for L qubits can be interpreted as a weighted graph of L vertices, where the weight of the edge which connects the vertex i to the vertex j is g_{ij}. If $g_{ij}=0$ it means that there is no edge between these two vertices.

However, let us start by working only with the homogeneous Ising Hamiltonian. This means that, instead of weighted graphs, we will be dealing with unweighted ones. In this representation, an ATA Ising Hamiltonian of L qubits becomes a complete graph K_L, a graph with edges among every the possible vertices without repetition. On the other hand, the NN Ising Hamiltonian becomes a Hamiltonian path, that is, a path that visits all the possible vertices, but just once.

In this point, it is noteworthy to mention that a Hamiltonian path can also be represented as a permutation of all the vertices in the graph. To recover a Hamiltonian path from a given vertex permutation, it suffices to connect with an edge the vertices that are adjacent in the permutation. An example of a complete graph, together with two Hamiltonian paths, are shown in Fig. 3.3, where we also show the permutation of vertices.

In this representation, our problem can be viewed as a decomposition of a K_L graph into a set of Hamiltonian paths. In the following section we show how to get this decomposition and how to obtain the Ising Hamiltonian which a given Hamiltonian path represents.

3.3.3 Obtaining an arbitrary Hamiltonian path

For the sake of clarity, we will use U_{ij} to represent the $iSWAP$ gate between qubits i and j. In this section we show how this gate transforms the $H_{NN}(g)$ Hamiltonian, obtaining a Hamiltonian whose graph representation is a Hamiltonian path. For that, we first need to generalize the transformation that U_{ij} performs.

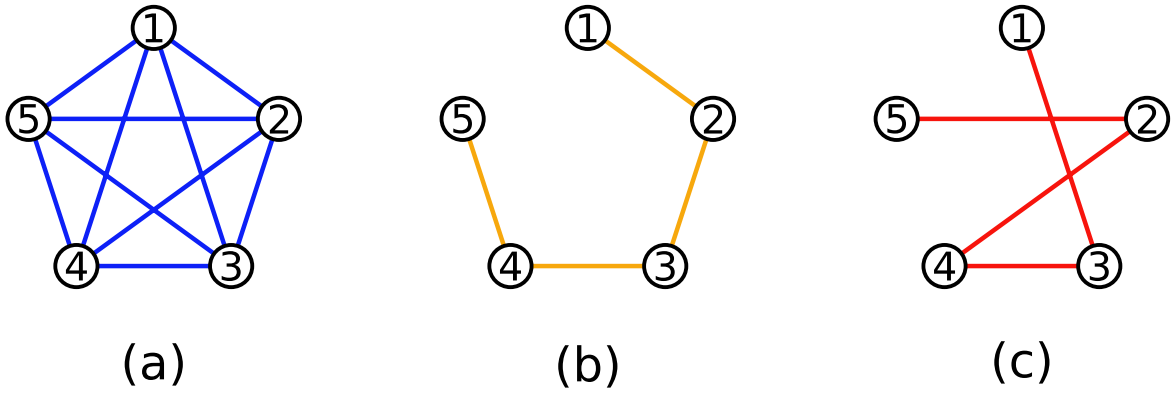


Figure 3.3: Examples of Ising Hamiltonians as their graph representation. Figure (a) shows a complete K_5 graph which represents a $H_{ATA}(g)$ Hamiltonian for 5 qubits. Figure (b) shows a Hamilton path which represents a $H_{NN}(g)$ for 5 qubits. Figure (c) shows also a Hamiltonian path with the vertex permutation $P = [1, 3, 4, 2, 5]$.

If we sandwich a $\sigma_z^k \sigma_z^l$ gate, a gate that applies the Z gate to the qubits k and l, with the gate U_{ij}, we obtain

$$U_{ij}\sigma_z^k \sigma_z^l U_{ij}^\dagger = \sigma_z^{\tau_{ij}(k)} \sigma_z^{\tau_{ij}(l)}. \tag{3.14}$$

We define the function τ_{ij} to be a permutation of the indices i and j, that is, a transposition. More precisely, if $k \neq i, j$, $\tau_{ij}(k) = k$, otherwise, $\tau_{ij}(i) = j$ and $\tau_{ij}(j) = i$. Basically, the U_{ij} gate changes σ_z gates acting on qubit i to act on qubit j.

Performing this operation in an $H_{NN}(g)$ evolution leads to $U_{ij} e^{itH(g)_{NN}} U_{ij}^\dagger = e^{itU_{ij}H(g)_{NN}U_{ij}^\dagger}$. If we focus on the exponent,

$$U_{ij}H(g)_{NN}U_{ij}^\dagger = \sum_{k=1}^{L-1} gU_{ij}\sigma_z^k \sigma_z^{k+1} U_{ij}^\dagger = \sum_{k=1}^{L-1} g\sigma_z^{\tau_{ij}(k)} \sigma_z^{\tau_{ij}(k+1)}. \tag{3.15}$$

This means that, if the initial vertex permutation was P $= [1, 2, 3, ..., L]$, the permutation after applying the U_{ij} gate is P' $= [\tau_{ij}(1), \tau_{ij}(2), \tau_{ij}(3), ..., \tau_{ij}(L)]$. This is the vertex permutation $\pi = \tau_{ij}$.

As the set of all transpositions τ_{ij} is a generator of the permutation group S_L, we can obtain, with the correct transpositions, any desired permutation. This means that, applying the correct set of U_{ij}^k gates, we can obtain any desired Hamiltonian whose path representation is obtained from the set of τ_{ij}^k transpositions. For example, the Ising Hamiltonian that represents the Fig. 3.3(c), which we will call H_1, is just obtained using the following transformations $H_1(g) = U_{23}U_{24}H_{NN}(g)U_{24}^\dagger U_{23}^\dagger$.

We now face the problem of decomposing a K_L graph into different Hamiltonian paths. This can only be achieved exactly using our method, if the system has an even number of qubits. Since each Hamiltonian path has $L - 1$ edges, and the K_L graph has $\frac{(L-1)L}{2}$ edges, we need $\frac{L}{2}$ Hamiltonian paths to exactly fill a K_L graph, which is only possible if L is even. We will assume that L is even and let for further work adapting this method for L odd.

3.3.4 Filling the complete graph

In order to fill the complete graph, we generalize the procedure exemplified in Fig. 3.4. To obtain te first Hamiltonian path start at node 1. Continue one node forward, two backward, three forward, ... until you visit each node. With this strategy we obtain one of the Hamiltonian paths we require. In order to obtain the rest of the paths, it suffices to start in the next

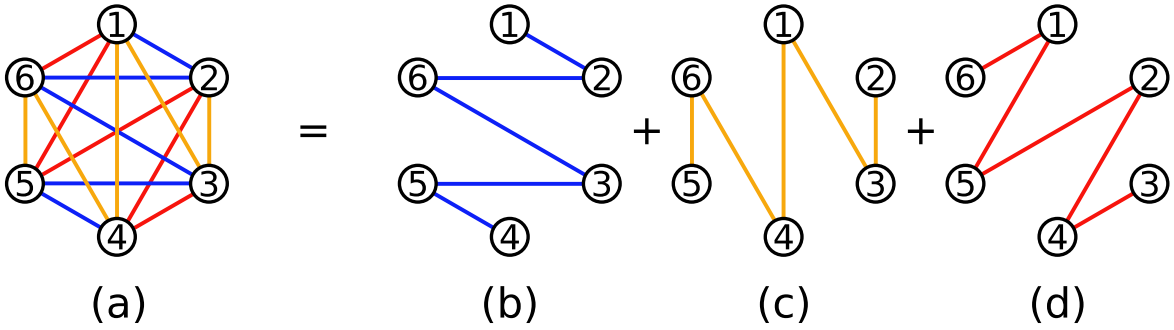


Figure 3.4: An example of how to fill exactly a complete graph of 6 vertices using Hamiltonian paths. (a) is the complete path we are trying to get. (b) is the Hamiltonian path we use. It is constructed starting in the first node, going forward to the next node, then 2 nodes backward , 3 forward... The other two Hamiltonian paths are obtained rotating the first one. The permutations for (b), (c), and (d) are $P_b = [1, 2, 6, 3, 5, 4]$, $P_c = [2, 3, 1, 4, 6, 5]$ and $P_d = [3, 4, 2, 5, 1, 6]$.

nodes. This has the graphical interpretation of rotating the first Hamilton path obtained while keeping the nodes fixed in their positions, as it is shown in Fig. 3.4 (b)-(d).

The vertex permutation of Hamiltonian paths obtained are going to be labelled P_L^k, where L is the number of vertices and k is the starting node. We will also refer to the position j of a permutation by $P_L^k(j)$. For example $P_6^1(3) = 6$, as depicted in Fig. 3.4 (b).

The analytical form of the function $P_L^k(j)$ is

$$P_L^k(j) = \begin{cases} (k - 1 + \frac{j}{2}) \bmod L \ + 1, & \text{if} \quad j \text{ even} \\ \\ (k - 1 - \frac{j-1}{2}) \bmod L \ + 1 & \text{if} \quad j \text{ odd} \end{cases} \tag{3.16}$$

In Appendix A.3 we prove that the function $P_L^k(j)$ gives the correct Hamiltonian paths, whose union generates a complete graph.

We now need to obtain an efficient decomposition of these permutations as adjacent transpositions. Once we have these adjacent transpositions, it suffices to use the evolution described in Eq. (3.15) employing the U_{ij} gates related to the transpositions τ_{ij}.

3.3.5 Efficient iSWAP decomposition

In order to obtain an efficient decomposition of the permutation P_L^k as a product of transpositions, we used the Bubble sort algorithm. This algorithm compares the value of two adjacent entries of an array and swaps them if they are not sorted. In our case, the permutation P_L^k will play the role of array.

The Bubble sort algorithm provides us with the changes in positions that sort P_L^k. However, we are looking for the transpositions, not the changes in positions. The relation between the two operations is

$$c_{j_N\, j_N} \circ c_{i_{N-1}\, j_{N-1}} \circ \cdots \circ c_{i_2\, j_2} \circ c_{i_1\, j_1} = \tau_{i_1\, j_1} \circ \tau_{i_2\, j_2} \circ \cdots \circ \tau_{i_{N-1}\, j_{N-1}} \circ \tau_{i_N\, j_N}, \tag{3.17}$$

where we denoted by $c_{i_k\, j_k}$ the k-th operation of changing the values in the positions i_k and j_k. We also used the notation $b \circ a$ to denote the composition of the operations a and b performed in this order. This means that if the set $C = c_{i_{N-1}\, j_{N-1}} \circ \cdots \circ c_{i_1\, j_1}$ is used to sort a given P_L^k permutation, the set of transpositions needed to generate the permutation P_L^k are $T = \tau_{i_N\, j_N} \circ \cdots \circ \tau_{j_1\, j_1+1}$. This relation is proven in Appendix A.4. Notice that the order of

application of these two operations does not change, unlike in Eq. (3.17). This is due to the fact that, with the set C, we are trying to short P_L^k to finally get the identity permutation $I = [1, 2, ..., L]$, while in the second case, we are employing the set T of transpositions to permute the identity I in order to get P_L^k, which is the inverse operation.

For the sake of clarity, we will work an example. If we perform the operations $\tau_{34} \circ \tau_{23}$ to the identity permutation, the permutation we get is $[1, 4, 2, 3]$. However, if we perform the operations $c_{34} \circ c_{23}$, the permutation we obtain is $[1, 3, 4, 2]$, which is clearly different from our previous permutation. On the other hand, performing the operations $c_{23} \circ c_{34}$ to the permutation $[1, 4, 2, 3]$, we obtain the ordered array, as expected.

We can take advantage of commuting adjacent transpositions to apply them simultaneously. Notice that the operation $\tau_{i\,i+1} \circ \tau_{i+2\,i+3}$ is related to the unitary matrix

$$U_{i\,i+1}U_{i+2\,i+3} = e^{i\frac{\pi}{4}(X^iX^{i+1}+X^{i+2}X^{i+3}+Y^iY^{i+1}+Y^{i+2}Y^{i+3})},$$

which can be regarded as two inhomogeneous NN Ising Hamiltonians, instead of four. Taking advantage of this commutation relation, we seek to group all the commuting transposition in the same Hamiltonian. That is why we use a parallelized version of the Bubble sort algorithm. This parallelized version of the algorithm pairwise compares and interchanges every element of an array simultaneously. From these parallel changes and the Eq. (3.17), we can obtain the desired transpositions.

It is possible to obtain the exact transpositions required by analysing how the parallelized Bubble sort algorithm works. We will define a sequence of transpositions to be

$$S_{i\to j} = \tau_{i\,i+1} \circ \tau_{i+1\,i+2} \circ \cdots \circ \tau_{j-1\,j}. \tag{3.18}$$

It is noteworthy to mention that any S_{ij} can be implemented making use of a single $H_{NN}(g_j)$ Hamiltonian evolution.

In order to get the permutation P_L^k, we need to perform the transposition sequences $G^1(k)$ and $G^2(k, L)$ defined as

$$G^1(k) = S^{2k-2}_{1\to 2} \circ S^{2k-3}_{2\to 3} \circ \cdots \circ S^4_{1\to 2k-4} \circ S^3_{2\to 2k-3} \circ S^2_{1\to 2k-2} \circ S^1_{2\to 2k-1} \tag{3.19}$$

$$G^2(k, L) = S^{L-2k-1}_{L-1\to L} \circ S^{L-2k-2}_{L-2\to L-1} \circ \cdots \circ S^4_{2k+4\to L-1} \circ S^3_{2k+3\to L} \circ S^2_{2k+2\to L-1} \circ S^1_{2k+1\to L} \tag{3.20}$$

See Appendix A.5 for the proof. Note that the two groups of sequences commute between them, so they are embedded in the same $H_{NN}(g_j)$ Hamiltonian. Besides, we labelled the sequences with their order of application, for the sake of clarity.

Regarding to the total resources required, the total amount of sequences needed to implement each of the P_L^k permutations is $\mathcal{O}(L)$. Besides, the number of $H_{NN}(g)$ analog blocks needed for each sequence is $\mathcal{O}(L)$, consequently, the total amount of analog blocks sums up to $\mathcal{O}(L^2)$.

To sum up, in order to obtain the P_L^k Ising Hamiltonian we need to apply the circuit shown in Fig. 3.5, which requires $\mathcal{O}(L^2)$ $H_{NN}(g)$ Hamiltonian evolution blocks. It is noteworthy that the total amount of SQR requires is also $\mathcal{O}(L^2)$.

3.3.6 Final circuit and total resources

The circuit shown in Fig. 3.6, allows us to obtain the ATA Ising Hamiltonian. The total amount of gates P_L^k is $\frac{L}{2}$, with L the number of qubits. That means that the algorithm developed in this section requires a total amount of $\mathcal{O}(L^3)$ $H_{NN}(g)$ Hamiltonian analog blocks. It is noteworthy to mention that it also requires $\mathcal{O}(L^3)$ SQR.

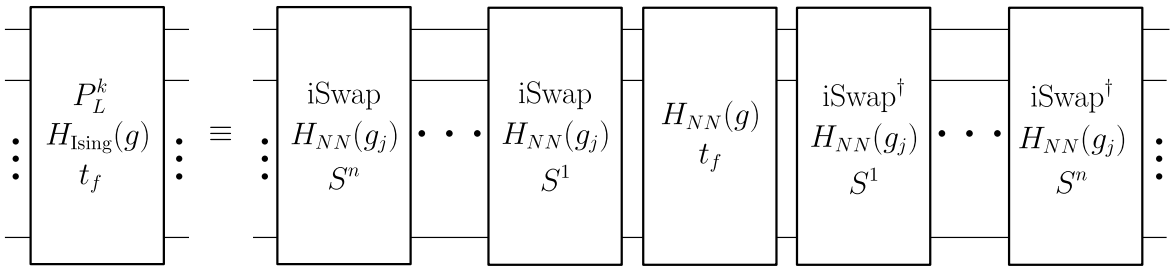


Figure 3.5: Quantum circuit simulating the evolution of an Ising Hamiltonian whose graph representation is given by the vertex permutation of P_L^k shown in Eq. (3.16). The simulation lasts a time t_f. Each block represents an analog evolution. The iSWAP analog blocks represent inhomogeneous NN Hamiltonian implementing the iSWAP gates dictated by the sequences S^i. This sequences refer to those shown in Eq. (3.19) and Eq. (3.20), which both are applied at the same time. The central Homogeneous NN analog block is applied for a time t_f. It can be substituted by an inhomogeneous one in order to get the inhomogeneous version of the P_L^k block.

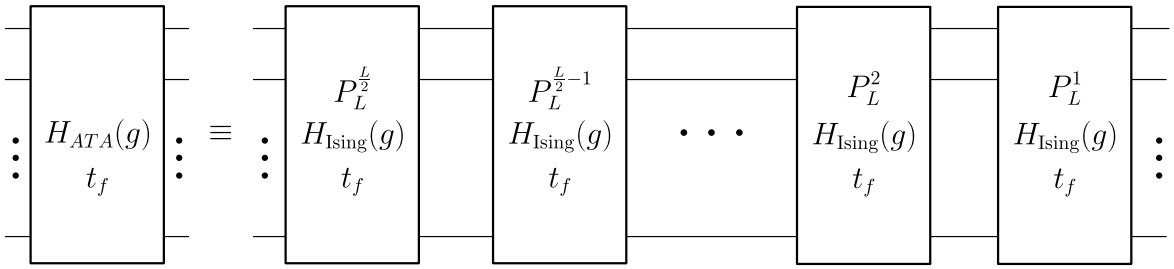


Figure 3.6: Quantum circuit simulating an all-to-all Ising Hamiltonian evolution for a time t_f. The blocks P_L^k are generated using the circuit of Fig. 3.5.

3.3.7 Extending the range of application

To extend the use of this algorithm in order to simulate the evolution of an ATA inhomogeneous Ising Hamiltonian, it suffices to change the NN homogeneous Hamiltonian blocks used in the circuit shown in Fig. 3.5 for inhomogeneous versions of them. Tuning correctly the parameters of each one of this $H_{NN}(g_j)$, we can obtain the desired $H_{ATA}(g_{ij})$.

Changing each of the $H_{NN}(g)$ Hamiltonian blocks in the circuit of Fig. 3.5 affects negligibly to the final efficiency of the circuit. It only adds $\mathcal{O}(L^2)$ SQR and $H_{NN}(g)$ Hamiltonian blocks to the total amount. Hence, the final upper bound of SQR and $H_{NN}(g)$ Hamiltonian blocks when simulating any $H_{ATA}(g_{ij})$ is still $\mathcal{O}(L^3)$.

3.4 Conclusions

In this section, we have developed an algorithm which efficiently simulates the evolution of a homogeneous ATA Ising Hamiltonian for an even number of qubits. We also show that it is possible to extend this algorithm to deal with inhomogeneous Ising ATA Hamiltonians with negligible impact on the performance. It is also possible to use as main resource an inhomogeneous NN Hamiltonian, as long as every coupling is different form zero.

The total amount of SQR and NN analog blocks needed to perform this algorithm is $\mathcal{O}(L^3)$, making this an efficient algorithm that could be employed in current technology to

artificially enhance connectivity between qubits in quantum chips.

As a further work to extend this algorithm to different architectures, one can think of next-to-nearest neighbour interaction or periodic boundary conditions of the NN Ising Hamiltonian. This last version of the Ising Hamiltonian refers to a modification of the NN case, which also has non-zero coupling between first and last qubits. It is also necessary to adapt this algorithm to the case of an odd number of qubits.

Conclusions

Quantum computing is a computation paradigm that embraces the quantum behaviour of nature to outperform classical computers in certain tasks. This has motivated the development of this field to this day, when we have already built our firsts quantum computers. It has also motivated us to develop a software based algorithm that enhances the connectivity in current quantum platforms.

We started this TFG reviewing the utility of quantum computers for information processing tasks. In the first chapter, we showed that quantum computers can simulate classical ones by proving that any classical irreversible gate may be implemented as a unitary matrix by adding ancillary qubit [1]. We continued by proving that the CNOT gate, together with SQR, constitute a set of universal gates, which means that they can be combined to generate an arbitrary unitary matrix. Finally, we also discussed the difficulties associated to performing quantum state tomography, i.e. to characterize a quantum state.

Regarding chapter 2, we explored in more detail the real advantages which quantum computers posses over classical ones. For that, we analysed the QFT, Shor's algorithm and Grover's algorithm. The implementation of the QFT was reviewed from Ref. [1], while the other two algorithms were studied from Ref. [2].

We saw that even though the QFT has a more efficient implementation than the Discrete Fourier Transform algorithm, it must be employed as part of another algorithm to harness its advantages. This is due to the inefficiency of quantum state tomography. We also reviewed Shor's algorithm, which shows an exponential speed-up compare to the best classical algorithm known. This is thanks to the ability of quantum computers to efficiently solve the period finding problem. Solving this efficiently has serious implications in modern cryptography. Regarding to Grover's algorithm, even if it only has a quadratic speed-up over classical algorithms, it has a wider range of applications than the rest of quantum algorithms shown.

In Chapter 3, we introduced the paradigm of digital-analog quantum computation (DAQC) paradigm, which aims at exploiting the natural evolution of quantum systems to perform efficient quantum computation. We proved that nearest-neighbour (NN) Ising Hamiltonian, which naturally arises in quantum platforms, leads to universal computation in the DAQC framework. We developed an efficient algorithm to simulate the evolution of a L spin all-to-all (ATA) Ising Hamiltonian using $\mathcal{O}(L^3)$ SQR and NN analog blocks. Finally, we discussed how to extend the range of application of this algorithm to generate an arbitrary inhomogeneous ATA Ising Hamiltonian. This algorithm can be used to improve the performance of algorithms which admit a natural decomposition in terms of ATA Hamiltonians, Ref. [4]. Further work is related to adapt the algorithm to the case of an odd number of qubits. Besides, it could also be extended to different quantum computer architectures, for instance, one with next-to-nearest neighbour interaction or periodic boundary conditions of the NN Hamiltonian.

Quantum computers are still in their infancy, somehow comparable with the first vacuum tube computers. Even though quantum computers are still far from quantum supremacy, i.e, from performing calculations beyond any classical computer, the perspectives are exciting and thrilling. Indeed, we may not see quantum computers in our daily life in the incoming

years, but the astonishing developments in this field are incredibly promising.

Bibliography

[1] M. A. Nielsen and I. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. (Cambridge University Press, Cambridge, United Kingdom, 2011).

[2] R. de Wolf. *Quantum Computing: Lecture Notes*. Amsterdam, January 2019. `https://homepages.cwi.nl/~rdewolf/qcnotes.pdf`

[3] A. Parra-Rodriguez, P. Lougovski, L. Lamata, E. Solano, and M. Sanz. Digital-analog quantum computation. *arXiv:1812.03637*, 2018.

[4] A. Martin, L. Lamata, E.Solano, and M. Sanz. Digital-analog quantum algorithm for the quantum Fourier transform. *arXiv:1906.07635*, 2019.

Appendices

Appendix A

Proofs for Chapter 3

A.1 Invertible matrix

In this Appendix we compute the determinant of the matrix shown in Eq. (3.6), hence proving that it is invertible.

The most general determinant $N \times N$ that we need to calculate is,

$$\begin{vmatrix} -1 & -1 & 1 & 1 & \cdots & 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 & \cdots & 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 & \cdots & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 & \ddots & 1 & 1 & 1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots \\ 1 & 1 & 1 & 1 & \ddots & -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 & \cdots & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & \cdots & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & \cdots & 1 & 1 & 1 & -1 \end{vmatrix}.$$

We start by doing the operation $f_i = f_i - f_{i+1}$, $i \in [1, N-1]$, where f_i denotes the i-th row of the determinant. After performing this operation, we obtain the determinant

$$\begin{vmatrix} -2 & 0 & 2 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 2 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & \ddots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ddots & -2 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -2 & 0 & 2 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -2 & 0 \\ 1 & 1 & 1 & 1 & \cdots & 1 & 1 & 1 & -1 \end{vmatrix}.$$

We continue taking out all the 2 factors, obtaining

$$2^{N-1}\begin{vmatrix} -1 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & \ddots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ddots & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -1 & 0 \\ 1 & 1 & 1 & 1 & \cdots & 1 & 1 & 1 & -1 \end{vmatrix}.$$

We now perform the operation $c_{j+2} = c_j + c_{j+2}$ for $j \in [1, N-2]$, where c_j denotes j-th column of the determinant, obtaining,

$$2^{N-1}\begin{vmatrix} -1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & \ddots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ddots & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -1 & 0 \\ a_1 & a_2 & a_3 & a_4 & \cdots & a_{N-3} & a_{N-2} & a_{N-1} & a_N \end{vmatrix},$$

where a_k is a succession that determines the k-th coefficient of the N-th row. This last determinant is lower triangular. Therefore, the determinant will be $\det(M) = 2^{N-1}a_N$.

Due to the column operations, a_k satisfies $a_k = a_{k-2} + 1$ for $k \neq N$, with $a_N = a_{k-1} - 1$. It can be checked by substitution that the explicit form for the succession is $a_k = \lceil \frac{k}{2} \rceil - 2\delta_{k,N}$ for $k \in [1, N]$. Thus, the determinant is $\det(M) = 2^{N-1}\left(\lceil \frac{N}{2} \rceil - 2\right)$, which is different from zero for $N \neq 3, 4$.

A.2 Unitary transform

In this chapter we will show that the unitary gate iSWAP is indeed the most simple gate, i.e the gate with fewer ZZ gates and SQR involved, that fulfils Eq. (3.12).

For each condition of Eq. (3.12) we get a unitary matrix of the form

$$\begin{pmatrix} a & 0 & a & 0 \\ a & 0 & a & 0 \\ 0 & a & 0 & a \\ 0 & a & 0 & a \end{pmatrix}, \qquad \begin{pmatrix} a & a & 0 & 0 \\ 0 & 0 & a & a \\ a & a & 0 & 0 \\ 0 & 0 & a & a \end{pmatrix}, \tag{A.1}$$

where each a is an arbitrary complex number. Thus, in order to fulfil both equation, the most general unitary matrix, up to a global phase factor, must be

$$U(\alpha, \beta, \gamma) = \begin{pmatrix} e^{i\pi\alpha} & 0 & 0 & 0 \\ 0 & 0 & e^{-i\pi\beta} & 0 \\ 0 & e^{i\pi\beta} & 0 & 0 \\ 0 & 0 & 0 & e^{i\pi\gamma} \end{pmatrix}. \tag{A.2}$$

It can be proven by substitution that the decomposition

$$U(\alpha,\beta,\gamma) = R_z^1\left[\pi\left(\frac{\gamma-\alpha}{2}+\frac{1}{2}+\beta\right)\right] e^{i\frac{\pi}{2}(X^1X^2+Y^1Y^2+(\gamma+\alpha)Z^1Z^2)} R_z^1\left[\pi\left(\frac{\gamma-\alpha}{2}-\frac{1}{2}-\beta\right)\right] \tag{A.3}$$

is indeed the gate shown in Eq. (A.2). We now set $\alpha = \gamma = 0$ $\beta = -\frac{1}{2}$ to have the lesser amount of SQR and ZZ gates. This leads to an optima representation of U using inhomogeneous Ising Hamiltonians.

A.3 Permutations proof

In this appendix we prove that the set of Hamiltonian paths $P_L = \{P_L^k(j)\}_{k=1}^{\frac{L}{2}}$ with $P_L^k(j)$ defined in Eq. (3.16), have as union the complete graph K_L and as intersection an empty graph.

Let us denote by $q_k(t)$ the function that holds $P_L^k(q_L^k(t))) = t$, that is, $q_L^k(t)$ is the position of the number t in the permutation $P_L^k(t)$. Let us also define $R_k(t) = P^k\left(q_L^k(t)+1\right)$ and $L_L^k(t) = P^k(q_L^k(t)-1)$, which are the numbers adjacent to t at the right side and left side of t in P_L^k. Recall that the edges of a Hamiltonian path are obtained from the adjacent numbers in a given vertex permutation. Thus, the proof consist in obtaining an explicit representation of both $R_L^k(t)$ and $L_L^k(t)$ and proving that no number adjacent to every number $t \in [1, L]$ repeats in the set of permutations P_L. This means that $R_L^k(t) \neq L_L^{k'}(t)$ for $k, k' \in [1, \frac{L}{2}]$. Besides, we also need to proof that $L_L^k(t) \neq L_L^{k'}(t)$ and $R_L^k(t) \neq R_L^{k'}(t)$.

We first proof that,

$$q_L^k(t) = \begin{cases} 2(t-k) & \text{if} \quad 0 < t-k \leq \frac{L}{2} \\ 2(k-t)+1 & \text{if} \quad -\frac{L}{2} < t-k \leq 0 \\ 2(k-t+L)+1 & \text{if} \quad \frac{L}{2} < t-k < L \end{cases} \tag{A.4}$$

Substituting this expression in $P_L^k(q_L^k(t))$ must return, by definition, t.

$$\begin{aligned} P_L^k(q_L^k(t)) &= \begin{cases} (k-1+\frac{2(t-k)}{2}) \mod L+1 & \text{if} \quad 0 < t-k \leq \frac{L}{2} \\ (k-1-\frac{2(k-t)+1-1}{2}) \mod L+1 & \text{if} \quad -\frac{L}{2} < t-k \leq 0 \\ (k-1-\frac{2(k-t+L)+1-1}{2}) \mod L+1 & \text{if} \quad \frac{L}{2} < t-k < L \end{cases} \\ &= \begin{cases} (t-1) \mod L+1 & \text{if} \quad 0 < t-k \leq \frac{L}{2} \\ (t-1) \mod L+1 & \text{if} \quad -\frac{L}{2} < t-k \leq 0 \\ (t-1-L) \mod L+1 & \text{if} \quad \frac{L}{2} < t-k < L \end{cases} \\ &= (t-1) \mod L+1 = t. \end{aligned}$$

In the last equality, recall that $t \in [1, L]$, and hence, $t-1 > L\ \forall t$.

We continue obtaining an explicit representation for $R_L^k(t)$ and $L_L^k(t)$ by substituting the expression obtained for $q_L^k(t)$.

$$R_L^k(t) = \begin{cases} (2k-t-1) \mod L+1 & \text{if} \quad 0 < t-k \leq \frac{L}{2} \\ (2k-t) \mod L+1 & \text{if} \quad -\frac{L}{2} < t-k \leq 0 \text{ and } \frac{L}{2} < t-k < L \end{cases},$$

$$L_L^k(t) = \begin{cases} (2k-t) \mod L+1 & \text{if} \quad 0 < t-k \leq \frac{L}{2} \\ (2k-t-1) \mod L+1 & \text{if} \quad -\frac{L}{2} < t-k \leq 0 \text{ and } \frac{L}{2} < t-k < L \end{cases}.$$

For the shake of clarity, I will define the two intervals $I_1 = [1, \frac{L}{2}]$ and $I_2 = \left[-\frac{L}{2}+1, 0\right] \bigcup \left[\frac{L}{2}+1, L-1\right]$.

With this two definitions, the equations for $R_L^k(t)$ and $L_L^k(t)$ are

$$R_L^k(t) = \begin{cases} (2k-t-1) \mod L+1 & \text{if } t-k \in I_1 \\ (2k-t) \mod L+1 & \text{if } t-k \in I_2 \end{cases}, \tag{A.5}$$

$$L_L^k(t) = \begin{cases} (2k-t) \mod L+1 & \text{if } t-k \in I_1 \\ (2k-t-1) \mod L+1 & \text{if } t-k \in I_2 \end{cases}. \tag{A.6}$$

The next step is to prove $R_L^k(t) \neq R_L^{k'}(t)$ for $k \neq k'$. We distinguish two cases:

- **Case 1:** $k-t, k'-t \in I_1$ or $k-t, k'-t \in I_2$. In order to have $R_L^k(t) = R_L^{k'}(t)$, it needs to hold $(2(k-k')) \mod L = 0$. However, this is only possible for $k = k'$.
- **Case 2:** $k-t \in I_1$ ans $k'-t \in I_2$ or $k-t \in I_2$ ans $k'-t \in I_1$. In order to have $R_L^k(t) = R_L^{k'}(t)$, we need to solve $(2(k-k')) \mod L = 1$. This leads to $k = k' + m\frac{L}{2} + \frac{1}{2}$, for $m \in \mathbb{N}$. This leads to a contradiction because $k \in [1, \frac{L}{2}]$.

Performing similar steps, we can prove both $L_L^k(t) \neq L_L^{k'}(t)$ and $R_L^k(t) \neq L_L^{k'}(t)$ for $k \neq k'$. These proofs require the same steps to be completed, reaching to the same contradictions.

A.4 Relation between transpositions and position changes

In this section we proof Eq. (3.17) by induction. For that, we need to prove first the following equation,

$$c_{ij} \circ \pi = \pi \circ \tau_{ij}, \tag{A.7}$$

where c_{ij} is the change of position of the elements i and j in an sorted set; π is a permutation, more specifically, a bijection between two equal sets of natural numbers, and τ_{ij} is a transposition, a permutation between i and j. We use $b \circ a$, the composition of functions, to denote that we apply the operations a and b in this order.

We denote by $\pi(l)$ and $\tau(l)$ the natural number obtained when applying these operations to l, which is also a natural number. We define $\pi' = \pi \circ \tau_{ij}$ and, since

$$\tau_{ij}(l) = \begin{cases} i & \text{if } l = j \\ j & \text{if } l = i \\ l & \text{if } l \neq i, j \end{cases}, \tag{A.8}$$

the composition of permutations π' becomes

$$\pi'(l) = \begin{cases} \pi(i) & \text{if } l = j \\ \pi(j) & \text{if } l = i \\ \pi(l) & \text{if } l \neq i, j \end{cases}. \tag{A.9}$$

However, this is, by definition, $c_{ij} \circ \pi$. Recall that c_{ij} changes the value in the position i, $\pi(i)$, for the value in the position j, $\pi(j)$. It also changes the value in the position j, $\pi(j)$, for the value $\pi(i)$.

Now, we define $C_N = c^N \circ c^{N-1} \circ \cdots \circ c^2 \circ c^1$ and $T_N = \tau^1 \circ \tau^2 \circ \cdots \circ \tau^{N-1} \circ \tau^N$, where τ^k and c^k refers to $\tau_{i_k\, j_k}$ and $c_{i_k\, j_k}$ respectively. We need to prove the that $C_N = T_N$.

We proof this using induction. By definition of these operations, $T_1 = C_1$. We now suppose that $T_{N-1} = C_{N-1}$ and we prove that $T_N = C_N$,

$$C_N = c^N \circ C_{N-1} = c^N \circ T_{N-1} = T_{N-1} \circ \tau^N = T_N. \tag{A.10}$$

This completes the proof of Eq. (3.17).

A.5 Sequences proof

In this appendix we proof that the combination of sequences $G^1(k)$ and $G^2(k, L)$, defined in Eq. (3.19), generate P_L^k. This means that the same sequences defined with the c_{ij} operations instead of the $\tau_{i,j}$ operations order the permutation P_L^k. Thus, we need to prove $G^1(k) \circ G^2(k, L) \circ P_L^k = 1$, where 1 stands for the identity permutation and we use $b \circ a$, the composition of functions, to denote that we apply the operations a and b in this order.

First note that,

$$P_L^k(j) \begin{cases} P_{2k}^k & \text{if} \quad j \leq 2k \\ P_{L-2k}^{L-2k} + 2k & \text{if} \quad j > 2k \end{cases}, \tag{A.11}$$

that is, we need to order two independent permutations. This is why two commuting groups of sequences, $G^1(k)$ and $G^2(k, L)$, arise. Note that the second permutation, which can be regarded as $P_{L'}^{L'}$ with $L' = L - 2k$, is ordered using

$$G'^2(L') = S_{L'-1\to L'}^{L'-1} \circ S_{L'-2\to L'-1}^{L'-2} \circ \cdots \circ S_{4\to L'-1}^4 \circ S_{3\to L'}^3 \circ S_{2\to L'-1}^2 \circ S_{1\to L'}^1. \tag{A.12}$$

$G'^2(L')$ differs from $G^2(k, L)$ by a factor of $2k$ that appears in all the sequences. This extra factor in $G^2(k, L)$ comes from Eq. (A.11), where it appears adding to the permutation P_{L-2k}^{L-2k}. It has de effect of changing the action of all transpositions from τ_{ij} to $\tau_{i+2k\, j+2k}$.

I will only prove how to order the permutation that $G^1(k) \circ P_{2k}^k = 1$ because proving that $G'^2(k, L') \circ P_{L'-2k}^{L'} = 1$ requires the same steps. We prove this by induction. First, $G^1(1)$ is the identity operation, but P_2^1 is also the identity permutation so it is clear that $G^1(1) \circ P_2^1 = 1$.

We now suppose that $G^1(k-1) \circ P_{2k-2}^{k-1} = 1$ and we prove that, with this condition, $G^1(k) \circ P_{2k}^k = 1$. Since $G^1(k) \circ P_{2k}^k = G^1(k-1) \circ S_{1\to 2k-2}^2 \circ S_{2\to 2k-1}^1 \circ P_{2k}^k$, it suffices to prove $S_{1\to 2k-2}^2 \circ S_{2\to 2k-1}^1 \circ P_{2k}^k = P_{2k-2}^{k-1}$. Notice that $S_{2\to 2k-1}^1$ has the effect of changing the position of all the entries in P_L^k except for the last and the first one. All the numbers in an odd position change to their left position and all the number in a even position change to their right position. Hence, the permutation obtained from $\pi_1 = S_{2\to 2k-1}^1 \circ P_{2k}^k$ is

$$\begin{aligned} \pi_1(j) &= \begin{cases} P_{2k}^k(j+1) & \text{if} \quad j \text{ even} \\ P_{2k}^k(j-1) & \text{if} \quad j \text{ odd} \\ P_{2k}^k(1) & \text{if} \quad j = 1 \\ P_{2k}^k(2k) & \text{if} \quad j = 2k \end{cases} \\ &= \begin{cases} P_{2k}^k(j+1) & \text{if} \quad j \text{ even} \\ P_{2k}^k(j-1) & \text{if} \quad j \text{ odd} \\ 2k & \text{if} \quad j = 2k \end{cases}, \end{aligned}$$

where we used the property $P_{2k}^k(0) = P_{2k}^k(1)$.

We now compute the operation $\pi_2 = S_{1\to 2k-2}^2 \circ \pi_1$, which changes the position of all the number except from the last two. Hence,

$$\begin{aligned} \pi_2(j) &= \begin{cases} \pi_1(j-1) & \text{if} \quad j \text{ even} \\ \pi_1(j+1) & \text{if} \quad j \text{ odd} \\ \pi_1(2k-1) & \text{if} \quad j = 2k-1 \\ \pi_1(2k) & \text{if} \quad j = 2k \end{cases} \\ &= \begin{cases} P_{2k}^k(j-2) & \text{if} \quad j \text{ even} \\ P_{2k}^k(j+2) & \text{if} \quad j \text{ odd} \\ 2k-1 & \text{if} \quad j = 2k-1 \\ 2k & \text{if} \quad j = 2k \end{cases} \\ &= \begin{cases} P_{2k-2}^{k-1}(j) & \text{if} \quad j < 2k-1 \\ 2k-1 & \text{if} \quad j = 2k-1 \\ 2k & \text{if} \quad j = 2k \end{cases} \end{aligned}$$

Since $G^1(k-1)$ does not affect to the positions $2k$ and $2k-1$,

$$G^1(k-1) \circ \pi_1(j) = \begin{cases} G^1(k-1) \circ P_{2k-2}^{k-1}(j) & \text{if} \quad j < 2k-1 \\ 2k-1 & \text{if} \quad j = 2k-1 \\ 2k & \text{if} \quad j = 2k \end{cases}$$
$$= 1$$

This completes the proof.

Appendix B

Programs

B.1 Mathematica program

This program has been used to prove Eq. (3.14) and to check the veracity of Fig. 3.2.

This document aims at showing some of the operations performed to obtain some of the results discussed in the TFG.

Definition of gates and operations

In[81]:=
```
com[x_, y_] := x.y - y.x
```

In[82]:=
```
acom[x_, y_] := x.y + y.x
GetGenerator[m_] := -I MatrixLog[m]
                    · logaritmo matricial
```

In[84]:=
```
$PrePrint = If[MatrixQ[#], MatrixForm[#], #] &;
pre-escribe  si  ¿matriz?  forma de matriz
```

In[85]:=
```
z = {{1, 0}, {0, -1}};
x = {{0, 1}, {1, 0}};
y = {{0, -I}, {I, 0}};
h = HadamardMatrix[2];
    matriz Hadamard
s = {{1, 0}, {0, I}};
st = ConjugateTranspose[s];
     transpuesto conjugado
i = IdentityMatrix[2];
    matriz identidad
```

$$sw = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$cnot = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix};$$

In[94]:=
```
ZZ[phi_] := MatrixExp[-I phi KroneckerProduct[z, z]];
            exponencial···  nú···  producto Kronecker
Z[phi_] := {{1, 0}, {0, Exp[I phi]}};
```

In[96]:=
```
U[β_, δ_, γ_] := {{Exp[-I (β + δ) / 2] Cos[γ / 2], Exp[-I (β - δ) / 2] Sin[γ / 2]},
                  exp···  número i  coseno  exp···  número i  seno
  {Exp[I (β - δ) / 2] Sin[γ / 2], Exp[I (β + δ) / 2] Cos[γ / 2]}}
   ex···  número i  seno  ex···  número i  coseno
```

In[97]:=
```
MatrixBasis = List[i, x, y, z];
              lista
MatrixBasis2 =
  Table[KroneckerProduct[a, b], {a, MatrixBasis}, {b, MatrixBasis}];
  tabla  producto Kronecker
```

Obtaining the unitary gate that transfors a ZxI gate

In[99]:=
```
u = {{Exp[I α π], 0, 0, 0},
     {0, 0, Exp[-I β π], 0},
     {0, Exp[I β π], 0, 0},
     {0, 0, 0, Exp[I γ π]}}
```

Out[99]=
$$\begin{pmatrix} e^{i\pi\alpha} & 0 & 0 & 0 \\ 0 & 0 & e^{-i\pi\beta} & 0 \\ 0 & e^{i\pi\beta} & 0 & 0 \\ 0 & 0 & 0 & e^{i\pi\gamma} \end{pmatrix}$$

In[100]:=
```
rotZ1 = MatrixExp[I z π / 2 (1 / 2 + β)].MatrixExp[I z π / 4 (x1 - x2)]
```
exponencia··· número i exponencia··· número i

Out[100]=
$$\begin{pmatrix} e^{\frac{1}{4} i \pi (x1-x2)+\frac{1}{4} i \pi (1+2\beta)} & 0 \\ 0 & e^{-\frac{1}{4} i \pi (x1-x2)-\frac{1}{4} i \pi (1+2\beta)} \end{pmatrix}$$

In[101]:=
```
rotZ2 = MatrixExp[-I z π / 2 (1 / 2 + β)].MatrixExp[I z π / 4 (x1 - x2)]
```
exponencial··· número i exponencia··· número i

Out[101]=
$$\begin{pmatrix} e^{\frac{1}{4} i \pi (x1-x2)-\frac{1}{4} i \pi (1+2\beta)} & 0 \\ 0 & e^{-\frac{1}{4} i \pi (x1-x2)+\frac{1}{4} i \pi (1+2\beta)} \end{pmatrix}$$

In[102]:=
```
rotZZ = MatrixExp[I KroneckerProduct[z, z] π (x1 + x2) / 4]
```
exponencia·· · producto Kronecker

Out[102]=
$$\begin{pmatrix} e^{\frac{1}{4} i \pi (x1+x2)} & 0 & 0 & 0 \\ 0 & e^{-\frac{1}{4} i \pi (x1+x2)} & 0 & 0 \\ 0 & 0 & e^{-\frac{1}{4} i \pi (x1+x2)} & 0 \\ 0 & 0 & 0 & e^{\frac{1}{4} i \pi (x1+x2)} \end{pmatrix}$$

In[103]:=
```
M = Exp[I π (x1 + x2) / 4]
```
ex··· número i

```
KroneckerProduct[rotZ2, i].iswap.rotZZ.KroneckerProduct[rotZ1, i] // FullSimplify
```
producto Kronecker producto Kronecker simplifica completamen

Out[103]=
$$\begin{pmatrix} e^{i\pi x1} & 0 & 0 & 0 \\ 0 & 0 & e^{-i\pi\beta} & 0 \\ 0 & -e^{i\pi\beta} & 0 & 0 \\ 0 & 0 & 0 & e^{i\pi x2} \end{pmatrix}$$

CNOT to ZZ

In[104]:=
```
rotZZ = MatrixExp[I KroneckerProduct[z, z] π / 4]
```
exponencia·· · producto Kronecker

Out[104]=
$$\begin{pmatrix} e^{\frac{i\pi}{4}} & 0 & 0 & 0 \\ 0 & e^{-\frac{i\pi}{4}} & 0 & 0 \\ 0 & 0 & e^{-\frac{i\pi}{4}} & 0 \\ 0 & 0 & 0 & e^{\frac{i\pi}{4}} \end{pmatrix}$$

In[112]:=
```
Cz = E^(-I π / 4) KroneckerProduct[s, s].rotZZ
```
nú··· número i producto Kronecker

Out[112]=
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

In[114]:=
```
KroneckerProduct[i, h].Cz.KroneckerProduct[i, h]
```
producto Kronecker producto Kronecker

Out[114]=
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

B.2 Python program

This is a Python program used for checking the definitions of Eq. (3.16) and Eq. (3.19), along with the proofs of appendices A.3 and A.5.

This is a program written in Python 3.7. It has been employed to check that some of the definitions were correct. More precisely, the definitions of appendix A.3 and A.5.

```
In [1]: import numpy as np
```

```
In [2]: def createPermutations(numberQubits):
            if numberQubits%2 != 0:
                raise Exception("The number of qubits: {} must be even. ".format(numberQubits))

            permutations = [[] for i in range(numberQubits//2)]
            j = 0
            for per in permutations:
                n = j
                sign = 1
                for i in range(numberQubits):
                    sign *= -1
                    n = (n +sign* i) % numberQubits
                    per.append(n+1)
                j += 1
            return permutations
        createPermutations(8)
```

```
Out[2]: [[1, 2, 8, 3, 7, 4, 6, 5],
         [2, 3, 1, 4, 8, 5, 7, 6],
         [3, 4, 2, 5, 1, 6, 8, 7],
         [4, 5, 3, 6, 2, 7, 1, 8]]
```

```
In [3]: def createPermutations2(numberQubits):
            #create permutations for k = N
            if numberQubits%2 != 0:
                raise Exception("The number of qubits: {} must be even.".format(numberQubits))

            permutations = []

            n = numberQubits-1
            sign = 1
            for i in range(numberQubits):
                sign *= -1
                n = (n +sign* i) % numberQubits
                permutations.append(n+1)

            return permutations
        print(createPermutations2(10))
```

```
[10, 1, 9, 2, 8, 3, 7, 4, 6, 5]
```

```
In [4]: def createPermutations3(numberQubits):
            #create permutations for k = N/2
```

```
            if numberQubits%2 != 0:
                raise Exception("The number of qubits: {} must be even.".format(numberQubits))

            permutations = []

            n = numberQubits//2-1
            sign = 1
            for i in range(numberQubits):
                sign *= -1
                n = (n +sign* i) % numberQubits
                permutations.append(n+1)

            return permutations
        print(createPermutations3(10))
```

```
[5, 6, 4, 7, 3, 8, 2, 9, 1, 10]
```

```
In [5]: def createAdjacentTranspositions(permutation):
            per = permutation
            length = len(per)
            sorted_per = False
            start = 0
            times_nosort_perfomed = 0
            adj_trans  = []
            n = 0


            while not sorted_per and times_nosort_perfomed != 2:
                sorted_per = True

                for start in range(2):
                    parallel_trans = []
                    for i in range(start,length-1,2):
                        if per[i] >= per[i+1]:
                            per[i],per[i+1] = per[i+1],per[i]
                            parallel_trans.append([i+1,i+2])
                            sorted_per = False

                    if len(parallel_trans) != 0:
                        adj_trans.append(parallel_trans)
                n+=1
                if n>length*10:
                    raise Exception('Ciclo demasiado largo')
            return adj_trans

        permutations = createPermutations(12)
        per = permutations[0]
        createAdjacentTranspositions(per)
```

```
Out[5]: [[[3, 4], [5, 6], [7, 8], [9, 10], [11, 12]],
         [[4, 5], [6, 7], [8, 9], [10, 11]],
         [[5, 6], [7, 8], [9, 10], [11, 12]],
         [[6, 7], [8, 9], [10, 11]],
         [[7, 8], [9, 10], [11, 12]],
         [[8, 9], [10, 11]],
         [[9, 10], [11, 12]],
         [[10, 11]],
         [[11, 12]]]
```

```
In [6]: #This Funtion test if a given set of permutations can be used to obtain the ATA Ising.
        def testIsing(n):
            qubits = n

            permutations = createPermutations(qubits)

            adj_trans = []
            for per in permutations:
                adj_trans.append(createAdjacentTranspositions(per.copy()))
            ising = [Ising(qubits) for i in range(qubits//2)]
            for i in range(len(ising)):
                set_trans = adj_trans[i]

                for trans in set_trans:
                    ising[i].transSeq(trans)


            ising_ata = np.zeros((qubits,qubits))

            for i in ising:
                for term in i.terms:
                    k,l = term-1
                    ising_ata[k,l] += 1
                    ising_ata[l,k] += 1
            check_with = np.ones((qubits,qubits))
            check_with -= np.diag(np.ones(qubits))
            return (ising_ata == check_with).all()
```

```
In [7]: def convertTranspositionInSequence(t):
            seq = []
            seq.append(t[0])
            for i in range(1,len(t)):
                first, last = t[i]
                if seq[-1][1] == first-1:
                    seq[-1][1] = last
                else:
                    seq.append(t[i])
            return seq
```

```
permutations = createPermutations(20)
per = permutations[9]
adt = createAdjacentTranspositions(per)
#for ad in adt:
    #print('the ad transposition is:')
    #print(ad)
    #print('The sequence is')
    #print(convertTranspositionInSequence(ad))
```

The funciton createPermutation returns an array of the permutations need to transform an NN Ising into an ATA Ising.

The function createAdjacentTransposition is employed to sort a given permutation using the Bubble sort algorithm. It returns the operations need to permor as transpositions.

The function convertTranspositionInSequence is used to understand better the output of the previous function.

0.0.1 Definiciones

P_L^k is the permutation k para L qubits. $k \in [1, L/2]$.

$\tau_{i,j}$ is the transposition between i and j. For example: $\tau_{1,3}$ [3,4,1,5,2] = [1,4,3,5,2]

If only one idex is used, then the transposition is performed betweeen adjacent indices $\tau_i = \tau_{i,i+1}$.

A set of transpositons that commute will be called sequence: $S_{i,j} = \tau_i \tau_{i+2} \tau_{i+4} ... \tau_{j-1}$

0.0.2 Examples

Example of permutations

```
In [8]: nqubits = 6
        P_6 = createPermutations(nqubits)
        print('Permutations for {} qubits:\n {} '.format(nqubits,np.array(P_6)))
```

```
Permutations for 6 qubits:
 [[1 2 6 3 5 4]
 [2 3 1 4 6 5]
 [3 4 2 5 1 6]]
```

Example of adjacent transpositions $[i, j] = \tau_{i,j}$

```
In [9]: for permutation in P_6:
            print('For the permutation: {}\n'.format(permutation))
            print('The transpositions that need to be applied are:')
            adj = createAdjacentTranspositions(permutation)
            for i,conmutingTransposit in enumerate(adj):
                print(i+1,' of ',len(adj))
                print(conmutingTransposit)
```

```
For the permutation: [1, 2, 6, 3, 5, 4]

The transpositions that need to be applied are:
1  of  3
[[3, 4], [5, 6]]
2  of  3
[[4, 5]]
3  of  3
[[5, 6]]
For the permutation: [2, 3, 1, 4, 6, 5]

The transpositions that need to be applied are:
1  of  3
[[5, 6]]
2  of  3
[[2, 3]]
3  of  3
[[1, 2]]
For the permutation: [3, 4, 2, 5, 1, 6]

The transpositions that need to be applied are:
1  of  4
[[2, 3], [4, 5]]
2  of  4
[[1, 2], [3, 4]]
3  of  4
[[2, 3]]
4  of  4
[[1, 2]]
```

If we represent everything using the definitions mentioned above, these are all sequences. I will use the following notation: $S_{i,j}$ = S:i->j. I used this to came up with the sequences shown in the TFG.

```
In [46]: #CARE: k wrongly defined. It still works -> k = k-1.
         for n in range(6,8,2):
             print('Qubits = ',n)
             P = createPermutations(n)
             for k,permutation in enumerate(P):
                 print('For the permutation with {} qubits and k = {} : {}\n'.format(n,k+1,np.ar
                 print('The transpositions that need to be applied are:')
                 adj = createAdjacentTranspositions(permutation)
                 for i,conmutingTransposit in enumerate(reversed(adj)):
                     print(i+1,' of ',len(adj),'\n')
                     sequences = convertTranspositionInSequence(conmutingTransposit)
                     for seq in sequences:
                         print('S: {} -> {}'.format(seq[0],seq[1]))
```

```
                    print('--------- \n')
                print(np.array(permutation)+1)
            print('-------------------------------------------')
```

```
Qubits =  6
For the permutation with 6 qubits and k = 1 : [2 3 7 4 6 5]

The transpositions that need to be applied are:
1  of  3

S: 5 -> 6
---------

2  of  3

S: 4 -> 5
---------

3  of  3

S: 3 -> 6
---------

[2 3 4 5 6 7]
For the permutation with 6 qubits and k = 2 : [3 4 2 5 7 6]

The transpositions that need to be applied are:
1  of  3

S: 1 -> 2
---------

2  of  3

S: 2 -> 3
---------

3  of  3

S: 5 -> 6
---------

[2 3 4 5 6 7]
For the permutation with 6 qubits and k = 3 : [4 5 3 6 2 7]

The transpositions that need to be applied are:
1  of  4
```

```
S: 1 -> 2
---------

2  of  4

S: 2 -> 3
---------

3  of  4

S: 1 -> 4
---------

4  of  4

S: 2 -> 5
---------

[2 3 4 5 6 7]
--------------------------------------------
```

In [41]:
```
#Sortin k=N. Output reduced for the sake of clarity.

        for n in range(6,10,2):
            t = createAdjacentTranspositions(createPermutations2(n))
            for per in t:
                print(per)
            print('\n ---------------\n')
```

```
[[1, 2], [3, 4], [5, 6]]
[[2, 3], [4, 5]]
[[3, 4], [5, 6]]
[[4, 5]]
[[5, 6]]

 ---------------

[[1, 2], [3, 4], [5, 6], [7, 8]]
[[2, 3], [4, 5], [6, 7]]
[[3, 4], [5, 6], [7, 8]]
[[4, 5], [6, 7]]
[[5, 6], [7, 8]]
[[6, 7]]
[[7, 8]]

 ---------------
```

```
In [40]: #Sorting k=N/2. Output reduced for the sake of clarity.

         for n in range(6,10,2):
             print('number of qubits {}'.format(n))
             per  = np.array(createPermutations(n)) +1
             per = per[(n//2-2)]
             t = createAdjacentTranspositions(per)
             for per in t:
                 print(per)
             print('\n ---------------\n')
```

```
number of qubits 6
[[5, 6]]
[[2, 3]]
[[1, 2]]

 ---------------

number of qubits 8
[[7, 8]]
[[2, 3], [4, 5]]
[[1, 2], [3, 4]]
[[2, 3]]
[[1, 2]]

 ---------------
```

I used the funcitons of the next section to check if the functions defined in the appedix A for the proof of P_Lˆk are correct.

```
In [30]: def Right(k,t,L):
             if (t-k)> 0 and (t-k)<= L/2 :
                 return (2*k-t-1)%L + 1
             else:
                 return (2*k-t)%L + 1
         def Left(k,t,L):
             if (t-k)> 0 and (t-k)<= L/2 :
                 return  (2*k-t)%L + 1
             else:
                 return (2*k-t-1)%L + 1


         def q(k,t,L):
             if (t-k)> L/2 :
                 return 2*(k-t+L) + 1
             elif (t-k)>0 and (t-k)<= L/2:
```

```
            return  2*(t-k)
        elif (t-k)<= 0:
            return 1 + 2*(k-t)
    def P(k,j,L):
        if j%2 ==0:
            return (k-1 + j//2) %L +1
        else:
            return (k-1 - (j-1)//2) %L +1
```

In [33]:
```
#R Correctly defined, otherwise it would output something
for NumberQ in range(6,100,2):
    permutations = createPermutations(NumberQ)
    for j in range(len(permutations)):
        per = permutations[j]
        k = j+1
        for i in range(len(per)-1):
            t = per[i]
            #print(Right(k,t,NumerQ) == per[i+1])
            if not Right(k,t,NumberQ) == per[i+1]:
                print('k:{} t:{} '.format(k,t))
                print(per)
```

In [35]:
```
#L Correctly defined, otherwise it would output something
for NumberQ in range(6,100,2):
    permutations = createPermutations(NumberQ)
    for j in range(len(permutations)):
        per = permutations[j]
        k = j+1
        for i in range(1,len(per)):
            t = per[i]
            #print(Right(k,t,NumerQ) == per[i+1])
            if not Left(k,t,NumberQ) == per[i-1]:
                print('k:{} t:{} '.format(k,t))
                print(per)
```

In [36]:
```
#q Correctly defined, otherwise it would output something
for NumerQ in range(6,100,2):
    permutations = createPermutations(NumerQ)
    for j in range(len(permutations)):
        per = permutations[j]
        k = j+1
        for i in range(len(per)-1):
            t = per[i]
            #print(Right(k,t,NumerQ) == per[i+1])
            if not q(k,t,NumerQ) == i+1:
                print('k:{} t:{} '.format(k,t))
                print(per)
                print('i:{}  q:{}'.format(i+1,q(k,t,NumerQ)))
```

In [37]:
```
#P Correctly defined, otherwise it would output something
for NumerQ in range(6,100,2):
    permutations = createPermutations(NumerQ)
    for j in range(len(permutations)):
        per = permutations[j]
        k = j+1
        for i in range(len(per)-1):
            t = per[i]
            #print(Right(k,t,NumerQ) == per[i+1])
            if not P(k,i+1,NumerQ) == t:
                print('k:{} t:{} '.format(k,t))
                print(per)
                print('i:{}  P:{}'.format(i+1,P(k,i+1,NumerQ)))
```