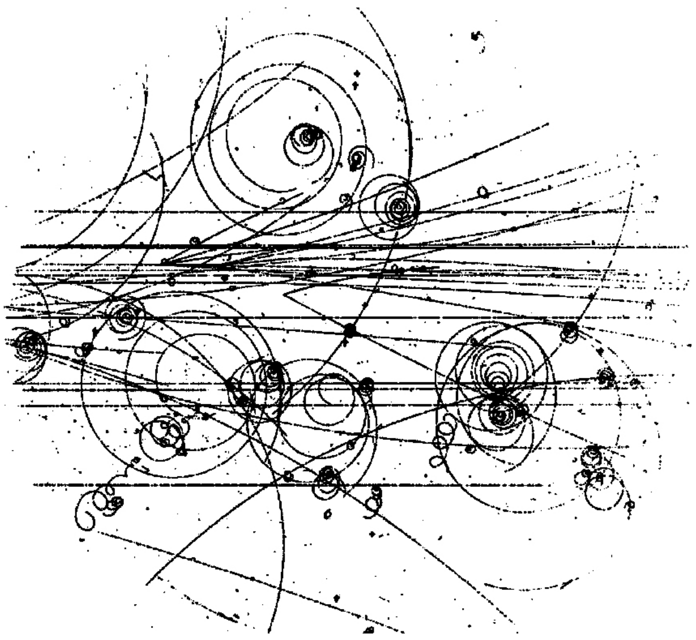
DAVIDE CAMINO

QUANTUM COMPUTING FOR LOGICAL INFERENCE

Subtitle



CONTENTS

Part I

THEORETICAL BASES

1 QUANTUM MECHANICS

In this chapter we will explore the basics of quantum mechanics in order to understand what we can or cannot do with a quantum computer. The reference architecture for this work is the quantum gate-based quantum computer. In the next pages we will try to justify why the algorithms that run on this hardware need to be reversible.

The chapter starts from the very beginning with the definition of a quantum system and presents the basis to understand the evolution of a quantum system, trying to justify why every evolution needs to be reversible and what exactly reversibility means. At the end of the chapter we will put all of our new knowledge together to derive the famous Schrödinger equation.

With all this work we will be able to imagine the function of a quantum gate and understand the limitations that are imposed when we develop an algorithm for a quantum computer.

1.1 EXPERIMENTS

We start our introduction to quantum mechanics with an experiment. Experiments are not only an excuse to introduce the topic, but the essential key of physics, both classical and modern.

Theory and models need to adapt to the experiments, and when the experimental results are in contradiction with the actual model it means that the model needs to be changed to respect the behavior of the world.

1.1.1 Spin

We analyze the experiment of an electron in a magnetic field. An electron is an electrically charged particle; when some electrons are shot in an electric field all of them are influenced by Coulomb's law; if all electrons have the same initial velocity the beam of electrons remains intact.

What happens to the same beam in a magnetic field? Again electrons are deflected by a force, but this time the beam splits. If the initial velocity was parallel to the x axis and the magnetic field is oriented along the z axis some electrons are deflected upward, some downward, but the intensity of the deflection is the same for all the electrons. This means that no electrons are deflected less or more than the others and the beam splits exactly into two parts.

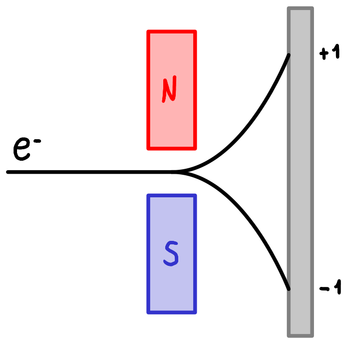


Figure 1.1: Experiment's schema

Starting from this experiment we can make a measuring instrument: this apparatus $\mathcal{A}$ can be oriented along an arbitrary axis, and in the previous configuration $\mathcal{A}$ displays $+1$ if the electron is deflected upward, -1 otherwise. We call this number the spin of the electron.

Repeatability of measure

If we measure the spin of an electron and $\mathcal{A}$ displays $+1$, we can confirm the experiment's results by measuring the spin again and we obtain spin $+1$ every time. This means that the measurements are repeatable (an essential property to construct models and make predictions). We can think, and it will be clear later why it is useful, that the first experiment prepares the spin $+1$ and the others confirm this result.

Spin is a quantum property and all the visual representations such as the rotation of the electron around its axis would lead to misrepresentation. Spin and rotation, however, have some similarities. Let's analyze what would happen if we consider a charged sphere in a magnetic field with the laws of classical physics. We consider a sphere rotating around its axis, and this axis is parallel to the z axis. The x or y component of the angular momentum is zero. Measuring the component along a generic axis, oriented like the versor $\hat{n}$[1], we would obtain a result proportional to the projection of $\hat{z}$ on $\hat{n}$. This projection can be found with the scalar product $\hat{z} \cdot \hat{n} = \cos\theta$[2], where θ is the angle between the axes.

Now we consider the quantum version of this phenomenon. Let's start by measuring the z component of the spin and assume that the result is $\sigma_z = +1$; if we rotate the apparatus $\mathcal{A}$ around, for example, the x axis, we can measure σ_x. This component would not be zero, and $\mathcal{A}$ keeps displaying only $+1$ or -1. The single result is not helpful, but we can repeat the experiment, namely:

1. orienting $\mathcal{A}$ along the z axis and preparing a spin $\sigma_z = +1$
2. rotating $\mathcal{A}$ around x
3. measuring the x component of the spin

statistically we would observe the same number of $\sigma_x = +1$ and $\sigma_x = -1$.

If we start the experiments with a spin prepared as $\sigma_z = +1$ and then orient $\mathcal{A}$ along a generic axis $\hat{n}$ each measure would be binary and unpredictable, but the mean of the measures tends to $\hat{z} \cdot \hat{n} = \cos\theta$ where θ is the angle between $\hat{z}$ and $\hat{n}$. In the most general case we can start with the apparatus oriented like m and prepare the spin $\sigma_m = +1$, then we rotate $\mathcal{A}$ around $\hat{n}$ without interfering with the spin and measure again; we would obtain the statistical result $\langle \sigma_n \rangle = \hat{m} \cdot \hat{n}$[3].

The result of a single measure is non deterministic, but we can make predictions over the mean values of the measures: the expected values behave as the single results of the classic experiment.

Invasive experiments

Considering now a sequence of three measures: starting with $\mathcal{A}$ oriented along z we prepare the spin $\sigma_z = +1$, then we rotate $\mathcal{A}$ to measure σ_x obtaining, let's say, $+1$ (the reasoning is the same if we obtain -1); lastly returning with $\mathcal{A}$ parallel to z we cannot make any prediction on the single

1 A versor is a vector of magnitude 1 (unit vector), it is normally used to specify a direction.
2 We can use directly the angle because we are considering versors.
3 The Dirac bracket $\langle\ \rangle$ denotes the statistical mean of a quantity. We call that expectation value.

result, the initial configuration (with $\sigma_z = +1$) is lost forever, the only result we can predict is that $\langle \sigma_z \rangle = 0$.

1.1.2 Qubit

We have introduced the spin referring to electrons in a magnetic field. However, we can study the spin without examining the associated electron; we have isolated a simple physical system, the simplest we can study.

Spin belongs to a class of simple physical systems called *qubit*; in all of these systems the result of a measure is binary. We will see that, even if the result of a measure is equal to the classical *bit*, the qubit system is described in a very different way compared to its classical alter ego.

1.1.3 Boolean Logic

In this paragraph we try to understand why we need two different ways to describe a classical and a quantum state space. To do so we analyze the results of some logical propositions, both basic and composed via logical connectives.

Starting with the classical case we consider a bag of colored and numbered balls. We can construct the state space by enumerating all states, namely taking each ball from the bag and annotating the pair number–color. The basic propositions we analyze are:

- The extracted ball is red.
- The number on the extracted ball is even.

If we consider a particular state we can say if a proposition is true or false; we can also define two subsets of balls, the first with all the red balls (for this subset the first proposition is true), the second one with the balls that show an even number (subset that makes the second proposition true). Considering now disjunction and conjunction:

- The extracted ball is red *or* even.
- The extracted ball is red *and* even.

Again it is simple to associate a truth value to these propositions if we consider a single state; also we can construct two subsets that satisfy the propositions from the subsets we defined before: the new subsets are respectively the union and intersection of the old ones.

In the quantum world the situation is very different. Let's start from propositions that can be verified with a simple experiment:

- The z component of the spin is $+1$.
- The x component of the spin is $+1$.

If we want to check the first proposition we can orient the apparatus $\mathcal{A}$ along z and make a measurement; the same procedure can be followed for the second proposition. The disjunction and conjunction of these propositions are:

- The z component of the spin is $+1$ *or* the x component is $+1$.
- The z component of the spin is $+1$ *and* the x component is $+1$.

Starting with the disjunction. Considering a state prepared, without our knowledge, with $\sigma_z = +1$. If our first measure is along the z axis, $\mathcal{A}$ will always display $+1$ and we can immediately conclude that the proposition is true. If we start measuring the x component, we have a 50% chance that $\mathcal{A}$ displays $+1$ or -1; also this measurement destroys the initial state and the measure of σ_z becomes non predictable. In this scenario we have a 25% chance of deducing that the proposition is false; Figure 1.2 shows all the possible measurement results in this case. The logical value of a proposition depends on the order in which we perform the measurements.

The disjunction is not commutative

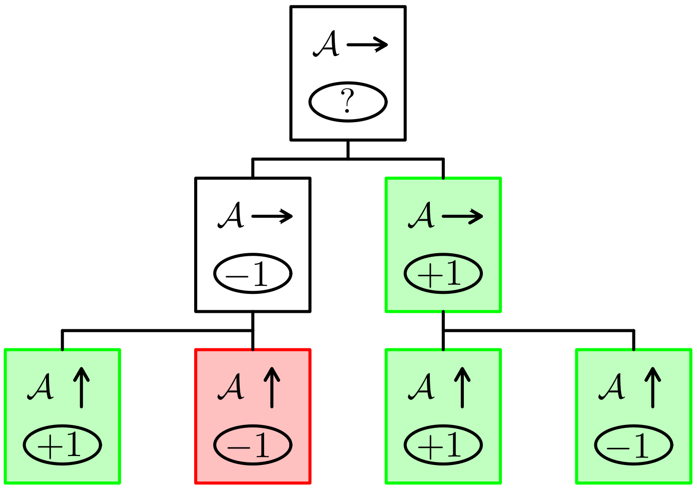


Figure 1.2: The apparatus $\mathcal{A}$ is represented as a box, the arrow represents the direction along which the apparatus is oriented, the display (ellipse) shows the result of measurement. We have highlighted in green the cases in witch we can immediately conclude that the disjunction of the propositions is true

The conjunction is even worse: no matter the order of the measurements, the second one destroys the result of the first. The disjunction is true if at least one of the sub-propositions is true, and if we find a spin component that is $+1$ we can always confirm this result with another measurement. In the conjunction the two sub-propositions must be true *at the same time*, but with the second measurement we lose all the knowledge of the first one. We can never conclude that the conjunction is true.

The conjunction loses its meaning

1.2 QUANTUM STATES

In the previous section we have understood that a state space of a quantum system cannot be represented in the same way as a classical state space. Now we present a formal mathematical model to describe the state space for spin.

Axiom 1. *The state space for a quantum system is a complex vector space.*

This is a physical axiom, which means that it is true because there are a lot of experiments that confirm this model and none that shows a contradiction.

1.2.1 Vector Spaces

A vector space is a mathematical and abstract construction that can have multiple dimensions (even infinite) and has, as components, integers, real or complex numbers, or other elements. An example that shows well how abstract a vector space can be is the complex-valued continuous function of variable x; the set of these functions generates a vector space.

Hilbert space

In quantum mechanics the state space is described by a vector space having as element $|A\rangle$ called *ket*. The properties of this space are:

- the sum of two kets is a ket;
- addition is commutative;
- addition is associative;
- existence of identity element for addition;
- existence of inverse elements for addition;
- existence of identity element for scalar multiplication;
- linearity property.

1.2.2 Bra and Ket

An example of ket that we will find often is the column vector of two dimensions:

$$|A\rangle = \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}$$

where α_1 and α_2 are complex numbers. With this simple example of ket it is easy to verify the validity of all previously described properties.

If, for complex numbers, exists the complex conjugate, for every ket there exists a *bra*. The set of bra generates a dual conjugate space with respect to the state space of ket. We denote a bra as $\langle A|$. If $|A\rangle$ is the ket of the previous example the corresponding bra is a row vector having as elements the complex conjugate of $|A\rangle$:

$$\langle A| = (\alpha_1^*, \alpha_2^*)\,.$$

Inner product

Name and symbol associated with elements of Hilbert spaces become clear when we define the product *bra-ket*, this is the corresponding scalar product of an ordinary vector and is called inner product. Considering bra and ket of two dimensions we can evaluate the inner product by adding the products of corresponding components:

$$\langle A\,|\,B\rangle = (\alpha_1^*, \alpha_2^*) \cdot \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} = \alpha_1^*\beta_1 + \alpha_2^*\beta_2.$$

Having the inner product we can define:

VERSOR normalized vector $|A\rangle$ in which $\langle A\,|\,A\rangle = 1$;

ORTHOGONAL VECTOR vectors that have a null inner product: $\langle A\,|\,B\rangle = 0$.

We are familiar with these concepts in two and three dimensions, the first one is a vector of length one, the second is the right angle between two vectors. This representation is misleading in our case, we cannot imagine a ket like an arrow and the state space is completely abstract even if there are properties and operations in common between this space and the 3D space that we are familiar with.

We have lost the geometric interpretation, and it seems that we have defined two completely abstract and useless concepts, we will see next that these are key concepts in the description of quantum systems and have a precise and important physical meaning.

Orthonormal basis

By having a vector space is possible to build a set of orthogonal versors that generates all vectors in the given space. This set is called orthonormal basis and the cardinality of the set is equal to the dimension of the space.

Formally having a basis $\mathscr{B} = \{ |i_1\rangle, |i_2\rangle, \dots, |i_N\rangle \}$ of a space with N dimensions, we can write a generic vector in that space as

$$|A\rangle = \sum_{n=1}^{N} \alpha_n |i_n\rangle = \sum_{n=1}^{N} |i_n\rangle \langle i_n | A\rangle \tag{1.1}$$

this is the linear combination of the basis versors; where kets $|i_n\rangle$ are the versors in the basis and α_n are the vector components. We can obtain those components with the inner product between the vector $|A\rangle$ and the basis versors:

$$\alpha_i = \langle i | A\rangle \,. \tag{1.2}$$

1.2.3 Hidden variables

In a classical system we can measure all the variables associated to a physical system and then make a deterministic prediction of the evolution of that system. From the experiments described in the first section we have learned that a quantum system is not completely predictable even if we can make all the measurements that we want[4]. We can ask ourselves if our measurements aren't enough, if there are other variables that can make the prediction completely deterministic. About that topic we don't have any experimental proof, the opinion of physicists is divided in two main visions:

OPINION ONE : there are hidden variables and, if we manage to measure them, the prediction of results become deterministic. These variables can be

- very difficult to measure
- unknowable to us because also we are constituted by quantum material.

OPINION TWO : hidden variables don't exist, we already know all the information about a given system and quantum mechanics is intrinsically non deterministic.

No hidden variables

Probably no experiment could determine which vision is correct, but this doubt doesn't worsen our comprehension of the physical world. We can

4 We remember that a measure along one axis destroys our knowledge about the result along another axis.

simply choose one vision and build our model coherently. We choose the simpler one, without hidden variables, all that we have to model are the quantities that we can measure and the measurements allow us to know all the information about a given system.

Even if we have lost complete determinism, knowing the state of a system gives us some information about the system and the successive measurements. In the next section we will see what we can deduce about spin.

1.2.4 Spin states

Let's start enumerating all possible spin states along the coordinate axes. If we rotate the apparatus $\mathcal{A}$ around z, we can obtain $\sigma_z = \pm 1$; we call these states *up* and *down* and label them with kets $|u\rangle$ and $|d\rangle$. Orienting $\mathcal{A}$ along x, we obtain *left* $|l\rangle$ and *right* $|r\rangle$. Lastly, along the y axis, we measure the states *in* $|i\rangle$ and *out* $|o\rangle$.

Spin space states have two dimensions

The hypothesis that there aren't hidden variables allows us to represent the space state in a simple way: each spin state can be represented as a ket in a two-dimensional complex vector space.

To express a vector we need a basis; we choose $\mathscr{B} = \{\, |u\rangle \,, |d\rangle \,\}$[5] and try to obtain all states as a linear combination (*superposition*) of the basis vectors. A generic state $|A\rangle$ can be expressed as:

$$|A\rangle = \alpha_u \,|u\rangle + \alpha_d \,|d\rangle$$

where α_u and α_d are the components of $|A\rangle$ along $|u\rangle$ and $|d\rangle$, and can be obtained by projection: $\alpha_u = \langle u \,|\, A\rangle$ and $\alpha_d = \langle d \,|\, A\rangle$ (as in Equation 1.2).

Probability amplitudes

$|A\rangle$ components are complex numbers and their physical meaning is: having a spin prepared in the state $|A\rangle = \alpha_u \,|u\rangle + \alpha_d \,|d\rangle$[6]; $\alpha_u^* \alpha_u$ is the probability of measuring $\sigma_z = +1$, while $\alpha_d^* \alpha_d$ is the probability that a measurement of σ_z will yield -1. Formally we can denote the probability of measuring $+1$ and -1 as P_u and P_d respectively and write:

$$\begin{aligned} P_u &= \langle A \,|\, u\rangle \langle u \,|\, A\rangle \\ P_d &= \langle A \,|\, d\rangle \langle d \,|\, A\rangle \,. \end{aligned} \tag{1.3}$$

Components α_u and α_d are called probability amplitudes, and their physical meaning is given by the square of the magnitude. This is the actual probability, and we want the sum of all probabilities to be one. This is equivalent to requiring that $|A\rangle$ is normalized: $\langle A \,|\, A\rangle = 1$.

Now we will show why $|u\rangle$ and $|d\rangle$ have to be orthogonal:

$$\begin{aligned} \langle u \,|\, d\rangle &= 0 \\ \langle d \,|\, u\rangle &= 0. \end{aligned}$$

We try to give an idea with a *reductio ad absurdum*: if $|u\rangle$ and $|d\rangle$ were not orthogonal, the projection of one on the other would not be null. This means

5 We will show that these vectors are in fact orthogonal and why they need to be.

6 From now on we use "prepared" or "measured" as synonyms: every measurement is invasive and can change the spin state, so no matter what was the previous state, after a measurement the state is the one we have measured.

that if we orient $\mathcal{A}$ along z and measure $\sigma_z = +1 = |u\rangle$, we would have $\alpha_d = \langle d\,|\,u\rangle \neq 0$, which is a contradiction to experimental results. If $\alpha_d \neq 0$, then $\alpha_d^*\alpha_d > 0$; we started with a state prepared as $\sigma_z = +1$ and ended with a nonzero probability of measuring $\sigma_z = -1$: this is absurd.

Orthogonal states are mutually exclusive

We can extend the reasoning to a general and key concept of quantum mechanics: two orthogonal states are distinct and mutually exclusive. If the system is in the first state, the probability of finding it in the second is zero.

Now we are ready to express spin states as linear combinations of the basis vectors $\mathscr{B} = \{\,|u\rangle\,,|d\rangle\,\}$. The representation of the basis vectors themselves is naturally easy:

$$|u\rangle = \begin{pmatrix}1\\0\end{pmatrix} \tag{1.4}$$

$$|d\rangle = \begin{pmatrix}0\\1\end{pmatrix}. \tag{1.5}$$

To construct vector *right*, let's consider a spin prepared in the state $|r\rangle$. If we measure σ_z, we have a 50% chance of obtaining +1 (and 50% for −1); this means that for $|r\rangle$ we have $\alpha_u^*\alpha_u = \alpha_d^*\alpha_d = 1/2$. A vector that satisfies this constraint is:

$$|r\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{pmatrix}. \tag{1.6}$$

The reasoning is the same for state *left*; we also add the constraint that a state *left* cannot be *right* and vice versa: $\langle r\,|\,l\rangle = \langle l\,|\,r\rangle = 0$. We can express *left* as:

$$|l\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}}\\-\frac{1}{\sqrt{2}}\end{pmatrix}. \tag{1.7}$$

Lastly, the constraints to find explicit forms for *in* and *out* are:

- states must be orthogonal: $\langle i\,|\,o\rangle = \langle o\,|\,i\rangle = 0$;
- if we have a spin prepared as *in* or *out*:
  - equiprobability of measuring $\sigma_z = +1$ and $\sigma_z = -1$;
  - equiprobability of measuring $\sigma_x = +1$ and $\sigma_x = -1$.

Two vectors that satisfy these constraints are:

$$|i\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}}\\\frac{i}{\sqrt{2}}\end{pmatrix} \tag{1.8}$$

$$|o\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}}\\-\frac{i}{\sqrt{2}}\end{pmatrix}. \tag{1.9}$$

This last derivation shows why it is important that the state space is complex: if we only accepted real components for our vectors, the system of equations we have implicitly defined would not have any solution[7].

7 To avoid confusion, we point out that $|i\rangle$ is the ket of state *in*. i, instead, is the imaginary unit.

1.3 OBSERVABLES

We have learned that in classical mechanics we can trust our intuition, and we can do one or more measurements to know exactly the state of a system: a measurement does not perturb the state, which is the same before, during, and after the measurement.

In quantum mechanics the situation is more complex; our intuition is misleading, and we need mathematical tools to describe what we can measure: the observables. These tools are mathematical operators called *machines* (**M**) and have as both input and output state vectors.

Axiom 2. *Machines associated with observables are described by linear operators.*

We will show that machines are Hermitian operators, so let's start defining these operators and describing their properties[8].

1.3.1 Hermitian operator

Formally, machines modify a state vector in this way:

$$\mathbf{M}\left|A\right\rangle = \left|B\right\rangle$$

The linearity of machines implies that:

$$\mathbf{M}\left|A\right\rangle = \left|B\right\rangle \Rightarrow \mathbf{M}z\left|A\right\rangle = z\left|B\right\rangle$$

and:

$$\mathbf{M}(\left|A\right\rangle + \left|B\right\rangle) = \mathbf{M}\left|A\right\rangle + \mathbf{M}\left|B\right\rangle .$$

If we choose a basis to represent machines and state vectors, we can write explicitly the linear operator as an $N \times N$ matrix, where N is the dimension of the vector space of the state vectors. A generic machine that transforms spins can be expressed as:

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} .$$

When we fix a basis, we are forced to express all state vectors and operators in that basis, but now we have a set of rules to define the application of the operator to a state vector, i.e. the matrix multiplication:

$$\mathbf{M}\left|A\right\rangle = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \times \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} = \left|B\right\rangle$$

Eigenvalues and eigenvectors

When we consider a linear operator, we can search for eigenvalues and eigenvectors (if they exist). Eigenvectors are vectors that don't change their direction when multiplied by the operator; their magnitude is scaled by a constant factor called the eigenvalue. Formally:

$$\mathbf{M}\left|\lambda\right\rangle = \lambda\left|\lambda\right\rangle$$

where $\left|\lambda\right\rangle$ is the eigenvector and λ the eigenvalue.

8 The reason why we need this kind of operator will be clear in Section 1.3.2.

Considering the transformation between ket $|A\rangle$ and $|B\rangle$: $\mathbf{M}|A\rangle = |B\rangle$, taking into account the dual space of bras and searching for a machine that transforms the bra $\langle A|$ into $\langle B|$, we cannot simply use the matrix having as elements the complex conjugate of $\mathbf{M}$; the correct operator is the *Hermitian conjugate* of $\mathbf{M}$, which is the transpose of the matrix having as elements the complex conjugates of $\mathbf{M}$. We denote the Hermitian conjugate with the dagger †:

$$\mathbf{M}^\dagger = [\mathbf{M}^*]^T = [\mathbf{M}^T]^*.$$

We can now write:

$$\mathbf{M}|A\rangle = |B\rangle \Rightarrow \langle A|\mathbf{M}^\dagger = \langle B|\,.$$

An operator that is equal to its Hermitian conjugate is called a *Hermitian operator*. Formally, $\mathbf{M}$ is Hermitian if and only if

$$\mathbf{M} = \mathbf{M}^\dagger.$$

Hermitian operators have some important properties:

- all eigenvalues are real;
- eigenvectors form a *complete set*: all vectors obtained with the application of the operator can be expressed as a linear combination of eigenvectors;
- if λ_1 and λ_2 are different eigenvalues, the associated eigenvectors are orthogonal;
- if two eigenvalues are equal (*degeneracy*), it is always possible to find two associated eigenvectors that are orthogonal.

Fundamental theorem

The last three properties can be summed up in the following way:

Theorem 1. *The eigenvectors of a Hermitian operator form an orthonormal basis.*

1.3.2 Principles of quantum mechanics

Let's introduce the first four principles of quantum mechanics, the ones about observables[9].

Principles 1. *Observables in quantum mechanics are described by linear operators* $\mathbf{L}$.

$\mathbf{L}$ must also be a Hermitian operator: we can consider this proposition an axiom itself or deduce it from the other principles.

Principles 2. *The results of a measurement can only be the eigenvalues associated with the observable operator.*

Calling λ_i a generic eigenvalue and $|\lambda_i\rangle$ the associated eigenvector, if the system is in the *eigenstate* $|\lambda_i\rangle$, the measurement always returns λ_i. Since all λ_i must be physical quantities they must be real, a peculiar property of Hermitian operators.

9 The fifth, and last one, concerns the temporal evolution. It will be discussed later on (Section 1.4).

Principles 3. *Unambiguously distinguishable states are represented by orthogonal vectors.*

Distinguishable states can be separated without ambiguity by a measurement. For example, if we want to distinguish between $|u\rangle$ and $|d\rangle$, we measure σ_z: *up* and *down* are distinct. We cannot, instead, say if a certain system is in state *up* or *right*, because even if the system is in the state $|u\rangle$ we can still measure σ_x and find (with 50% chance) that the system is in state $|r\rangle$.

Overlap

The inner product is a measure of how much two states are indistinguishable; for that reason it is also called overlap. Two states are physically distinct if the overlap is zero.

$$\langle u | d \rangle = 0$$
$$\langle u | r \rangle \neq 0$$

Principles 4. *If the system is in state $|A\rangle$ and we measure the observable* **L**, *the probability of obtaining λ_i is:*

$$P(\lambda_i) = \langle A | \lambda_i \rangle \langle \lambda_i | A \rangle \,.$$

where λ_i is a generic eigenvalue of **L** and $\langle \lambda_i|$, $|\lambda_i\rangle$ are the bra and ket associated with that eigenvalue (eigenvector of λ_i).

1.3.3 Spin Operator

The principles tell us what properties a machine must have to represent an observable. Let's construct the spin operator σ.

Until now, we have measured spins with the apparatus $\mathcal{A}$, orienting $\mathcal{A}$ along the component of our interest. σ is a mathematical tool that allows us to make predictions about the result of a measurement with $\mathcal{A}$ (fourth principle); as we can rotate $\mathcal{A}$, we must also rotate σ (mathematically). For this spatial property, σ is called a *3-vector operator*.

OPERATOR σ_z: Let's start with the simplest operator[10]. The second principle says that all eigenvectors of σ_z are $|u\rangle$ and $|d\rangle$, with associated eigenvalues $+1$ and -1. We can write this assertion as equations:

$$\sigma_z |u\rangle = |u\rangle$$
$$\sigma_z |d\rangle = -|d\rangle \,.$$

In matrix form:

$$\begin{pmatrix} (\sigma_z)_{11} & (\sigma_z)_{12} \\ (\sigma_z)_{21} & (\sigma_z)_{22} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$\begin{pmatrix} (\sigma_z)_{11} & (\sigma_z)_{12} \\ (\sigma_z)_{21} & (\sigma_z)_{22} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = -\begin{pmatrix} 0 \\ 1 \end{pmatrix} \,.$$

The solution of this system is[11]:

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \,.$$

10 This is because we have chosen $\mathscr{B} = \{\, |u\rangle \,, |d\rangle \,\}$ as the basis.
11 It is easy to verify that this operator is also linear.

OPERATOR σ_x: With the same reasoning, we can construct the operator along the x axis. We have already deduced the representations of *right* and *left* in Equations 1.6 and 1.7. The equations that allow us to construct σ_x are:

$$\begin{pmatrix} (\sigma_x)_{11} & (\sigma_x)_{12} \\ (\sigma_x)_{21} & (\sigma_x)_{22} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$
$$\begin{pmatrix} (\sigma_x)_{11} & (\sigma_x)_{12} \\ (\sigma_x)_{21} & (\sigma_x)_{22} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix} = - \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix}.$$

The solution of this system is:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

OPERATOR σ_y: The last direction is along the y axis. Considering the expressions for *in* and *out* given in Equations 1.8 and 1.9, and following the second principle, we can write:

$$\sigma_y |i\rangle = |i\rangle$$
$$\sigma_y |o\rangle = -|o\rangle.$$

We can rewrite this in matrix form, and the solution we would obtain is:

$$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

Pauli matrices

We have obtained a matrix representation of the three spin operators σ_z, σ_x, and σ_y:

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}. \tag{1.10}$$

These famous and important matrices are named after their inventor, Wolfgang Ernst Pauli.

1.3.4 Theory and experiments

Thanks to the operators σ_z, σ_x, and σ_y, if we know the state vector, we can statistically predict the result of a measurement of the spin along one of the three coordinate axes. What can we say about a measurement taken by orienting the apparatus $\mathcal{A}$ along a generic direction?

Considering $\mathcal{A}$ oriented along the unit vector $\hat{n}$, if σ behaves as a 3-vector, in order to obtain σ_n we only need the inner product:

$$\sigma_n = \vec{\sigma} \cdot \hat{n}$$

Expanding the components:

$$\sigma_n = \sigma_x n_x + \sigma_y n_y + \sigma_z n_z.$$

If we choose the basis $\mathscr{B} = \{|u\rangle, |d\rangle\}$, we can use the Pauli matrices to express in matrix form the expression for σ_n:

$$\sigma_n = n_x \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + n_y \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} + n_z \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} n_z & n_x - i n_y \\ n_x + i n_y & -n_z \end{pmatrix}.$$

Given a direction (expressed by the unit vector $\hat{n}$), we can construct the matrix we have now made explicit, and then, after finding eigenvalues and eigenvectors, we can know all possible results of a measurement and obtain the probability associated with each result. For example, considering a direction in the x–z plane, the operator σ_n would be:

$$\sigma_n = \begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & -\cos\theta \end{pmatrix}$$

where θ is the angle between $\hat{n}$ and z. For this matrix, the eigenvalues and eigenvectors are:

$$\lambda_1 = 1 \qquad |\lambda_1\rangle = \begin{pmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2} \end{pmatrix}$$

and

$$\lambda_2 = -1 \qquad |\lambda_2\rangle = \begin{pmatrix} -\sin\frac{\theta}{2} \\ \cos\frac{\theta}{2} \end{pmatrix}.$$

It should be pointed out that the theory is in agreement with experimental results[12]. Eigenvalues are $+1$ and -1, exactly the only results that the apparatus $\mathcal{A}$ can retrieve. The probability of obtaining a certain result can be evaluated as:

$$P(+1) = |\langle u|\lambda_1\rangle|^2 = \cos^2\frac{\theta}{2}$$
$$P(-1) = |\langle u|\lambda_2\rangle|^2 = \sin^2\frac{\theta}{2}$$

Lastly, let's calculate the average value for the measurement σ_n. From the first experiment we have seen in Section 1.1.1, we already know that the result of repeated measurements with $\mathcal{A}$ is $\cos\theta$. Let's verify if our model is coherent with the world.

Expectation value

Expected values are obtained as:

$$\langle L\rangle = \sum_i \lambda_i P(\lambda_i)$$

Specifically:

$$\langle\sigma_n\rangle = (+1)\cos^2\frac{\theta}{2} + (-1)\sin^2\frac{\theta}{2} = \cos\theta.$$

This is in complete agreement with the experimental results.

Before going on, we present, without proof, a useful theorem about expectation values:

Theorem 2. *To know the expectation value of an observable, we can simply place the operator associated with the observable between the bra and ket of the state vector:*

$$\langle \mathbf{L}\rangle = \langle A|\mathbf{L}|A\rangle \tag{1.11}$$

where $\mathbf{L}$ is an observable, $|A\rangle$ is a state vector, and $\langle A|$ is the corresponding bra.

12 If not, we must abandon this model and build another one.

1.3.5 Operator and Measure

Operators allow us to know the probability of measuring a certain spin given the direction of the measurement and the state vector. This probability is expressed by the state vector that we obtain when we apply the operator σ to the initial state.

It is important not to confuse the measurement act with the application of a machine that represents the observables. The spin state after the measurement is not the same as the one we obtain after the application of the operator. The operator is only an abstract mathematical construct that allows us to make statistical predictions about results, but doesn't have physical implications.

Let's consider an example to clarify the previous assertion. Having a spin prepared in the *up* state, its state vector is $|u\rangle = \binom{1}{0}$. If we apply the operator σ_z, we would obtain again $\binom{1}{0}$, and if we measure the spin with $\mathcal{A}$ oriented along z, it will always display $+1$, and we conclude that the state after the measurement is $|u\rangle$.

Consider now a spin prepared *right*, i.e. $|r\rangle = 1/\sqrt{2}\,|u\rangle + 1/\sqrt{2}\,|d\rangle$. Applying again the operator σ_z, the new state vector is $1/\sqrt{2}\,|u\rangle - 1/\sqrt{2}\,|d\rangle$. This vector tells us the probability of measuring $\sigma_z = +1$ (50%), but it is not the spin state after the measurement. Using the apparatus $\mathcal{A}$, we could measure:

- $+1$: the final state will be *up*;
- -1: the final state will be *down*.

No matter the result of the measurement, the final state will be different from the one we obtained by applying the operator.

1.4 TEMPORAL EVOLUTION

Let's explore the laws that describe the temporal evolution of a quantum system. In particular, we will see how the state vector can evolve over time.

1.4.1 Unitarity

In classical mechanics we are used to having a motion law that links different states of our system deterministically; this means being able to know precisely the following state given the previous one. A good law, however, doesn't allow us only to know the future, but also the past states that brought the system to the current state[13].

Reversibility

In other words, we want physical transformations to be reversible. This requirement is so important that we call this property the *minus first law*, because it underlies everything else. If we think about the system states as nodes in an oriented graph, reversibility imposes that each node has exactly one input edge and one output edge. This fundamental law is also true

13 For example, if we observe a ball in free fall touching the floor with a certain speed and at a certain time, we can know exactly when and from what height the ball started its fall.

in quantum mechanics and is called *unitarity*[14], and it assures us that no information is lost. The unitarity law can be expressed as:

Axiom 3. *If two identical isolated systems are in different states, they stay in different states, and they were in different states in the past.*

1.4.2 Time-Development Operator

Considering a system in the state $|\Psi(t)\rangle$, where the t indicates that the state vector evolves over time, quantum motion equations allow us to obtain the state at time t given the initial state:

$$|\Psi(t)\rangle = \mathbf{U}(t)\,|\Psi(0)\rangle\,. \tag{1.12}$$

Determinism

Thanks to the operator $\mathbf{U}(t)$ we can know exactly the state vector $|\Psi(t)\rangle$ at time t, given $|\Psi(0)\rangle$. This assertion can be rephrased as:

Axiom 4. *The temporal evolution of the state vector is deterministic.*

Quantum mechanics is still non-deterministic, because knowing the state vector doesn't mean knowing the result of a measurement.

In order for $\mathbf{U}(t)$ to behave as we want, it has to:

- be a linear operator;
- respect reversibility.

The second constraint allows us to define the mathematical properties of $\mathbf{U}(t)$. Considering two initially different states $|\Psi(0)\rangle$ and $|\Phi(0)\rangle$, since there exists an experiment capable of certainly distinguishing the states, $|\Psi(0)\rangle$ and $|\Phi(0)\rangle$ must be orthogonal:

$$\langle\Psi(0)\,|\,\Phi(0)\rangle = 0.$$

Conservation of Distinctions

The minus first law assures that during the entire temporal evolution of the two systems, the state vectors $|\Psi(t)\rangle$ and $|\Phi(t)\rangle$ will continue to be distinguishable (orthogonal):

$$\langle\Psi(t)\,|\,\Phi(t)\rangle = 0 \qquad\qquad \forall t \geqslant 0.$$

If we rewrite this equation using Formula 1.12, we obtain:

$$\langle\Psi(0)|\,\mathbf{U}^{\dagger}(t)\mathbf{U}(t)\,|\Phi(0)\rangle = 0.$$

From this we can see that $\mathbf{U}^{\dagger}(t)\mathbf{U}(t)$ must behave as the identity operator, that is:

$$\mathbf{U}^{\dagger}(t)\mathbf{U}(t) = \mathbf{I}. \tag{1.13}$$

An operator that behaves as $\mathbf{U}$ is *unitary*.

Principles 5. *The temporal evolution of state vectors is unitary.*

From the unitarity of $\mathbf{U}$ descends the *conservation of overlaps*: the overlap between two states (their inner product), subjected to the same temporal-development operator, is preserved over time.

14 We will see in the next paragraph the reason for this name

1.4.3 The Hamiltonian

Often, in classical physics, a motion law is the result of a differential equation where we have exchanged a finite time interval with an infinite number of infinitesimal intervals.

Continuity

In quantum mechanics we can follow the same path and consider time intervals ϵ close to zero. In this scenario, after an ϵ amount of time, the state vector will change slightly and "smoothly", and the operator $\mathbf{U}(\epsilon)$ will be very similar to the identity. We can rewrite $\mathbf{U}(\epsilon)$ in order to highlight the difference with the identity $\mathbf{I}$ as:

$$\mathbf{U}(\epsilon) = \mathbf{I} - i\epsilon\mathbf{H}. \tag{1.14}$$

For now, i is a mere scale factor that later will help us recognize in $\mathbf{H}$ the quantum version of the classical Hamiltonian.

We can now express the infinitesimal evolution of a quantum system by combining Equations 1.12 and 1.14:

$$|\Psi(\epsilon)\rangle = |\Psi(0)\rangle - i\epsilon\mathbf{H}\,|\Psi(0)\rangle\,.$$

Bringing to the left the time interval:

$$\frac{|\Psi(\epsilon)\rangle - |\Psi(0)\rangle}{\epsilon} = -i\mathbf{H}\,|\Psi(0)\rangle\,.$$

Now considering the limit for $\epsilon \to 0$, we can see in the left member the time derivative of the state vector:

$$\frac{\partial\,|\Psi(t)\rangle}{\partial t} = -i\mathbf{H}\,|\Psi(0)\rangle\,.$$

Before using $\mathbf{H}$ as the quantum Hamiltonian, we have to verify the dimensional correctness. As in classical mechanics, the Hamiltonian is the mathematical construct that represents the energy. In our formula, however, ignoring the state vector, we have the inverse of time on the left and the energy on the right. To resolve this problem, let's introduce an important physical constant: the reduced Planck constant, $\hbar$.

The equation becomes:

Time-dependent Schrödinger equation

$$\hbar\frac{\partial\,|\Psi\rangle}{\partial t} = -i\mathbf{H}\,|\Psi\rangle \qquad \text{or} \qquad \frac{\partial\,|\Psi\rangle}{\partial t} = \frac{-i\mathbf{H}\,|\Psi\rangle}{\hbar}. \tag{1.15}$$

The constant $\hbar$ has units of $\mathrm{kg}\cdot\mathrm{m}^2/\mathrm{s}$ and resolves the incompatibility between the two members. This equation is fundamental and is called the *generalized Schrödinger equation*, or time-dependent Schrödinger equation. If we know the Hamiltonian of an undisturbed system, we can know the evolution of the state vector.

If $\mathbf{H}$ represents the energy of the system, we should be able to measure it, so $\mathbf{H}$ has to be an observable. If $\mathbf{H}$ is an observable, it must be a Hermitian operator; let's verify it. Starting from Equation 1.13 and substituting $\mathbf{U}$ with Expression 1.14, we obtain:

$$(\mathbf{I} + i\epsilon\mathbf{H}^\dagger)(\mathbf{I} - i\epsilon\mathbf{H}) = \mathbf{I}.$$

Expanding to first order in ϵ, we find:

$$\mathbf{H}^\dagger - \mathbf{H} = 0 \Rightarrow \mathbf{H}^\dagger = \mathbf{H}.$$

We have concluded that **H** is an Hermitian operator that represents an observable: the energy of the system. Eigenvalues of **H** are the results of all possible direct measurements of the energy of the system.

Quantum Hamiltonian

1.4.4 Commutators

In a system that evolves with time, we expect that the expectation values for a certain observable **L** will also change. Thanks to equation 1.11 on page 15, we can write explicitly the time dependence of expectation values:

$$\langle \mathbf{L} \rangle = \langle \Psi(t) \,|\, \mathbf{L} \,|\, \Psi(t) \rangle\,.$$

The time derivative[15] is:

$$\frac{d}{dt}\langle \Psi(t) \,|\, \mathbf{L} \,|\, \Psi(t) \rangle = \left\langle \dot{\Psi}(t) \,\middle|\, \mathbf{L} \,\middle|\, \Psi(t) \right\rangle + \left\langle \Psi(t) \,\middle|\, \mathbf{L} \,\middle|\, \dot{\Psi}(t) \right\rangle.$$

Substituting bra and ket with the time-dependent Schrödinger Equation 1.15 (namely $\left|\dot{\Psi}(t)\right\rangle = \frac{-i}{\hbar}\mathbf{H}\,|\Psi(t)\rangle$), we obtain:

$$\frac{d}{dt}\langle \Psi(t) \,|\, \mathbf{L} \,|\, \Psi(t) \rangle = \frac{i}{\hbar}\langle \Psi(t) \,|\, \mathbf{HL} \,|\, \Psi(t) \rangle - \frac{i}{\hbar}\langle \Psi(t) \,|\, \mathbf{LH} \,|\, \Psi(t) \rangle\,.$$

That can be rewritten as:

$$\frac{d}{dt}\langle \Psi(t) \,|\, \mathbf{L} \,|\, \Psi(t) \rangle = \frac{i}{\hbar}\langle \Psi(t) \,|\, [\mathbf{H},\mathbf{L}] \,|\, \Psi(t) \rangle\,.$$

The quantity $\mathbf{HL} - \mathbf{LH}$ is called the *commutator*, and since, in general, the product between operators (matrices) is not commutative, the commutator is not zero (when it is zero, we say that **H** and **L** commute). Commutators are important in physic, and the commutator between two operators, in this case **H** and **L**, is denoted by:

$$\mathbf{HL} - \mathbf{LH} = [\mathbf{H},\mathbf{L}].$$

With the commutator we can express concisely the derivative of the expectation value for the observable **L**:

$$\frac{d}{dt}\langle \mathbf{L} \rangle = \frac{i}{\hbar}\langle [\mathbf{H},\mathbf{L}] \rangle \tag{1.16}$$

or equivalently:

$$\frac{d}{dt}\langle \mathbf{L} \rangle = -\frac{i}{\hbar}\langle [\mathbf{L},\mathbf{H}] \rangle\,. \tag{1.17}$$

This equation links variations of the expectation values of an observable (**L**) to the expectation values of another physical observable ($-\frac{i}{\hbar}[\mathbf{L},\mathbf{H}]$)[16].

15 Derivative of a product: **L** doesn't depend on time and the dot denotes the time derivative (Newton notation).

16 It is possible to demonstrate that if **L** and **H** are Hermitian, then $[\mathbf{L},\mathbf{H}]$ is also Hermitian.

1.4.5 Conservation of Energy

In quantum mechanics, when we say that a quantity is conserved, we mean that the expectation value of that quantity doesn't change. If we look at Equation 1.17, the condition for the expectation value not to change is that the commutator between this quantity and the Hamiltonian is zero. It is possible to demonstrate that:

Theorem 3. *Having an observable $\mathbf{Q}$, if $[\mathbf{Q}, \mathbf{H}] = 0$, then every power satisfies $[\mathbf{Q}^n, \mathbf{H}] = 0$. This means that the expectation value $\langle Q \rangle$ is conserved, and any power of the expectation value $\langle Q^n \rangle$ does not change with time.*

The most obvious quantity that is conserved is the Hamiltonian $\mathbf{H}$ and, since every operator commutes with itself, we always have:

$$[\mathbf{H}, \mathbf{H}] = 0.$$

We can conclude that, under very general conditions, energy is conserved in quantum mechanics.

1.5 CONCLUSIONS

We conclude this chapter with a recap of what we have discovered in these pages, trying to put everything together to answer the question that opened this chapter: what are the physical limits of quantum computing, and why must our algorithms be reversible?

We started the chapter with an experiment that shows that quantum mechanics is not deterministic. We can, however, make some predictions if we consider the expectation value of a measurement instead of a single result.

We have built state vectors and understood their mathematical meaning, focusing on the fact that knowing the state vector doesn't allow us to know the result of a measurement. We have defined the inner product between state vectors, observed that it is a measure of the overlap between states, and concluded that two distinguishable states must be orthogonal.

We have linked a state vector to the result of a measurement –to be precise, to the average of the results of multiple measurements– with machines, Hermitian operators that represent observables. We have built the spin operator and used it to predict the result of a simple experiment, showing how the theory we have built so far is in accordance with experimental results.

Our introduction continues with the analysis of the temporal evolution of a quantum system. We have described the evolution of a state vector with an unitary operator; the application of this operator to a state vector produces the new state in which the system will be. We understood that the temporal evolution of the state vector is deterministic and that indeterminacy is caused only by the act of measuring.

Considering infinitesimal time intervals, we have deduced the time-dependent Schrödinger equation and, thanks to this equation, we have shown how to describe the temporal evolution of expectation values for a certain observable. During this analysis, we also introduced the Hamiltonian of the system, a Hermitian operator that describes the energy of the system.

The discussion ends with a comforting result: as in classical physics, the energy of a closed system is conserved. We have obtained this result by presenting the commutator and linking the temporal evolution of an observable with the commutator between the observable and the Hamiltonian (energy) of the system. The commutator of the Hamiltonian with itself is trivially zero, so the expectation value for the energy doesn't change.

All the information that we have learned allows us to understand the constraint of writing only reversible algorithms for quantum-gate-based quantum computers. Quantum gates operate on qubits through physical transformations[17]. These transformations, like all transformations in quantum mechanics, are described by unitary operators that are intrinsically reversible. This means that all quantum gates are reversible.

In other words, we can build only quantum gates that, having as input different (distinguishable) states, return orthogonal states; also, due to the conservation of overlaps, the inner product between input states is conserved during the quantum gate transformation.

Reversibility doesn't mean that we can go forward and backward in time as we please, but that all quantum gates express injective functions: if we know the output, we can know the input, or in more physical terms, if we know the final state of qubits[18] and the transformations applied to this system (i.e., those implemented by the quantum gates), we can determine the initial state.

Since every quantum algorithm has to be implemented as a path through quantum gates, and every quantum gate is reversible, the algorithms as a whole must also be reversible.

17 How depend strongly on the particular physical implementation.

18 This is a complex system (composed of more than one qubit); to fully understand these systems, we should take into account entanglement. Since our discussion is already quite long, and the temporal evolution of an entangled system is still unitary (reversible), we exclude entanglement from our introduction.

604

2 | QUANTUM GATE

3 QUANTUM ANNEALING

4 ONTOLOGY

In this chapter we explain what kind of knowledge base (KB) an ontology is, how to build an ontology, and why this knowledge representation is important. To clarify and demonstrate why ontologies are useful, we present an example of a foundational ontology[1], briefly discussing its utility.

The rest of the chapter is about reasoning on ontologies; we discuss the semantics of the formal language used to represent knowledge, what we mean when saying interpretation of a KB, and the complexity of finding an interpretation.

4.1 KNOWLEDGE BASE

In the field of information technologies, an ontology is a structured representation of knowledge about a certain domain of interest; however, the study of knowledge began much before informatics. To better understand what an ontology is, let's start with the philosophical definition and then point out the differences between this vision and the IT one.

4.1.1 Ontology in philosophy

Ontology was born as a branch of philosophy. In this context it is the science of what is, of the kinds and structures of objects, properties, events, processes, and relations in every area of reality [1].

The goal of an ontology is to give a definitive and exhaustive classification of entities in all spheres of being. With the term "definitive" we mean that an ontology should answer questions such as: "What classes of entities are needed for a complete description and explanation of all the goings-on in the universe?" With the term "exhaustive", instead, we mean that all types of entities and relations between these entities are included in our ontology [1].

4.1.2 Ontology in computer science

Thanks to the advent of the internet and the development of bigger and bigger software used by bigger and bigger groups of users, what we might call the Tower of Babel problem emerged. Each research group develops its KB with terms and concepts shared and accepted only inside the group. For example, different databases may use identical labels but with different meanings, and the same meaning may be expressed with different names [1].

1 a very general template the can be used as base (foundation) to build an ontology about a specific domain (more about that in Section 4.2.2).

To address the incompatibility problem between software, databases, and research groups, ontologies have become an important research topic in computer science where the goal is to define standards for data exchange, information integration, and interoperability [2].

In this field the term ontology gains a new meaning:

Definition 1. *Ontologies represent a formal and explicit specification of a shared conceptualization [3].*

In this definition the keywords are:

CONCEPTUALIZATION: an ontology creates an abstract model identifying and defining only the relevant concepts;

EXPLICIT: the types of concepts and constraints on their use are explicitly defined;

FORMAL: an ontology should be machine-readable;

SHARED: the knowledge represented by the ontology has to be accepted by a group of people, ideally by everyone.

When we use an ontology to represent knowledge we are describing a graph where entities are bound together through relationships, and classified according to a formal description of the world [4]. Knowledge bases expressed with this formalism are divided into two components [5]:

T-BOX: stores a set of universally quantified assertions (inclusion assertions) stating general properties of concepts and roles;

A-BOX: contains assertions on individual objects (instance assertions).

The T-Box is the conceptualization of the world, while the A-Box is a certain instance of the world we have modelled in the T-Box.

We can see some similarities between an ontology and a database: the T-Box can be seen as the Entity-Relation schema and the A-Box as the set of all entries of the database. There is, however, a logical difference between the world represented by an ontology and the world represented by a database.

Databases make the *closed world assumption*: everything that is not present in the database is automatically false; for example, if a person does not appear in a bank registry it means that that person is not a client of the bank.

Ontologies, on the other hand, make the *open world assumption* [6], which means, for example, that we can assert that a certain person is a parent even if we have not specified any son or daughter.

4.1.3 OWL Language

OWL 2 Web Ontology Language is an ontology language for the Semantic Web with a formally defined meaning [7]. Thanks to OWL we can model classes and relations between classes (T-Box) and individuals with their specific properties and relations between individuals (A-Box).

OWL is a declarative language and defines the state of the world in a logical way. In particular, we are interested in OWL DL where the meaning of ontologies expressed with this language is assigned in a Description Logic style. OWL DL is, therefore, decidable and an appropriate tool (so-called reasoner) can then be used to infer further information about that state of the world [7].

OWL per se does not specify any syntax, it states only what can or cannot be expressed in an ontology. The World Wide Web Consortium (W3C) standardizes various syntaxes, some inspired by functional languages, others more suitable for storing on web pages. The only syntax that must be implemented by all tools to be compliant with the OWL standard is the RDF/XML syntax [7] (examples of this syntax are provided in Section 4.2.1).

4.1.4 Importance of ontologies

Ontologies are important in various fields, from interoperability to machine learning.

In the Semantic Web context, ontologies are a main vehicle for data integration, sharing, and discovery [8]. Different research groups can use the same ontology to share a unified vocabulary that helps build common knowledge and helps to better integrate the results obtained by each group.

In a more commercial scenario, an ontology can be used as a translation layer between different databases or software that are built by different teams and use different vocabularies.

In the machine learning field an ontology could be used to support the sharing and reuse of formally represented knowledge among neuro-symbolic AI systems [3].

4.2 EXAMPLE ONTOLOGIES

To help understanding the structure of ontologies and to show a practical example of ontology, we present two ontologies: a simple ontology about family relationships and DOLCE, a foundational ontology.

4.2.1 Simple ontology

This simple ontology about parental relationships shows the basic structure of an ontology, helping to understand the graph structure of these KBs and the relations between the T-Box and A-Box.

In Figure 4.1, we can see the T-Box of the ontology: this structure specifies what our domain of interest is, and what entities could possibly populate our world. This ontology is about people, so the main class/concept is `People`: this class has several subclasses that represent parents, children, and married people. We can assert that a person belongs to the married class without specifying the partner (open world assumption) but we can also infer that a person belongs to the parents class because we have created a relationship of type `parent_of` between that person and another person.

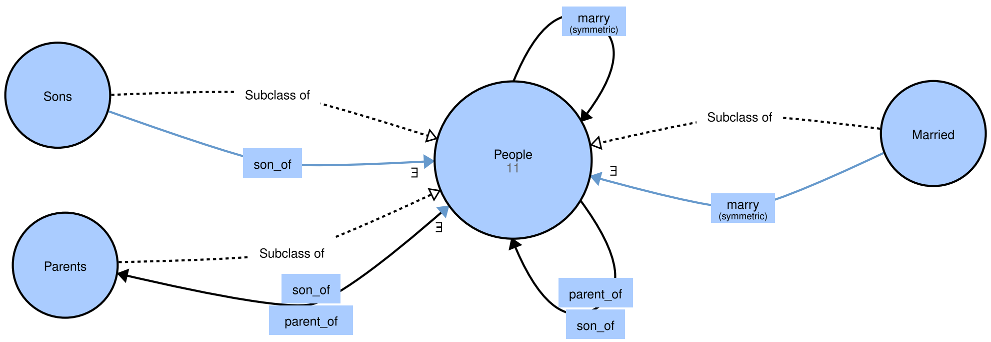


Figure 4.1: Graph for T-Box

OWL allows us to express rules to infer when a member of a class belongs also to another class. The following code shows (in the RDF/XML syntax) the definition of the class `Parent`[2]:

```
1 <owl:Class rdf:about="http://people#Parent">
2   <owl:equivalentClass>
3     <owl:Restriction>
4       <owl:onProperty rdf:resource="http://people#parent_of"/>
5       <owl:someValuesFrom rdf:resource="http://people#Person"/>
6     </owl:Restriction>
7   </owl:equivalentClass>
8   <rdfs:subClassOf rdf:resource="http://people#Person"/>
9 </owl:Class>
```

Listing 4.1: Definition of parents

At line 8 we can see that `Parent` is a subclass of `People`, and at lines 4 and 5 it is specified that a parent is a person that is `parent_of` another person.

From Figure 4.1 we can also see some properties of the relations:

- relation `marry` is symmetric;
- relation `parent_of` is the inverse of `son_of`;
- we can specify a domain and a range for relations.

OWL gives us constructs for all of these specifications (and other more complex ones).

Now we can populate the ontology by adding individuals and relations between individuals. For this small example we take inspiration from the Simpson family, and in the family tree (Figure 4.2 on the right) we can see the small portion of the family represented. To show what we mean by open world assumption we have asserted that Jackie is a married person even if in our representation there is no husband.

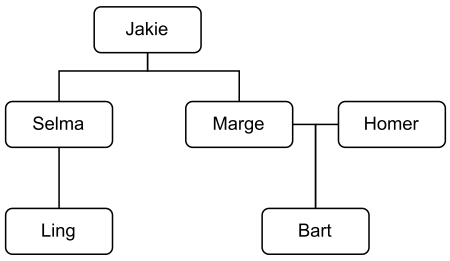


Figure 4.2: Simpson family tree

2 The complete code of the ontology can be seen at `url`.

Our ontology covers a small domain, the types of entities that populate our model are very limited; the next example shows the commitment of engineering an ontology to represent virtually anything in the universe.

4.2.2 DOLCE ontology

DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) is a top-level (foundational) ontology [9]; this means that this ontology describes fundamental aspects of reality and should be used as a base for constructing an ontology about a particular domain of interest. For this reason DOLCE defines only the T-Box; the user will then expand the T-Box with specific classes and relations of interest, and lastly will populate the A-Box.

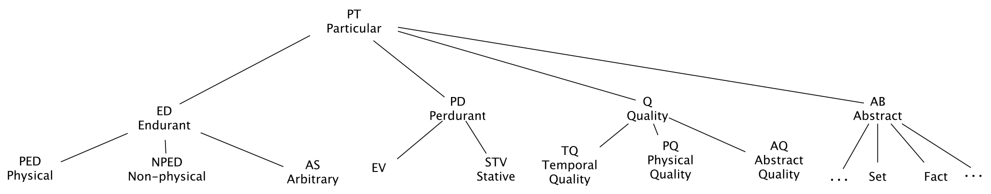


Figure 4.3: First layer of DOLCE taxonomy

STRUCTURE OF DOLCE: in DOLCE we can model the modification of objects during time; for this reason DOLCE distinguishes between endurants and perdurants. Endurants may acquire and lose properties and parts through time, perdurants are fixed in time [9]. With a simplification we can see endurants as the physical entities that are modified by the passing of time (like objects, animals, and people) and perdurants as events that, once they have passed, cannot be changed anymore (like a tennis match or a conference).

The relation connecting endurants and perdurants is called participation. A physical entity can be in time by participating in a perdurant, and perdurants happen in time by having endurants as participants [9].

Another important aspect of DOLCE is the way we attribute a property to an entity; this is done by using qualities, which are what can be perceived and measured. To do so we can assert that a certain entity has a specific quality and then, when it is possible, quantify that quality.

IMPORTANCE OF DOLCE: foundational ontologies can be useful in several fields, from conceptual modeling to natural language processing. DOLCE, today, is used in a variety of domains where it provides the general categories and relations needed to give a coherent view of reality [9].

4.3 REASONING ON ONTOLOGY

In Section 4.1.3 we have introduced the standard language to encode an ontology; in order to infer new information, starting from the one we already have, we need to better specify the semantics of OWL DL.

4.3.1 $\mathcal{SROIQ}$ DL

The semantics of OWL DL extends the semantics of the description logic (DL) $\mathcal{SROIQ}$ to provide support for datatypes and punning [10]. For constructs available both in OWL DL and in $\mathcal{SROIQ}$ the semantics correspond exactly.

Description logics allow the modeling of the domain of interest with three kinds of entity: concepts, roles, and individual names. These entities correspond to unary predicates, binary predicates, and constants in first-order logic [6]. From the point of view of ontology and OWL, concepts are classes, roles are relationships, and individual names are the individuals that can belong to one or more classes.

$\mathcal{SROIQ}$ is one of the most expressive description logics where we have constructors for:

- transitive roles: $\mathcal{S}$
- role inclusions, local reflexivity, universal role, symmetry, asymmetry, role disjointness, reflexivity, and irreflexivity: $\mathcal{R}$;
- nominals: $\mathcal{O}$;
- inverse roles: $\mathcal{I}$;
- qualified number restrictions: $\mathcal{Q}$.

For example, we can construct the ontology shown in Figures 4.1 and 4.2 with a set of assertions like:

```
person(selma)        married(jackie)        parent_of(marge, bart)
```

Each of these statements is called an axiom and the set of all axioms constitutes our KB.

4.3.2 Interpretation of a knowledge base

An interpretation I consists of a domain Δ^I and an interpretation function $\cdot^I$ that maps:

$$\begin{gathered}\text{concept } A \rightarrow A^I \subseteq \Delta^I\\ \text{role } R \rightarrow R^I \subseteq \Delta^I \times \Delta^I\\ \text{named individual } a \rightarrow a^I \in \Delta^I\end{gathered}$$

In other words I assigns a fixed meaning to all entities in the KB [6]. By having a fixed meaning, we can say if an axiom α holds in I or not; in the first case we say that I satisfies α and we write $I \models \alpha$.

If all axioms in an ontology are satisfied by I we say that I is a *model* of the ontology. An ontology is consistent if it accepts at least one model.

A reasoner should at least be capable of saying if an ontology is consistent, but we are also interested in querying knowledge to retrieve new information.

QUERY INTERPRETATION: Considering a KB K, a query q consists of axiom templates where $\mathcal{SROIQ}$ axioms are composed of concept names, role names, and individual names, but also of concept variables, role variables, and individual variables. A solution for the query is an interpretation μ that allows us to rewrite all variables in q with names; we denote with $\mu(q)$ the result of the substitution.

The evaluation of q over K is a set of solutions μ with: [11]

$$\{\ \mu | K \cup \mu(q) \text{ is a } \mathcal{SROIQ} \text{ knowledge base and } K \models \mu(q)\}$$

In other words μ binds all free variables of q to names present in K [11].

A naive approach to find the solution to a query is to simply test for each possible solution mapping μ, if $K \models \mu(q)$; however, in the worst case, the number of mappings that have to be tested is exponential in the number of variables in the query [11].

4.3.3 Complexity of reasoning

Since presenting an actual algorithm for reasoning on ontologies is out of the scope of this work, we only give some hints about the reasons for the complexity and then present the theoretical results that prove the problem of reasoning on ontology is at least PSpace-hard.

It is easy to convince oneself that the more axioms there are in an ontology, the fewer interpretations exist that satisfy all axioms. On the other hand, if an ontology has fewer models, the more axioms hold in all of them and the more logical consequences follow from the ontology.

In other words the semantics of description logics are *monotonic*: the more knowledge we embed in an ontology, the more results it returns [6]. A more formal view is given in [12], where two *sources of complexity* are identified:

- OR-branching: the presence of disjunctive constructors;
- AND-branching: the presence of qualified existential and universal quantifiers.

The AND-branching is responsible for the exponential size of a single interpretation, and the OR-branching is responsible for the exponential number of different interpretations.

To discuss the complexity of reasoning we take into account the description logic $\mathcal{ALC}$; this DL is a restriction of $\mathcal{SROIQ}$ [6], so its complexity is a lower bound for $\mathcal{SROIQ}$. It is possible to prove the PSpace-hardness of satisfiability in $\mathcal{ALC}$ [12], therefore also $\mathcal{SROIQ}$ DL is at least PSpace-hard.

This result shows that, unless PSpace = PTime, the exponential time complexity of any algorithm that makes inference on an ontology cannot be improved .

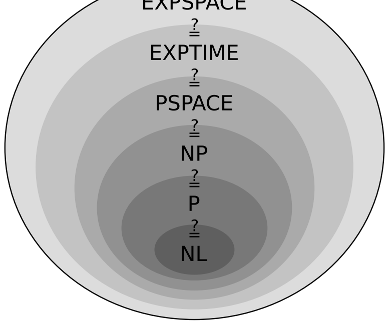


Figure 4.4: Complexity classes

For those interested in some numerical examples to better understand what this class of complexity means in a real context, [13] presents the reasoner HermiT and evaluates its performance on some real ontologies.

4.4 CONCLUSIONS

In this chapter we have explained what an ontology is and we have motivated the interest in this field.

We showed that an ontology can be useful both in academic research and in industry. Moreover, being a formal and machine-readable structure, it can be queried and used to perform logically demonstrable reasoning whose subject is precisely the knowledge represented within the ontology.

We have shown both theoretically and with examples what can be expressed in an ontology and what cannot. We have formally defined what the interpretation of a KB is and showed what a query and its results are.

Lastly, we have characterized the complexity of reasoning on ontologies. This complexity is what motivated us to search for other paradigms to infer new knowledge starting from an ontology. In the next chapters we will build the tools necessary to achieve this goal.

Part II

TOOLS

5 ENVIRONMENT SETUP

In this chapter we describe the environment, libraries and tools we use to execute our tests.

In the following sections we install the SDKs to develop and interact with quantum computers from IBM and D-Wave. We also present two other useful tools to easily write optimization problems.

5.1 PYTHON ENVIRONMENT

The language used to interface with quantum computers is usually Python. In this section we create a virtual environment in Python in order to communicate with the IBM quantum computer and the D-Wave quantum computer.

For our tests we manage Python environments with `conda`. Let's start by creating the virtual environment named `quantum` and activating it with:

```
$ conda create --name quantum python=3.12 pip
$ conda activate quantum
```

For our tests and to follow the various examples presented both by IBM and D-Wave, it is also useful to be able to run a Jupyter notebook. We can install Jupyter with:

```
$ pip install jupyter
```

5.2 IBM QISKIT

To program a gate-based architecture and to access IBM quantum computers we use the *Qiskit* software stack. The name Qiskit is a general term referring to a collection of software for executing programs on quantum computers.

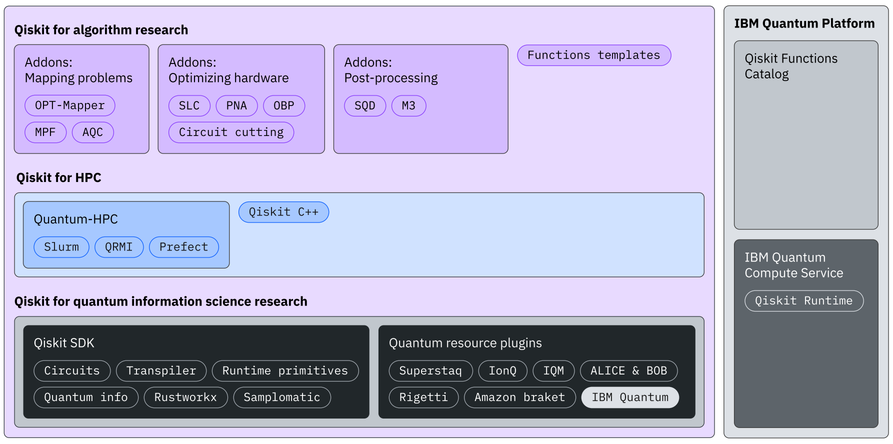


Figure 5.1: Qiskit software stack

The core components are *Qiskit SDK* and *Qiskit Runtime*. The first one is completely open source and allows the developer to define his circuit; the second one is a cloud-based service for executing quantum computations on IBM quantum computers.

5.2.1 Hello World

Following the IBM documentation[1] we can install the SDK and the Runtime with:

```
$ pip install qiskit matplotlib qiskit[visualization]
$ pip install qiskit-ibm-runtime
$ pip install qiskit-aer
```

Last command installs Aer, which is a high-performance simulator for quantum circuits written in Qiskit. Aer includes realistic noise models, and we will use it later to test our circuit.

Sometimes the Qiskit stack suffers from incompatibilities between the various software components that compose the environment. At the moment of writing, the latest packages seem to work without any problem. For our tests we will use `qiskit: 2.2.3`, `qiskit-ibm-runtime: 0.43.1` and `qiskit-aer: 0.17.2`.

If the setup is successful we are now able to run a small test to build a Bell state (two entangled qubits). The following code assembles the gates, shows the final circuit and uses a sampler to simulate on the CPU the result of 1024 runs of the program.

```python
from qiskit import QuantumCircuit
from qiskit.primitives import StatevectorSampler

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

sampler = StatevectorSampler()
result = sampler.run([qc], shots=1024).result()
print(result[0].data.meas.get_counts())
qc.draw("mpl")
```

Listing 5.1: Building Bell state

5.2.2 Transpilation

Each Quantum Processing Unit (QPU) has a specific topology. We need to rewrite our quantum circuit in order to match the topology of the selected device on which we want to run our program. This phase of rewriting, followed by an optimization, is called transpilation.

Considering, for now, a fake hardware (so we do not need an API key) we can transpile the quantum circuit `qc`, from the code above, to match the

1 https://quantum.cloud.ibm.com/docs/en/guides/install-qiskit

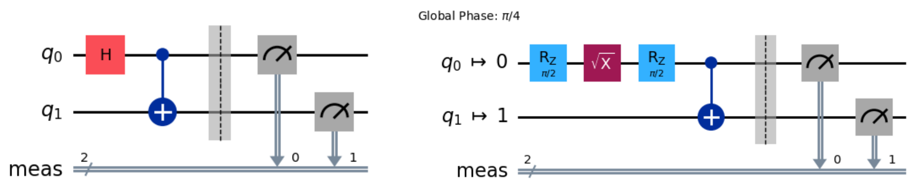
topology of a specific QPU. Listing 5.2 implement the transpilantion, and Figure 5.2 shows the circuits generated: (a) is the circuit whe have defined in Listing 5.1 and (b) is the transpiled version where the Hadamard gate is replaced to match the actual topology of the QPU.

```python
from qiskit_ibm_runtime.fake_provider import FakeWashingtonV2
from qiskit.transpiler import generate_preset_pass_manager

backend = FakeWashingtonV2()
pass_manager = generate_preset_pass_manager(backend=backend)

transpiled = pass_manager.run(qc)
transpiled.draw("mpl")
```

Listing 5.2: Transpilation



(a) Original circuit **(b)** Transpiled circuit

Figure 5.2: Transpilation example

5.2.3 Execution

To test our transpiled circuit we use Aer, which allows us to simulate also the noise of real quantum hardware. Listing 5.3 shows how to simulate the execution.

```python
from qiskit_aer.primitives import SamplerV2

sampler = SamplerV2.from_backend(backend)
job = sampler.run([transpiled], shots=1024)
result = job.result()
print(f"counts Bell circuit: {result[0].data.meas.get_counts()}")
```

Listing 5.3: Simulated execution

If we look at the results of the execution we can observe that some answers present non-entangled qubits; this is caused by the (simulated) noise of the quantum device. A typical output of the execution could be:

```
> counts Bell circuit: {'00': 504, '11': 503, '01': 10, '10': 7}
```

Where states `01` and `10` should not be present in an ideal execution with no errors.

5.3 D-WAVE OCEAN

To define an optimization problem that can be solved on a D-Wave quantum computer we use the Ocean software stack. Ocean also allows us to interact with D-Wave hardware, submit a problem, and simulate the execution on a classical CPU.

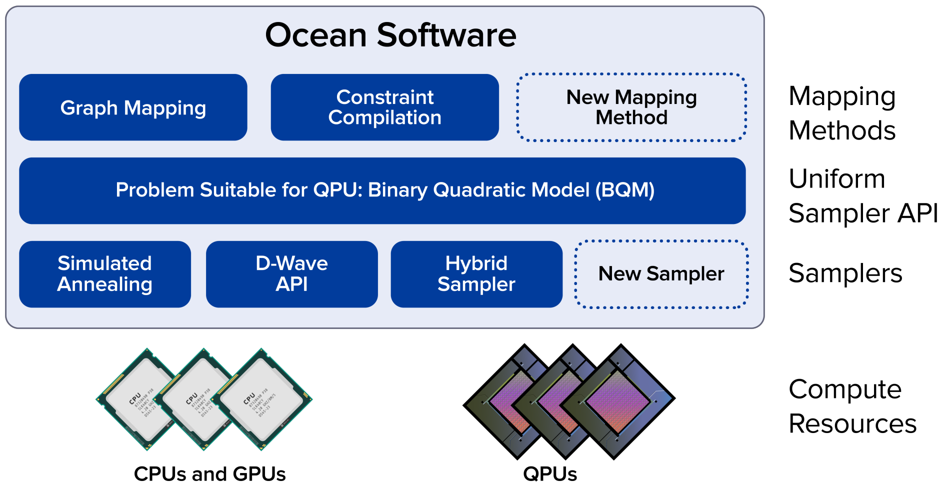


Figure 5.3: Ocean software stack

All tools that implement the steps needed to solve your problem on a CPU, a D-Wave quantum computer, or a quantum-classical hybrid solver can be installed with:

```
$ pip install dwave-ocean-sdk
```

After the installation, running the command `dwave setup` will start an interactive prompt that guides us through a full setup. During the setup it is also possible to add an API token or connect to the D-Wave account to import a key directly to use the quantum hardware.

5.3.1 Hello World

To present a simple optimization program we consider the minimum vertex cover (MVC) problem. Given a graph $G = (V, E)$, the problem asks to find a subset $V' \subseteq V$ such that, for each edge $\{u, v\} \in E$, at least one of u or v belongs to V', and the number of nodes in V' ($|V'|$) is the lowest possible.

The reduction from MVC to an Ising formulation is well known. The cost function that we want to minimize can be expressed by:

$$\text{cost} = \sum_{i=1}^{|V|} v_i + 2 \cdot \sum_{\{i,j\} \in E} (1 - v_i - v_j + v_i v_j)$$

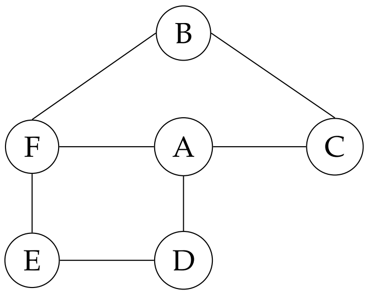
where $v_i \in \{-1, 1\}$ and $v_i = 1$ means that $v_i \in V'$, otherwise $v_i = -1$.

Like all problems in Ising form we can express the cost as a symmetric matrix, so our function becomes

$$\text{cost} = v^T \times \mathbf{M} \times v$$

where v is the vector containing the binary variables v_i.

The figure shows an example graph (5.4a) and the corresponding matrix (5.4b) expressing the cost function.



(a) Original graph

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 2 & 2 & 0 & 2 \\ 0 & 1 & 2 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Corresponding matrix

Figure 5.4: Ising formulation

The following code presents a possible implementation of the Ising model described above. We have defined two dictionaries to store the matrix coefficients. The last line of code finds ten possible answers to the problem using the simulated annealing function implemented by D-Wave.

```python
from dwave.samplers import SimulatedAnnealingSampler
linear = {'A': 1, 'B': 1, 'C': 1, 'D': 1, 'E': 1, 'F': 1}
quadratic = {('B', 'C'): 2, ('B', 'F'): 2, ('C', 'A'): 2, ('D', 'A'): 2, ('E', 'D'): 2, ('E', 'F'): 2, ('F', 'A'): 2}
sampler = SimulatedAnnealingSampler()
result = sampler.sample_ising(linear, quadratic, num_reads=10)
```

Listing 5.4: Ising example

If we print the results with `print(result.aggregate())` we can observe something similar to this:

```
  A  B  C  D  E  F energy num_oc.
0 -1 -1 +1 +1 -1 +1  -14.0       6
1 +1 +1 -1 -1 +1 -1  -14.0       4
['SPIN', 2 rows, 10 samples, 6 variables]
```

The two different results represent the two correct answers to our particular instance of the MVC problem.

5.4 PYQUBO AND QUBOVERT

In Listing 5.4 we have manually built the matrix representing the function that we want to minimize. It can be useful to have some tools that allow us to work at a higher level, defining cost functions like Equation **??** that we defined in the section about quantum annealing (Section **??**).

Considering again the MVC problem, the objective function tends to minimize the number of nodes in our subset, while the penalty increases the cost if we leave out some edges. This interpretation allows us to transform the Ising model into the more familiar —from the point of view of a computer scientist— QUBO model, where all variables $x_i \in \{0, 1\}$. Let's see how PyQUBO and qubovert help us in this task.

5.4.1 PyQUBO

Reading from the documentation on the PyQUBO site[2], PyQUBO allows us to create QUBOs or Ising models from flexible mathematical expressions easily. Some of the features of PyQUBO are:

- Python based (C++ backend);
- Fully integrated with Ocean SDK;
- Automatic validation of constraints;
- Placeholder for parameter tuning.

We can install PyQUBO with `$ pip install pyqubo` and rewrite our MVC problem by defining the Hamiltonian that we want to minimize.

```
 1  from pyqubo import Binary, Placeholder, Constraint
 2  from dwave.samplers import SimulatedAnnealingSampler
 3
 4  A, B, C, D, E, F  = Binary('A'), Binary('B'), Binary('C'),
    ↪  Binary('D'), Binary('E'), Binary('F')
 5
 6  H_objective = (A + B + C + D + E + F)
 7  H_penalty = Constraint(((1 - A - C + A*C) +\
 8  (1 - A - D + A*D) +\
 9  (1 - A - F + A*F) +\
10  (1 - B - C + B*C) +\
11  (1 - B - F + B*F) +\
12  (1 - D - E + D*E) +\
13  (1 - E - F + E*F)) ,label='cnstr0')
14
15  L = Placeholder('L')
16  H = H_objective + L*H_penalty
17  H_internal = H.compile()
18  bqm = H_internal.to_bqm(feed_dict={'L': 2})
19
20  sampler = SimulatedAnnealingSampler()
21  result = sampler.sample(bqm, num_reads=10)
```

Listing 5.5: Rewriting MVC with pyQUBO

Listing 5.5 presents a possible re-implementation of Listing 5.4, where we also see how PyQUBO interfaces with the Ocean SDK (line 17), and how to create (lines 14–16) and instantiate (line 17) a parametric Hamiltonian.

5.4.2 qubovert

As written in the documentation[3], qubovert is the one-stop package for formulating, simulating, and solving problems in boolean and spin form. Using our nomenclature, boolean and spin form are respectively QUBO and Ising form.

2 https://pyqubo.readthedocs.io/en/latest/
3 https://qubovert.readthedocs.io/en/latest/index.html

Qubovert allows us to define various types of optimization problems that can be solved by brute force, with qubovert's simulated annealing, or with D-Wave's solver. Models defined in qubovert are:

- QUBO: Quadratic Unconstrained Boolean Optimization;
- QUSO: Quadratic Unconstrained Spin Optimization (Ising model);
- PUBO: Polynomial Unconstrained Boolean Optimization;
- PUSO: Polynomial Unconstrained Spin Optimization;
- PCBO: Polynomial Constrained Boolean Optimization;
- PCSO: Polynomial Constrained Spin Optimization.

In addition to generic models, qubovert has a library of famous NP-complete problems mapped to QUBO and Ising forms.

```
1  from qubovert import boolean_var
2  from dwave.samplers import SimulatedAnnealingSampler
3
4  A, B, C, D, E, F  = boolean_var('A'), boolean_var('B'),
   ↪  boolean_var('C'), boolean_var('D'), boolean_var('E'),
   ↪  boolean_var('F')
5
6  model = A + B + C + D + E + F
7  model.add_constraint_OR(A, C, lam=2)
8  model.add_constraint_OR(A, D, lam=2)
9  model.add_constraint_OR(A, F, lam=2)
10 model.add_constraint_OR(B, C, lam=2)
11 model.add_constraint_OR(B, F, lam=2)
12 model.add_constraint_OR(D, E, lam=2)
13 model.add_constraint_OR(E, F, lam=2)
14
15 qubo = model.to_qubo()
16 dwave_qubo = qubo.Q
17
18 sampler = SimulatedAnnealingSampler()
19 result = sampler.sample_qubo(dwave_qubo, num_reads=10)
```

Listing 5.6: Rewriting MVC with qubovert

Listing 5.6 shows a possible implementation of the MVC problem using the tools provided by qubovert. Qubovert allows us to express our problem as a PCBO; we use this formulation to express constraints in a more natural way. In our example we ensure that each edge is covered simply by enforcing that at least one of the nodes linked by the edge is present in the solution. This constraint is repeated for each edge in the graph (lines 7–13). To specify the Lagrange multiplier (Equation **??**) we use the keyword `lam`.

Qubovert, like PyQUBO, can interface with the Ocean SDK, transforming a PCBO problem into a QUBO problem (line 15) and then rewriting it in the format accepted by the D-Wave solver (or sampler).

5.5 CONCLUSION

In this chapter we have set up an environment to run our future experiments and tests. We have also shown some small examples to present the main characteristics and test the tools we will use in our work.

Following this setup allows anyone to recreate exactly the same configuration we use, avoiding (for what we know and test) incompatibilities between Python packages.

6 QA-PROLOG

QA-Prolog is a tool that allows one to write a program in a logic programming language and execute it on a quantum annealer. QA-Prolog also retrieves the results returned by the quantum annealer and presents them in a natural and comprehensible way.

In this chapter we introduce the project and give some pointers to other related works. We describe extensively the pipeline of transformations starting from a Prolog code and ending with a Hamiltonian $\mathbf{H}_f$, as we have described in Section **??**. Next we present our contribution to the project, discussing the changes we have made to the original QA-Prolog code to restore compatibility with the modern framework used to interface with the D-Wave quantum annealer and to support the latest version of the library used in the project.

The chapter ends with an installation guide that ensures a working and reproducible environment to run experiments, both for this chapter and for the following ones.

6.1 THE PROJECT

QA-Prolog is a project developed by Scott Pakin[1] in $2017 - 2019$. It starts from the question: "Can one express constraint logic programming in the form accepted by quantum-annealing hardware?" [14].

The hope is that even if today we live in the *NISQ*[2] era of quantum computing, quantum annealers are more easily scalable than quantum gate-based computers [15], and QA-Prolog could improve Prolog program execution by replacing backtracking with fully parallel annealing into a solution state [16].

6.1.1 Reason

As we have shown in Chapters **??** and **??**, programming a quantum computer is not an easy task. We express our algorithm in a very low-level way.

On a quantum annealer we have to define a cost function, without constraints (which must be transformed into a penalty function). Even if there are libraries that allow us to express these functions in an easier way, we need at least to find a QUBO representation of our problem.

Even worse is the situation on quantum gate-based computers. The programmer has to build a quantum circuit gate by gate, an approach similar to what is done with FPGAs (Field Programmable Gate Arrays) [17]. We can indeed see a strong analogy:

FPGAs are components that the majority of computer scientists are not used to and are probably out of the interest of a programmer. In the same

1 Los Alamos National Laboratory: `pakin@lanl.gov`.
2 Noisy intermediate-scale quantum computing.

FPGA	Quantum gates computer
programmable logic blocks which implement logic functions	quantum gates
programmable routing that connects logic functions	possibility to define order of gates
I/O blocks connected to logic	input *qubits* and output *qubits* that can be measured

way, the hardness of programming a quantum computer could be a significant deterrent to attracting new researchers in the field.

Today some "high-level gates" are available that wrap multiple low-level gates into useful patterns. There also exist, both for quantum gate-based computers and quantum annealers, some templates of well-known problems that need only fine-tuning to be useful for a specific problem.

Despite these sorts of abstractions, programming a quantum computer is difficult and very close to machine language.

The goal of QA-Prolog is to have a tool that allows programming in a high-level style, with a powerful *logic programming language*, quantum computers.

6.1.2 Prolog

We can see QA-Prolog as a compiler from Prolog to $\mathbf{H_f}$, where the ground state of $\mathbf{H_f}$ is the solution of our Prolog program. Before starting with the compilation process, it is useful to understand the main characteristics of Prolog, because it is not an imperative programming language like C or Java, but a *declarative* one like SQL, moreover, Prolog is a logic programming language. This means that the core of programming is not to tell the computer what to do, but to tell it what is true and ask it to try to draw conclusions [18].

Discussing in detail characteristics of Prolog and logic programming language are out of scope of this work, more information about Prolog can be found in *Programming in PROLOG* [18] and *Learn Prolog Now!* [19], for the ones interested in logic programming language *Logic for problem solving*[20] and *Introduction to logic programming*[21] are recommended.

In Prolog we do not specify step by step an algorithm that resolves our problem, we describe instead the formal relationship between the objects in our problem and which relations have to be true in our solution [18].

Programming in Prolog consists of:

- listing *facts* about objects and relationships between objects;
- specifying *rules* to derive new facts from the ones already asserted;
- asking questions (*queries*) about objects and their relationships.

From these characteristics we can understand what "declarative" means: the program is a list of statements about our problem (our domain of interest); the relation between a Prolog program and an ontology is very strict.

In Prolog we encode a KB made of facts and rules; in Capter **??** we can see a complete example of rewriting from an OWL ontology into a Prolog KB[3].

ARITHMETIC PREDICATES: To better understand some QA-Prolog features that differ from basic Prolog we explain here how to assert equality between arguments when referring to integer. Predicates offered by Prolog are:

- `+Expr1 =:= +Expr2`: true if expression `Expr1` evaluates to a number equal to `Expr2`;
- `-Number is +Expr`: true when `Number` is the value to which `Expr` evaluates.

Where signs before arguments are *arguments mode indicators*. Sign `+` specifies that the following argument must be instantiated, `-` indicates that the argument is a output [22]. This mean that we cannot have variables when using `=:=` predicate because everything must be known at the time of evaluation. Using `is` the only variable can be the value of `Number`; `Expr`, on the other hand, has to be known.

EXAMPLE: Now we present a basic Prolog program in order to show the syntax and the usage of queries. We encode a KB about a party, we describe two people, Yolanda and Mia, asserting what are they doing and giving some rules to derive other informations about our small scenario.

```
1 sings(mia).
2 listens2Music(yolanda).
3 party.
4
5 dance(yolanda):- listens2Music(yolanda).
6 happy(yolanda):- dance(yolanda).
7 happy(mia):- sings(mia).
8
9 smile(X) :- happy(X).
```

Listing 6.1: Basic KB

In Listing 6.1, adapted from [19], lines 1 to 3 are facts: we are asserting that Yolanda is listening to music, Mia is singing, and there is a party. The other lines are rules: we can identify rules by the `:-` sign that divides the *head* of the rule on the left, from the *body* on the right. The head of a rule is true if the body is true.

For example, the rule at line 5 can be read as: "If Yolanda is listening to music, then Yolanda is dancing". Line 9 shows the usage of a variable: variables start with an uppercase letter and are placeholders for information. We can read this rule as "If someone is happy, they smile".

We can query our KB, asking for example if Yolanda is happy. In SWI-Prolog [23] we can interact with the interpreter, and the query we evaluate is `?- happy(yolanda).` (the full stop is part of the syntax and tells the interpreter that the query is complete). Prolog will answer `yes.`; this is because Yolanda

3 Knowledge Base

is listening to music, and if she is listening to music she dances, and if she dances she is happy.

By analogous reasoning it should be clear why the result of `?- smile(X).` is `X = mia; X = yolanda.`, where `;` means logical disjunction: *or*.

6.1.3 Features of QA-Prolog

QA-Prolog does not support all the features of Prolog, but enough to make basic logic programming possible [14].

QA-Prolog supports atoms and positive integers but not floating-point numbers, strings, or lists. It supports arithmetic and relational operations, and rules can reference other rules but not recursively. QA-Prolog supports unification, backtracking, and predicates comprising multiple clauses [14].

QA-Prolog also supports some features not present in the basic version of Prolog. In particular, operations can be performed on variables even before they are ground [14]; this means that QA-Prolog is more powerful in manipulating free variables. We can understand this feature looking back to Paragraph 6.1.2 where we have shown that aritmetica expression can have a free variable at most (`is` predicate) on the left operand.

In QA-Prolog we can have variables on the left and on the right of predicate[4], that way we can pass the input to the formula to obtain the result, but also assign a value to the result to retreive the input. We can already see that QA-Prolog allows a reversible computation: from input to output but also from output to input.

6.1.4 Related works

There are some other attempts to develop a high-level programming language for quantum computers, both for the quantum gate model and for quantum annealers.

Quantum Prolog [27] demonstrates that one can express the equivalent of a pure version of Prolog over finite relations in terms of a model of discrete quantum computing. This work targets quantum gate computers and focuses on the mathematical equivalence of relational programming and discrete quantum computing over the field of Booleans [14]. Quantum Prolog, however, remains a theoretical work that has the goal of proves some mathematical properties, there is no implementation and therefore is not suitable to run experiments.

C to D-Wave [28] reuses the pipeline we will discuss, replacing the starting step. The paper addresses the difficulty of programming quantum annealers by presenting a compilation framework that compiles a subset of C code into quantum machine instructions to be executed on a quantum annealer.

4 In Prolog is possible to obtain the same result with library like CLP(FD) [24, 25] and CLP($\mathbb{Z}$) [26]

6.2 PIPELINE

We are now ready to describe the pipeline of transformations that brings us from a Prolog program to a Hamiltonian $\mathbf{H}_f$.

The chain of transformations is shown in Figure 6.1, where the last step (in orange) is the quantum annealer capable of finding the ground state of $\mathbf{H}_f$. In purple we can see the various file formats throughout the pipeline, and in yellow the software that performs the rewriting. Most of this software is made ad hoc for the QA-Prolog pipeline on the other hand, we will see that Yosys is a tool used in digital circuits design.

From a high-level point of view, the pipeline rewrites the initial KB expressed in Prolog into different objects. The logical meaning of the entities we build during the pipeline is:

1. Prolog program (KB);
2. High-level digital circuit in Verilog;
3. Low-level digital circuit in EDIF format;
4. Symbolic Hamiltonian in qmasm formalism;
5. Physical Hamiltonian.

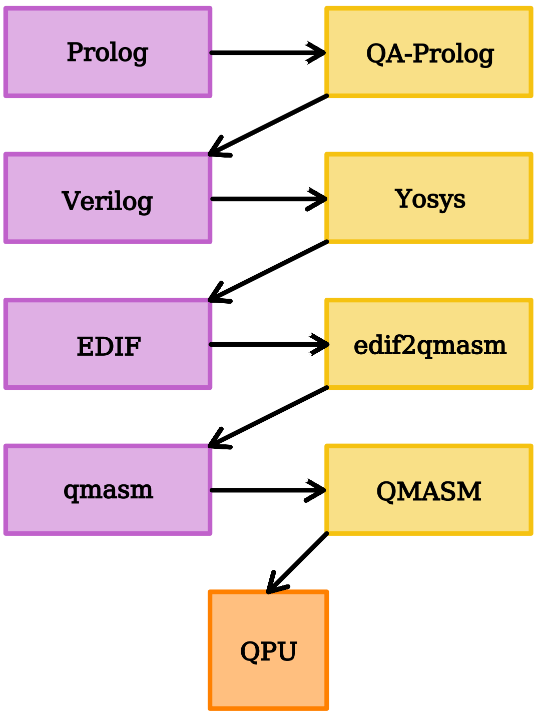


Figure 6.1: QA-Prolog pipeline

Let us start analyzing the pipeline from the last step, the one nearest to the QPU.

6.2.1 QMASM

QMASM is a *quantum macro assembler* [29]; it processes a symbolic Hamiltonian and assembles a physical Hamiltonian that can be embedded on a D-Wave quantum annealer. It was developed in Python by Scott Pakin with the goal of filling a gap in tools for the creation of D-Wave programs. QMASM is an abstraction layer that allows the programmer not to care about manually setting specific point weights and coupler strengths on the physical topology.

This software can be considered an assembler in the sense that it maps a symbolic representation of the operations (assembly language) to the machine language. QMASM is not only an assembler, but extends its functionality by including macros: parameterized named blocks of assembly language that a program can instantiate multiple times [29].

FEATURE: QMASM provides a number of features to simplify low-level D-Wave programming [29]. Moreover, we can also use QMASM to assemble a physical Hamiltonian that we can later manipulate or solve using classical methods or other quantum systems.

Some of the most useful and interesting features of QMASM are:

- *qubits* are referenced symbolically, not numerically, both in the source code and when QMASM reports execution results;
- *qubits* can be pinned to `true` or `false`;
- *qubit* patterns can be encapsulated into macros and instantiated repeatedly;
- groups of macros can be encapsulated into libraries and reused across multiple programs;
- QMASM can automatically exclude from the results the solutions known to be incorrect and shows only "interesting" *qubits*.

Thanks to this set of features, QMASM is already a useful abstraction layer that simplifies the development of programs that target an annealer, classical or quantum.

EXAMPLE: Let us consider an example that shows the potential of QMASM and that is also useful for the following steps of the pipeline. The satisfiability problem is well known to be an NP-complete problem [30], so it is a good candidate for our example. We take into account the simple formula:

$$y = x_1 \wedge \neg(x_2 \vee x_3) \tag{6.1}$$

QMASM allows us to define a macro for every logical operator needed to represent the formula and then assemble the final formula by calling these macros. In Listings 6.2 we call `A` and `B` the input *qubits* and `Y` the output *qubit*. Weights are specified as an Ising problem (Section **??**), which is the default format for QMASM source files[5].

There are multiple interesting details about these macros:

- the symbol `#` identifies a comment line not processed;
- the symbol `$` is used to tag a *qubit* as ancillary: unless explicitly requested by the programmer, the intermediate results are not reported in the solutions;
- compared with what we have done in Listing 5.4, defining weights is much easier and the result is more readable;
- thanks to the directive `!assert` we can inform QMASM about constraints on the solution. This directive does not change the weights, but allows the programmer to exclude from the solutions those that are surely incorrect.

```
# Y = A AND B
!begin_macro AND
    !assert $Y=$A&$B
    $A -0.5
    $B -0.5
    $Y  1

    $A $B  0.5
    $A $Y -1
    $B $Y -1
!end_macro AND
```

(a) *and* gate

```
# Y = NOT A
!begin_macro NOT
    !assert $Y=!$A
    $A $Y 1.0
!end_macro NOT
```

(b) *not* gate

```
# Y = A OR B
!begin_macro OR
    !assert $Y=$A|$B
    $A  0.5
    $B  0.5
    $Y -1

    $A $B  0.5
    $A $Y -1
    $B $Y -1
!end_macro OR
```

(c) *or* gate

Listing 6.2: Logical operators

It is possible to verify, for each macro in Listings 6.2, that given a configuration of the input *qubits*, the value of `Y` that minimizes the energy corresponds to the output of the logic gate we are modeling.

We can save these macros in a file named `gates.qmasm` and use it to solve our problem. To compute the formula $y = x_1 \wedge \neg(x_2 \vee x_3)$ we will use some intermediate results: we start with $x_4 = (x_2 \vee x_3)$, then we apply the negation $x_5 = \neg x_4$, and lastly we compute the result as $y = x_1 \wedge x_5$.

```
1   # Solve circuit-satisfiability problem.
2
3   !include <gates>
4
5   !use_macro OR x2_or_x3
6     x2_or_x3.$A = x2
7     x2_or_x3.$B = x3
8     x2_or_x3.$Y = $x4
9
10  !use_macro NOT not_x4
11    not_x4.$A = $x4
12    not_x4.$Y = $x5
13
14  !use_macro AND x1_and_x5
15    x1_and_x5.$A = x1
16    x1_and_x5.$B = $x5
17    x1_and_x5.$Y = y
```

Listing 6.3: Circuit satisfiability

The QMASM code implementing this procedure is reported in Listing 6.3. This example shows how to use macros: we instantiate a macro and give it a name with `!use_macro <macro_name> <instance_name>` (e.g. line 5), and then instantiate the *qubits* defined in the macro with our actual *qubits*. For example, considering the `!use_macro OR` at line 5, we can see that we use x_2 and x_3 as input and an ancillary *qubit* $\$x_4$ as output. Another detail to point out is the directive `!include` (line 3), which imports all gates used in this source file.

5 as specified in https://github.com/lanl/qmasm/wiki/File-format.

Finally, we can query the quantum annealer (or in this case a classical solver) to find a solution for our satisfiability problem. To do so we pin the output variable y to be sure that in the solution its value will be `true`. We can query the solver with:

```
qmasm --run --pin="y := true" --solver="sim_anneal" circ_sat.qmasm
```

where `circ_sat.qmasm` is the file name of our source code.

QMASM interfaces directly with the Ocean framework and can query one of the solvers made available by D-Wave. Retrieving the solutions results in a call to Ocean's library functions very similar to the one shown in Listing 5.4 on line 5. QMASM also offers the possibility to set the annealing time and the number of reads directly from the command line; in this case, the default values were used, which are the default annealing time stetted by D-Wave for the specific solver and 1000 samples, respectively.

Results are reported in Listing 6.4. Here we can see that the solver has correctly found the solution, but only 642 times out of the default 1000 samples. This is caused by the stochastic nature of annealing, simulated or quantum. QMASM automatically removed the solutions that do not respect the `!assert` directive or do not have the minimum energy.

```
# x1 -->   12
# x2 -->    3
# x3 -->    4
# y --> [True]
Solution #1 (energy = -20.0000, tally = 642):

Variable  Value
--------  -----
x1        True
x2        False
x3        False
y         True
```

Listing 6.4: Circuit satisfiability results

QMASM already offers some powerful abstractions to work with quantum annealers. Now we add two additional layers that allow a programming style more similar to the paradigms computer scientists are used to, while preserving strong control over variable dimensions and therefore over the number of *qubits* used.

6.2.2 Yosys and edif2qmasm

These steps of the pipeline take as input a *Verilog HDL* (Hardware Description Language) program and transform it into a symbolic Hamiltonian.

We aggregate two steps because Yosys is not a tool developed for this pipeline and is used mostly to optimize the intermediate results of the transformations. Also, Yosys and edif2qmasm work on the same logical entity: a digital circuit that, in these steps of the pipeline, is converted into a symbolic Hamiltonian.

Verilog is a hardware description language that offers different levels of design abstraction. The highest level is *behavioural* and uses programming constructs such as assignments, conditionals, and while-loops. The lowest level is a connection of gates (*netlists*) [31].

Hardware description languages are not something that the majority of programmers are familiar with, but they offer some advantages when targeting a quantum annealer [32]:

- they provide precise control over bit widths in order not to waste any *qubit*;
- they can be compiled to a small set of primitives: the gates we can define with QMASM.

In these steps of the pipeline we start using behavioural constructs, then we automatically generate a netlist as the output of *synthesisers*.

The synthesiser used is Yosys [33], a free and open-source software for Verilog HDL synthesis.

The two most useful features of Yosys are the possibility of specifying a *cell library*, the set of gates used to synthesize the netlist, and the use of the external tool *Berkeley ABC* [34] (incorporated in Yosys), providing additional code optimizations.

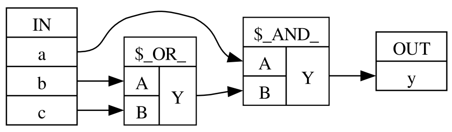
Yosys lowers a Verilog program to an EDIF netlist that uses only a specified set of gates; then edif2qmasm lowers the netlist into a symbolic Hamiltonian. This Hamiltonian uses a set of macros defining the energy for each type of gate Yosys can use[6]. What we end up with is a symbolic Hamiltonian that can be used as input for QMASM.

EXAMPLE: Considering again the simple formula $y = x_1 \wedge \neg(x_2 \vee x_3)$ (Equation 6.1).

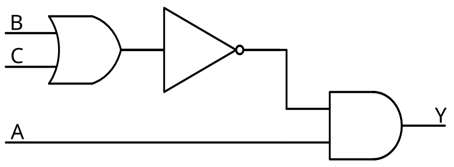
```
module sat (A, B, C, Y);
   input A, B, C;
   output Y;

   assign  Y = A & ~(B | C);
endmodule
```

(a) Verilog Code



(b) Netlist synthesized



(c) Corresponding circuit

Figure 6.2: Yosys processing

Figure 6.2 shows a possible Verilog implementation of the formula (a), the netlist Yosys has produced from the Verilog code (b), and, only for clarification purposes, the corresponding digital circuit implemented with logic gates (c).

6 The actual set of gates extends the logical ports *and*, *or*, and *not* and is available at: https://github.com/lanl/edif2qmasm/blob/master/stdcell.qmasm.

Now we can feed the obtained netlist to edif2qmasm to obtain the code reported in Listing 6.5.

```
1  !include <stdcell>
2
3  !begin_macro sat
4    !use_macro AND $id00006
5    !use_macro NOT $id00004
6    !use_macro OR $id00005
7    $id00004.A = $id00005.Y  # $or$sat.v:5$1_Y
8    $id00005.A = b
9    $id00005.B = c
10   $id00006.A = a
11   $id00006.B = $id00004.Y  # $not$sat.v:5$2_Y
12   $id00006.Y = y
13 !end_macro sat
14
15 !use_macro sat sat
```

Listing 6.5: Simbolic Hamiltonian

Comparing the automatically generated Listing 6.5 and the manually written 6.3, we can observe some differences:

- in Listing 6.5 we define the new macro `sat`: each module in the Prolog program is converted into a macro;
- the generated code is a little less readable but more compact. We do not need to generate new ancillary variables, but instead use directly as input for macros the output of other macros.

Even if it is less readable, we can comment on the code in Listing 6.5. At lines 4, 5, and 6 we declare that we need the macros `AND`, `NOT`, and `OR`, giving to each macro a name like `$id<number>`[7]. At line 7 we assign as input `A` of the macro `NOT` the output `Y` of the macro `OR`. Given these hints, the rest of the code should be clear.

6.2.3 QA-Prolog

This is the last step of the pipeline, the one that provides the maximum level of abstraction from the hardware that finds the solution to our problem.

QA-Prolog, like edif2qmasm, is written in GO [14, 32] and compiles a Prolog program to a Verilog one; each fact or rule is converted into a module. Once QA-Prolog has compiled the Prolog source code, including the query that can be specified on the command line, into Verilog, QA-Prolog invokes Yosys, edif2qmasm, and QMASM, as illustrated in Figure 6.1 [14].

QMASM executes the user's program on a D-Wave system or on a simulator and reports the value of each symbol appearing in the QMASM source file. QA-Prolog maps these lists of Booleans back to integers and named atoms, associates those values with variables named in the user's query, and reports all variables and their values to the user just as a typical Prolog environment would [14].

7 The symbol $ tells QMASM that we are not interested in the value of *qubits* inside the macro.

EXAMPLES: The first example we present shows a small KB that can be summarized as "the enemy of my enemy is my friend".

```
1 hates(alice, bob).
2 hates(bob, charlie).
3
4 enemy(X, Y) :- hates(X, Y).
5 enemy(X, Y) :- hates(Y, X).
6
7 friend(X, Y) :-
8   enemy(X, Z),
9   enemy(Z, Y).
```

Listing 6.6: KB enemy of my enemy is my friend

The code is reported in Listing 6.6. Thanks to the declarative nature of Prolog we can immediately understand the meaning of our KB: we describe relations between three people, assert that if a person hates another one they are enemies (lines 4 and 5), and that if two people share an enemy they are friends (lines 7, 8, and 9).

We can now query the KB to find if there are two people who are friends with:

```
QA-Prolog --qmasm-args="--solver=tabu" --query='friends(P1, P2).'
 ↪ --work-dir="~/work" friends.pl
```

In this command we can see that QA-Prolog allows us to query the KB with the exact same syntax we would use to interact with a Prolog interpreter. As expected, QA-Prolog *binds* the variables `P1` and `P2` to Alice and Charlie.

Another detail to point out is the parameter `--work-dir="<path>"`, which specifies the folder in which QA-Prolog outputs all files and the solver: we use a *tabu search algorithm* because the simulated annealing implemented in Ocean is not capable of finding a solution even in this small scenario (more considerations about that in Chapter **??**). From the study reported in [35], we can observe that real-world ontologies define a number of classes, individuals, and properties[8] that can be on the order of hundreds of thousands.

If we inspect the working directory we can see the output of every step of the pipeline; in particular, to see how exactly the KB is rewritten into a Verilog program, in Appendix .1 there is the result of compilation, and it is possible to observe in detail how each fact or rule has been transformed.

The second example concludes our transformation of the satisfiability problem (Equation 6.1); unfortunately, this is not very didactic because the Prolog program suitable for QA-Prolog processing introduces a consistent amount of overhead.

Indeed, we have to implement logical operators by defining their truth tables, then assemble the operators in our logical formula. Listing 6.7 shows the implementation we use in our test.

We can now run QA-Prolog with the query `sat(A, B, C, true)` to find the correct assignment of Boolean variables that satisfy our logic formula. Again, we are forced to use the tabu search algorithm. If we now explore the working directory of QA-Prolog we can see that the symbolic matrix

8 That we will transform in facts and rules in Chapter **??**.

```prolog
and(false, false, false).
and(false, true, false).
and(true, false, false).
and(true, true, true).

not(false, true).
not(true, false).

or(false, false, false).
or(false, true, true).
or(true, false, true).
or(true, true, true).

sat(A, B, C, Y) :-
  or(B, C, X),
  not(X, Z),
  and(A, Z, Y).
```

Listing 6.7: Satisfiability in Prolog

generated by the pipeline is a lot more complex that the one reported in Listing 6.5, we can interpret that as the cost of abstraction.

6.2.4 Overview

We have described a pipeline of rewritings that, layer after layer, makes it possible to abstract away from the hardware that actually solves our problem, moving toward a language, and thus a way of reasoning, that is at a higher level. We speak of rewritings because the object we are manipulating remains logically unchanged, just as it is essential that the solutions we are searching for remain unchanged; the problem and its solutions are simply expressed in different languages as one moves down the pipeline.

The advantages of abstraction are essentially the same as those obtained when moving from assembly language to a high-level language: greater ease of expression, improved readability, and the possibility of working with already familiar paradigms. The drawbacks are the overheads that naturally arise during the translation process. QA-Prolog is still a prototype, and if today compilers produce better machine code than a programmer could write manually, it is because significant effort and many years of research have been invested to achieve excellent results. The hope is that QA-Prolog and related works represent a first step in the same direction within the field of quantum computing.

6.3 UPDATE TO THE PROJECT

In this section we briefly describe the updates we have made to the project in order to restore compatibility with the Ocean framework and with the other libraries used. We also provide a small guide to install the tools to ensure that all packages work properly and our experiments are reproducible.

6.3.1 Restoring QMASM

The last commit to QA-Prolog was made in 2019, and the last one to QMASM in 2021. Since then, packages and libraries have changed and the compatibility with the QA-Prolog pipeline has broken.

The most fragile component of the pipeline is QMASM, because it is the one that interacts with external frameworks that are likely to change. In particular, the development of Ocean is very active, and some functions used in QMASM have now been removed, deprecated, or moved to other packages.

The incompatibilities encountered between imports used in QMASM and external libraries are:

- in the `dwave.cloud` and `hybrid` libraries;
- in the libraries defining D-Wave samplers (classical solvers);
- in the `scipy.stats` library.

The documentation available for Ocean[9] and for scipy[10] helped us resolve these incompatibilities: most of the time the solution consisted simply in correcting the name of the library.

Another small bug, caused by a different name of a solver in the command line helper and in the actual code, was fixed by restoring the correspondence.

6.3.2 Fixing interaction edif2qmasm-QMASM

During the execution of the complete pipeline we encountered an error caused by undefined macros in the QMASM source file.

```
1 // Define hates(atom, atom).
2 module \hates/2 (A, B, Valid);
3   ...
4 endmodule
```

(a) Verilog Code

```
1 # hates/2
2 !begin_macro id00011
3   ...
4 !end_macro id00011
```

(b) Macro definitions

```
1 # enemies/2
2 !begin_macro id00010
3   !use_macro hates/2 $id00032  # hates_PWYwG/2
4   ...
5 !end_macro id00010
```

(c) Macro usage

Listing 6.8: Undefined macros error

The error can be seen in Listings 6.8: edif2qmasm rewrites the netlist generated from (a) into the macro (b); here `hates/2` is just a comment, the actual name of the macro is `id00011` (specified at line 2). In (c), however, we can see that the macro previously defined is called symbolically and not with its actual name (lines 3). This obviously produces an error: the name `hates/2` in the QMASM file has no meaning, it is only a comment.

9 Available at https://docs.dwavequantum.com/en/latest/index.html.
10 Available at https://docs.scipy.org/doc//scipy/index.html.

```python
with open(file, 'r') as input:
  first = input.readline()
  second = input.readline()
  while(second_row != ""):
    if first.startswith("#") and second.startswith("!begin_macro"):
      self.name[first[2:-1]] = second[len("!begin_macro "):-1]
    first_row = second_row
    second_row = input.readline()

with open(file, 'r') as input:
  doc = input.read()
  lines = doc.splitlines()
  for line in lines:
    for word in line.split():
      if word in self.name.keys():
        line = line.replace(word, self.name[word])
      self.new_lines.append(line)
```

Listing 6.9: Preprocessor's core

To solve the problem we added a preprocessing step before parsing with QMASM. During the preprocessing all symbolic names are substituted with the actual name of the corresponding macro.

The core code of the preprocessor is reported in Listing 6.9. The program scans the source file two times: during the first reading a Python dictionary *symbolic_name-actual_name* is built, then, during the second reading, all instances of the symbolic name are replaced with the actual name.

Now all components should work properly, and we can install all the software needed and run some experiments.

6.3.3 Installation guide

In order to have a working environment where we can run our experiments, we need to install all the software required by the QA-Prolog pipeline. In Chapter 5 we have set up a Python environment with all the essential tools; to start the installation of the QA-Prolog pipeline we need at least the Ocean SDK installed as described in Section 5.3.

The first component we need is QMASM: we can install the updated and fixed version from `https://github.com/DavideCamino/qmasm.git`. The installation procedure consists of the following three commands:

```
$ git clone https://github.com/DavideCamino/qmasm.git
$ cd qmasm
$ python setup.py install
```

Next we need Yosys and GO: Yosys is part of the pipeline, and GO allows us to compile edif2qmasm and QA-Prolog. Both Yosys and GO are available from their official website `https://yosyshq.net/` and `https://go.dev/`, also on most of GNU/Linux distributions Yosys and GO can be installed directly from the package manager of the distribution.

The `go install` command installs Go executables in the default directory `$HOME/go/bin`. It is useful to add the bin subdirectory to `PATH`. This can be done by editing the `.bashrc` file (or the corresponding one for the specific shell), adding:

```
PATH=$PATH:$HOME/go/bin
export PATH
```

Finally we can install edif2qmasm and QA-Prolog with:

```
$ go install github.com/lanl/edif2qmasm@latest
$ go install github.com/lanl/QA-Prolog@latest
```

6.4 CONCLUSION

In this chapter, we presented QA-Prolog, a rewriting pipeline that enables the transformation of a Verilog program into a Hamiltonian whose ground state represents the solution of the original program. We discussed the various steps of the pipeline in detail, also showing how some parts were modified to make them compatible again with current frameworks. Finally, we described how to install a working version of the pipeline.

From our discussion, it emerges that QA-Prolog is a modular software stack in which it is possible to replace a component of the pipeline with another to modify the result; for example, we presented work that starts from C instead of Prolog.

This work has the potential to provide a foundation for many other experiments, that can rely on an already implemented and functioning infrastructure, in order to develop new extensions, both upstream and downstream, of the pipeline.

Part III

EXPERIMENTS

7 A QUANTUM ONTOLOGY

7.1 ONTOLOGY STRUCTURE

7.2 PROLOG VERSION

7.3 INFERENCE ON THE ONTOLOGY

7.4 CONCLUSION

8 QAOA

8.1 QAOA

8.2 FROM QUBO TO PAULI OPERATOR

8.3 EXPERIMENTS

8.4 CONCLUSION

1488

9 CONCLUSION

BIBLIOGRAPHY

[1] Barry Smith. “Ontology”. In: (2012).

[2] Gian Piero Zarri. “Ontologies and Their Practical Implementation”. In: *Encyclopedia of Database Technologies and Applications*. IGI Global Scientific Publishing, 2005, pp. 438–449.

[3] Thomas R Gruber. “A translation approach to portable ontology specifications”. In: *Knowledge acquisition* 5.2 (1993), pp. 199–220.

[4] Marco Fossati, Emilio Dorigatti, and Claudio Giuliano. “N-ary relation extraction for simultaneous T-Box and A-Box knowledge base augmentation”. In: *Semantic Web* 9.4 (2018), pp. 413–439.

[5] Giuseppe De Giacomo, Maurizio Lenzerini, et al. “TBox and ABox reasoning in expressive description logics.” In: *KR* 96.316-327 (1996), p. 10.

[6] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. “A description logic primer”. In: *arXiv preprint arXiv:1201.4089* (2012).

[7] Pascal Hitzler et al. *OWL 2 Web Ontology Language Primer (Second Edition)*. W3C Recommendation REC-owl2-primer-20121211. World Wide Web Consortium (W3C), Dec. 2012. URL: https://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[8] Pascal Hitzler. “A review of the semantic web field”. In: *Communications of the ACM* 64.2 (2021), pp. 76–83.

[9] Stefano Borgo et al. “DOLCE: A descriptive ontology for linguistic and cognitive engineering”. In: *Applied ontology* 17.1 (2022), pp. 45–69.

[10] Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau. *OWL 2 Web Ontology Language: Direct Semantics (Second Edition)*. W3C Recommendation REC-owl2-direct-semantics-20121211. World Wide Web Consortium (W3C), Dec. 2012. URL: https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/.

[11] Ilianna Kollia, Birte Glimm, and Ian Horrocks. “Query answering over SROIQ knowledge bases with SPARQL”. In: *Proceedings of the 24th International Workshop on Description Logics, Barcelona, Spain*. 2011, pp. 13–16.

[12] Franz Baader. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[13] Birte Glimm et al. “HermiT: an OWL 2 reasoner”. In: *Journal of automated reasoning* 53.3 (2014), pp. 245–269. URL: http://www.hermit-reasoner.com/.

[14] Scott Pakin. “Performing fully parallel constraint logic programming on a quantum annealer”. In: *Theory and Practice of Logic Programming* 18.5-6 (2018), pp. 928–949.

[15] William M Kaminsky, Seth Lloyd, and Terry P Orlando. "Scalable superconducting architecture for adiabatic quantum computation". In: *arXiv preprint quant-ph/0403090* (2004).

[16] Los Alamos National Laboratory / Scott Pakin. *QA-Prolog: Quantum Annealing Prolog*. Accessed: 2026-02-13, GitHub repository. URL: https://github.com/lanl/QA-Prolog.

[17] Umer Farooq, Zied Marrakchi, and Habib Mehrez. "FPGA architectures: An overview". In: *Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization* (2012), pp. 7–48.

[18] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[19] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now! – Free Online Version*. Accessed: 2026-02-13. 2012. URL: https://lpn.swi-prolog.org/lpnpage.php?pageid=online.

[20] Robert Kowalski and Steve Smoliar. "Logic for problem solving". In: *ACM SIGSOFT Software Engineering Notes* 7.2 (1982), pp. 61–62.

[21] Christopher John Hogger. *Introduction to logic programming*. Academic Press Professional, Inc., 1984.

[22] SWI-Prolog Developers. *SWI-Prolog Reference Manual*. Accessed: 2026-02-20. SWI-Prolog. URL: https://www.swi-prolog.org/pldoc/doc_for?object=manual.

[23] Jan Wielemaker et al. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96. ISSN: 1471-0684.

[24] Markus Triska. "The finite domain constraint solver of SWI-Prolog". In: *International Symposium on Functional and Logic Programming*. Springer. 2012, pp. 307–316.

[25] Markus Triska. *clpfd: Constraint Logic Programming over Finite Domains*. https://github.com/triska/clpfd. GitHub repository, accessed 2026-02-20.

[26] Markus Triska. *clpz: Constraint Logic Programming over Integers*. https://github.com/triska/clpz. GitHub repository, accessed 2026-02-20.

[27] Roshan P James, Gerardo Ortiz, and Amr Sabry. "Quantum computing over finite fields". In: *arXiv preprint arXiv:1101.3764* (2011).

[28] Mohamed W Hassan, Scott Pakin, and Wu-chun Feng. "C to D-wave: a high-level C compilation framework for quantum Annealers". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–8.

[29] Scott Pakin. "A quantum macro assembler". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2016, pp. 1–8.

[30] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. wh freeman New York, 2002.

[31] Mike Gordon. "The semantic challenge of Verilog HDL". In: *Proceedings of tenth annual IEEE symposium on logic in computer science*. IEEE. 1995, pp. 136–145.

[32] Scott Pakin. “Targeting classical code to a quantum annealer”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 529–543.

[33] Clifford Wolf, Johann Glaser, and Johannes Kepler. “Yosys-a free Verilog synthesis suite”. In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. Vol. 97. 2013.

[34] Robert Brayton and Alan Mishchenko. “ABC: An academic industrial-strength verification tool”. In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 24–40.

[35] Hongyu Zhang, Yuan-Fang Li, and Hee Beng Kuan Tan. “Measuring design complexity of semantic web ontologies”. In: *Journal of Systems and Software* 83.5 (2010), pp. 803–814.

APPENDIX 1

```
// Verilog version of Prolog program friends.pl
// Conversion by QA-Prolog, written by Scott Pakin <pakin@lanl.gov>
//
// This program is intended to be passed to edif2qmasm, then to qmasm, and
// finally run on a quantum annealer.
//
// Note: This program uses 3 bit(s) for atoms and 1 bit(s) for (unsigned)
// integers.

// Define all of the symbols used in this program.
`define alice   3'd0
`define bob     3'd1
`define charlie 3'd2
`define enemies 3'd3
`define friends 3'd4
`define hates   3'd5

// Define hates(atom, atom).
module \hates/2 (A, B, Valid);
  input [2:0] A;
  input [2:0] B;
  output Valid;
  wire [1:0] $v1;
  assign $v1[0] = A == `alice;
  assign $v1[1] = B == `bob;
  wire [1:0] $v2;
  assign $v2[0] = A == `bob;
  assign $v2[1] = B == `charlie;
  assign Valid = &$v1 | &$v2;
endmodule

// Define enemies(atom, atom).
module \enemies/2 (A, B, Valid);
  input [2:0] A;
  input [2:0] B;
  output Valid;
  wire $v1;
  \hates/2 \hates_pujkx/2 (A, B, $v1);
  wire $v2;
  \hates/2 \hates_DoUbH/2 (B, A, $v2);
  assign Valid = &$v1 | &$v2;
endmodule

// Define friends(atom, atom).
module \friends/2 (A, B, Valid);
  input [2:0] A;
  input [2:0] B;
  output Valid;
  (* keep *) wire [2:0] C;
  wire [2:0] $v1;
  \enemies/2 \enemies_GcbiL/2 (A, C, $v1[0]);
  \enemies/2 \enemies_GNLnA/2 (C, B, $v1[1]);
  assign $v1[2] = A != B;
  assign Valid = &$v1;
endmodule

// Define Query(atom, atom).
module Query (P1, P2, Valid);
  input [2:0] P1;
  input [2:0] P2;
  output Valid;
  wire $v1;
  \friends/2 \friends_KMGjP/2 (P1, P2, $v1);
  assign Valid = &$v1;
endmodule
```

Listing .1: QA-Prolog compilation result