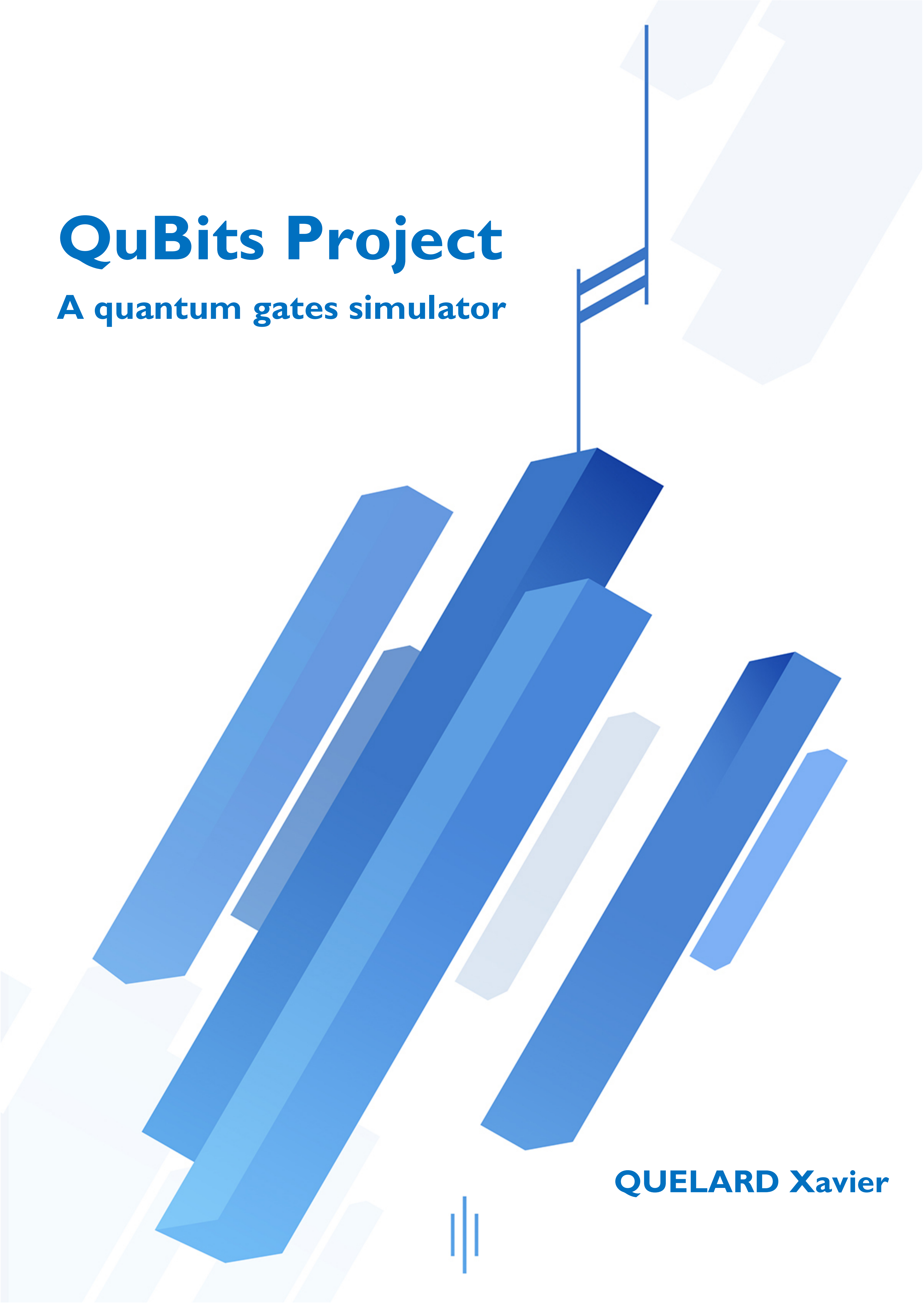
QuBits Project

A quantum gates simulator



QUELARD Xavier

Table of Contents

1. Introduction

Quantum computers are on everyone's lips nowadays. These futuristic computers, built following the strange rules of quantum mechanics, are suppossedly going to be much faster that our current computer. But right now, very few and only very specialised quantum computers even exists.

But how is a quantum computer supposed to work? What are the rules of the microscopic world? And why quantum computing might be much faster than "classical" computing? That's what will be discussed in the first chapter of this report. A short mathematical background will also be delivered for the most interested lectors. Then, the project QuBits, which consisted of creating a quantum computer simulator, will be explained and detailled, and its results shared and compared to other simulators.

2. Project's context

2.1 History of quantum physics

In this very first section of the report, we will focus on the story of quantum physics*[1] [2]*.

One could say the first ideas of quantum mechanics were born in 1897, when the physicist Max Planck tries to theorize the black body radiation (see figure 2.1), a physicist problem that was unsolved at the time. He came with a formula that was describing the reality very accurately, but without any clue on why this was the right answer. Indeed, the physics world was divided between the atomists and the non-atomists, and as one of the latter, Plank was trying to reconstruct his results without atoms.

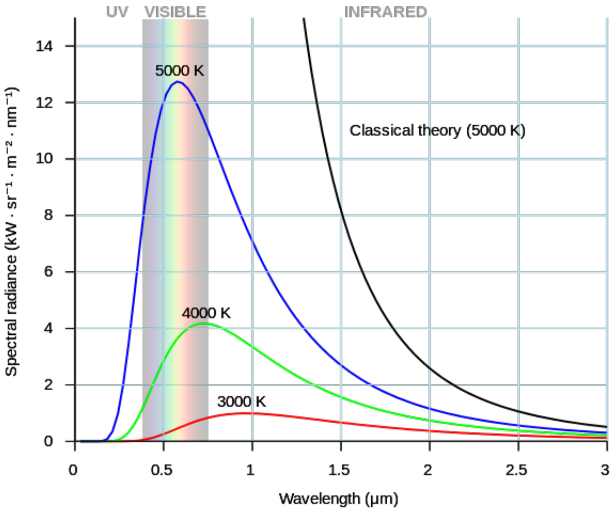


Figure 2.1: *The curve Planck was trying to describe and later explain. It basically shows how the wavelength of an object differs with its temperature.*

After 3 years of hard work, Plank was not able to explain the black body radiations without atoms, so he completely changed his mind on the subject. After another year of reseach, he was able to explain the curve by taking atoms into account. But to successfully derive his mathematical law, he also needed to include more restrictions : atoms can only be at certain levels of energy. He discovered that in the atomic world, energy was discrete, quantified (see figure 2.2).

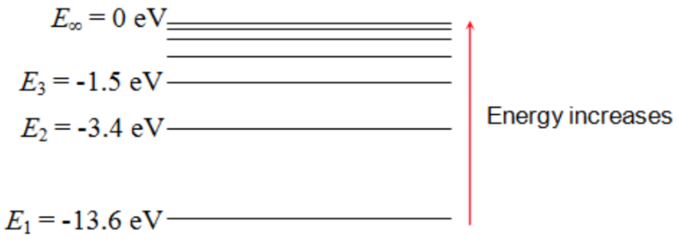


Figure 2.2: *The curve Planck was trying to describe and later explain. It basically shows how the wavelength of an object differs with its temperature.*

But for five years, not a single paper was written to talk about Plank's quantum ideas. To the more enthusiastics, these ideas were an elegant way to get a mathematical law to work but probably not very realistic, while for the others, it was simply a failure without any sense. That's when a young Albert Einstein wrote his first scientific papers, proving that Plank's theory was not working just for black body radiation, but was a general principal of physics instead, that can and should be applied for a lot of other objects. By applying it to the light, Einstein theorizes the existence of photons, which are small quantums of light. The same way atoms's energy can only have certian values, light's energy is discrete and can be reduced to those particles.

Albert Einstein's paper could have been laughable form the perspective of the physicists community. But with his photons theory, he was able to explain an effet that was still at mystery at the time : the photoelectric effect (see figure 2.3). Where classical physics failed to give a correct explanation of the phenomenon, the quantified light of Einstein was working very well, and ultimately this paper (among other brilliants one he published in the same time period) won his the Nobel prize later on.

The next step for the quantum theory was discovered by Neils Bohr in 1912. He was the first scientific able to combine Plank's atom theory and Einstein's light theory to create an atom model. In this model, and atom can jump on or off in energy levels by respectively absorbing or emitting a photon. Depending on the quantity of energy emitted, a different wavelength of light was obtained (see figure 2.4). This gave a direct explaination of elements spectroscopy (each element reflects a different discrete spectrum when observed with light).

The final step of quantum theory was made by Werner Heisenberg and Erwin Schrödinger in respectively 1925 and 1926. This time, they were not explaining only isolated phenomenon, but based on the previous works of Plank, Einstein and Bohr, they created an equivalent to the prevalent Newton's theory. At the

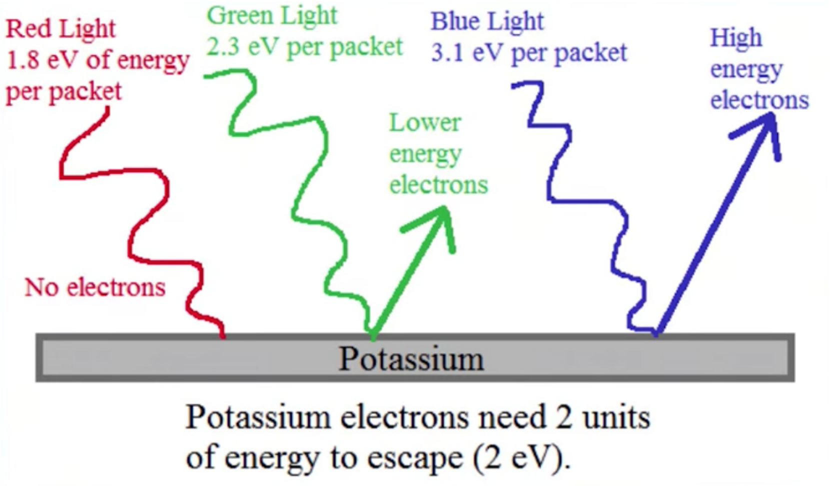


Figure 2.3: *The photoelectric effect describes how a metal reacts to the emmision of a light, and how it changes electrically speaking depending of the wavelength.*

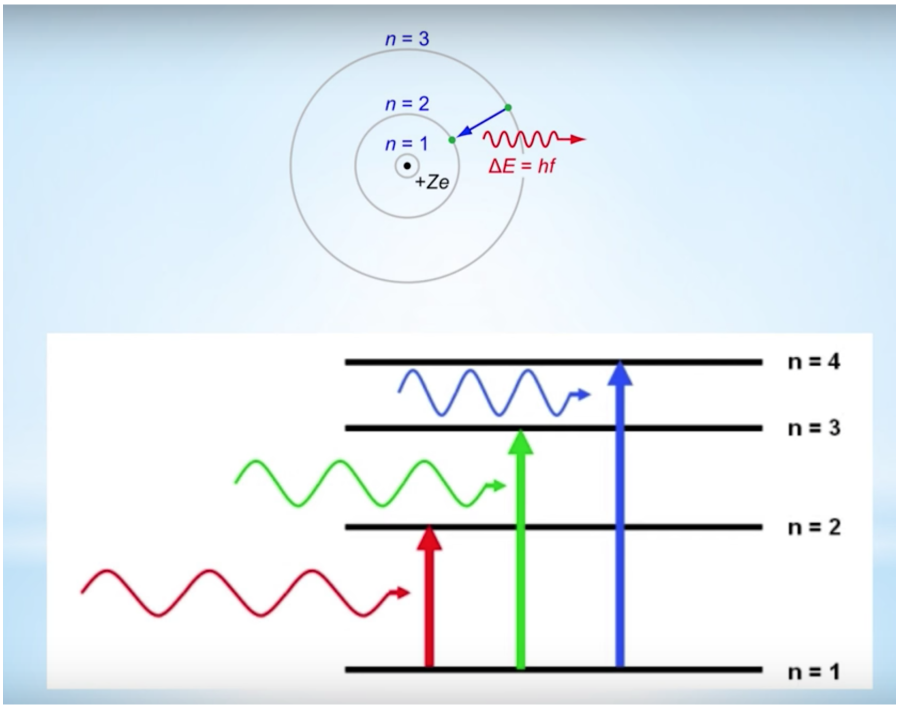


Figure 2.4: *The formula relates discrete amount of energy and light's frequency. As frequency and wavelength are inversionally proportionnal, this relation gives a link between energy absorbed by an atom and the spectrum of it.*

time, these two theories were competing and both better at predicting events than Newton's laws of motion. What was fascinating is that although these two theories were using completely different mathematical objects (matrices and waves), they had incredibly similar results. Years after that, it was in fact mathematically proven that those theories were strictly equivalent, and by no one else than Schrödinger himself.

2.2 Quantum physics basic principles

As we have seen previously, quantum theory slowly became accepted in the physicists world, ans as time went on, it proved to be the most successfull theory ever created, explaining and predicting with incredible precision the rules of the atomic realm. Nowadays, it is used in most piece of technologies, and is for example the reason why GPS are so precise. But now that we have learned a bit more about its history, let's dive into its fundamental principles and its strange laws. We will learn about several very important principles of quantum mechanics that we need in order to understand quantum computing. We will discover how counterintuitive most of these are, highlighting that the world does not behave the same way at all in the microscopic scale.

2.2.1 Superposition principle

The first idea is called the superposition principle. While a classical object is too big to follow the quantum mechanics's rules, a quantum object -for example an atom or an electron- follows this superposition principle. A quantum object can be in a superposition of states, which means that instead of having predetermined and fixed variables (position, speed, etc...), it can be in a superposition of very different states. Therefore, an atom might "be" at two places and move at two different speeds at the same time. In quantum theory, a superposition of 4 states is noted like this (this bracket notation is called "ket") :

$$|\boldsymbol{atom}\rangle = |\boldsymbol{state1}\rangle + |\boldsymbol{state2}\rangle + |\boldsymbol{state3}\rangle + |\boldsymbol{state4}\rangle \tag{2.1}$$

A famous illustration of this phenomenon is Schrödinger's cat : an animal that would be both dead and alive according to quantum mechanics.

2.2.2 Measurment indeterminism

The second key idea in quantum mechanics is the indeterminism of the measurment. Measuring a quantum object's properties is very different from a macroscopic measurment, because this time, the object may be in several different states. And quantum physics shows that when measuring a quantum object, only one of its many states will be observed, and the outcome is purely random. There is no

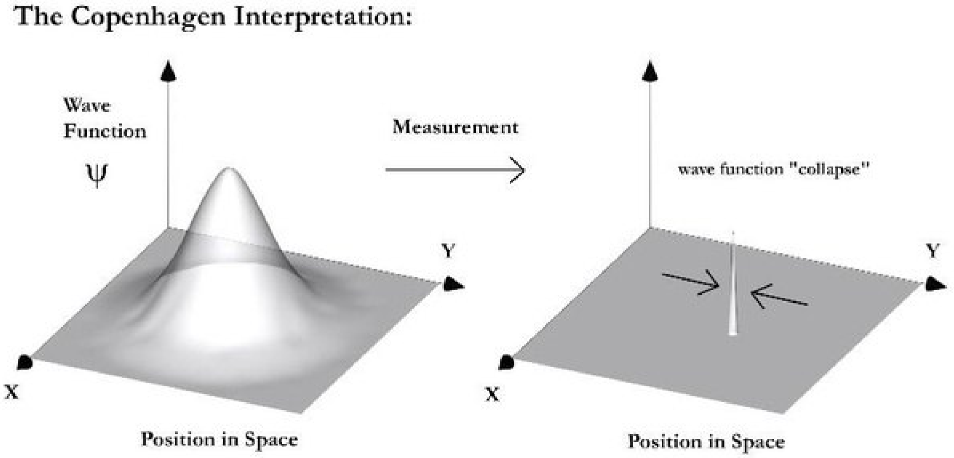


Figure 2.5: *The Copenhangen interpretation of quantum idea is the most accepted one in the physicists community. Before measurment, a particle can be anywhere on the graph, but as soon as the measure is made, the uncertainty drops and the atom has a single position in space.*

deterministic law that can tell for sure which state will be "chosen" upon measurment : it is purely and simply random.

2.2.3 Quantum states reduction

However, once one state has been measured, any further measurment will give the same state again 100 percent of the time. This means that not only the measurment is randomly "made" between all the superposed states, but once it is done, the quantum object is in one single state : the measure affects the object it tries to analyse. This is called the reduction of quantum states, as measuring a quantum object reduces him from a superposition of states to a single one (see figure 2.7).

2.2.4 Particle-wave duality

The fourth key idea in quantum mechanics is the particle-wave duality. It is again a very strange phenomenon that describes the fact that in the quantum world, some objects like electrons behaves like a wave or like a particle, depending on the context. The experiment that showed it was probably the most famous of the 20th centuary : the double vent Feynman experiment. He used a electron cannon to fire electrons, one by one, at a "wall" with two vents and was registering the impacts of the electrons on the backstop. If the two vents are open, interferences figures were appearing on the screen, suggesting that electrons were behaving live waves. On the other hand, opening a single vent was showing electrons shot like gun's bullets, suggesting a particle's behaviour (see figure 2.6).

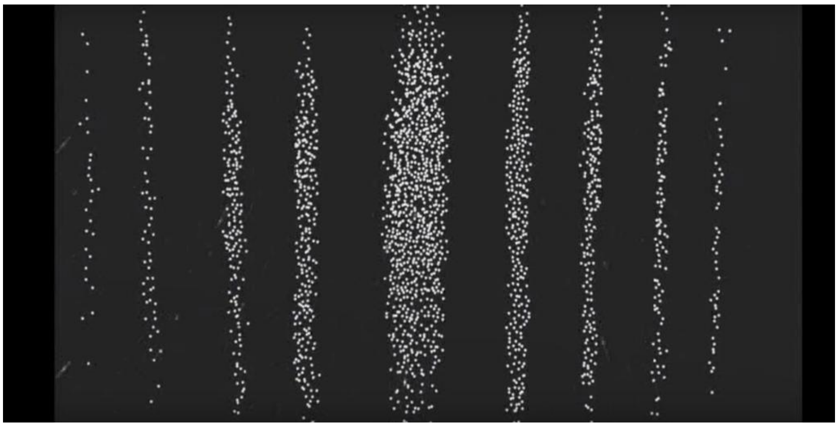


Figure 2.6: *Results if the two vents were open. We can clearly see some interferences figures. On the other, with one vent close, a single column of dots would be observed.*

2.2.5 Heisenberg uncertainty principle

The last important law of quantum physics that needs to be explained here is the Heinsenberg incertainty principle. It is quite simple but very confusing, as it states that it is impossible to have a quantum state where both position and speed of a particle are well defined at the same time. In fact, the more precise a speed measurment is, the less accuracy there will be on its momentum (equation (2.2) shows this law for a particle on a single axis position x).

$$\Delta p \Delta x \geq \frac{1}{2}\hbar \tag{2.2}$$

2.3 Introduction to quantum computing

2.3.1 Classical computer

In order to understand the project's stakes, it is important to know how a quantum computer basically works and why it differs from what we call a « classical » computer, which is the type of computer everyone uses nowadays. A well known fact is that computers uses electrical signals to exchange informations. For stability reasons, specialists chose to only code two inputs : 0 and 1, which stands for wether or not there is a signal. It is indeed a clever choice, as it is quite simple to check for only two possibilities that are very distincts. Using those two values, every information is stored in bits, and with clever maths, it is possible to encode anything you can see on a computer : numbers, text, colors, images, etc.. Calculations are not exceptions, and are also done using bits on classical computers, so every operation has been reprogrammed to work with bits instead of numbers. With a few amount of basic « logical gates » (= a circuit that takes one or multiple bits and apply a basic operation), addition, then multiplication, and other

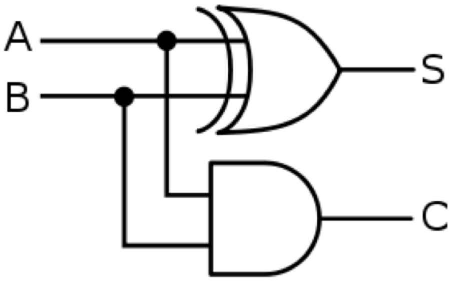


Figure 2.7: *A circuit that takes A and B as 1-bit inputs, and outputs the sum S and the carry C. It uses only two logical gates, the XOR and AND gates.*

operations were available to computers.

2.3.2 Quantum computer and project's goals

We have seen that in order to work, a computer basically needs two crucial features :

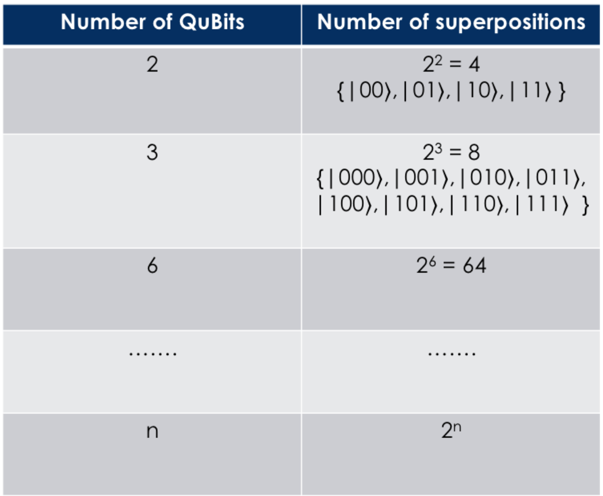
- A way to store and exchange informations, with bits
- A way to apply operations on information, with logical gates

A quantum computer seems just like a classical computer, doing those same two things, but using quantum bits (or qubits) and quantum gates instead. Basically, a quantum bit is just a bit following the quantum mechanics rules. But as it is ruled by these strange laws, it can perform way better than current computers on few keys problems. As an example, a working quantum computer would be able to search for an entry in a database incredibly faster than a normal one, thanks to its quantum algorithms. Even better : it is mathematically proven that no classical computer can perform better than what we already built, which means the only way to progress in by using quantum devices.

But for now, working on real qubits is not really possible for anyone but few scientists. Therefore, simulating those qubits is a very real possibility. It can allow scientists to test and discover new algorithms using these QuBits, in order to make progress in computing science. And this is exactly the subject of the QuBits project : to simulate qubits and quantum gates. The goal was to create a program that simulates these two quantum object, and therefore simulate a basic quantum computer.

2.3.3 Why is there such an interest on quantum computing?

But a question remains : why is the interest so high in quantum computing? What are its promises? There are two main reasons for that : the first one is quantum



Number of QuBits	Number of superpositions
2	$2^2 = 4$ $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$
3	$2^3 = 8$ $\{|000\rangle, |001\rangle, |010\rangle, |011\rangle,$ $|100\rangle, |101\rangle, |110\rangle, |111\rangle\}$
6	$2^6 = 64$
....	....
n	2^n

Figure 2.8: *The number of superpositions explodes when n, the number if qubits, is growing. This exponential growth shows the amazing possibilites brougth by quantum computing.*

computing power, and the second one is quantum algorithms.

As qubits are quantum objects, they can be in a superposition of states. So while a classical bit would either be in the 0 or 1 state, a qubit can be in a superposition of the states $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$, (as seen on part (2.2.1)). This means that doing an operation on a qubit in both state is like doing two operations on two classical bits. But quantum computers are not just two times "faster" than classical ones : if a register of 2 qubits is used, there is 4 states : $|\mathbf{00}\rangle$, $|\mathbf{01}\rangle$, $|\mathbf{10}\rangle$ and $|\mathbf{11}\rangle$. In general, if you use a register of n qubits, there is 2^n states, so if a qubit is in each of these states, the computing power is much greater than a classical computer (see figure (2.8)).

However, the quantum state reduction principle (section (2.2.3)) states that once a measurment is made, a quantum object only stays in one quantum state. This means that although the calculations can be made on each state, only one measure can be made, so only one state will remain. A lot of information can be used during the caculations, but in the end, only one can be kept. It looks a bit like using complex numbers to factorize polynomials : you can use these intermediates to simplify calculations, but in the end, you come back to the real numbers realm, and complex numbers were just an intermediate.

This is were quantum algorithm comes into place. We will discuss further this point later on, but in order to use quantum computers computing power, algorithm that can work with this measurment limitation needs to be used. Of course, not all problems can be solved this way, and probably most of them will not. But for those where a quantum solution is found, a great acceleration in performances is the reward.

For example, right now, a lot of cryptography relies on the RSA algorithm[1]. This algorithm make the operation of decrypting a message without the key very difficult, because it relies on the factorisation of two very big prime numbers, an operation that takes an insane amount of time for classical computers. Therefore, a hacker would need million years to hack crypted messages with RSA. However, a quantum algorithm called Shor's algorithm could make the same mathematical operation within seconds. But then, would quantum chips be the end of numerical security? Not really, because it could also bring new ways of crypting messages that would this time be impossible to crack, according to the laws of physics.

Searching a match in a database is another algorithm that is quite slow on a classical computer. But with Grover's quantum algorithm, a quantum computer could drastically outperform classical computer on this task. While a classical algorithm needs to check every single element of the database (and therefore has a temporal complexity of $O(N)$ operations, where N is the size of the database), Grover's algorithm has a complexity of $O(\sqrt{N})$! On a large database, the computation gains are incredibly high.

[1] Rivest Shamir Adleman ares the inventor of this algorithm

3. Mathematical reminders

Most mathematical notions and definitions in this chapter comes from the courses I had during my preparatory classes*[5]*.

3.1 Qubits

3.1.1 Qubit definition

Let $\mathbb{H}_n$ be a Hilbert complex space of n dimensions. Any vector of the $\mathbb{H}_2$ space is defined by : $|\boldsymbol{\Phi}\rangle = \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle$, where λ and μ are complex coefficients, and $\{|\mathbf{0}\rangle, |\mathbf{1}\rangle\}$ is a orthonormal base (both vectors are orthogonal and normalized). The scalar or dot product of two vectors is then defined by (3.1), and its norm is defined by (3.2).

$$\langle \Phi / \Psi \rangle = \langle\ \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle\ /\ \nu|\mathbf{0}\rangle + \sigma|\mathbf{1}\rangle\ \rangle = \lambda^* \nu + \mu^* \sigma = \langle \Phi / \Psi \rangle^* \tag{3.1}$$

$$||\Phi||^2 = \langle \Phi / \Phi \rangle = \langle\ \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle\ /\ \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle\ \rangle = |\lambda|^2 + |\mu|^2 \tag{3.2}$$

We also add a constraint to each qubit : the normalisation condition (3.4). This will prove to be useful later on.

$$|\boldsymbol{Q}\rangle = \alpha \cdot |\mathbf{0}\rangle + \beta \cdot |\mathbf{1}\rangle \tag{3.3}$$

$$(\alpha, \beta) \in \mathbb{C}^2\ /\ |\alpha|^2 + |\beta|^2 = 1 \tag{3.4}$$

3.1.2 Measuring a qubit

Measuring a qubit*[6]* can be described as a function $\mathbb{M} : \mathbb{H}_2 \to \{|\mathbf{0}\rangle, |\mathbf{1}\rangle\}$: take a random qubit, and measuring it will either give $|\mathbf{0}\rangle$ state or $|\mathbf{1}\rangle$ state. But as seen previously, measurment is not deterministic in quantum mechanics : in fact, measuring the state $|\boldsymbol{Q}\rangle = |\mathbf{0}\rangle$ in the $\{|\mathbf{0}\rangle, |\mathbf{1}\rangle\}$ base will give $|\mathbf{0}\rangle$ every single time. Similarly, measuring $|\boldsymbol{Q}\rangle = |\mathbf{1}\rangle$ will give $|\mathbf{1}\rangle$. However, in the general case, where the qubit in a combination of both states, it becomes trickier. Let $|\boldsymbol{Q}\rangle$ be a vector of $\mathbb{H}_2$. The probability of measuring it in the $|\boldsymbol{\Omega}\rangle$ state is defined by (3.5).

$$P(\ M(|\boldsymbol{Q}\rangle) = |\boldsymbol{\Omega}\rangle\ ) = |\langle\ |\boldsymbol{Q}\rangle\ /\ |\boldsymbol{\Omega}\rangle\ \rangle|^2 = P(\ M(Q) = \Omega\ ) \tag{3.5}$$

So, for a given qubit $|\boldsymbol{Q}\rangle = \alpha \cdot |\mathbf{0}\rangle + \beta \cdot |\mathbf{1}\rangle$, it has λ^2 chances of being measured in the $|\mathbf{0}\rangle$ state, and μ^2 chances of being in the $|\mathbf{1}\rangle$ state. This is where the normalisation (3.4) suddenly makes quite a lot of sense. This works for sure as long as a orthonormal base is chosen.

3.1.3 Bloch's sphere

The goal of this subsection is to give the lector a visual representation of a quantum state. In order to do so, the Bloch's sphere shall be introduced. Let's examine the following equivalence relation $R : \mathbb{H}^2 \to \mathbb{H}^2$

$$|\boldsymbol{\Psi}\rangle R|\boldsymbol{\Psi'}\rangle \Leftrightarrow \exists z \in \mathbb{C}/ \begin{cases} |\boldsymbol{\Psi}\rangle R|\boldsymbol{\Psi'}\rangle \Leftrightarrow |\boldsymbol{\Psi}\rangle = z|\boldsymbol{\Psi'}\rangle \\ \hfill |z| = 1 \end{cases} \tag{3.6}$$

R trivially verifies reflexivity, symetry et and transitivity. Belonging to the same equivalence class means the qubits are all equals except for a phase factor $z = e^{i\theta}$. We can then obserb that multiplying a state by a phase factor doesnt change its probability of being measured in a given $|\boldsymbol{\alpha}\rangle$ state :

$$\begin{aligned} P(\ M(z|\boldsymbol{\Psi}\rangle) = |\boldsymbol{\alpha}\rangle\ ) &= \langle z|\boldsymbol{\Psi}\rangle / |\boldsymbol{\alpha}\rangle\rangle & (3.7)\\ &= |e^{i\theta}|\langle|\boldsymbol{\Psi}\rangle/|\boldsymbol{\alpha}\rangle\rangle & (3.8)\\ &= P(\ M(|\boldsymbol{\Psi}\rangle) = |\boldsymbol{\alpha}\rangle\ ) & (3.9)\end{aligned}$$

This means that two qubits can be considered equals if belonging to the same equivalence class, which leads to the following rewriting :

$$\begin{aligned} \forall|\boldsymbol{\Psi}\rangle \in \mathbb{H}^2, \exists\theta \in [0,\pi]\ /\ |\boldsymbol{\Psi}\rangle &= \cos(\theta/2) \cdot |\mathbf{0}\rangle + \sin(\theta/2)e^{i(\varphi)} \cdot |\mathbf{1}\rangle & (3.10)\\ &\equiv \cos(\theta/2)e^{-i(\varphi/2)} \cdot |\mathbf{0}\rangle + \sin(\theta/2)e^{i(\varphi/2)} \cdot |\mathbf{1}\rangle & (3.11)\end{aligned}$$

With this new equation, it is possible to represent even single qubit (with the phase factor ignored) on a sphere, where θ and φ are the colatitude and the longitude (see figure (3.1)). There is a isomorphism between the qubit space and the Bloch's sphere. However, one may say that this sphere is only a visual representation : linearity is NOT preserved, as we know $|\mathbf{0}\rangle \neq -|\mathbf{1}\rangle$. This Bloch's sphere is just here to help visualizing the state of a qubit.

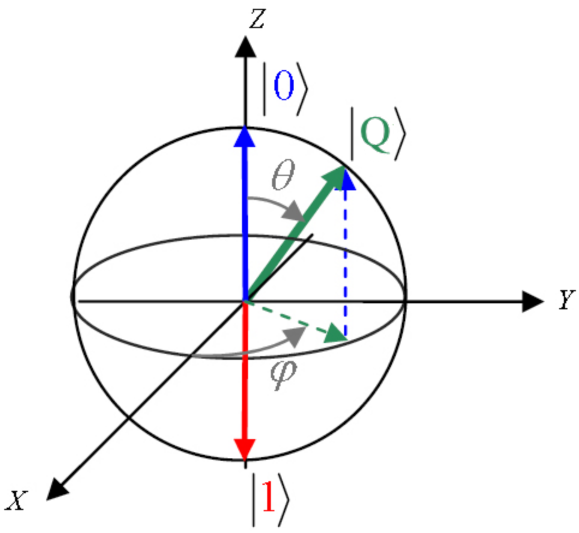


Figure 3.1: *Visualizing a quantum state with Bloch's sphere.*

3.1.4 Multiple qubits register

Here is defined the tensorial product ⊗ of two qubits (3.12) .

$$(\cdot)\otimes(\cdot) : \begin{cases} \mathbb{H}^2 \times \mathbb{H}^2 \to \mathbb{H}^4 \\ (|\boldsymbol{\varphi}\rangle, |\boldsymbol{\Psi}\rangle) \mapsto |\boldsymbol{\varphi}\rangle \otimes |\boldsymbol{\Psi}\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \lambda \\ \mu \end{pmatrix} = \begin{pmatrix} \alpha\lambda \\ \alpha\mu \\ \beta\lambda \\ \beta\mu \end{pmatrix} \end{cases} \tag{3.12}$$

With this product*[8]* of two $\mathbb{H}^2$, we can define the space of two qubits (so with up to 4 quantum states) :

$$\{|\boldsymbol{0_A}\otimes\boldsymbol{0_B}\rangle, |\boldsymbol{0_A}\otimes\boldsymbol{1_B}\rangle, |\boldsymbol{1_A}\otimes\boldsymbol{0_B}\rangle, |\boldsymbol{1_A}\otimes\boldsymbol{1_B}\rangle\} = \{|\boldsymbol{00}\rangle, |\boldsymbol{01}\rangle, |\boldsymbol{10}\rangle, |\boldsymbol{11}\rangle\} \tag{3.13}$$

$$= \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right\} \tag{3.14}$$

$$= \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right\} \tag{3.15}$$

3.2 Quantum gates

3.2.1 One qubit gates

A quantum gate can be described as a function L :

$$L(...) : \left\{ \begin{array}{rl} \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times ... & \rightarrow \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times ... \\ (|\boldsymbol{\varphi}\rangle, |\boldsymbol{\Psi}\rangle, |\boldsymbol{\Omega}\rangle, ...) & \mapsto (L(\varphi), L(\Psi), L(\Omega), ...) \end{array} \right. \tag{3.16}$$

It's simply a linear function which takes several qubits as entry, and outputs the same qubits, modified. Since a quantum gates does not make any measurment, the transformation needs to maintain the normalisation condition. Therefore, we can simply modelize a quantum gate with a matrix M (norm = 1). The state after the gate can then be measured by $L(\Psi) = M \cdot \Psi$.
By definition, quantum calculus is reversible, which means every quantum gate matrix must be reversible.

We are now able to define two one-qubit gates : phase L_Φ and rotation L_α. Let's call $Mat()$ the function that maps the matrix corresponding to a quantum gate (we position ourself in the $\{|\mathbf{0}\rangle, |\mathbf{1}\rangle\}$ orthonormal base). Then :

$$Mat(L_\Phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\Phi} \end{pmatrix} \tag{3.17}$$

$$Mat(L_\alpha) = \begin{pmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{3.18}$$

The first gate adds a phase factor to the $|\mathbf{1}\rangle$ while the second once makes a rotation of α. Here are two other gates :

$$Mat(NOT) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{3.19}$$

$$Mat(HADAMARD) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.20}$$

The NOT gate is just like the not gate of classical computer. However, the hadamard gate in unique to quantum computing.The goal of this gate is to balance the $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$ weights of a given qubit.

3.2.2 Two qubits gates

The $CNOT = Controlled\ NOT$ gate is a NOT gate, but acting on the targeted qubit only. The second qubit is named the controlled qubit, and is here to apply a if-statement : if the controlled qubit is in the $|\mathbf{1}\rangle$ state, then apply NOT on the targeted qubit. Otherwise, this gate does nothing. The $CPHASE$ gate will work the same way, applying the $PHASE$ gate on the targeted qubit only if the condition on the controlled qubit is met. Finally, the swap gate simply "exchange" the values of the two qubits it takes.

$$Mat(CNOT) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.21}$$

$$Mat(CPHASE) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\Phi} \end{pmatrix} \tag{3.22}$$

$$Mat(SWAP) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.23}$$

4. Project's development

Now that we have all the historical and mathematical background needed to understand quantum computers, we will discuss how the project was done, step by step.

4.1 Establishing a golden model

4.1.1 Class diagram

The very first step of the project was to understand how quantum computers should work in theory, in order to develop a simulator. For this reason I wrote a detailled report (in french) about quantum computer's mathematics than can be found in the appendix. The previous chapter (3) was a shorter version of this document, as its goal was only to give a minimal and condensed mathematical background to the lector.

The first development step was then to create a golden model of the qubits simulator. A golden model is a first implementation of a program. Its goal is to be implemented quickly in order to allow for the first tests and further refinments. As such, it is also preferrable to use a high-level programming language to benefit from its more readable syntax and shorter development time.

Before choosing any language, a class diagram of the simulator was established, in order to define once and for all the structure of the simulator. The diagram shown on the figure (4.1) is the final result.

The first class, called QuObject, is an abstract class which only goal is to be inherited by the two others in order for these to share the same structures and have the same basic operations. It contains a string to binary convertion method (the simulator allow users to define qubits with strings rather than vectors, so this operation is needed everywhere in the simulator), but also a norm method (useful for calculating probabilities) and an Int to Complex conversion method, as the user can enter real or complex numbers as parameters of a qubit, but the simulator needs only Complex numbers to work.

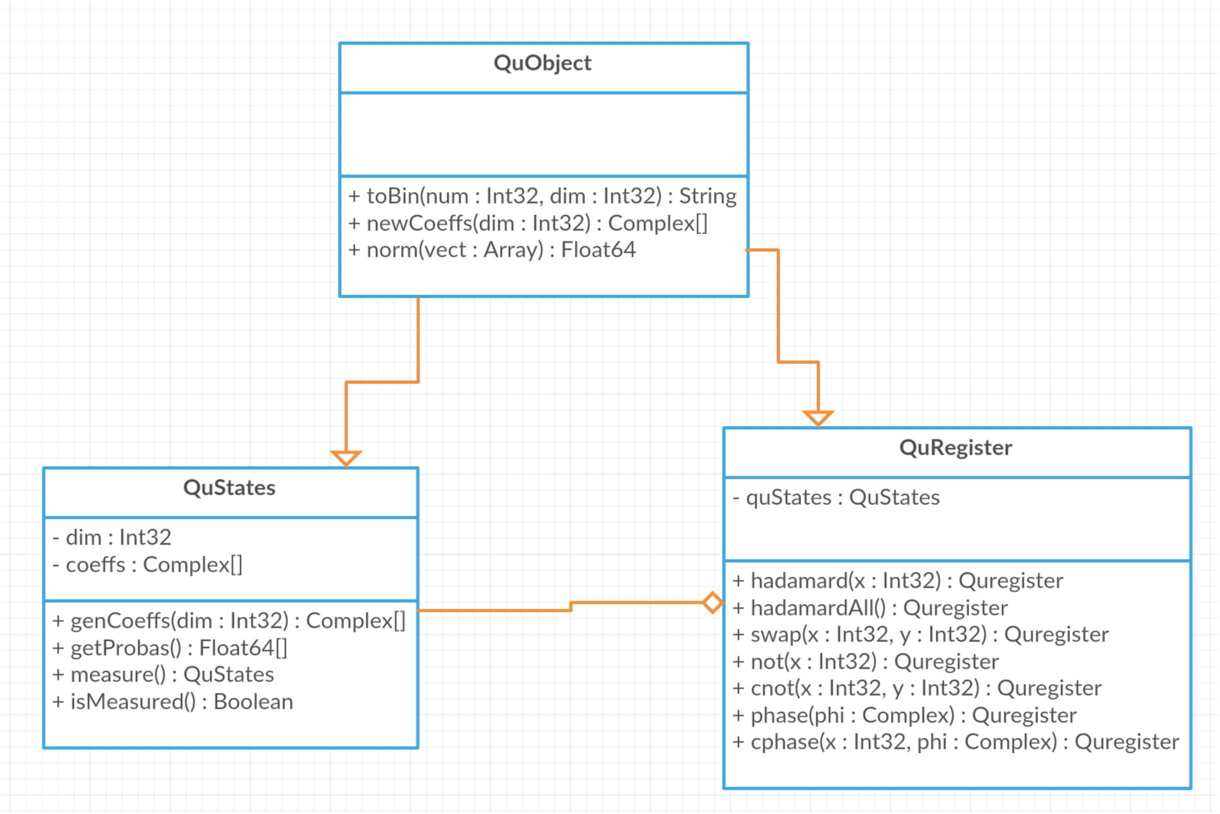
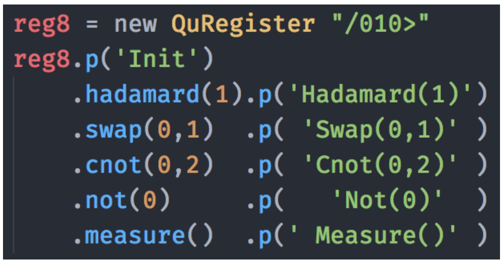

QuObject
+ toBin(num : Int32, dim : Int32) : String
+ newCoeffs(dim : Int32) : Complex[]
+ norm(vect : Array) : Float64
QuStates
- dim : Int32
- coeffs : Complex[]
+ genCoeffs(dim : Int32) : Complex[]
+ getProbas() : Float64[]
+ measure() : QuStates
+ isMeasured() : Boolean
QuRegister
- quStates : QuStates
+ hadamard(x : Int32) : Quregister
+ hadamardAll() : Quregister
+ swap(x : Int32, y : Int32) : Quregister
+ not(x : Int32) : Quregister
+ cnot(x : Int32, y : Int32) : Quregister
+ phase(phi : Complex) : Quregister
+ cphase(x : Int32, phi : Complex) : Quregister

Figure 4.1: *The projet's class diagram, divided in three main classes. Most important methods are shown, but some like getter and setters and internal operations are hidden for clarity purposes.*

The second class, called QuStates, is a container. It saved all the informations needed for a register of several qubits (or only one qubit) : the dimension and the complex coefficients of each superposed state. And then it provides a measurment method, which uses the getProbas method to simulate a random measurment.

The last class QuRegister, is the object the user interacts with. Therefore, it has several constructors (either the user instanciate with a string, or with an array of complex numbers) which convert the data and register it in a single QuStates container. Then, this class offers a lot of method for the user to act on the QuState : the quantum gates. Since the QuRegister contains one QuStates on which it acts, there is an aggregation between these two classes.

It is interesting to point out that each quantum gates return a QuRegister. This is a programming idea that allows the user to chain the quantum gates method calls, like seen in the figure (4.2).



```
reg8 = new QuRegister "/010>"
reg8.p('Init')
    .hadamard(1).p('Hadamard(1)')
    .swap(0,1)  .p( 'Swap(0,1)' )
    .cnot(0,2)  .p( 'Cnot(0,2)' )
    .not(0)     .p(   'Not(0)'  )
    .measure()  .p(' Measure()' )
```

Figure 4.2: *Method calls are chained because each method return the this/self object. The p method between each quantum gate called is here to print a text, in order for the user to get a understandable chain of results.*

4.1.2 First implementation in Coffescript

Once the object-oriented model was defined, a language to program it had to be chosen. Since the goal was to develop a golden model, the choice was made on Coffeescript. Indeed, Coffeescript is :

- A small language that compiles into Javascript, with the goal of giving it a simpler syntax.
- A way of writing object-oriented Javascript with an elegant syntax (very close to a language like Ruby) and a very easy to read language.
- An "extension" of Javascript. Javascript is a multi-platform web language that has a very easy-to-use object-oriented philosophy, simplifying the development of the golden model.

As a simple, very readable and easy language to use, Coffeescript was a good high level language choice to create the golden model. The source code of the golden model can be found in the appendix.

4.2 Development of a more advanced simulator

4.2.1 Programming language's choice

For the final implementation of the project, another language needed to be used, in order to gain performances. Mr Lelann suggested an emerging new language called Crystal. Crystal is :

- A compiled language, therefore way more performance oriented.
- A Ruby-inspired syntax language, which goes quite far, as most of simple Ruby programs can directly be converted into Crystal without changing anything.
- An emerging language than is not very well know nor very well documented.

The first two points are what lead to the descision of using the Crystal language : with a Ruby-like syntax, converting and enhancing the code from Coffescript would be fast and both codes would look quite similar structurally speaking, while hopefully allowing for some performance gains. However, the last point was the reason why this transition was not immediate. To even use the fundamental libraries, it was often necessary to check Crystal's own source code as the API itself is very lackluster in terms of comments and explanations.

4.2.2 Implementation

On the other hand, it was a difficult but pleasant experience to discover a whole new very interesting language by experimenting, looking into the source code and scarce documentation. The final implemention in Crystal can be found in the appendix.

5. Quantum algorithms and experimental results

In this part of the report, we will introduce several well known quantum algorithms and how they are overperforming classical computers. The Deutsch-Jozsa algorithm will be more detailled as I was able to implement it on the QuBits quantum simulator.

5.1 Deutsch-Jozsa algorithm

5.1.1 Description

Let $f : \{0,1\}^n \mapsto \{0,1\}$ be a function either balanced (equally gives 0 or 1 as an output) or constant (always return 0 (resp. 1)). The Deutsch-Jozsa algorithm is a quantum algorithm that allows a quantum computer to determine whether such a f is balanced or constant with a single operation (while a classical algorithm would need at worst to check half of the function's values to decide). The figure (5.1) shows the quantum circuit of this algorithm.

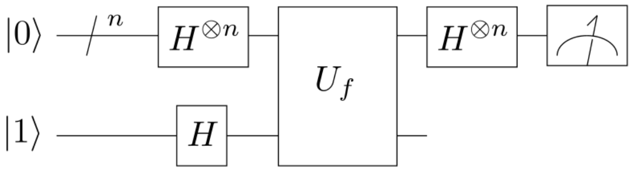
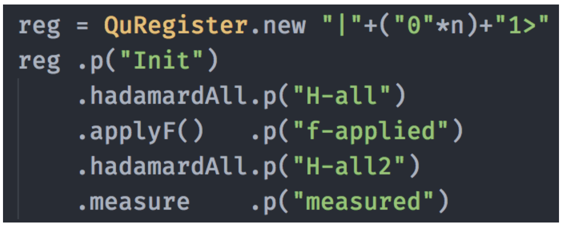


Figure 5.1: *Deutsch-Jozsa algorithm's quantum circuit.*

First, you form a n+1 qubits register $|\mathbf{000...0001}\rangle$ where there is n zeros. You then apply the Hadamard gates to all those qubits. The U_f oracle can then be applied : it only changes the very last qubits, applying the operation $f(x) \oplus q$, where q is the current value of the qubit and $\oplus$ is an addition modulo 2. Finally, the Hadamard gate is applied once again, and you measure the first n qubits. If there are only zeros, the f function is constant. In any other case, it is balanced.

5.1.2 Implementation

The figure (5.2) shows the implementation in Crystal of this algorithm. We simply create the $(n * 1)$ dimensions qubits (here, $n = 2$), then we apply the oracle between two Hadamard-all gates, and we measure the result.



```
reg = QuRegister.new "|"+("0"*n)+"1>"
reg .p("Init")
    .hadamardAll.p("H-all")
    .applyF()   .p("f-applied")
    .hadamardAll.p("H-all2")
    .measure    .p("measured")
```

Figure 5.2: *Deutsch-Jozsa implementation. Method calls are chained thanks to the object oriented structure chosen in the golden model.*

5.2 Shor and Grover's algorithms

5.2.1 Shor's algorithm

The Shor's algorithm*[3]* is a very famous quantum algorithm that solves the prime factorisation problem : given a natural number N, find its prime factors. The problem cannot be solved better than with a sub-exponential time complexity with a classical-only algorithm. It has been proved that Shor's algorithm solves the same problem with a $(O(log(N)))$ time complexity! With such a superior speed, the factorisation problem brought by the RSA algorithm could be brute forced in a very reasonable amount of time. The figure (5.3) shows the quantum part of the algorithm.

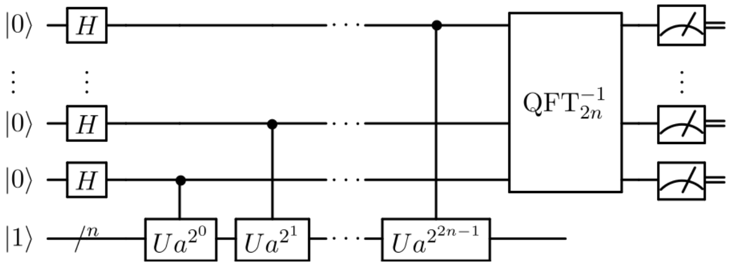


Figure 5.3: *Shor's algorithm quantum circuit. The difficult part is the implementation of the quantum Fourier transformation, which is quite complicated.*

5.2.2 Grover's algorithm

Grover's*[4]* algorithm solves the problem of finding an specific element in a dataset with a very high probability. Any classical algorithm will have at best a $O(N)$

time complexity (where N is the size of the dataset) because in the worst case, the last element is the searched one. However, Grover's algorithm can find a match with high accuracy with a complexity of $O(\sqrt{(N)})$. Furthermore, four other mathematicians[1] proved that Shor's algorithm, is asymptotically optimal, which means that no other quantum algorithm could perform better concerning this problem.

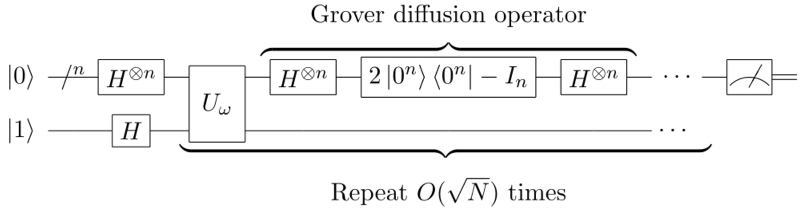

Grover diffusion operator
$|0\rangle$
n
$H^{\otimes n}$
$|1\rangle$
H
U_ω
$H^{\otimes n}$
$2|0^n\rangle\langle 0^n| - I_n$
$H^{\otimes n}$
Repeat $O(\sqrt{N})$ times

Figure 5.4: *Grover's algorithm quantum circuit. The difficult part is the implementation of the Grover diffusion operator.*

While Deutsch-Jozsa algorithm is deterministic (it will always give the right answer given the correct inputs), Grover's algorithm is probabilistic. But by iterating it multiple times, it is possible to get the probability of having a correct result insanely close to 1, which is acceptable.

5.3 Experimental results

In this last section, we will take a look at the QuBits simulator performances and compare it to its previous iteration, but also to other and more complete quantum libraries.

5.3.1 Comparison with other libraries

The most advanced work done with the project's simulator was the implementation of the Deutsch-Jozsa algorithm. To compare the project's simulator with other libraries, I decided to run this algorithm on different solutions and compare the execution times. I chose the javascript library "jsqubits" as my implementation and structure was very similar to this one, thus making the re-implementation of the DJ algorithm much easier. The second library I chose was "qubiter", a Python 3 library. The figure (5.5) shows how these three solutions compares to each other.

As expected, the Javascript and Python libraries work quite faster than the project's Crystal solution. On average, QuBits's simulator was 3.7 times slower than the two others libraries. However, this is not a bad result for a minimalist library made from-scratch by a single student, and without any particular

[1] Bennett, Bernstein, Brassard, and Vazirani

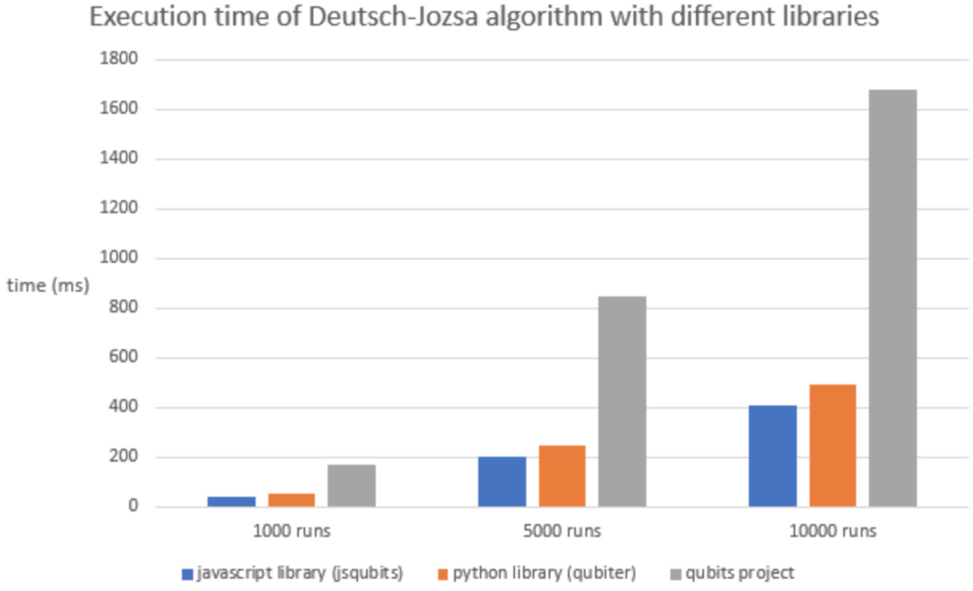


Figure 5.5: *Three libraries compared on their execution time.*

optimization other than changing the programming language from Javascript to Crystal.
In conclusion, even thougth other libraries outperform the project's final implementation, these three solutions are relatively on the same scale of execution time (less than a single miliseconds to run the DJ algorithm once).

5.3.2 Comparison with the previous implementation

But another interesting comparison would be between the first implementation of the project (the golden model in coffeescript) and the last one (the final result with Crystal). Since the golden model was developped early during the project, it was not yet an option to implement quantum algorithm (the project's goal was initially to simply modelize quantum gates). Therefore, the golden model and the final model cannot be compared on the DJ algorithm execution. However, we can instead compare them on the execution time of the hamadard gate. The figure (5.6) shows the results of this experiment.

The results are this time extremely clear : the final version of the project is, on average, 24.2 times faster than the golden Coffeescript model! This illustrates very well how a smarter implementation and the use of a compiles language can accelerate caculations. While the final model takes 0.1 second to run the Hadamard gate 5000 times, the golden model asks for almost three seconds to do the same task!

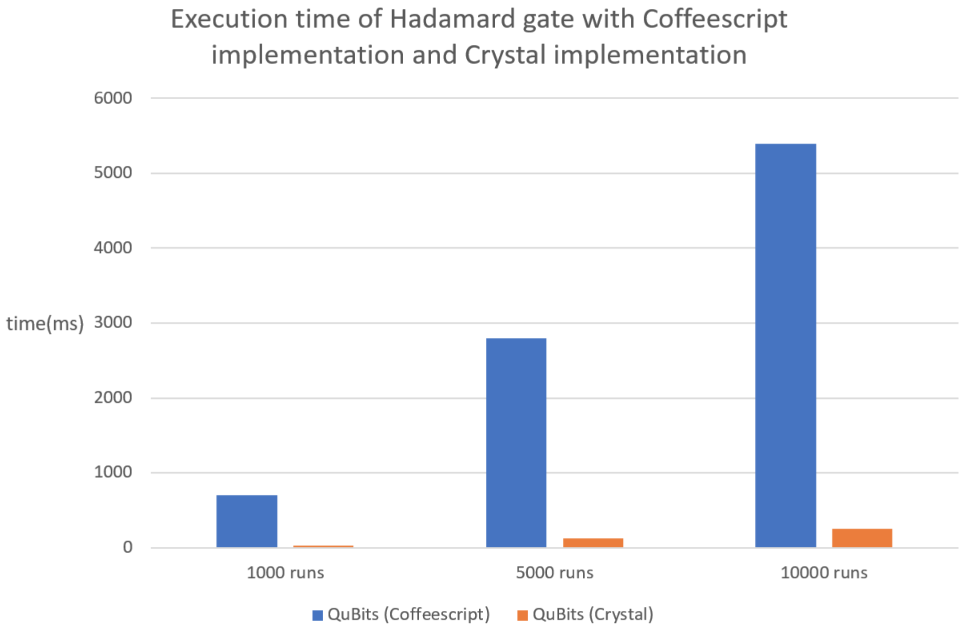


Figure 5.6: *Comparing the execution time of Hadamard gate between the first and the last implementation of the QuBits project.*

6. Conclusion

Diving into the strange world of quantum mechanics and computers was a fascinating experience. I was thrilled to learn so many aspects of this topic by myself, and very proud of being able to implement -if not the hardest one- a quantum algorithm with my simulator. I also learned a new language, Crystal, that is very promising for 2018 and very pleasant to use.

The pleasure I had to make the reseach part of this Project confirms my desire of working in the same way in my future engineer career : a thesis in the quantum computing world would be ideal to me and corresponds to how I like to spend my working time.

Bibliography

[1] https://en.wikiversity.org/wiki/Quantum_mechanics/Timeline Quantum physics timeline, wikiversity, march 2018

[2] https://www.youtube.com/watch?v=oQYsqnRYb_o TED Talk The Crazy History of Quantum Mechanics | Leonard Mlodinow, feb 2016

[3] https://en.wikipedia.org/wiki/Shor%27s_algorithm Shor's algorithm, Wikipedia, march 2018

[4] https://en.wikipedia.org/wiki/Grover%27s_algorithm Grover's algorithm, Wikipedia, march 2018

[5] Mathematics courses of Mr Quibel (Descartes Highschool), second year, Tours, 2014

[6] http://dept-info.labri.fr/~ges/ENSEIGNEMENT/CALCULQ/polycop_calculq.pdf Introduction to quantum conmputing, Y. Leroyer et G. Senizergues, 2016-2017

[7] http://stla.github.io/stlapblog/posts/BlochSphere.html Bloch's sphere, stlaPBlog, Dec 2015

[8] http://stla.github.io/stlapblog/posts/JonctionQuantique.html Junction of two quantum systems, stlaPBlog, Fev 2016

[9] https://tel.archives-ouvertes.fr/tel-00180890/document

Appendix

6.1 Golden model source code (Coffeescript)

```
math = require 'mathjs'

class QuObject
    constructor: () ->

    # (array a#, array a2) -> (float |<a1/a2>|^2)
    prodSc: (a1,a2) ->
        math.norm( math.dot(a1,a2) )^2

    # "00010010" -> [0,0,0,1,0,0,1,0]
    getArr: (str) ->
        ret = []
        for i in [0...str.length]
            ret.push( parseInt(str[i]) )
        ret

    # [0,0,0,1,0,0,1,0] -> "00010010"
    getStr: (arr) ->
        ret = ""
        for i in [0...arr.length]
            ret += "" + arr[i] + ""
        ret

    toBin: (int, dim) ->
        bin = int.toString(2)
        for i in [0...(dim - bin.length)]
            bin = "0" + bin
        bin

    strReplace: (str, index, replacement) ->
        str.substr(0, index) + replacement + str.substr(index +
            replacement.length)


# this class represents the quantum state of a register of qubits
class QuState extends QuObject
```

```
# we suppose coeffs.length == dim
# dim is needed, coeffs might be optional, measured is a boolean
constructor: (@dim, coeffs = [], @measured = false) ->
    super()
    @coeffs = if (coeffs.length == 0) then ( @genCoeffs(@dim) ) else
        coeffs
    @coeffs = math.multiply( 1/math.norm(@coeffs), @coeffs )
        #normalization

# (int dim) -> (array complex [a1, ... an]) / n = 2^dim
genCoeffs: (dim) ->
    [ret, reals, ims] = [ [], math.random([1,@dim])[0],
        math.random([1,@dim])[0] ]
    for i in [0...Math.pow(2,@dim)]
        ret.push( math.complex(reals[i],ims[i]) )
    ret

# (QuState state) -> (QuState postMeasureState = finalState)
getProbas: () ->
    #array of the probability of each state to be measured, the
        random choice, a sum, and the future return result
    probas = []
    for i in [0...Math.pow(2,@dim)]
        # c'est pas propre mathmatiquement mais a marche et c'est +
            rapide
        probas.push( Math.pow( math.norm([@coeffs[i]]),2 ) )
    probas

# return the state after measurment
measure: () ->
    [probas, choice, sum, result] = [ @getProbas(), math.random(), 0,
        [] ]
    #console.log 'CHOICE : ' + choice
    for i in [0...Math.pow(2,@dim)]
        [prevSum, sum] = [sum, sum+probas[i]]
        result.push( if(choice >= prevSum && choice <= sum) then 1
            else 0 )
    #console.log 'RESULT : [' + result + ']'
    new QuState(@dim, result, true)

# if the state is measured, return the value of the n-th bit.
getQubit: (n) ->
    if (!@measured)
        false
    else
        @coeffs[n-1]

getCoeffs: () ->
    @coeffs
    getState: () ->
        ret = ''
        for i in [0...Math.pow(2,@dim)]
            if math.norm(@coeffs[i]) != 0
                ret += '(' + @coeffs[i] + ')*|' + (@toBin i, @dim) + '> + '
        ret.slice(0,-2)

    isMeasured: () ->
        @measured


class QuRegister extends QuObject
    # give "|001>", "/001>", or [0,1,0,0,0,0,0,0], which represents the
        same state
    constructor: (st) ->
        super()
        if ( (st[0] == "|") || (st[0] == "/") ) && ( st[(st.length)-1] ==
            ">" )
            bin = st.slice(1,(st.length)-1)
            dim = bin.length
            arr = ( 0 for [1..(Math.pow(2,dim))] )
            arr[parseInt(parseInt(bin,2))] = 1
        else
            dim = parseInt Math.log2(st.length)
            arr = st

        @quState = new QuState dim, arr


    getState: () ->
        @quState.getState()

    p: (st) ->
        console.log '[' + st + '] '+ @getState()
        @


    measure: () ->
        #replace the quState with its value after measurment
        @quState = @quState.measure()
        @


    hadamard: (x) ->
        dim = @quState.dim
        newCoeffs = ( 0 for [1..(Math.pow(2,dim))] )

        for i in [0...(Math.pow(2,dim))]
```

```
            bin = (@toBin i, dim)
            if (bin[x] == ’0’) # H(|0>) = ( 1/sqrt(2) )*( |0> + |1> )
                [c1,c2] = [bin, (@strReplace bin, x, "1" )]
                console.log "1/ " + c1 + " -- " + c2
                newCoeffs[parseInt(c1,2)] +=
                    (@quState.coeffs[i]/Math.sqrt(2))
                newCoeffs[parseInt(c2,2)] +=
                    (@quState.coeffs[i]/Math.sqrt(2))
            else # H(|1>) = ( 1/sqrt(2) )*( |0> - |1> )
                [c1,c2] = [(@strReplace bin, x, "0" ), bin]
                console.log "2/ " + c1 + " -- " + c2
                newCoeffs[parseInt(c1,2)] +=
                    (@quState.coeffs[i]/Math.sqrt(2))
                newCoeffs[parseInt(c2,2)] -=
                    (@quState.coeffs[i]/Math.sqrt(2))

        @quState = new QuState dim, newCoeffs
        @


    hadamardAll: () ->
        for i in [0...@quState.dim]
            @hadamard i
        @


    swap: (x, y) ->
        dim = @quState.dim
        newCoeffs = ( 0 for [1..(Math.pow(2,dim))] )

        for i in [0...(Math.pow(2,dim))]
            bin = (@toBin i, dim).split(’’)
            [ bin[x], bin[y] ] = [ bin[y], bin[x] ]
            bin = bin.join(’’)
            newCoeffs[parseInt(bin,2)] = @quState.coeffs[i]

        @quState = new QuState dim, newCoeffs
        @


    not: (x) ->
        dim = @quState.dim
        newCoeffs = ( 0 for [1..(Math.pow(2,dim))] )

        for i in [0...(Math.pow(2,dim))]
            bin = (@toBin i, dim)
            if (bin[x] == "0") # NOT(|0>) = |1>
                bin = @strReplace(bin, x, "1")
            else # NOT(|1>) = |0>
```

```
                bin = @strReplace(bin, x, "0")
            newCoeffs[parseInt(bin,2)] += @quState.coeffs[i]

        @quState = new QuState dim, newCoeffs
        @


    cnot: (x,y) ->
        dim = @quState.dim
        newCoeffs = ( 0 for [1..(Math.pow(2,dim))] )

        for i in [0...(Math.pow(2,dim))]
            bin = (@toBin i, dim)
            if (bin[x] == "1")
                if (bin[y] == "0") # NOT(|0>) = |1>
                    bin = @strReplace(bin, y, "1")
                else # NOT(|1>) = |0>
                    bin = @strReplace(bin, y, "0")
            newCoeffs[parseInt(bin,2)] += @quState.coeffs[i]

        @quState = new QuState dim, newCoeffs
        @


    phase: (x,phi) ->
        dim = @quState.dim
        newCoeffs = ( 0 for [1..(Math.pow(2,dim))] )

        for i in [0...(Math.pow(2,dim))]
            bin = (@toBin i, dim)
            if (bin[x] == "1")
                newCoeffs[parseInt(bin,2)] = math.multiply(math.complex({r
                    : 1, phi : phi}), @quState.coeffs[i])
            else
                newCoeffs[parseInt(bin,2)] = @quState.coeffs[i]

        @quState = new QuState dim, newCoeffs
        @


    cphase: (x,y,phi) ->
        dim = @quState.dim
        newCoeffs = ( 0 for [1..(Math.pow(2,dim))] )

        for i in [0...(Math.pow(2,dim))]
            bin = (@toBin i, dim)
            if (bin[x] == "1" && bin[y] == "1")
                newCoeffs[parseInt(bin,2)] = math.multiply(math.complex({r
                    : 1, phi : phi}), @quState.coeffs[i])
            else
                newCoeffs[parseInt(bin,2)] = @quState.coeffs[i]

        @quState = new QuState dim, newCoeffs
        @


# testing measure()
reg0 = new QuRegister [math.complex(0,4),0,0,4,0,0,0,0]
console.log "before measure : " + reg0.getState()
reg0.measure()
console.log "after measure : " + reg0.getState()

# testing hadamard
reg1 = new QuRegister [1,0,0,0,0,0,1,0]
console.log "HADAMARD(0) : " + reg1.getState() + " ---> " +
    reg1.hadamard(0).getState()

# testing hadamard on all qubits
reg2 = new QuRegister [1,0,0,0,0,0,1,0]
console.log "HADAMARD ALL : " + reg2.getState() + " ---> " +
    reg2.hadamardAll().getState()

# testing swap on qubits 0 and 2
reg3 = new QuRegister "|001>" # = [0,1,0,0,0,0,0,0]
console.log "SWAP(0,2) : " + reg3.getState() + " ---> " +
    reg3.swap(0,2).getState()

# testing not on qubit 0
reg4 = new QuRegister "|001>"
console.log "NOT(0) : " + reg4.getState() + " ---> " +
    reg4.not(0).getState()

# testing cnot on qubit 2 with controlled qubit 0
reg5 = new QuRegister [0,0,0,0,1,0,0,0]
console.log "CNOT(0,2) : " + reg5.getState() + " ---> " +
    reg5.cnot(0,2).getState()

# testing phase on qubit 1
reg6 = new QuRegister "/010>"
console.log "PHASE(1,e^(i*PI/4)) : " + reg6.getState() + " ---> " +
    reg6.phase(1,(math.PI)/4).getState()

# testing phase on qubit 1 with controlled qubit 0
reg7 = new QuRegister "/110>"
console.log "CPHASE(0,1,e^(-i*PI/4)) : " + reg7.getState() + " ---> " +
    reg7.cphase(0,1,-(math.PI)/4).getState()
```

```
console.log "\n\n
    -------------------------------------------------------------------
    \n\n"


reg8 = new QuRegister "/010>"
reg8.p('Init')
    .hadamard(1).p('Hadamard(1)')
    .swap(0,1) .p( 'Swap(0,1)' )
    .cnot(0,2) .p( 'Cnot(0,2)' )
    .not(0)    .p(  'Not(0)' )
    .measure() .p(' Measure()' )

start = new Date().getTime()
for i in [0...10000]
    reg9 = new QuRegister "/010>"
    reg9.hadamardAll()
end = new Date().getTime()
console.log("time taken : "+(end-start))
```

6.2 Final implementation source code (Crystal)

```
require "complex"

class QuObject
    #def initialize() end

    def toBin(num : Int32, dim : Int32)
        bin = num.to_s 2
        (dim-bin.size).times do |i|
            bin = "0" + bin
        end
        bin
    end

    def newCoeffs(dim : Int32)
        newCoeffs = [] of Complex
        (2**dim).times do |i|
            newCoeffs.push Complex.new 0, 0
        end
        newCoeffs
    end

    def norm(vect : Array)
        norm = 0
        vect.each do |el|
            norm += (el*el.conj).abs
        end
        Math.sqrt norm
    end

end

class QuState < QuObject

    def initialize(dim : Int32, coeffs = [] of Complex, measured = false)
        @dim = dim
        if coeffs.size == 0
            @coeffs = genCoeffs @dim
        else
            @coeffs = coeffs
        end
            val = (1/norm(@coeffs))
        #@coeffs = (1/norm(@coeffs))*@coeffs #normalization
            (2**dim).times do |i| #normalization
               @coeffs[i] *= val
            end
    end

    def genCoeffs(dim)
        ret = [] of Complex
        (2**dim).times do |i|
            ret.push Complex.new(Random.rand(1),Random.rand(1))
        end
        ret
    end

    def getProbas()
        probas = [] of Float64
        (2**@dim).times do |i|
            probas.push @coeffs[i].abs2()
        end
        probas
    end

   def getDim()
      @dim
   end

   def getCoeffs()
      @coeffs
   end

    def measure()
        probas, choice, sum, result = getProbas(), Random.rand(1.0), 0,
            [] of Complex
        (2**@dim).times do |i|
            prevSum, sum = sum, sum + probas[i]
            if (choice >= prevSum && choice <= sum)
                result.push Complex.new 1, 0
            else
                result.push Complex.new 0, 0
            end
        end
        QuState.new @dim, result, true
        #@coeffs = result
    end

   def getState()
        ret = ""
        (2**@dim).times do |i|
            if (@coeffs[i].abs != 0)
                ret += "(" + @coeffs[i].to_s + ")*|" + (toBin i,@dim) + ">
                    + "
         end
        end
        ret[0..-4]
    end

    def isMeasured()
        @measured
    end

end


class QuRegister < QuObject
    # give "|001>", "/001>", or [0,1,0,0,0,0,0,0], which represents the
        same state
    def initialize(st : String)
        bin = st[1..-2]
        dim = bin.size
        arr = [] of Complex
        (2**dim).times do |i|
            arr.push Complex.new 0, 0
        end
        arr[(bin.to_i 2)] = Complex.new 1, 0
        @quState = QuState.new dim, arr
    end

    def initialize(st : Array(Complex))
        dim = Math.log2(st.size).to_i()
        arr = st
        @quState = QuState.new dim, arr
    end
```

```
def getState()
    @quState.getState()
end

def getQuState()
    @quState
end

def p(st)
    puts "[" + st + "] " + getState()
    self
end

def measure()
    @quState = @quState.measure()
    self
end

def hadamard(x)
    dim = @quState.getDim
    newCoeffs = newCoeffs(dim)

    (2**dim).times do |i|
        bin = toBin i, dim
        if bin[x] == ’0’ # H(|0>) = ( 1/sqrt(2) )*( |0> + |1> )
            c1,c2 = bin, bin.sub(x, "1")
           newCoeffs[c1.to_i 2] += @quState.getCoeffs[i] / Math.sqrt(2)
            newCoeffs[c2.to_i 2] += @quState.getCoeffs[i] /
                Math.sqrt(2)
        else # H(|1>) = ( 1/sqrt(2) )*( |0> - |1> )
            c1,c2 = bin.sub(x, "0"), bin
            newCoeffs[c1.to_i 2] += @quState.getCoeffs[i] /
                Math.sqrt(2)
            newCoeffs[c2.to_i 2] -= @quState.getCoeffs[i] /
                Math.sqrt(2)
        end
    end

    @quState = QuState.new dim, newCoeffs
    self
end

def hadamardAll()
    (@quState.getDim).times do |i|
        hadamard i
    end
    self
end
```

```
def swap(x, y)
    dim = @quState.getDim
    newCoeffs = newCoeffs(dim)

    (2**dim).times do |i|
        bin = (toBin i, dim).chars
        bin[x], bin[y] = bin[y], bin[x]
        newCoeffs[bin.join.to_i 2] = @quState.getCoeffs[i]
    end

    @quState = QuState.new dim, newCoeffs
    self
end

def not(x)
    dim = @quState.getDim
    newCoeffs = newCoeffs(dim)

    (2**dim).times do |i|
        bin = (toBin i, dim)
        if bin[x] == ’0’ # NOT(|0>) = |1>
            bin = bin.sub(x, "1")
        else # NOT(|1>) = |0>
            bin = bin.sub(x, "0")
        end
        newCoeffs[bin.to_i 2] += @quState.getCoeffs[i]
    end

    @quState = QuState.new dim, newCoeffs
    self
end

def cnot(x,y)
    dim = @quState.getDim
    newCoeffs = newCoeffs(dim)

    (2**dim).times do |i|
        bin = (toBin i, dim)
        if bin[x] == ’1’ # Control bit
            if bin[y] == ’0’ # NOT(|0>) = |1>
                bin = bin.sub(y, "1")
            else # NOT(|1>) = |0>
                bin = bin.sub(y, "0")
            end
        end
        newCoeffs[bin.to_i 2] += @quState.getCoeffs[i]
    end
```

```
        @quState = QuState.new dim, newCoeffs
        self
    end

    def phase(x, phi)
        dim = @quState.getDim
        newCoeffs = newCoeffs(dim)

        (2**dim).times do |i|
            bin = (toBin i, dim)
            if bin[x] == ’1’
                newCoeffs[bin.to_i 2] = (@quState.getCoeffs[i]) *
                    (Complex.new 1, phi)
            else
                newCoeffs[bin.to_i 2] = (@quState.getCoeffs[i])
            end
        end

        @quState = QuState.new dim, newCoeffs
        self
    end

    def cphase(x, y, phi)
        dim = @quState.getDim
        newCoeffs = newCoeffs(dim)

        (2**dim).times do |i|
            bin = (toBin i, dim)
            if bin[x] == ’1’ && bin[y] == ’1’
                newCoeffs[bin.to_i 2] = (@quState.getCoeffs[i]) *
                    (Complex.new 1, phi)
            else
                newCoeffs[bin.to_i 2] = (@quState.getCoeffs[i])
            end
        end

        @quState = QuState.new dim, newCoeffs
        self
    end

    def applyF()
        dim = @quState.getDim
        newCoeffs = newCoeffs(dim)

        (2**dim).times do |i|
            bin = (toBin i, dim)
            x, q = bin.rchop, bin[bin.size-1].to_i
            arr = [] of Complex
            (2**x.size).times do |i|
                arr.push Complex.new 0, 0
            end
            arr[(x.to_i 2)] = Complex.new 1, 0
            newqs = QuState.new x.size, arr
            q = (q + c0(newqs)) % 2 # HERE WE USE THE ORACLE
            newbin = x + q.to_s
            newCoeffs[newbin.to_i 2] = @quState.getCoeffs[bin.to_i 2]
        end

        @quState = QuState.new dim, newCoeffs
        self
    end

    def  findI(qs : QuState) #here, we assume the qubits regiter is in
        one single state
        coeffs = qs.getCoeffs
        ret = -1
        (coeffs.size).times do |i|
            if(coeffs[i].real == 1.0)
                ret = i
            end
        end
        ret
    end

    def c0(qs : QuState) #constant 0
        0
    end

    def c1(qs : QuState) # constant 1
        1
    end

    def b1(qs : QuState) #balance n1
        bin = toBin findI(qs), qs.getDim
        if(bin[0] == ’0’)
            0
        else
            1
        end
    end

    def b2(qs : QuState) # balance n2
        bin = toBin findI(qs), qs.getDim
        if(bin[qs.getDim-1] == ’0’)
            0
        else
            1
        end
    end
end

# CMD + /

#pour l'instant a  galre , je devrais peut etre forcer le typage en full
    nombre complexe, et forcer le constructeur par coeffs seulement,
    qutte  ecrire des convertisseurs? Dans cette version , je fais a

def c(re, im = 0)
    Complex.new(re,im)
end

def ca(arr) # array of int or floats (not complex for now)
    (arr.size).times do |i|
        # a faire
    end
end

#DEUTSH-JORZA
n = 3
tstart = Time.now()
(5000).times do |i|
    reg = QuRegister.new "|"+("0"*n)+"1>"
    reg .p("Init")
        .hadamardAll.p("H-all")
        .applyF()  .p("f-applied")
        .hadamardAll.p("H-all2")
        .measure   .p("measured")
end
tend = Time.now()
puts "time : "
puts (tend - tstart).milliseconds()
puts (tend - tstart).seconds()

#HADAMARD-ALL
n = 3
tstart = Time.now()
(10000).times do |i|
    reg = QuRegister.new "|010>"
    reg.hadamardAll
end
tend = Time.now()
puts "time : "
puts (tend - tstart).milliseconds()
puts (tend - tstart).seconds()
```

6.3 Mathematics of qubits (French)

QUBITS

QUÉLARD Xavier

12 mars 2018

Table des matières

1. Simulation informatique

1.1 Rappels mathématiques

Toutes les notions et les définitions abordées dans le chapitre 1 sont issues des cours de mathématiques auxquels j'ai assisté en classe préparatoire[1] .

1.1.1 Loi de composition interne et groupe

Une loi de composition interne $\star$ sur l'ensemble X est une aplication de la forme :

$$\star : X \times X \to X$$

Soit G un ensemble non vide, muni d'une loi de composition interne $\oplus$. (G, $\oplus$) est un groupe abélien $\Leftrightarrow$

- $\oplus$ associative : $\forall(x, y, z) \in G^3, (x \oplus y) \oplus z = x \oplus (y \oplus z)$
- $\exists e \in G/x \oplus e = e \oplus x = x$ (e est le neutre du groupe G selon la loi $\oplus$)
- $\oplus$ commutative : $\forall(x, y) \in G^2, x \oplus y = y \oplus x$

1.1.2 Anneaux et corps

Disposant de la définition d'un corps commutatif, nous pouvons maintenant donner la définition d'un anneau commutatif. Soit A un ensemble non vide, muni de deux lois de compositions interne $\oplus$ et $\otimes$.

Alors $(A, \oplus, \otimes)$ anneau commutatif $\Leftrightarrow$

- $(A, \oplus)$ est un groupe commutatif
- la loi $\otimes$ est associative
- la loi $\otimes$ est distributive par rapport à la loi $\oplus$, c'est-à-dire que :

$$\forall(x, y, z) \in A^3, (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$$

- la loi $\otimes$ est commutative : $\forall(x1, x2) \in A^2, x1 \otimes x2 = x2 \otimes x1$

Un corps commutatif est alors simplement un anneau commutatif dont tous les éléments sont inversibles, exceptés le neutre pour l'opération $\oplus$. Il est alors aisé de remarquer que l'ensemble $\mathbb{R}$ ou bien $\mathbb{C}$ sont, munis des opérations $(+, \times)$, des corps.

1.1.3 espace vectoriel et produit scalaire

Soit E un ensemble non vide et $(\mathbb{K}, +, \times)$ un corps de neutre $0_{\mathbb{K}}$ pour $+$ et $1_{\mathbb{K}}$ pour $\times$. On note $(E, +, \cdot)$ l'ensemble E muni de la même loi $+$ que $\mathbb{K}$ (c'est donc une loi interne à E), et d'une loi externe $\cdot : \mathbb{K} \times E \to E$.

Alors E est un espace vectoriel $\Leftrightarrow$

- $(E, +)$ est un groupe commutatif
- $\forall \lambda \in \mathbb{K}, \forall (x, y) \in E^2, \lambda \cdot (x + y) = (\lambda \cdot x) + (\lambda \cdot y)$ (distributivité)
- $\forall (\lambda, \mu) \in \mathbb{K}^2, \forall x \in E, (\lambda + \mu) \cdot x = \lambda \cdot x + \mu \cdot x$
- $\forall (\lambda, \mu) \in \mathbb{K}^2, \forall x \in E, (\lambda \times \mu) \cdot x = \lambda \cdot (\mu \cdot x)$
- $\forall x \in E, 1_{\mathbb{K}} \cdot x = x$

On appele les éléments de E des vecteurs, les éléments de $\mathbb{K}$ des scalaires, et le vecteur $0_{\mathbb{K}}$ est appelé le vecteur nul. En résumé, un espace vectoriel est un espace E consitué d'éléments appelés vecteurs, qui sont stables par addition et par multiplication d'un scalaire. Les espaces $(\mathbb{R}, +, \cdot)$ et $(\mathbb{C}, +, \cdot)$ sont donc des espaces vectoriels (le second est appelé espace vectoriel complexe).

On appele produit scalaire sur E (espace vectoriel) toute forme bilinéaire symétrique et définie positive, c'est-à-dire :

- forme : c'est une application du type $\langle \cdot / \cdot \rangle : \begin{cases} E \times E \to \mathbb{K} \\ (u, v) \mapsto \langle u/v \rangle \end{cases}$
- symétrie : $\forall (u, v) \in E^2, \langle u/v \rangle = \langle v/u \rangle$
- linéarité à droite : $\forall (u, v, w) \in E^3, \langle u/v + w \rangle = \langle u/v \rangle + \langle u/w \rangle$ (de la symétrie et la linéarité découle alors la bi-linéarité)
- defini positif : $\forall u \in E, \langle u/u \rangle \geq 0$ et $\forall u \in E, \langle u/u \rangle = 0 \Leftrightarrow u = 0_E$

Il est important de remarquer que s'il l'on se place dans un espace vectoriel complexe, le produit scalaire donne un nombre complexe, tandis qu'en se plaçant dans un espace vectoriel sur le corps des réels, le produit scalaire donnera lui même un réel. De manière générale, il associe à vecteur un élément du corps $\mathbb{K}$.

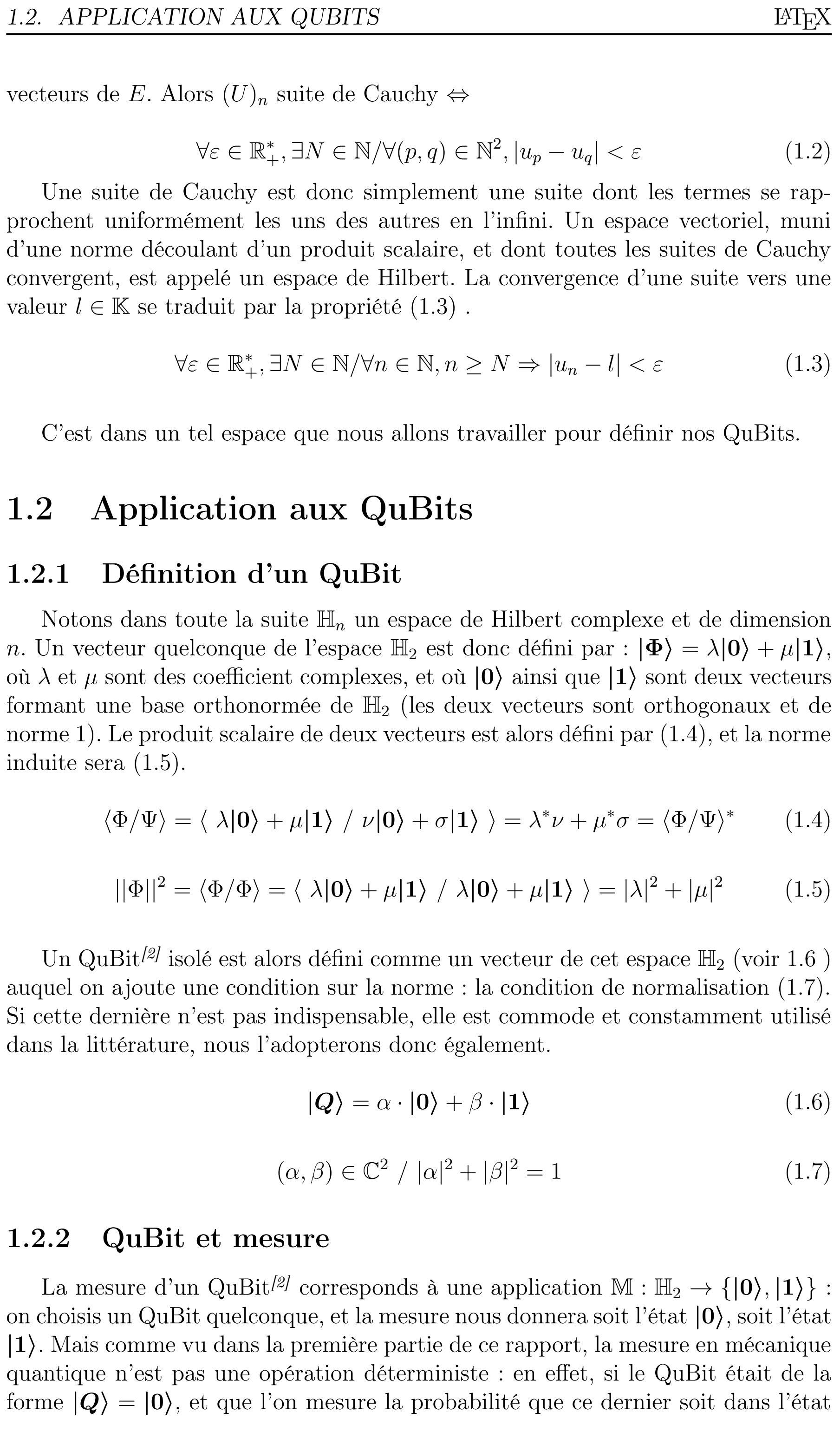
1.1.4 norme induite, suites de Cauchy et espace de Hilbert

Une norme est une application $N : E \to \mathbb{R}_+$ et qui satisfait l'hypothèse de séparation ($\forall u \in E, N(u) = 0 \Rightarrow u = 0_E$), d'absolue homogénéité ($\forall \lambda \in \mathbb{K}, \forall u \in E, N(\lambda \cdot u) = \lambda \cdot N(u)$), et l'inégalité triangulaire ($\forall (u, v) \in E^2, N(u + v) \leq N(u) + n(v)$).

A chaque produit scalaire est associé une norme, que l'on appele norme induite par le produit scalaire . Elle est définie par :

$$N(\cdot) : \begin{cases} E \to \mathbb{R}_+ \\ u \mapsto \sqrt{\langle u/u \rangle} \end{cases} \tag{1.1}$$

Une fois que nous possédons une norme, il est possible d'introduire la notion de convergence, mais aussi de définir les suites de Cauchy . Soit $(U)_n$ une suite de



vecteurs de E. Alors $(U)_n$ suite de Cauchy $\Leftrightarrow$

$$\forall \varepsilon \in \mathbb{R}_+^*, \exists N \in \mathbb{N} / \forall (p,q) \in \mathbb{N}^2, |u_p - u_q| < \varepsilon \tag{1.2}$$

Une suite de Cauchy est donc simplement une suite dont les termes se rapprochent uniformément les uns des autres en l'infini. Un espace vectoriel, muni d'une norme découlant d'un produit scalaire, et dont toutes les suites de Cauchy convergent, est appelé un espace de Hilbert. La convergence d'une suite vers une valeur $l \in \mathbb{K}$ se traduit par la propriété (1.3) .

$$\forall \varepsilon \in \mathbb{R}_+^*, \exists N \in \mathbb{N} / \forall n \in \mathbb{N}, n \geq N \Rightarrow |u_n - l| < \varepsilon \tag{1.3}$$

C'est dans un tel espace que nous allons travailler pour définir nos QuBits.

1.2 Application aux QuBits

1.2.1 Définition d'un QuBit

Notons dans toute la suite $\mathbb{H}_n$ un espace de Hilbert complexe et de dimension n. Un vecteur quelconque de l'espace $\mathbb{H}_2$ est donc défini par : $|\boldsymbol{\Phi}\rangle = \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle$, où λ et μ sont des coefficient complexes, et où $|\mathbf{0}\rangle$ ainsi que $|\mathbf{1}\rangle$ sont deux vecteurs formant une base orthonormée de $\mathbb{H}_2$ (les deux vecteurs sont orthogonaux et de norme 1). Le produit scalaire de deux vecteurs est alors défini par (1.4), et la norme induite sera (1.5).

$$\langle \Phi / \Psi \rangle = \langle\ \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle\ /\ \nu|\mathbf{0}\rangle + \sigma|\mathbf{1}\rangle\ \rangle = \lambda^* \nu + \mu^* \sigma = \langle \Phi / \Psi \rangle^* \tag{1.4}$$

$$||\Phi||^2 = \langle \Phi / \Phi \rangle = \langle\ \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle\ /\ \lambda|\mathbf{0}\rangle + \mu|\mathbf{1}\rangle\ \rangle = |\lambda|^2 + |\mu|^2 \tag{1.5}$$

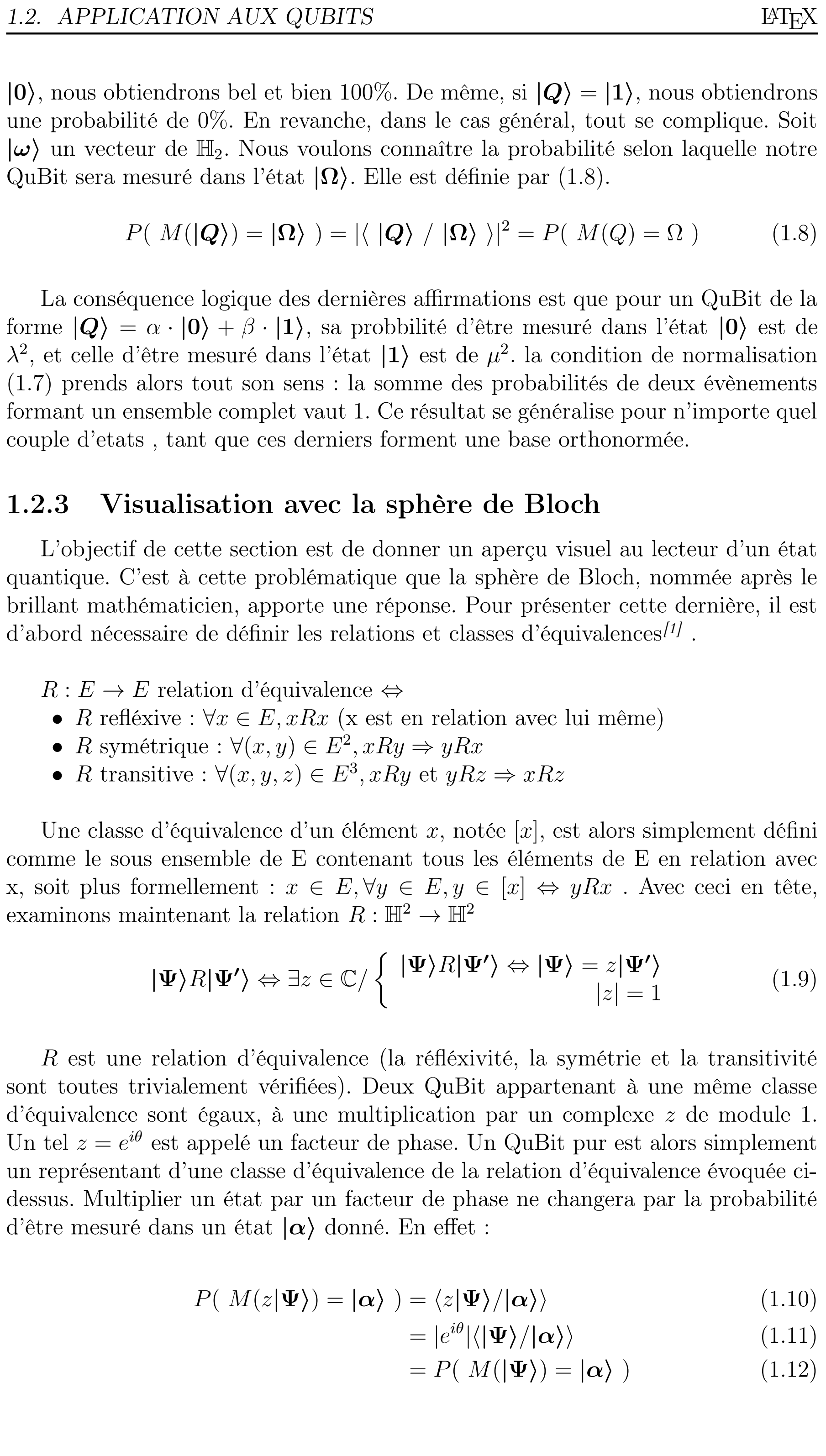
Un QuBit/2/ isolé est alors défini comme un vecteur de cet espace $\mathbb{H}_2$ (voir 1.6) auquel on ajoute une condition sur la norme : la condition de normalisation (1.7). Si cette dernière n'est pas indispensable, elle est commode et constamment utilisé dans la littérature, nous l'adopterons donc également.

$$|\boldsymbol{Q}\rangle = \alpha \cdot |\mathbf{0}\rangle + \beta \cdot |\mathbf{1}\rangle \tag{1.6}$$

$$(\alpha, \beta) \in \mathbb{C}^2\ /\ |\alpha|^2 + |\beta|^2 = 1 \tag{1.7}$$

1.2.2 QuBit et mesure

La mesure d'un QuBit/2/ corresponds à une application $\mathbb{M} : \mathbb{H}_2 \to \{|\mathbf{0}\rangle, |\mathbf{1}\rangle\}$: on choisis un QuBit quelconque, et la mesure nous donnera soit l'état $|\mathbf{0}\rangle$, soit l'état $|\mathbf{1}\rangle$. Mais comme vu dans la première partie de ce rapport, la mesure en mécanique quantique n'est pas une opération déterministe : en effet, si le QuBit était de la forme $|\boldsymbol{Q}\rangle = |\mathbf{0}\rangle$, et que l'on mesure la probabilité que ce dernier soit dans l'état



$|\mathbf{0}\rangle$, nous obtiendrons bel et bien 100%. De même, si $|\boldsymbol{Q}\rangle = |\mathbf{1}\rangle$, nous obtiendrons une probabilité de 0%. En revanche, dans le cas général, tout se complique. Soit $|\boldsymbol{\omega}\rangle$ un vecteur de $\mathbb{H}_2$. Nous voulons connaître la probabilité selon laquelle notre QuBit sera mesuré dans l'état $|\boldsymbol{\Omega}\rangle$. Elle est définie par (1.8).

$$P(\ M(|\boldsymbol{Q}\rangle) = |\boldsymbol{\Omega}\rangle\ ) = |\langle\ |\boldsymbol{Q}\rangle\ /\ |\boldsymbol{\Omega}\rangle\ \rangle|^2 = P(\ M(Q) = \Omega\ ) \tag{1.8}$$

La conséquence logique des dernières affirmations est que pour un QuBit de la forme $|\boldsymbol{Q}\rangle = \alpha \cdot |\mathbf{0}\rangle + \beta \cdot |\mathbf{1}\rangle$, sa probbilité d'être mesuré dans l'état $|\mathbf{0}\rangle$ est de λ^2, et celle d'être mesuré dans l'état $|\mathbf{1}\rangle$ est de μ^2. la condition de normalisation (1.7) prends alors tout son sens : la somme des probabilités de deux évènements formant un ensemble complet vaut 1. Ce résultat se généralise pour n'importe quel couple d'etats , tant que ces derniers forment une base orthonormée.

1.2.3 Visualisation avec la sphère de Bloch

L'objectif de cette section est de donner un aperçu visuel au lecteur d'un état quantique. C'est à cette problématique que la sphère de Bloch, nommée après le brillant mathématicien, apporte une réponse. Pour présenter cette dernière, il est d'abord nécessaire de définir les relations et classes d'équivalences[1] .

$R : E \to E$ relation d'équivalence $\Leftrightarrow$

- R refléxive : $\forall x \in E, xRx$ (x est en relation avec lui même)
- R symétrique : $\forall (x, y) \in E^2, xRy \Rightarrow yRx$
- R transitive : $\forall (x, y, z) \in E^3, xRy$ et $yRz \Rightarrow xRz$

Une classe d'équivalence d'un élément x, notée $[x]$, est alors simplement défini comme le sous ensemble de E contenant tous les éléments de E en relation avec x, soit plus formellement : $x \in E, \forall y \in E, y \in [x] \Leftrightarrow yRx$. Avec ceci en tête, examinons maintenant la relation $R : \mathbb{H}^2 \to \mathbb{H}^2$

$$|\boldsymbol{\Psi}\rangle R |\boldsymbol{\Psi'}\rangle \Leftrightarrow \exists z \in \mathbb{C} / \begin{cases} |\boldsymbol{\Psi}\rangle R |\boldsymbol{\Psi'}\rangle \Leftrightarrow |\boldsymbol{\Psi}\rangle = z|\boldsymbol{\Psi'}\rangle \\ \hfill |z| = 1 \end{cases} \tag{1.9}$$

R est une relation d'équivalence (la refléxivité, la symétrie et la transitivité sont toutes trivialement vérifiées). Deux QuBit appartenant à une même classe d'équivalence sont égaux, à une multiplication par un complexe z de module 1. Un tel $z = e^{i\theta}$ est appelé un facteur de phase. Un QuBit pur est alors simplement un représentant d'une classe d'équivalence de la relation d'équivalence évoquée ci-dessus. Multiplier un état par un facteur de phase ne changera par la probabilité d'être mesuré dans un état $|\boldsymbol{\alpha}\rangle$ donné. En effet :

$$\begin{aligned} P(\ M(z|\boldsymbol{\Psi}\rangle) = |\boldsymbol{\alpha}\rangle\ ) &= \langle z|\boldsymbol{\Psi}\rangle / |\boldsymbol{\alpha}\rangle\rangle && (1.10) \\ &= |e^{i\theta}|\langle|\boldsymbol{\Psi}\rangle / |\boldsymbol{\alpha}\rangle\rangle && (1.11) \\ &= P(\ M(|\boldsymbol{\Psi}\rangle) = |\boldsymbol{\alpha}\rangle\ ) && (1.12) \end{aligned}$$

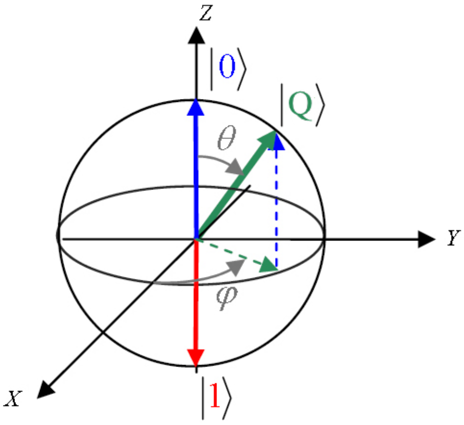


FIGURE 1.1 – *Visualisation d'un état quantique dans la sphère de Bloch.*

On peux alors supposer que deux vecteurs de $\mathbb{H}^2$ sont équivalents s'ils ne diffèrent que par un facteur de phase*[3]* , ce qui amène à la réécriture suivante :

$$\forall|\mathbf{\Psi}\rangle \in \mathbb{H}^2, \exists\theta \in [0,\pi] \ / \ |\mathbf{\Psi}\rangle = \cos(\theta/2)\cdot|\mathbf{0}\rangle + \sin(\theta/2)e^{i(\varphi)}\cdot|\mathbf{1}\rangle \quad (1.13)$$

$$\equiv \cos(\theta/2)e^{-i(\varphi/2)}\cdot|\mathbf{0}\rangle + \sin(\theta/2)e^{i(\varphi/2)}\cdot|\mathbf{1}\rangle \quad (1.14)$$

Sous cette nouvelle forme, il est possible de représenter tout QuBit pur comme point sur une sphère, où θ et φ font office de colatitude et longitude (voir figure 1.1). La correspondance entre l'espace des QuBit purs et une sphère de Bloch est un isomorphisme : une application bijective entre deux espaces, dont la réciproque est également bijective. On constate cependant sur la figure 1.1 que la linéarité n'est pas préservée (en effet, $|\mathbf{0}\rangle \neq -|\mathbf{1}\rangle$). Il faut ainsi considérer cette relation comme une occasion de visualiser plus simplement l'état des QuBits, mais ne permettant pas nécessairement d'interprétations graphiques cohérentes.

1.2.4 extension à plusieurs QuBits

Le passage de un à deux QuBit apporte de nombreuses nouveautés et dévoile une complexité qui pourrait sembler à priori innatendue. Mais c'est cette dernière qui donne tout son potentiel à l'informatique quantique. Pour expliquer ce dernier, il est nécessaire de définir le produit tensoriel $\otimes$ de deux QuBits (1.15) .

$$(\cdot)\otimes(\cdot) : \begin{cases} \mathbb{H}^2 \times \mathbb{H}^2 \to \mathbb{H}^4 \\ (|\boldsymbol{\varphi}\rangle, |\boldsymbol{\Psi}\rangle) \mapsto |\boldsymbol{\varphi}\rangle \otimes |\boldsymbol{\Psi}\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix} \otimes \begin{pmatrix}\lambda\\ \mu\end{pmatrix} = \begin{pmatrix}\alpha\lambda\\ \alpha\mu\\ \beta\lambda\\ \beta\mu\end{pmatrix}\end{cases} \tag{1.15}$$

Le produit tensoriel[4] de deux espaces $\mathbb{H}^2$, qui corresponds à l'espace dans lequel deux QuBits existent, est défini par un espace dont la base est égale à toutes les combinaisons possibles de produit tensoriels entre les vecteurs de bases des deux espaces de départ. C'est donc un espace de dimension 4 ayant pour base orthonormée :

$$\{|\boldsymbol{0_A} \otimes \boldsymbol{0_B}\rangle, |\boldsymbol{0_A} \otimes \boldsymbol{1_B}\rangle, |\boldsymbol{1_A} \otimes \boldsymbol{0_B}\rangle, |\boldsymbol{1_A} \otimes \boldsymbol{1_B}\rangle\} = \{|\mathbf{00}\rangle, |\mathbf{01}\rangle, |\mathbf{10}\rangle, |\mathbf{11}\rangle\} \tag{1.16}$$

$$= \left\{ \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \right\} \tag{1.17}$$

$$= \left\{ \begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}, \begin{pmatrix}0 & 1\\0 & 0\end{pmatrix}, \begin{pmatrix}0 & 0\\1 & 0\end{pmatrix}, \begin{pmatrix}0 & 0\\0 & 1\end{pmatrix} \right\} \tag{1.18}$$

On définit les états quantiques purs[4] comme étant ceux pouvant s'écrire $|\boldsymbol{\varphi}\rangle \otimes |\boldsymbol{\Psi}\rangle$. En reprenant les coefficients utilisés dans l'equation (1.15), on en déduit que les états quantiques purs sont ceux de la forme :

$$|\boldsymbol{\varphi}\rangle \otimes |\boldsymbol{\Psi}\rangle = \alpha\lambda \cdot |\mathbf{00}\rangle + \alpha\mu \cdot |\mathbf{01}\rangle + \beta\lambda \cdot |\mathbf{10}\rangle + \beta\mu \cdot |\mathbf{11}\rangle \tag{1.19}$$

Les états quantiques n'étant pas purs sont dits intriqués, et l'existence de ces derniers a remis en cause le principe de localité que nous avons évoqués dans la partie 1. En effet, si l'on considère l'état de Bell : $|\boldsymbol{\Psi}\rangle = \frac{1}{\sqrt{2}} \cdot (\ |\mathbf{00}\rangle + |\mathbf{11}\rangle\ )$, alors une mesure de l'un des deux QuBit dans sa base respective nous donnera immédiatement l'information de la mesure du second QuBit dans sa propre base, même si les deux QuBit, après intrication, sont séparés d'un potentielle très grande distance[1].

Il semblerait tentant de faire un parallèle à l'informatique classique : deux bits peuvent ainsi former un registre dans seulement 4 états possibles : $00, 01, 10, 11$.

[1] Des tests expérimentaux confirme de nos jours cet effet pour une distance supérieure à 10km.

Cependant, là où il n'y a que quatre possibilités fixes en informatique classiques, il est nécessaire de se rappeler que dans $\mathbb{H}^4$, les vecteurs / registres de deux QuBits sont des combinaisons linéaires des vecteurs $\{|\mathbf{00}\rangle, |\mathbf{01}\rangle, |\mathbf{10}\rangle, |\mathbf{11}\rangle\}$. Cela veut donc dire qu'avant une mesure, un registre de QuBit effectue simultanément les calculs pour **chacune** des quatre possibilitées. Pour un registre de deux QuBit, nous avons donc $4 = 2^2$ calculs effectués en parallèle. Un registre de trois QuBits réaliserait ainsi $8 = 2^3$ calculs en parallèle. Ainsi, chaque ajout de QuBit augmente exponentiellement (en 2^n, où n est le nombre de QuBits du registre quantique concerné) la puissance de calcul comparé à l'équivalent registre composé de bits classiques.

Il convient en revanche de clarifier les affirmations faites précédemment. Un registre quantique de taille n fera effectivement 2^n calculs en parallèle. En revanche, avant une mesure, il nous est impossible d'exploiter cette puissance de calcul, et post-mesure, nous n'obtiendrons qu'un seul résultat. C'est pour cela que la puissance quantique ne s'applique pas à tout type de problème, et qu'il est si dur de créer de nouveaux algorithme quantiques : en plus d'obéir à une nouvelle forme de logique, il est nécessaire de créer un algorithme utilisant cette puissance de calcul en parallèle, mais dont le résultat final ne nécessite qu'une seule valeur [2].

1.3 Portes logiques

1.3.1 évolution dans le temps d'un QuBit

Les relations établies dans les parties précédentes ont été écrites dans un espace $\mathbb{H}^2$ concernant un unique QuBit, puis généralisées à un ensemble de deux QuBit. Le passage à un nombre supérieur de QuBit s'effectuera exactement de la même manière, avec le produit tensoriel, défini de manière légèrement plus généralisée, pour pouvoir s'appliquer à deux espaces $\mathbb{H}_1$ et $\mathbb{H}_2$ de dimension quelconque.

Nous pouvons donc désormais considérer un état quantique appartenant à un espace $\mathbb{H}^{2^n}$, c'est-à-dire un registre composé de n QuBits. Notons cet espace plus simplement $\mathbb{H}^{\otimes n}$. Il est alors possible de donner l'équation de Schrödinger[2] , qui décrit l'évolution dans le temps de tout état quantique précédant une mesure par :

$$\frac{d\Psi(t)}{dt} = -iH\Psi(t) \tag{1.20}$$

Où H est l'opérateur Hamiltonien. Dans ce cas particulier de la résolution de l'équation de Schrödinger, nous avons tout simplement une solution de la forme $\Psi(t) = U(t)\Psi(0)$, où $U(t) = e^{-itH}$. Notons que cette évolution temporelle ne modifie pas la norme de l'état quantique Ψ.

[2] Dans la dernière section sont rapidement présentés quelques algorithmes quantiques réputés.

1.3.2 algèbre des portes logiques

Faire fonctionner un ordinateur classique requiert de savoir créer des bits (de nos jours, il s'agit de transistors de quelques dizaines de nanomètres), mais également de pouvoir utiliser des opérations sur ces derniers. Ainsi, avec des portes NOT, AND, OR, XOR etc, il est possible de fabriquer toutes les opérations indispensables à un algèbre de base, donnnant ainsi à l'ordinateur une grande puissance de calcul.

L'analogie avec l'ordinateur quantique est parfaitement justifiée : il nous faut créer des portes logiques quantiques pouvant agir sur des QuBits afin de pouvoir faire d'un ordinateur quantique une réalité. De manière générale, une porte logique est une application de la forme :

$$L(...) : \begin{cases} \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times ... & \to \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times \mathbb{H}^{\otimes n} \times ... \\ (|\boldsymbol{\varphi}\rangle, |\boldsymbol{\Psi}\rangle, |\boldsymbol{\Omega}\rangle, ...) & \mapsto (L(\varphi), L(\Psi), L(\Omega), ...) \end{cases} \tag{1.21}$$

C'est donc simplement une application linéaire qui a un nombre donné de QuBit tous de même dimension, renvoi un même nombre de QuBits dans un état potentiellement différent. Une porte quantique n'effectuant pas de mesure, il est nécessaire que la transformation maintienne la condition de normalité. Au final, pour représenter une porte logique, il suffira d'utiliser une matrice M de norme 1. L'état de chaque QuBit après passage dans L sera ainsi facilement calculé par $L(\Psi) = M \cdot \Psi$. Par définition, il apparait que le calcul quantique est spontanément réversible. On en déduit ainsi que toute matrice définissant une porte logique quantique sera inversible (donc de déterminant non nul).

1.3.3 portes logiques à 1 QuBit

Nous allons maintenant nous intéresser à plusieurs portes quantiques classiques*[5]* , en commençant bien sur par le niveau le plus bas : les portes agissant sur un registre d'un seul QuBit. Nous allons définir deux porte quantiques, phase L_Φ et rotation L_α, à partir desquelles il est possible de reconstruire toutes les portes de "dimension 1". Notons $Mat()$ l'application qui à une porte quantique associe l'unique matrice la représentant (on se positionne dans la base orthonormée $\{|\mathbf{0}\rangle, |\mathbf{1}\rangle\}$). Alors :

$$Mat(L_\Phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\Phi} \end{pmatrix} \tag{1.22}$$

$$Mat(L_\alpha) = \begin{pmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{1.23}$$

La première porte ajoute un vecteur de phase à la composante en $|\mathbf{1}\rangle$, tandis que la seconde effectue une rotation de α du QuBit. Ces transformations rappellent exactement la définition d'un point sur la sphère de Bloch (equation 1.13), ce qui nous permet d'affirmer qu'il est effectivement possible d'aboutir à n'importe quelle transformation à partir de ces deux portes logiques. On peux ainsi définir la classique porte NOT (négation du QuBit) et la porte de Hadamard :

$$Mat(NOT) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{1.24}$$

$$Mat(HADAMARD) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{1.25}$$

Si l'interprétation de la porte NOT est immédiate, il convient de revenir rapidement sur la porte de $HADAMARD$. C'est une porte indispensable en programmation quantique : en effet, elle sert souvent de première étape lors d'un algorithme, pour faire passer un registre d'un état initial $|\mathbf{0}\rangle_i$ pour chacun de ses QuBit à un état final de superposition des 2^n états $|\mathbf{0}\rangle_i$ et $|\mathbf{1}\rangle_i$ de norme toutes égales.

1.3.4 portes logiques à 2 QuBit

Certaines portes logiques à deux QuBit suivent une certaine logique : l'un des deux QuBit est appelé QuBit contrôle (d'où $CNOT = Controlled\ NOT$), et l'autre QuBit cible (usuellement, nous les "positionnons" dans cet ordre). L'idée est alors de vérifier l'état du QuBit de contrôle, et d'agir -ou non- sur le QuBit cible en fonction da la valeur du QuBit de contrôle.

Nous pouvons alors très facilement définir deux premières portes quantiques sur deux QuBits : la porte $CNOT$ qui appliquera ou non la transformation NOT sur le QuBit cible, et la porte $CPHASE$ qui appliquera potentiellement une phase au QuBit ciblé. Enfin, la porte $SWAP$ va simplement échanger les composantes $|\mathbf{01}\rangle$ et $|\mathbf{10}\rangle$. Voici leur matrices :

$$Mat(CNOT) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{1.26}$$

$$Mat(CPHASE) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\Phi} \end{pmatrix} \tag{1.27}$$

$$Mat(SWAP) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.28}$$

1.3.5 Ensemble universel

Un ensemble universel[5] est un ensemble de portes quantiques tel que tout calcul quantique soit décomposable en opération composées de ces différentes portes. Les ensembles $\{PHASE, ROTATION, CPHASE\}$ et $\{PHASE, ROTATION, CNOT\}$ sont de tels ensembles universels. En effet, avec les deux premières portes, toute opération sur un unique QuBit est réalisable, tandis qu'avec la porte $CPHASE$ ou $CNOT$, une intrication est possible, ce qui permet de "descendre en dimension" et donc assurer que tout calcul est réalisable.