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 measures 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:

122   • The z component of the spin is +1 *or* the x component is +1.

123   • The z component of the spin is +1 *and* the x component is +1.

124   Starting with the disjunction. Considering a state prepared, without our
125   knowledge, with $\sigma_z = +1$. If our first measure is along the z axis, $\mathcal{A}$ will
126   always display +1 and we can immediately conclude that the proposition
127   is true. If we start measuring the x component, we have a 50% chance that
128     $\mathcal{A}$ displays +1 or −1; also this measurement destroys the initial state and
129   the measure of σ_z becomes non predictable. In this scenario we have a 25%
130   chance of deducing that the proposition is false; figure 1.2 shows all the
131   possible measurement results in this case. The logical value of a proposition
132   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

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

The conjunction loses its meaning

1.2   QUANTUM STATES

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

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

145   This is a physical axiom, which means that it is true because there are a lot
146   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 $\mathscr{B} = \{|i_1\rangle, |i_2\rangle, \ldots, |i_N\rangle\}$ of a space with N dimensions, we can write a generic vector in that space as

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

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:

$$\alpha_i = \langle i | A\rangle. \qquad (1.2)$$

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* $|o\rangle$.

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.

Spin space states have two dimensions

To express a vector we need a basis; we choose $\mathcal{B} = \{\,|u\rangle\,,|d\rangle\,\}$[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).

$|A\rangle$ 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:

Probability amplitudes

$$P_u = \langle A|u\rangle \langle u|A\rangle$$
$$P_d = \langle A|d\rangle \langle d|A\rangle. \tag{1.3}$$

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:

$$\langle u|d\rangle = 0$$
$$\langle d|u\rangle = 0.$$

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.

252 that if we orient $\mathcal{A}$ along z and measure $\sigma_z = +1 = |u\rangle$, we would have
253 $\alpha_d = \langle d\,|\,u\rangle \neq 0$, which is a contradiction to experimental results. If $\alpha_d \neq 0$,
254 then $\alpha_d^* \alpha_d > 0$; we started with a state prepared as $\sigma_z = +1$ and ended with
255 a nonzero probability of measuring $\sigma_z = -1$: this is absurd.

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

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

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

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

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

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

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

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

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

270 • states must be orthogonal: $\langle i\,|\,o\rangle = \langle o\,|\,i\rangle = 0$;

271 • if we have a spin prepared as *in* or *out*:

272 – equiprobability of measuring $\sigma_z = +1$ and $\sigma_z = -1$;

273 – equiprobability of measuring $\sigma_x = +1$ and $\sigma_x = -1$.

274 Two vectors that satisfy these constraints are:

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

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

275 This last derivation shows why it is important that the state space is complex:
276 if we only accepted real components for our vectors, the system of equations
277 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.

278 ## 1.3 OBSERVABLES

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

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

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

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

290 ### 1.3.1 Hermitian operator

291 Formally, machines modify a state vector in this way:

$$\mathbf{M}\,|A\rangle = |B\rangle$$

292 The linearity of machines implies that:

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

293 and:

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

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

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

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

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

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

$$\mathbf{M}\,|\lambda\rangle = \lambda\,|\lambda\rangle$$

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

---

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

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

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

313   We can now write:

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

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

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

316   Hermitian operators have some important properties:

317   • all eigenvalues are real;

318   • eigenvectors form a *complete set*: all vectors obtained with the applica-
319   tion of the operator can be expressed as a linear combination of eigen-
320   vectors;

321   • if λ_1 and λ_2 are different eigenvalues, the associated eigenvectors are
322   orthogonal;

323   • if two eigenvalues are equal (*degeneracy*), it is always possible to find
324   two associated eigenvectors that are orthogonal.

Fundamental theorem 325   The last three properties can be summed up in the following way:

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

327   **1.3.2   Principles of quantum mechanics**

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

330   **Principles 1.** *Observables in quantum mechanics are described by linear operators*
331 $\mathbf{L}$.

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

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

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

---

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

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

342   Distinguishable states can be separated without ambiguity by a measure-
343 ment. For example, if we want to distinguish between $|u\rangle$ and $|d\rangle$, we mea-
344 sure σ_z: *up* and *down* are distinct. We cannot, instead, say if a certain system
345 is in state *up* or *right*, because even if the system is in the state $|u\rangle$ we can
346 still measure σ_x and find (with 50% chance) that the system is in state $|r\rangle$.
347   The inner product is a measure of how much two states are indistinguish-   *Overlap*
348 able; for that reason it is also called overlap. Two states are physically dis-
349 tinct if the overlap is zero.

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

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

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

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

354 ### 1.3.3   Spin Operator

355 The principles tell us what properties a machine must have to represent an
356 observable. Let's construct the spin operator σ.
357   Until now, we have measured spins with the apparatus $\mathcal{A}$, orienting $\mathcal{A}$
358 along the component of our interest. σ is a mathematical tool that allows
359 us to make predictions about the result of a measurement with $\mathcal{A}$ (fourth
360 principle); as we can rotate $\mathcal{A}$, we must also rotate σ (mathematically). For
361 this spatial property, σ is called a *3-vector operator*.

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

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

365 In matrix form:

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

366 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.

367 **OPERATOR σ_x:** With the same reasoning, we can construct the operator
368 along the x axis. We have already deduced the representations of *right* and
369 *left* in equations 1.6 and 1.7 on page 10. The equations that allow us to
370 construct σ_x are:

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

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

371 The solution of this system is:

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

372 **OPERATOR σ_y:** The last direction is along the y axis. Considering the
373 expressions for *in* and *out* given in equations 1.8 and 1.9 on page 10, and
374 following the second principle, we can write:

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

375 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 376 We have obtained a matrix representation of the three spin operators σ_z,
377 σ_x, and σ_y:

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

378 These famous and important matrices are named after their inventor, Wolf-
379 gang Ernst Pauli.

380 **1.3.4 Theory and experiments**

381 Thanks to the operators σ_z, σ_x, and σ_y, if we know the state vector, we can
382 statistically predict the result of a measurement of the spin along one of
383 the three coordinate axes. What can we say about a measurement taken by
384 orienting the apparatus $\mathcal{A}$ along a generic direction?
385 Considering $\mathcal{A}$ oriented along the unit vector $\hat{n}$, if σ behaves as a 3-vector,
386 in order to obtain σ_n we only need the inner product:

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

387 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:

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

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 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 $|u\rangle = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$. If we apply the operator σ_z, we would obtain again $\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$, 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:

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

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:

Determinism

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:

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

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):

Conservation of Distinctions

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

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

484 1.4.3 The Hamiltonian

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

Continuity 488   In quantum mechanics we can follow the same path and consider time
489 intervals ϵ close to zero. In this scenario, after an ϵ amount of time, the state
490 vector will change slightly and "smoothly", and the operator $\mathbf{U}(\epsilon)$ will be
491 very similar to the identity. We can rewrite $\mathbf{U}(\epsilon)$ in order to highlight the
492 difference with the identity $\mathbf{I}$ as:

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

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

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

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

497 Bringing to the left the time interval:

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

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

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

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

506   The equation becomes:

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

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

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

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

516 Expanding to first order in ϵ, we find:

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

517 We have concluded that $\mathbf{H}$ is an Hermitian operator that represents an *Quantum Hamiltonian*
518 observable: the energy of the system. Eigenvalues of $\mathbf{H}$ are the results of all
519 possible direct measurements of the energy of the system.

1.4.4 Commutators

521 In a system that evolves with time, we expect that the expectation values for
522 a certain observable $\mathbf{L}$ will also change. Thanks to equation 1.11 on page 15,
523 we can write explicitly the time dependence of expectation values:

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

524 The time derivative[15] is:

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

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

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

527 That can be rewritten as:

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

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

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

533 With the commutator we can express concisely the derivative of the expec-
534 tation value for the observable $\mathbf{L}$:

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

535 or equivalently:

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

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

---

15 Derivative of a product: $\mathbf{L}$ doesn't depend on time and the dot denotes the time derivative
(Newton notation).

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

538 **1.4.5 Conservation of Energy**

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

544 **Theorem 3.** *Having an observable* $\mathbf{Q}$, *if* $[\mathbf{Q}, \mathbf{H}] = 0$, *then every power satisfies*
545 $[\mathbf{Q}^n, \mathbf{H}] = 0$. *This means that the expectation value* $\langle Q \rangle$ *is conserved, and any*
546 *power of the expectation value* $\langle Q^n \rangle$ *does not change with time.*

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

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

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

551 **1.5 CONCLUSIONS**

552 We conclude this chapter with a recap of what we have discovered in these
553 pages, trying to put everything together to answer the question that opened
554 this chapter: what are the physical limits of quantum computing, and why
555 must our algorithms be reversible?
556 We started the chapter with an experiment that shows that quantum me-
557 chanics is not deterministic. We can, however, make some predictions if we
558 consider the expectation value of a measurement instead of a single result.
559 We have built state vectors and understood their mathematical meaning,
560 focusing on the fact that knowing the state vector doesn't allow us to know
561 the result of a measurement. We have defined the inner product between
562 state vectors, observed that it is a measure of the overlap between states,
563 and concluded that two distinguishable states must be orthogonal.
564 We have linked a state vector to the result of a measurement –to be precise,
565 to the average of the results of multiple measurements– with machines, Her-
566 mitian operators that represent observables. We have built the spin operator
567 and used it to predict the result of a simple experiment, showing how the
568 theory we have built so far is in accordance with experimental results.
569 Our introduction continues with the analysis of the temporal evolution of
570 a quantum system. We have described the evolution of a state vector with an
571 unitary operator; the application of this operator to a state vector produces
572 the new state in which the system will be. We understood that the tempo-
573 ral evolution of the state vector is deterministic and that indeterminacy is
574 caused only by the act of measuring.
575 Considering infinitesimal time intervals, we have deduced the time-dependent
576 Schrödinger equation and, thanks to this equation, we have shown how to
577 describe the temporal evolution of expectation values for a certain observ-
578 able. During this analysis, we also introduced the Hamiltonian of the system,
579 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.

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, 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].

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].

27

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).

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). 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.

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 doesn't 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 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.
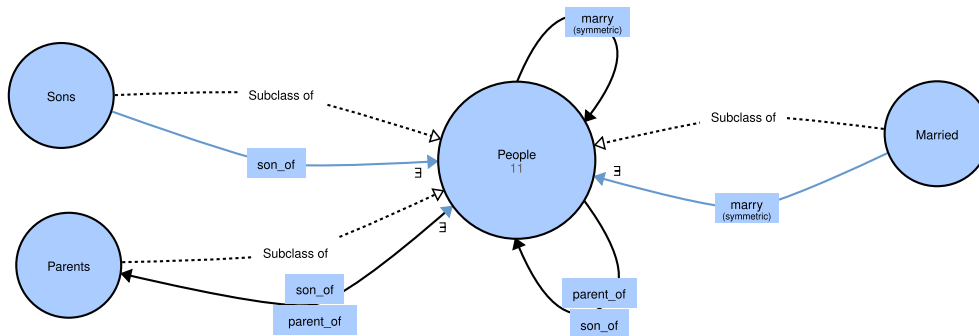

Figure 4.1: Graph for T-Box

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.

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`[1]:

```
<owl:Class rdf:about="http://people#Parent">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://people#parent_of"/>
      <owl:someValuesFrom rdf:resource="http://people#Person"/>
    </owl:Restriction>
  </owl:equivalentClass>
  <rdfs:subClassOf rdf:resource="http://people#Person"/>
</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.
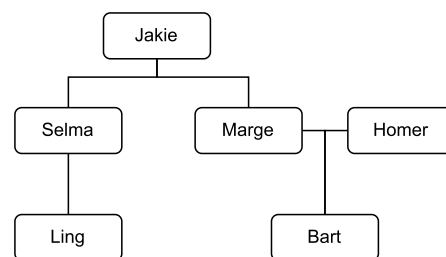

Figure 4.2: Simpson family tree

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.

---

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

⁷⁴⁵ **4.2.2 DOLCE ontology**

⁷⁴⁶ DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) is a
⁷⁴⁷ top-level (foundational) ontology[9]; this means that this ontology describes
⁷⁴⁸ fundamental aspects of reality and should be used as a base for constructing
⁷⁴⁹ an ontology about a particular domain of interest. For this reason DOLCE
⁷⁵⁰ defines only the T-Box; the user will then expand the T-Box with specific
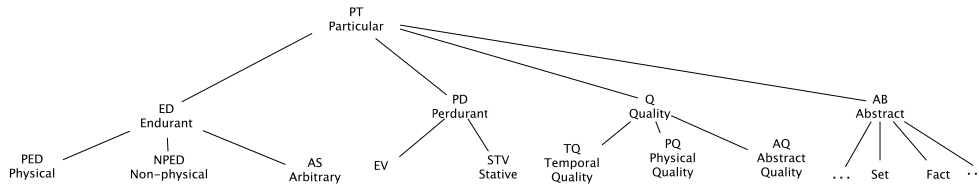⁷⁵¹ classes and relations of interest, and lastly will populate the A-Box.

Figure 4.3: First layer of DOLCE taxonomy

⁷⁵² **STRUCTURE OF DOLCE:** in DOLCE we can model the modification of ob-
⁷⁵³ jects during time; for this reason DOLCE distinguishes between endurants
⁷⁵⁴ and perdurants. Endurants may acquire and lose properties and parts through
⁷⁵⁵ time, perdurants are fixed in time[9]. With a simplification we can see en-
⁷⁵⁶ durants as the physical entities that are modified by the passing of time (like
⁷⁵⁷ objects, animals, and people) and perdurants as events that, once they have
⁷⁵⁸ passed, cannot be changed anymore (like a tennis match or a conference).
⁷⁵⁹ The relation connecting endurants and perdurants is called participation.
⁷⁶⁰ A physical entity can be in time by participating in a perdurant, and perdu-
⁷⁶¹ rants happen in time by having endurants as participants[9].
⁷⁶² Another important aspect of DOLCE is the way we attribute a property to
⁷⁶³ an entity; this is done by using qualities, which are what can be perceived
⁷⁶⁴ and measured. To do so we can assert that a certain entity has a specific
⁷⁶⁵ quality and then, when it is possible, quantify that quality.

⁷⁶⁶ **IMPORTANCE OF DOLCE:** foundational ontologies can be useful in several
⁷⁶⁷ fields, from conceptual modeling to natural language processing. DOLCE,
⁷⁶⁸ today, is used in a variety of domains where it provides the general cate-
⁷⁶⁹ gories and relations needed to give a coherent view of reality[9].

⁷⁷⁰ ## 4.3 REASONING ON ONTOLOGY

⁷⁷¹ In 4.1.3 we have introduced the standard language to encode an ontology; in
⁷⁷² order to infer new information, starting from the one we already have, we
⁷⁷³ need to better specify the semantics of OWL DL.

⁷⁷⁴ **4.3.1 $\mathcal{SROIQ}$ DL**

⁷⁷⁵ The semantics of OWL DL extends the semantics of the description logic
⁷⁷⁶ (DL) $\mathcal{SROIQ}$ to provide support for datatypes and punning[10]. For con-

structs 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:

$$\text{concept A} \to A^I \subseteq \Delta^I$$
$$\text{role R} \to R^I \subseteq \Delta^I \times \Delta^I$$
$$\text{named individual } a \to a^I \in \Delta^I$$

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 μ(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.

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.

We can rephrase these two statements by saying that 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.
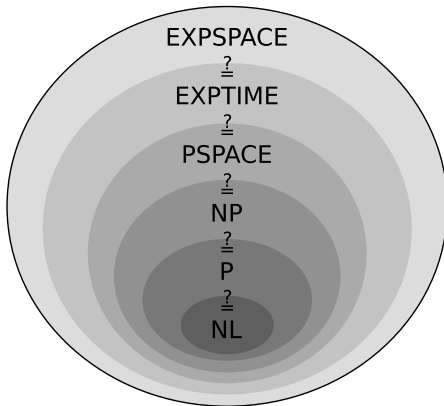

Figure 4.4: Complexity classes

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 .

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 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 softwares for executing programs on quantum computers.
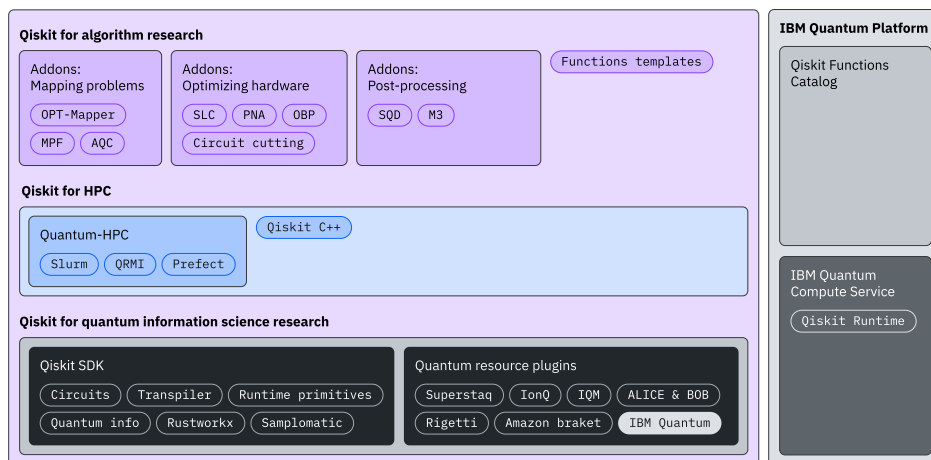

Figure 5.1: Qiskit software stack

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

893 5.2.1 Hello World

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

```
1  pip install qiskit matplotlib qiskit[visualization]
2  pip install qiskit-ibm-runtime
3  pip install qiskit-aer
```

896 Line 3 installs Aer, which is a high-performance simulator for quantum
897 circuits written in Qiskit. Aer includes realistic noise models, and we will
898 use it later to test our circuit.

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

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

```python
1  from qiskit import QuantumCircuit
2  from qiskit.primitives import StatevectorSampler
3
4  qc = QuantumCircuit(2)
5  qc.h(0)
6  qc.cx(0, 1)
7  qc.measure_all()
8
9  sampler = StatevectorSampler()
10 result = sampler.run([qc], shots=1024).result()
11 print(result[0].data.meas.get_counts())
12 qc.draw("mpl")
```

Listing 5.1: Building Bell state

908 5.2.2 Transpilation

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

---

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

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

916 The following picture shows (5.2a) the quantum circuit that builds a Bell
917 state, and (5.2b) the transpiled version where the Hadamard gate is replaced
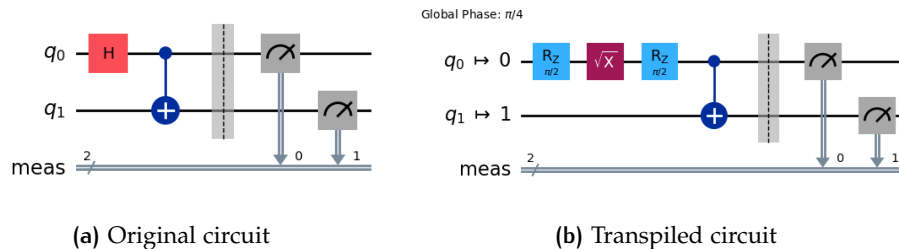to match the actual topology of the QPU.

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

Figure 5.2: Transpilation example

918

919 ### 5.2.3   Execution

920 To test our transpiled circuit we use Aer, which allows us to simulate also
921 the noise of real quantum hardware. We can execute our program with:

```python
from qiskit_aer.primitives import SamplerV2

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

Listing 5.3: Simulated execution

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

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

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

5.2.4 A complete example on real hardware

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.
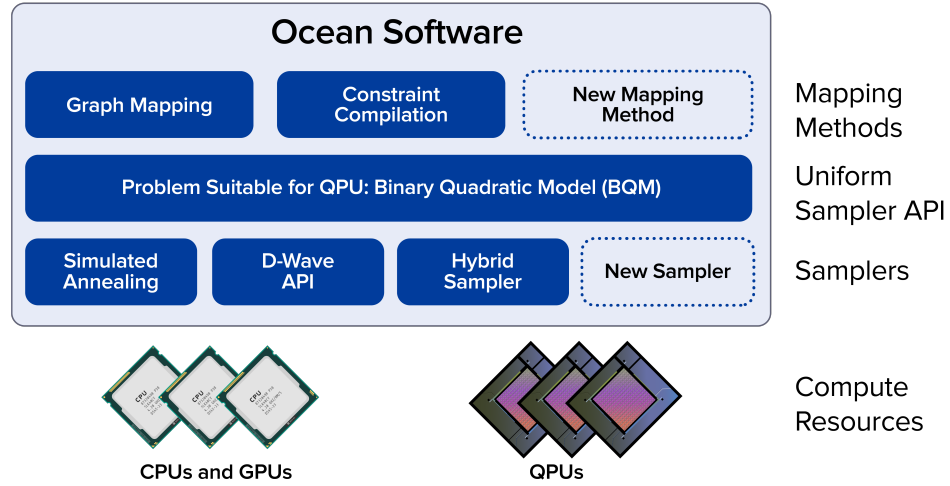

Figure 5.3: Ocean software stack

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:

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

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

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

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

950 where v is the vector containing the binary variables v_i.
951 The figure shows an example graph (5.4a) and the corresponding matrix
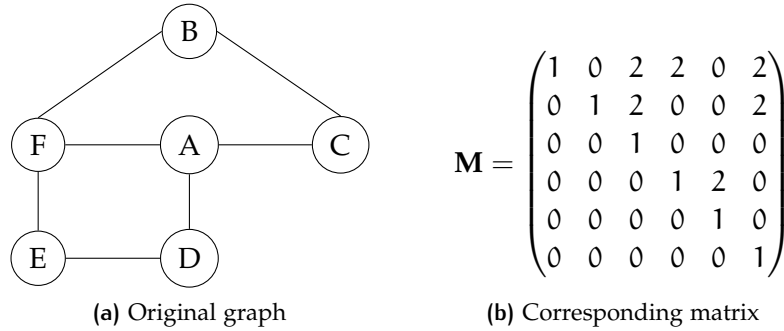952 (5.4b) expressing the cost function.

(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

953 The following code presents a possible implementation of the Ising model
954 described above. We have defined two dictionaries to store the matrix coeffi-
955 cients. The last line of code finds ten possible answers to the problem using
956 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

957 If we print the results with `print(result.aggregate())` we can observe
958 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]
```

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

961 ### 5.3.2 Example on real hardware

962 ### 5.3.3 Minor embedding

963 ## 5.4 PYQUBO AND QUBOVERT

964 In listing 5.4 we have manually built the matrix representing the function
965 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 **??** that we defined in the section about quantum annealing (**??**).

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.

```python
from pyqubo import Binary, Placeholder, Constraint
from dwave.samplers import SimulatedAnnealingSampler

A, B, C, D, E, F  = Binary('A'), Binary('B'), Binary('C'), Binary('D'),
↪  Binary('E'), Binary('F')

H_objective = (A + B + C + D + E + F)
H_penalty = Constraint(((1 - A - C + A*C) +\
(1 - A - D + A*D) +\
(1 - A - F + A*F) +\
(1 - B - C + B*C) +\
(1 - B - F + B*F) +\
(1 - D - E + D*E) +\
(1 - E - F + E*F)) ,label='cnstr0')

L = Placeholder('L')
H = H_objective + L*H_penalty
H_internal = H.compile()
bqm = H_internal.to_bqm(feed_dict={'L': 2})

sampler = SimulatedAnnealingSampler()
result = sampler.sample(bqm, num_reads=10)
```

Listing 5.5: Rewriting MVC with pyQUBO

---

2 https://pyqubo.readthedocs.io/en/latest/

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

987 **5.4.2 qubovert**

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

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

995 QUBO: Quadratic Unconstrained Boolean Optimization;

996 QUSO: Quadratic Unconstrained Spin Optimization (Ising model);

997 PUBO: Polynomial Unconstrained Boolean Optimization;

998 PUSO: Polynomial Unconstrained Spin Optimization;

999 PCBO: Polynomial Constrained Boolean Optimization;

1000 PCSO: Polynomial Constrained Spin Optimization.

1001 In addition to generic models, qubovert has a library of famous NP-complete
1002 problems mapped to QUBO and Ising forms.

```python
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

---

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

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 to write a program in a logic programming language and execute it on a quantum annealer, QA-Prolog also retreive the results returned by the quantum annealer and present them in a natural and comprensible way.

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

The chapter ends wit an installation guide that ensure a working and reproducible environment to run experiments, both of this chapter and of 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 computer quantum annealer are more easily scalable that quantum gate based computer[15] and QA-Prolog could improving Prolog program execution by replacing backtracking with fully parallel annealing into a solution state[16].

6.1.1 Reason

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

On quantum annealer we have to define a cost function, without constrain (that must be transformed in a penalty function), even if there are library that allow ut to express this functions in a easier way we need at least find a QUBO representation of our problem.

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

---

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

- programmable logic blocks which implement logic functions → quantum gate;

- programmable routing that connects these logic functions → possibility to define the order of gates;

- I/O blocks connected to logic → input *qubit* and output *qubit* that we can measure.

FPGAs are components that the majority of computer scientist are not used to and probably is out of the interest for a programmer. In the same way the hardness of programming a quantum computer could be a big distinctiveness to attract new researcher in the field.

In conclusion even if there exist some sort of abstraction with "high level gates" that wrap multiple low level gates in useful pattern and exist both for quantum gate based computer and quantum annealer some template of well known problem that need only a fine-tuning to be useful for a specific problem programming a quantum computer is, today, very near to the machine language.

The goal of QA-Prolog is to fill the gap between the high level description with a powerful logic programming language and the promising 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 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. For more information about Prolog lecture of [18] and [19] are recommended.

In Prolog we do not specify step by step an algorithm that resolve our problem; instead we describe the formal relationship between the object in our problem and what relation has to be true in our solution[18].

Programming in Prolog consist in:

- listing *facts* about objects and relationship between objects;

- specify *rules* to derive new facts from the ones already asserted;

- asking question (*query*) about objects and their relationship.

From these characteristics we can understand what means "declarative": 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 knowledge base made of facts and rules, in **??** we can see a complete example of rewriting from an OWL ontology in a Prolog KB. Moreover, Prolog is a logic programming language, this means that the core of programming is not tell to the computer what to do, but telling it what is true and asking it to try and draw conclusion[18]. The idea behind logic programming are very interesting, for more details [20] and [21] are recommended.

EXAMPLE: In the following listing, adapted from [19], we present a basic Prolog program in order to show the syntax and the usage of query.

```prolog
sings(mia).
listens2Music(yolanda).
party.

dance(yolanda):- listens2Music(yolanda).
happy(yolanda):- dance(yolanda).
happy(mia):- sings(mia).

smile(X) :- happy(X).
```

Listing 6.1: text

The first three lines are facts, we are asserting that Yolanda is listening to music, Mia is singing and there is a party. 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 rule at line seven can be read as: "If Mia is singing, therefore Mia is happy". Line nine shows the usage of variable, variables starts with uppercase letter and are placeholders for information; we can read this rule as "if someone is happy, he smiles".

We can query our KB asking for example if Yolanda is happy. In SWI-Prolog[22] we can interact with the interpreter and the query we evaluate is: `?- happy(yolanda).` (the full stop is part of the syntax and tell to the interpreter that the query is complete). Prolog will answer `yes.`, this is because Yolanda is listening to music, but if she is listening to music she dances end if she dances she is happy.

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

6.1.3 Feature of QA-Prolog

QA-Prolog doesn't support all the feature of Prolog, but enough to make it possible basic logic programming[14].

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

QA-Prolog support also some feature not presents in the basic version of Prolog, In particular, operations can be performed on variables even before they are ground[14], this mean that QA-Prolog is more powerful in manipulating free variables.

6.1.4 Related works

There are some other attempts to develop a high level programming language for quantum computers, both for quantum gates model and for quantum annealer.

Quantum Prolog [23] 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 it focuses on the mathematical equivalency of relational programming and discrete quantum computing over the field of Booleans [14].

C to D-Wave [24] 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 to a quantum machine instruction to be executed on a quantum annealer.

6.2 PIPELINE

We are now ready to descrobe the pipeline of transformation that bring us from a Prolog program to a Haniltonian $\mathbf{H}_f$.

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 format trough the pipeline and in yellow the software that make the rewrite. Some of these software are made ad hoc for the QA-Prolog pipeline other are also used in very different field.

From a high level point of view the pipeline rewrite the initial knowledge base expressed in Prolog in different objects. The logical meaning of the entities we will build during the pipeline are:

1. Prolog program (KB);

2. Hight level digital circuit in Verilog;

3. Low level digital circuit in EDIF;

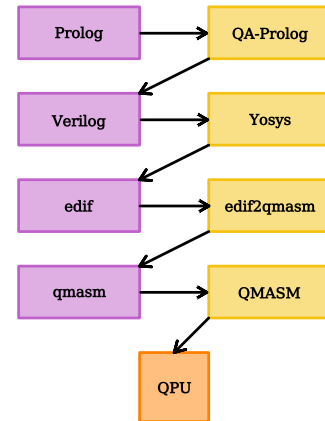4. Symbolic Hamiltonian;

5. Physical Hamiltonian.

Figure 6.1: QA-Prolog pipeline

Let' start analizing the pipeline starting from the last step, the one nearest to the QPU.

6.2.1 QMASM

QMASM is a quantum macro assembler[25], it processes a symbolic Hamiltonian and assemble a physical Hamiltonian that can be immediatly embedded on a D-Wave quantum annealer. It is developed by Scott Pakin with the goal of filling a gap in tools for creation of D-Wave programs. QMASM

is an abstraction layer the allows the programmer to not 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 maps a symbolic representation of the operation (assembly language) to the machine language. QMASM is not only an assebler, but extends his functionalities by including macro: named blocks of assembly language, parameterized, that a program can instantiate multiple times[25].

FEATURE: QMASM provides a number of features to simplify low-level D-Wave programming[25], but not only, we can also use QMASM to assemble a physical Hamiltonian that we can later manipulate or solve both using classical method or other quantum system.

Some of the most useful and interesting feature 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 exlude from the results the solution know to be incorrect;

- the programmer can select the "interesting" *qubits* and these are the only ones reported in the results.

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

EXAMPLE: Let's consider an example that show the potentiality of QMASM and that is useful also for the following steps of the pipeline. The satisfiability problem is well known to be an NP-complete problem[**bibid**], so is a good candidate for our example. We take in account the simple formula:

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

QMASM allow us to define a macro for every logical operator needed to represent the formula and then assemble the final formula calling these macros. In Listings 6.2 we call `A` and `B` the input *qubits* and `Y` the output *qubit*. Wheigts are specified as an Ising problem **??** that is the default format for QMASM source file[3].

There are multiple interesting detail about these macros:

- the sign `#` tells QMASM that the following line is a comment and should not be processed;

---

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

```
# Y = A AND B                                    # Y = A OR B
!begin_macro AND                                 !begin_macro OR
  !assert $Y = $A&$B                               !assert $Y = $A|$B
  $A -0.5              # Y = NOT A                  $A  0.5
  $B -0.5              !begin_macro NOT            $B  0.5
  $Y  1                 !assert $Y = !$A           $Y -1
                        $A $Y 1.0
  $A $B  0.5          !end_macro NOT                $A $B  0.5
  $A $Y -1                                          $A $Y -1
  $B $Y -1                                          $B $Y -1
!end_macro AND                                   !end_macro OR
```

Listing 6.2: Logical operators

- the sign `$` is used to tag a *qubit* as ancillary, this means that, 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 constraint on the solution, this directive does not change the weights, but allow the programmer to exclude from the solutions the ones surely incorrect.

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 minimize the energy corresponds to the output of the logic gate we are modeling.

We can save this macros in a file named `gates.qmasm` and using 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 $x_5 = \neg x_4$ lastly we compute the result as $y = x_1 \wedge x_5$. The QMASM code implementing this procedure is reported in listing 6.3.

This example shows how to use macros: we instance a macro and give it a name with `!use_macro <macro_name> <instance_name>` (es. line five), and then instantiate the *qubits* defined in the macro with our actual *qubits*; for example considering the `!use_macro OR` at line five 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 three) that import all gates used in this source file.

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.

Results are reported in listing 6.4, here we can see that the solver has correctly founded the solution, but only 642 times over the default 1000 sample. This is caused by the stochastic nature of annealing, simulated or

```
1   # Solve a 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

```
1   # x1 -->    12
2   # x2 -->     3
3   # x3 -->     4
4   # y --> [True]
5   Solution #1 (energy = -20.0000, tally = 642):
6
7   Variable  Value
8   --------  -----
9   x1        True
10  x2        False
11  x3        False
12  y         True
```

Listing 6.4: Circuit satisfiability results

quantum; QMASM removed automatically the solution that do not respect the `!assert` directive or do not have the minimum energy.

QMASM offer already some powerful abstraction to work with quantum annealer, now we add another two layer that allow a programming style far to similar with paradigm computer scientists are used to, but preserving a strong control over variable dimension therefore over number of *qubits* used.

6.2.2 Yosys and edif2qmasm

These steps of the pipeline took in input a Verilog HDL (Hardware Description Language) program and transform it in 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 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 languages that offer 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) [26].

Hardware description languages are not something that the majority of programmers are familiar with, but they offer some advantages to target a quantum annealer [27]:

- provide precise control over bit widths in order to not waste any *qubit*;

- can be compiled to a small set of primitives: the gates we can define with QMASM.

In these steps of the pipeline we follow the same procedure used in the design of digital system: we start with using behavioural constructs then we generated automatically a netlist as the output of synthesisers.

The synthesiser used is Yosys [28] the first free and open source software for Verilog HDL synthesis which supports the vast majority of synthesizable Verilog features.

The two most useful features of Yosys are the possibility of specify a *cell library*: the set of gates used to syntetize the netlist; and the use of the external tool Berkeley ABC [29] (incorporated in Yosys) providing additional code optimizations.

Yosys lower a Verilog program to an EDIF netlist that use only a specifyed set of gates, then edif2qmasm lower the netlist into a symbolic Hamiltonian; these Hamiltonian uses a set of macros defining the energy for each type of gate Yosys can use[4]. What we end with is a symbolic Hamiltonian tha 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) in Figure 6.2 we can see in (a) a possible Verilog implementation of the formula, in (b) the netlist Yosys has produced from the Verilog code and in (c), only for clarification purpose, the corresponding digital circuit implemented with logic gates.
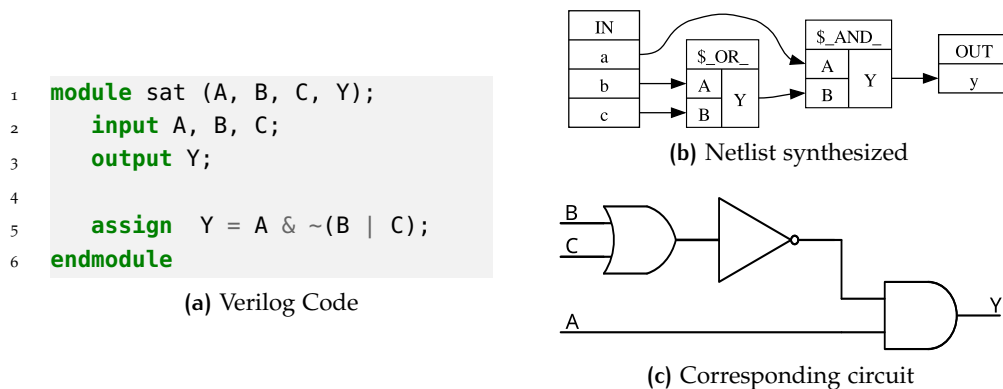
```verilog
1  module sat (A, B, C, Y);
2     input A, B, C;
3     output Y;
4
5     assign  Y = A & ~(B | C);
6  endmodule
```

(a) Verilog Code

(b) Netlist synthesized

(c) Corresponding circuit

Figure 6.2: Yosys processing

---

4 The actual set of gates extend 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 result netlist to edif2qmasm to obtain:

```
1   !include <stdcell>
2
3   !begin_macro sat
4     !use_macro AND $id00006  # $auto$simplemap.cc:83:simplemap_bitop$83
5     !use_macro NOT $id00004  # $auto$simplemap.cc:38:simplemap_not$82
6     !use_macro OR $id00005   # $auto$simplemap.cc:83:simplemap_bitop$81
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: text

Comparing the autamatically 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`, this is because 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, instead we use as input for macros directly the output of other macros.

Even if it is less readable, we can comment the code 6.5. At lines 4, 5 and 6 we declare we need the macro `AND`, `NOT` and `OR`, giving to each macro a name like `$id<number>`[5]. 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 gives the maximum level of abstraction from the hardware that finds the solution to our problem.

QA-Prolog, like edif2qmasm, is written in GO[**bibid**] 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, to 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 like a typical Prolog environment would [14].

---

5 the symbol $ tells QMASM that we are not interested to the value of *qubits* inside the macro

EXAMPLES: The first example we present shows a small knowledge base that can be summed up with "the enemy of my enemy is my friend".

```
1  hates(alice, bob).
2  hates(bob, charlie).
3
4  enemy(A, B) :- hates(A, B).
5  enemy(A, B) :- hates(B, A).
6
7  firend(A, B) :-
8    enemy(A, C),
9    enemy(C, B).
```

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 relation between three people, assert that if a person hates another one they are enemy (lines 4 and 5) and if two people share an enemy they are friends (lines 7, 8, and 9).

We can now query the KB to find if there is two people that are friends, we can do that with:

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

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

Other detail to point out, are the parameter `--work-dir="<path>"` that specify the folder in witch QA-Prolog output all files and the solver: we use a tabu search algoritm because the simulated annealing implemented in Ocean is not capable of finding a solution even in this small scenario (more consideration about that in the conclusion Chapter **??**).

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 rewrited in a Verilog program in Appendix **??** 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 introduce a consistent amount of overhead.

Indeed, we have to implement logical operator defining their truth table, 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.

```
1   and(false, false, false).
2   and(false, true, false).
3   and(true, false, false).
4   and(true, true, true).
5
6   not(false, true).
7   not(true, false).
8
9   or(false, false, false).
10  or(false, true, true).
11  or(true, false, true).
12  or(true, true, true).
13
14  sat(A, B, C, Y) :-
15    or(B, C, X),
16    not(X, Z),
17    and(A, Z, Y).
```

Listing 6.7: text

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 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 describe briefly the update we have done to the project in order to restore the compatibility with Ocean framework and with the other library used. We also provide a small guide to install the tools to be sure that all package work properly and our experiments are reproducible.

6.3.1 Restoring QMASM

The last commit to QA-Prolog has been done in 2019, and the last one to QMASM in 2021. From these years packages and libraries are changed and the compatibility with QA-Prolog pipeline broken.

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

The incompatibility encountered between imports used in QMASM and external library are:

- in the `dwave.cloud` and `hibrid` libraries;

- in the libraries defining D-Wave samplers (calssical solver);

- in `scipy.stats` library.

The documentation available for Ocean[6] and for scipy[7] helped us to resolve those incompatibilities: most of the time the solution consists simply in the correction of 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 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.

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

(a) Verilog Code

```
# hates/2
!begin_macro id00011
  ...
!end_macro id00011
```

(b) Macro definitions

```
# enemies/2
!begin_macro id00010
  !use_macro hates/2 $id00032  # hates_PWYwG/2
  ...
!end_macro id00010
```

(c) Macro usage

Listing 6.8: text

The error can be seen in Listings 6.8: edif2qmasm rewrite the netlist generated from (a) in the macro (b), here `hates/2` is just a comment, the actual name of the macro is `id00011`. In (c), however, we can see that the macro previously defined is called symbolically not with its actual name (lines 4 and 5). This produce obviously an error: the name `hates/2` in the QMASM file have no meaning, is only a comment.

---

6 available at https://docs.dwavequantum.com/en/latest/index.html

7 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: text

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

Core's code of the preprocessor is reported in Listing 6.9. The program sass 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, we can install all software needed and run some experiments.

6.3.3 Installation guide

In order to have a working environment where run our experiments we need to install all software required by the QA-Prolog pipeline. In Chapter 5 we have setted-up a python environment with all the essential tools, to start the installation of QA-Prolog pipeline we need at least the Ocean SDK installed in 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 installing procedure consists in 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, GO allows us to compile edif2qmasm and QA-Prolog. Both Yosys and GO are available as binary package for the majority of GNU/Linux distribution, on Arch[8] they can be installed with:

---

8 https://archlinux.org/

```
1  # pacman -S yosys
2  # pacman -S go
```

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

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

Finally we can install edif2qmasm and QA-Prolog with:

```
1   $ go install github.com/lanl/edif2qmasm@latest
2   $ 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

61

8 | QAOA

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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