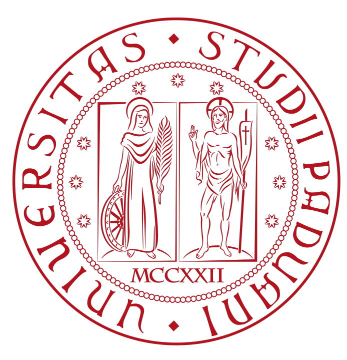


UNIVERSITÀ DEGLI STUDI DI PADOVA

Physics and Astronomy Department “Galileo Galilei”

Master Degree in Physics of Data

Final Dissertation

Programming a Gate-based Quantum Computer: a Comparative Analysis of the Software Development Kits for Circuit Design Automation

Thesis supervisor

Prof. Simone Montangero

Thesis external supervisor

Dr. Davide Corbelletto

Candidate

Andrea Lazzari

Matricola: 2045247

Academic Year 2022/2023

Contents

Foreword **i**

Abstract **iii**

1 Introduction **1**
1.1 Quantum Computing 1
1.2 Limits of Classical Computing 2
1.2.1 Storage and Processing Power 3
1.2.2 Complexity of Algorithms 4
1.3 Principles of Quantum Computing 6
1.3.1 Bit vs. Qubit 6
1.3.2 Superposition and Entanglement 9
1.4 Quantum Computer Paradigms 12
1.5 QC Hardware Overview 14
1.6 QC Applications 15

2 Programming Gate-based Quantum Computers **17**
2.1 Classical Logic Gates 17
2.2 Quantum Gates 18
2.3 Quantum Circuits 21

3 Quantum Application Development **25**
3.1 Quantum Implementation Process 26
3.2 Quantum Programming Languages 26
3.3 Quantum Computing as a Service 27
3.4 Quantum Execution Resources 28
3.4.1 Quantum Processing Units 28
3.4.2 Quantum Simulators 29
3.5 Compilation and Transpilation 30

4 Python SDKs for QC **31**
4.1 Qiskit 32
4.1.1 Syntax and Code Examples 34
4.2 Cirq 39
4.2.1 Syntax and Code Examples 40
4.3 Quantum Development Kit - Q# 46
4.3.1 Syntax and Code Examples 48
4.4 t$|ket\rangle$ 53
4.4.1 Syntax and Code Examples 54
4.5 Quantum Matcha TEA 60
4.5.1 Syntax and Code Examples 61

5 Comparative analysis and Discussion **67**

5.1 Diversity of Quantum Programming Frameworks . . . 67
5.2 Quantum Teleportation Protocol . . . 68
5.2.1 Teleportation Protocol with the analyzed SDKs . . . 71
5.3 Cyclomatic Complexity Analysis . . . 75

6 Conclusions **77**

Bibliography **82**

Foreword

Quantum Computing is not only a matter of conceiving a new kind of hardware but designing completely different algorithms by means of a new kind of software too. Therefore, it is paramount to accurately know all the preeminent libraries, software development kits and programming languages that enable to do so. This work purpose is exactly to explore the variety of options available and compare them highlighting their benefits and drawbacks.

Particularly, the author weigh several solutions to code quantum circuits against each other and came to his conclusions also interviewing and working directly in contact with a quantum algorithm designers and software engineers team. Although this thesis does not claim to be exhaustive (for example, it focuses only on gate-based Quantum Computing SDKs and does not cover the ones dedicated to quantum annealing or measurement-based Quantum Computing), it is not only a good starting point for further detailed studies and extensions but most of all a very useful reference guide for both students and professionals who want to begin in programming a Quantum Computer, taking advantage from the insights it provides.

Dr. Davide Corbelletto

Quantum Technology Specialist at Intesa Sanpaolo

Abstract

The rapid development of gate-based Quantum Computers has opened new possibilities for solving complex computational problems. However, programming these Quantum Computers introduces new challenges due to the fundamental differences between classical and Quantum Computing paradigms. Programming quantum systems requires a new set of tools and programming languages that are specifically designed to deal with the nature of quantum physics. This thesis presents a comparative analysis of Software Development Kits (SDKs) conceived for circuit design automation in gate-based Quantum computers. The objective of this research is to evaluate and compare the capabilities, features, and usability of existing SDKs focusing on the functionalities such as allowing users to define quantum circuits, apply gate operations, and simulate their behaviour.

Apart from the widely adopted frameworks such as Qiskit, TKET, and Cirq, the analysis also includes the recently developed SDK from the University of Padova: Quantum Matcha TEA. The comparative analysis is conducted through a series of experiments and benchmarks performed on each SDK having as central points the programming interfaces usability, the documentation completeness, and the availability of support provided by the vendor or the related developer community. Another goal of this work is to explore the efficiency and flexibility of the various SDKs in handling common Quantum Computing tasks, such as quantum circuit design, gate operation, and circuit execution both on simulators and real quantum hardware.

The ambition of this comparative analysis is to give useful insights to researchers, developers, and practitioners in order to identify strengths and weaknesses of different SDKs depending on the specific requirements of the algorithms that need to be implemented. Additionally, the research aims to contribute to the advancement of SDKs by identifying areas of improvement and potential future directions in the development of quantum programming tools.

Keywords: Quantum Computing, Gate-based Quantum Computing, Software Development Kits (SDKs), Circuit Design Automation

Chapter 1

Introduction

"Future can not be predicted, but futures can be invented."

Dennis Gabor

Quantum Computing (QC) is a different kind of hardware (qubit-based) with respect to classical computing, but more importantly a different kind of software (quantum mechanics based algorithms).

The aim of this Thesis' chapter is to give an introduction to Quantum Computing, starting from basic concepts and emphasizing why and how this new technological paradigm can help to overcome the limitations of a classical approach in solving intractable problems. There are types of problems which in fact are impossible to solve by traditional computers, even by the most powerful ones. This hardness is often expressed in terms of unsatisfactory quality of the solution, for example:

- When one tries to approximate the simulation of a chemical compound, like a drug, and one wants to find the best configuration of atoms that can minimize the energy of the molecule.
- When one attempts to optimize a financial portfolio searching the best solution which maximizes the revenue minimizing at the same time the volatility of the investment.
- When one wants to optimize a logistic chain such as the best route for a delivery service.

In most of the cases these problems are impossible in terms of execution time, for instance:

- The decomposition of an integer number into its prime factors ("unfeasibility" on which is based the current state-of-the-art asymmetric cryptography).
- The daily forecast of a financial scenario when the market is still open.
- When one tries to calculate the optimal route in real time.

So there are many questions: how these limitations can be overcomed? Why a quantum approach can be more efficient than a classical one? This chapter covers the main differences between a classical and a quantum computation, the main principles and paradigms of QC and, at the end, a brief overview of the current QC hardware and software together with their main applications.

1.1 Quantum Computing

The modern incarnation of computer science was announced by the great mathematician Alan Turing in a remarkable 1936 paper [1]. Turing developed in detail the notion of what nowadays is called programmable computer: a model of computation now known as the Turing machine [2]. Furthermore,

he claimed the concept of "Universal Turing Machine" which provided a theoretical framework for understanding computation, in particular what it means to perform a task in an algorithmic way of thinking.

The invention of the transistor in the late 1940s revolutionized computing by enabling smaller, faster, and more reliable electronic components [3]. In the 1950s and 1960s, the field of computer science further expanded with the introduction of high-level programming languages like FORTRAN and COBOL, as well as the development of operating systems [4]. Advancements in integrated circuits and microprocessors in the 1970s and 1980s enabled the miniaturization and widespread adoption of computers [5], leading to the personal computer revolution and, in the late of 20th century, to the advent of Internet and World Wide Web [6]. These incredible milestones drove the development of the modern computing era, connecting computers worldwide and revolutionizing communication, commerce, and information sharing. In the 1990s and 2000s, researchers began exploring the concept of Quantum Computing, leveraging the principles of quantum mechanics to perform computations. Quantum computing is a beautiful fusion of quantum physics with computer science. It incorporates some of the most stunning ideas of physics from the twentieth century into an entirely new way of thinking about computation [7].

Below is an overview of the key reasons why QC is so important and why it is considered to be one of the next technological revolutions. Quantum Computers, by harnessing the principles of quantum physics, provide the ability to simultaneously represent and manipulate vast amounts of information. Moreover, we have good reason to believe that a Quantum Computer would be able to efficiently simulate any process that occurs in Nature [8]. With QC one should be able to examine more deeply the properties of complex molecules and exotic materials, and also to explore fundamental physics in new ways, for example by simulating the properties of elementary particles, or even *reproducing* the quantum behaviour of a black hole. However, it remains a great challenge for future physicists and engineers to develop techniques to perform large-scale quantum information processing.

The main reason is that the quantum world is very different from the classical world, and the quantum behaviour of a system is often very difficult to control and manipulate. However, Quantum Computing has the potential to change many fields by solving complex problems faster and more efficiently than classical computers.

1.2 Limits of Classical Computing

Traditional computers are reaching the limits of their computing power: the ever-increasing demands for performance, miniaturization and production are becoming more and more difficult to meet. Moore's law (which can be visually understood in the Fig. 1.1 below) tells us that the number of transistors in a fixed size integrated circuit doubles about every two years.

This statement (true from 1965 to the present) is indeed reaching a point where the size of transistors is so small ($\approx 3nm$) that quantum effects are becoming important, and the classical approximation, which ignores quantum effects, is no longer valid. At the atomic scale, indeed, nature obeys to quantum mechanics laws. Quantum Computing can therefore provide a technological solution to help High Performance Computers (HPC, the most powerful classical devices) to solve some specific problems and, at the same time, to the challenge posed by the eventual failure of Moore's law.

Quantum Computers indeed offer a speed advantage over classical computers. For example, the Shor [9] and the Grover [10] algorithms give exponential and quadratic speed-up respectively over classical solutions. This speed advantage is so significant that many researchers believe that no conceivable amount of progress in classical computation would be able to overcome the gap between the power of a classical computer and the power of a quantum computer [2].

The first place where this *quantum speed advantage* is being tested is in the simulation of quantum systems. Simulating complex systems without considering quantum correlation effects, can require computational power that grows exponentially with the size of the system, making accurate simulations

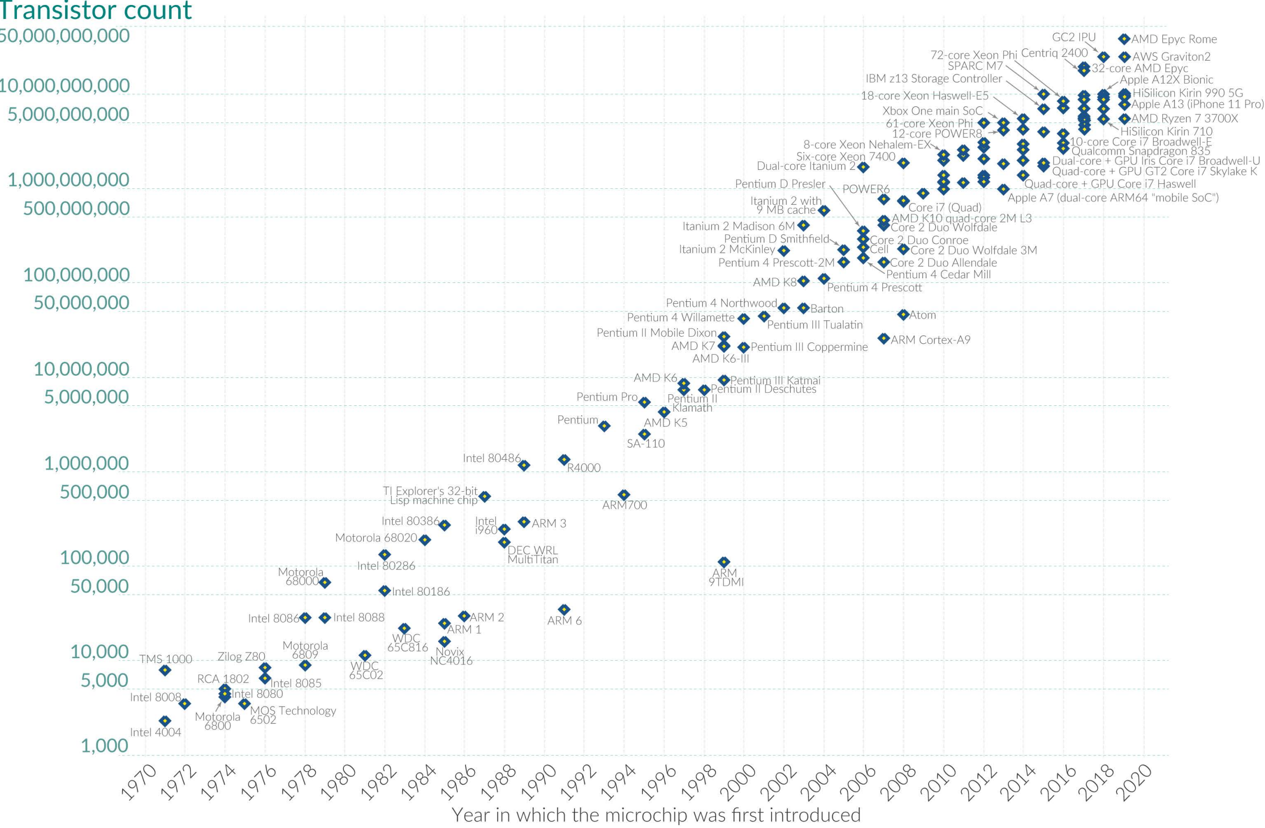


Figure 1.1: Visual representation of the Moore's law: a semi-log plot of transistor counts for microprocessors against dates of introduction, nearly doubling every two years

on traditional HPC difficult or even impossible. Indeed, classical computers are particularly weak at simulating the dynamics of many-particle quantum systems; therefore quantum dynamics is a promising area where Quantum Computers may have a significant advantage over classical ones [8]. In the worst case scenario, storing the quantum state of a system with n distinct components arrives to require c^n bits of memory on a classical computer, where c is a constant which depends upon details of the system being simulated, and the desired accuracy of the simulation. By contrast, a Quantum Computer can perform the simulation using kn qubits, where k is again a constant which depends upon the details of the system being simulated. This allows QC to efficiently perform simulations of quantum mechanical systems.

Achieving faster and more accurate simulations of such systems may therefore have the welcome effect of enabling advances in other areas where quantum phenomena are important. Nevertheless a significant caveat is that, though a Quantum Computer can simulate much more efficiently than a classical computer, this does not mean that a faster simulation will lead to a better result in terms of accuracy [2].It is therefore important to keep in mind that QC is not a panacea for all computational problems, but it is a new powerful tool that requires a deep understanding.

1.2.1 Storage and Processing Power

The two main limitations of traditional computer science are represented by the storage and the processing power.

Even if classical HPC are continuously growing, they will not be able to store all the information required, for example, by simulation problems (e.g., molecule configuration or market forecast). This is also called Space Complexity, which is a measure of the amount of memory that an algorithm needs. The bit is the fundamental unit of classical information. Quantum Computing and quantum information theory, are built upon an analogous concept, the quantum bit, or qubit for short. A more detailed explanation of this concept will be given in the section 1.3.1, for now it is important to

understand that qubits, thanks to quantum properties, lead to more information being processed with respect to classical bits.

The attempt of representing all the possible configurations of a molecule requires the storage of a huge amount of *bits* of information. Why is it important to efficiently represent all the possible configurations of a molecule? If it were possible to fully simulate all chemical configurations, one could study and produce antigens and drugs with 100% efficacy and, most importantly, without side effects. Comparing this number with the number of *qubits* needed to represent the same information, the difference is huge.

Molecule	Chemical Formula	# Bits	# Qubits
Water	H_2O	10^3	10
Ethyl Alcohol	C_2H_6O	10^{12}	40
Caffeine	$C_8H_{10}N_4O_2$	10^{48}	160
Penicillin	$C_{16}H_{18}N_2O_4S$	10^{86}	286

Table 1.1: How many bits are needed vs qubits (*in superposition*) to simulate all the configurations of a specific molecule

The Tab. 1.1 shows the calculations made in [11], considering all the energy configurations given by the orbitals of the atoms in the molecule. One can see that it would take 10^{48} bits to represent all the possible configurations of a molecule of caffeine, which is about the number of atoms in the planet Earth ($\approx 10^{50}$); penicillin even requires 10^{86} bits (approximately the estimated number of atoms in the universe) versus 286 *interconnected* qubits.

The approach taken by Quantum Computing can also help traditional computers improve their processing power by increasing the number of operations a machine can perform in a given time. This is also part of what is known as complexity, which is a measure of the amount of resources, such as memory and time, needed to run an algorithm.

1.2.2 Complexity of Algorithms

Computational complexity theory is the classification of computational tasks' hardness, both classical and quantum. A complexity class can be thought of as a collection of computational problems, all of which share some common features with respect to the computational resources needed to solve those problems [2].

An idea behind commplexity is that one imposes an upper bound on how much resources your computer can use [12]. One can define $TIME(f(n))$ to be the class of problems for which every instance of size n is solvable in an amount of $O(f(n))$ steps. Here "solvable" means tractable in reasonable time by some particular type of idealized computer which is fixed as a reference. Likewise, $SPACE(f(n))$ is the class of problems solvable by our reference machine using an amount of space (i.e. units of memory) that grows like $O(f(n))$.

An important reference to understand the complexity of algorithms is Fig. 1.2 below, which shows the behaviour of the computational time t as the size of the problem n increases.

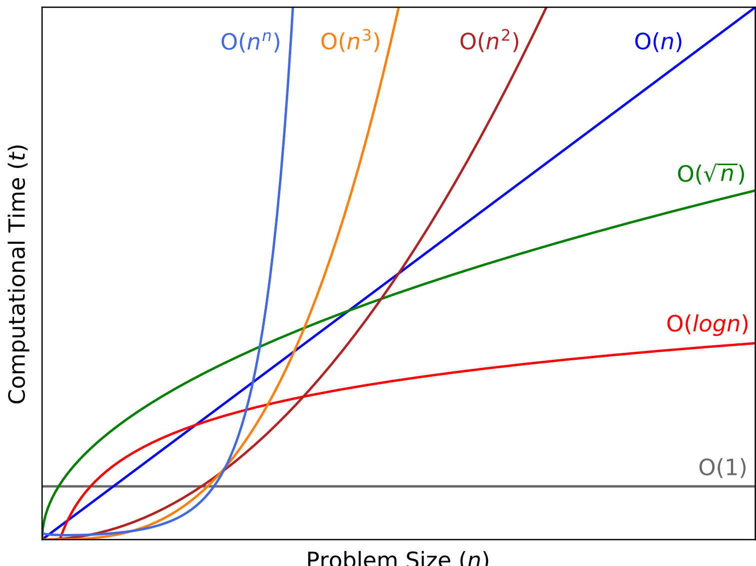


Figure 1.2: Computational time vs. size of the problem

To summarize, complexity looks at how the size of $TIME(f(n))$ and $SPACE(f(n))$ grow as the problem size n grows. In computer science, an algorithm is called efficient if, solving a task, it takes polynomial time or less to execute a certain number of steps. Polynomial time can also mean that the units of memory used by the algorithm scale as a polynomial in n. Problems of this kind belong to complexity class P, which stands for Polynomial-Time. Some problems in P include matchmaking in the "stable marriage problem", determining if a number is prime, and maximizing a linear function constrained by linear inequalities.

On the other hand, an algorithm is inefficient if it takes more than polynomial time, called superpolynomial time. This includes algorithms that take exponential time, such as $O(2^n)$, which are regarded to be infeasible for large n.

Another complexity class is the problems for which a solution can be quickly verified by a computer in polynomial time. This class is called NP (Non-deterministic Polynomial), and it includes problems such as factoring or testing if two networks are equivalent. Certain problems within NP have a special property called completeness, these are called NP-COMPLETE. If one can find an efficient solution to any NP-COMPLETE problem, then it can be used to find an efficient solution to any NP problem since all NP problems can be transformed into an instance of an NP-COMPLETE problem. Some NP-COMPLETE problems includes:

- Finding the shortest possible tour that visits a list of cities exactly once and returns to its starting point, known as the "traveling salesman problem"
- Determining whether a tour that visits each location once and returns to its starting point exists, which is called the "Hamiltonian path problem"

It is known that all problems in P are contained within NP, since if one can efficiently solve a problem, one can also efficiently check proposed solutions by comparing them to the answer. However, it is unknown whether NP contains any problems that are not in P. The general conjecture is that $P \neq NP$. Another complexity class is PSPACE, which contains all the problems that can be solved by a computer using a polynomial amount of memory, without any limits on time. It is known that NP is contained in PSPACE because one has unlimited of time to check all possible answers. Although it seems like PSPACE should be a larger class of problems than NP, there is currently no proof [13]. The BPP complexity class (Bounded-error Probabilistic Polynomial time) contains decision problems that can be efficiently solved by a probabilistic Turing machine in polynomial time with a bounded probability of error. The BPP class covers problems that can be solved with the help of randomness [2]. In simpler

terms, a problem is in the BPP class if there exists an algorithm that can decide the problem with high confidence in a polynomial amount of time.

Focusing on the QC *contributions* to complexity theory, problems efficiently solved by a Quantum Computer are classified in the BQP class. This stands for Bounded-Error Quantum Polynomial-Time. Quantum Computers can efficiently simulate classical computers, so they can effectively solve everything that a classical computer can optimally solve. Moreover, as it stated in [14] BQP $\supset$ BPP.

To summarize, a convenient division of complexity classes includes:

- **P** (Polynomial) class: problems solvable by a Turing machine in polynomial time. In other words, P is the union, over all positive integers k, of $TIME(n^k)$.
- **NP** (Nondeterministic Polynomial) class: problems which have solutions that can be checked in polynomial time.
- **NP-Complete** class: the hardest tasks in NP. In other words, they are the problems in NP whose solutions can be used for any other problem in NP.
- **PSPACE** class: problems solvable in polynomial space (but unlimited time). In other words, it is the union over all integers k of $SPACE(n^k)$.
- **EXP** class: problems solvable in exponential time. In other words, it is the union over all integers k of $TIME(2^{n^k})$.
- **BPP** class: problems that can be solved using randomized algorithms in polynomial time, if a bounded probability of error in the solution is allowed.
- **BQP** (Bounded-Error Quantum Polynomial-Time) class: problems efficiently solved by a Quantum Computer, where a bounded probability of error is allowed.

The relationship between classical and quantum complexity classes is shown in Fig. 1.3 below.

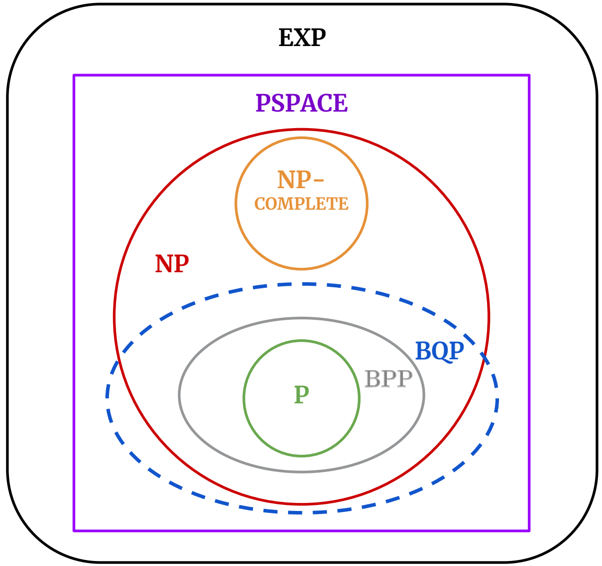


Figure 1.3: Relationship between complexity classes

1.3 Principles of Quantum Computing

1.3.1 Bit vs. Qubit

As a first concept, it is indispensable to understand the differences between a classical bit and a quantum bit.

A bit is just a binary digit, which can be 0 or 1, and it represents the minimal unit of information storage used in classical computation. A qubit is the quantum counterpart of a bit. Unlike a bit, which is limited to one of the two possible states 0 or 1, qubits can take on an infinite number of values that can be expressed as a linear combination of these two states.

While a classical bit corresponds to one of two alternatives (e.g., on/off, black/white, north pole/south pole), a qubit can be represented as a column vector in a Hilbert two-dimensional complex vectorial space. The special states $|0\rangle$ and $|1\rangle$[1] form an orthonormal computational basis for this vector space and the general state of a qubit can be written as a linear combination of these two:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \tag{1.1}$$

Where the coefficients α and β are complex numbers.

$|0\rangle$ and $|1\rangle$ can also be expressed as column vectors:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

With this additional representation, the generic state which a qubit takes on (Eq. (1.1)) can be written as

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

The vector $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ represents $\alpha |0\rangle + \beta |1\rangle$. Here α and β respectively represent the amplitude of outcome $|0\rangle$ and $|1\rangle$.

The bit has the property that it can be measured to determine whether it is in the state 0 or 1. For example, computers do this all the time when they retrieve the contents of their memory. Rather remarkably, one cannot examine a qubit to determine its quantum state, that is, the values of α and β. In quantum mechanics one can only acquire much more restricted information about the quantum state. When a qubit is measured in the Z-basis one gets either the result 0, with probability $|\alpha|^2$, or the result 1, with probability $|\beta|^2$. Since the probabilities must sum to one, the state must be normalized to 1 which means $|\alpha|^2 + |\beta|^2 = 1$.

The following geometric representation is a useful way of thinking about qubits. First of all, one may reparametrize the coefficients of Eq. (1.1) in spherical coordinates as

$$\alpha = r \cdot \cos\left(\frac{\theta}{2}\right) \quad \text{and} \quad \beta = r \cdot e^{i\varphi} \cdot \sin\left(\frac{\theta}{2}\right),$$

where $0 \leq r \leq 1$ is a real number, as well as θ and ϕ, which indicates the Euclidean distance of the statevector to the origin. θ is the polar angle from the positive z-axis and the statevector and φ is the azimuthal angle from the x-axis to the orthogonal projection in the xy-plane of the statevector (Fig. (1.4)).

Qubits describe normalized state so one can assume $r = 1$. Other conditions for θ and φ are $0 \leq \theta \leq \pi$ and $0 \leq \varphi < 2\pi$. Except for a global phase ($e^{i\gamma}$) the state can therefore be written as

$$|\psi\rangle = e^{i\gamma} \left[\cos\left(\frac{\theta}{2}\right) |0\rangle + e^{i\varphi} \cdot \sin\left(\frac{\theta}{2}\right) |1\rangle \right]. \tag{1.2}$$

The global phase given by $e^{i\gamma}$ has no observable effects and for that reason one can effectively write

[1] According to Dirac's notation, a column vector is expressed with '$|\ \rangle$' which is called *ket*

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\cdot\sin\left(\frac{\theta}{2}\right)|1\rangle\,. \tag{1.3}$$

The angles θ and φ define a point on the unit three-dimensional sphere, as shown in Fig. (1.4): the first one is correlated with the amplitude probability while the second represents the phase. This unit sphere is often called the Bloch sphere; it provides a useful means of visualizing the state of a single qubit, and is often used as an excellent testbed for ideas about quantum computation and quantum information.

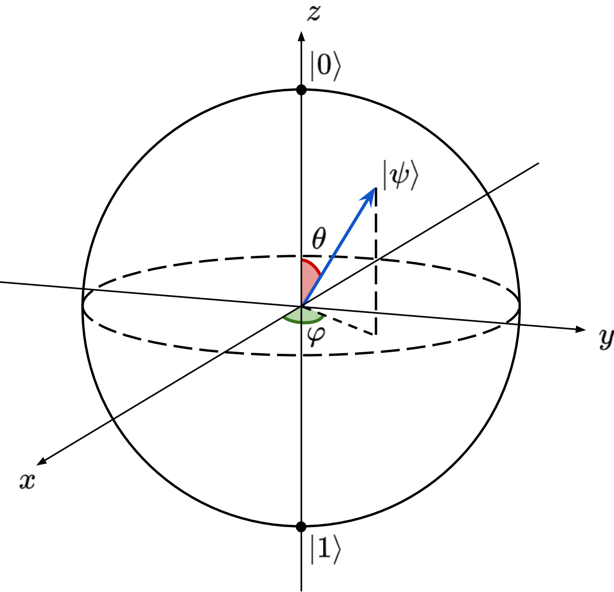


Figure 1.4: Bloch sphere representation of a single qubit state $|\psi\rangle$ which is the vector represented in blue

The set of states a system can occupy is called the state space $\mathbb{S}$ of that system. For classical bits, $\mathbb{S}$ contains just two states, $\mathbb{S} = \{0, 1\}$ whereas a qubit can be in infinitely-many states: the north pole corresponds to $|0\rangle$, the south pole to $|1\rangle$, but in general a qubit can be any point on the Bloch sphere. It is like moving from being able to tell whether a point is in the north or south hemisphere (which is the same as telling if a bit is 0 or 1) to being able to identify the exact position on the Earth, by adding the latitude and longitude coordinates.

Recall that measuring a qubit, e.g. at the end of a computation in order to read the result, gives a single, definite value, only either 0 or 1. Furthermore, the measurement changes the state of a qubit, collapsing it from the linear combination $|\psi\rangle$ to one of the two basis state $|0\rangle$ or $|1\rangle$.A single measurement only provides a single bit of information about the state of the qubit.It turns out that only by measuring an infinite number of identically prepared qubits would one be able to determine α and β for a qubit in the state given by Eq. (1.1). It is fundamental to emphasize, as in [7], that the act of measurement changes the qubit with a permanent loss of information. This phenomenon, known as the *collapse of the quantum state*, occurs for almost all the experiments one might run and it is a probabilistic process.

The study becomes even more interesting when one considers a system that involves many qubits.

When dealing with multiple qubits, the states are written as a tensor product $\otimes$. For example, two qubits, both in the $|0\rangle$ state, are written

$$|0\rangle \otimes |0\rangle = |00\rangle$$

With two quibits the possible states are four (2^2): $\{\,|00\rangle, |01\rangle, |10\rangle, |11\rangle\,\}$. In this case, a general state

can be expressed as

$$c_0 |00\rangle + c_1 |01\rangle + c_2 |10\rangle + c_3 |11\rangle$$

where c_0, c_1, c_2, c_3 are complex numbers.

If a measurement is performed, one gets $|00\rangle$ with probability $|c_0|^2$, $|01\rangle$ with probability $|c_1|^2$, $|10\rangle$ with probability $|c_2|^2$ and $|11\rangle$ with probability $|c_3|^2$. Thus, the total probability is $|c_0|^2 + |c_1|^2 + |c_2|^2 + |c_3|^2$ and it should equal to 1.

More generally, if a system of n qubits is considered, the computational basis states are like $|x_1 x_2 \ldots x_n\rangle$, where $x_i \in \{0, 1\}$. The possibile quantum states are 2^n. For $n = 300$ this number is greater than the estimated number of atoms in the Universe. Trying to store all these complex numbers would be impossible on any classical computer. To quote [2], *Hilbert space is indeed a big place.*

Finally, to summarize the differences between a bit and a qubit, as in [15], a classical bit:

- admits two states, labelled 0 and 1,
- can be freely read

With 'can be freely read' is that one can read the state of any bit in a computer's memory without any kind of obstruction and without changing that state. The qubit instead:

- admits an entire space of states, which can be represented as the Bloch Sphere
- can only be subjected to *rotations* of that sphere,
- can only be accessed through quantum measurements, having as a consequence the *collapse of the quantum state*

1.3.2 Superposition and Entanglement

This section explores the concept of superposition and introduces entanglement: the two fundamental properties of quantum mechanics.

For any isolated region of the universe that one wants to consider, quantum mechanics describes the evolution in time of the state of that region, which can be represented as a linear combination of all the possible configurations of elementary particles in that region. This linear combination is called superposition of states. As shown in the previous section, a qubit can be in a superposition of the states $|0\rangle$ and $|1\rangle$, which means that it can be in a state $|\psi\rangle$ that is a probabilistic combination of $|0\rangle$ and $|1\rangle$. This is the ability of a quantum system to be in multiple states simultaneously until it is measured.

To understand this concept, let us consider the following example. Suppose one has a coin and wants to toss it. In the classical world, the coin can be either heads or tails, but not both. In the quantum world, the coin can be in a superposition of heads and tails, that is, it can be in a state $|\psi\rangle$ that is a linear combination of the two sides of the coin. It is as if one were observing the coin spinning in the air, with the possibility of working with a state that is a probabilistic combination of both heads and tails. Only after performing a measurement operation on the coin, it will collapse to either heads or tails, but not both. It is like when the coin lands on the ground and one observes it.

For example, an equiprobable combination of $|0\rangle$ and $|1\rangle$ can be written mathematically as

$$\frac{1}{\sqrt{2}}\big(\,|0\rangle + |1\rangle\,\big) \quad \text{or} \quad \frac{1}{\sqrt{2}}\big(\,|0\rangle - |1\rangle\,\big) \tag{1.4}$$

In these states, the coefficient of $|0\rangle$ are $\frac{1}{\sqrt{2}}$ and the coefficient of $|1\rangle$ also are $\pm\frac{1}{\sqrt{2}}$, so they are equally probable[1]. These particular states are called *plus state* and *minus state* respectively, often denoted by $|+\rangle$ and $|-\rangle$. Given this, if one represents them geometrically on the Bloch Sphere as in Fig. (1.5), being the two vectors on the equator (x-axis), the couple $\{|+\rangle\,,|-\rangle\}$ is also called X-basis since they are the eighenvectors of σ_x.

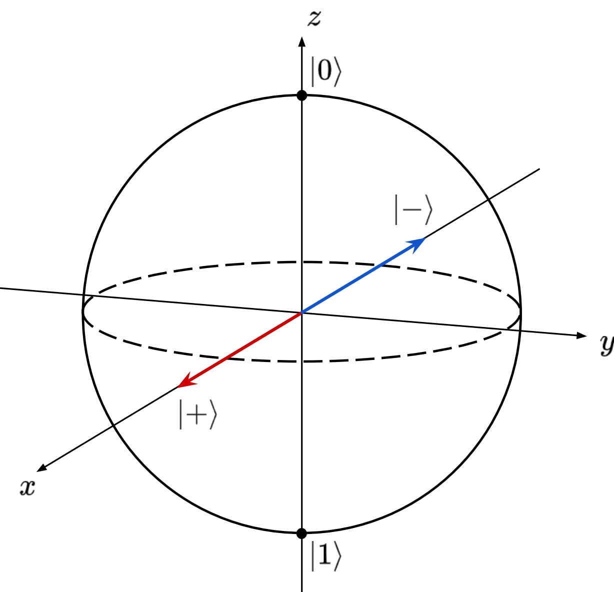


Figure 1.5: Bloch sphere representation of the plus state $|+\rangle$ and the minus state $|-\rangle$

The superposition principle enables QC to process and manipulate vast amounts of information simultaneously. As in Fig. (1.6), the representable information grows exponentially. With 3 Qubits one can express $2^3 = 8$ states in a coherent superposition, gaining an advantage in the amount of information that can be managed with respect to the classical paradigm.

[1] Remembering that the outcome probability is equal to the square modulus of the amplitude

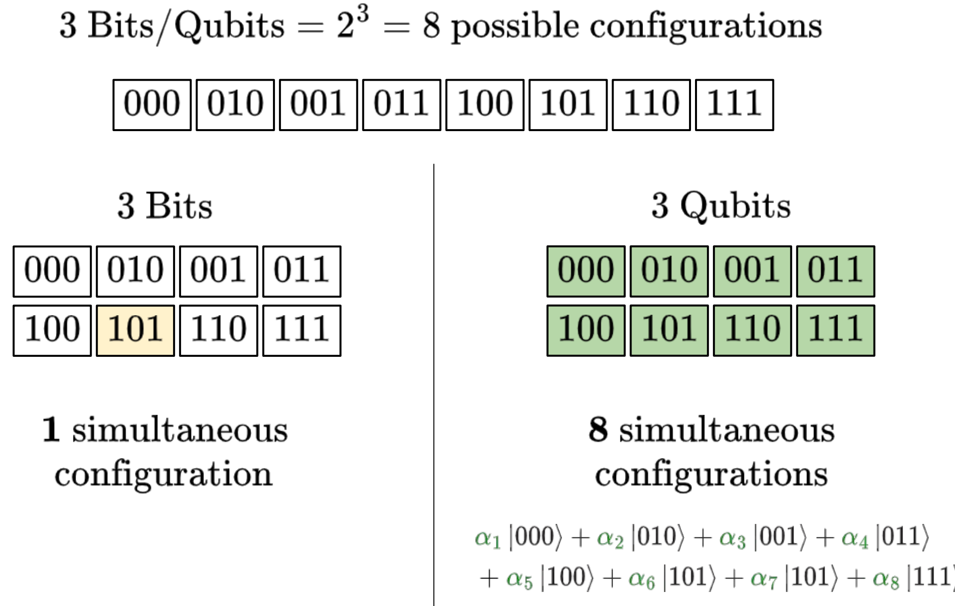


Figure 1.6: Analysis of the expressible information classicaly with 3 bits and quantumly with 3 qubits

The other fundamental principle of quantum mechanics is entanglement, which plays a key role in all the applications of QC.

Entanglement is the term used to describe the quantum correlations between the parts of a quantum system, which are quite different from the classical correlations. To understand the concept of entanglement, imagine a system with many parts, for example a book which is 100 pages long. For an ordinary classical 100-page book, every time you read another page you learn an additional 1% of the book's content, and after you have read all of the pages, one by one, you know everything that is in the book. Classical (not entangled) systems are like this, the information they contain is additive. Now suppose instead you are dealing with a *quantum book*, where the pages are very highly entangled, i.e. very interconnected with one another. Information in such a quantum book does not follow the order in which is imprinted on the individual pages. Rather, it is almost entirely encoded in how the pages are correlated with each other. Therefore, if one wanted to read the book, it would be needed a collective observation on many (if not all) pages at once [8]. This is the essence of quantum entanglement, the unique feature that makes information carried by quantum systems very different from information processed by ordinary digital computers.

To understand the property of entanglement, let us return to quantum state vectors expressed in Dirac notation. Some quantum states can be tensorably factored into individual qubit states. For example,

$$\begin{aligned}\frac{1}{2}\big(\,|00\rangle - |01\rangle + |10\rangle - |11\rangle\,\big) &= \underbrace{\frac{1}{\sqrt{2}}\big(\,|0\rangle + |1\rangle\,\big)}_{|+\rangle} \otimes \underbrace{\frac{1}{\sqrt{2}}\big(\,|0\rangle - |1\rangle\,\big)}_{|-\rangle} \\ &= |+\rangle \otimes |-\rangle\end{aligned}$$

Such factorizable states are called *product states* or *separable states*. In the example above, each single-qubit state can be visualized on the Bloch sphere, so $|+\rangle\,|-\rangle$ would be two Bloch spheres, with the first on the x-axis, and the other on the $-x$-axis as follows in Figure (1.7)

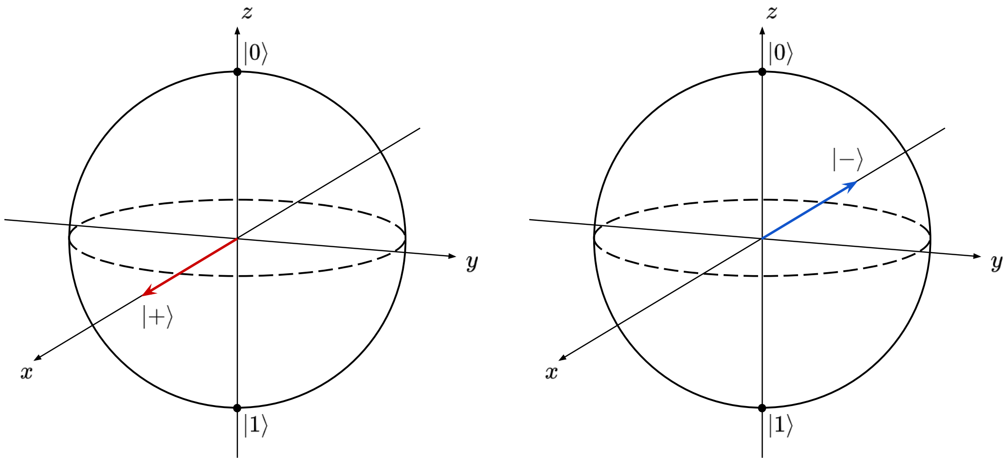


Figure 1.7: Bloch sphere representation of the two qubits in the state $|+\rangle\,|-\rangle$ after being separated

However, there exist quantum states that cannot be factored into product states. For example, with two qubits,

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}\big(\,|00\rangle + |11\rangle\,\big)$$

Such states are called *entangled states* since the state of the qubits are intertwined one with each other.

In a product state, measuring one qubit cannot affect the others, while in an entangled state, measuring one qubit can affect the other qubits. For example, consider the entangled state $|\phi^+\rangle = \frac{1}{\sqrt{2}}\big(\,|00\rangle + |11\rangle\,\big)$. If one measures the left qubit, one gets $|0\rangle$ or $|1\rangle$, each with probability $\frac{1}{2}$, and the state collapses to $|00\rangle$ or $|11\rangle$, respectively. So, if one measures the left qubit and get $|0\rangle$, one knows that the right qubit is also in the state $|0\rangle$, and similarly, if one measures the left qubit and get $|1\rangle$, one knows that the right qubit is also in the state $|1\rangle$.

As in Fig. (1.8), a measurement of one part of a system affects the other.

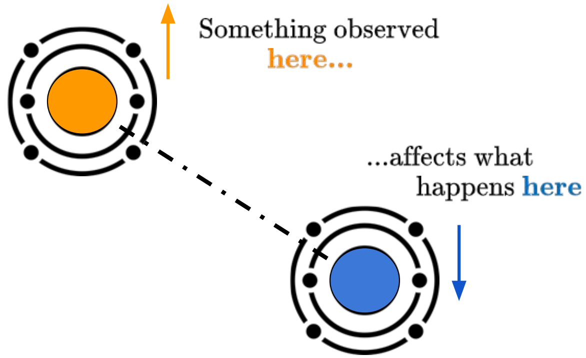


Figure 1.8: Schematic representation of entanglement

1.4 Quantum Computer Paradigms

When exploring the landscape of Quantum Computing, one has to face with the intriguing challenge of selecting between two primary paradigms: gate-based Quantum Computing, with its circuit design, and quantum annealing, tailored for optimization problems.

- **Gate-Based Quantum Computing**

  Gate-based QC, also known as the circuit model, is the paradigm at the center of this work. It involves the manipulation of individual qubits and entangling them through a series of quantum logic gates to perform computations. In this paradigm, qubits are initialized in a desired state, properly prepared, and then a sequence of quantum gates is applied to change their states. Quantum gates are analogous to logic gates in classical computing except that they performed operations on qubits. They involve operations that will be the focus of the next chapter 2. By combining these gates and applying them to qubits in a controlled manner, gate-based Quantum Computers can perform various quantum algorithms and computations.

- **Quantum Annealing**

  Quantum annealing is a different approach to Quantum Computing that is particularly suited for optimization problems. It focuses on finding the global minimum or maximum of an objective function, known as the energy landscape, by exploiting quantum effects. In this approach, qubits are used to represent variables of the optimization problem, and the goal is to find the lowest energy configuration of the system that corresponds to the optimal solution.

  The system is prepared in a superposition of all possible states, and quantum fluctuations are used to explore the energy landscape and find the solution. The annealing process involves gradually decreasing the energy of the system to steer it towards the low-energy states. The system evolves over time and ideally settles into a state corresponding to the optimal solution where energy is minimized.

  For example, suppose one has the graph given in Fig. (1.9) and want to find the lowest point—the absolute minimum. Think of the graph as being the bottom of a two-dimensional bucket. The ball is drop into the bucket. It will settle at the bottom of one of the valleys. These are labeled A, B, and C in the figure. One wants to find C.

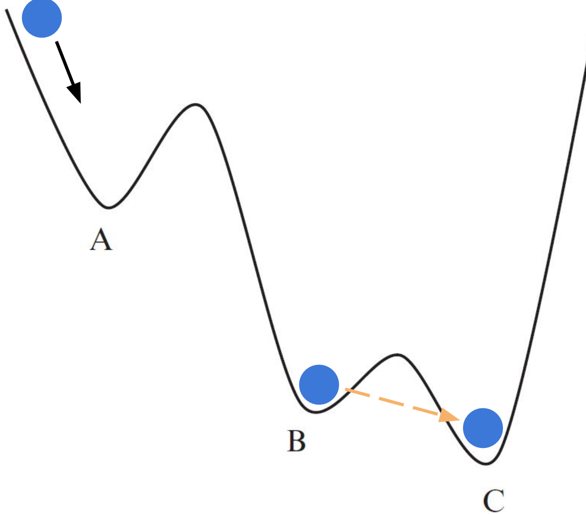


Figure 1.9: Example of an energy landscape of an optimization problem

  Quantum annealing adds quantum tunneling. This is a quantum effect (with no classical analogue) where *the ball* is able to overcome a hill but, instead of going over it, it can go through. This happens since the particle is also a wave and can pass through the barrier even if it does not have enough energy to overcome it.

It is important to note that gate-based and quantum annealing are not mutually exclusive. They represent two different approaches to harnessing the power of quantum mechanics for computation. Gate-based quantum computing offers more general-purpose computational capabilities, while quantum annealing is focused on efficiently solving optimization problems.

1.5 QC Hardware Overview

Currently we are in the NISQ (Noisy Intermediate-Scale Quantum) era, which is characterized by quantum devices that have a number of qubits typically ranging from tens to a few hundreds [8]. Quantum Computers are not yet powerful enough to outperform classical computers for a wide range of practical problems, primarily due to the presence of errors and noise. These errors arise from various sources, such as qubit decoherence, imperfect gate operations, and limited qubit connectivity.

The hardware scene of Quantum Computing is currently heterogeneous, meaning that there are different types of QC platforms with different characteristics at the technical level, but also in terms of the underlying physical principles. This heterogeneity arises because there are different approaches to implementing and manipulating qubits, the fundamental building blocks of quantum computation. Each type of framework has its own unique characteristics, architectures, and limitations.

These differences pose several challenges for the development, standardization, and practical implementation of Quantum Computing technology. Regarding gate-based QC, the hardware includes various platforms such as superconducting, trapped ion, photonic, and neutral atoms. There are also other approaches such as topological qubits [16], silicon quantum dots [17] and diamond vacancies [18]. Each architecture has its own strengths and weaknesses, and the choice of a particular framework is usually application dependent. Each Quantum Computing hardware implementation has its own performance metrics that are decisive in determining computational power, reliability, and scalability, such as coherence times, gate fidelities, and qubit connectivity. Due to this lack of a standardized set of metrics, comparisons and benchmarks of the different platforms performances are currently non-trivial. The following is a brief overview of the main QC hardware platforms, with the key features, advantages and drawbacks of each one.

- **Superconducting** [19]

  It is based on circuits made of superconducting materials that exhibit quantum behavior at extremely low temperatures, close to the absolute zero. Superconductive hardwares involve things called Cooper pairs and Josephson junctions. The electrons in a superconductor pair up, forming what are called Cooper pairs. These pairs of electrons act like individual particles. If one sandwiches thin layers of a superconductor between thin layers of an insulator, one obtains a Josephson junction. These junctions are now used in physics and engineering to create sensitive instruments for measuring magnetic fields. The energy levels of the Cooper pairs in a superconducting loop that contains a Josephson junction are discrete and can be used to encode qubits.

  - Advantages: extremely fast gate operations and good scalability.
  - Drawbacks: the cryostat must be maintained at working temperatures around 0 K; huge sensibility to the environment, i.e. reduced coherence that leads to a degradation of the information contained in the qubits

- **Trapped ion** [20]

  Trapped ion architectures use, as qubits, individual ions typically trapped by using an array of lasers or electromagnetic fields. The qubits are typically encoded in the internal energy levels of the ions, and their states can be manipulated using laser pulses.

  - Advantages: extreme stability, long coherence times, high precision in operations and very interconnected qubits.
  - Drawbacks: ususally slow in terms of gate execution time; hard to scale, challenges in trapping and manipulating large ion arrays.

- **Photonic** [21]

  Photonic hardware is based on the use of photons as carriers of quantum information. It exploits

the quantum properties of photons, such as polarization, which can be vertical or horizontal, or a superposition of both. Photons can be generated, manipulated, and detected using optical components.

  - Advantages: high reliability; no special environment setup required; high-speed computations.
  - Drawbacks: non-universal set of gates due to the lack of strong interactions between photons; complex output digitalization.

- **Neutral atoms** [22]

  This technological design uses atoms with no eletric charge as qubits, often manipulated by light beams in the encoding and readout phases. The qubits are typically encoded in the internal energy levels or electronic states of the atoms.

  - Advantages: long coherence times; easily scalable; high connectivity within qubits.
  - Drawbacks: low processing rate; as a working pressure they must be kept at $\approx 10^{-7} Pa$.

All these different technologies are currently being developed by several companies and research groups around the world. They all suffer from the same problems. As it is disclosed on [7], the most serious is *decoherence*: the problem that qubits interact with something from the environment that is not part of the computation. Since qubits are more sensitive to errors than classical bits, small interactions with the environment can move the qubit to a different location inside the Bloch sphere. In practice, decoherence is the biggest obstacle to building large-scale Quantum Computers, because it is very difficult to isolate a qubit from its environment while making it accessible for quantum gates and measurements.

Furthermore, heterogeneity in hardware necessitates platform-specific programming languages, software stacks, and development tools. Developers need to adapt their algorithms and software implementations to the specific hardware they are targeting. This fragmentation can hinder the development of a unified software ecosystem and make it more difficult for researchers and developers to work with different platforms.

1.6 QC Applications

This section explores some of the applications that are benefiting from the impact of Quantum Computing.

- **Simulation**

  Quantum Computers can simulate complex systems more efficiently than classical computers. This has applications in fields such as materials science, drug discovery, and quantum chemistry [23]. The use of these technologies can accelerate drug discovery by simulating the behaviour and interactions of molecules, exploring chemical reactions, understanding protein folding, and designing new materials with specific properties. In addition, QC can be used in fundamental physics simulations such as high-energy physics and condensed matter physics [24]. Furthermore, a quantum approach can be applied to financial modelling, risk analysis, and portfolio optimization [25].

- **Optimization**

  QC can have a significant and valuable contribution to optimization problems in areas such as logistics, supply chain management, financial portfolio optimization, and resource allocation [26]. In this field the paradigm of quantum annealing can be used to find the lowest energy state of a system that corresponds to the optimal solution of a problem [27].

- **Machine Learning and Artificial Intelligence**

  Quantum Computing can enhance Machine Learning and Artificial Intelligence algorithms [28]. Quantum algorithms such as the Quantum Support Vector Machine (QSVM) and the Quantum Neural Network (QNN) have the potential to improve pattern recognition and optimization tasks [29]. Quantum Machine Learning (QML) can also benefit from quantum data clustering, dimensionality reduction, and generative modelling.

- **Cryptography and security**

  Shor's algorithm [9] can efficiently be employed to factorize large numbers, posing a threat to widely used encryption methods such as RSA. Moreover, the Quantum Key Distribution (QKD) protocol [30], which enables secure communication through quantum entanglement, is developing to become a future standard for communication.

These are just some of the applications that are taking advantage of the potential of Quantum Computing. In general, it is important to remember that QC is not a replacement for classical computing, but rather a complementary technology that can be used to solve specific problems more efficiently.

As it is remarked in [2] designing algorithms for Quantum Computers is challenging because designers face two difficult problems not faced in the construction of traditional algorithms. First, our human intuition is rooted in the classical world. If one uses that intuition as an aid to the construction of algorithms, then the algorithmic ideas one comes up with will be classical. To design good quantum algorithms one must *turn off* the classical intuition, using truly quantum effects to achieve the desired goal. In the following sections the fundamental principles of quantum mechanics are presented, with the aim of providing the necessary background to understand the quantum approach to computation.

Chapter 2

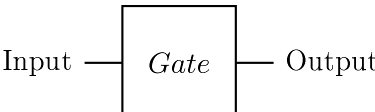
Programming Gate-based Quantum Computers

The history of classical computing teaches us that when hardware becomes available, it stimulates and accelerates the development of new algorithms [8]. For example, theorists eventually explained why the simplex method for linear programming worked well in practice, but only long after it had been found to be useful experimentally. A more recent example can be found in deep learning: there is still a lack for a good theoretical explanation for why it works as well as it does. Gate-based quantum computing is a programming paradigm that allows users to harness the power of quantum systems by manipulating individual qubits and applying quantum logic gates to perform computations. The algorithms developed with this approach involve the design and implementation of quantum circuits, which are sequences of quantum gates applied to qubits to perform specific operations. Quantum circuits are analogous to classical digital circuits, but it is fundamental to understand the difference between the logic gates of the two paradigms.

This chapter introduces the basic concepts of quantum circuits and quantum gates, starting with the classical logic gates and then analyzing how they can be extended to the quantum case.

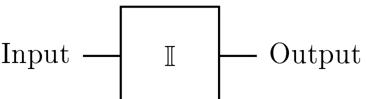
2.1 Classical Logic Gates

Classical computation can be broken down into a series of steps [31], each of which is a logical operation called a logic gate. Classical logic gates are fundamental building blocks of classical digital circuits; they perform logical operations on classical bits, taking one or more input bits and producing an output bit based on predefined truth tables. The simplest logic gates take one bit as input and then output one bit, which one can draw as a circuit diagram:



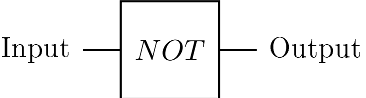
This circuit is read left to right. The input bit on the left travels along the line or wire into the gate, which one has drawn as a generic box. A bit comes out of the gate on the right, traveling along the line, and it is the output. Depending on which gate, the outputs will be different. Some examples of single bit gates are shown below:

- The identity gate does nothing to the bit: how it is shown in the following truth table 0 remains 0, and 1 remains 1



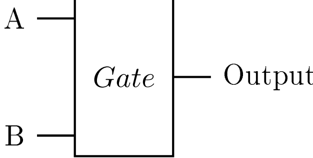
Input	Output
0	0
1	1

- The NOT gate reverses the input: as in the truth table, from 0 to 1 and from 1 to 0



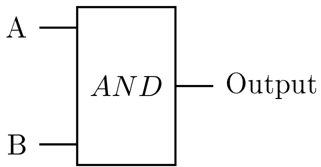
Input	Output
0	1
1	0

A two-bit logic gate takes two bits as input, say A and B. Therefore there will be four possibile inputs, that can be listed in 00, 01, 10, 11.



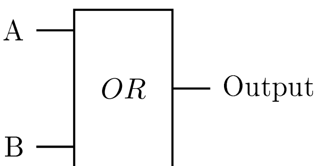
Some examples of two-bit gates are shown below:

- The AND gate outputs 1 if both inputs are 1, and 0 otherwise



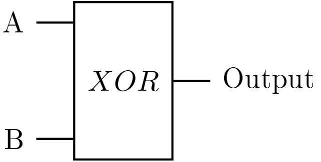
A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

- The OR gate outputs 1 if either input (or both) is 1, and 0 otherwise



A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

- The XOR gate outputs 1 if either input (but not both) is 1, and 0 otherwise



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

These above are the most common logic gates, but there are many others, such as NAND, NOR, XNOR, etc. With these basic ingredients, one can build classical circuits to perform computation.

2.2 Quantum Gates

Classical devices are built using wires and logic gates to perform operations on bits, executing the instructions of a classical algorithm to obtain the desired output. Gate-based QC is based on the same principle, but with the fundamental difference that the logic gates operate on qubits. In fact, changes that occur in a quantum state can be described using the language of quantum computation.

A quantum gate transforms the state of a qubit into another state, and it is often labelled with the capital letter U. The action of a quantum gate on a qubit can be described by a matrix with some specific properties due to the quantum mechanichs behind the computation. First of all, one has to keep in mind that the normalization condition for a quantum state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ requires $|\alpha|^2 + |\beta|^2 = 1$. This condition must also hold for the quantum state $|\psi'\rangle = \alpha'|0\rangle + \beta'|1\rangle$ after the gate has been applied. Therefore a quantum gate must be linear, which means that, to be valid, the overall probability must remain 1.

Consequently, the matrix U representing the action of the gate must be unitary i.e. it has to satisfy the property

$$UU^\dagger = \mathbb{I}$$

where $U^\dagger$ is the conjugate transpose of U.

For this reason, quantum gates must be reversible, which is another aspect of the difference between classical and quantum computation. In fact, not all classical gates are reversible and cannot be used in QC.

A reversible gate, is a logic gate where, given the output(s) of the gate, one can always determine what the input(s) was (were). An example is the NOT gate, where the outputs are unique and it is always possible to reverse the operation. That is, if one knows that the output of the NOT gate is 1, one knows that the input must have been 0, and if one knows that the output is 0, one knows that the input must have been 1. The gate is reversible because, given the output, one can always determine the input.

An irreversible gate is the opposite of a reversible gate. Given the output(s) of the gate, it is not always possible to determine what the input(s) was (were). An example is the AND gate: if the output of the gate is 1, then one knows with certainty that the inputs were both 1. If the output of the gate is 0, however, then it is impossible to know from this information alone which of the other three inputs $(00, 01,$ and $10)$ were used. (see Tab. 2.1) As a general rule, classical reversible logic gates are valid quantum gates, therefore to give some examples:

- The identity gate $\mathbb{I}$ turns $|0\rangle$ into $|0\rangle$ and $|1\rangle$ into $|1\rangle$, hence:

$$\mathbb{I}|0\rangle = |0\rangle \qquad \mathbb{I}|1\rangle = |1\rangle \qquad \mathbb{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- The Pauli X gate, or NOT gate, turns $|0\rangle$ into $|1\rangle$, and $|1\rangle$ into $|0\rangle$:

$$X|0\rangle = |1\rangle \qquad X|1\rangle = |0\rangle \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- The Pauli Y gate turns $|0\rangle$ into $i|1\rangle$, and $|1\rangle$ into $-i|0\rangle$:

$$Y|0\rangle = i|1\rangle \qquad Y|1\rangle = -i|0\rangle \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

- The Pauli Z gate has no effect on $|0\rangle$, while it turns $|1\rangle$ into $-|1\rangle$:

$$\mathbb{Z}|0\rangle = |0\rangle \qquad \boxed{Z} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$
$$\mathbb{Z}|1\rangle = -|1\rangle$$

- The Phase gate S leaves untouched $|0\rangle$ while it turns $|1\rangle$ into $i|1\rangle$:

$$S|0\rangle = |0\rangle \qquad \boxed{S} \qquad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$
$$S|1\rangle = i|1\rangle$$

- The Hadamard gate H turns $|0\rangle$ into $|+\rangle$, and $|1\rangle$ into $|-\rangle$:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle \qquad \boxed{H} \qquad H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

These gates represent just a few examples of the most basic operations that one can perform on a qubit. Some of these gates are equivalent to the classical ones, such as the identity $\mathbb{I}$ and the NOT gate, others are only applicable to quantum systems, since they deal with complex numbers and leverage quantum mechanical principles that have no classical counterpart.

In general one-qubit quantum gates correspond to rotations on the Bloch sphere. As it is displayed in Fig. 2.1 the X, Y, and Z gates correspond to a rotation of π respectively around the x,y and z axes of the Bloch sphere.

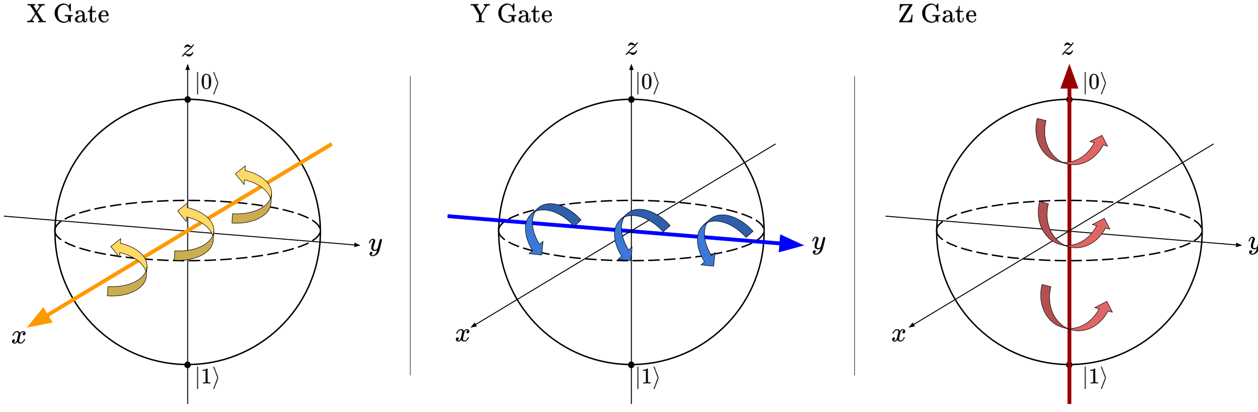


Figure 2.1: Bloch sphere representation of the Pauli gates X, Y, Z

Moreover, as is it shown in Fig. 2.2, the Hadamard gate corresponds to a rotation of π around the $x + z$-axis.

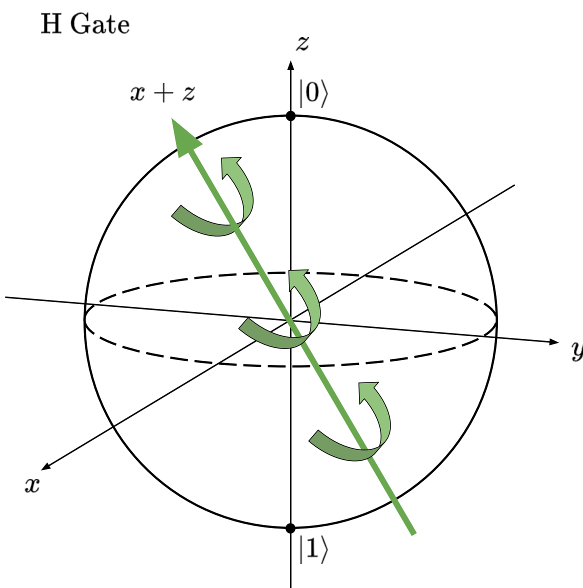


Figure 2.2: Bloch sphere representation of the Hadamard gate

The Hadamard gate is a fundamental quantum gate that plays a crucial role in several Quantum Computing algorithms and protocols. Its usefulness and its uniqueness derive from its ability to create superpositions and perform basis transformations. When applied to a qubit, the Hadamard gate transforms the computational basis states $|0\rangle$ and $|1\rangle$ into equal superpositions of these states.

2.3 Quantum Circuits

Quantum circuits are the quantum analog of classical circuits. They are a sequence of quantum gates applied to qubits to perform specific operations. One can combine these operations to create more complex gates, and can also combine them to create quantum circuits. Quantum circuit diagrams consisting of qubits and quantum gates. They are read left-to-right, just like a classical circuit diagram. The following example starts with a single qubit in the $|0\rangle$ state and apply a Hadamard gate H to it, then a phase gate S, and finally another H gate. It is implied that one measures the qubit at the end of the circuit and can also explicitly draw the measurement as a meter:

$$|0\rangle \; — \boxed{H} — \boxed{S} — \boxed{H} — \boxed{\text{meter}}$$

Quantum gates can also operate on two qubits at the same time. The most important example is the CNOT gate. The Controlled-NOT gate or CNOT gate takes two inputs and gives two outputs. The first input is called the control bit x. If it is $|0\rangle$, then it has no effect on the second bit (called the target y). If the control bit is $|1\rangle$, it acts as a NOT gate on the second bit. The first input bit x has never changed and becomes the first output. The second output equals the second input y if the control bit is $|0\rangle$, but it is flipped when the control bit is $|1\rangle$. To summarize, the CNOT gate inverts the second qubit (the target y) if the first qubit (the control x) is $|1\rangle$:

$$\begin{aligned} CNOT\,|00\rangle &= |00\rangle \\ CNOT\,|01\rangle &= |01\rangle \\ CNOT\,|10\rangle &= |11\rangle \\ CNOT\,|11\rangle &= |10\rangle \end{aligned} \qquad CNOT(x,y) = \begin{cases} (x,y) & \text{if } x = 0 \\ (x,\overline{y}) & \text{if } x = 1 \end{cases}$$

The control qubit is unchanged by CNOT, while the target qubit becomes the XOR (exclusive OR) of the inputs, as also shown in the truth table, Tab. 2.3:

$$CNOT\left|x\right\rangle\left|y\right\rangle = \left|x\right\rangle\left|x\oplus y\right\rangle$$

Input		Output	
x	y	x	x ⊕ y
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Thus, CNOT is a quantum XOR gate and as a matrix, the columns correspond to CNOT acting on $\left|00\right\rangle, \left|01\right\rangle, \left|10\right\rangle, \left|11\right\rangle$ respectively:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$\left|x\right\rangle$ target

$\left|y\right\rangle$ control

Notice that this operation is invertible: given any pair of output values, there is exactly one pair of input values that corresponds to it. The CNOT gate is not just invertible, but it also has the nice property that it is its own inverse. This means that if you put two CNOT gates in series, where the output of the first gate becomes the input of the second gate, the output from the second gate is identical to the input to the first gate. The qubits 0 and 1 correspond to the bits 0 and 1. If one runs the quantum CNOT gate just using the qubits $\left|0\right\rangle$ and $\left|1\right\rangle$, and not any superpositions, then the computation is exactly the same as running a classical CNOT gate with 0 and 1.

This gate is one of the most important in QC since it can be employed combined with the Hadamard Gate H to produce entanglement. For example:

$$\begin{aligned}
CNOT\big(H\left|0\right\rangle \otimes \left|0\right\rangle\big) &= CNOT\big(\left|+\right\rangle \otimes \left|0\right\rangle\big) = CNOT\frac{1}{\sqrt{2}}\big(\left|00\right\rangle + \left|10\right\rangle\big) = \frac{1}{\sqrt{2}}\big(\left|00\right\rangle + \left|11\right\rangle\big) = \left|\phi^+\right\rangle \\
CNOT\big(H\left|0\right\rangle \otimes \left|1\right\rangle\big) &= CNOT\big(\left|+\right\rangle \otimes \left|1\right\rangle\big) = CNOT\frac{1}{\sqrt{2}}\big(\left|01\right\rangle + \left|11\right\rangle\big) = \frac{1}{\sqrt{2}}\big(\left|01\right\rangle + \left|10\right\rangle\big) = \left|\psi^+\right\rangle \\
CNOT\big(H\left|1\right\rangle \otimes \left|0\right\rangle\big) &= CNOT\big(\left|-\right\rangle \otimes \left|0\right\rangle\big) = CNOT\frac{1}{\sqrt{2}}\big(\left|00\right\rangle - \left|10\right\rangle\big) = \frac{1}{\sqrt{2}}\big(\left|00\right\rangle - \left|11\right\rangle\big) = \left|\phi^-\right\rangle \\
CNOT\big(H\left|1\right\rangle \otimes \left|1\right\rangle\big) &= CNOT\big(\left|-\right\rangle \otimes \left|1\right\rangle\big) = CNOT\frac{1}{\sqrt{2}}\big(\left|01\right\rangle - \left|11\right\rangle\big) = \frac{1}{\sqrt{2}}\big(\left|01\right\rangle - \left|10\right\rangle\big) = \left|\psi^-\right\rangle
\end{aligned}$$

These four states $\left|\phi^+\right\rangle, \left|\psi^+\right\rangle, \left|\phi^-\right\rangle, \left|\psi^-\right\rangle$ are called Bell states or EPR states (for Einstein, Podolsky, and Rosen). They form an orthonormal basis and they all are entangled states.

Another relevant quantum gate is the SWAP gate, which simply swap the two qubits:

$$\begin{aligned}
SWAP\left|00\right\rangle &= \left|00\right\rangle \\
SWAP\left|01\right\rangle &= \left|10\right\rangle \\
SWAP\left|10\right\rangle &= \left|01\right\rangle \\
SWAP\left|11\right\rangle &= \left|11\right\rangle
\end{aligned}$$

In other words,

$$SWAP\left|x\right\rangle\left|y\right\rangle = \left|y\right\rangle\left|x\right\rangle$$

This gate cannot produce entanglement because, if the qubits are in a product state, swapping the factors results in a product state. The matrix representation and the circuit diagram of the SWAP gate are following:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$|x\rangle$ $|y\rangle$ $|x\rangle$ $|y\rangle$

$|x\rangle$ $|y\rangle$ $|y\rangle$ $|x\rangle$

A three-qubit gate that often appears in quantum computing is the Toffoli gate, or controlled-controlled-NOT gate, Since it is reversible, it is a quantum gate, and it flips the right qubit (the target) if the left and middle qubits (the controls) are both $|1\rangle$:

$$\begin{aligned} \text{Toffoli } |000\rangle &= |000\rangle \\ \text{Toffoli } |001\rangle &= |001\rangle \\ \text{Toffoli } |010\rangle &= |010\rangle \\ \text{Toffoli } |011\rangle &= |011\rangle \\ \text{Toffoli } |100\rangle &= |100\rangle \\ \text{Toffoli } |101\rangle &= |101\rangle \\ \text{Toffoli } |110\rangle &= |111\rangle \\ \text{Toffoli } |111\rangle &= |110\rangle \end{aligned}$$

Therefore it can be expressed as

$$\text{Toffoli } |a\rangle |b\rangle |c\rangle = |a\rangle |b\rangle |ab \oplus c\rangle$$

Also here it is reported for completeness the matrix form and the circuit representation of the gate:

$$\text{Toffoli } = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

target
control
control

Quantum circuits are constructed by arranging and connecting quantum gates in a specific sequence. The order and connectivity of gates determine the flow of quantum information and the outcome of computations. Designing quantum circuits involves carefully selecting gates, considering qubit connectivity, and optimizing the circuit to achieve the desired computational results. Programming gate-based Quantum Computers and quantum circuits requires a solid understanding of quantum mechanics, quantum algorithms, and quantum gates. As the field advances, it offers exciting opportunities to explore new computational paradigms and solve complex problems that are beyond the reach of classical computing.

As gate-based quantum computing continues to advance, programming techniques and tools are evolving. Researchers and developers are working on optimizing quantum circuits, designing efficient quantum algorithms also thanks to the several Software Development Kits (SDKs) that are available. In line with this theme, the next chapters introduce and compares some of the main frameworks that have been developed developed for programming gate-based Quantum Computers.

Chapter 3

Quantum Application Development

This chapter delves into the pivotal intersection between quantum theory and practical implementation. It addresses the challenges and opportunities presented by translating theoretical concepts into tangible quantum algorithms, applications, and solutions. As the field rapidly advances, bridging the gap between quantum theories and real-world applications has become a primary focus, and for this reason the development of quantum software plays a key role.

However, the development of quantum applications differs significantly from classical application development and currently depends heavily on the hardware used [32]. The quantum paradigm indeed requires a new way of thinking, concerning the implementation process of algorithms and the choice of programming resources to be employed in practice. The implementation of a quantum algorithm can also be done in different programming languages, which require specific compilers and transpilers. As a result, there is a wide range of tools and services available which can be used to develop quantum applications.

Moreover, the choice of the more proper technique for a particular QC program relies on several factors. First, it depends on the quantum hardware being used. On the one hand, Software Development Kits (SDKs) that enable quantum applications to be implemented and run are often tailored to Quantum Computers with a specific design, thereby limiting the execution on such particular hardware. On the other hand, the physical limitations of current quantum architectures play a central role in the implementation of quantum algorithms. Today NISQ's devices have computational results which are not completely accurate, and their size is at an "intermediate scale" in terms of qubits. Nontheless, the most appropriate setting for a particular quantum application development use case requires a fine selection of (i) the quantum cloud service provider, (ii) the quantum hardware used for the execution, (iii) and the implemented quantum algorithm itself. In addition, developer preferences and capabilities, including programming language and available tutorials, also play an important role in deciding which tools to use. Due to the variety of options and the lack of a unified view and characterization, it is not so simple to compare individual tools and services.

Finally, programming languages employed in quantum applications could be considered as a separate category. As the different SDKs, compilers, transpilers, and quantum cloud services support different coding idioms, programming languages are a key factor when considering the compatibilities between different technological settings. In the next sections, after looking at the *quantum workflow* in more detail, all the phases and resources involved in quantum application development will be discussed.

3.1 Quantum Implementation Process

As one may know, programming a computer means telling it to perform certain actions in a specific language that the machine understands, either directly or through the intermediary of an interpreter. A program is a set of instructions that define the behaviour of a computing device and can be summarized in two main steps: control and data [33]. Control here means that the program is built from a set of basic instructions and control structures (i.e. conditionals, jumps, loops) which operate on the input data. In QC, these two steps change. Quantum data is represented by an addressable set of qubits, and the allowed operations to control and manipulate it, are of two types: unitary operations which evolve quantum data, and measurments which are used to inspect their value.

Furthermore, it remains challenging to understand which type of problem can be efficiently solved by which paradigm and corresponding quantum algorithm. A typical quantum algorithm workflow on a gate-based Quantum Computer is shown in Fig. 3.1. The implementation process starts with a high-level problem definition, e.g., 'solve the Travelling Salesman Problem on graph X'. The first step is to choose an appropriate quantum algorithm for the problem at hand. A quantum algorithm is defined as a finite sequence of steps for solving a problem, where each step can be executed on a Quantum Computer. Then, the coder implements the algorithm into a program which is called quantum circuit. This is a sequence of quantum gates that are applied to the qubits in order to solve the problem, for example find the optimal solution in this case. If the scale of the quantum system is still classically simulable, the resulting quantum circuit can be executed directly on a quantum simulator. Otherwise, for the compilation stage a Quantum Processing Unit (QPU) is chosen based on quantum hardware paradigms seen in Sec. 1.5.

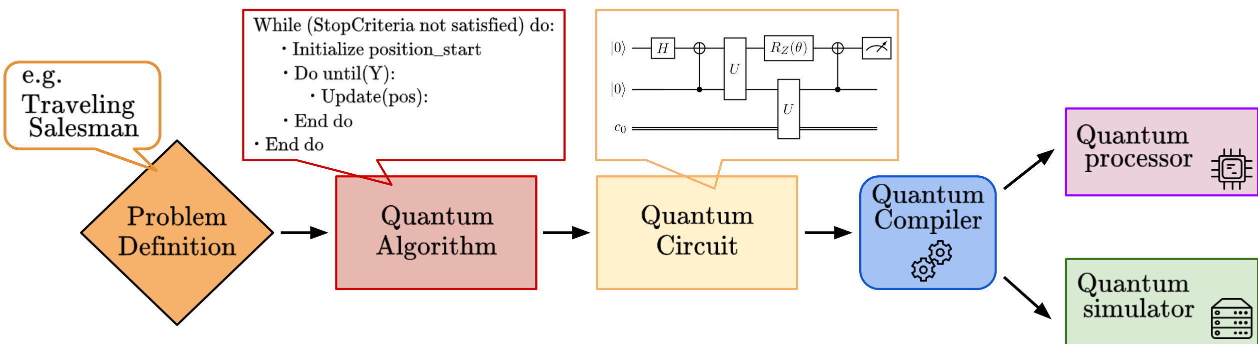


Figure 3.1: Visualization of a typical quantum algorithm workflow on a gate-based Quantum Computer. First the problem is defined at a high-level and an appropriate quantum algorithm is chosen based on the nature of the problem. Next, the quantum algorithm is expressed as a quantum circuit, which has to be compiled to a specific quantum gate set. Finally, the quantum circuit is either executed on a quantum processor or simulated with a quantum simulator

In the gate-model paradigm a full-stack library is defined as software that covers the creation, compilation/embedding, simulation and execution of quantum instructions as illustrated in Fig. 3.1. Open source software in Quantum Computing covers all paradigms and all stages of expressing a quantum algorithm. The software comes in different forms, implemented in different programming languages, each with its own vocabulary, or occasionally even defining a domain-specific programming language.

3.2 Quantum Programming Languages

A Quantum Programming Language (QPL) consists of low-level instructions that specify which gates to perform on which qubits. QPLs are essential for translating ideas into operations that can be executed by a quantum computer [34], facilitating the discovery and development of quantum algorithms, the control of existing physical devices, and even on the design of new ones. The purpose of programming languages is to enable such a communication by representing an idea or concept in a way that allows the reader to visually distinguish and identify the key components and how they fit together. QPLs serve

as a facilitator for clearly expressing and executing quantum algorithms, but also for exploring and developing applications and the hardware to support them. Suitable tools can help analyzing, understanding and mitigating noise in quantum programs, and developing automatic calibration and tuning protocols for quantum devices. Considering the characteristics of available programming languages for quantum applications, the type of language, the syntax implementation, and standardization become relevant [32]. Currently, four types of programming languages can be distinguished:

- Assembly Languages, such as Open-QASM [35] are low-level and provide a textual representation of every operation the Quantum Computer is performing.
- High-level programming languages such as Q# [36] are machine independent and provide features, such as loops and recursion.
- Workflow Languages allow modelling the control flow of (hybrid quantum-classical) applications.
- Graphical Circuit Description Languages provide graphical representations of quantum circuits.

Regarding syntax implementation one can distinguish between two cases. On one hand there are QPLs that have their own independent syntax (Standalone Quantum Programming Languages). On the other hand, a programming language can also be embedded into another programming language, for example, Qiskit [37] and Cirq [38] which provide a convenient set of Python Application Programming Interfaces (API). Finally, the standardization of QPLs is an important aspect. The standardization consist in developing a set of common rules for programming languages.

When a program is sent to a quantum backend, it is first compiled into gates that the computer can actually implement. This compilation is expressed in a lower level language - Quantum Assembly or command language - which is sent to the computer. At the lowest level, the gates are implemented by physical operations on the qubits. These physical operations can be microwave pulses, laser pulses or other interactions acting on a qubit, depending on the physical realization of the architecture. Since it would be very tedious to program this manually, when interfacing Quantum Computers it is better to use a higher-level QPL in a development library. In fact, to avoid the intricacies associated with hardware, QPLs are abstracted into SDKs that provide features to help developers in circuit design automation. All of these languages are tought to be used with a QC framework, either on real hardware on-premises or in the cloud. A fundamental phase in the development of quantum applications after the implementation of the quantum algorithm, it is represented by the execution: the following sections explore the different exploitation resources available for quantum applications.

3.3 Quantum Computing as a Service

The Quantum Cloud Services aspect identifies the layer at which the development tools and services are available and how they can be accessed by users. In general, four different types of *service models* can be distinguished, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Function as a Service (FaaS), and Software as a Service (SaaS).

- IaaS offerings provide access to processing, storage, networking, and other fundamental computation resources that users can use in much the same way as they would on-premises hardware. The difference is that the cloud service provider hosts, manages and maintains the hardware and computing infrastructure in its own data centers.
- PaaS offerings provide an application platform for developing, running, and managing applications. The cloud service provider hosts, manages and maintains all the hardware and software included in the platform.
- FaaS allows users to execute code without managing the complex infrastructure typically associated with building and running applications. In this model, the physical hardware, virtual machine operating system, and web server software management are all handled automatically by the cloud service provider. This allows developers to focus solely on individual functions in their application code.

- SaaS offerings provide users ready-to-use softwares. The application and all of the infrastructure required to deliver it (e.g., servers, storage, networking, etc.) are hosted and managed by the SaaS provider.

In the specific case of Quantum Computing things are slightly different. The lack of a direct access to quantum hardware and the need for specialized expertise to develop quantum applications has led to a different service model, which is a sort of *overlap* of the ones previously listed. In fact, one speaks of Quantum Computing as a Service (QCaaS), where QC providers offer their quantum hardware, software tools, and other resources as a cloud service, enabling researchers, developers, and organizations to harness the power of Quantum Computing without the need to own quantum hardware. Cloud computing is therefore a key enabler for QC, which is *serverless* by nature, as it represents more or less the only way to access quantum resources by now. Major technology companies and research institutions provide QCaaS platforms, contributing to the advancement and democratization of Quantum Computing, and increasing the accessibility to quantum hardware and software, reducing at the same time the overall costs.

In order to access a cloud service, providers offer different access methods: SDKs, are a typical way by which quantum cloud providers offer access to their services. For example, Qiskit and Cirq offer the ability to execute written source code on their respective quantum cloud services. In addition, access can be provided via web services. For example, IBM provides online API for circuit design and execution. IBM Quantum Composer [39] is a graphical quantum programming tool that allows drag and drop operations to build small quantum circuits and run them on real quantum hardware or simulators. On the other hand, in Quantum Lab [40] users can write scripts that combine Qiskit code, equations, visualizations, and narrative text without the need to install any software. Cloud Quantum Computers make it possible to run algorithms on real quantum hardware, and development libraries enable this capability. However, in order to test code before running it on a Quantum Computer, most QC frameworks provide a simulator which runs on a classical computer.

3.4 Quantum Execution Resources

The quantum execution resources aspect considers the characteristics of available resources for the execution of quantum algorithms. Since quantum applications are currently highly dependent on quantum infrastructure and resource availability, developers need to considere this early in the development phase. Two general types of execution are available for running quantum algorithms: Quantum Processing Units (QPUs) and simulators. QPUs represent physical hardware capable to compute quantum programs. Simulators are an alternative, as they simulate quantum programs on classical hardware. As QPUs are still limited, in number and capabilities, simulators are essential for developers to build and test quantum applications before migrating to real hardware.

3.4.1 Quantum Processing Units

A Quantum Processing Unit can be tought as the brain of a Quantum Computer, responsible for performing quantum computations. Unlike classical Central Processing Units (CPUs), which execute sequential instructions on bits, QPUs manipulate qubits and quantum gates to perform quantum algorithms. QPUs are designed to exploit the unique properties of quantum mechanics, which allow qubits to exist in superposition and entanglement states. As shown in Sec. 1.5, QPUs are available in a variety of technologies, such as superconducting qubits, trapped ions, or photonic qubits, each offering advantages but also challenges.

Nowadays one can speak of *Quantum Computing scarcity* that is a concept related to the limited availability of quantum hardware. Since the number of QPUs is limited, the access to them is restricted and the execution of quantum algorithms is subject to a queuing system. At the moment, there are not enough QPUs to meet user demand, so the question is: who gets priority access to them? Fortunately, the availability of QPUs in the cloud or via software is increasing, making Quantum Computing more accessible to researchers, developers, and organizations. Several technology companies and research

institutions offer cloud-based Quantum Computing services, allowing users to access quantum hardware remotely. Here are some notable platforms that provide cloud-based access to QPUs:

- IBM Quantum Experience: IBM provides cloud access to its Quantum Computers through its platform. It allows users to run quantum circuits on real quantum devices.
- Microsoft Azure Quantum: Microsoft's Azure platform provides access to various quantum hardware technologies, including superconducting qubits and trapped ions from vendors such as Quantinuum, IonQ, and Pasqal.
- Amazon Braket: Amazon Web Services (AWS) offers cloud access to Quantum Computing through Amazon Braket. Users can run quantum algorithms on a variety of quantum hardware providers, including Rigetti Computing, IonQ and QuEra.
- Google Quantum Computing Service: Although Google has developed its own quantum processors, it does not provide access to them through its cloud platform. Instead, it allows researchers to run quantum experiments and quantum algorithms by accessing quantum hardware of other providers, such as IonQ and Pasqal.
- D-Wave Leap: D-Wave offers cloud access to its quantum annealing processors, enabling users to solve optimization problems using quantum annealing techniques.

3.4.2 Quantum Simulators

Not all quantum software platforms provide connectivity to real Quantum Computers, but many platforms include a quantum circuit simulator. This is a program that runs on a classical CPU that mimics the evolution of a quantum-based system, as shown in Fig. 3.2. As with quantum hardware, it is important to consider not only how many qubits a simulator can handle, but also how fast it can process them, in addition to other parameters such as adding noise to precisely emulate the behaviour of the current NISQ Quantum Computers.

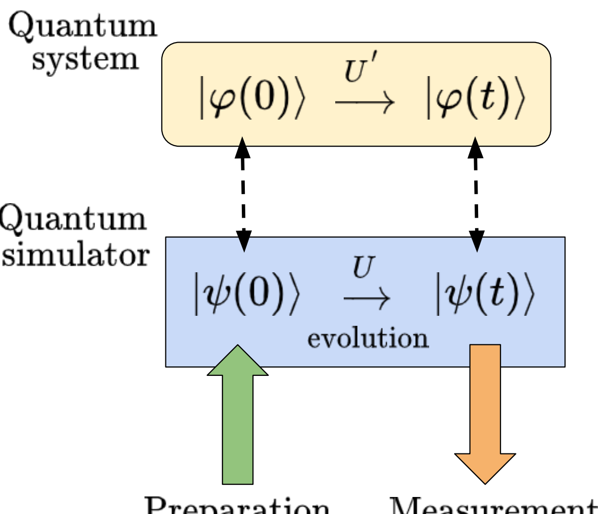


Figure 3.2: Quantum simulators mimic quantum processes. In the preparation step, the initial state is prepared in $|\psi(0)\rangle$, an analogous state with respect to the quantum system $|\varphi(0)\rangle$. Then, applying the U operation to the initial state, the simulator emulates the evolution given by U'. At the end, in the measurement step, the final state $|\psi(t)\rangle$ is measured.

Quantum simulators are powerful tools that allow researchers and developers to explore and test quantum algorithms without the need for physical quantum hardware. These software-based simulations provide a valuable platform for algorithm development, verification, and optimization. Quantum simulators offer several advantages, including the ability to run large-scale quantum circuits more efficiently and at lower cost compared to real QPUs. They also facilitate debugging and fine-tuning of quantum

code, allowing researchers to gain insights into the behaviour of quantum algorithms under different conditions and error scenarios.

Simulations are a valuable tool for validating whether the algorithm and the implementation work as expected. Depending on the application and the quantum algorithm, some simulation tasks can be much simpler. In fact, the ability to analyze or simulate a quantum program on classical hardware is often much more convenient and informative than actually running it on quantum hardware. However, quantum simulators are limited by their classical nature, and as quantum systems grow in complexity, they face challenges in emulating quantum effects at scale. Since they operate on a classical computer, they obviously cannot process actual quantum states, but it is helpful to test the code syntax and the workflow. There are many techniques to classically simulate quantum circuits, but they all suffer from the *"exponential explosion"* of classical memory: to store the most general state of an n qubit system, all 2^n complex numbers of the system's wavefunction must be stored. The problem is that this memory requirement is reaching the limits of today's best supercomputers, even for a modest number of qubits. Despite these limitations, simulators remain indispensable resources in the QC ecosystem, accelerating research and providing a stepping stone in the development of quantum algorithms and applications for future quantum hardware.

3.5 Compilation and Transpilation

An indispensable part of the quantum software stack is represented by the compilation process: quantum compilers and transpilers are essential components of the Quantum Computing ecosystem, enabling the efficient translation and optimization of quantum algorithms for execution on different quantum hardware platforms.

Quantum compilation is the process of translating high-level quantum algorithms, expressed in Quantum Programming Languages, into lower-level quantum instructions that a specific quantum hardware can understand. This crucial step bridges the gap between the abstract mathematical representation of quantum algorithms and the physical constraints of quantum processors. Every Quantum Computer has a basis set of gates and a given connectivity, and the compiler's job is to take a given input circuit and return an equivalent circuit that satisfies that requirements. Compilation actually connects the abstract quantum circuit description to the actual hardware or the simulator: it is the process of mapping the quantum gate set G of a quantum circuit C to another quantum gate set G^*, resulting in a new equivalent quantum circuit C^*. The main reason for this is that QPUs and some simulators are usually only able to implement a limited set of quantum gates that can be used for quantum circuits design.

On the other side, quantum transpilation is a specific aspect of quantum compilation that focuses on mapping quantum circuits from one quantum hardware platform to another. Since different quantum processors differing for qubit connectivity and support different gate sets, quantum transpilation aims to adapt a quantum circuit designed for one type of quantum hardware to run efficiently on a different quantum device. Compilers and transpilers are often included in the quantum SDKs and are used to translate quantum circuits into the gate set of the target quantum hardware. These tools are essential for developers who need to create quantum applications and execute them on real hardware or simulators.

Chapter 4

Python Software Development Kits for Quantum Computing

With the growing interest in Quantum Computing, there is an increasing number of development libraries and tools for the field. Indeed, there is a wide range of environments and frameworks that allow researchers and developers to explore the possibilities of Quantum Computing, in all the languages they are aquainted to including Python, C/C++, Java and others. Many of the leading QC research centers have focused on Python as the language of choice for circuit design automation [41]. One of the reasons for choosing Python is that it is a flexible, high-level language that allows programmers to focus on the problem at hand, rather than on the details of the language. Moreover, Python is well-known and has a large ecosystem of libraries and tools that can be used to build QC applications. A wide variety of different tools, services, and techniques are available for the development of quantum applications. Choosing which SDK/library to use is the very first step that a developer has to take - for both the implementation of classical and quantum applications. However, for the implementation of quantum applications, this decision can also be constrained on which quantum hardware one wants to use.

SDKs, distributed by quantum cloud service providers or by third-party providers, offer advanced developer tools which are the interfaces between the high-level programming language and the low-level assembly language. They can include libraries that provide implementations of algorithms from diverse fields such as chemistry, cryptography, and Machine Learning. SDKs also usually include compilers and transpilers and a local simulator that can be installed locally on a classical computer. However, libraries, compilers, transpilers, and simulators are not necessarily part of an SDK, but can also be available as standalone technologies, such as in TKET [42]. Humans write algorithms in quantum programming languages that can then be applied to solve various problems. In recent years organisations have developed cloud services to access a quantum computer through API or Graphical User Interface (GUI). The user writes a program in one of the QPLs, which is then queued to run on a real quantum computer but, as it is shown in Fig. 4.1, the program can also be simulated on a local or a cloud-based classical computer.

The aim of this chapter is to discuss Software Development Kits for Quantum Computing. As QC becomes increasingly important in various fields of science and technology, Python is emerging as a widely used programming language due to its versatility and extensive ecosystem. The SDKs provide a crucial bridge between the powerful capabilities of quantum hardware and the *convenient nature* of Python programming, enabling researchers and developers to easily explore, simulate, and develop quantum algorithms.

In addition, this chapter provides a comparative overview of the Python SDKs conceived for circuit design automation in gate-based Quantum Computers, including Qiskit, Cirq, and other notable libraries. It highlights the capabilities, features, and usability of existing SDKs, focusing on the functionalities that allow users to define quantum circuits, apply gate operations, and simulate their behaviour. By

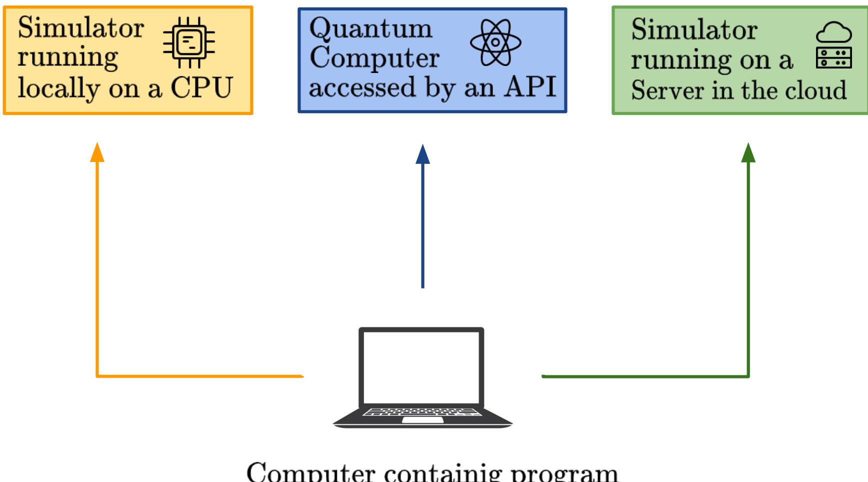


Figure 4.1: Different ways to run a program written in a Quantum Programming Language

gaining insight into these SDKs and their operational characteristics, researchers and developers can start, with more awareness, their Quantum Computing journey towards the development of quantum algorithms and applications.

4.1 Qiskit

Institution / Vendor	IBM
First Release	Mar 2017
Open Source	Yes
Homepage	Homepage [43]
GitHub	Git [44]
Documentation	Online Doc [45]
Supported OS	Mac, Linux, Windows
Classical Host Language	Python
Quantum Language	OpenQASM

Table 4.1: Overview of the Qiskit development library

IBM Quantum Experience is a cloud-based platform that allows the general scientific community to access real Quantum Computers. The Quantum Information Science Kit, or Qiskit for short, is an open source SDK developed by IBM. The Qiskit library is a flexible framework to construct quantum programs and run them on simulators or on real hardware. The Qiskit development library consists of two core modules distributed across the Quantum Computing stack:

- Qiskit Terra: Terra provides core elements and fundamental data structures for composing quantum programs at the level of circuits and pulses, and optimizing them for the constraints of a particular physical quantum processor.
- Qiskit Aer: Aer provides a simulator framework and tools for constructing noise models for performing realistic noisy simulations of the errors that occur during execution on real devices.

Recently, two other core modules (Qiskit Ignis and Qiskit Aqua) have been deprecated. The Qiskit Ignis module, a framework for understanding and mitigating noise in quantum circuits and devices, has been integrated into the Qiskit Terra module. Moreover, the Qiskit Aqua module, a library of cross-domain quantum algorithms, has been divided into separate packages:

- Qiskit Finance: contains components for stock/securities problems. It includes applications such as portfolio optimization, and data providers to source real or random data for finance experiments.
- Qiskit Nature: supports solving quantum mechanical natural science problems using Quantum Computing algorithms. This includes finding ground and excited states of electronic and vibrational structure problems, measuring the dipole moments of molecular systems, and solving the Ising and Fermi-Hubbard models on lattices, among other tasks.
- Qiskit Machine Learning: introduces fundamental computational building blocks - such as Quantum Kernels and Quantum Neural Networks - used in different applications, including classification and regression.
- Qiskit Optimization: enables the representation and modelling of a wide range of optimization problems.

Utilizing some backends, Qiskit cloud services allow users to execute quantum programs in specialized hardware that leverages quantum mechanical phenomena for quantum computation, not only designed by IBM but also by other companies, such as Rigetti, IonQ, IQM and Quantinuum. Qiskit is agnostic with respect to the underlying architecture of a specific quantum system, allowing the compilation of a quantum circuit adapting to the characteristics of a quantum device, mapping the native set of gates and optimizing the resulting circuit. Furthermore, the library of IBM includes several simulators that run locally (both con CPU and GPU) or on cloud computing resources, such as:

- Statevector: It simulates a quantum circuit by computing the wavefunction of the qubit's statevector as gates and instructions are applied. It supports general noise modeling.
  - Qubits: 32
  - Noise modeling: Yes
- Stabilizer: An efficient simulator of Clifford[1] circuits. It can simulate noisy evolution if the noise operators are also Clifford gates.
  - Qubits: 5000
  - Noise modeling: Yes (Clifford only)
- Extended Stabilizer: It approximates the action of a quantum circuit using a ranked-stabilizer decomposition. The number of non-Clifford gates determines the number of stabilizer terms.
  - Qubits: 63
  - Noise modeling: No
- MPS: A tensor-network simulator that uses a Matrix Product State (MPS) representation for the state that is often more efficient for states with weak entanglement.
  - Qubits: 100
  - Noise modeling: No
- QASM: A general-purpose simulator for simulating quantum circuits both ideally and subject to noise modeling. The simulation method is automatically selected based on the input circuits and parameters.
  - Qubits: 32
  - Noise modeling: Yes

[1] In Quantum Computing and quantum information theory, the Clifford gates are the elements of the Clifford group, a set of mathematical transformations which normalize the n-qubit Pauli group, i.e., map tensor products of Pauli matrices to tensor products of Pauli matrices through conjugation

Regarding the available QPU provided by IBM, the following Tab. 4.2 shows the characteristics:

QPU	Max Number of Qubits	Notable Features
Canary	16	Optimized connectivity with a 2D lattice
Falcon	27	Qubits with good coherence properties
Egret	33	Fast and high-fidelity two-qubit gates (approaching 99.9%)
Hummingbird	65	Heavy-hexagonal qubit layout, nice scalability
Eagle	127	Multiple chip layers with high-density I/O without sacrificing performance
Osprey	433	3-layers architechture, more control of noise (Access granted at enterprise level)

Table 4.2: Overview of the Qiskit quantum processors types

According to IBM's roadmap, Condor will be the next quantum processor to be released. It will reach the peak of 1121 qubits. Another key implementation for the growth of quantum hardware will be IBM Heron. It will be characterized by a modular architecture that can be stacked and interconnected to enable parallelized Quantum Computers.

The documentation provides instructions on installing and configuring, including example programs and information about actual quantum devices, project organization, and release notes. Background information on Quantum Computing is also available for those new to the field. The SDK reference is a valuable resource where users can access source code documentation. In addition, Qiskit provides a significant number of tutorial notebooks (Qiskit Textbooks [46]) that cover a broad spectrum of topics, ranging from fundamental quantum algorithms to didactic quantum games.

The IBM library provides a Python-based programming environment that allows one to generate and manipulate OpenQASM programs. OpenQASM is a gate-based intermediate representation for quantum programs. It expresses quantum programs as lists of instructions, often intended to be consumed by a quantum processor without further compilation [34]. OpenQASM allows for abstractions in the form of quantum gates, which can be composed in a hierarchical fashion, based on a set of intrinsic functions that are assumed to be available on the target processor. An example can be a Toffoli gate composed of CNOT gates, and H gates. OpenQASM also supports single-qubit measurement and basic classical control operations. Qiskit SDK leverages Python's abstraction capabilities, such as the ability to synthesize gate decompositions and certain unitary transformations. As an open source SDK, Qiskit has a large community of users and contributors. Anyone can contribute to the project by requesting bug fixes or suggesting new features through GitHub issues. There is also a dedicated Slack channel [47] where the community can discuss or ask for support.

4.1.1 Syntax and Code Examples

When using Qiskit a user workflow nominally consists of following four high-level steps:

1. **Build**: Design a quantum circuit(s) that represents the problem you are considering.
2. **Compile**: Compile circuits for a specific quantum service, e.g. a quantum system or classical simulator.
3. **Run**: Run the compiled circuits on the specified quantum service(s). These services can be cloud-based or local.
4. **Analyze**: Compute summary statistics and visualize the results of the experiments.

Here is an example of the entire workflow, with each step explained in detail below.

```python
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram

# Create a Quantum Circuit with a 2 qubits register and 2 classical bits
circuit = QuantumCircuit(2, 2)

# Add a Hadamard gate on qubit 0
circuit.h(0)

# Add a CX (CNOT) gate on control qubit 0 and target qubit 1
circuit.cx(control_qubit = 0, target_qubit = 1)

# Map the quantum measurement to the classical bits
circuit.measure([0, 1], [0, 1])

# Draw the circuit
circuit.draw("mpl")

# Select the quantum simulator. In this case Aer's QASM_simulator
simulator = AerSimulator()

# Compile the circuit for the support instruction set (basis_gates)
# and topology (coupling_map) of the backend
compiled_circuit = transpile(circuit, simulator)

# Execute the circuit on the Aer simulator for 1000 shots
job = simulator.run(compiled_circuit, shots=1000)

# Grab results from the job
result = job.result()

# Return measurement counts from the results
counts = result.get_counts(compiled_circuit)
print("Total count for states |00> and |11> are:", counts)

# Plot a histogram of the simulation results
plot_histogram(counts)
```

After importing the Qiskit development libraries, the program declares a quantum and classical register with two qubits each, which is then used to create a circuit. Then, the circuit is initialized with a Hadamard gate applied on the first qubit and a CNOT gate where the first qubit acts as control and the second qubit is the target (controlled). Finally, the circuit is measured and the results are stored in the classical registers. In order to simulate this simple circuit, in Qiskit one has to choose a specific simulator and declare a backend for the execution. Furthermore, the circuit is compiled and simulated. At the end, the results are retrieved and the measurement statistics (counts) are printed. Running this code one obtains a random output due to the probabilistic nature of Quantum Computing. In Qiskit, measurement outcomes are stored as dictionaries (a Python data type consisting of key-value pairs) where keys are bit strings and values are the number of times each bit string was measured.

```

"Total count for states |00> and |11> are: {'00': 492, '11': 508}"
```

```
3
```

Qiskit has the ability to draw and save circuits as files in addition to printing out text representations.

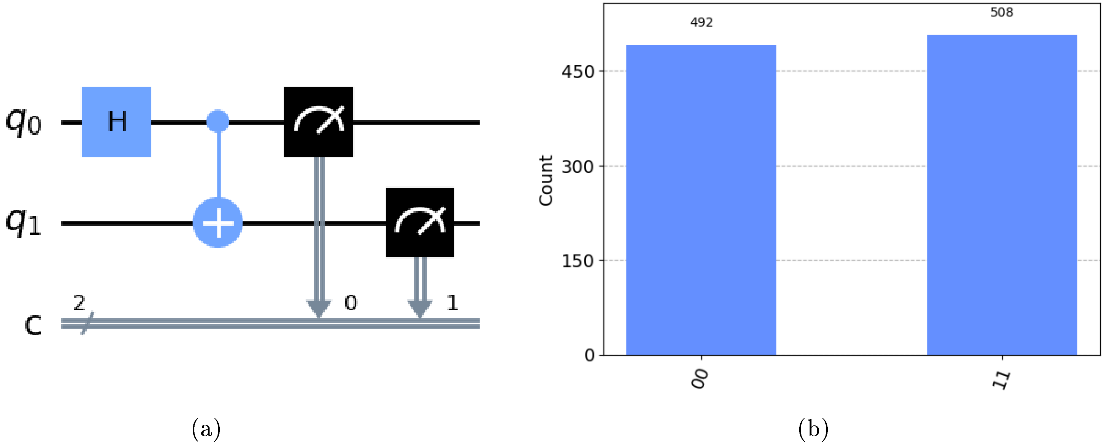


Figure 4.2: Drawing of the circuit synthesized by the code above (a) and histogram of the simulation results of the code above (b)

The program above can be broken down into six stages.

Import Packages

When programming with Python, the first step is to import the packages that are needed, the basic elements of the program are imported as follows:

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
```

More in detail, the imports are:

- `QuantumCircuit` : instructions that hold all the quantum circuit operations.
- `AerSimulator` : a high-performance simulator framework for quantum circuits that includes noise models and backend options.
- `plot_histogram` : a functionn to visualize in a histogram the results of a simulation.

Variables and Gates

```

circuit = QuantumCircuit(2, 2)

```

The function `QuantumCircuit` specifies that the circuit is composed by a register with 2 qubits initialized to the zero state and with 2 classical bits set to zero. At this stage, gates (operations) can be added to manipulate the registers of the quantum circuit.

```
    circuit.h(0)
    circuit.cx(control_qubit = 0, target_qubit = 1)
```

```
3        circuit.measure([0, 1], [0, 1])
```

The gates are added to the circuit one-by-one to form the state

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

The code above applies the following gates:

- `QuantumCircuit.h(0)` : A Hadamard gate H on qubit 0, putting it in a **superposition state**.
- `QuantumCircuit.cx(control_qubit = 0, target_qubit = 1)` : A controlled-NOT operation (CX or CNOT) on control qubit 0 and target qubit 1, putting the qubits into an **entangled state**.
- `QuantumCircuit.measure([0,1], [0,1])` : when passing the entire quantum and classical registers to `measure`, the measurement result of the ith qubit is stored in the ith classical bit.

Circuit Visualization

One can use `qiskit.circuit.QuantumCircuit.draw()` to view the designed circuit in the various forms used in many textbooks and research articles.

```
1
2  circuit.draw("mpl")
3
```

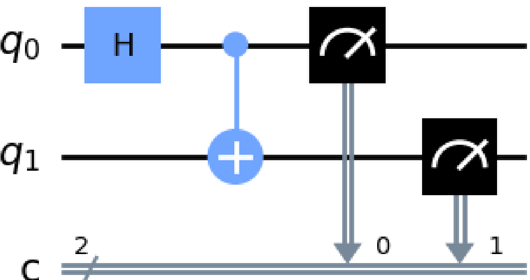


Figure 4.3: Drawing of the designed circuit

In Qiskit qubits are arranged in numerical order, starting from qubit 0 at the top and qubit 1 at the bottom. The circuit is read from left to right, so that gates that are applied earlier will appear farther to the left. The `text` backend is the default for `QuantumCircuit.draw()` . In this case, the `mpl` backend is used to generate a `Matplotlib` figure which is the most commonly used Python visualization library.

Simulation and Results

Qiskit Aer is framework for simulating quantum circuits with high performance. It provides several backends to achieve different simulation objectives. To simulate the circuit, in this case the `AerSimulator` was chosen. Every circuit run will yield either the bit string $|00\rangle$ or $|11\rangle$.

```
1  simulator = AerSimulator()
2  compiled_circuit = transpile(circuit, simulator)è
```

```
job = simulator.run(compiled_circuit, shots=1000)
result = job.result()
counts = result.get_counts(compiled_circuit)
print("Total count for states |00> and |11> are:", counts)
```

As expected, the output bit string $|00\rangle$ appears about 50 % of the time. It is possible to specify the number of times the circuit runs through the use of `shots` argument of the `execute` method. The number of shots of the simulation was set to be 1000 (the default is 1024).

Qiskit offers various visualizations functions, in addition to the histogram for the simulation results one can also plot the statevector on the Bloch sphere to observe the state of the qubits.

```
plot_histogram(counts)
```

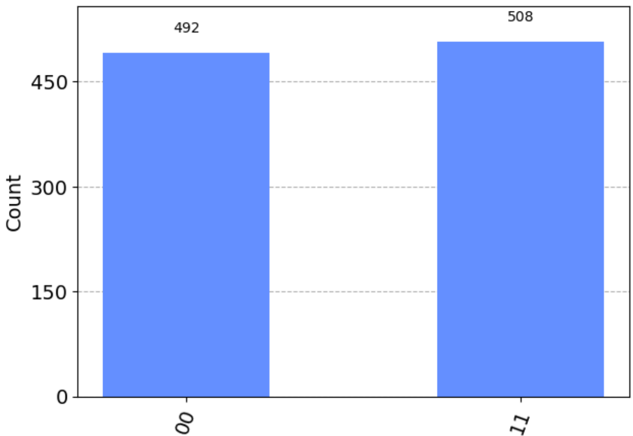


Figure 4.4: Histogram of the simulation results

The observed probabilities $Pr(00)$ and $Pr(11)$ can be computed by taking the respective counts and dividing by the total number of shots.

Qiskit provides a full-stack approach to Quantum Computing, including tools and modules for circuit creation, simulation, error mitigation, and accessibility to real quantum hardware through IBM cloud services. This approach allows users to conduct experiments with quantum algorithms and applications at different levels of abstraction. Additionally, Qiskit is actively maintained and updated, and the official documentation and community forums are excellent resources to learn more and get support.

The choice of the SDK typically depends on factors such as the specific use case, research interests, and hardware compatibility. Nevertheless, Qiskit's design, which is oriented towards accessibility and ease of use for both beginners and experienced quantum researchers, has made it one of the most widely used SDK.

4.2 Cirq

Institution / Vendor	Google
First Release	Apr 2018
Open Source	Yes
Homepage	Homepage [38]
GitHub	Git [48]
Documentation	Online Doc [49]
Supported OS	Mac, Linux, Windows
Classical Host Language	Python
Quantum Language	/

Table 4.3: Overview of the Cirq development library

Cirq is an open source Python library developed by Google for programming Quantum Computers. With the increasing availability of NISQ computers, the development of algorithms to harness the power of these machines is becoming increasingly important. This SDK provides an abstraction for writing, manipulating, and optimizing quantum circuits, and then running them on real hardware or on quantum simulators.

Cirq is composed by three main libraries:

- OpenFermion: library used to compile and analyze quantum algorithms for simulating fermionic systems, which includes quantum chemistry. The package contains everything for translating problems in chemistry and materials science into quantum circuits that can be executed on existing platforms.
- TensorFlow Quantum: library that facilitate the prototyping of hybrid quantum-classical ML models. It integrates QC algorithms and logic designed in Cirq, and provides functions that are compatible with existing TensorFlow APIs, along with high-performance quantum circuit simulators.
- qsim: C++ library for simulating quantum circuits. The module is integrated in Cirq and has the capacity to run simulations of up to 40 qubits.

The initial two modules are adaptations of existing *classical* libraries, intended to provide interoperability within the Cirq framework.

Cirq is not based on a specific Quantum Language, but rather on a Python API. This allows the user to write quantum circuits in a more natural way, taking advantage of the flexibility of Python. Although Google has its own quantum processors, specifically Foxtail (22 qubits), Bristlecone (72 qubits) and Sycamore (53 qubits), they are currently not accessible to commercial users. However, Cirq can be used to run programs on local simulators or via Google cloud on real hardware from third-parties such as IonQ, Rigetti, Pasqal and Alpine Quantum Technologies (AQT).

In terms of simulators, Cirq has several built-in features that also support general noise modelling, such as:

- **State vector simulator**: This calculates the final quantum state of a quantum circuit, representing the amplitudes of all possible computational basis states. It allows users to retrieve the final state of a quantum circuit without performing measurements.
- **Density matrix simulator**: This simulator provides the final density matrix of a quantum circuit, capturing the probabilities of each computational basis state. This is useful for simulating quantum circuits with measurements.
- **Clifford simulator**: An efficient simulator of Clifford circuits. It can also simulate noisy evolution when the operators are Clifford gates.

- **Monte Carlo simulator**: A Monte Carlo simulation tool that estimates the output probabilities by repeatedly sampling measurement outcomes.

- **Simulator**: The basic simulator in Cirq, based on sparse matrix statevector methods using NumPy.

Additionally, in the Cirq SDK it is possible to use Quantum Virtual Machine (QVM). It is a virtual Google quantum processor that allows circuits to be run by using a virtual engine interface. Behind this interface, it uses simulation with noise data to mimic Google quantum hardware processors with high accuracy: in internal tests, the virtual and actual hardware are within experimental error of each other [38]. It also supports internal use of the high-performance qsim simulator, for fast execution of larger circuits. In this case, circuits can then be simulated using either `QSimSimulator` or `QSimhSimulator`, depending on the desired output. The first is a Schrödinger full-statevector simulator and essentially performs repeated matrix-vector multiplications. One matrix-vector multiplication corresponds to the application of one gate. It is suitable for acquiring the complete state of a reasonably-sized circuit (≈ 25 up to 40 qubits). The second consists in a different approach, it is a hybrid Schrödinger-Feynman simulator and raises its upper bound on the number of simulated qubits (50+ qubits, depending on depth) at the cost of being limited to return amplitudes for certain output bitstrings.

Detailed documentation for Cirq is available online, including an installation guide and a tutorial to help new users familiarize with Cirq. In-depth sections are also available for more detailed descriptions of Cirq's features, including a complete API reference. As Cirq is a Python framework, it can also take advantage of the wide range of tools available to facilitate the development of quantum applications. Furthermore, the Google team offers the opportunity to partecipate in weekly organized meetings to discuss about the development of the SDK. As with any open source SDK, Cirq receives contributions from the community and is supported through the GitHub repository and the Stack Exchange forum [50]. Notable features of Cirq include built-in utilities for optimizing quantum circuits by reducing the number of gates, automatic hardware-specific compilation, and several useful tools for managing noise and working with variational hybrid algorithms. An interesting functionality is the ability to simulate a "sweep" of the parameters, i.e. to try a particular set of angles in parameterized gates. This simplifies and can speed-up the optimization process. Moreover, Cirq allows programmers to define schedules and devices that operate at the lowest level of algorithm execution, for example, specifying the duration of physical pulses and gates.

4.2.1 Syntax and Code Examples

Unlike other languages where qubits can be dinamically allocated, Cirq layout is performed manually. Qubits can only be arranged by specifying their position (e.g., row and column for a `GridQubit`) or some other globally unique identifier (for instance, a string for a `NamedQubit`). This means that programmers must decide which physical qubits to use for each part of an algorithm, but as a result they have more control over how a NISQ computer's limited number of qubits are used. In addition, programmers have several options when it comes to scheduling each quantum operation. A circuit in Cirq is divided into 'moments', which are discrete units of time in which all operations at the same moment are performed simultaneously. Only a single operation can affect a particular qubit at any given moment. When operations are added to the circuit, they can be added as part of a new moment (increasing the total length of time of the program), or instead can 'slide' back to an earlier moment, if the affected qubits are not already in use at that time. Since Cirq is embedded in Python, it is easy to manipulate circuits as their division into moments makes them behave similarly to other Python sequences. For example, higher-level quantum operations can be created by defining Python functions, which can also be iterated, transformed or filtered.

Various aspects of the Cirq syntax are examined in detail in what follows.

Qubits

The first part of creating a quantum circuit is to define a set of qubits (also known as a quantum register) to act on.

Cirq has three main ways of defining qubits:

- `cirq.NamedQubit` : used to label qubits by an abstract name.
- `cirq.LineQubit` : qubits labelled by number in a linear array.
- `cirq.GridQubit` : qubits labelled by two numbers in a rectangular lattice.

Here are some examples of defining each type of qubit.

```
1  import cirq
2
3  # Using named qubits can be useful for abstract algorithms
4  # as well as algorithms not yet mapped onto hardware.
5  q0 = cirq.NamedQubit('source')
6  q1 = cirq.NamedQubit('target')
7
8  # Line qubits can be created individually
9  q3 = cirq.LineQubit(3)
10
11 # Or created in a range
12 # This will create LineQubit(0), LineQubit(1), LineQubit(2)
13 q0, q1, q2 = cirq.LineQubit.range(3)
14
15 # Grid Qubits can also be referenced individually
16 # The following create a 2d-grid with 4 rows and 5 columns -> 20 qubits in total
17 q4_5 = cirq.GridQubit(4, 5)
18
19 # Or created in bulk in a 2d-square
20 # This will create 16 qubits from (0,0) to (3,3)
21 qubits = cirq.GridQubit.square(4)
```

There are also pre-packaged sets of qubits called Devices. These are qubits along with a set of rules for how they can be used. A `cirq.Device` can be used to ensure that two-qubit gates are only applied to qubits that are adjacent in the hardware, and other constraints.

Gates and Operations

The next step is to use the qubits to create operations that can be used in the circuit. Cirq has two fundamental concpets:

- A `Gate` is a single instruction that can affect one or more qubits.
- An `Operation` is the overall effect of a gate applied in a cirquit to an input set of qubits.

For instance, `cirq.H` is the quantum Hadamard and is a `Gate` object.

`cirq.H(cirq.LineQubit(1))` is an `Operation` object and consists in the Hadamard gate applied to a specific qubit (line qubit number 1).

Gates and Operations in Cirq are considered to be immutable objects. This means that a `cirq.Gate` or `cirq.Operation` should not be modified after its creation. If attributes of these objects need to be modified, a new object should be created.

`Gates` can generally be applied to any type of qubit (`NamedQubit` , `LineQubit` , `GridQubit` , etc.) to create an `Operation` . Many gates explained in 2.2 are included within Cirq. `cirq.X` , `cirq.Y` , and `cirq.Z` refer to the single-qubit Pauli gates. `cirq.CNOT` , `cirq.SWAP` are just a few of the common two-qubit gates. `cirq.measure` is a function to apply a `MeasurementGate` and perform a measurement in the computational basis. Here are some examples of operations that can be performed on gates and operations:

```
# Example gates
cnot_gate = cirq.CNOT
pauli_z = cirq.Z

# Use exponentiation to get square root gates.
sqrt_x_gate = cirq.X**0.5

# Some gates can also take parameters
sqrt_sqrt_y = cirq.YPowGate(exponent=0.25)

# Create two qubits at once, in a line.
q0, q1 = cirq.LineQubit.range(2)

# Example operations
z_op = cirq.Z(q0)
not_op = cirq.CNOT(q0, q1)
sqrt_iswap_op = cirq.SQRT_ISWAP(q0, q1)

# One can also use the gates specified earlier.
cnot_op = cnot_gate(q0, q1)
pauli_z_op = pauli_z(q0)
sqrt_x_op = sqrt_x_gate(q0)
sqrt_sqrt_y_op = sqrt_sqrt_y(q0)
```

Circuits and Moments

A `Circuit` is a collection of `Moments` . A `Moment` is a collection of `Operations` that all act during the same abstract time slice. Each `Operation` must be applied to a disjoint set of qubits compared to each of the other `Operations` in the `Moment` . A `Moment` can be thought of as a vertical slice of a quantum circuit diagram, as shown in Fig. 4.5.

Circuits can be constructed in several different ways. By default, Cirq will attempt to slide the operation into the earliest possible `Moment` with respect to the one when the operation is inserted. One can use the `append` function in two ways:

- By appending each operation one-by-one:

```
circuit = cirq.Circuit()
qubits = cirq.LineQubit.range(3)
circuit.append(cirq.H(qubits[0]))
circuit.append(cirq.H(qubits[1]))
circuit.append(cirq.H(qubits[2]))
```

- Or by appending some iterable of operations. A preconstructed list works:

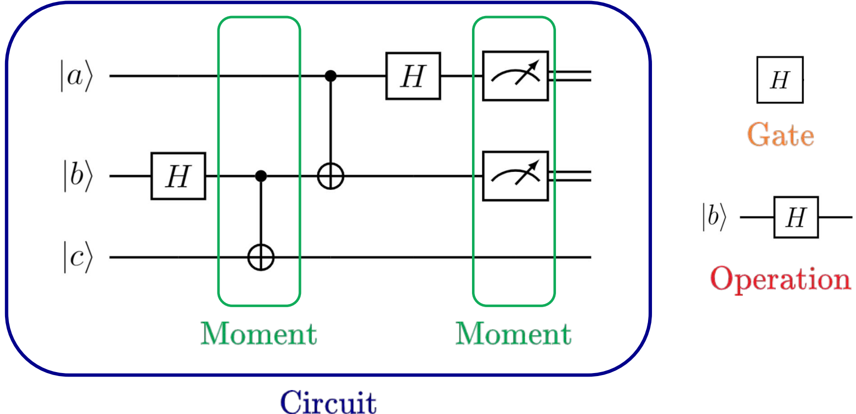


Figure 4.5: Circuit moments are vertical slices of a circuit diagram. A `Gate` can be thought as a *factory* that, given input qubits, produces an associated `Operation` object

```
1  circuit = cirq.Circuit()
2  ops = [cirq.H(q) for q in cirq.LineQubit.range(3)]
3  circuit.append(ops)
```

A generator that yields operations also works. This syntax is often used in the documentation, and works both with the `cirq.Circuit()` initializer and the `cirq.Circuit.append()` function. Note that all of the Hadamard gates are *pushed as far left as possible*, and put into the same `Moment` since none overlap.

If operations are applied to the same qubits, they will be put in sequential, insertion-ordered moments.

Simulation

A `Simulator` can be used to calculate the outcomes of applying a a quantum circuit. The base simulator included in Cirq is capable of calculating circuits results up to 20 qubits. It can be initialized with `cirq.Simulator()`.

There are two different approaches to use a simulator:

- `simulate()`: During classical circuit simulations , the simulator can directly access and view the resulting wave function. This feature can be useful for debugging, learning, and understanding how circuits operate.
- `run()`: When using real quantum devices, one can only access the final computation result and need to sample it to obtain the result's distribution. Running the simulator as a sampler mimics this behavior, and the output will only contain a sequence of bit strings.

As an example, one can consider the following code to simulate a 2 qubits state:

```
1  # Create a circuit to generate a the quantum state:
2  # 1/√2 (|00⟩ + |11⟩)
3
4  # Define the circuit
5  bell_circuit = cirq.Circuit()
6
7  # Create a register with two qubits in Line topology
8  q0, q1 = cirq.LineQubit.range(2)
9
```

```
# Add a Hadamard gate on qubit 0, putting this qubit in superposition.
bell_circuit.append(cirq.H(q0))

# Add a CNOT gate on control qubit 0 and target qubit 1
bell_circuit.append(cirq.CNOT(q0, q1))

# Display the circuit
print(bell_circuit)

# Initialize Simulator
sim = cirq.Simulator()

print('Simulate the circuit:')
results = sim.simulate(bell_circuit)
print(results)

# For sampling, need to add a measurement at the end
bell_circuit.append(cirq.measure(q0, q1, key='result'))

# Sample the circuit
samples = sim.run(bell_circuit, repetitions=1000)
```

The output of the code above will be:

```

0: ----H----@----
            |
1: ---------X----

Simulate the circuit:
measurements: (no measurements)

qubits: (cirq.LineQubit(0), cirq.LineQubit(1))
output vector: 0.707|00> + 0.707|11>

phase:
output vector: |>
```

The circuit is printed out in ASCII text, which is the standard way of displaying circuits in Cirq. Besides the output vector that contains each state and the corresponing amplitudes, the simulator also shows the phase of the output vector.

Visualizing Results

One can obtain a sample distribution of measurements by using the `run()` method. Then, simulated samples can be represented as a histogram using `cirq.plot_state_histogram` and `Matplotlib`.

```
import matplotlib.pyplot as plt

cirq.plot_state_histogram(samples, plt.subplot())
plt.show()
```

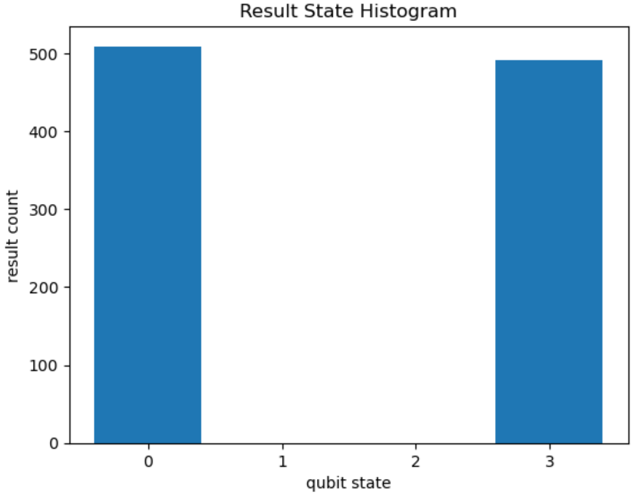


Figure 4.6: Histogram of the simulation results

The measurement results are expressed in decimal form. Nevertheless, this histogram has some unoccupied qubit states, which may become problematic working with a larger number of qubits. To plot sparsely sampled data, retrieve the `Counts` with the `histogram()` function from the obtained results and pass them to `cirq.plot_state_histogram` for plotting. By collecting the results into counts, all the unobserved qubit states are excluded.

```
# Pull of histogram counts from the result data structure
counts = samples.histogram(key='result')
print(counts)

# Graph the histogram counts instead of the results
cirq.plot_state_histogram(counts, plt.subplot())
plt.show()
```

The output will be return a `Counter` object of key-value pairs corresponding to measurement outcome and frequency of occurrence:

```

Counter({3: 516, 0: 484})

```

Cirq provides fine-grained control to the end users. It provides mechanisms for fine-tuning exactly how a quantum program runs on the targeted quantum hardware, and tools for simulating hardware constraints, such as limitations due to noise or the physical layout of the qubits. The qubit types available to the programmer further demonstrate Cirq's focus on NISQ hardware. Cirq also allows for qubit types that do not impose any physical layout, either for developing quantum programs that are only intended to be simulated, or to be used as part of the definition of a custom layout. Once a qubit type is selected, device constraints can be specified programmatically, and Cirq will validate that a given circuit satisfies all the constraints.

Cirq is a Python library for quantum programming with a strong focus on supporting near-term quantum hardware. The primary goal of Cirq is to facilitate the development of programs that can run on Quantum Computers available now or in the near future (NISQ hardware), depending on specific device topologies. Google's SDK is tailored for NISQ hardware, but this has a potential limitation: it may not be suitable and flexible enough for when Quantum Computers will scale up in terms of qubit numbers and performance.

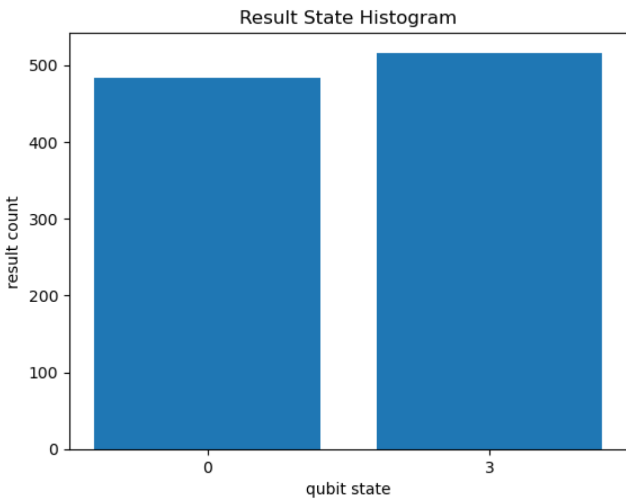


Figure 4.7: A histogram over the states that were actually observed can often be more useful when analyzing results

4.3 Quantum Development Kit - Q#

Institution / Vendor	Microsoft
First Release	Jan 2018
Open Source	Yes
Homepage	Homepage [36]
GitHub	Git [51]
Documentation	Online Doc [52]
Supported OS	Mac, Linux, Windows
Classical Host Language	Python, C#
Quantum Language	Q#

Table 4.4: Overview of the Quantum Development Kit library

Q# (pronounced "Q sharp") is part of Microsoft's Quantum Development Kit (QDK) and provides support for an Integrated Development Environment (IDE) and tools for program visualization and analysis. One of its main purposes is to assist in the development of future large-scale applications, while supporting the efforts of users working with current quantum hardware. Q# is a standalone QPL with a high level of abstraction. It implements programs in the form of statements and expressions, like in classical programming languages. This language has the possibility to be integrated into a Python SDK for greater flexibility, using tools and packages with a lower level of abstraction.

In addition to the standard library, which provides a set of essential functions and operations for developing quantum programs, Q# includes domain-specific packages such as:

- Quantum Chemistry: library for simulating problems in chemistry and material science.
- Quantum Machine Learning: library which is designed to perform hybrid quantum/classical Machine Learning experiments.
- Quantum Numerics: library that provides a set of numerical methods and functionalities for QC.

As with the other major open source programming languages, Q# programs can be targeted to run on different quantum hardware backends. In this case, Microsoft has developed Azure Quantum [36], a full-stack cloud ecosystem for quantum solutions. It provides a set of tools and services, from the design of quantum algorithms to the deployment of quantum applications. On Azure Quantum, it is possible to run Q# programs on local simulators or via cloud on real hardware from third-parties such

as IonQ, Rigetti, Pasqal, Toshiba and more others.

As written on the GitHub page, the Q# language is designed according to some principles:

- Q# is hardware agnostic. The goal is to express and exploit powerful Quantum Computing concepts regardless of how hardware evolves in the future.
- Q# is designed to scale to the full range of quantum applications. To be applicable to a wide range of applications, Q# allows the creation of reusable components and layers of abstractions. To achieve performance as quantum hardware scales, automation is critical.
- Q# focuses on expressing information to optimize execution. The goal is to ensure efficient execution of quantum components, independently of the context in which they are invoked.
- Q# will grow and evolve over time. Quantum devices are constantly growing: Q# is designed to adapt and change as the technology advances.

The Quantum Development Kit includes several quantum simulators that provide different environments for running and testing quantum programs. As shown in Tab. 4.5, quantum simulators are responsible for providing implementations of quantum operations for an algorithm. This includes primitive operations such as H, CNOT, and measure, as well as qubit management and tracking.

Simulator	**Syntax**	**Description**
Full state simulator	`QuantumSimulator`	Base Q# simulator that runs and debugs quantum algorithms up to 30 qubits
Sparse simulator	`SparseSimulator`	Simulates quantum algorithms with sparse states, small number of states in superposition
Trace-based resource estimator	`QCTraceSimulator`	Runs advanced analysis of resources consumptions for quantum algorithm instances, and supports thousands of qubits
Toffoli simulator	`ToffoliSimulator`	Simulates quantum algorithms that are limited to `X`, `CNOT`, and multi-controlled operations, supporting million of qubits
Noise simulator	`NoiseSimulator`	Simulates quantum algorithms under the presence of noise

Table 4.5: Overview of the simulator types available in Q#

In addition to these local *in-memory* simulators, the QDK supports backend quantum simulators that are provided by third-party Microsoft partners. In particular, IonQ, Quantinuum, and Rigetti supply quantum emulators with specific purposes and capabilities.

The code samples and libraries are a great way to learn the Q# language, and the online documentation contains information on how to get started with the QDK, including how to install it and set up the environment, how to run a first quantum program, and lots of advice on developing algorithms using the Q# programming language and the Python SDK. This documentation is very detailed and contains a large number of tutorials and examples that are very useful for learning the language and developing quantum programs. In addition, the QDK includes a set of Quantum Katas [53], which are a series of self-paced tutorials that guide the user through the process of learning Quantum Computing and Q# programming by solving programming exercises. Being Q# and open source language, through GitHub it is possible to access the source code and contribute to its development. An addition or modification to the Q# language starts with a suggestion, for instance by opening an issue on Git. After a review by the Language Design Team, the proposal feature can be approved or declined. In the first positive case, the suggestion is implemented and tested, and finally released into the master branch.

4.3.1 Syntax and Code Examples

The syntax of Q# is characterized by a high level of abstraction. It is a strongly-typed language, which implies that Q# does not implicitly cast between distinct types. Q# is supported by a number of useful operations, functions, and user-defined types that make up the Q# standard libraries. The standard libraries allow working with data structures and modelling. The `Microsoft.Quantum.Canon` namespace provides operations, functions, and types for working with classical data, such as tuples and arrays. In addition, this namespace provides a variety of different control constructs to make it easier to express high-level algorithms as quantum programs. The `Microsoft.Quantum.Intrinsic` and `Microsoft.Quantum.Math` namespaces contain the implementations of the Pauli operators and other common quantum gates on one hand, and mathematical functions and data types on the other. In general, three separate files are required to execute programs using the QDK:

1. A `.qs` file, where quantum operations (analogues of functions in Python) are stored.
2. A `.cs` driver file where quantum operations are executed in the main program.
3. A `.csproj` file that defines the project and contains metadata about the chosen architecture and package references.

In the following, the general parts that compose a Q# program will be explored in more detail. Consider for intstance the following code:

```
1  namespace HelloQuantum {
2
3  open Microsoft.Quantum.Canon;
4  open Microsoft.Quantum.Intrinsic;
5
6  @EntryPoint()
7  operation SayHelloQ() : Unit {
8      Message("Hello quantum world!");
9      }
10 }
```

`@EntryPoint` can be used to tell the Q# compiler where to begin executing the program. The program prints this message:

```
1  Hello quantum world!
```

Namespaces

Every Q# file typically starts with a `Namespace` declaration. A Namespace is a container for a set of related operations, functions, and user-defined types.

```
1  namespace HelloQuantum {
2      // Here functions and operations are defined
3  }
```

Libraries

Q# makes extensive use of libraries. A library is a package that contains functions and operations that can be used in quantum programs. For example, the `Microsoft.Quantum.Chemistry` library allows to perform quantum calculations related to chemistry. There are several standard libraries that include all sorts of basic operations. When a function or an operation is called, it is necessary to

specify the namespace of the library in which this functionality is contained. Here is an example that calls the `Message` function from the `Microsoft.Quantum.Intrinsic` library to print a message to the console:

```
1  namespace HelloQuantum {
2
3  operation SayHelloQ() : Unit {
4      Microsoft.Quantum.Intrinsic.Message("Hello quantum world!");
5      }
6  }
```

To include the namespace of a library, it is possible to use the `open` keyword. This allows to invoke library's functions and operations without having to specify the namespace each time. The Q# documentation provides complete reference documentation for each built-in library that can be included.

Types

Q# provides many built-in types such as `Int`, `Double`, `Bool`, and `String`, as well as types that are specific to Quantum Computing. For example, the `Result` type represents the result of a qubit measurement and can take one of two possible values: `One` and `Zero`. Q# also provides types that define ranges, arrays, and tuples, in addition to the ability to define custom types.

Allocating Qubits

In Q#, qubits are allocated through the `use` keyword. Users can allocate one or many qubits at a time. Here there is a simple example that allocates one qubit:

```
1  // Allocate a qubit.
2  use q = Qubit();
```

By default, every allocated qubit with the `use` keyword starts in the zero state $|0\rangle$.

Quantum Operations and Functions

Once allocated, a qubit can be passed to operations and functions, also referred as *callables*.

- `Operations` are the basic building blocks of a Q# program. A Q# operation is a quantum subroutine that contains quantum operations that modify the state of the qubit register. In this language they are non-deterministic: they take a single (potentially tuple) input argument and return a single, again possibly tuple, value as output. Calls to operation values can have side effects, and the output can vary for each call, even when called with the same argument.
- `Fuctions` instead represents a deterministic callable. Functions have no side effects, and the output will always be the same given the same input.

To define a Q# operation, you specify a name for the operation along with its inputs and its output. A basic example:

```
1  operation SayHelloQ() : Unit {
2  Message("Hello quantum world!");
3  }
```

Here, `SayHelloQ` is the name of the operation. It takes zero arguments as its input and returns of type `Unit`, which means that the operation returns no information. Q# treats qubits as *opaque*

objects that can be passed to both functions and operations, but can only be interacted with by passing them to instructions that are native to the targeted quantum processor. Such instructions are always defined as operations, since their purpose is to modify the quantum state. The Q# libraries also provide operations such as the Hadamard or `H` operation. Given a qubit in Z-basis, the `H` operation puts the qubit into a superposition of $|0\rangle$ and $|1\rangle$. All direct actions on the state of a qubit are all defined by *intrinsic* callables such as `X` and `H`, that is callables whose implementations are not defined within Q# but are instead defined by the target machine. What these operations actually do is only made concrete by the target machine chosen for the execution of the Q# program.

For example, if the program is run on the full-state simulator, the simulator will perform the appropriate mathematical operations on the simulated quantum system. When the target machine is a real Quantum Computer, calling such operations in Q# will instruct the quantum device to perform the corresponding real operations on the real quantum hardware (e.g., in trapped-ions quantum operations are realized by precisely timed laser pulses). A Q# program recombines these operations as defined by a target machine to create more general and higher-level operations to express quantum computation.

Measuring Qubits

A Q# program has no ability to introspect into the state of a qubit, and is therefore completely agnostic about what a quantum state is or on how it is realized. Rather, a program can call operations such as `Measure` to learn information about the quantum state of the computation. There are many types of quantum measurement, but Q# focuses on projective measurements on single qubits, also known as Pauli measurements. When measuring in a given basis (for example, the computational basis $|0\rangle$ and $|1\rangle$) the qubit state is projected onto the basis state that was measured, destroying any superposition between the two.

In Q#, Pauli measurements are achieved by applying `Measure` operation, which performs a joint measurement of one or more qubits in the specified Pauli bases. `Measure` operation returns a `Result` type. For example, to perform a measurement in the computational basis $|0\rangle$ and $|1\rangle$ (Pauli Z basis), one can use `Measure([PauliZ], [qubit])` or the equivalent `M(qubit)`. A simple example is the following program, which allocates one qubit in the $|0\rangle$ state, then applies a Hadamard operation `H` to it and measures the result in the `PauliZ` basis.

```
1  operation MeasureOneQubit() : Result {
2      // Allocate a qubit, by default it is in zero state
3      use q = Qubit();
4
5      // Apply a Hadamard operation H to the state
6      // It now has a 50% chance of being measured 0 or 1
7      H(q);
8
9      // Now measure the qubit in Z-basis.
10     let result = M(qubit);
11
12     // Reset the qubit before releasing it.
13     if result == One { X(qubit); }
14
15     // Finally, return the result of the measurement.
16     return result;
17 }
```

The `Microsoft.Quantum.Measurement` namespace contains more specific measurement operations. For instance, the `MultiM` operation takes an array of qubits and returns an array of measurement results in the computational basis.

The following simulates the same circuit tested for the other SDKs in Sec. 4.1 and Sec. 4.2. First of all, `Namespaces` and a Q# `operation` has to be defined in the `Circuit.qs` file.

```
namespace Circuit{
    open Microsoft.Quantum.Intrinsic;        // for H and CNOT operations
    open Microsoft.Quantum.Measurement;      // for the measurement operations

    // Operation to prepare an entangled state
    // q0 and q1 are the inputs with type Qubit
    // Unit means that the operation has no return value
    operation PrepareEntangledState(q0: Qubit, q1: Qubit) : Unit {
        H(q0);              // apply Hadamard gate to qubit q0
        CNOT(q0, q1);       // apply CNOT gate to with control q0 and target q1
    }

    // Operation to measure the entangled state
    // no input arguments , a tuple of Result type is returned (the two measurements)
    operation MeasureEntangledState() : (Result, Result) {

        use q0 = Qubit();       // allocate qubit q0 in |0>
        use q1 = Qubit();       // allocate qubit q1 in |0>

        // call the operation to prepare the entangled state
        PrepareEntangledState(q0, q1);

        // Return the measurement results
        return (M(q0), M(q1));
    }
}
```

The `PrepareEntangledState` operation takes two qubits as input and prepares an entangled state between them. The `MeasureEntangledState` operation allocates two qubits, calls the previously defined entanglement operation, and then measures the state of the qubits. In this case the qubits are defined as separate variables, but it is also possible to define them as an array `Qubit[]` or a tuple `(Qubit, Qubit)`. The same applies for the measurement results, which can be defined as an array `Result[]` instead of a tuple `(Result, Result)`. To simulate this circuit, using the Python SDK of Q#, it is possible to avoid the `.cs` and `.csproj` files, only requiring:

```
# import the Python SDK for Q#
import qsharp

# import the operation defined in the .qs file
from Circuit import MeasureEntangledState

simulation = []      # define an empty list to store results

# simulate the operation MeasureEntangledState for 1000 shots
# and append the results to the list
for shots in range(1000):
    simulation.append(MeasureEntangledState.simulate())
```

The imported `qsharp` package allows Q# namespaces to appear as Python packages, and one can

import Q# callables more easily. One can employ Q# functions and operations as Python objects, and use methods on these objects to take advantage of more flexibility of an environment with a lower level of abstraction (e.g., the `for` statement in the example above). Furthermore, with some basic operations on the retrieved Python list and with `Matplotlib`, it is possible to obtain the histogram of the simulation results (Fig. 4.8).

```
1
2  {'00': 494, '11': 506}
3
```

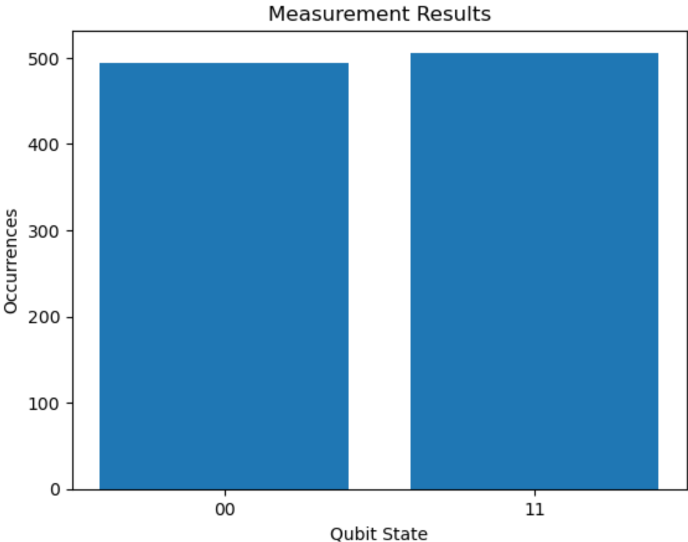


Figure 4.8: The histogram retrieved after measure the entangled two qubits state 1000 times

By default, the `import qsharp` command loads all of the `.qs` files in the current folder and makes their Q# operations and functions available for use within the Python script. To load Q# code from a different folder, the `qsharp.projects` API can be used to add a reference to a `.csproj` file for a Q# project that contains defined operations and functions.

With the `%trace` magic command (available only in Q# Jupyter Notebooks) one can *trace* a run of the Q# program and visualize the quantum circuit based on that execution. An example of the output of the `%trace` command is shown in Fig. 4.9.

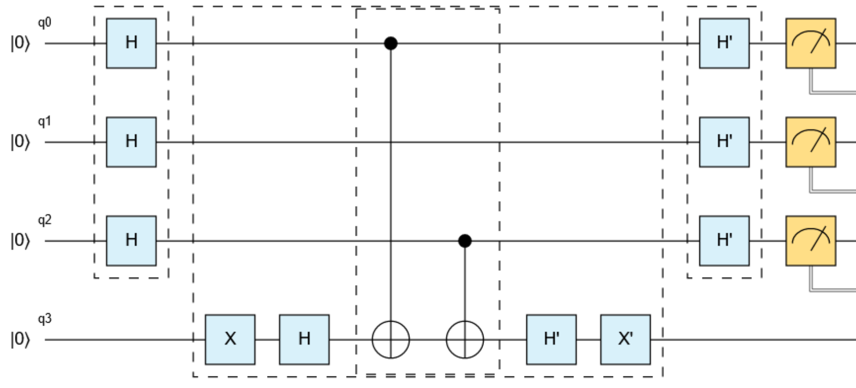


Figure 4.9: The quantum circuit obtained with the Q# magic command `%trace` available on the Python SDK

The Q# Microsoft SDK for QC provides a powerful and comprehensive platform for quantum algorithm

development and experimentation. Its nature as a standalone high-level programming language makes it easier to define a process for creating quantum programs. Q# also provides an extensive library of quantum operations and functions, enabling users to design complex quantum circuits and algorithms with ease, also thanks to its interoperability with Python. Another notable feature of this SDK is that it supports integration with quantum hardware through the Azure Quantum cloud service, allowing researchers and coders to harness the power of real quantum processors for their experiments.

4.4 t|*ket*⟩

Institution / Vendor	Quantinuum
First Release	Dec 2018
Open Source	Yes
Homepage	Homepage [54]
GitHub	Git [55]
Documentation	Online Doc [56]
Supported OS	Mac, Linux, Windows
Classical Host Language	C++, Python
Quantum Language	/

Table 4.6: Overview of the t|*ket*⟩ development library

t|*ket*⟩ is an advanced Software Development Kit developed by Quantinuum for the creation and execution of programs for gate-based Quantum Computers. This SDK is a powerful tool for optimizing and manipulating platform-agnostic quantum circuits, and it is specifically designed to focus on the development of quantum algorithms for NISQ devices. t|*ket*⟩ consists of two main components: (i) a powerful optimising compiler written in C++ and (ii) a flexible user interface and runtime system written in Python. This Python layer allows the user to define circuits and call compiler functions, while the runtime environment manages kernels for execution, and provides convenient methods for defining loops, updating parameters, and collecting statistics.

t|*ket*⟩ is open source and easily accessible through the Pytket Python package, with extension modules providing compatibility with many quantum hardwares, classical simulators, and popular quantum software libraries. Indeed, the Pytket package provides an API for interacting with t|*ket*⟩ and transpiling to and from other quantum circuit specifications. There are also separate packages for managing the interoperability between the Quantinuum SDK and other quantum software packages, such as Qiskit, Cirq, Q#, and more others.

Pytket provides many shortcuts and higher-level components for building circuits, including custom gate definitions, circuit composition, gates with symbolic parameters, and conditional gates. On the other hand, Pytket's flexible interface allows to import circuits defined in other SDK languages, or even in *raw* source code languages such as OpenQASM. In addition to the core Pytket package, extension modules are available to interface with quantum devices and simulators from various providers including Quantinuum, IBM, IonQ, and others. In t|*ket*⟩ this is done by defining a `Backend`, which represents a connection to a QPU or simulator for processing quantum circuits, locally or via the cloud. The types of `Backend` supported in Pytket are the following:

- QPUs: Real Quantum Computers that return shot-based results. E.g., `QuantinuumBackend`.
- Cloud Access: Interfaces with cloud platforms to access additional QPUs and simulators. E.g., `AzureBackend`.
- Emulators: Classically simulate a circuit and produce shot-based results.
  In some cases emulators use a noise model and have connectivity constraints to emulate real QPUs. E.g., `IBMQEmulatorBackend`.

- Statevector Simulators: Calculate the pure quantum state prepared by a circuit. Examples of statevector simulators are the `ForestStateBackend` and the `AerStateBackend`.
- Unitary Simulators: Calculate the unitary operator that is applied by a circuit. `AerUnitaryBackend`, an example of such a simulator, returns a unitary matrix/array.
- Density Matrix Simulators: Compute the density matrix generated by a circuit. The result can be a statistical mixture of states, in contrast to the statevector simulation. E.g., `CirqDensityMatrixSampleBackend`
- Other specialised simulators: There are extensions for simulating specific types of circuits. For instance, the `SimplexBackend` is designed to simulate Clifford circuits.

t|*ket*⟩ attempts to provide a consistent interface across the various backend platforms, so that a user can easily switch backends for an experiment without changing anything else in their code. t|*ket*⟩ is indeed designed to be retargetable, meaning that it can generate code for many different quantum devices, and is language agnostic, meaning that it accepts input from most of the major quantum software platforms. This is related to the compilation process of this SDK. In particular, the compiler performs an intermediate step between the frontend and backend, where it performs data and control flow analysis on an Intermediate Representation (IR) of the program, which is independent of both the source and the target languages. Standard quantum circuits are easily embedded in the t|*ket*⟩ IR, and this simplifies the *translation* with respect to different frameworks. Once the input has been translated into the IR, the central circuit transformation engine can begin its work. The transformation engine performs a user-configurable sequence of rewrites of the IR. This process is usually done in two stages:

1. Optimisation phase: an architecture-independent step aimed at reducing the size and complexity of the circuit.
2. Preparation phase: an architecture-dependent stage that prepares the circuit for execution on the target machine. This phase is itself divided into a rebase, which maps the gates present in the circuit to those supported by the chosen device, and a qubit mapping phase. The mapping phase is necessary to ensure that all the qubits that need to interact during the program are physically able to do so; this typically increases the size of the circuit, since most devices have limited interactions between qubits.

The end product of this process is a kernel: a circuit that can be executed on the chosen target device. The kernel can then be scheduled for execution by the runtime environment, or simply stored for later use. Thanks to its retargetability, t|*ket*⟩ can be used as a cross-compiler: source programs produced from any supported frontend can be compiled to run on hardware produced by any vendor.

The t|*ket*⟩ documentation available online it is very detailed and complete, with an accurate and constantly updated changelog. It also contains a large number of tutorials and examples that are very useful for learning the language and how to interface with the other SDK extensions. The Github repository contains the full source code of the quantum SDK. Opening an issue is the preferred way to report bugs with t|*ket*⟩ or request the implementation of new features. There is also a Slack channel [57] for community discussion and support and a Stack Exchange forum [58] for Pytket-related questions.

4.4.1 Syntax and Code Examples

To use Pytket, one can simply import the appropriate modules into Python code or in an interactive Python notebook. Circuits can be built directly using the Pytket interface by creating a blank circuit and adding gates in the desired order.

```
1  from pytket.circuit import Circuit, OpType
2
3  # Create a circuit with 4 qubits
4  c = Circuit(4, name="Example Circuit")
```

```
5
6   # Add gates to the circuit
7   c.H(0).X(1).Y(2).Z(3)
8   c.X(0).CX(1, 2).Y(1).Z(2).H(3)
9   c.Y(0).Z(1)
10  c.add_gate(OpType.CU1, 0.5, [0, 1])
11  c.Z(0).H(1).CX(3, 0)
```

In this example, the `Circuit` class (the main interface for creating circuits in Pytket) is used to create a new circuit with 4 qubits. Then, gates such as Hadamard `H`, Pauli gates (`X`, `Y`, `Z`) are added to the circuit. Note that special gates that require additional parameters, such as `CU1`, must be specified with the `add_gate` method. The generated circuit is returned in the output as a string containing all the operations that have been applied. Some methods of the `Circuit` class allow to access to the ciruit parameters:

- `circuit.name` returns the name of the circuit.
- `circuit.n_qubits` and `circuit.n_bits` return the number of qubits and classical bits respectively.
- `circuit.depth` returns the depth of the circuit.
- `c.get_commands` returns a list of all the operations in the circuit.

There are several ways to produce useful visualizations of circuits from Pytket, mainly using the methods in the `pytket.circuit.display` class. Using the function `render_circuit_jupyter` it is possible to render a circuit in a Jupyter notebook obtaining Fig. 4.10.

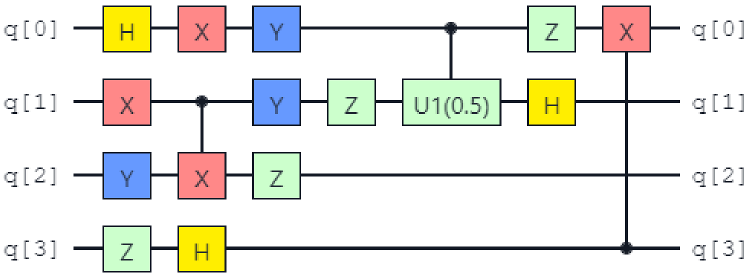


Figure 4.10: The circuit generated with the Pytket interface

Alternatively, t|*ket*⟩ Python converters provide support for several SDKs including Qiskit, Cirq, and Q#. These converters allow users to write code in the source language, and then convert it to the t|*ket*⟩ IR for compilation. An example of this conversion is shown below:

```
1   from qiskit import QuantumCircuit
2   from pytket import Circuit
3
4   # Define qiskit QuantumCircuit
5   qc = QuantumCircuit(3)
6   qc.h(0)
7   qc.cx(0, 1)
8
9   # Convert to pytket
10  tk_circ = qiskit_to_tk(qc)
```

In this case, the conversion is accomplished by the `qiskit_to_tk` function, which takes a Qiskit `QuantumCircuit` object as input and returns a Pytket `Circuit` object. Conversion in the opposite direction can be performed with `tk_to_qiskit()` or `tk_to_cirq()`. If there is no replacement for a Pytket operation in Qiskit or Cirq, the unsupported operation is implemented in terms of the available gates. Then, to render the quantum circuit, the visualization methods of the *arrival* language can be used.

Circuits, Gates and Boxes

The standard intermediate representation (IR) in t$|ket\rangle$ is the circuit. A circuit can be tought as a labelled graph with some additional structures. The vertices of the graph correspond to operations, usually quantum or classical logic `Gates`, but also `Boxes`. Edges in the graph trace the flow of computational resources from operation to operation. Typically, these resources are qubits, and the operations are unitary gates. There is a wide array of allowed gates in t$|ket\rangle$, covering the native gates of the platforms that can be interfaced. The most common quantum gates are one- and two-qubits gates, mirroring the gates on physical superconducting and ion trap hardware, but some others with arbitrary quantum controls are allowed. Boxes are a special class of operations in t$|ket\rangle$. A box is a container which encapsulates an entire circuit. Boxes allow users to incorporate additional subroutines. As these subcircuits can in turn contain further boxes, a single circuit can contain a hierarchy of arbitrary rank.

The `CircBox` is the most general type of box, implementing an arbitrary circuit. But Pytket supports several other useful box types:

- `Unitary1qBox` (implementing an arbitrary 2×2 unitary matrix);
- `Unitary2qBox` (implementing an arbitrary 4×4 unitary matrix);
- `ExpBox` (implementing e^{itA} for an arbitrary 4×4 hermitian matrix A and parameter t);
- `PauliExpBox` (implementing $e^{-\frac{1}{2}i\pi t(\sigma_0 \otimes \sigma_1 \otimes \cdots)}$ for arbitrary Pauli operators $\sigma_i \in \{\mathrm{I}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$ and parameter t).

An example of a circuit containing a box is shown in the following code:

```
import numpy as np
from pytket.circuit import Circuit, ExpBox

# Create a circuit with 4 qubits
circ = Circuit(4)
circ.H(0).CX(0, 1).CX(1, 2).CX(2, 3)

# Define the 4x4 matrix A
A = np.asarray([[1, 2, 3, 4 + 1j],
                [2, 0, 1j, -1],
                [3, -1j, 2, 1j],
                [4 - 1j, -1, -1j, 1]])

# Define the ExpBox with Matrix A and exp parameter 0.5
ebox = ExpBox(A, 0.5)

# Add the ExpBox to the circuit circ on qubits 0, 1
circ.add_expbox(ebox, 0, 1)
```

The `ExpBox` class is used to implement the exponential of a matrix A on a subset of qubits. In this

case, the matrix A is defined as a 4×4 `numpy array` and the `ExpBox` is added to the defined circuit on qubits 0 and 1.

Moreover, Circuits can be composed either serially, by connecting wires together, or in parallel, using the `append` command.

Symbolic Parameters

To enable the efficient compilation of algorithms, t$|ket\rangle$ supports symbolic parameters. Many of the gates supported by Pytket are parametrized by one or more phase parameters, representing rotations in multiples of π. For example, Rz($\frac{1}{2}$) represents a quarter turn, i.e. a rotation of $\pi/2$, around the Z axis. The gates can be directly implemented specifying the values of these parameters or, to construct and manipulate circuits in a different way, without specifying these values. This makes it possible to perform calculations in a general setting and only later specify values for the parameters. Thus, Pytket allows to declare any of the parameters as a symbol. All manipulations (such as combining and cancelling gates) are performed on the symbolic representation:

```
1  from pytket.circuit import Circuit
2  from sympy import Symbol
3
4  # Create a circuit with 1 qubit
5  c = Circuit(1)
6
7  # Add a Rz gate with angle ½π
8  c.RZ(0.5, 0)
9
10 # Declare a symbol "a"
11 a = Symbol("a")
12 # Add a Rz gate with symbolic parameter "a"
13 c.Rz(a, 0)
14
15 # at this point the circuit is: [Rz(0.5) q[0];, Rz(a) q[0];]
16 from pytket.transform import Transform
17
18 Transform.RemoveRedundancies().apply(c)
19
20 # after the transformation the circuit is: [Rz(0.5 + a) q[0];]
21
22 # Substitute the symbol "a" with the value 0.75
23 c.symbol_substitution({a: 0.75})
24 # after the substitution the circuit is: [Rz(1.25) q[0];]
```

This class of circuits is handled using partial compilation: the circuit is precompiled with unknown, symbolic parameters using an expressive symbolic manipulation (e.g., `simpy`). The result can be used as a template circuit and, after parameter values at a given iteration have been substituted, the circuit can be compiled to obtain the resulting kernel.

In order to substitute values for symbols one can use the `symbol_substitution` method, supplying a Python dictionary `symbol:value`. Above, the symbol `a` has been substituted with the value 0.75, obtaining the final circuit with a single Rz gate with angle 1.25π. Note that in the code above it has been applied a transformation to remove the redundant gates, obtaining a circuit with a single gate with parameter $\frac{1}{2} + a$.

t|ket⟩ Transform System and Backends

In general, a quantum algorithm can be expressed in multiple ways using a given set of gates; the goal is to express it in a way that minimises the gate count and circuit depth. The core of t|ket⟩ is a powerful circuit rewriting engine called the transform system. A function that performs rewrites using this system is called a `Transform` pass. Apart from optimization, the transform system plays an essential role in generating circuits that are executable on the target hardware.

t|ket⟩ provides several `Backends`, each supporting a different quantum hardware or classical simulator. Supporting a specific platform means, firstly, generating a circuit that respects the constraints of the hardware or simulator (generally, connectivity and primitive gate limitations); secondly, the backend must dispatch the kernel for execution and collect the results.

The following code shows an example of using a backend to execute the circuit simulated also with the other analyzed SDKs.

```
from pytket.circuit import Circuit
from pytket-qiskit import AerBackend

# Create a circuit with a register of 2 qubits and 2 bits
c = Circuit(2,2)
c.H(0)       # Hadamard gate on qubit 0
c.CX(0, 1)  # CNOT gate with qubit 0 as control and 1 as target

# Measure all qubits and store the result in the classical register
c.measure_all()

# Create an AerBackend object
backend = AerBackend()

# Compile the circuit for the AerBackend
c = backend.get_compiled_circuit(c)

# Execute the circuit on the AerBackend for 1000 shots
handle = backend.process_circuit(c, n_shots=1000)

# Get the result of the execution
counts = backend.get_result(handle).get_counts()
```

After creating a circuit with t|ket⟩, the Qiskit `AerBackend` is selected for the execution. A key step is represented by `Backend.get_compiled_circuit()`. The compilation step maps from the universal computer abstraction presented at `Circuit` construction to the restricted fragment supported by the target `Backend`. Each aspect of the compilation procedure is handled by Pytket: first it solves the constraints of the `Backend` to get from the abstract model to something executable. Secondly, it optimizes/simplifies the `Circuit` to make it faster, smaller, and less prone to noise. When the circuit processing is complete, the final output of the two qubits state is retrieved from the Qiskit backend and the results are stored in a Python dictionary.

```

Counter({(0, 0): 514, (1, 1): 486})

```

Furthermore, with `Matplotlib` it is possible to get the histogram of the simulation results (Fig. 4.11).

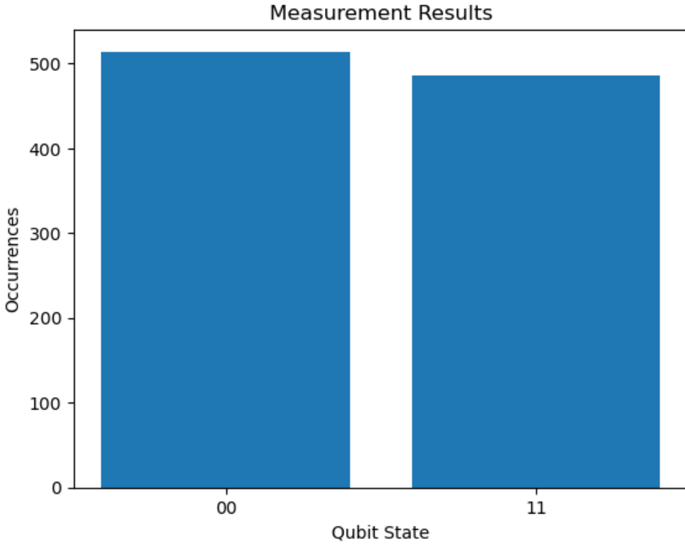


Figure 4.11: The histogram retrieved after measure the entangled two qubits state 1000 times

Predicates and Platform Agnosticism

t|*ket*⟩ is designed to focus on circuit compilation: taking a circuit representation and solving the gate-level constraints of the target device such as the restricted gate set, physical connections layout, and measurement model. The functions that verify that properties are satisfied are called `Predicates`. Each predicate is a function from a `Circuit` to a Boolean value: `True` if the circuit satisfies the corresponding property and `False` otherwise. These functions can incorporate some external information about the target hardware, such as connectivity graph and desired gate set. The set of `Predicates` required by a `Backend` can be queried with `Backend.required_predicates`. Moreover, calling `Backend.valid_circuit()` one can check if a `Circuit` fullfils all the requirements to run on a `Backend`. If the answer is negative, then `Backend.get_compiled_circuit()` will try to solve any remaining constraints if possible and return a new circuit.

Platform agnosticism is the principle that the tools and software developed can be made independent of the target hardware on which they will run on, rather than being locked behind specific interface software. t|*ket*⟩ enables this by providing `Predicates` and `Transformations`, key features for converting circuits between a variety of other quantum software frameworks.

t|*ket*⟩ is characterized by a compiler system for NISQ devices that aims to optimize the use of available hardware. The core of this open source SDK is indeed the compiler, which handles the translation of circuits between different QC frameworks. It is specifically designed for NISQ devices, and this can represent a possible limitation in relation to constantly evolving hardware. Nevertheless, the flexible design of t|*ket*⟩ offers many possibilities for future improvements.

4.5 Quantum Matcha TEA

Institution / Vendor	University of Padova & INFN - Istituto Nazionale di Fisica Nucleare
First Release	Nov 2021
Open Source	Yes
Homepage	Homepage [59]
GitHub	GitLab [60]
Documentation	Online Doc [61]
Supported OS	Mac, Linux, Windows
Classical Host Language	Python, FORTRAN
Quantum Language	QASM

Table 4.7: Overview of the Quantum Matcha Tea development library

Quantum TEA stands for Quantum Tensor network Emulator Applications. This SDK combines a suite of applications that use tensor network methods to simulate quantum systems and solve Machine Learning problems. It has been developed by the University of Padova and INFN (Istituto Nazionale di Fisica Nucleare) and it consists of a set of different applications, also called flavours:

- **Quantum Green TEA**: solves the static and time-dependent Schrödinger and Lindblad equations, e.g., one can simulate ground states, finite temperature states, and time evolutions.
- **Quantum Chai TEA**: contains the Machine Learning applications using tensor networks.
- **Quantum Red TEA**: contains the tensor libraries on which the other TEA libraries rely on for their tensor operations. It provides the interfaces to BLAS/LAPACK and CUDA for the higher-level applications.
- **Quantum TEA Leaves**: Auxiliary libraries, Python-FORTRAN interfaces, and Python solutions for common tensor network geometries are combined in this part of the Quantum TEA library.
- **Quantum Matcha TEA**: This is the main Python interface for users to interact with the Quantum TEA libraries and applications.

This thesis analyzes this last component of the Quantum TEA library, also because this and the Quantum TEA Leaves are the only two parts of the library that are currently available as open source projects.

Quantum Matcha TEA is a Quantum Computer emulator based on Matrix Product States (MPS). Its interface is completely written in Python, eliminating the need for the user does to care about the backends. It can be used to simulate quantum circuits defined with the Qiskit API or directly from its own API. However, it is also possible to simulate *qudits* systems: through the FORTRAN backend, one can emulate bosonic systems not only with 2 levels, but with an arbitrary number of dimensions. The simulation backends are tunable between CPUs and GPUs, and between Python and FORTRAN, and are scalable up to Message Passing Interface (MPI) for running on HPC clusters. With access to the final MPS state, one can perform any measurement accessible on a Quantum Computer, such as projective measurements, local observables, and Hamiltonians decomposed into Pauli matrices. In addition, users can access, for example, the entanglement entropy between different subsystems, and optimized methods for sampling the final state.

Quantum Matcha TEA (Quantum MAny Target quantum Circuit Hpc App) interfaces with the FORTRAN MPS quantum circuit simulator. This library allows to simulate quantum circuit with different local dimensions. Indeed, one can simulate qubits using the Qiskit interface, photonic circuits using the Strawberry Fields [62] interface, or d-level quantum systems or qudits, coding a suitable class. The Python library is meant as an interface for the FORTRAN core, which is composed of an optimized

tensor network library, able to run on multiple cores. The `run_simulation()` function transpile the circuit to adapt it to the linear structure of the MPS and run the circuit, obtaining in output the measurements.

It is possible to choose between different approaches for running the simulation by setting several parameters such as (i) the backend (Python `'PY'` or FORTRAN `'FR'`); (ii) the machine precision; (iii) the device of the simulation (`cpu` or `gpu`); and (iv) the number of processes for the MPI simulation (serial or in parallel). The default way of running programs in this SDK is serial. Although this it may be slower than a parallel implementation, it is safer as the approximations made to the states are less. There are various options to run the program serially:

- FORTRAN backend (`mpi_approach='SR'`). This is the fastest implementation, but requires the presence of the `main_qmatchatea.exe` executable.
- Python backend (`backend='PY'`). The backend is entirely built with `NumPy`, it is optimized but slightly slower than the FORTRAN backend.
- Python GPU backend (`backend='PY'`, `device='gpu'`). The backend is fully built with `CuPy` and runs the simulation on a GPU, returning back the results on the CPU.
- Using the Python FORTRAN backend (approach='PF'). This backend uses the Python backend, but simulates the FORTRAN I/O files.

Instead, there are three possibilities when selecting a program to run in parallel on multiple cores:

- Master/Worker approach (`mpi_approach='MW'`), where the state is stored in the master process and the application of the evolution operators are applied on the workers. It has the advantage that fewer workers are not in use during the computation.
- Cartesian approach (`mpi_approach='CT'`), where the MPS is evenly partitioned among different processes, and each process performs the evolution only on its subsystem.
- Running multiple independent serial simulations on multiple cores, using the `mpi4py` library.

At the end of the evolution, the MPS is reassembled into the master process and the measurements are performed. Tensor network emulators are able to efficiently compress quantum state information. This allows a large number of qubits to be simulated at the cost of limited entanglement and correlation within qubits.

The Quantum Matcha TEA documentation available online it is very detailed and complete, with an accurate and constantly updated changelog. It also contains many useful explanations on how to best use the interaction between Python and FORTRAN. The GitLab repository contains the complete source code. On this platform, users can open issues to report bugs, discuss new features, and ask for support.

4.5.1 Syntax and Code Examples

Tensor Network Interface

The Quantum Matcha TEA interface is based on tensor networks. In general, to represent a quantum state one has to deal with a number of parameters that grows exponentially with the number of qubits. Tensor networks are able to efficiently represent some subsets of the state space, in particular those with low entanglement. The main idea is to represent the state of a quantum system as a tensor network, where each tensor representing a subsystem is connected to its neighbours by a set of coefficients as shown in Fig. 4.12. Such networks compress these coefficients, i.e. they compress the quantum entanglement between the subsystems in the sense only the most important, the most meaningful correlations are preserved. Instead of using all the coefficients to represent the state, only those that express more information are retained.

$$|\psi\rangle = \sum_{\alpha=1}^{\chi} |A_\alpha\rangle \overset{\lambda_\alpha}{—} |B_\alpha\rangle$$

Figure 4.12: Schematic representation of a tensor network. The $|\psi\rangle$ state is decomposed into a sum of subsystems $|A\rangle$ and $|B\rangle$. Tensor networks compress the quantum correlations between these subsystems acting on the coefficient λ_α.

With this representation, where in each qubit encodes a subsystem of the quantum state and the links between qubits encode the entanglement between them, Quantum TEA obtains the MPS representation. Using this decomposition, the memory requirements pass from being exponential in the number of qubits ($O(2^n)$) to being linear in the number of qubits and polinomial in the bond dimension ($O(2n\chi^2)$). The bond dimension, denoted by χ, determines the level of entanglement that can be represented between neighbouring subsystems. Tensor networks are not able to represent highly-entangled states since the bond dimension parameter scales exponentially having to capture an exponentially large number of degrees of freedom.

Workflow

The Quantum Matcha TEA workflow is based on the following steps:

1. **Preprocessing**: the circuit is preprocessed to be suitable for the MPS simulation.
2. **Circuit definition**: the quantum circuit is defined using the Qiskit API or the Quantum Matcha TEA API.
3. **Observables definition**: the observables to be measured are defined using Quantum TEA Leaves package.
4. **Circuit simulation**: the circuit is simulated, serially or in parallel, using the MPS emulator on the chosen backend (CPU or GPU).
5. **Measurements and statistics**: the measurements on the observables are performed, the run-time statistics and the convergences checks are computed.

Preprocessing Phase

In order to simulate a quantum circuit a crucial phase is the preprocessing. First, the circuit must be exploited in a way that is suitable for the MPS simulation. In this type of simulation, each degree of freedom (qubit, mode, particle) is treated as a rank-3 tensor. Thus, the allowed operators are local or applied to nearest neighbours. This means that, when a circuit is passed to the preprocessing, it is translated into an equivalent circuit that follows these two properties. When using Qiskit, some transformations are performed:

- The circuit is mapped using a pre-defined set of gates, called `basis_gates`. These gates can be passed to the function, to satisfy a particular constraint of a physical machine wanted to be simulated.
- Map the circuit to a linear circuit. By an optimized application of swap gates, the non-local two-qubit gates are mapped into a series of local two-qubit gates.

These operations are performed automatically by the higher-level function `run_simulation()`. However, it is important to note that these operations take time proportionally to the size of the circuit. Afterwards, if a FORTRAN backend is selected, the parsing process is applied, i.e. a suitable input file is generated for the FORTRAN backend.

Circuit Definition

Quantum circuits are defined in QMatcha TEA as a list of layers, and each layer is a list of Instructions, i.e. a list [`QCOperation` , `sites`]. The first element of the list, `QCOperation` , represents all the possible operations that can be applied to the circuit, gates, but also the so-called `ClassicalConditions` : conditions to be checked for applying the operation on the simulation. The `sites` element instead, stands for the qubits on which the operations are applied: Moreover, there are two key features when one has to define qubits and classical bits of the circuit:

- `qregisters` : registers to keep track of quantum bits. Qubits can be added or removed, respectively using the methods `add_qregister()` and `remove_qregister()` . To apply an operation one can always specify to which register are referring, without having to worry about the indexing. In particular, qubits in the same register are naturally ordered from left to right.

```python
from qmatchatea.circuit import circuit as qcirc

# Create a Quantum Circuit with 2 sites/qubits
circ = qcirc.Qcircuit(num_sites = 2)

# Attach the register "q1" to the left
circ.add_qregister("q1", [0])

# Attach the register "q2" in the middle of the two original sites
circ.add_qregister("q2", [1])

# Attach the register "q3" to the right
circ.add_qregister("q3", [2])

# Apply a NOT gate on the first qubit of the register "q3"
qcirc.Qcircuit.x(circ , 0, "q3")
```

  This code creates a circuit with 2 qubits, and then it adds three quantum registers in different positions, to track the indexes of the qubits in the circuit with a name. So, if one wants to apply a NOT gate to the left-most qubit of the register "q3" of the circuit `circ` has to call `Qcircuit.x(circ, 0, "q3")` , since the left-most has index 0 for the register "q3".

- `cregisters` : classical registers to store the information about projective measurements performed on the system and the relative probability.
  Thus, the position index of the `cregister` will contain the tuple `(meas_result, meas_prob)` . By calling the method `inspect_cregister(creg, idx)()` , where `creg` is the string name of the classical register, one can access the value of the bit at the index `idx` .

Furthermore, there are three different ways of adding operations to the circuit, as shown in the following code:

```python
from qmatchatea.circuit import circuit as qcirc

# Create a Quantum Circuit with 2 sites/qubits
circ = qcirc.Qcircuit(num_sites=2)

# Add a Hadamard gate with the add() method
circ.add(operation = qcirc.QCHadamard(), sites = 0)

```

```
9  # Add a Hadamard gate with the insert() method
10 circ.insert(operation = qcirc.QCHadamard(), sites = 0, layer = 0)
11
12 # Add a Hadamard directly from the gate operation
13 circ.h(pos = 0)
```

For example, to add a Hadamard gate, the `add()` method requires to specify the `QCOperation` to be applied and the qubit/site on which the operation is applied. In addition to these arguments, with the `insert()` method needs the layer of the circuit in which the operation is inserted. Finally, with the `Qcircuit.h()` method, one can apply the Hadamard gate directly to the circuit `circ`, only specifying the qubit/site position `pos`. In all these three ways, it is also possible to specify the quantum register name in which the operation will be applied.

Circuits can also be initialized from Qiskit circuits, using the `from_qiskit()` method.

Observables Definition

The code related to the observables is included in the Python package QTEA Leaves. However, they are a fundamental element of the Quantum Matcha flavour. Observables define the quantities one is interested in measuring at the end of the simulation and can be of several types. For instance, `Projective Observables` allows the final projective measurements after the quantum state evolution. This observable gives single-shot measurements: the system is projectively measured a number of times equal to the argument `num_shots`. In this way, users can observe a statistic of the distribution of the states. The output of this measurement is a dictionary where the keys are the measured states on a given basis and the values are the occurrences of each measured state. When working with qubits, the measurement is performed on the computational basis.

Other types of observables that can be defined are `Local Observables`, which measure an operator defined at a single site over the complete system (e.g., the expectation value of the Pauli matrix σ_z); `Bond entropy Observables` which measure the quantum correlation between two subsystems (the Von Neumann bond entropy S_V) at the end of a circuit. More generally, one can have observables defined as the tensor product (or weighted sum of a tensor product) between one-site or two-site local observables (`Tensor Product Observables`), or even observables to measure the probabilities of the state configurations at the end of an evolution (`Probabilities Observables`).

Circuit Simulation and Measurements

The last stage of the workflow is the simulation of the circuit and the measurements of the observables. First of all one has to define the backend and the device on which the simulation will be performed. Optionally, one can also specify the number of processes for the MPI simulation and the approach to be used (serial or parallel). In some simulations of optimization problems users have to tuned some important convergence parameters such as:

- Maximum bond dimension χ: encodes the amount of entanglement that can be encoded in the quantum state, and also the computational resources that will be used. A low bond dimension means a quick but (maybe) inaccurate simulation. In general, users have to find the correct value to coherently represent the system.
- Cut ratio ε: represents the threshold for the singular values cut. The singular values cut are an indication of the approximation taking place during the simulation.
- Fidelity: the lower bound of the fidelity of the simulation. The fidelity is a measure of the closeness between the simulated state and the real state. It gives a measure of how much the simulation results reflect what would be obtained on a real Quantum Computer. If the fidelity is equal to 1, then the simulation is exact. Otherwise, if it is 0 there is probably a truncation error due to the chosen bond dimension, so not always this correspond to a bad simulation.

As done for the other analyzed SDKs, in the following code is shown an example of a simulation of a quantum circuit with Quantum Matcha TEA.

```
from qmatchatea.circuit import Qcircuit
from qmatchatea.py_emulator import QcMps
from qmatchatea import QCConvergenceParameters

# Create a quantum circuit with 2 qubits
qcirc = Qcircuit(num_sites = 2)

# Hadamard gate on qubit 0
qcirc.h(0, qreg="default")

# CNOT gate on qubits as a control 0 and 1 as a target
qcirc.cx([0, 1], qreg=["default", "default"])

# Add a classical register with two bits
qcirc.add_cregister(name="creg", num_bits=2)

# Measure the qubits and store the result in the classical register
qcirc.measure_projective(pos=0, cl_idx=0, qreg="default", creg="creg")
qcirc.measure_projective(pos=1, cl_idx=1, qreg="default", creg="creg")

# Initialize the MPS simulator using the default convergence parameters
mps = QcMps(num_sites=2, convergence_parameters=QCConvergenceParameters())

# Run the simulation
mps.run_from_qcirc(qcirc)

# Print the measurement results
mps.meas_projective(nmeas=1000)
```

The first part of the code defines the quantum circuit, with two qubits and two classical bits. Then, the Hadamard gate is applied to the first qubit, and the CNOT gate is applied to the first qubit as the control and the second qubit as the target. The two qubits are then measured and the results are stored in the properly added classical register `creg`. Then, the MPS simulator is initialized with the default convergence parameters, and the simulation is run using the `run_from_qcirc()` method. Finally, the dictionary obtained by projective measurement on the computational basis (`meas_projective`), for 1000 shots, is printed.

The output of the simulation is the following:

```

{'00': 505, '11': 495}

```

Furthermore, with `Matplotlib` it is possible to get the histogram of the simulation results (Fig. 4.13).

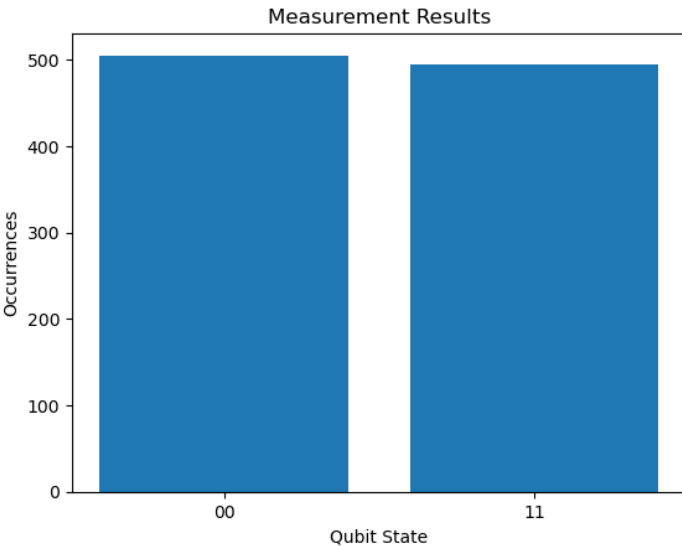


Figure 4.13: The histogram retrieved after measure the entangled two qubits state 1000 times

Quantum Matcha TEA still lacks of its own circuit visualization: to graphically represent circuits users have to go through Qiskit.

This SDK cannnot be used to run quantum algorithms and applications directly on real quantum hardware, but only to simulate them through tensor networks. Matrix Product State simulations are not limited by the number of qubits, but by the entanglement one wishes to describe. Therefore, Quantum Matcha TEA is not suitable for general quantum states, but only for those without strong correlations. In fact, one of the main purposes of this library is to run algorithms on hybrid quantum-classical computers, in particular HPCs. Therefore, a core feature of this library is to optimize and parallelize algorithms as much as possible in order to achieve the maximum efficiency in the implementation process. This last capability distinguishes Quantum TEA from the other analyzed SDKs, and provides the opportunity to experiment with Quantum Computing from a different perspective.

Chapter 5

Comparative analysis and Discussion

The aim of this chapter is to provide a comparative analysis of the different quantum programming frameworks, in order to highlight their strengths and weaknesses. The first part of the analysis is based on characteristics and key features of the Python SDKs presented in the previous chapter. Furthermore, after presenting the quantum teleportation protocol, the second part of the analysis focuses on highlighting the differences in the implementation in the various frameworks. Finally, the third part of the comparison deals with a coding specific metric, the cyclomatic complexity, in order to evaluate the code quality of the analyzed libraries.

5.1 Diversity of Quantum Programming Frameworks

The premise of this section is that all the analyzed platforms are significant achievements in the field of Quantum Computing and excellent tools for students and researchers to program real Quantum Computers. Open source projects are fundamental for the development of QC, to share knowledge and build communities and ecosystems. Discussion channels and forums are indeed essential to progress, to share ideas and strengthen the software application.

The choice of one SDK with respect to the others depends on various factors. Obviously, the application one wants to develop will drive the choice of the framework. In addition, some syntax elements or certain statements of the programming language can be very important. However, one of the most relevant factors is definitely the ability to run the designed circuit on simulators and on real quantum hardware. Some SDKs, as seen in the previous chapter, take the path of being hardware agnostic, while others are more focused on a specific hardware architecture. Moreover, some Software Development Kits are designed to focus on current (or near-term) NISQ architectures, while others think about future Quantum Computers with a higher number of qubits, more connectivity and also *fault-tolerant*, i.e. without errors in computation.

This leads to another discussion: which of these two approaches is better? There is no definitive answer to this question, as each direction has its own advantages and disadvantages. On one hand, the first approach allows the designed circuit to run on current hardware architectures, which are constantly growing and improving. SDKs such as Cirq and t$|ket\rangle$ are of this type; the goal of optimizing algorithms and applications can achieve advances in QC architectures, but at the same time it can represent a limitation. The second approach indeed, not focusing only to NISQ hardwares, offers the opportunity to design circuit without being constrained by the limitation of the present architectures (such happens in Qiskit or in Q#). In addition, this second framework allows researchers to design *long-term* codes, without having to rewrite and adapt programs when new hardwares becomes available. As a disadvantage, the design of these SDKs is more complex, since it is not easy to organize a flexible structure that is able to translate the code on any physical layout of the qubits. It is important to underline that current softwares are built to encode only notable vectors standing on the Bloch Sphere, so that the preparation of the quantum state is not completely free, but it is constrained starting from these vectors.

SDK	Simulator backend availability	Own QPU backend availability	Level of agnosticism	Level of interoperability
Qiskit	✓	✓	Low	Medium
Cirq	✓	✗	High	Medium
Q#	✓	✓	High	Medium
t$|ket\rangle$	✓	✓	High	High
QMatcha TEA	✓	✗	Medium	High

Table 5.1: Comparison of the main features of the analyzed frameworks.

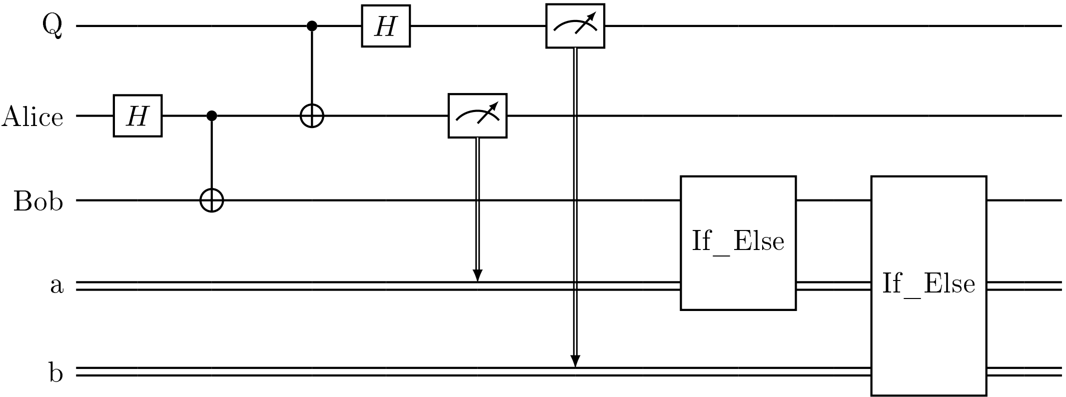
Another important consideration when choosing a QC framework is the interoperability with other programming languages and with hardwares from different vendors. This allows to take advantage of the strenghts of different platforms and to combine them to solve a specific problem. In this sense, Q# and QMatcha TEA are good examples, since they are designed to be interoperable with other frameworks. Moreover, Cirq provides an interface with third parties hardwares, validating the designed circuit to be exectuted following the constraints of the target architecture. The various SDKs also have different levels of hardware agnosticism, focusing on a specific type of hardware (such as the superconducting for Qiskit) or offering the ability to design algorithms to be run on different technologies.

The Tab. 5.1 summarizes the main features of the analyzed frameworks, highlighting some differences between them.

Gate-based Quantum Computing SDKs are powerful tools, but like any software, they have their own set of challenges, limitations, and potential drawbacks. A common problem associated with these SDKs is certainly the complexity for beginners, as learning how to code a quantum algorithm is not trivial, since it requires knowledge of quantum mechanics. Documentation and tutorials are essential to help users to learn how to use the framework and to understand the quantum concepts behind the code, but apart from the fact of not always being exaustive, they do not always cover the interoperability with other frameworks. Each SDK is designed with its own purposes and specific domains of application, so in most of the cases users have to make a single choice for the development of their use cases. This is indeed a very important aspect of quantum programming, since it is really difficult (if not impossible) to formulate Quantum Computing languages which are universal.

5.2 Quantum Teleportation Protocol

The quantum teleportation protocol is a fundamental protocol in quantum computing. It is a protocol that allows the state of a qubit to be transferred from one location to another, without physically moving the qubit itself. A sender (Alice) transmits a qubit state to a receiver (Bob) by making use of a shared entangled state, together with two bits of classical communication. Alice holds a qubit A, Bob holds a qubit B, and together the pair (A, B) is in the entangled state $|\phi^+\rangle$. It could be, for instance, that Alice and Bob were at the same place in the past, they prepared the qubits A and B in the state $|\phi^+\rangle$, and then went their separate ways with their qubits in hand. Or it could be that some other process, such as one involving a third party or a complex distributed process, was used to create this shared entangled state. These details are not part of the teleportation protocol itself. Alice then comes into possession of a third qubit Q which she wishes to send to Bob. The state of the qubit Q is assumed to be unknown to Alice and Bob, and no assumptions are made about it. For example, the qubit Q might be entangled with one or more other systems that neither Alice nor Bob can access. To say that Alice wants to transmit the qubit Q to Bob means that Alice would like Bob to be holding a qubit that is in the same state that Q was at the start of the protocol, with all the correlations that Q had with other systems, as if Alice had physically handed Q to Bob. The quantum circuit implementation of the this protocol is show below.



The quantum teleportation can be divided into various steps:

1. Alice and Bob share an entangled pair of qubits $|\phi^+\rangle$.
2. Alice performs a controlled-NOT operation on the pair (A, Q), with Q being the control and A being the target, and then performs a Hadamard operation on Q.
3. Alice then measures both A and Q, with respect to a standard basis measurement in both cases, and transmits the classical outcomes to Bob. The outcomes of the measurement of A and B are referenced as a and b respectively.
4. Bob receives a and b from Alice, and depending on the values of these bits he performs these operations:
   - if $a = 1$ then Bob performs a bit flip applying a Pauli-X gate (NOT) on his qubit B.
   - if $b = 1$ then Bob performs a phase flip applying a Pauli-Z gate on his qubit B.

   That is, conditioned on ab being 00, 01, 10, or 11, Bob performs one of the operations $\mathbb{I}$, $\mathbb{Z}$, $\mathbb{X}$, or $\mathbb{ZX}$ on the qubit B.

Therefore, the various possibilities of measure results for ab are:

- Alice measures $|00\rangle$: Bob does nothing. $\mathbb{I}\,|B\rangle$
- Alice measures $|01\rangle$: Bob applies a Pauli-X gate. $\mathbb{X}\,|B\rangle$
- Alice measures $|10\rangle$: Bob applies a Pauli-Z gate. $\mathbb{Z}\,|B\rangle$
- Alice measures $|11\rangle$: Bob applies a Pauli-X gate followed by a Pauli-Z gate. $\mathbb{XZ}\,|B\rangle$

This is the complete description of the teleportation protocol, it has effectively implemented a perfect qubit communication channel, where the state of Q has been "teleported" into B. When the protocol is finished, the state of the qubit Q will have changed from its original value to $|b\rangle$ as a result of the measurement performed on it. Note also that the initial shared entangled state has effectively been "burned" in the process: the state of A has changed to $|a\rangle$ and is no longer entangled with B (or any other system). This is the *cost* of teleportation.

To analyze the teleportation protocol, the behavior of the circuit described above is now examined, one step at a time, beginning with the situation in which the qubit Q is initially in the state $\alpha\,|0\rangle + \beta\,|1\rangle$. With this assumption on Q, the state of the three qubits (B, A, Q) together at the start of the protocol is therefore

$$|\pi_0\rangle = |\phi^+\rangle \otimes \Big(\alpha\,|0\rangle + \beta\,|1\rangle\Big) = \frac{\alpha\,|000\rangle + \alpha\,|110\rangle + \beta\,|001\rangle + \beta\,|111\rangle}{\sqrt{2}}$$

The first gate that is performed is the controlled-NOT gate, which transforms the state $|\pi_0\rangle$ into

$$|\pi_1\rangle = \frac{\alpha\,|000\rangle + \alpha\,|110\rangle + \beta\,|011\rangle + \beta\,|101\rangle}{\sqrt{2}}$$

Then the Hadamard gate is applied, which transforms the state $|\pi_1\rangle$ into

$$|\pi_2\rangle = \frac{\alpha|00\rangle|+\rangle + \alpha|11\rangle|+\rangle + \beta|01\rangle|-\rangle + \beta|10\rangle|-\rangle}{\sqrt{2}}$$
$$= \frac{\alpha|000\rangle + \alpha|001\rangle + \alpha|110\rangle + \alpha|111\rangle + \beta|010\rangle - \beta|011\rangle + \beta|100\rangle - \beta|101\rangle}{\sqrt{2}}$$

Using the multilinearity of the tensor product, we may alternatively write this state as follows:

$$\begin{aligned}|\pi_2\rangle &= \frac{1}{2}\Big(\alpha|0\rangle + \beta|1\rangle\Big)|00\rangle \\ &+ \frac{1}{2}\Big(\alpha|0\rangle - \beta|1\rangle\Big)|01\rangle \\ &+ \frac{1}{2}\Big(\alpha|1\rangle + \beta|0\rangle\Big)|10\rangle \\ &+ \frac{1}{2}\Big(\alpha|1\rangle - \beta|0\rangle\Big)|11\rangle\end{aligned}$$

Considering the four possible outcomes of Alice's standard basis measurements, it is possible to examine the actions that Bob performs on his qubit.

- $ab = 00$: The outcome of Alice's measurement is $|00\rangle$ with probability

$$\left\|\frac{1}{2}\Big(\alpha|0\rangle + \beta|1\rangle\Big)\right\|^2 = \frac{|\alpha|^2 + |\beta|^2}{4} = \frac{1}{4}$$

  in which case the state of (B, A, Q) becomes

$$\Big(\alpha|0\rangle + \beta|1\rangle\Big)|00\rangle$$

  Bob does nothing in this case, and so this is the final state of these three qubits.

- $ab = 01$: The outcome of Alice's measurement is $|01\rangle$ with probability

$$\left\|\frac{1}{2}\Big(\alpha|0\rangle - \beta|1\rangle\Big)\right\|^2 = \frac{|\alpha|^2 + |-\beta|^2}{4} = \frac{1}{4}$$

  in which case the state of (B, A, Q) becomes

$$\Big(\alpha|0\rangle - \beta|1\rangle\Big)|01\rangle$$

  In this case Bob applies a $\mathbb{Z}$ gate to B, leaving (B, A, Q) in the state

$$\Big(\alpha|0\rangle + \beta|1\rangle\Big)|01\rangle$$

- $ab = 10$: The outcome of Alice's measurement is $|10\rangle$ with probability

$$\left\|\frac{1}{2}\Big(\alpha|1\rangle + \beta|0\rangle\Big)\right\|^2 = \frac{|\alpha|^2 + |\beta|^2}{4} = \frac{1}{4}$$

  in which case the state of (B, A, Q) becomes

$$\Big(\alpha|1\rangle + \beta|0\rangle\Big)|10\rangle$$

  In this case Bob applies a $\mathbb{X}$ gate to B, leaving (B, A, Q) in the state

$$\Big(\alpha|0\rangle + \beta|1\rangle\Big)|10\rangle$$

- $ab = 11$: The outcome of Alice's measurement is $|11\rangle$ with probability

  $$\left\| \frac{1}{2}\Big(\alpha\,|1\rangle - \beta\,|0\rangle\Big) \right\|^2 = \frac{|\alpha|^2 + |-\beta|^2}{4} = \frac{1}{4}$$

  in which case the state of (B, A, Q) becomes

  $$\Big(\alpha\,|1\rangle - \beta\,|0\rangle\Big)\,|11\rangle$$

  In this case Bob performs the operations $\mathbb{ZX}$ on the qubit B, leaving (B, A, Q) in the state

  $$\Big(\alpha\,|0\rangle + \beta\,|1\rangle\Big)\,|11\rangle$$

In all four cases, that Bob's qubit B is left in the state $\alpha\,|0\rangle + \beta\,|1\rangle$ at the end of the protocol, which is the initial state of the qubit Q.

5.2.1 Teleportation Protocol with the analyzed SDKs

In this section, the quantum teleportation protocol is coded in the analyzed SDKs, in order to highlight the differences in the implementation. Indeed, each SDK has its own syntax and its own way of representing quantum circuits, providing an additional level of comparison between the different frameworks.

Teleportation in Qiskit

The following code shows the implementation of the quantum teleportation protocol in Qiskit.

```
1  from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
2
3  qubit = QuantumRegister(1, "Q")     # Create a q register - Qubit to be teleported
4  ent0 = QuantumRegister(1, "A")      # Create a q register - Alice
5  ent1 = QuantumRegister(1, "B")      # Create a q register - Bob
6  a = ClassicalRegister(1, "a")       # Create a cl register to store measurement
7  b = ClassicalRegister(1, "b")       # Create a cl register to store measurement
8
9  # Create a quantum circuit with the created registers
10 protocol = QuantumCircuit(qubit, ent0, ent1, a, b)
11
12 # Prepare the entangled state used for teleportation
13 protocol.h(ent0)           # Apply a Hadamard gate to the first qubit
14 protocol.cx(ent0, ent1)   # Apply a CNOT gate using A as control and B as target
15 protocol.barrier()
16
17 # Alice's operations
18 protocol.cx(qubit, ent0)   # Apply a CNOT gate using Q as control and A as target
19 protocol.h(qubit)           # Apply a Hadamard gate to the Q qubit
20 protocol.barrier()
21
22 # Alice measures and sends classical bits to Bob
23 protocol.measure(ent0, a)   # Measure Alice's qubit A and store the result in bit a
24 protocol.measure(qubit, b)   # Measure qubit Q and store the result in bit b
25 protocol.barrier()
26
```

```
27  # Bob uses the classical bits to conditionally apply gates
28  with protocol.if_test((a, 1)):      # If 'a' is 1, apply X gate to qubit B
29      protocol.x(ent1)
30  with protocol.if_test((b, 1)):      # If 'b' is 1, apply Z gate to qubit B
31      protocol.z(ent1)
```

After importing the necessary libraries, the first step is to create the quantum and classical registers. Then, the quantum circuit is created on the registers, and the shared entangled state is prepared for the teleportation protocol. The circuit first initializes (A, B) to be in a $|\phi^+\rangle$ state (which is not part of the protocol itself), followed by Alice's operations, then her measurements, and finally Bob's operations. The first two gates are the Hadamard gate and the CNOT gate, which entangles the two qubits. Then, the controlled-NOT gate and the Hadamard gate are applied to the qubit to be teleported. After that, the two qubits are measured and the classical bits are sent to Bob. Finally, Bob applies the Pauli gates depending on the classical bits received from Alice, using the method `if_test()`.

The circuit makes use of a few features of Qiskit such as the `barrier` and `if_test` functions. The `barrier` function creates a visual separation making the circuit diagram more readable, and it also prevents Qiskit from performing various simplifications and optimizations across barriers during compilation when circuits are run on real hardware. The `if_test` function applies an operation conditionally depending on a classical bit or register.

Teleportation in Cirq

The following code shows the implementation of the quantum teleportation protocol in Cirq.

```
1   import cirq
2
3   circuit = cirq.Circuit()
4
5   # Prepare the three qubits involved in the teleportation protocol
6   # Q -> the qubit to be teleported
7   # A -> Alice's qubit
8   # B -> Bob's qubit
9
10  Q , A , B = cirq.LineQubit.range(3)
11
12  # Create the entangled pair between Alice and Bob
13  circuit.append([cirq.H(A), cirq.CNOT(A, B)])
14
15  # Measurements of the message Q and Alice's entangled qubit A
16  circuit.append([cirq.CNOT(Q, A), cirq.H(Q)])
17  circuit.append(cirq.measure(Q, key='Q'))
18  circuit.append(cirq.measure(A, key='A'))
19
20  # Use two classical bits to recover the original quantum state on Bob's qubit
21  # if Alice's qubit is 1, then apply a NOT gate on B
22  # if Q is 1, then apply a Z gate on B
23  circuit.append(cirq.X(B).with_classical_controls('A'))
24  circuit.append(cirq.Z(B).with_classical_controls('Q'))
```

In this SDK the qubits are not in a register but are created individually in a linrear array. Then the operations are appended to the circuit, and the `with_classical_controls` method is used to apply the Pauli gates conditionally on the classical bits received from Alice.

Teleportation in Q#

In the next block of code there is the implementation of the quantum teleportation protocol in Q#.

```
namespace Teleport{
open Microsoft.Quantum.Arrays;
open Microsoft.Quantum.Canon;
open Microsoft.Quantum.Measurement;
open Microsoft.Quantum.Intrinsic;
open Microsoft.Quantum.Preparation;

// This operation prepares an entangled Bell pair between two qubits.
operation PrepareBellPair(left: Qubit, right: Qubit) : Unit is Adj + Ctl {
    H(left);
    CNOT(left, right);
}

// The teleportation protocol
@EntryPoint()
operation TeleportProtocol(prepBasis: Pauli, measBasis: Pauli) : Result {
    use Q = Qubit();   // The qubit to be teleported
    use A = Qubit();   // Alice's qubit
    use B = Qubit();   // Bob's qubit

    PreparePauliEighenstate(prepBasis, Q); // Prepare Q in a random state
    PrepareBellPair(A, B);             // Prepare the entangled Bell pair A-B
    Adjoint PrepareBellPair(Q, A);     // Prepare the entangled Bell pair A-Q

    if (MresetZ(A) == One) { X(B); }   // Apply X gate to B if A is 1
    if (MResetZ(Q) == One) { Z(B); }   // Apply Z gate to B if Q is 1

    let result = Measure([measBasis], [B]); // Measure B in the desired basis
    Reset(B);

    return result;                 // Return the measurement result of B
  }
}
```

First of all the namespace is defined, opening the necessary libraries. Then, the `PrepareBellPair` operation is defined, which prepares an entangled Bell pair between two qubits. Finally, the teleportation protocol is implemented, using the `PreparePauliEighenstate` operation to prepare the qubit to be teleported in a random state. The operations on the Bob's qubit are applied conditionally on the classical bits received from Alice, using `if` statements with respect to the `MResetZ` operation, which measures the qubit in the Z basis and resets it to $|0\rangle$. At the end, the state of the qubit B is measured in the desired basis and the result is returned as output.

Teleportation in t$|ket\rangle$

Following the implementation of the protocol using t$|ket\rangle$ SDK.

```
from pytket.circuit import Circuit

# Create a circuit with 3 qubits and 2 classical bits
# Qubit 0 -> Qubit to be teleported
```

```
5   # Qubit 1 -> Qubit A
6   # Qubit 2 -> Qubit B
7   circuit = Circuit(3,2)
8
9   # Prepare the Bell state A-B
10  circuit.H(1)
11  circuit.CX(1,2)
12
13  # Prepare the Bell state Q-A
14  circuit.CX(0,1)
15  circuit.H(0)
16
17  # Perform measurements of Q and A
18  circuit.Measure(0,0)
19  circuit.Measure(1,1)
20
21  # Apply corrections to B wrt the measurement results
22  circuit.X(2, condition_bits=[1])   # Apply X to B if A is 1
23  circuit.Z(2, condition_bits=[0])   # Apply Z to B if Q is 1
```

First of all the circuit is created, specifying the number of qubits and classical bits. Then, the Bell state is prepared between the qubits A and B, followed by the preparation of the Bell state between the qubits Q and A. After that, the qubits Q and A are measured and the Pauli gates are applied conditionally on the classical bits received from Alice, using the `condition_bits` parameter of the `X` and `Z` methods.

Teleportation in QMatcha TEA

The following code shows the implementation of the quantum teleportation protocol in QMatcha TEA.

```
1   from qmatchatea.circuit import Qcircuit, ClassicalCondition
2   from qmatchatea.py_emulator import QcMps
3   from qmatchatea import QCConvergenceParameters
4
5   # Instantiate the Qcircuit with Alice qubits
6   # Qubit 0 -> Qubit to be teleported
7   # Qubit 1 -> Qubit A
8
9   num_qubit = 2
10  qc = Qcircuit(num_qubit)
11
12  # Create Bell's pair
13  qc.h(0, qreg="default")
14  qc.cx([0, 1], qreg=["default", "default"])
15
16  # Add Bob's register
17  qc.add_qregister(new_register="qbob", positions=[0], reference_register="default")
18
19  # Entangle Bob's qubit with one of Alice's
20  qc.cx([0, 0], qreg=["qbob", "default"])
21  qc.h(0, qreg="qbob")
22
23  # Add the classical register necessary to store the output of a projective
```

```
24 # measurement in the circuit.
25 qc.add_cregister(name="cbob", num_bits=2)
26
27 # Measure the qubit in position 'pos' of the quantum register 'qreg'
28 # storing the result in the index 'cl_idx' of the classical register 'creg'
29 qc.measure_projective(pos=0, cl_idx=0, qreg="qbob", creg="cbob")
30 qc.measure_projective(pos=0, cl_idx=1, qreg="default", creg="cbob")
31
32 # Define the classically conditioned gates, i.e. gates that will be applied
33 # based on the result of a previous measurement.
34 cond_x = ClassicalCondition(cregister="cbob", value=1, idx=0)
35 qc.x(1, qreg="default", c_if=cond_x)  # Apply X gate if A is 1
36
37 cond_z = ClassicalCondition(cregister="cbob", value=1, idx=1)
38 qc.z(1, qreg="default", c_if=cond_z)  # Apply Z gate if Q is 1
39
40 mps = QcMps(num_qub, convergence_parameters=QCConvergenceParameters())
41 mps.run_from_qcirc(qc)
```

After the import of the necessary libraries, the quantum circuit is created, specifying the number of qubits. Later, the Bell state is prepared between the qubits Q and A, followed by the preparation of the entangled state between the qubits A and B. Moreover, the classical register necessary to store the output of a projective measurement in the circuit is added. Then, the qubits Q and A are measured and the Pauli gates are applied conditionally on the classical bits received from Alice (employed with `ClassicalCondition`), using the `c_if` parameter of the `x` and `z` methods. Finally, the circuit is run on the emulator using the default convergence parameters.

5.3 Cyclomatic Complexity Analysis

Cyclomatic complexity [63] is a software metric used to measure the complexity of a program's control flow. It is a quantitative measure of the number of independent paths through the code, which can be used to assess the code's complexity, maintainability, and potential points of failure. Therefore, a lower score (corresponding to lower complexity) is considered better, as it indicates a less convoluted codebase. Cyclomatic complexity is easy to extract from Python projects, using tools such as radon [64]. The concept of cyclomatic complexity was introduced by Thomas J. McCabe in 1976 and it is calculated based on the graph representation of the code's control flow, where nodes represent decision points (such as conditional statements) and edges represent the possible transitions between nodes. The formula to calculate this metric M is:

$$M = E - N + 2P$$

where:

- E is the number of edges in the control flow graph
- N is the number of nodes in the control flow graph
- P is the number of connected components (usually 1, unless the program has disconnected parts)

Cyclomatic complexity provides several insights:

1. Code readability and maintainability: Higher values of cyclomatic complexity indicate more complex code with multiple decision points and paths. Such code may be more difficult to read, understand, and maintain.

SDK	Cyclomatic Complexity
Qiskit	2.42
Cirq	2.77
Q#	2.61
t$\|ket\rangle$	2.91
Quantum Matcha TEA	3.05

Table 5.2: Cyclomatic complexity of the analyzed SDKs.

2. Testing: The cyclomatic complexity metric can be used as a guide for testing. It suggests the minimum number of test cases required to achieve full coverage of different paths in the code.
3. Error-prone code: Areas with high cyclomatic complexity tend to be more error-prone because they have a greater number of possible paths, increasing the likelihood of errors.
4. Refactoring: Cyclomatic complexity can help identify parts of the code that might benefit from refactoring to simplify the control flow and reduce complexity.
5. Code reviews: Teams can use cyclomatic complexity as a basis for code reviews, helping to identify sections that might need improvement before being merged into the codebase.

Cyclomatic complexity is calculated using the radon library, which provides an interface for measuring the cyclomatic complexity of a Python project. This metric is computed for each Python file in the codebase, and the average of these values is taken as the final score, excluding the minimum and the maximum.

It's important to note that cyclomatic complexity is just one metric among many used to assess code quality. While it provides valuable insights into control flow complexity, it does not take into account other factors such as code duplication, function length, or overall design. In the following table 5.2, the cyclomatic complexity of the analyzed SDKs is shown.

From this table it is possible to see that the SDK with the lowest cyclomatic complexity is Qiskit. This is due to the fact that Qiskit is the most mature SDK, and it has been developed for a longer time with respect to the others. After that, Cirq and Q# have a similar result in terms of this metric, meaning that is in both cases the code is stable and well structured. As expected, t$|ket\rangle$ and Quantum Matcha TEA have the highest cyclomatic complexity, since they are newer SDKs with respect to the others. The latter two achieve a good result anyway, considering that the first rank score for the radon library has a cyclomatic complexity between 1 and 5.

Chapter 6

Conclusions

The research and analysis presented in this thesis has attempted to clarify the landscape of programming gate-based Quantum Computers, focusing on the comparative evaluation of various Software Development Kits for circuit design automation. Various features, capabilities, and limitations of prominent SDKs, including Qiskit, Cirq, and Q#, have been examined. This study has highlighted the importance of these tools in democratizing access to Quantum Computing and supporting advancements in quantum algorithm development. The comparative analysis has not only illustrated the distinctive programming paradigms employed by each SDK, but also revealed the underlying challenges and opportunities presented by quantum programming. As it turns out, the inherent complexity of quantum systems requires a fine balance between high-level abstraction and low-level control, and each SDK approaches this balance differently, responding to various user preferences and skill levels. These challenges show the nature of Quantum Computing and the ongoing efforts required to harness its full potential.

As QC moves forward, driven by technological advancements and innovative research, the SDKs analyzed in this work contribute to its development. The decision of which SDK to use depends on factors ranging from user expertise and project requirements, to the SDK's community support and adaptability to evolving quantum hardware. As quantum technologies continue to mature, these SDKs will continue to evolve, transforming the theoretical potential of Quantum Computing into practical solutions for complex challenges across a wide range of disciplines.

In the end, Quantum Computing is not a strange way of doing special calculations, but a new way of thinking about computation. QC can be used to experiment and see what new things can be built. There is no telling what we will discover, but now is the time for exploration and innovation.

Probably, the greatest years for Quantum Computing are ahead of us.

Bibliography

[1] Turing A. *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society. 1936 Nov. Available from: https://doi.org/10.1112/plms/s2-42.1.230.

[2] Nielsen MA, Chuang IL. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press; 2010.

[3] Riordan M, Hoddeson L. *Crystal Fire: The Birth of the Information Age*. Norton. 1997 Jan.

[4] Sammet JE. *Programming Languages: History and Fundamentals*. Prentice-Hall; 1969.

[5] Faggin F, Hoff ME, Mazor S, Shima M. *The History of the 4004*. IEEE Micro. 1996 Jun. Available from: https://doi.org/10.1109/40.546561.

[6] Hafner K, Lyon M. *Where Wizards Stay up Late: The Origins of the Internet*. Simon & Schuster; 1996.

[7] Bernhardt C. *Quantum Computing for Everyone*. The MIT Press; 2020.

[8] Preskill J. *Quantum Computing in the NISQ era and beyond*. Quantum. 2018 Aug. Available from: https://doi.org/10.22331%2Fq-2018-08-06-79.

[9] Shor PW. *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*. SIAM Journal on Computing. 1997 Oct. Available from: https://doi.org/10.1137%2Fs0097539795293172.

[10] Grover LK. *A fast quantum mechanical algorithm for database search*. ACM Symposium on the Theory of Computing. 1996 Jul. Available from: https://doi.org/10.1145/237814.237866.

[11] Sutor R. *Dancing with Qubits*. Packt Publishing; 2019.

[12] Aaronson S. *Quantum Computing since Democritus*. Cambridge University Press; 2013.

[13] Wong T. *Introduction to Classical and Quantum Computing*. Rooted Grove; 2022.

[14] Bernstein E, Vazirani U. *Quantum Complexity Theory*. SIAM Journal on Computing. 1997 Jun. Available from: https://doi.org/10.1137/S0097539796300921.

[15] Coecke B, Kissinger A. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press; 2017.

[16] Lian B, Sun XQ, Vaezi A, Qi XL, Zhang SC. *Topological quantum computation based on chiral Majorana fermions*. Proceedings of the National Academy of Sciences. 2018 Oct. Available from: https://doi.org/10.1073/pnas.1810003115.

[17] Loss D, DiVincenzo DP. *Quantum computation with quantum dots*. Phys Rev A. 1998 Jan. Available from: https://doi.org/10.1103/PhysRevA.57.120.

[18] Weber JR, Koehl WF, Varley JB, Janotti A, Buckley BB, de Walle CGV, et al. *Quantum computing with defects*. Proceedings of the National Academy of Sciences. 2010 Apr. Available from: https://doi.org/10.1073/pnas.1003052107.

[19] Kjaergaard M, Schwartz ME, Braumüller J, Krantz P, Wang JIJ, Gustavsson S, et al. *Superconducting Qubits: Current State of Play*. Annual Review of Condensed Matter Physics. 2020 Mar. Available from: https://doi.org/10.1146/annurev-conmatphys-031119-050605.

[20] Bruzewicz CD, Chiaverini J, McConnell R, Sage JM. *Trapped-ion quantum computing: Progress and challenges*. Applied Physics Reviews. 2019 May. Available from: https://doi.org/10.1063/1.5088164.

[21] Madsen L, Laudenbach F, Askarani M, Rortais F, Vincent T, Bulmer J, et al. *Quantum computational advantage with a programmable photonic processor*. Nature. 2022 Jun. Available from: https://doi.org/10.1038/s41586-022-04725-x.

[22] Henriet L, Beguin L, Signoles A, Lahaye T, Browaeys A, Reymond GO, et al. *Quantum computing with neutral atoms*. Quantum. 2020 Sep. Available from: https://doi.org/10.22331%2Fq-2020-09-21-327.

[23] Cao Y, Romero J, Olson JP, Degroote M, Johnson PD, Kieferová M, et al. *Quantum Chemistry in the Age of Quantum Computing*. Chemical Reviews. 2019 Aug. Available from: https://doi.org/10.1021/acs.chemrev.8b00803.

[24] Nachman B, Provasoli D, de Jong WA, Bauer CW. Quantum Algorithm for High Energy Physics Simulations. Phys Rev Lett. 2021 Feb. Available from: https://doi.org/10.1103/PhysRevLett.126.062001.

[25] Herman D, Googin C, Liu X, Galda A, Safro I, Sun Y, et al. *A Survey of Quantum Computing for Finance*. Qunatum Physics. 2022 Jan. Available from: https://doi.org/10.48550/arXiv.2201.02773.

[26] Ajagekar A, You F. *Quantum computing for energy systems optimization: Challenges and opportunities*. Energy. 2019 Apr. Available from: https://doi.org/10.1016/j.energy.2019.04.186.

[27] Kadowaki T, Nishimori H. *Quantum annealing in the transverse Ising model*. Phys Rev E. 1998 Nov. Available from: https://doi.org/10.1103/PhysRevE.58.5355.

[28] Biamonte J, Wittek P, Pancotti N, Rebentrost P, Wiebe N, Lloyd S. *Quantum Machine Learning*. Nature. 2017 Sep. Available from: https://doi.org/10.1038/nature23474.

[29] Schuld M, Petruccione F. *Supervised Learning with Quantum Computers*. Springer; 2018.

[30] Scarani V, Bechmann-Pasquinucci H, Cerf N Nicolas J, Peev M. *The security of practical quantum key distribution*. Rev Mod Phys. 2009 Sep. Available from: https://link.aps.org/doi/10.1103/RevModPhys.81.1301.

[31] Hey T. *Quantum computing: An introduction*. Computing & Control Engineering Journal. 1999.

[32] Vietz, Daniel, Barzen, Johanna, Leymann, Frank, et al. *On Decision Support for Quantum Application Developers: Categorization, Comparison, and Analysis of Existing Technologies*. Computational Science. 2021 Jun. Available from: https://doi.org/10.1007/978-3-030-77980-1_10.

[33] Yanofsky NS, Mannucci MA. *Quantum Computing for Computer Scientists*. Cambridge University Press; 2008.

[34] Heim B, Soeken M, Marshall S, Granade C, Roetteler M, Geller A, et al. *Quantum Programming Languages*. Nature Reviews Physics. 2020 Nov. Available from: https://doi.org/10.1038/s42254-020-00245-7.

[35] Cross AW, Bishop LS, Smolin JA, Gambetta JM. *Open Quantum Assembly Language*. Quantum Physics. 2017. Available from: https://doi.org/10.48550/arXiv.1707.03429.

[36] Microsoft. *Microsoft Quantum Development Kit - Q#*; 2021. Available from: https://azure.microsoft.com/en-us/products/quantum/.

[37] contributors IQ. Qiskit: An Open-source Framework for Quantum Computing; 2017. Available from: https://doi.org/10.5281/zenodo.2573505.

[38] Google. *Cirq*; 2018. Available from: https://quantumai.google/cirq.

[39] IBM. *IBM Quantum Composer*;. Available from: https://quantum-computing.ibm.com/composer/docs/iqx.

[40] IBM. *IBM Quantum Lab*;. Available from: https://lab.quantum-computing.ibm.com/.

[41] Hidary JD. *Quantum Computing: An Applied Approach*. Springer; 2019.

[42] Quantinuum. *t|ket>: a retargetable compiler for NISQ devices*. Quantum Science and Technology. 2020 Nov. Available from: https://dx.doi.org/10.1088/2058-9565/ab8e92.

[43] IBM. *Qiskit Homepage*;. Available from: https://qiskit.org/.

[44] IBM. *Qiskit GitHub Page*;. Available from: https://github.com/Qiskit.

[45] IBM. *Qiskit Documentation*;. Available from: https://qiskit.org/documentation/.

[46] IBM. *Qiskit Textbooks*;. Available from: https://qiskit.org/learn.

[47] IBM. *IBM Qiskit Slack Channel*;. Available from: https://qiskit.slack.com/.

[48] Google. *Cirq GitHub Page*;. Available from: https://github.com/quantumlib/Cirq.

[49] Google. *Cirq Documentation*;. Available from: https://quantumai.google/cirq/build.

[50] Google. *Cirq Stack Exchange Forum*;. Available from: https://quantumcomputing.stackexchange.com/questions/tagged/cirq.

[51] Microsoft. *Q# GitHub Page*;. Available from: https://github.com/Microsoft/Quantum.

[52] Microsoft. *Q# Documentation*;. Available from: https://learn.microsoft.com/en-us/azure/quantum/?view=qsharp-preview.

[53] Microsoft. *Q# Quantum Katas*;. Available from: https://github.com/Microsoft/QuantumKatas/.

[54] Quantinuum. *t|ket> Homepage*;. Available from: https://www.quantinuum.com/developers/tket.

[55] Quantinuum. *t|ket> GitHub Page*;. Available from: https://github.com/CQCL/tket.

[56] Quantinuum. *t|ket> Documentation*;. Available from: https://cqcl.github.io/tket/pytket/api/.

[57] Quantinuum. *t|ket⟩ Slack Channel*;. Available from: https://tketusers.slack.com/.

[58] Quantinuum. *t|ket⟩ Stack Exchange Forum*;. Available from: https://quantumcomputing.stackexchange.com/questions/tagged/pytket.

[59] INFN, UniPD. *Quantum Tea Homepage*;. Available from: https://www.quantumtea.it/.

[60] INFN, UniPD. *Quantum Matcha Tea GitLab Page*;. Available from: https://baltig.infn.it/quantum_matcha_tea/py_api_quantum_matcha_tea.

[61] INFN, UniPD. *Quantum Matcha Tea Documentation*;. Available from: https://quantum_matcha_tea.baltig-pages.infn.it/py_api_quantum_matcha_tea/.

[62] Xanadu. *Strawberry Fields*;. Available from: https://strawberryfields.ai/.

[63] McCabe TJ. *A Complexity Measure*. IEEE Transactions on Software Engineering. 1976 Dec. Available from: https://doi.org/10.1109/TSE.1976.233837.

[64] MIT. *Radon Documentation*;. Available from: https://radon.readthedocs.io/en/latest/index.html.

[65] Feynman RP. *Feynman Lectures on Computation*. Addison-Wesley Publishing Company, Inc. Reading, Massachusetts; 1996.

[66] Hundt R. *Quantum Computing for Programmers*. Cambridge University Press; 2022.

[67] Hassija V, Chamola V, Saxena V, Chanana V, Parashari P, Mumtaz S, et al. *Present landscape of Quantum Computing*. IET Quantum Communication. 2020 Dec. Available from: https://doi.org/10.1049/iet-qtc.2020.0027.

[68] Fingerhuth M, Babej T, Wittek P. *Open source software in Quantum Computing*. PLOS ONE. 2018 Dec. Available from: https://doi.org/10.1371%2Fjournal.pone.0208561.

[69] LaRose R. *Overview and Comparison of Gate Level Quantum Software Platforms*. Quantum. 2019 Mar. Available from: https://doi.org/10.22331%2Fq-2019-03-25-130.

[70] Garhwal S, Ghorani M, Ahmad A. *Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages*. Archives of Computational Methods in Engineering. 2019 Dec. Available from: https://doi.org/10.1007/s11831-019-09372-6.