POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

A quantum circuit library for image processing

Supervisors
Prof. Maurizio Zamboni
Prof.ssa Mariagrazia Graziano
Prof.ssa Giovanna Turvani

Candidate
Chiara Dolciami

October 2022

Summary

Today's interest in applications like pattern recognition and computer vision makes image processing algorithms of great importance. However, the rapidly increasing volume of visual information weighs on the computational capabilities currently available in classical computers. Quantum Image Processing (QImP) focuses on providing a counterpart of conventional image processing strategies in the Quantum Computing domain, exploiting its intrinsic parallel nature. Over the years, many QImP algorithms have been proposed to encode and process images using quantum formalism. Despite this, in the state-of-the-art, not enough room is given for direct and practical comparisons between the available techniques. Therefore, difficulties arise when trying to understand whether they represent effective opportunities with respect to classical counterparts, especially when considering the limitations and non-idealities of nowadays quantum hardware. The goal of this thesis is to define a Python software library of QImP algorithms compatible with Qiskit, an open-source software-development kit for quantum computing, to provide users with the ability to flexibly compare the different techniques on reference input images and analyze their suitability through particular figures of merit. First, a preliminary study of the current literature on QImP has been carried out in order to identify the most promising algorithms. Then, they have been implemented as parametrical modules, which progressively formed the library. Jupyter Notebooks were considered to provide a practical user guide for a conscious application of the algorithms provided. The selection of the supported techniques spans from encoding methods, basic processing tools, compression, and edge detection algorithms and takes into account the limited computational resources of quantum hardware and the possibility of practical applications. Tests have been conducted on all the implemented circuits, both through simulations on classical computers and tests on real quantum hardware. The strengths and weaknesses in the application of the different algorithms have been put into evidence considering several use cases. This thesis lays the foundations for exploring the QImP scenario, while the implemented library gives the possibility to include and characterize new algorithms and compare them with the others, thanks to its flexible and modular nature.

Contents

Chapter 1

Introductory concepts

Quantum Computing is an interdisciplinary subject that intersects physics, mathematics and computer science. Its goal is to find methods to exploit natural quantum-mechanical effects to do information processing, overcoming the limited computational capabilities of conventional computation[1]. Its applications span a wide range of topics, including the optimization of problem solving and the simulation of many scenarios, e.g. in finance and machine learning. This first chapter is aimed at giving the reader the tools to understand the working principles of quantum computing, starting from the mathematical concepts from which it builds off. An in-depth presentation of Quantum Computing can be found in [2].

1.1 Quantum Systems

Analogously to what happens in classical computers, quantum computers use quantum bits, called **qubits**. Qubits are the smallest unit of information and they are implemented using two-dimensional quantum systems. The physical quantities that are canonically used for this purpose are the spin of a particle or excited states of atoms [2].

By assembling more qubits, it is possible to obtain quantum systems whose dynamics are described by complex vector spaces. These vector spaces are based on complex numbers, which are defined by the expression:

$$c = a + b \times i = a + bi, \tag{1.1}$$

$$a, b \ \in \mathcal{R} \tag{1.2}$$

where a is called the real part of c and b its imaginary part of c. The set of all complex numbers is denoted as $\mathcal{C}$. The canonical representation of quantum states is done via column vectors of complex elements. The state of a qubit, being

a two-dimensional quantum system, is therefore given by

$$|\Psi\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \alpha|0\rangle + \beta|1\rangle \; where \; \alpha, \beta \in \mathbb{C} \tag{1.3}$$

The difference between a standard bit and a qubit lies in the inherent indeterminacy on the knowledge of the physical state of a quantum particle. While a bit can only assume one of its two possible values, the qubit state's is denoted as a linear combination of its basis states, weighted by α and β. These complex parameters are called **probabilities amplitudes** and their magnitude squared represents the probability of measuring one of the two possible outcomes, i.e $|0\rangle$ and $|1\rangle$ whose sum must therefore be 1:

$$|\alpha|^2 + |\beta|^2 = 1. \tag{1.4}$$

The reason why the probabilities are described by complex numbers is that, when complex numbers are added together, they can cancel each other out, meaning that amplitudes can lower their probability. This is related to a quantum mechanics physical phenomenon called **interference**, for which an individual particle, being into more than one state at the same time, can cross its own trajectory and interact with itself, actually interfering with the probability of being in a state or another.

To describe the state of a quantum system obtained from assembling qubits, it is necessary to perform an operation called tensor product between the state vectors describing its starting qubits. Taking for example two qubits whose state vectors are:

$$|\Psi\rangle = \begin{bmatrix}a\\ b\end{bmatrix}, \; |\Phi\rangle = \begin{bmatrix}c\\ d\end{bmatrix}, \tag{1.5}$$

their tensor product is defined as:

$$|\Psi\rangle \otimes |\Phi\rangle = \begin{bmatrix}a\\ b\end{bmatrix} \otimes \begin{bmatrix}c\\ d\end{bmatrix} = \begin{bmatrix}a \cdot c\\ a \cdot d\\ b \cdot c\\ b \cdot d\end{bmatrix} \tag{1.6}$$

Equivalent ways of representing qubits of the same system can be:

- $|\Psi\rangle \otimes |\Phi\rangle$
- $|\Psi\rangle|\Phi\rangle$
- $|\Psi\Phi\rangle$

1.2 Superposition and Measurement

As seen in the previous section, a single particle can be in a combination of different states weighted by specific probability amplitudes. This phenomenon, referred to as **superposition**, can be extended to an arbitrary number of qubits and it is exploited in quantum computing to condense more information into the same set of qubits. This effect, though, does not survives after a measurement is performed on the system. The quantum information is lost irreversibly and classical information is obtained, making the measurement a destructive operation. Therefore re-obtaining the information embedded in the states of a quantum system becomes a statistical process. For example, if the state of a qubit is given by $|\Psi\rangle = \begin{bmatrix} a & b \end{bmatrix}^T$, the result of the measurement can be:

- $|0\rangle$ with probability $|\alpha|^2$
- $|1\rangle$ with probability $|\beta|^2$

After the measurement is performed, the state of the system collapses into one of the basis states and any further measurement will therefore give the same result with a probability equal to 1.

The goal of quantum algorithms is therefore to exploit the parallelism made possible by the superposition of states, which allows evaluating more inputs at same time, and take advantage of the interference phenomenon to increase the probability amplitude associated with the states representing the sought-after solution, in order to make sure it will be the one measured.

1.3 Entanglement

Another peculiarity of quantum systems is **entangled** states. The basic states of assembled systems are obtained with the tensor product of the basic states of its constituents, these are called **separable** states. But they are not the only states that quantum systems can be found in. For example, if a 2-qubit quantum system is described

$$|\Psi\rangle = \alpha|00\rangle + \beta|11\rangle$$

its state cannot be expressed as a tensor product of two basic states. The two states are intimately related to one another in such a way that performing a measurement on one of the two qubits will determine the state of the other as well. If, for example, the result of a measurement on the first qubit is $|0\rangle$, the second one will necessarily be $|0\rangle$. By exploiting entangled states, quantum algorithms can speed-up their processing capabilities since processing and measuring one qubit will reveal information about the entangled one.

1.4 Bloch Sphere

The **Bloch Sphere** is a unitary radius sphere, centered in the origin that serves as a graphic representation of the state of a qubit and can be a valuable way to understand one-qubit operations. The previous definition of a complex number, expressed by equation 1.1, is referred to as the Cartesian representation. Another way to describe it is by using polar coordinates:

- $\rho = \sqrt{(a^2 + b^2)}$ is referred to as the magnitude
- $\theta = \tan^{-1}(\frac{b}{a})$ is instead defined as its phase

By defining the probability amplitudes of a qubit state in this way, it is possible to rearrange its expression as:

$$|\Psi\rangle = \cos(\theta)|0\rangle + e^{i\phi}\sin(\theta)|1\rangle, \ where\ \theta \in [0;\pi]\ and\ \phi \in [0;2\pi] \tag{1.7}$$

Where the angles θ and ϕ can be used to describe the vector that starts from the origin and arrives at the surface of the Bloch sphere, referred to as the **Bloch Vector**, as shown in Figure 1.1.

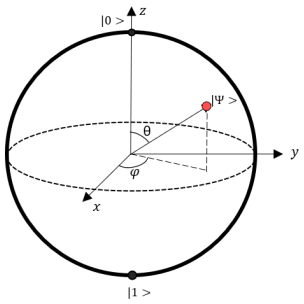


Figure 1.1. Bloch Sphere

1.5 Quantum Circuits and Quantum Gates

Similarly to what happens in classical digital circuits, quantum circuits do calculations by manipulating the information stored into qubits. They do so through the use of devices called quantum gates, which can be combined in different ways to implement many algorithms. Quantum dynamics is expressed through linear transformation between complex vector spaces, represented by complex square matrices. Given a matrix A, defined as

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \tag{1.8}$$

where a_{ij} is a complex number, it is possible to define its transpose, conjugate, adjoint:

- The **transpose** of a matrix is obtained by switching the rows and the columns of a matrix, and is defined as

$$A^{\mathrm{T}} = \begin{bmatrix} a_{11} & \dots & a_{m1} \\ a_{12} & \ddots & \vdots \\ \dots & \dots & a_{mn} \end{bmatrix} \tag{1.9}$$

- To obtain the **conjugate** matrix, the sign of the imaginary part of each complex entry is inverted, which is referred to as its complex conjugate $\overline{a_{ij}}$. The conjugate matrix is therefore defined as

$$\overline{A} = \begin{bmatrix} \overline{a_{11}} & \overline{a_{12}} & \dots \\ \vdots & \ddots & \vdots \\ \overline{a_{m1}} & \dots & \overline{a_{mn}} \end{bmatrix} \tag{1.10}$$

- The **adjoint** is a conjugate matrix that has been transposed as well.

$$A^{\dagger} = \begin{bmatrix} \overline{a_{11}} & \dots & \overline{a_{m1}} \\ \overline{a_{12}} & \ddots & \vdots \\ \dots & \dots & \overline{a_{mn}} \end{bmatrix} \tag{1.11}$$

Furthermore, a matrix is invertible if there exists a matrix A^{-1} such that

$$A^{-1}A = AA^{1} = I \tag{1.12}$$

where A^{-1} is defined as its inverse.

The transformations implemented by a quantum gate are represented by **unitary complex matrices**, which are matrices for which adjoint and inverse coincide

$$U^{\dagger}U = UU^{\dagger} = I. \tag{1.13}$$

After applying a gate on an a quantum system, to obtain the resulting state a tensor product between the state vector and the gate matrix must be performed. Given a state $|\Psi\rangle$ defined as

$$|\Psi\rangle = [c_0, c_1, \dots, c_n]^T \tag{1.14}$$

and a unitary matrix U given by

$$U = \begin{bmatrix} u_{11} & \dots & u_{n1} \\ u_{12} & \ddots & \vdots \\ \dots & \dots & u_{nn} \end{bmatrix} \tag{1.15}$$

The resulting state $|\Psi'\rangle$ is defined as

$$|\Psi'\rangle = U|\Psi\rangle = \begin{bmatrix} U_{11} \cdot c_1 + U_{12} \cdot c_2 + \dots + \dots + U_{1n} \cdot c_n \\ \dots + \dots + \dots + \dots + \dots \\ \vdots + \vdots + \vdots + \vdots + \vdots \\ U_{n1} \cdot c_1 + U_{n1} \cdot c_2 + \dots \dots U_{nn} \cdot c_n \end{bmatrix} \tag{1.16}$$

By exploiting the properties of unitary matrices, applying a gate whose unitary is equal to $U^\dagger$ on $|\Psi'\rangle$, the original state $|\Psi\rangle$ can be re-obtained, making the evolution of a quantum state reversible in time. Measurement is the only quantum operation that is not represented by a unitary matrix and is in fact a non-reversible operation, as stated previously.

1.5.1 One-Qubit gates

Some common one-qubit quantum gates and their matrices will be discussed in this section.

Pauli X, Y, Z Gates

The Pauli gates are ubiquitously used in quantum computing. Each of them performs a rotation of the qubit state along one of the three axes present in the Bloch sphere representation.

The **Pauli X-gate** performs a π-radiants rotation around the X-axis which causes the state of the qubit to be flipped. It is the equivalent of the classical NOT gate in the binary domain, which is why it is also referred to the *NOT* gate. The matrix that corresponds to the transformation applied by this gate and its symbols are given in equation 1.17 and in Figure 1.2 respectively.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1.17}$$

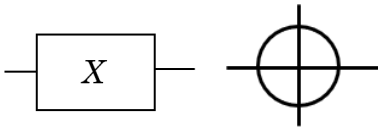


Figure 1.2. Symbols for Pauli X-Gate

For a qubit in the general state $|\Psi\rangle = \begin{bmatrix}\alpha & \beta\end{bmatrix}^T$ to whom an X-gate transformation is applied, will have a resulting state equal to:

$$|\Psi'\rangle = X|\Psi\rangle = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}\beta\\ \alpha\end{bmatrix} \tag{1.18}$$

In Figure 1.3, an example of a X-gate applied on an a state initialized at $|0\rangle$ is shown.

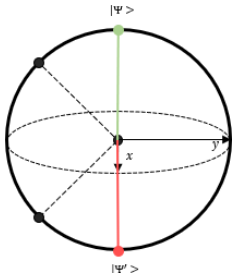


Figure 1.3. Bloch Sphere representation of the X gate transformation

The **Pauli Z-Gate** applies a rotation around the Z-axis of π-radiants, which means flipping the relative phase of a generic qubit state. It is often referred to as the *phase gate*. Its matrix and symbol are reported in Figure 1.4.

$$Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} \tag{1.19}$$

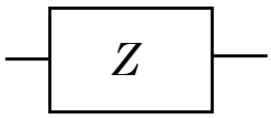


Figure 1.4. Symbol for Pauli Z-Gate

In general, for a qubit state defined as $|\Psi\rangle = \begin{bmatrix}\alpha & \beta\end{bmatrix}^T$ to whom a Z-gate transformation is applied will have a resulting state equal to:

$$|\Psi'\rangle = Z|\Psi\rangle = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}\alpha\\ -\beta\end{bmatrix} \tag{1.20}$$

The example shown in Figure 1.5 shows the effect of a Z-gate applied on a state given by $|\Psi\rangle = \frac{|0\rangle+1\rangle}{\sqrt{2}}$.

The **Pauli Y-Gate** rotates the state of a qubit of π-radiants around the Y-axis, which means the final state will have both a different relative phase and a different amplitude probability. Since its action on the qubit state can be achieved

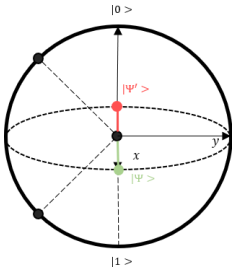


Figure 1.5. Bloch Sphere representation of the Z gate transformation

by combining a Pauli X gate and a Pauli Z gate. It is referred to as the *bit-phase-flip gate*. Its matrix and symbol are reported in Figure 1.6.

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \tag{1.21}$$

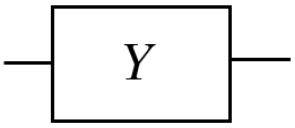


Figure 1.6. Symbol for Pauli Y-Gate

In general,

$$|\Psi'\rangle = Y|\Psi\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} -i \cdot \beta \\ i \cdot \alpha \end{bmatrix} \tag{1.22}$$

Figure 1.7 depicts the result of applying a Y-gate on a state initialized at $|0\rangle$.

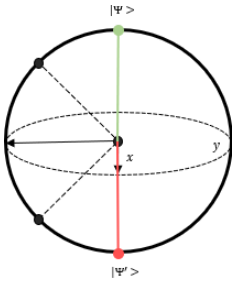


Figure 1.7. Bloch Sphere representation of the Y gate transformation

Hadamard Gate

The Hadamard Gate is one of the most important gates because it permits to get a superposition of basis states with uniform amplitudes, which is one of the main

ingredients of most quantum algorithms. The matrix that represents this operation transform is given in equation 1.23 and its symbol in Figure 1.8.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad (1.23)$$

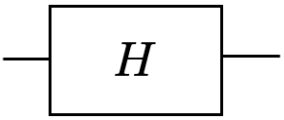


Figure 1.8. Symbol for Hadamard Gate

It can be considered a change of basis since it transforms the basis states $|0\rangle$ and $|1\rangle$ into two other remarkable states, denoted as $|+\rangle$ and $|-\rangle$:

$$|0\rangle \longrightarrow \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle, \qquad |1\rangle \longrightarrow \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle \quad (1.24)$$

By looking at the Bloch Sphere in Figure 1.9, it is possible to visualize the basis change better, as the frame of reference switches from the Z-axis to the X-axis.

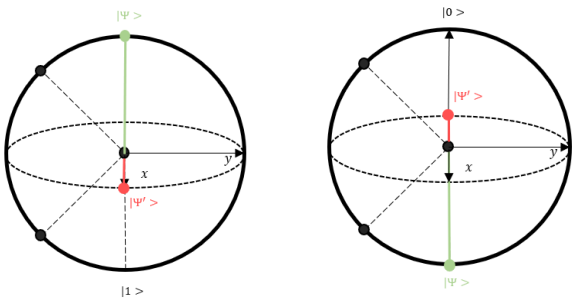


Figure 1.9. Bloch Sphere representation of the Hadamard gate transformation

Arbitrary Rotation Gates

Rotation gates allow the application a rotation around one of the axes by an arbitrary angle. By convention, there are three of them, R_x, R_y, and R_z. Here only R_y is presented because it is the only one used in the algorithms present in this document. Its matrix and symbol are the following:

$$R_y(\theta) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \tag{1.25}$$

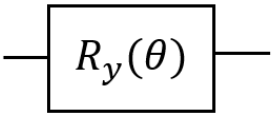


Figure 1.10. Symbol for R_y Gate

The example in Figure 1.11 shows a rotation of $\frac{3}{4}\pi$ radiants around the y-axis on the Bloch sphere.

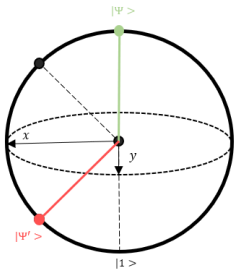


Figure 1.11. Bloch Sphere example for the R_y gate

1.5.2 Two-qubit gates

SWAP Gate

The Swap gate is a two-qubit gate that allows swapping the position of the qubits. The swap gate is of crucial importance in the compilation of quantum circuits, taking into consideration the limited connectivity of the current circuits' topologies. Its matrix is reported in equation 1.26 and its symbol is shown in Figure 1.12.

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.26}$$

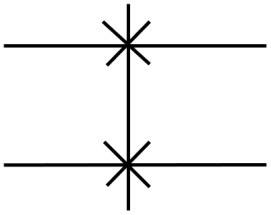


Figure 1.12. Symbol for SWAP gate

Controlled gates

A controlled gate is a gates that performs an IF-THEN operation. Its inputs are divided into controls and targets, and its gate transformation is performed on the target qubits only when the control qubits are true (or false, depending on the type of controls). Starting from a gate U described by an n-qubit unitary matrix, adding a control qubit will result into a gate described by an $(n+1)$-unitary matrix ${}^{\mathrm{C}}U$. Controlled gates are represented by the symbol in Figure 1.13.

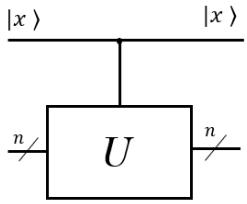


Figure 1.13. Symbol for a ${}^{\mathrm{C}}U$ gate

For a given unitary U describing a gate, its controlled version is given by ${}^{\mathrm{C}}U$

$$U = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \longrightarrow {}^{\mathrm{C}}U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix}. \tag{1.27}$$

CNOT gate

The CNOT gate is the controlled version of the Pauli X gate, which means that the flip of the target qubit is applied only when the control qubit is true. Its matrix is reported in equation 1.5.2 and its symbol is shown in Figure 1.14.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.28}$$

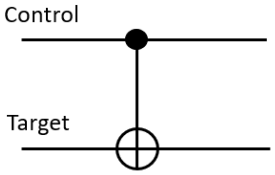


Figure 1.14. Symbol for CNOT gate

To change the condition on which the target qubit is flipped, a NOT gate can be applied before and after the control qubit. This approach has its own symbol, reported in Figure 1.15, which is often used to simplify circuits' schematics. The

equivalent gate is referred to as $0CNOT$ gate and, as described by the matrix in equation 1.5.2, it works in the same exact way as the CNOT, except that the transformation is applied if the control qubit is in state $|0\rangle$.

$$0CNOT = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.29)$$

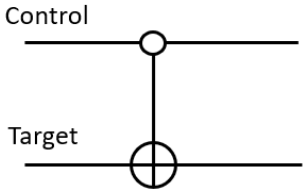


Figure 1.15. Symbol for 0CNOT gate

Controlled Rotation gate

The controlled rotation gate applies the rotation only if the control qubit is true. Only the controlled rotation along the Y-axis is described, because it is the only one that finds is use in the algorithms presented in this document.

$$CR_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ 0 & 0 & \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

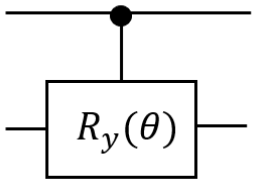


Figure 1.16. Symbol for $C - R_y$ gate

1.5.3 Three-qubit gates

Toffoli Gate

The Toffoli Gate is a three-qubit gate, also known as the CCX gate or CCNOT. It is a NOT gate with two control qubits and one target. In this case, the operation is performed only when both of the control qubits are true.

Friedkin Gate

The Friedkin Gate, or CSWAP, is also a three qubit gate, but in this case, it has one control and two targets. It is the controlled version of the SWAP gate, in fact

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

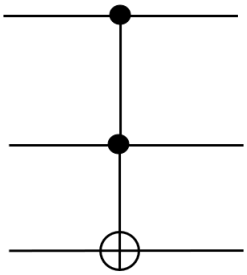


Figure 1.17. Symbol for Toffoli gate

the swap between the two qubits takes place only if the control qubit is true. Its unitary matrix is represented in equation 1.30 and its symbol in Figure 1.18

$$CSWAP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.30}$$

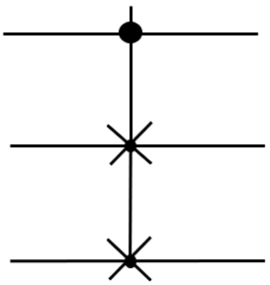


Figure 1.18. Symbol for Friedkin gate

1.5.4 Measurement Operation

As said before, the measurement is a non-reversible operation, therefore it is not described by a unitary matrix. This operation is only applied to the circuit at the end of the computation. It is represented by the symbol in Figure 1.19.

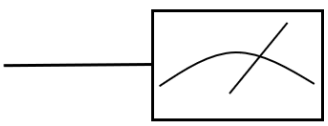


Figure 1.19. Measurement Operation Symbol

Chapter 2

Quantum Image Processing: state of the art

2.1 Introduction

Nowadays, the interest around applications like pattern recognition and computer vision makes image processing tasks like storage and processing of visual information of great importance. However, the rapidly increasing volume of visual information weighs on the computational capabilities currently available in classical computers. In the meantime, the Quantum Computing paradigm began to establish itself and grow in popularity, thanks to the recent advancement in the development of quantum hardware. Exploiting the phenomena presented in section 1 like superposition and entanglement increases the density of processable information, while introducing a computational speed-up in the attainment of the results. The inherent parallelism of the quantum framework can therefore be exploited to reduce the computational complexity of the image processing tasks, and from the intersection of these two research domains, Quantum Image Processing (**QImP**) was born. In this context, many algorithms that permit to encode an image in a quantum circuit and process it, have been presented. This chapter gives an overview of QImP techniques that were considered promising in terms of feasibility and application scenarios. The selection spans from encoding methods, basic processing tools, and edge detection techniques. Classical compression algorithm that allows the reduction of the quantum resources needed has also been investigated. First, the formalism and workflow of the algorithms are presented and then they are analyzed in terms of complexity and dimension of the corresponding circuit.

2.2 Quantum Image Representation

In order to process the information on quantum computers, the images need to be stored in a quantum circuit. Many different models have been proposed that differ in the way they encode the intensity information of pixels as well as positions, making them different in terms of image processing applications and algorithmic complexities. The two most popular Quantum Image Representations (QIRs) are here discussed, followed by a less-known model that has promising perspectives in terms of computer vision applications.

2.2.1 FRQI

Flexible Representation of Quantum Images (FRQI) is a phase encoding technique in the QIR panorama presented in [3]. It represents the color information of an image through the phase of a qubit. It uses a Cartesian coordinate system and captures information about the intensity and position of the pixels in the state of a quantum circuit. For a gray-scale image, the FRQI state is defined as

$$|I(\theta)\rangle = \frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} (cos\theta_i|0\rangle + sin\theta_i|0\rangle) \otimes |i\rangle, \tag{2.1}$$

$$\theta_i \in [0, \frac{\pi}{2}], i = 0,1,...,2^{2n} \tag{2.2}$$

From equation 2.1 is possible to see that the FRQI state is a normalized state, i.e. $||I(\theta)|| = 1$ as given by

$$||I(\theta)|| = \frac{1}{2^n} \sqrt{\sum_{i=0}^{2^{2n}-1} (cos^2\theta_i + sin^2\theta_i)^2} = 1 \tag{2.3}$$

and that the state is made of

- information that encodes the gray-scale information: $cos\theta_i|0\rangle + sin\theta_i|0\rangle$, with θ proportional to the intensity of the pixel
- information that encodes the corresponding position of the pixel: $|i\rangle$

Taking for example a 2×2 image like in Equation 2.2.1, its FRQI state will be described by

$$
\begin{aligned}
|I\rangle \quad &= \frac{1}{2}[(\cos\theta_0|0\rangle + \sin\theta_0|1\rangle) \otimes |00\rangle + (\cos\theta_1|0\rangle + \sin\theta_1|1\rangle) \otimes |01\rangle + \\
&+ (\cos\theta_2|0\rangle + \sin\theta_2|1\rangle) \otimes |10\rangle + (\cos\theta_3|0\rangle + \sin\theta_3|1\rangle) \otimes |11\rangle] = \\
&= \frac{1}{2}(\cos\theta_0|000\rangle + \sin\theta_0|001\rangle + \cos\theta_1|010\rangle + \sin\theta_1|011\rangle + \cos\theta_2|100\rangle + \\
&+ \sin\theta_2|101\rangle + \cos\theta_3|110\rangle + \sin\theta_3|111\rangle
\end{aligned} \tag{2.4}
$$

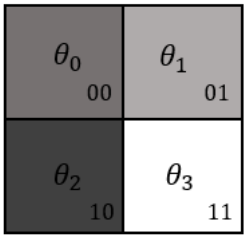


Figure 2.1. 2×2 FRQI image

In the FRQI encoding method, the number of qubits needed to encode the positions of a $2^n \times 2^n$ image is given by $N_{pos} = \log_2 2^n + \log_2 2^n = 2n$. An additional qubit is used to encode the color information, for a total of

$$N = N_{pos} + N_{color} = 2n + 1$$

To build an FRQI state representing an image, the $2n$ position qubits have to be put into a superposition of states with uniform probability amplitudes. In this way, the 2^{2n} combination of their basis states will represent all of the 2^{2n} positions present in the image with equal probability. This can be achieved by applying a Hadamard gate on each position qubit.

Starting from the initialized state $|0\rangle^{\otimes 2n+1}$, the subsequent state is obtained by applying the transform $\mathcal{H} = I \otimes H^{\otimes 2n}$ which is equivalent to applying $2n$ Hadamard gates on each qubit dedicated to the position. The new state is given by

$$|H\rangle = \frac{1}{2^n}|0\rangle \otimes \sum_{i=0}^{2^{2n}-1} |i\rangle \tag{2.5}$$

After that, the intensity of the pixels is mapped on the color qubit by changing its phase by an angle proportional to the intensity value. This transformation is achieved through the application of a R_y gates for each pixel. To make sure that the angle encoded corresponds only to a specific pixel in the image, the R_y gate has to be controlled by the position qubits and its controls have to be true or negated

depending on the position of the pixel that is being encoded, obtaining the final state:

$$|I(\theta_i)\rangle = \mathcal{R}|H\rangle = (\prod_{i=0}^{2^{2n}-1} R_i)|H\rangle \tag{2.6}$$

The algorithm can therefore be divided into two steps:

- *Step 1*: putting the position qubits into superposition, through the use of Hadamard gates.
- *Step 2*: applying phase change on the color qubit through $C^{2n} - R_y$ gates.

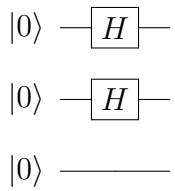
Example Circuit

Taking for example the 2×2 image schematized in Equation 2.2.1, in order to translate it into an FRQI image, the number of qubits needed to encode the position of the pixels are

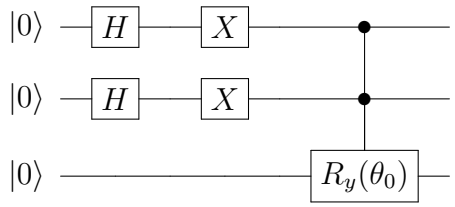
$$2^1 \times 2^1 \longrightarrow n = 1 \longrightarrow N_{pos} = 2n = 2$$

for a total of $N = 3$.

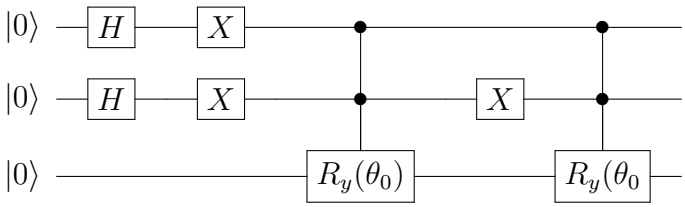
Now the position qubits have to be put into superposition by applying a Hadamard gate on each of them:



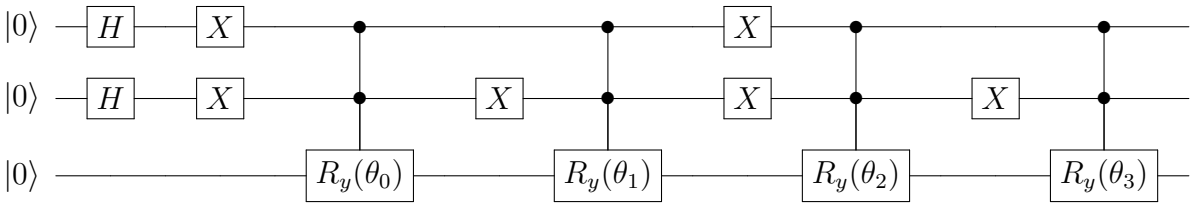
The first pixel is in position 00, therefore to activate the controls of the rotation gate when the position qubits are in state $|00\rangle$, a NOT gate is applied on each. To add the color information, a controlled rotation gate is applied on the color qubit with an angle proportional to the intensity of the pixel:



In order to keep the circuit schematic simple, the NOT gates used to change the conditions on which the $C^2 - R_y$ gate activates will be applied taking into consideration the gates that have been previously applied. Therefore, to encode the position of the second pixel 01, an X-gate is applied to the second position qubit. It is then followed by the controlled rotation:

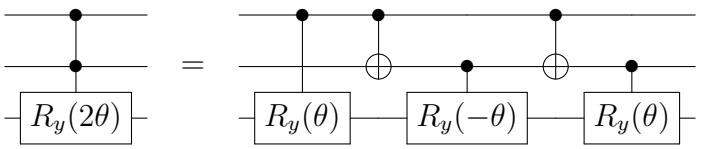


The procedure is repeated for the last two pixels as well, obtaining:



Computational Complexity

The computational complexity of an algorithm measures the number of elementary steps that an algorithm runs on inputs of arbitrary size. Complex quantum gates, like controlled gates, are constituted by a combination of simpler gates, therefore, to implement them on real devices and to understand their actual depth, they have to be decomposed. As shown in the previous section, the FRQI algorithm can be implemented by $2n$ Hadamard gates and 2^{2n} controlled rotations $C^{2n}(R_y(2\theta_i))$. The controlled rotations can be broken down into $2^{2n}-1$ simple rotations, $C(R_y(\frac{2\theta_i}{2^{2n}-1}))$ and $C(R_y(-\frac{2\theta_i}{2^{2n}-1}))$, and $2^{2n}-2$ $CNOT$ [3]. Taking the example of a 2×2 image, i.e. $n = 1$, a $C^2(R_y(2\theta_i))$ operation can be broken down as shown in the schematic below:



Therefore the total number of simple operations needed to prepare an FRQI image is given by:

$$2n + 2^{2n} \times (2^{2n} - 1 + 2^{2n} - 1 - 2) = 2^{4n} - 3 \cdot 2^{2n} + 2n \tag{2.7}$$

The calculation complexity of FRQI is thus $O(2^{4n})$, which is quadratic to the 2^{2n} pixels present in the image. The calculation does not take into consideration the number of X-gates needed to encode the position because it is negligible in the approximation of the complexity.

Image Retrieval

To retrieve the image from the quantum framework to the classical one and be able to visualize it, the qubits of the circuit must be measured. A measurement performed on an FRQI circuit gives out a state of the form $|c\rangle|i\rangle_{2n}$, where $|c\rangle$ is the color qubit and $|i\rangle_{2n}$ are the qubits that encode the position. A single measurement of this circuit, however, does not give any information about the intensity of the pixel, since that information is encoded in the phase between the amplitudes of the state. Therefore, to get the amplitudes, multiple measurements of many identical FRQI states are required. The results will produce a probability distribution that is then elaborated to obtain the angles encoding the intensity levels.
Taking the mathematical expression of an FRQI state that encodes only the intensity information of the $i-th$ pixel of a $2^n \times 2^n$ image

$$|c_i\rangle = \frac{1}{2^n}\cos(\theta_i)|0\rangle + \sin(\theta_i)|1\rangle \tag{2.8}$$

and performing a number N_{meas} of measurements on the circuit, it is possible to define N_{0i} and N_{1i} as the number of times that the measurement of the color qubit is found to be 0 and 1 respectively, in the same position. The amplitudes of the state are re-obtained as:

$$c_{0i} = \frac{1}{2^n}\cos(\theta_i) = \sqrt{\frac{N_{0i}}{N_{meas}}} \tag{2.9}$$

$$c_{1i} = \frac{1}{2^n}\sin(\theta_i) = \sqrt{\frac{N_{1i}}{N_{meas}}} \tag{2.10}$$

To extrapolate the information about the angle, it is sufficient to apply an inverse function, after doing some calculations:

$$\theta_i = \arctan\left(\sqrt{\frac{N_{0i}}{N_{1i}}}\right) \tag{2.11}$$

It is important to notice that results obtained from measuring an FRQI state will always be an approximation of the original image and that, to obtain an accurate

approximation, it is necessary to perform a sufficiently high number of measurements, depending on the size of the image.

Multi-Channel Representation

To extend FRQI to the representation of color images, a Multi-Channel Representation for Quantum Images (MCRQI) was introduced in [4]. It uses the widely popular RGB color model to represent the color information, adding two other channels to the circuit. The RGB model, that stands for red (R), green (G), and blue (B), is an additive color model that uses primary colors with different intensities to reproduce a wide array of colors. To accomplish this, three qubits are instantiated to encode the color information, instead of just one. The resulting expression is:

$$|I\rangle = \frac{1}{2^{n+1}} \sum_{i=0}^{2^{2n}-1} |C^i_{RGB} \otimes |i\rangle, \tag{2.12}$$

where $|C^i_{RGB}\rangle$ encodes the information of the R, G, and B channels

$$|C^i_{RGB}\rangle = |C_R\rangle|00\rangle + |C_G\rangle|01\rangle + |C_B\rangle|10\rangle \tag{2.13}$$

with

$$\begin{aligned} |C_R\rangle &= \cos\theta^i_R|0\rangle + \cos\theta^i_R|1\rangle, \\ |C_G\rangle &= \cos\theta^i_G|0\rangle + \cos\theta^i_G|1\rangle, \\ |C_B\rangle &= \cos\theta^i_B|0\rangle + \cos\theta^i_B|1\rangle \end{aligned}$$

Taking as an example a 2×2 image, as in Figure 2.2, the overall state is described as:

$$\begin{aligned} |I\rangle = \frac{1}{4}[&(\cos\theta^R_0|000\rangle + \cos\theta^G_0|001\rangle + \cos\theta^B_0|010\rangle + \sin\theta^R_0|100\rangle + \sin\theta^G_0|101\rangle + \\ &+ \sin\theta^B_0|110\rangle) \otimes |00\rangle + (\cos\theta^R_1|000\rangle + \cos\theta^G_1|001\rangle + \cos\theta^B_1|010\rangle + \sin\theta^R_1|100\rangle + \\ &+ \sin\theta^G_1|101\rangle + \sin\theta^B_1|110\rangle) \otimes |01\rangle + (\cos\theta^R_2|000\rangle + \cos\theta^G_2|001\rangle + \cos\theta^B_2|010\rangle + \\ &+ \sin\theta^R_2|100\rangle + \sin\theta^G_2|101\rangle + \sin\theta^B_2|110\rangle) \otimes |10\rangle + (\cos\theta^R_3|000\rangle + \cos\theta^G_3|001\rangle + \\ &+ \cos\theta^B_3|010\rangle + \sin\theta^R_3|100\rangle + \sin\theta^G_3|101\rangle + \sin\theta^B_3|110\rangle) \otimes |01\rangle] \end{aligned}$$

The procedure to build an MCRQI state is fairly similar to an FRQI one. The position qubits are put into superposition and the intensity of the pixels is encoded into the phase of one color qubit, referred to as the R qubit, by applying a rotation gate. The main differences instead are are:

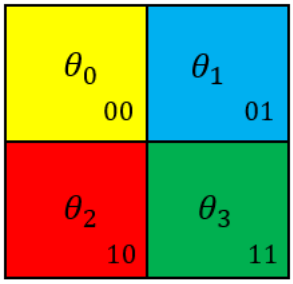


Figure 2.2. 2×2 MCRQI image

- two more qubits are added to the circuit, referred to as the G and B qubits.
- The rotation gates applying the phase change are also controlled by the G and B qubits, as well as by the position qubits.
- The measurement is performed conditioned by the state of the G and B channel qubits.

The two added color qubits are used to generate more combinations of the basis states to embed the intensity information of all three channels. For each of the positions, adding two more qubits into superposition, can expand the number of basis state combinations to four instead of one. The MCRQI encoding exploits only three, one for each channel. The phase of the R qubit will cyclically represent the intensity of each of the three channels. By applying controls on the measurement operation, it is possible to distinguish between the three channels.

The total number of qubits required for an MCRQI image is therefore

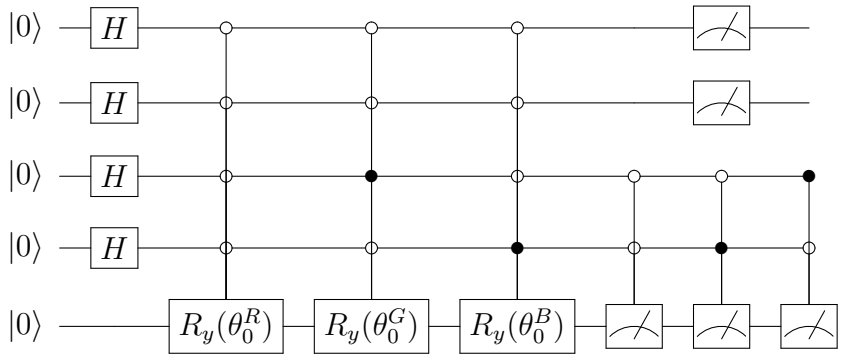
$$N = N_{pos} + N_{color} = 2n + 3$$

The quantum circuit for a 2×2 MCRQI image is reported below. For simplicity, only the first pixel is encoded, seeing as the procedure for the other ones is constant. Instead of explicitly showing the application of X gates that changes the state of the control qubits, the negated controls are represented by a white dot, as seen in section 1 for the $0CNOT$ gate.

As can be seen from the circuit, for each pixel three controlled rotations are needed and, in the end, three measurement operations that allow to distinguish between the three color channels when retrieving the image.

The simple operations needed to implement the algorithm are:

- $2 + 2n$ Hadamard Gates
- 3×2^{2n} $C^{2n+2}(R_y(2\theta))$ that are broken down into



- $3 \times 2^{2n} \times (2^{2n+1} - 1)$ $C(R_y(2\theta))$, $C(R_y(-2\theta))$
- $3 \times 2^{2n} \times (2^{2n+1} - 2)$ CNOT gates

giving a total of $24 \times 2^{4n} - 9 \times 2^{2n} + 2n + 2$ elementary gates. The computational complexity is $O(2^{4n})$, the same as for FRQI.

2.2.2 NEQR

FRQI presents some drawbacks that limit its capabilities of handling an image, since it stores the gray-scale information as the probability amplitude associated with the basis states of a single qubit. This limits the processing possibilities and makes it impossible to retrieve the accurate image from measuring the circuit. For these reasons, a novel enhanced quantum representation (**NEQR**) was introduced in [5]. This model uses the basis states of a qubit sequence to store the intensity information of every pixel, instead of the phase of a qubit state, and the Cartesian coordinates system to encode the position. For a $2^n \times 2^n$ image with a gray-scale range of 2^q values, a NEQR state is defined as:

$$|I\rangle = \frac{1}{2^n} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} |f(Y,X)\rangle |YX\rangle = \frac{1}{2^n} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} \bigotimes_{i=0}^{q-1} |C_{YX}^i| YX\rangle \tag{2.14}$$

where $f(Y,X)$ is the gray-scale value of the corresponding pixel (Y,X), defined as:

$$\begin{aligned} &f(Y,X) = C_{YX}^0 C_{YX}^1 \dots C_{YX}^{q-2} C_{YX}^{q-1} C_{YX}, \\ &C_{YX}^k \in [0,1], \\ &f(X,Y) \in [0,2^q - 1]. \end{aligned} \tag{2.15}$$

To encode a NEQR state, q qubits must be instantiated for the color encoding and $2n$ qubits for the position encoding:

$$N = N_{pos} + N_{color} = 2n + q$$

Taking, for example, a 2×2 image like in Figure 2.3, where the intensity gray-scale values are indicated in decimal form, the corresponding NEQR state will be:

$$|I\rangle = \frac{1}{2}(10001000\rangle \otimes |00\rangle + |00010000\rangle \otimes |01\rangle + |11000000\rangle \otimes |10\rangle + |1001100\rangle \otimes |11\rangle) \tag{2.16}$$

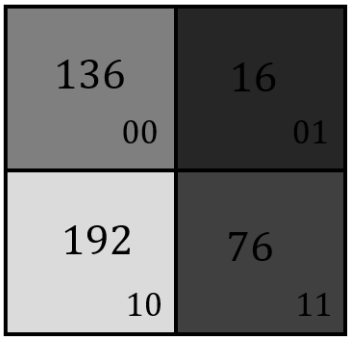


Figure 2.3. 2×2 NEQR image

Starting from a state initialized at $|\mathbf{0}\rangle$, there are two steps to obtain a NEQR image:

- *Step 1:* Similarly to FRQI, the position qubits are put into superposition through the use of Hadamard gates, while the sequence of color qubits is kept in the same state.
- Step 2:To set the gray-scale value of each pixel, $C^{2n}NOT$ gates are applied on the color qubits corresponding to the bits of the intensity values encoding a 1.

Step 1 of the algorithm can be represented by the transform U_1 defined as:

$$U_1 = I^{\otimes q} \otimes H^{\otimes 2n} \tag{2.17}$$

The first transform brings the initialized state $|\Psi\rangle_0$ to the intermediate state $|\Psi\rangle_1$:

$$|\Psi\rangle_0 = |0\rangle^{\otimes 2n+q} \tag{2.18}$$

$$|\Psi\rangle_1 = U_1(|\Psi\rangle_0) = (I|0\rangle)^{\otimes q} \otimes (H|0\rangle)^{\otimes 2n} = \frac{1}{2^n}|0\rangle^{\otimes q} \otimes \sum_{i=0}^{2^{2n}-1} |i\rangle = \frac{1}{2^n} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} ||0\rangle^q |YX\rangle \tag{2.19}$$

Step 2 is subdivided into 2^{2n} sub-operation, one for each pixel, that take care of setting the intensity value on the color qubits. The sub-operation, U_{YX} is instead defined as:

$$U_{YX} = (I \otimes \sum_{j=0}^{2^n-1} \sum_{i,j \neq YX}^{2^n-1} |ji\rangle\langle ji|) + \Omega_{YX} \otimes |YX\rangle\langle YX| \tag{2.20}$$

where Ω_{YX} is the value-setting operation for pixel (Y, X):

$$\Omega_{YX} = \bigotimes_{i=0}^{q-1} \Omega_{YX}^{i} \tag{2.21}$$

which consists of q quantum oracles (one for every qubit of the gray-scale range) as given by

$$\Omega_{YX}^{i} : |0\rangle \longrightarrow |0 \oplus C_{YX}^{i}\rangle. \tag{2.22}$$

If $C_{YX}^{i} = 1$, Ω_{YX}^{i} is a $C^{2n}NOT$ gate. Otherwise it is a simple Identity gate. For a detailed mathematical demonstration, the reader is invited to read [5].

Example Circuit

Taking the 2×2 image with a gray-scale range $q = 8$ in Figure 2.3 as an example, to build a NEQR circuit encoding that image the number of qubits needed is given by:

$$N = N_{pos} + N_{color} = 2n + q = 2 + 8 = 10$$

They start initialized at the state $|0\rangle^{\otimes 10}$.

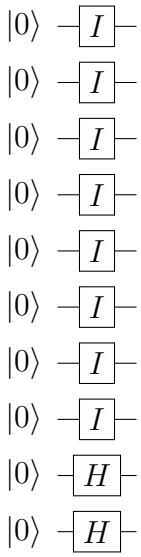
Applying the first step of the NEQR algorithm, the 2 position qubits are put into superposition through the use of Hadamard gates, while a NOP is applied on the color qubits:

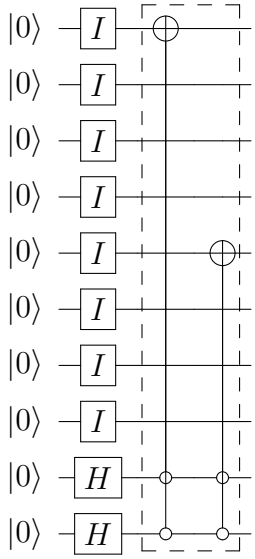
The first sub-operation of the second step, Ω_{00}, consists in encoding the intensity value of the pixel in position 00, whose binary value is "10010110", by applying a C^2NOT gate on the qubit corresponding to the bit encoding a 1, controlled by the position qubits.

The procedure is repeated for the other three pixels as well:

Computational Complexity

The first step of the algorithm is implemented through the use of $2n$ Hadamard gates and q Identity gates, achieving a complexity of $O(q+2n)$. The second step is





composed by 2^{2n} sub-operations U_{YX}, who in turn are made of q $C^{2n}NOT$ gates, in the worst case.
C^kNOT operations can be decomposed into $(4k-8)$ Toffoli gates when $k-2$ auxiliary qubits are present, as shown in the circuit below, making their complexity $O(k)$. The complexity of the whole U_{YX} operation is therefore $O(2qn)$ and that of the integral step is $O(2qn2^{2n})$. The complexity of the whole circuit is due to $Step2$ of the algorithm.

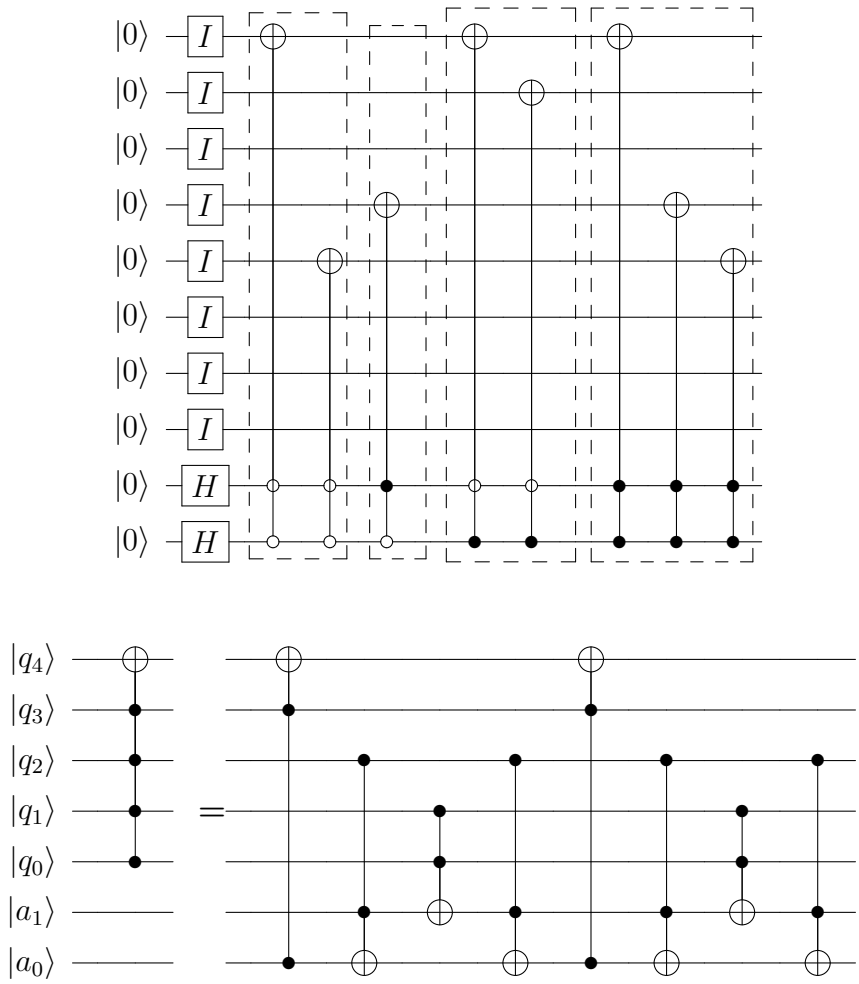


Image Retrieval

Unlike FRQI, NEQR does not store the intensity information in the probability amplitude of a qubit, but rather in the basis state of a sequence of qubits. Performing a measurement on the NEQR will give as a result one of the states:

$$I_{meas} = |f(Y, X)\rangle |YX\rangle$$

that represents, in binary form, the intensity and the position of the pixel.

Therefore, through this method, the actual intensity value can be recovered by decoding the result of the measurement.

Multi-Channel Representation

To handle RGB images also with the NEQR model, a novel quantum representation of color images (NCQI) was proposed in [6]. It extends the NEQR model by adding two more qubit sequences to embed the intensity information, for a total of three color channels. The number of qubits instantiated for $2^n \times 2^n$ image and with an intensity range of 2^q values, represented with the NCQI model, is equal to:

$$N_{tot} = N_{pos} + N_{color} = 2n + 3q \tag{2.23}$$

The procedure to build an NCQI state is exactly the same as for NEQR, with the exception that *Step 2* of the algorithm is performed for each color channel. Therefore $2n$ Hadamard gates are applied on each position qubit and, in the worst case scenario, $3q$ $C^{2n}NOT$ are applied for each 2^{2n} pixel, which corresponds the encoding of the maximum intensity value of the range.

A NCQI state for a $2^n \times 2^n$ image can be shown in the following equation:

$$|I\rangle = \frac{1}{2^n} \sum_{y=0}^{2^n-1} \sum_{x=0}^{2^n-1} |c(y,x)\rangle \otimes |yx\rangle \tag{2.24}$$

where $|c(y,x)\rangle$ denotes the intensity encoded in a binary sequence of all three channels of the corresponding pixel:

$$|c(y,x)\rangle = |R_{q-1} \dots R_0 G_{q-1} \dots G_0 B_{q-1} \dots B_0\rangle \tag{2.25}$$

Taking a 2×2 image as in Figure 2.4, with an intensity range for each channel of 256 values, the number of qubits needed to represent it is given by:

$$N_{tot} = 2 + 3 \cdot 8 = 26$$

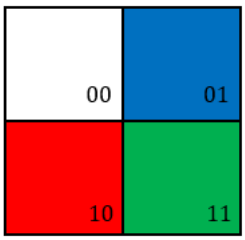


Figure 2.4. 2×2 NCQI image

The NCQI state that corresponds to the image is:

$$
\begin{aligned}
|I\rangle = \quad & \frac{1}{2}(|111111111111111111111111\rangle \otimes |00\rangle + \\
& + |000000000000000011111111\rangle \otimes |01\rangle + \\
& + |111111100000000000000000\rangle \otimes |10\rangle + \\
& + |000000001111111100000000\rangle \otimes |11\rangle)
\end{aligned}
\tag{2.26}
$$

To obtain the NCQI state, the required gates are:

- $2n$ Hadamard gates
- $2n \times 3q \times 2^{2n}$ Toffoli gates, in the worst case

therefore its complexity is $O(6qn \cdot 2^{2n})$, very similar to the NEQR one.

2.2.3 QPIE

FRQI and NEQR are by far the most popular methods for encoding images in the quantum domain. Since their proposal, several extensions, operations, and possible applications based on them have been presented. Their structures have advantages and disadvantages, therefore it is useful to explore other encoding methods, depending on the applications for a certain type of processing. A *Quantum Probability Image Encoding* (QPIE) has been proposed in [7] that exploits the probability amplitudes of the quantum states to store the gray-scale values of the pixels of an image and uses the Cartesian coordinate system like the previously presented encoding.

In particular, for a $2^{N_1} \times 2^{N_2}$ image represented as a $2D$ matrix of pixel's intensities as:

$$
I = (I_{yx})_{N_1 \times N_2} \tag{2.27}
$$

its QPIE state is given by:

$$
|I\rangle = \sum_{i=0}^{2^n - 1} c_i |i\rangle \tag{2.28}
$$

where c_i is the normalized pixel intensity so that the squared sum of all the probabilities amplitudes is equal to 1.

$$
c_i = \frac{I_{yx}}{\sqrt{\sum I_{yx}^2}} \tag{2.29}
$$

The number of qubits required for this encoding is $2n$, since the intensities values are encoded directly on the position qubits, making it the method that requires the least amount of qubits for the representation of a gray-scale image.

Taking Figure 2.5 as an example, the corresponding QPIE state will be given by:

$$|I\rangle = c_0|00\rangle + c_1|01\rangle + c_2|10\rangle + c_3|11\rangle \tag{2.30}$$

where $c_i = \frac{I_i}{\sqrt{\sum_{j=0}^{3} I_j}}$.

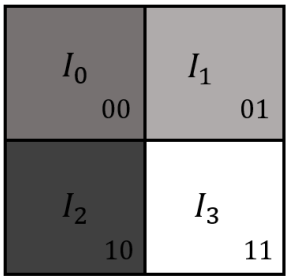


Figure 2.5. 2×2 QPIE image

2.2.4 Computational Complexity

The preparation of an n-qubit quantum state into a specific probability distribution $(c_0, c_1, ..., c_n)$ on a superposition of the orthonormal states of a set of qubits can be very efficiently implemented using CNOT gates and single-qubit rotations. Various algorithms have been proposed to implement the initialization of a quantum system whose complexity to prepare an arbitrary state starting from a known distribution is $O(2^n)$ [8], [9].

2.2.5 Image Retrieval

The extraction of a QPIE image has the same disadvantages as an FRQI image since the gray-scale information is encoded in the probability amplitudes. In order to retrieve a good approximation of the image, many measurements must be performed and the exact image can never be obtained. The probability of a state encoding a pixel's position, which is its intensity, is described as

$$c_i = \frac{I_{yx}}{\sqrt{\sum I_{yx}^2}} \tag{2.31}$$

Performing a measurement on a QPIE circuit will give as a result a generic state $|yx\rangle$. Defining N_{yx} as the number of times this result occurs and as N_{meas} the number of total measurements, the intensity of the pixel in position (y, x) can be approximated as:

$$I_{yx} = \sqrt{\frac{N_{yx}}{N_{meas}}} \times RMS \tag{2.32}$$

Furthermore, the squared sum of the intensities, used to normalize the QPIE state, is not embedded in the image. For this reason, only the proportions between the intensities can be retrieved, not their actual value. The retrieved value and the original one will differ by a proportional coefficient. Assuming that all the intensities have an intensity value equal to the middle of the range is a possible operative choice that introduces further approximation.

To compare the different types of encoding, table 2.1, gives an overview of the most important parameters of the three encoding methods.

Comparison of QImRs for a $2^n \times 2^n$ image and grey-scale range of 2^q values				
Representation method	Encoding type	Qubit resource	Complexity	Image Retrieval
FRQI	Phase	$2n + 1$	$O(2^{4n})$	Approximated
NEQR	Basis	$2n + q$	$O(qn2^{2n})$	Exact
QPIE	Amplitude	$2n$	$O(2^n)$	Approximated

Table 2.1. Comparison table between FRQI, NEQR, and QPIE

2.3 Geometric Transformations

After defining the models through which an image can be represented on a quantum circuit, processing can be applied to the embedded information. Geometric Transformations are a basic type of image processing that can be used as a building block in more advanced algorithms. Fast Geometric Transformations on Quantum Images are proposed in [10] that manipulates the position information of the pixels of the image through the use of elementary gates. This type of processing applies to every representation that uses Cartesian coordinate models, therefore it is applicable to all of the three previously presented encoding methods.

2.3.1 Flip operation

The flip operations consist in inverting the pixels' position with respect to X- or the Y-axis. They are defined as:

$$F_I^X(|I\rangle) = \frac{1}{2^n}\sum_{k=0}^{2^{2n-1}} |c_k\rangle \otimes F^X(|k\rangle \tag{2.33}$$

$$F_I^Y(|I\rangle) = \frac{1}{2^n}\sum_{k=0}^{2^{2n-1}} |c_k\rangle \otimes F^Y(|k\rangle \tag{2.34}$$

where c_k represents the color information, $|k\rangle = |y\rangle|x\rangle$ and

$$F^X(|y\rangle|x\rangle) = |\overline{y}\rangle|x\rangle, \tag{2.35}$$

$$F^Y(|y\rangle|x\rangle) = |y\rangle|\overline{x}\rangle, \tag{2.36}$$

$$|x\rangle = |x_{n-1}x_{n-2}\dots x_0\rangle, |y\rangle = |y_{n-1}y_{n-2}\dots y_0\rangle, \tag{2.37}$$

$$|\overline{x}\rangle = |\overline{x}_{n-1}\overline{x}_{n-2}\dots \overline{x}_0\rangle, |\overline{y}\rangle = |\overline{y}_{n-1}\overline{y}_{n-2}\dots \overline{y}_0\rangle, \tag{2.38}$$

where

$$\overline{x}_i = 1 - x_i, \overline{y}_i = 1 - y_i, i = 0,1,\dots,n-1 \tag{2.39}$$

To achieve an F_X or F_Y operation, NOT gates must be applied on the qubits that encode the positions related to the opposite coordinate axis, respectively the Y- or X-axis. An example of an F_Y flip operation on a 4×4 FRQI image and its expected result can be seen in the depictions below. The red dotted line represents the original image.

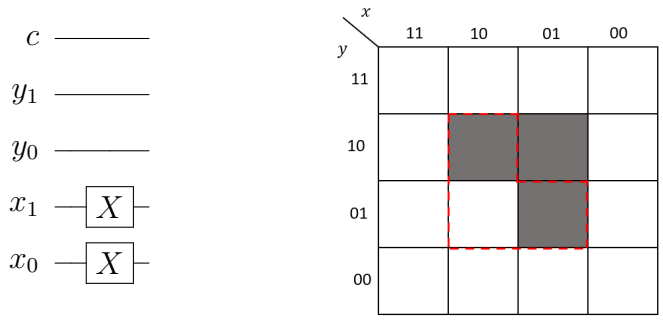


Figure 2.6. F_Y operation circuit on a 4×4 FRQI image and its result

Of the $2n$ qubits instantiated to encode the position, n are dedicated to specifying the position on the X-axis of an image, and n on the Y-axis. The number of NOT gates applied to the circuit to perform a flip along one of the two axes is equal to n, therefore the complexity of this algorithm is $O(n)$.

2.3.2 Coordinate Swap Operation

The coordinate-swapping C_I operation inverts the position of the pixels between the two coordinate axes. When applied to $|I\rangle$ produces outputs of the form:

$$C_I(|I\rangle) = \frac{1}{2^n} \sum_{k=0}^{2^{2n}-1} |c_k\rangle \otimes C(|k\rangle) \tag{2.40}$$

where c_k represents the color information, $|k\rangle = |yx\rangle$ and

$$C(|yx\rangle) = |xy\rangle \tag{2.41}$$

This operation is carried out by applying n SWAP gates that take (y_i, x_i) as inputs, where $i = 0, \ldots, n-1$. SWAP gates can be built using three CNOT gates. Therefore the total amount of elementary gates necessary is $3n$ CNOT gates, making the complexity of this algorithm $O(n)$.

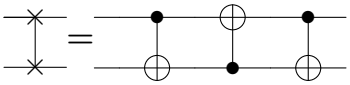


Figure 2.7 shows the coordinate swap circuit implemented on an FRQI 4×4 image and its expected result.

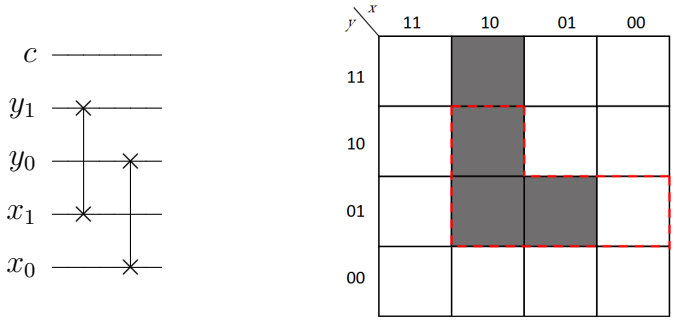


Figure 2.7. C_I operation circuit on an 4×4 FRQI image and its result

2.3.3 Orthogonal Rotation Operations

By combining the previous two operations, Flip and Coordinate-swap, it is possible to apply Orthogonal Rotations on quantum images, i.e. rotations with angles of 90°, 180°, and 270°. These orthogonal rotations operations are the operations R_I^{90}, R_I^{180}, and R_I^{270}, which are defined as:

$$R^{90}(|yx\rangle) = |x\overline{y}\rangle, \tag{2.42}$$
$$R^{180}(|yx\rangle) = |\overline{yx}\rangle, \tag{2.43}$$
$$R^{270}(|yx\rangle) = |\overline{x}y\rangle. \tag{2.44}$$

When applied on a quantum image, they produce outputs of the form:

$$R_I^{\alpha}(|I\rangle) = \frac{1}{2^n}\sum_{k=0}^{2^{2n}-1} |c_k\rangle \otimes R^{\alpha}(|k\rangle) \tag{2.45}$$

where c_k represents the color information, $|k\rangle = |yx\rangle$, $\alpha \in \{90,180,270\}$. They can be built from flip and coordinate-swapping operations as

$$R^{90} = CF^X \tag{2.46}$$
$$R^{180} = F^Y F^X \tag{2.47}$$
$$R^{270} = CF^Y \tag{2.48}$$

Therefore their computational complexity is the same as for flip and coordinate swapping operations, $O(n)$.

In figures 2.8, 2.9, and 2.10 circuit and expected results of R_I^{90}, R_I^{180}, and R_I^{270} respectively will be shown.

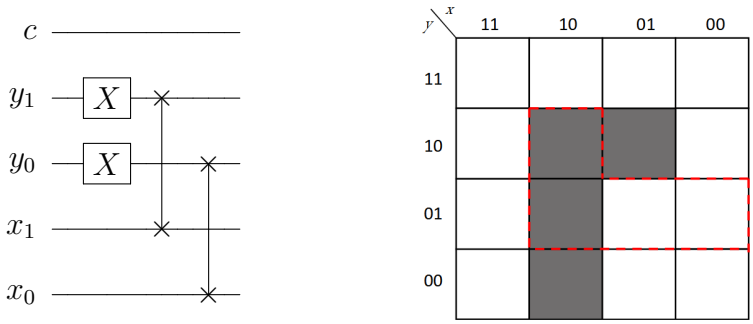


Figure 2.8. R^{90} operation circuit on an 4×4 FRQI image and its result

2.3.4 Restricted Geometric Transformations

After defining geometric transformations, it can be useful to limit their action to smaller subareas of the image. This type of operations is used in macro algorithms like watermarking, which is the process of hiding information in a carrier image. By changing the geometry of an image in a small area, the overall image looks

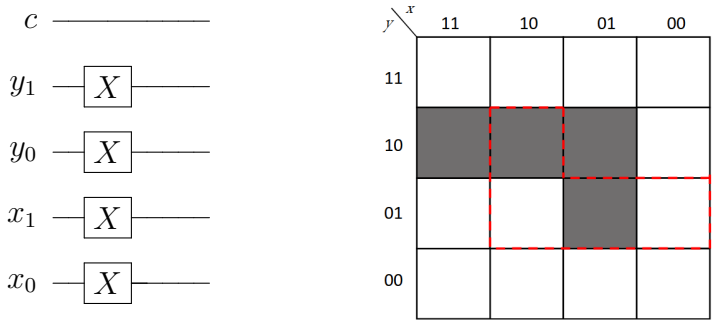


Figure 2.9. R^{180} operation circuit on an 4×4 FRQI image and its result

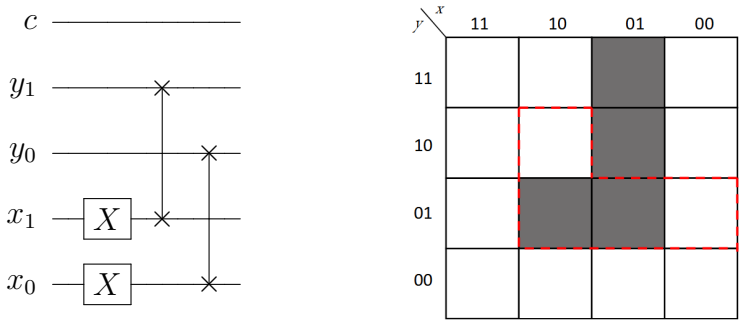


Figure 2.10. R^{270} operation circuit on an 4×4 FRQI image and its result

apparently the same but contains additional information when compared to the original. Restricted Geometric Transformations on Quantum Images (rGTQI) are presented in [11]. These operations are implemented by imposing control conditions on the gates that apply the geometric transformations. For example, imposing a condition on the MSB qubit encoding the Y coordinates, will result in dividing the image into two sub-areas, a lower and an upper one. Positions in the form $|1y_{n-2}\dots y_0\rangle|x\rangle$ will locate pixels in the upper half of the image. Adding a second condition on the MSB of the X coordinates will instead divide the image vertically into a left half and a right one. Positions in the form $|1y_{n-2}\dots y_0\rangle|0x_{n-2}\dots x_0\rangle$ will locate the right upper subarea of the image as can be seen in Figure 2.11, and so on. Increasing the number of conditions is equivalent to reducing the affected area.

To impose a flip with respect to the Y-axis on the lower half of the image for example, the NOT gates applied to the qubits of the X coordinate will have to be replaced by $0CNOT$ gates controlled by the MSB qubit of the Y coordinates. The circuit and the expected result are shown in Figure 2.12.

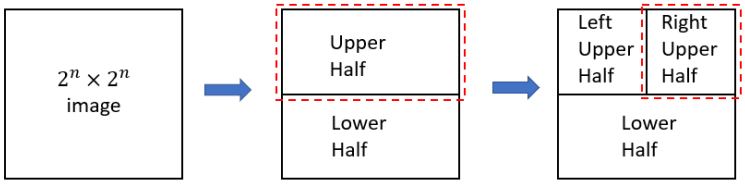


Figure 2.11. Dividing the image into subareas

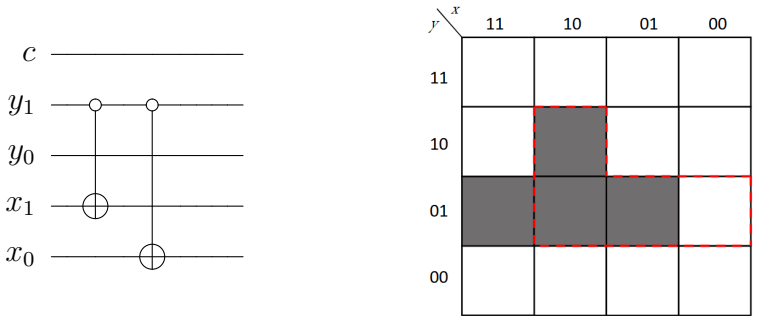


Figure 2.12. F_Y operation circuit applied on the lower half of an 4×4 FRQI image and its result

To apply the transformation only on the left lower part of the image, another control must be added to the gates, which will now be $Toffoli$ gates controlled by the MSB of both the Y and X coordinates. The gate that was applied on the MSB of X coordinate will not be needed anymore, because of the condition imposed on that qubit. Figure 2.13 shows the corresponding circuit and its result.

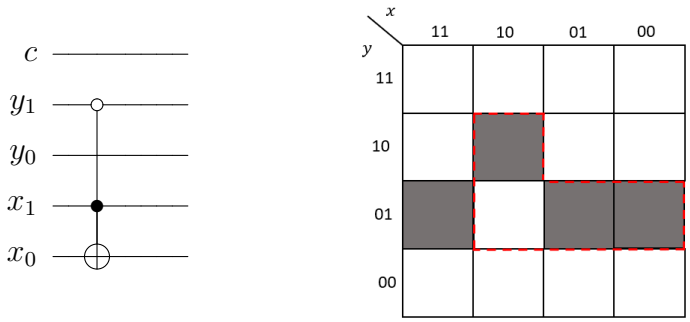


Figure 2.13. F_Y operation circuit applied on the left lower half of an 4×4 FRQI image and its result

The complexity of rGTQI operations depends on the number of conditions imposed on the qubits. Transformations that affect a smaller area will require a higher number of elementary gates. If the original transformations include a NOT gates, b $CNOT$ gates, and c $Toffoli$ gates, by adding a condition the gates will become respectively $CNOT$, $Toffoli$, and C^3NOT gates. As mentioned in the previous section C^kNOT gates can be decomposed into $4k-8$ $Toffoli$ gates, therefore the gate count will be a $CNOT$ gates, $b+4c$ $Toffoli$, obtaining a complexity of $O(a+b+4c)$ [11].

2.3.5 Position Shifting Operations

Another type of geometric transformation that acts on the location of the pixels is the Position Shifting Transformation presented in [12]. This type of transformation consists of globally displacing the pixels of a certain amount of positions. It is often used as a building block inside image filtering algorithms, where shifted copies of the original image are used to confront neighboring pixels, as will be described later in this document. The position shifting operation is defined as

$$P(|i\rangle) = |i'\rangle = |(i+c) mod 2^{2n}\rangle, i \in 0,1,\ldots,2^{2n}-1 \; and \; c \in 0,1,\ldots,2^{2n}-1 \quad (2.49)$$

where i is the decimal representation of the position information $|yx\rangle$, while c controls of how many positions the image is going to be shifted. To shift the image of one position, C^k-NOT gates are sequentially applied on all of the position qubits, starting from the MSB, controlled by the k remaining qubits. An example of a circuit and its result on an 4×4 FRQI image when $c=1$ are shown in Figure 2.14.

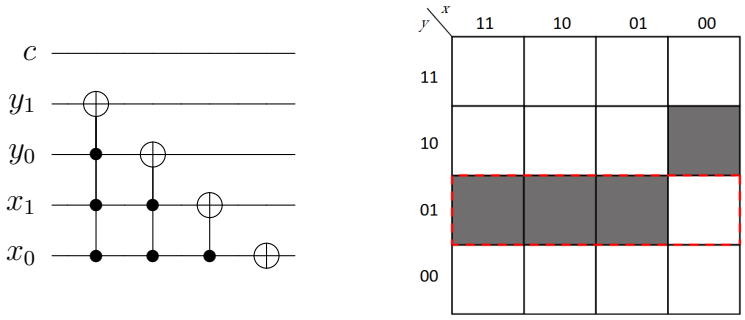


Figure 2.14. P operation circuit applied on an 4×4 FRQI image with $c=1$ and its result.

When $c=2^k$, the same type of circuit is applied on the $2n-k$ highest qubits. In Figure 2.15, the circuit of an 4×4 FRQI image when $c=2$ and its result are shown. In [12], also cases where c is not a power of 2 are detailed.

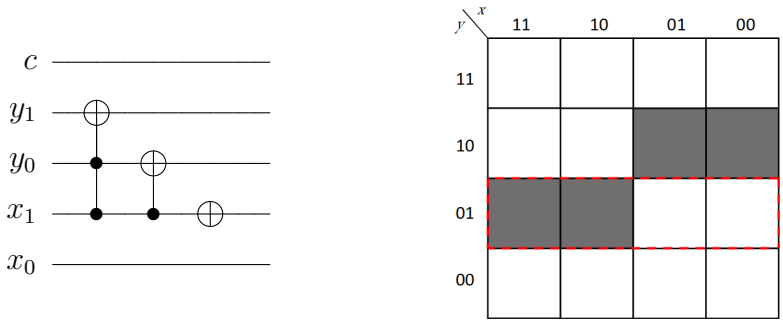


Figure 2.15. P operation circuit applied on an 4×4 FRQI image with $c = 2$ and its result.

To shift the image up or down the Y-axis, or left or right along the X-axis, the same type of circuit must be applied, but taking into consideration only the position qubits of the coordinate of interest. The direction of the shift depends on whether $C^k - NOT$ uses negated controls or not. The circuit to be applied on 4×4 FRQI image shifting the pixels to the right of one position and its result are shown in Figure 2.16.

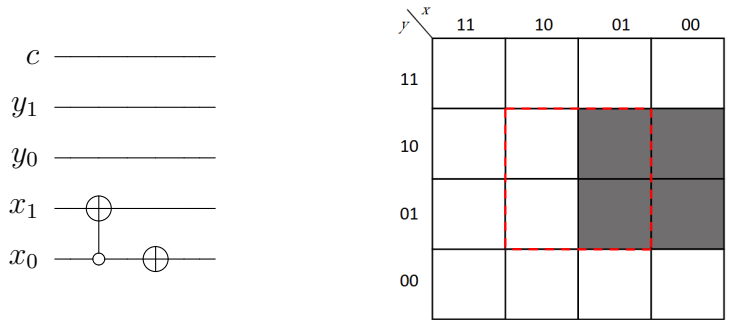


Figure 2.16. P operation circuit applied on an 4×4 FRQI image with $c = 1$ only on the X-coordinate its result.

The worst-case scenario, from a complexity point of view, is when $c = 1$, because it requires the highest number of gates. In that case, the complexity of the circuit, for an n-sized image is $O(n^2)$, as demonstrated in the reference [12].

2.4 Chromatic Transformations

Chromatic Transformations are another macro-category of basic processing that is used to modify the color information stored in the image. Color transformations for FRQI and NEQR are presented in the following sections.

2.4.1 Applications on FRQI

FRQI uses only the phase of one qubit to store the color information. Applying a gate on the color wire, the gray-scale information of the whole image is affected. Different basic gates can be used to obtain various results on the image as presented in [13], while adding controls on the position qubits allows restricting the affected areas, as seen for Restricted Geometric Transformations.

Color transformation based on X-gate

When the X-gate is applied to the color qubit of an FRQI image, the result is defined as:

$$X(|c(\theta_k)\rangle = |c(\frac{\pi}{2} - \theta_k)\rangle, \forall k \in 0, 1, \ldots, 2^{2n} - 1, \tag{2.50}$$

where $|c(\theta_k)\rangle$ represents the color information. This operation acts like a color inversion. The intensities of the pixels of the entire image invert from darker to lighter and vice versa. This complement color transformation makes the target in the image (notably medical images) easier to be found[14].

Color Transformation based on Ry-gate

The R_y gate is used in the FRQI algorithm to encode the color information. It can also be used to modify the color after that the image has already been encoded, specifically increasing or decreasing the gray-scale information. Applying an R_y on the color qubit has the effect of modifying the intensity of all of the pixels. The result is defined as

$$R_y(\theta)|c(\theta_k)\rangle = |c(\theta_k + \theta)\rangle \tag{2.51}$$
$$R_y(-\theta)|c(\theta_k)\rangle = |c(\theta_k - \theta)\rangle \tag{2.52}$$

It can be used to increase or decrease the brightness of an image.

2.4.2 Applications on NEQR

NEQR encodes the color information into the basis states of a further sequence of qubits. This allows for a much more flexible elaboration of the intensity of the pixels since it inherits every property of binary calculation. In the following section, some color transformations that were found useful in more advanced algorithms are presented.

Complement Color Transformation

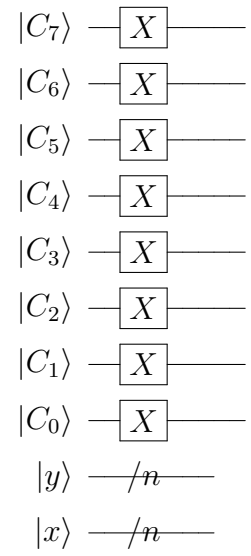
The complement color transformation inverts the grey-scale value of all the pixels [14]. It is implemented through the use of NOT gates and it is defined as:

$$U_C = X^{\otimes q} \otimes I^{\otimes 2n} \tag{2.53}$$

By applying this transformation to the NEQR image defined in section **??**, the result is

$$\begin{aligned} U_C(|I\rangle) \quad &= U_C(\frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}|f(Y,X)^i\rangle|YX\rangle) = \\ &= \frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}\bigotimes_{i=0}^{q-1}(X|C^i_{YX})\rangle|YX\rangle = \\ &= \frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}|2^q-1-f(Y,X))\rangle|YX\rangle) \end{aligned} \tag{2.54}$$

An example of the complement transformation circuit applied on a $2^n \times 2^n$ image is depicted below.



As it can be seen from the circuit, a NOT gate is required on all of the color qubits, therefore the complexity of this algorithm is $O(q)$.

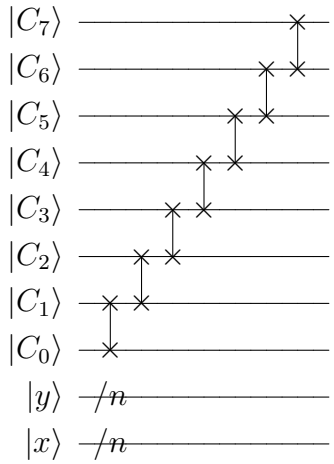
Halving Operation

The halving operation U_H is presented in [15] and consists in reducing by half the grey-scale information of all of the pixels, consequently halving the range as well.

It is used to calculate gradients between the intensity values of the pixels. When applied to a NEQR image, it produces outputs of the form:

$$\begin{aligned} U_H(|I\rangle) &= U_H(\frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}|f(Y,X)^i\rangle|YX\rangle) = \\ &= U_H(\frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}\bigotimes_{i=0}^{q-1}|C_{YX}^i\rangle|YX\rangle = \\ &= \frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}(|C_{YX}^0\rangle\bigotimes_{i=0}^{q-2}|C_{YX}^{i+1}\rangle)|YX\rangle) = \\ &= \frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}(|C_{YX}^0\rangle|f(Y,X)/2\rangle)|YX\rangle) \end{aligned} \tag{2.55}$$

This operation is implemented by swapping the position of each color qubit with its neighbor, shifting them from the LSB to the MSB. An example of the circuit is depicted below.



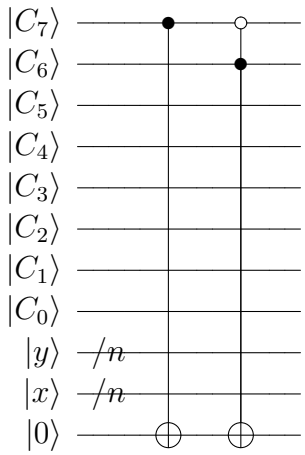
As can be seen from the circuit, $q-1$ swap gates are employed for this algorithm, which means that the complexity of this operation is $O(q)$.

Classification Operation

The classification operation U_T is also presented in [15]. It consists in confronting the whole sequence of color qubits with a threshold, effectively confronting all of the pixels' intensity at the same time. The result of the comparison is stored into an auxiliary qubit. This procedure is also called image segmentation and it reduces the intensity range of the image to only two possible values, 0 and 1. Applying this operation to an NEQR image gives out an output of the form

$$U_T(|I\rangle) = U_T(\frac{1}{2^n}\sum_{Y=0}^{2^n-1}\sum_{X=0}^{2^n-1}|f(Y,X)^i\rangle|YX\rangle) =$$
$$= \frac{1}{2^n}(\sum_{f(Y,X)\geq T}|f(Y,X)\rangle|YX\rangle|1\rangle + \sum_{f(Y,X)<T}|f(Y,X)\rangle|YX\rangle|0\rangle) \quad (2.56)$$

The choice of the threshold is limited to powers of two for a straightforward and efficient implementation. This limitation means that the configurable threshold values are denser in the lower part of the intensity range. Nevertheless, it is in line with the application intended for this operation, which is comparing gradients of intensities of neighboring pixels, which have typically low values.
To implement this algorithm, $C^k - NOT$ gates are applied on the auxiliary qubit, depending on the threshold. If the threshold is, for example 2^{q-2}, the two highest color qubits are used as controls. In this way, if either one is equal to 1 the intensities exceeds the threshold, the auxiliary qubit is put in state $|1\rangle$. The circuit referred to this example is depicted below.



The complexity of this circuit depends on the chosen threshold. The worst case scenario is when the threshold is equal to $2^0 = 1$. In that case the complexity is $O(q^2)$, as for the position shifting operation of section 2.3.5.

Addition/Subtraction Operation

In the NEQR framework, it is possible to define the addiction and subtraction of the intensity of two images because of its binary embedding. This operation is also used in applications like feature extraction of images. For what concerns the addition of two images, the pixels of the resulting image will have the arithmetic additions of the gray-scale values of the corresponding input images [15]. The two $2^n \times 2^n$ NEQR images are defined as

$$|I_A\rangle = \frac{1}{2^n}\sum_{YX=0}^{2^{2n}-1}|A_{YX}\rangle|YX\rangle$$
$$|I_B\rangle = \frac{1}{2^n}\sum_{YX=0}^{2^{2n}-1}|B_{YX}\rangle|YX\rangle \tag{2.57}$$
$$\tag{2.58}$$

where $|A_{YX}\rangle = \bigotimes_{i=0}^{q-1}|a_i\rangle$ and $|B_{YX}\rangle = \bigotimes_{i=0}^{b-1}|b_i\rangle$.

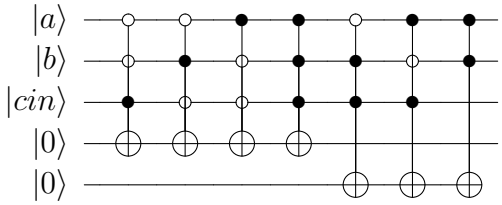
The resulting image $|I_C\rangle$ will have gray-scale values that are the sum of A_{YX} and B_{YX}.

$$|I_C\rangle = \frac{1}{2^n}\sum_{YX=0}^{2^{2n}-1}|C_{YX}\rangle|YX\rangle = \frac{1}{2^n}\sum_{YX=0}^{2^{2n}-1}|A_{YX}+B_{YX}\rangle|YX\rangle \tag{2.59}$$

The result $|C_{YX}\rangle$, being the sum of $A_{YX}, B_{YX} \in [0,2^q-1]$ will be included in the range $[0,2^q]$, therefore $q+1$ are needed to store the result. To obtain this operation, a quantum full adder is designed in [15]. The building blocks of a classical full adders are 1-bit full-adders, which have three inputs (a, b, the two inputs bits; cin, the previous carry bit) and two outputs (sum, the result of the addition; $cout$ the current carry bit). The quantum counterpart is designed in an analogous way, with the previously mentioned inputs that control through the use of $C^3 - NOT$ and Toffoli gates, the two outputs, for a total of 7 elementary gates. The quantum addition operation can be defined as:

$$|a\rangle|b\rangle|cin\rangle|0\rangle|0\rangle \longrightarrow |a\rangle|b\rangle|cin\rangle|sum\rangle|cout\rangle \tag{2.60}$$

A schematic of the circuit is depicted below.



To build a q-qubit sum circuit, q 1-qubit adders are needed. In Figure 2.17, a schematic of the quantum full adder is provided, where $|A\rangle = \bigotimes_{i=0}^{q-1}|a_i\rangle$ and $|B\rangle = \bigotimes_{i=0}^{q-1}|b_i\rangle$ are the input numbers, $|C\rangle = \bigotimes_{i=0}^{q}|a_i\rangle$ is the result and $|F\rangle = \bigotimes_{i=0}^{q-1}|f_i\rangle$ are the auxiliary qubits. The circuit is initialized with the values of the two inputs, while the result $q+1$ qubits and the q auxiliary qubits are initialized at $|\mathbf{0}\rangle$.

Quantum image subtraction of two images $|I_A\rangle$ and $|I_B\rangle$ can be obtained by the same circuit, using the complement color operation presented in section 2.4.2 to invert the gray-scale values of $|I_B\rangle$, obtaining

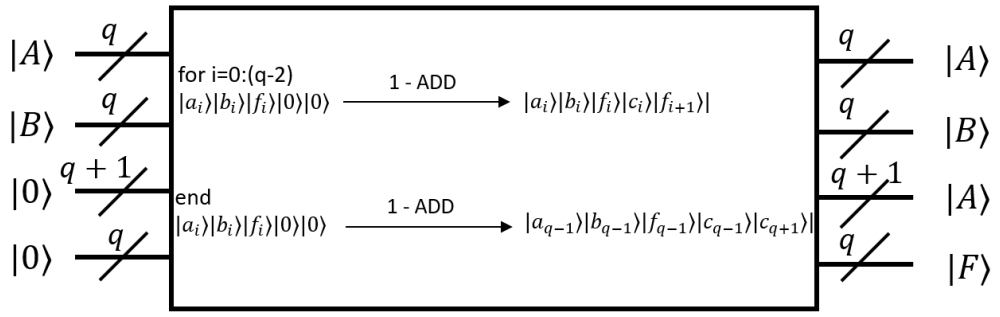


Figure 2.17. q-qubit sum circuit

$$|I_{\overline{B}}\rangle = U_C(|I_B\rangle),$$

and then summing $|I_A\rangle$ and $|I_{\overline{B}}\rangle$ together

$$|I_C\rangle = qSUB(|I_A\rangle, |I_B\rangle) = qADD(|I_A\rangle, |I_{\overline{B}}\rangle). \tag{2.61}$$

The result of the subtraction will be in the range $[-(2^q - 1), 2^q - 1]$ but the result of the subtraction will be in two's complement. Therefore the $q + 1$ qubit is needed to reinterpret the number at the retrieval.

Since q 1-qubit full adder are needed to construct a q-qubit full adder, which have complexity $O(7)$, the whole block has complexity $O(7q)$. It is worth noticing that this schematic is not the most efficient one, but it is the one that preserves the information about the original images, because it uses auxiliary qubits to store the result.

2.5 Image Compression

The limited computational resources of quantum hardware and the typical dimensions of images to be processed require a reduction of the circuits' complexity for a robust implementation of quantum image processing strategies. Decreasing the number of gates used to build a QImR state can be achieved by grouping redundant gray-scale information of the image. Therefore compression algorithm for FRQI and NEQR are presented in the following sections.

2.5.1 Applications on FRQI

In the FRQI encoding method, as seen in section 2.2.1, the encoding of the image has a complexity $O(2^{4n})$, which increases drastically as the image growths in

size. An algorithm to reduce the number of controlled rotations, which embed the intensity information, is presented in [3]. It consists in grouping the pixels with the same intensities, in order to apply the rotation with the corresponding angle the least possible amount of times. The difference is on the conditional part of the gate, which must encode all the positions of the pixels with a certain intensity.
Taking as an example the 8×8 FRQI image represented in Figure 2.18, its gate count would be of 64 $C^6 - R_y$, even though it only contains two intensities values for the whole image. Applying compression on the resources, a circuit that applies only four rotations is achievable, like in the schematic below.

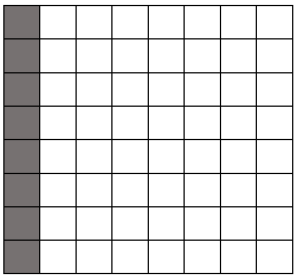
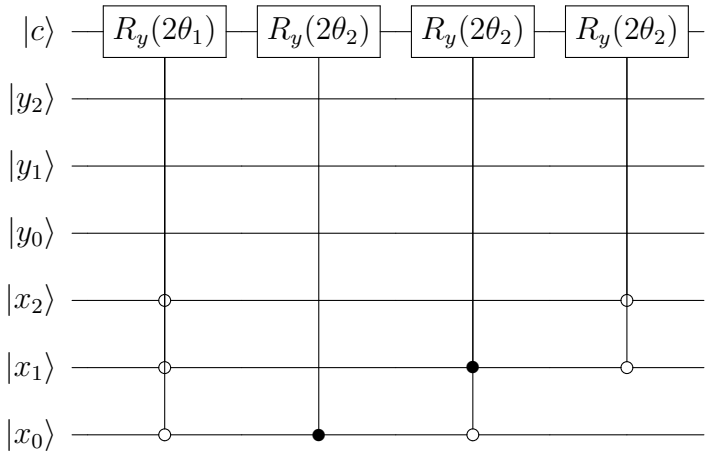


Figure 2.18. 8×8 FRQI image with only two intensity values



The four rotations that are applied in this circuit, not only are considerably less than they would have been with a standard procedure, but they also have less controls, which means that they correspond to less elementary gates.
In order to minimize the number of controls, the positions of the pixels must be transformed into binary strings, with each position qubit corresponding to a Boolean variable. Taking a variable x, if the value of that position qubit is 1, than it will be represented by x in the string, otherwise $\overline{x}$ is used. In this way, a

$1-1$ correspondence is defined between each position and each Boolean term. In Figure 2.18, the pixel position that have a darker color are:

$$|0\rangle, |8\rangle, |16\rangle, |24\rangle, |32\rangle, |40\rangle, |48\rangle, |56\rangle.$$

Their corresponding binary strings are represented by:

$$\begin{aligned}
&|0\rangle \longrightarrow |000000\rangle \longrightarrow \overline{x_5 x_4 x_3 x_2 x_1 x_0} \\
&|8\rangle \longrightarrow |001000\rangle \longrightarrow \overline{x_5 x_4} x_3 \overline{x_2 x_1 x_0} \\
&\dots \\
&|56\rangle \longrightarrow |111000\rangle \longrightarrow x_5 x_4 x_3 \overline{x_2 x_1 x_0}
\end{aligned}$$

The Boolean expression e that takes into consideration all of these positions is described by the sum of its terms:

$$e = \overline{x_5 x_4 x_3 x_2 x_1 x_0} + \overline{x_5 x_4} x_3 \overline{x_2 x_1 x_0} + x_5 x_4 x_3 \overline{x_2 x_1 x_0} \tag{2.62}$$

The minimized expression that represents all of the listed positions is given by

$$e = \overline{x_2 x_1 x_0} \tag{2.63}$$

The number of terms in the expression translates the number of rotations to be applied. In this case only one rotation is needed instead of 8, conditioned by the lowest three qubits being equal to 0, as seen in the circuit. The minimization of Boolean expressions like 2.5.1 has been studied extensively. The Espresso algorithm is a widely used option for achieving minimization of Boolean expressions, and does so within a reasonable running time [16]. The amount of compression applied to the circuit by this algorithm depends on how much the image's pixels are similar to each other. An image that does not have any pixel with the same intensity as another one, will not be compressed at all.

The flow chart of the quantum image compression algorithm for FRQI is depicted in Figure 2.19.

Even though this algorithm apparently implements lossless compression, there are cases in which it introduces errors into the image. When minimizing boolean functions, algorithms like the Espresso do not take into consideration that the minimized expression might be redundant. Assuming, for example, that an image has pixels in position 01, 10 and 11 of the same intensity. Their minimized version corresponds to $'x1', '1x'$. When applying $C-R_y$ gates with the corresponding conditions, the pixel in position 11 will be embedded two times.

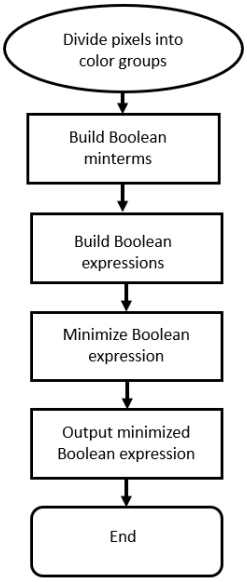


Figure 2.19. Flow chart of FRQI image compression

2.5.2 Application on NEQR

NEQR, as presented in table 2.1, has a significantly lower complexity compared to FRQI. Nevertheless, compression can be applied to further reduce the resources needed to implement it, using some of the same concepts as for the FRQI compression algorithm, as presented in [5]. In the case of NEQR, the positions are still grouped and minimized, but the compression takes place at the intensity qubit level, instead of being pixel-wise. In order to understand this process, *Step*2 (**??**) of the NEQR algorithm is grouped in an operation set Φ, defined as

$$\Phi = \bigcup_{Y=0}^{2^n-1} \bigcup_{X=0}^{2^n-1} \bigcup_{i=0}^{q-1} \Phi_{YX}^{i}, \quad \Phi_{YX}^{i} \in \{I, C^{2n}NOT\} \tag{2.64}$$

where Φ_{YX}^{i} represents the quantum operation for the i-th qubit in the color qubit sequence of the pixel in position (Y, X). This operation set can be divided into q groups Φ_i, each one representing the quantum operations applied on the same color qubit.

$$\Phi = \bigcup_{Y=0}^{2^n-1} \bigcup_{X=0}^{2^n-1} \bigcup_{i=0}^{q-1} \Phi_{YX}^{i} = \bigcup_{i=0}^{q-1} \left(\bigcup_{Y=0}^{2^n-1} \bigcup_{X=0}^{2^n-1} \Phi_{YX}^{i} \right) = \bigcup_{i=0}^{q-1} \Phi_i \tag{2.65}$$

Whether the type of operation applied by Φ_{YX}^{i} is an Identity gate or a $C^{2n}NOT$, depends on the value of C_{YX}^{i}. So the operation Φ_i on the i-th qubit can be divided into groups

$$\begin{aligned}\Phi_i &= \bigcup_{Y=0}^{2^n-1}\bigcup_{X=0}^{2^n-1}\Phi_{YX}^i \\ &= (\bigcup_{Y=0}^{2^n-1}\bigcup_{X=0,C_{YX}^i=0}^{2^n-1} I) \bigcup (\bigcup_{Y=0}^{2^n-1}\bigcup_{X=0,C_{YX}^i=1}^{2^n-1} (C^{2n}NOT)_{YX}) \end{aligned} \tag{2.66}$$

Because the Identity gates do not alter the quantum state, the first part of the operation Φ_i can be ignored. The second part instead, is the one that will be compressed, specifically the position information that controls the gate, in order to reduce the occurrences of C^kNOT on the specific color qubit. In the same way as for the rotation gates in FRQI, the controls of $C^{2n}NOT$ will be reduced, the difference is that the operations that are being grouped are the $C^{2n}NOT$ on the same qubit. In Figure 2.20, half of the pixels are black and the other half is white, which are encoded as $|00000000\rangle$ and $|11111111\rangle$ respectively. The operations needed to encode the image can therefore be divided into two groups, following the described procedure:

$$\begin{aligned}\Phi_i &= \bigcup_{Y=0}^{2^n-1}\bigcup_{X=0}^{2^n-1}\Phi_{YX}^i \\ &= (\bigcup_{Y=00}^{11}\bigcup_{X=00}^{01} I_{YX}) \bigcup (\bigcup_{Y=00}^{11}\bigcup_{X=10}^{11} (C^4NOT)_{YX}) \end{aligned} \tag{2.67}$$

$$i = 0, \ldots, q-1 \tag{2.68}$$

Ignoring the set of Identity gates, the positions that apply a C^kNOT gate on the color qubits are:

$$|2\rangle, |3\rangle, |6\rangle, |7\rangle, |10\rangle, |11\rangle, |14\rangle, |15\rangle.$$

Their corresponding binary strings are

$$\begin{aligned} |2\rangle &\longrightarrow |0010\rangle \longrightarrow \overline{x_3}\overline{x_2}x_1\overline{x_0} \\ |3\rangle &\longrightarrow |0010\rangle \longrightarrow \overline{x_3}\overline{x_2}x_1x_0 \\ &\ldots \\ |14\rangle &\longrightarrow |1110\rangle \longrightarrow x_3x_2x_1\overline{x_0} \\ |15\rangle &\longrightarrow |1111\rangle \longrightarrow x_3x_2x_1x_0 \end{aligned}$$

By applying the Espresso algorithm, the resulting Boolean expression is

$$e = x_1$$

Therefore on the i-th color qubit, the number of gates is reduced from 8 $C^4 - NOTs$ to one $CNOT$. In the schematic representing the compressed circuit, for space reasons only the X_1 position qubit is shown, since it is the only one used. The other three position qubits will only be put into superposition.

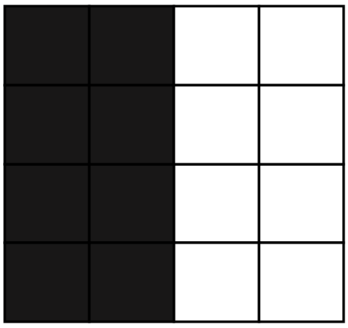
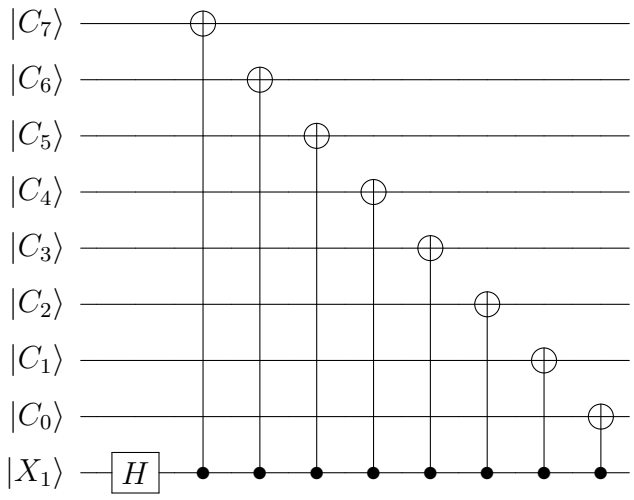


Figure 2.20. 4×4 NEQR image to be compressed



Because this algorithm takes into consideration each color qubit, the preparation for each can be compressed independently, increasing the probability of reducing the computational complexity for any type of image. It is worth noticing that the same problems due to the redundancy of the boolean functions highlighted for FRQI, also apply to the NEQR version of the algorithm.

2.6 Image Spatial Filtering

In order to expand the study of the state of the art of QImP with more advanced but essential algorithms in the image processing domain, quantum image filtering was investigated. Filtering places one self in pre-processing category of images algorithms, whose goal is too enhance the quality of the image for a more precise elaboration. In [17], a quantum image median filtering in the spatial domain for NEQR is presented. Spatial median filtering is a type of non-linear operation, used directly on the pixels' intensity of an image, that replaces the intensity value of each pixel with the median value of its neighbouring pixels. The adjacent pixels of a specific location in the image are sorted by intensity and the median value is used to replace the intensity of the pixel being considered. In the classical domain is proved to be very effective at removing noise from an image.

2.6.1 Building Blocks

The quantum filtering algorithm is based off the use of smaller modules, like the Position Shift Operation encountered in 2.3.5, a Comparator module and Sort module, to be introduced.

Comparator module

The comparator module is used to make a comparison between the neighboring pixels' intensities. It is capable of comparing two integers embedded in the basis states of two sequences of qubits of the same length and identify whether they are equal or which one is the largest. Taking two integers $x = x_{n-1}x_{n-2}\dots x_0$, $y = y_{n-1}y_{n-2}\dots y_0$, with $x_i, y_i \in \{0,1\}$, $i = 0,1,\dots,n-1$, the comparator uses the outputs e_1, e_0 to express which one is the largest. The results will be:

- $e_1e_0 = 10$ if $x > y$
- $e_1e_0 = 01$ if $x < y$
- $e_1e_0 = 00$ if $x = y$

As can be seen in the schematic below, the comparator module has n groups of quantum gates, one for each qubit in the sequence. Each group compares couples of qubits from the two sequences that are the same position, and stores the the partial results in two auxiliary qubits and on the e_1e_0 as well. In total, the number of auxiliary qubits added are $2n$. The controls k of the NOT gates increase the higher the index i of the group is, starting from the MSBs, in order to take into consideration the previous comparations as well. Therefore the controls on the gates for group i are $2n+2$. The complexity of the comparator module is $O(n^2)$, as detailed in the reference [18]. The symbol used to represent this circuit is instead depicted in Figure 2.21.

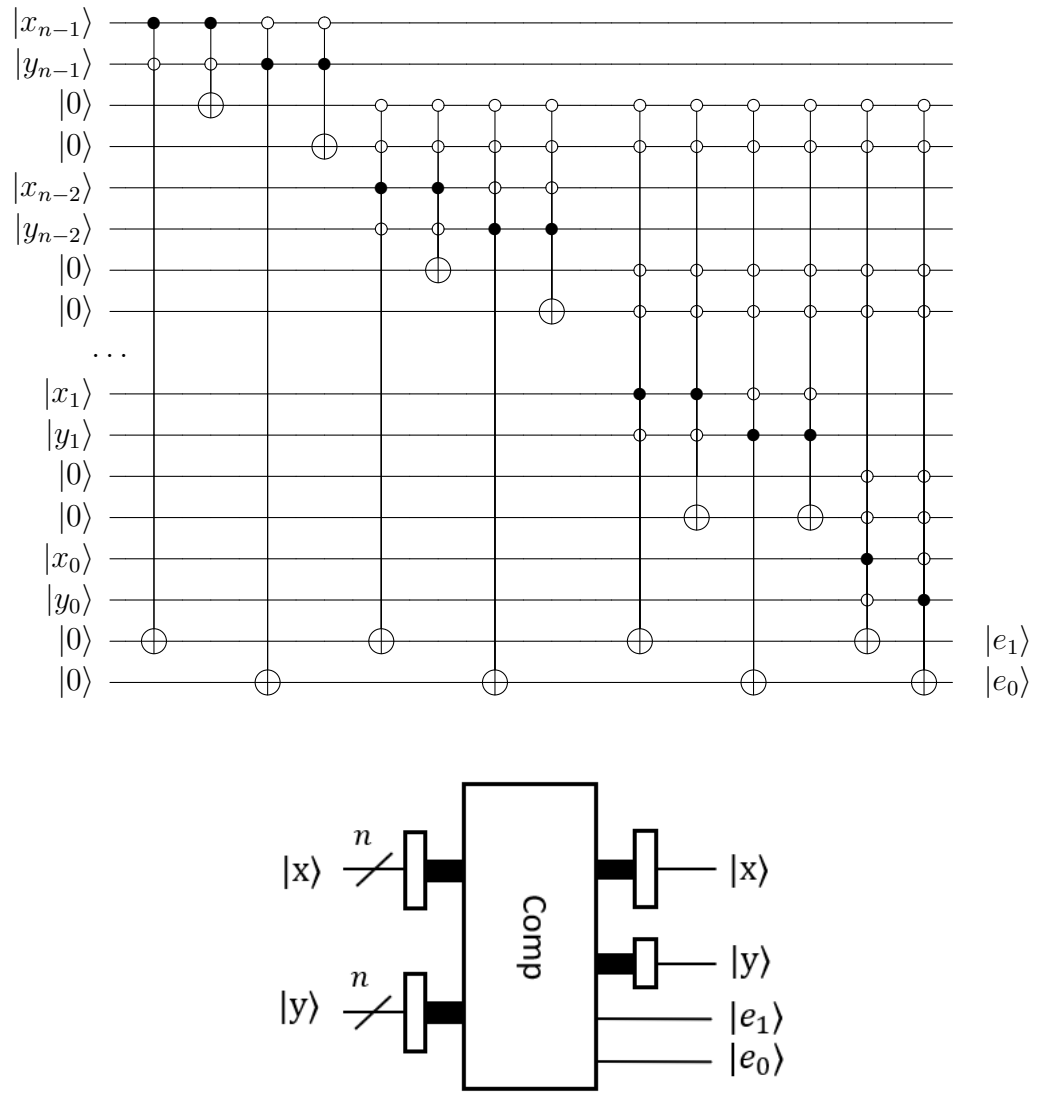


Figure 2.21. Symbol for the comparator circuit

Sort

The Sort module makes use of the *Comparator* and a *Swap* module to change the order of the two integers that represent the intensity. The Swap module is simply composed by $C^2 - SWAP$, controlled by the outputs e_1e_0 of the comparator, as depicted in the schematic below followed by its symbol 2.22. The inputs a and b are sent to the comparator, which will activate the swap gates based on whether its result is 01 or not, in order to put the larger integer is on the higher position. The Sort module schematic is reported in Figure 2.23.

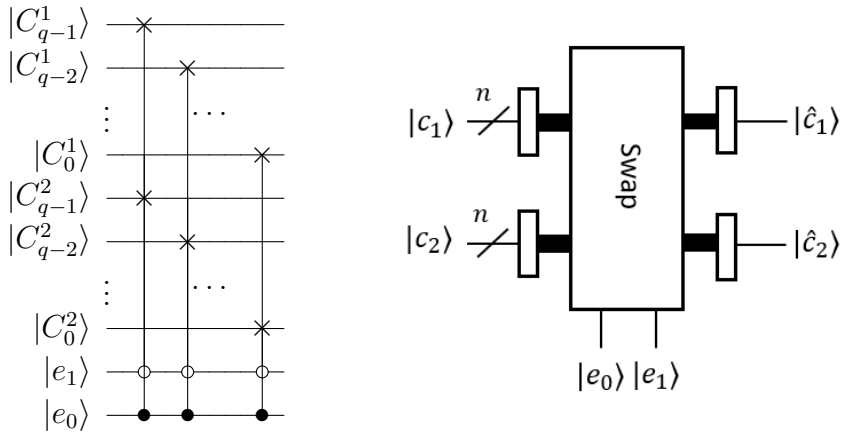


Figure 2.22. Swap module symbol

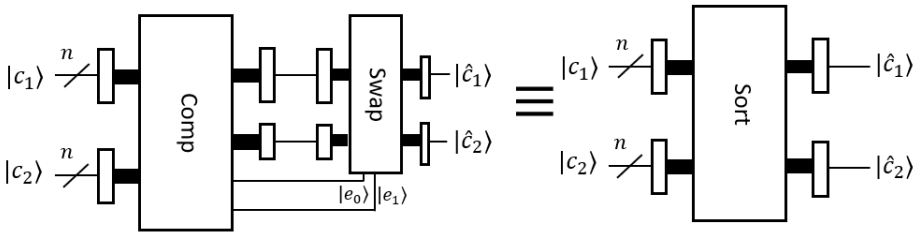


Figure 2.23. Sort module schematic and symbol

2.6.2 Workflow of the algorithm

The implementation of this algorithm consists in creating a 3×3 neighboring window around each pixel, in order to identify the median value of that window. Taking for example the image in Figure 2.24, a pixel and his neighboring window are highlighted.

Figure 2.24. Neighboring window of a pixel

The 8 copies are shifted in different directions in order to create images that have the adjacent pixels in the same position as the one taken into consideration. Figure 2.25 highlights the neighboring pixels of the window in each shifted copy.

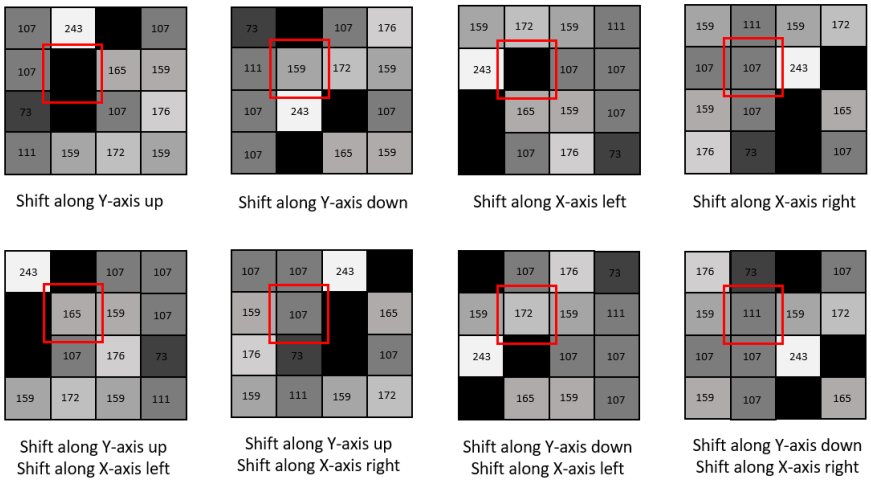


Figure 2.25. Shifted copies of the image in Figure 2.24

After obtaining the neighboring window, a *Median filtering* operation is achieved by applying 30 *Sort* modules on the intensity qubits of the 9 images, like in Figure 2.26. In this way, the intensities are ordered from the highest to the lowest, and, by taking only the value in the middle $|\hat{c}_5\rangle$, the filtering operation is completed. Exploiting the superposition of states, each pixel in the image can be evaluated in parallel. To make sure that the comparison is taking place in the same position in each image, 8 *Comparator* modules are used to compare the position qubits of the original and the 8 shifted copies. The results of this comparisons will control the application of the median filtering operation.

2.6.3 Dimensional Analysis

This algorithm, although it exploit quantum parallelism evaluating the neighboring windows of all the pixels at the same time, requires a great amount of resources to be implemented. As seen in the presentation of the *Comparator* module, in order to compare qubit by qubit its two input sequences, the comparator adds $2n$ auxiliary qubits. For a $2^n \times 2^n$ NEQR image, the qubits needed to apply quantum median filtering are:

- $9 \cdot (2n + q)$ qubits the encode the original image and the copies to be shifted.
- $8 \cdot 2 \cdot 2n$ auxiliary qubits to make the comparison between the positions of the 9 images.

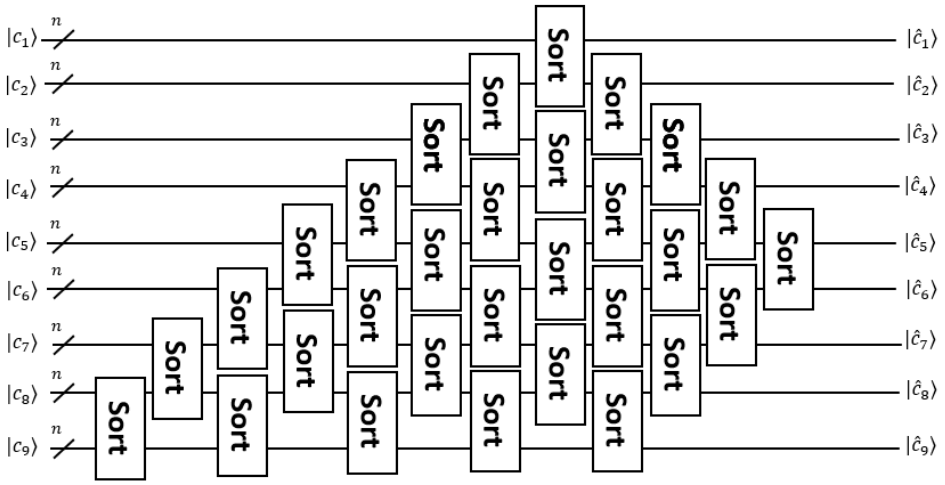


Figure 2.26. Median filtering operation

- $30{\cdot}2{\cdot}q$ auxiliary to compare the intensities in the 30 Sort modules that compose the Median filtering operation.

The total qubit count of the circuit is $N_{tot} = 9{\cdot}(2n{+}q){+}8{\cdot}2{\cdot}2n{+}30{\cdot}2{\cdot}q$. To save resources, some of the auxiliary qubits can be shared in between the same modules. The 8 compare modules applied on the position qubits can share the auxiliary qubits, except for the ones that encode the result of the comparison. Furthermore, since the sort modules that are used in parallel are just four, the auxiliary qubits of their comparator modules can be shared inside the median filtering operation. In this way the total amount of qubits would actually be $N_{tot} = (9 \cdot (2n + q)) + (2 \cdot 2n + 7 \cdot 2) + 4 \cdot 2q$. For a 4×4 with an intensity range expressed by 8 qubits, the corresponding circuit would have 194 qubits. The number of qubits required to implement this algorithm is comparable to the computational capabilities of the most sophisticated quantum computers available today, even for very small images. Therefore, given the impossibility of actual application scenarios in real quantum hardware, the algorithm is deemed not suitable to be actually implemented, but its modules instead allow to enrich the processing possibilities of NEQR.

2.7 Quantum Edge Detection

Edge detection is an image processing technique for identifying the borders of an image. It has important applications in areas like computer vision and pattern recognition, where the speed of the edge detection algorithm plays an important role. The increasing volume of information of higher resolution images weights on

the computational capabilities of classical edge detection pixel-by-pixel methods. A classical edge extraction algorithms for $2^n \times 2^n$ image is know to have lower bound complexity of $O(2^{2n})$ [19]. Quantum edge detection is an emerging field in which algorithms with exponential speed-up have been proposed for the encoding techniques discussed in the previous sections. They exploit quantum parallelism to evaluate all the pixels simultaneously, allowing to lower the complexity of the edge detection techniques by reducing the number of iterations of the algorithm.

2.7.1 Quantum Sobel Edge Detection for FRQI

Sobel edge detection is one of the most commonly used methods in classical feature extraction. It uses an approximation to the derivative to calculate the gradients between the intensities of the pixels in a neighboring window and assigns the edges of the image to the positions where the gradient is highest [19], comparing them to a threshold.

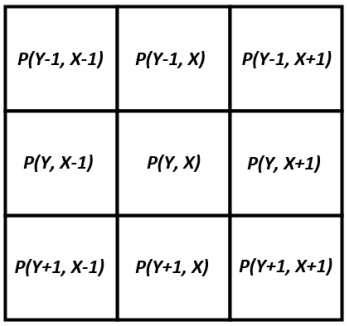


Figure 2.27. Pixel neighborhood window

Referring to the schematic of a 3×3 neighboring window in Figure 2.27, where $p(Y, X)$ is the gray-scale value in position (Y, X), the gradients along the Y- and X-axis are approximated as

$$\begin{aligned} G_x \quad &= (p(Y+1, X+1) + 2p(Y+1, X) + p(Y+1, X-1)) - \\ &- (p(Y-1, X+1) + 2p(Y-1, X) + (p(Y-1, X-1)) \end{aligned} \tag{2.69}$$

$$\begin{aligned} G_y = \quad &(p(Y+1, X+1) + 2p(Y, X+1) + p(Y-1, X+1)) - \\ &- (p(Y+1, X-1) + 2p(Y, X-1) + (p(Y-1, X-1)) \end{aligned} \tag{2.70}$$

while the overall gradient is given by

$$g = \sqrt{G_x^2 + G_y^2} \tag{2.71}$$

A quantum version of this algorithm, QSobel, has been proposed for FRQI in [20]. It consist in shifting the image in order to have access to all the pixels neighboring window simultaneously, in a similar way as in the median filtering algorithm presented in 2.6. The difference in this technique is that there are no copies of the entire image, only the color qubit is copied and stored in an auxiliary qubit after each shifting operation. In this way the gray-scale values of the neighboring windows can be compared to calculate their gradients. In Figure 2.28, a schematic of the QSobel circuit is shown, where the $Uy+$, $Ux+$, $Uy-$, and $Ux-$ are position shifting operations in the four directions along the Y- and X-axis. U_S is the copy operation defined as

$$U_S(|C\rangle|0\rangle) = |C\rangle|C\rangle \tag{2.72}$$

and U_Ω is a quantum black box used to calculate the gradients. An auxiliary qubits $\Omega(Y, X)$ is added to the circuit to store the result of U_Ω which is defined as

$$|\Omega(Y,X)\rangle = \begin{cases} |0\rangle, \sqrt{G_x(Y,X)^2 + G_y(Y,X)^2} < T \\ |1\rangle, \sqrt{G_x(Y,X)^2 + G_y(Y,X)^2} \geq T \end{cases} \tag{2.73}$$

where $G_x(Y, X)$ and $G_y(Y, X)$ are the approximations of the derivatives of the X-axis and Y-axis. They are expressed as

$$\begin{aligned} G_x(Y,X) &= (C_{Y-1,X+1} + 2C_{Y,X+1} + C_{Y+1,X+1}) - (C_{Y-1,X-1} + 2C_{Y,X-1} + C_{Y+1,X-1}) \\ G_x(Y,X) &= (C_{Y+1,X-1} + 2C_{Y+1,X} + C_{Y+1,X+1}) - (C_{Y-1,X-1} + 2C_{Y-1,X} + C_{Y-1,X+1}) \end{aligned} \tag{2.74}$$

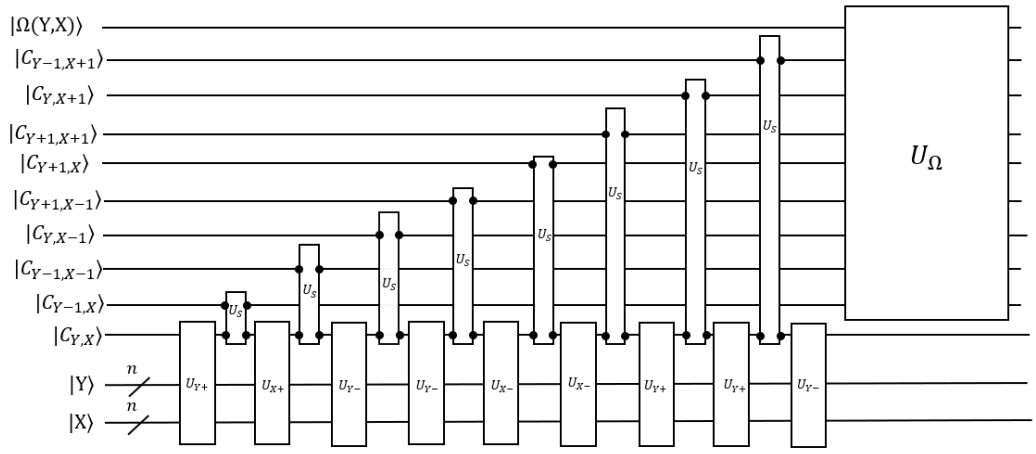


Figure 2.28. QSobel schematic

Considering that the intensity of the pixels is embedded in the phase of the color qubit, manipulating that information does not have the flexibility that the calculations of the gradients require. There are no studies yet that suggest how such a calculations should be achieved. It is also unclear how the copy operation should be implemented, since the cloning of an unknown quantum state is prohibited by one of quantum computing most important theorems [2]. For these reasons, this algorithm does not have any prospect of implementation yet and therefore can not be tested on realist scenarios.

2.7.2 Quantum Edge Detection for NEQR

In [15], a local feature point extraction algorithm is presented to find the boundaries of an image encoded with the NEQR technique. Its based off the use of position shifting operation and off the color transformations defined in 2.4.2. The steps used in this method are similar to the median filtering and the Sobel algorithm:

- *Step 1* consists in preparing 9 copies of the same image with the NEQR method.
- *Step 2* consists in shifting 8 of the copies in different directions to compare the neighboring pixel window.
- In *Step 3* the quantum addition operation and the halving operation are used to compute the gradients between the intensities.
- Finally, in Step 4, the classification operation is used to compare the gradients with a fixed threshold in order to identify the borders.

As seen for the median filtering algorithm, the 3×3 neighboring window is created by preparing 8 extra copies of the image and their intensities are used to compute the gradients between the pixels. In this algorithm, the method used to approximate the gradients is called the zero-cross method, which uses four sub-gradients defined as

$$\begin{aligned} G_1 &= |2C_{Y,X} - (C_{Y+1,X} + C_{Y-1,X))}|/2 \\ G_2 &= |2C_{Y,X} - (C_{Y+1,X+1} + C_{Y-1,X-1))}|/2 \\ G_3 &= |2C_{Y,X} - (C_{Y+1,X+1} + C_{Y-1,X-1))}|/2 \\ G_4 &= |2C_{Y,X} - (C_{Y+1,X-1} + C_{Y-1,X+1}|/2 \end{aligned} \tag{2.75}$$

where $C_{Y,X}$ is the gray scale of the pixel (Y, X). In order to achieve this computation, the addition/subtraction operation $qADD/qSUB$ (2.4.2) and the halving operation U_H (2.4.2) are used in the following way:

$$
\begin{aligned}
&|\phi_0\rangle = qADD(|I_{Y,X}\rangle, |I_{Y,X}\rangle), && \\
&|\phi_1\rangle = qADD(|I_{Y+1,X}\rangle, |I_{Y-1,X}\rangle), && |\phi_2\rangle = qSUB(|\phi_0\rangle, |\phi_1\rangle\rangle), \\
&|\phi_3\rangle = qADD(|I_{Y+1,X+1}\rangle, |I_{Y-1,X-1}\rangle), && |\phi_4\rangle = qSUB(|\phi_0\rangle, |\phi_3\rangle), \\
&|\phi_5\rangle = qADD(|I_{Y,X+1}\rangle, |I_{Y,X-1}\rangle), && |\phi_6\rangle = qSUB(|\phi_0\rangle, |\phi_5\rangle), \\
&|\phi_7\rangle = qADD(|I_{Y+1,X-1}\rangle, |I_{Y-1,X+1}\rangle), && |\phi_8\rangle = qSUB(|\phi_0\rangle, |\phi_7\rangle), \\
&|G_1\rangle = U_H(|\phi_2\rangle), && |G_2\rangle = U_H(|\phi_4\rangle), \\
&|G_3\rangle = U_H(|\phi_6\rangle), && |G_4\rangle = U_H(|\phi_8\rangle).
\end{aligned} \tag{2.76}
$$

(2.77)

After the gradients are calculated, they are compared with a fixed threshold using the operation U_T 2.4.2. The results will be stored on four additional qubits $|Z_i\rangle$, with $i = 0, \ldots, 3$. When measuring the circuits, only the positions that have the four gradients greater than then selected threshold will be identified as edges.

The circuit implementing this edge detection algorithm for NEQR will make use of 5 *Addition* modules and 4 *Subtraction* modules, which require respectively $2q + 1$ and $2q$ auxiliary qubits each. As said, four more qubits are needed for the classification operations. Considering the qubits needed to embed the 9 $2^n \times 2^n$ images with a gray-scale range of $[0, 2^q - 1]$, the total number of qubits needed to implement the algorithm is

$$N_{tot} = 9 \cdot (2n + q) + 5 \cdot (2q + 1) + 4 \cdot (2q) + 4$$

Applying edge detection on a 4×4 NEQR image with 256 intensity value with this algorithm, would require a 270 qubits circuit. With the same considerations done for the median filtering algorithm, the circuit does not seem like an applicable option nowadays, but its modules are useful in the prospect of further processing.

2.7.3 Quantum Hadamard Edge Detection

The Quantum Hadamard Edge Detection algorithm (QHED) is a technique proposed in [21] that allows feature extraction for images encoded with QPIE method 2.2.3. It exploits the properties of Hadamard gates to compute the gradients of adjacent pixels with an extremely low complexity.

To identify the boarders of an N-pixel image encoded with QPIE, the gradient between two neighboring pixel can be approximated as a subtraction between the amplitude values of two adjacent states, i and $i+1$, with $i \in \{0, N-1\}$. Therefore two adjacent pixels in QPIE state will have amplitudes c_i and c_{i+1}:

$$I = c_0|0\rangle + c_1|1\rangle + \cdots + c_i|i\rangle + c_{i+1}|i+1\rangle + \cdots + c_{N-2}|N-2\rangle + c_{N-1}|N-1\rangle$$

If the result of the subtraction is below a certain threshold, the two pixels belong to the same area, otherwise an edge has been identified. In order to compute the gradient, the Hadamard gate can exploited. Its application on a single qubit has outputs of the form

$$|0\rangle \longrightarrow \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle \tag{2.78}$$

$$|1\rangle \longrightarrow \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle \tag{2.79}$$

Applying an Hadamard gate on the last qubit of a 2^N-qubit system will result in

$$I_{2^{n-1}} \otimes H_0 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0\dots & 0 & 0 \\ 1 & -1 & 0 & 0\dots & 0 & 0 \\ 0 & 0 & 1 & 1 & \dots & 0 & 0 \\ 0 & 0 & 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 \\ 0 & 0 & 0 & 0 & \dots & 1 & -1 \end{bmatrix} \tag{2.80}$$

where $I_{2^{n-1}}$ is the $2^{n-1} \times 2^{n-1}$ Identity matrix. Applying this on a QPIE image gives instead the following state

$$(I_{2^{n-1}} \otimes H_0) \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-2} \\ c_{N-1} \end{bmatrix} \longrightarrow \frac{1}{\sqrt{2}} \begin{bmatrix} c_0 + c_1 \\ c_0 - c_1 \\ c_2 + c_3 \\ c_2 - c_3 \\ \vdots \\ c_{N-2} + c_{N-1} \\ c_{N-2} - c_{N-1} \end{bmatrix} \tag{2.81}$$

In a $2^n \times 2^n = N$ pixel image, two neighboring pixels are represented by position bit-strings where only the LSB changes

$$|b_{n-1}b_{n-2}\dots b_1 0\rangle,\ |b_{n-1}b_{n-2}\dots b_1 1\rangle \tag{2.82}$$

$$b_i \in \{0,1\}. \tag{2.83}$$

Therefore, if the measurement of the qubits in the circuit is conditioned on the LSB being in state $|1\rangle$, the horizontal gradients between *even* pairs of pixels can be obtained.

In order to get the gradients from both even and odd pairs of pixels in the same iteration of the algorithm, an auxiliary qubit can been added to the circuit.

Applying an Hadamard gate to this qubit, initialized to the state $|0\rangle$, results in the redundant state

$$|I\rangle \otimes \frac{(|0\rangle + |1\rangle)}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} c_0 \\ c_0 \\ c_1 \\ c_1 \\ c_2 \\ c_2 \\ \vdots \\ c_{N-2} \\ c_{N-2} \\ c_{N-1} \\ c_{N-1} \end{bmatrix} \tag{2.84}$$

In order to easily compute the gradients with the Hadamard gate, the amplitudes have to shift back of one position to obtain the state:

$$(c_0, c_1, c_1, c_2, c_2, c_3, \ldots, c_{N-2}, c_{N-1}, c_{N-1}, c_0)^T \tag{2.85}$$

To achieve this, an amplitude permutation is defined in [22]. It can be efficiently be decomposed into a set of NOT and $C^k - NOT$ gates with a $O(poly(n)$ complexity. This operation is referred to as a *Decrement gate*, which is described by the unitary

$$D_{2^{n+1}} = \begin{bmatrix} 0 & 1 & 0 & 0 \ldots & 0 & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 0 & 1 \\ 1 & 0 & 0 & 0 & \ldots & 0 & 0 \end{bmatrix} \tag{2.86}$$

After the permutation, applying the Hadamard gate on the auxiliary qubit will give output of the form

$$(I_{2^{n-1}} \otimes H) \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_1 \\ c_2 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-2} \\ c_{N-1} \\ c_{N-1} \\ c_0 \end{bmatrix} \longrightarrow \frac{1}{\sqrt{2}} \begin{bmatrix} c_0 + c_1 \\ c_0 - c_1 \\ c_1 + c_2 \\ c_1 - c_2 \\ c_2 + c_3 \\ c_2 - c_3 \\ \vdots \\ c_{N-2} + c_{N-1} \\ c_{N-2} - c_{N-1} \\ c_{N-1} - c_0 \\ c_{N-1} + c_0 \end{bmatrix} \tag{2.87}$$

Measuring the circuit conditioned on the auxiliary qubit being in state $|1\rangle$ results in obtaining the horizontal gradients. To obtain the vertical gradients, the intensity matrix that represents the digital image has to be transposed.

$$Img = \begin{bmatrix} I_0 & I_1 & \dots & I_{2^n-1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{2^{2n}-2^n} & \dots & \dots & I_{2^{2n}-1} \end{bmatrix}^T \longrightarrow \begin{bmatrix} I_0 & \dots & \dots & I_{2^{2n}-2^n} \\ I_1 & \vdots & \ddots & \vdots \\ I_{2^n-1} & \dots & \dots & I_{2^{2n}-1} \end{bmatrix} \tag{2.88}$$

After that, the transposed image is encoded in a QPIE state and the same process is repeated. After the measurements, the two images are retrieved and their intensity information are added together, obtaining the image stripped to its borders.

A schematic of the QHED algorithm for a 4×4 image is depicted below in Figure 2.29. The qubits denominated as c_i with $i = 0,1,\dots,4$ are the classical bits where the result is stored.

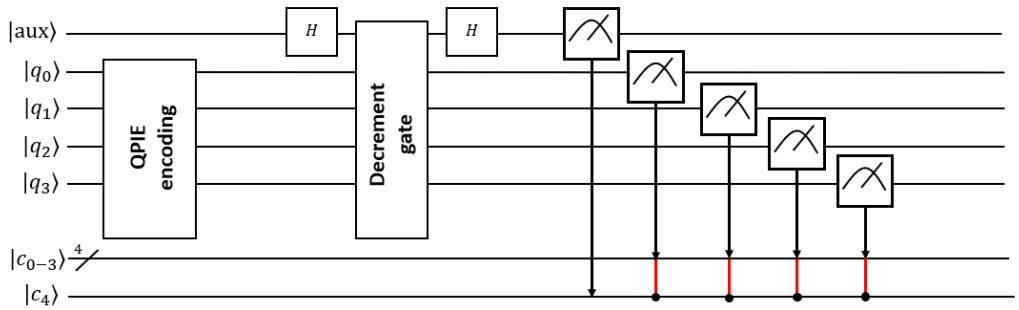


Figure 2.29. QHED schematic

This algorithm is presented to be very efficient, it requires only $2n+1$ qubits and its processing is appointed only to the position shifting operation and the Hadamard gate, which results in a complexity of $O(poly(n))$. Confronted with a classical edge detection algorithm $O(2^{2n})$, QHED gives a super-exponential speed up and is defined by simple and efficient operations, which can be easily implemented.

Chapter 3

Quantum Image Processing: software library

Translating the presented algorithms into circuit descriptions is a necessary step to be able to test and compare the different techniques. For this purpose, a modular Python software library was developed to give the possibility to flexibly compose quantum circuits implementing the techniques presented, for images of arbitrary size. The aim of this chapter is to give an overview of all the functions implemented in the library, providing a brief description for each, followed by the display of some practical use-cases that show all the necessary steps to exploit the supported functionalities, while also presenting the figure of merits used to characterize the results that can be obtained.

3.0.1 Qiskit

The implemented library is based on Qiskit, an open-source software development kit created by IBM that permits to describe quantum circuits and to run them either on simulators or on real quantum devices, through their Cloud quantum computing service. It is made up by four different components (Terra, Aer, Ignis and Aqua), each characterized by a different functionality set [23]. The ones used in this library are Terra and Aer. The Terra component allows the description, visualization and transpilation of the circuits, which is the process of changing the description of the implemented circuit into a given basis of gates, in order to match the topology of a specific quantum device or to optimize the circuit with respect to noise. It can also be used to understand the complexity of the implemented circuit, by looking at its depth and the number of used gates after decomposing it into an elementary set. Terra also allows to manage the test parameters of Qiskit's simulators and devices and to choose the results that are to be extrapolated. The Aer component instead provides a simulator framework for executing circuits compiled in Terra, to which

noise models can also be applied. The IBM quantum computing services includes devices of up to 127 qubits, but the free accessible ones have a maximum of 5 qubits. The functions and methods provided by Qiskit's Terra are used through the library to compose quantum circuits, while Aer simulators are used to test them.

3.1 Overview of the library

The QImP software library is made-up of three main packages, one for each of the encoding methods proposed. Each is divided into two sub-packages, one dedicated to the embedding and retrieval of images and one dedicated to processing algorithms. Furthermore, a package containing a set of testing functions is also present as well as a package for preparing the classical information to be encoded and to extrapolate figures of merit from the implemented circuits and their tests, both applicable across the different methods. The library is organized in such a way to facilitate future expansions, thanks to the code's modularity. The following sub-sections give a a brief overview of what each module does, but for further explications and details, the reader is invited to refer to the extensive documentation produced using Jupyter notebooks. The library's organization is shown in Figure 3.1.

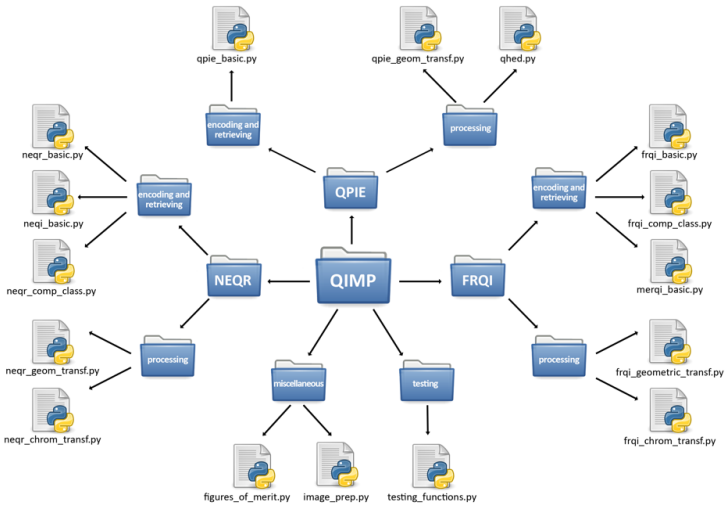


Figure 3.1. Organization of the library

3.1.1 FRQI software library package

The **frqi_software_library.py** package contains all the functions related to FRQI. In its current version its modules are divided into two more sub-packages,

one dedicated to encoding and retrieval functions and one to the processing ones.

Encoding and retrieval sub-package

The functions presented in this category allow the embedding of classical visual information in a quantum circuit using the FRQI technique, its compressed version or its extension to RGB images, MCRQI. The FRQI and MCRQI modules also contain their respective retrieval functions, which are used to decode the results obtained after testing the circuit. The compression module is instead made-up by functions that permit the user to follow the steps of the algorithm, as well as apply the encoding. All of the mentioned modules are set out in the following tables.

frqi_basic.py **module**	
Function name	**Description**
frqi_circuit()	It generates an FRQI circuit description starting from the pixels' intensities of an image.
frqi_decode()	It retrieves the pixel intensities of an image starting from the measurements obtained from an FRQI circuit.

mcrqi_basic.py **module**	
Function name	**Description**
mcrqi_circuit()	It generates an MCRQI circuit description starting from the pixels' intensities of an image.
mcrqi_decode()	It retrieves the pixel intensities of an image starting from the measurements obtained from an MCRQI circuit.

frqi_comp_class.py **module**	
Function name	**Description**
turn_to_funct()	It groups the positions of the pixels by intensity and turns them into boolean functions.
print_funct()	It allows to print the results of the boolean translation of the positions.
compress()	It concatenates the boolean functions referred to the same intensity and minimizes them, then turns them back into strings of character that express the position.

print_pos_comprsd()	It prints the minimized positions minimized.
frqi_image()	It uses the minimized positions and intensity information to generate the description of an FRQI circuit.

Processing sub-package

All the modules used to apply processing on a FRQI image belong in this sub-package. Currently, it is composed by a module dedicated to collecting functions implementing geometric transformations, and one for chromatic transformations.

frqi_geometric_transf.py **module**	
Function name	**Description**
frqi_axis_flip()	It applies the flip operation on an FRQI circuit with respect to one of the axis or both.
frqi_coord_swap()	It applies the coordinate swapping operation on an FRQI circuit.
frqi_ort_rotation()	It applies the orthogonal rotation operation on an FRQI circuit, given the chosen angle.
frqi_restr_flip()	It applies one of the flip operation on a FRQI circuit, in a restricted area chosen by the user.
frqi_restr_coord()	It applies the coordinate swap operation on a FRQI circuit, in a restricted area chosen by the user.
frqi_pos_shift()	It applies the position shifting operation on a FRQI circuit, of a magnitude and direction chosen by the user. The axis along which the shift is applied is also configurable.

frqi_chromatic_transf.py **module**	
Function name	**Description**
frqi_color_compl()	It applies the color complement operation on a FRQI circuit.
frqi_color_change()	It applies the color change operation on a FRQI circuit, of a value chosen by the user.

3.1.2 NEQR software library package

Modules whose functions are designed to generate the description of NEQR circuits and related algorithms, belong in the **neqr_software_library.py** package. Its organization is identical to the one done for the FRQI modules, dividing its encoding and retrieving modules from its processing ones.

Encoding and retrieval sub-package

This category is made-up of modules that allow the embedding of gray-scale images through NEQR and color images through NCQI, as well as the decoding of the measurements obtained from these circuit. A module dedicated to applying compression on NEQR is also present. The functions are described in tables briefly presented in the following tables.

neqr_basic.py **module**	
Function name	**Description**
neqr_circuit()	It generates an NEQR circuit description starting from the pixels' intensities of an image. The number of intensity qubits used in the encoding is chosen by the user.
neqr_decode()	It retrieves the pixel intensities starting from the the measurements of an NEQR circuit.

ncqi_basic.py **module**	
Function name	**Description**
ncqi_circuit()	It generates an NCQI circuit description starting from the pixels' intensities of an image. The number of intensity qubits used is chosen by the user.
ncqi_decode()	It retrieves the pixel intensities starting from the results of the measurements of a NCQI circuit and the number of intensity qubits used.

neqr_comp_class.py **module**	
Function name	**Description**

turn_to_funct()	It groups the positions of the pixels that apply a CNOT gate on the same intensity qubit and turns them into boolean functions.
print_funct()	It allows to print the results of the boolean translation of the positions.
compress()	It concatenates the boolean functions referred to the same intensity qubit and minimizes them, then turns them back into positions strings.
print_pos_comprsd()	It prints the minimized positions.
frqi_image()	It uses the minimized positions and intensity information to encode the image in a NEQR circuit.

Processing sub-package

This sub-package is designed to collect modules related to processing images embedded in NEQR circuits. It is consists of two modules, one for geometric transformations and one for chromatic transformations. It is worth noticing that functions describing the implementation, measurement and decoding of the comparison operation presented in 2.6.1, are present in both processing module, as they can be used to compare the position qubits of the image as well as the intensity ones.

neqr_geometric_transform.py **module**	
Function name	**Description**
neqr_axis_flip()	It applies the one or both flip operation on an FRQI circuit.
frqi_coord_swap()	It applies the coordinate swapping operation on an NEQR circuit.
frqi_ort_rotation()	It applies the orthogonal rotation operation on an NEQR circuit, given the chosen angle.
neqr_restr_flip()	It applies one of the flip operation on a NEQR circuit, in a restricted area chosen by the user.
neqr_restr_coord()	It applies the coordinate swap operation on a NEQR circuit, in a restricted area chosen by the user.
neqr_pos_shift()	It applies the position shifting operation on a NEQR circuit, of a magnitude and direction chosen by the user. The shift along one of the axis is also configurable.

neqr_comparator()	It compares two integers represented by two qubits sequences of equal length.
neqr_comparator$_m$*easure*()	It applies measurement only on the necessary qubits needed to obtain the results of the comparison for each pixel.
neqr_comparator$_d$*ecode*()	It retrieves the results of the comparison between two image intensities.

neqr_chromatic_transform.py **module**	
Function name	**Description**
neqr_color_compl()	Given an NEQR circuit, it applies the color complement operation .
neqr_half_int()	Given an NEQR circuit, it applies the halving operation on it.
neqr_classify_compl()	Given an NEQR circuit, it applies the classification operation on it.
neqr_classification_decode()	Given the results of measurements on a classification operation circuit, it retrieves the binary image.
*qc_add_*1()	It implements a 1-qubit add operation given two qubits.
q_ADD_SUB()	It implements a multi-qubit ADD/SUB operation given two sequences of qubits.
neqr_ADD_SUB_measure()	It applies measurement only on the necessary qubits needed to obtain the resulting image of an ADD/SUB.
neqr_ADD_SUB_decode()	It retrieves the NEQR image resulting from an ADD/SUB.
neqr_comparator()	It compares two integers represented in two qubits sequences of equal lenght.
neqr_comparator_measure()	It applies measurement only on the necessary qubits needed to obtain the results of the comparator module.
neqr_comparator$_d$*ecode*()	It retrieves the results of the comparison between two sequence of qubits.
neqr_sort()	It compares the intensity qubits of two NEQR circuits and sorts them depending on their value.

neqr_sort_measure()	It applies measurement only on the necessary qubits needed to obtain the results from the sort module.
neqr_sort_decode()	It retrieves the results of the sort module, outputting an array containing the informations about the image.

3.1.3 QPIE software library package

The **qpie_software_library.py** package is designed to collect modules related to the QPIE method. As for NEQR and FRQI, the embedding functions are separated from the processing ones.

3.1.4 Encoding and retrieval sub-package

This sub-package currently contains one module of functions implementing the embedding of images with QPIE and decoding necessary to visualize the image after performing measurements on it.

qpie_basic.py **module**	
Function name	**Description**
qpie_circuit()	It generates the description of a QPIE circuit starting from the pixels' intensities of an image.
qpie_decode()	It retrieves the pixel intensities starting from the measurements from a QPIE circuit.

Processing sub-package

The processing sub-package for QPIE is divided into a module that collects the implemented geometric functions and a module dedicated to the QHED algorithm. They are set out in the following tables.

qpie_geometric_transformations.py **module**	
Function name	**Description**
qpie_axis_flip()	It applies the one or both flip operation on an QPIE circuit.
qpie_coord_swap()	It applies the coordinate swapping operation on an QPIE circuit.

qpie_ort_rotation()	It applies the orthogonal rotation operation on an QPIE circuit, given the chosen angle.
qpie_restr_flip()	It applies one of the flip operation on a QPIE circuit, in a restricted area chosen by the user.
qpie_restr_coord()	It applies the coordinate swap operation on a QPIE circuit, in a restricted area chosen by the user.
qpie_pos_shift()	It applies the position shifting operation on a QPIE circuit, of a magnitude and direction chosen by the user. The shift along one of the axis is also available.

qhed.py **module**	
Function name	**Description**
qhed_filter()	It applies the QHED algorithm on a QPIE circuit.
qhed_decode()	It retrieves the gradients between the pixels from a QHED circuit.

3.1.5 Testing package

Inside the QImP software library, a package is dedicated to testing the circuits. It is currently made-up of only one module, that contains functions for running the circuits on both ideal and noisy simulations and on real quantum computers, chosen from the ones made available by Qiskit for free. The testing function run the circuit on ideal and noisy simulations on a local computer, while the test on a real hardware is done on the least busy device of the ones available on the cloud service. Generally speaking, functions in this module can also be used on circuits not generated by this library. All the available functions are listed in the following table.

3.1.6 Miscellaneous package

This package is conceived with the idea of inserting all the functions that do not strictly relate to the description of quantum circuits but still represent useful tools for analyzing the results and handling the information. It is composed by an image preparation module and module dedicated to obtain particular figures of merit.

testing_functions.py **module**	
Function name	**Description**
ideal_simulation()	It transpiles the circuit and tests it through the Qiskit Aer simulator.
noisy_simulation()	It transpiles the circuit and tests it through the Qiskit Aer simulator with a noise model.
device_test()	It transpiles the circuit and tests it through the least busy available IBM device.

Image preparation module

The image preparation module contains function that are useful to speed-up the overall process of handling images on quantum circuits. They check that the images have the right characteristic in terms of color channels and dimensions, to be encoded using the chosen embedding method.

img_prep.py **module**	
Function name	**Description**
image_conv()	It verifies that the image file opened meets the requirements to be encoded with FRQI, NEQR and QPIE, which means being squared and having one color channel.
color_image_conv()	It verifies that the image file opened meets the requirements to be encoded with MCRQI, and NCQI, which means being squared and having three color channels. If the image has an alpha channel, its information is discarded.

Figures of merit

The *figures_of_merit.py* module contains functions that allow the extrapolation of numerical quantities useful to describe the obtained results and to analyze and compare of the algorithms. To verify the faithful representation of visual information in the implemented encoding methods, Peak Signal to Noise Ratio (PSNR) was considered. It is one of the most used quantities for comparing the fidelity of an image in the image processing panorama. Its definition is based on the means squared error (MSE), which for $N \times N$ images I and J is given by

$$MSE = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} [(I(i,j) - J(i,j))^2], \tag{3.1}$$

Consequently, $PSNR$ is defined as

$$PSNR = 20log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right), \tag{3.2}$$

where MAX_I is the maximum intensity value of image I. Two images that are exactly the same, have a $PSNR = infdB$, while a with $PSNR = 0dB$ wthey have a MSE equal to the the maximum value of the original image. A function implementing this calculation is provided in this module, along with a function that permits to evaluate the complexity of the circuit. The functions are listed and described in table 3.1.6.

figures_of_merit.py **module**	
Function name	**Description**
PSNR()	It computes the PSNR between two given array of pixels.
transpile_circuit()	It transpiles the circuit in a set of gates made up by single qubit rotations a cx gates. It prints the Depth and the Operation counts of the transpiled circuit.

3.2 User guide

The purpose of this section is to illustrate the capabilities of the QImP library. In this regard, a few of the implemented techniques will be used in order to demonstrate the supported workflow, reported in Figure 3.2.

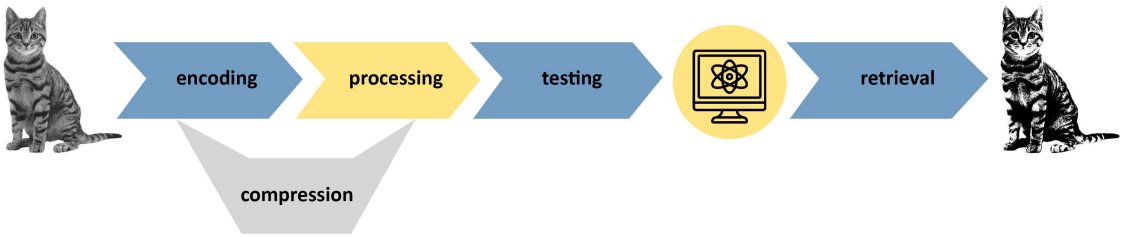


Figure 3.2. Workflow of the QImP software library

3.2.1 Example 1: Image fidelity

The first example shows the necessary steps to open an image file, encode it in a quantum circuit, test it and then analyze the obtained results to verify the fidelity of the selected encoding. To demonstrate this, the FRQI encoding has been chosen, but the same process applies to all three methods. In this example the FRQI image will be embedded using the compressed encoding to illustrate its features as well. It is worth noticing that in the case of QPIE, compression is not included in the library, therefore to implement the same process, only the standard function can be used.

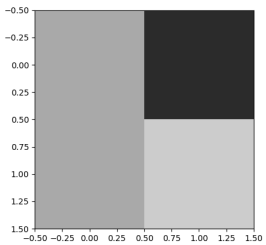
Encoding with Compression

The first step in this procedure is to open an image file and extract the pixel information in array form. This operation can be achieved by using the functions in the *image_preparation.py* module, by specifying the name of the file to be opened. In this example, it is referred to as *sample.png*. To check that the image meets the requirements necessary to be implemented using the FRQI encoding, the *image_conv*() function is used. It converts to gray-scale the image and compares its width and height to verify it is squared. This function also prints out the image and its matrix on the terminal, to have visual and numerical understanding of the file being handled. Independently from the encoding, this function outputs the pixels in array form, in order to facilitate the embedding.

```
px_array=image_conv('sample.png')
```

Output:

```
[170  45
 170 200]
```



As can be seen from the matrix, the image in this example contains has redundant intensity information, making it eligible for being compressed. The compression procedure, as it is defined in the library for both FRQI and NEQR, is divided into several steps. By exploiting the properties of the frqi_comp class, the positions of the pixels can be turned into boolean functions. The result of this procedure is also printed on the terminal

```
img_comp=frqi_comp(px_array)
img_comp.turn_to_funct()
img_comp.print_funct()
```

Output:

```
45 ['~a & b']
170 ['~a & ~b', 'a & ~b']
200 ['a & b']
```

After that, the compress() function applies minimization on the boolean function and turns back into position strings. The print_pos_comprsd() function allows to visualize on the terminal the resulting position strings.

```
img_comp.compress()
img_comp.print_pos_comprsd()
```

Output:

```
45 [' 01']
170 [' x0']
200 [' 11']
```

Now that the information is compressed, the quantum circuit can be described, according to the FRQI formalism. When the circuit of any of the supporting encoding is created using the QImP library, its depth and operation count (i.e. number of gates) are also shown. To understand the complexity of the circuit created, the *transpile* function unrolls the circuit in a base of gates made of Toffoli gates and single qubit rotations, which also shows the depth and operation count after the transpilying has taken place.

```
qc=img_comp.frqi_image()
transpile(qc)
```

Output

```
Depth :  15
Operations:  OrderedDict([('ry', 6), ('barrier', 4),
('x', 4), ('ccx', 4), ('h', 2), ('cx', 2)])

Depth after transpiler :  53
Operations after transpiler:  OrderedDict([('u1', 28),
('cx', 26), ('u2', 10), ('u3', 10), ('barrier', 4)])
```

Testing

In order to evaluate the fidelity of the technique in question, the processing step of the workflow is skipped, to confront the resulting image with the original. To obtain results from the circuit, measurement must be applied on the circuit and then the testing functions can be used to run it. Since the image chosen is described by a circuit that does not exceed the 5 qubit limit of the available quantum devices, the *device_test* function can be applied by selecting the number of shots. Its output will give the measured states and the number of times they have occurred during the test.

```
counts=testing.device_test(qc, numShots)
```

Output:

```
{'010': 252, '001': 982, '101': 79, '110': 772,
'100': 733, '011': 93, '000': 273, '111': 912}
```

Retrieval

The results of the measurements on FRQI, NEQR and QPIE circuits can be elaborated by their specific decoding functions, to re-obtain the pixel intensities ordered by position. Even when using the compressed version of the encoding method, if it is supported, there is no decompression to apply. The states embedded on the compressed FRQI circuit are the same as the standard version. Because of this, the decoding function from the *frqi_basic* module can be used. This function outputs an array with the pixels intensities ordered by position in the image. The $PSNR$ function uses the original array and the retrieve one to print on the terminal the Peak Signal to Noise ratio of the experiment.

```
px_array_r=frqi.frqi_decode(counts, numShots)
print(px_array_r)

PSNR(px_array,px_array_r)
```

Output:

```
[131 113 138 152]

PSNR = 12.276324453507348  dB
```

From the results, it is possible to evaluate the impact of noise on the FRQI encoding. The resulting array can then be reshaped into a matrix, by defining its dimensions, to visualize the retrieved image.

```
img_matrix= px_array_r.reshape(n_side, n_side)
plt.imshow(img_matrix, cmap='winter', vmin=0, vmax=255)
```

Output:

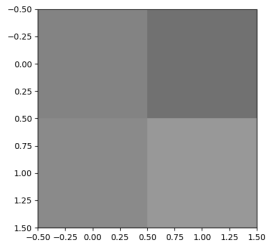


Figure 3.3. Image retrieved from testing FRQI circuit on the IBM quantum device Manila

3.2.2 Example 2: Processing two images

The second example is conceived with the idea of showing how to use the implemented functions that process two images together, like the *neqr_ADD_SUB*(), the *neqr_comparator*(), and the *neqr_sort*() functions, as well as demonstrating the processing capabilities of the *neqr_chromatic_transformation.py* module. To give a practical application scenario of the usage of one of this modules, the *neqr_sort*() function is used to process of two images in order to identify an object depicted in both, but with different lightning. The images taken into consideration are in Figure 3.2.2.

Figure 3.4. *'shadow_1.png'*

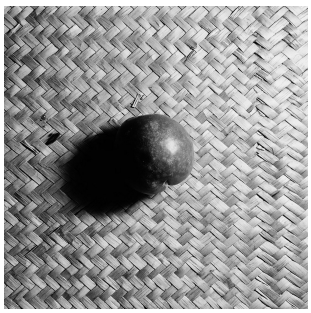


Figure 3.5. *'shadow_2.png'*

In order to keep the circuit under the dimension constraint of 30 qubits for the ideal simulation, the two images are scaled to their 8×8 version and their intensity range is reduced to $[0, 2^3]$.

Encoding

As seen in the previous example, the two images files, now referred to as *shadow_1.png* and *shadow_2.png* have to be opened and the characteristic of the images have to be checked. The *image_conv* function is used to open both files.

```
px_array_1=image_conv('shadow_1.png')
px_array_2=image_conv('shadow_2.png')
```

Output:

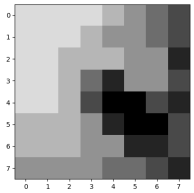


Figure 3.6. *'shadow_1.png'* 8×8 version, $q = 3$

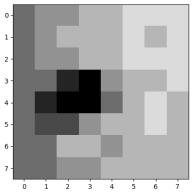


Figure 3.7. *'shadow_2.png'* 8×8 version, $q = 3$

Figure 3.8. Output after opening the two images

The standard encoding function is then used to embed the two images in two different quantum circuits, with $q = 3$.

```
q=3
qc_1=neqr.neqr_circuit(px_array_1, q)
qc_2=neqr.neqr_circuit(px_array_2, q)
```

Output:

```
Circuit Depth :  235
Operations:  OrderedDict([('x', 384), ('mcx_gray', 108), ('barrier', 65),
('h', 6)])

Circuit Depth :  218
Operations:  OrderedDict([('x', 384), ('mcx_gray', 91), ('barrier', 65),
('h', 6)])
```

To investigate the complexity of the implemented circuit, the *transpile* function is applied to both.

```
transpile(qc_1)
transpile(qc_2)
```

Output:

```
Depth of the circuit after unrolling:  33884
Operations after unrolling:  OrderedDict([('u1', 20412), ('cx', 20304),
('u3', 384), ('u2', 222), ('barrier', 65)])

Depth of the circuit after unrolling:  28611
Operations after unrolling:  OrderedDict([('u1', 17199), ('cx', 17108),
('u3', 384), ('u2', 188), ('barrier', 65)])
```

To process the two images together, their circuits must be merged together. This can be done by exploiting a few of the Qiskit tools. The *QuantumRegister* function is used to create a register capable of containing the second circuit, which is then added to the first circuit. Then the compose method is called to combine them, as shown in the example below. To use the *neqr_comparator* and the *neqr_ADD_SUB* function, the same procedure has to be applied. The *num_qubits* method is used to extrapolate the number of qubits in a circuit and save it into a variable.

```
n_qubits=qc_2.num_qubits
qc_2_reg=QuantumRegister(n_qubits)
qc_1.add_register(qc_2_reg)
qc_1=qc_1.compose(qc_2, range(n_qubits, n_qubits*2))
```

Processing

In order to sort the intensities expressed in the NEQR circuits, the sort module is applied. Its implementation allows to flexibly sort any two sequence of qubits in a circuit. It requires as inputs two lists containing the position of the two sequences in the circuit. The last parameter represents an index of the sort module being implemented, which is useful when applying more than one to the same circuit.

```
qc_TOT=neqr_sort(qc_TOT, range(0, q), range(qc_2.num_qubits,
qc_2.num_qubits + q),  0)
```

To facilitate the detection of the target, the intensity information of the brighter image is inverted. The *neqr_compl_color_* is used to apply the color complement operation on the intensity qubits of the circuit containing the brighter image, by specifying its LSB qubit on the circuit.

```
qc_TOT=neqr_color_compl(qc_TOT, 0, q)
```

Testing

Using modules that handle two images together, whether it is through the sort module, the comparator or the adder, adds several auxiliary qubits to the circuit that are are not useful in decoding process. Measurement function are supported in the library to speed-up the process of retrieving the information from this type of circuits. In this case, *neqr_SORT_measure* is used to apply measurement to the circuit only on the necessary qubits. The *ideal_simulation* is run, by using its corresponding function. Its output corresponds to the measured states and the number of times they have been measured.

```
qc_TOT=neqr_measure_SORT(qc_TOT, int(math.log((num_pix),2)), q)
counts=qimp.ideal_simulation(qc_TOT, numShots)
```

Output:

```
{'100000110111110010': 26, '000001110010010011': 23, .....
'110110001000110001': 41, '101001101011100011': 29}
```

Retrieval

Measuring a two-image circuit produces results that contain the position information and the intensity information for both images. The results are only valid when the position information is the same. The *neqr_SORT_decode* function is used to decode these measurements and find the actual sorted intensities. Functions that decode the results from testing the adder module and the comparator module are also provided in the library. In order to decode the information, the function requires the number of pixels in the image and the resolution of the intensity as well. By printing the resulting array, the concatenated pixel information are displayed.

```
sort_imgs=neqr_SORT_decode(counts, num_pix, q)
print(sort_imgs)
```

Output:

```
[3 4 4 5 5 .... 5 5 5 5]
```

The two images can now be visualized by splitting the array and re-shaping the results as shown in **??**. On the figure on the left, the brighter image resulting from the sort operation, whose intensities have been inverted, is displayed. On the one on the right, the darker one is shown. As expected, the brighter image contains a shape of higher intensity that corresponds to the target.

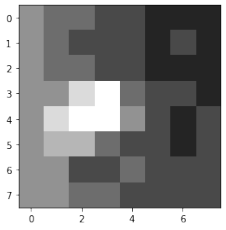


Figure 3.9. Brighter image with inverted intensities

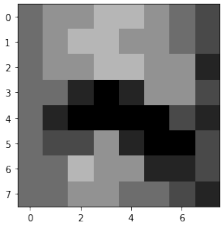


Figure 3.10. Darker image

3.2.3 Example 3: Edge Detection

The third example aims at showing the reader how to perform edge detection on an image, using the functions supported by the library. To demonstrate this, an image file referred to as '*cat.png*', shown in Figure 3.11 will be processed.

Figure 3.11. Image '*cat.png*'

The functions used are taken from the *qpie_software_library* package, more specifically from the *qpie_basic.py*, *qpie_geom_transformation.py* and *qhed.py* modules. To keep a contained circuit dimension, the image is scaled to its 32×32 version, which corresponds to 10 qubits in the QPIE encoding.

Encoding

As seen in the previous examples, the image file *cat.png* is opened and stored into an array.

```
px_array=image_conv('cat.png')
```

Output:

In order to embed it with the QPIE method, the standard encoding function is used to describe two different quantum circuits, one for the horizontal scan and one for the vertical scan.

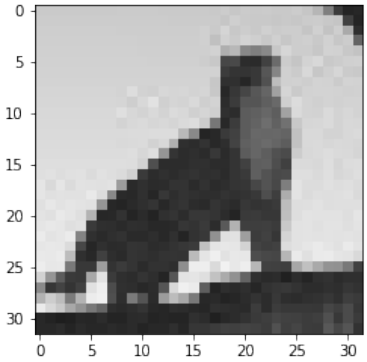


Figure 3.12. Output of calling *im_conv* Image *'cat.png'* 32×32 version

```
qc_h=qpie.qpie_circuit(px_array)
```

Output:

```
Circuit Depth :  1
Operations:  OrderedDict([('initialize', 1)])
```

In order to apply edge detection both in the horizontal and vertical direction of the image, the algorithm has to be applied on an image whose matrix is the transposed of the original. The obtained array is therefore rearranged. The encoding functions, print out the depth and operation count of each circuit. Because QPIE corresponds to a state preparation of a circuit, its encoding function consists into computing the probabilities corresponding to the intensities and then using a method provided by Qiskit, *initialize*, to apply the transformation on the circuit. For this reason the depth of the circuit, without transpiling, is 1.

```
px_mx=px_array.reshape(n,n)
px_array_T= (np.asarray(px_mx.T)).flatten()
qc_v=qpie.qpie_circuit(px_array_T)
```

```
Circuit Depth :  1
Operations:  OrderedDict([('initialize', 1)])
```

The actual complexity of the circuit is evaluated by using the *transpile*() function.

```
transpile(qc_h)
transpile(qc_v)
```

Output:

```
Depth of the circuit after unrolling:  2037
Operations after unrolling:  OrderedDict([('u3', 1023), ('cx', 1022),
('reset', 10)])

Depth of the circuit after unrolling:  2037
Operations after unrolling:  OrderedDict([('u3', 1023), ('cx', 1022),
('reset', 10)])
```

Processing

Processing may be applied on the image before its features are extracted. For example, this image might require to be shifted up, in order to position the target in the center of the frame. To exemplify this, the *qpie_pos_shift*() is applied on both circuits, but along opposite axis, given the transposition between the two images. The function requires as inputs the axis along which the shift is applied, the direction and the magnitude of the shift. The image is shifted along the Y-axis, in the up direction and of 2^3 pixels. The transposed image is shifted along the X-axis, of the same magnitude in the left direction.

```
c=3
qc_h= qpie_pos_shift(qc_h, 'y', 1, c)
qc_v= qpie_pos_shift(qc_v, 'x', 1, c)
```

The QHED algorithm can now be applied on both circuits, by simply calling the *qhed* function.

```
qc_h=qpie.qhed(qc_h)
qc_v=qpie.qhed(qc_v)
```

Before testing the circuits, measurement has to be performed on both.

```
qc_h.measure_all()
qc_v.measure_all()
```

Testing

Both of the circuits are then run separately, since the information they are elaborating is independent. The *ideal_simulation* function is used and the measurements results are printed on the terminal.

```
counts_h=qimp.ideal_simulation(qc_h, numShots)
counts_v=qimp.ideal_simulation(qc_v, numShots)
```

Output:

```
{'01001010001': 1, '10011010011': 1, '10101011111': 1, ...
'1110001100': 57, '11011110110': 86, '11001001000': 137}

{'01111100001': 1, '01000010111': 1, '11111100111': 1, ...
'10100001100': 79, '10101000111': 54, '00110100101': 47}
```

Retrival

Only the measurements in which the auxiliary qubit is equal to 1 correspond to the gradients of the pixels. In order to filter out the results and order them by position, the function *qhed_decode* is used for both simulations. Its output contains the information on the borders of the image extracted in the two directions. By summing the two array together, the edge detection algorithm is completed.

```
edges_h=qhed_decode(counts_h, px_array.size, numShots)
edges_v=qhed_decode(counts_v, px_array.size, numShots)
edges=edges_h+edges_v
```

The resulting array is then reshaped into a matrix. The brightness of the edges is configurable by changing the maximum intensity value of the resulting image.

```
max_gradient=5
edges_mx=edges.reshape(N,N)
plt.imshow(edges_mx, cmap='gray', vmin=0, vmax=max_gradient)
```

Output:

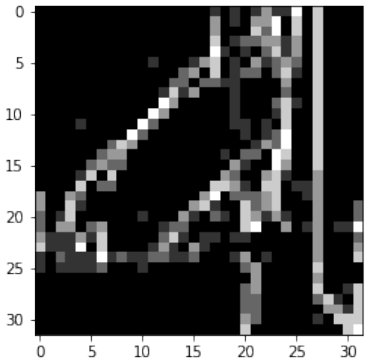


Figure 3.13. Result of applying the QHED algorithm

As expected, the results shows that areas with high contrast of intensities are reported on the obtained image, with a certain degree of approximation.

Chapter 4

Tests and simulations

Once the development of the QImP software library was complete, the implemented algorithms were tested using Qiskit's simulators and devices to verify their correct behaviour and characterize them.

The aim of this chapter is to present the characteristics of the chosen tools used to carry out the tests, explaining their limitations and the reason behind their usage. After that, the methodology followed to carry out the various tests in order to obtain meaningful results will be explained. The last section is dedicated present the experimental results of the conducted test and draws conclusions on them.

4.1 Set Up

In this section the quantum devices and simulators used are presented, followed by an explanation on how they were considered to address the characterization of the algorithms.

4.1.1 IBM quantum devices

The tests on quantum devices have been conducted on one of the free accessible quantum computers provided by IBM, Quito. This device is part of the processor family Falcon, specifically the Falcon r4T design version, which has 5 qubits [24]. The coupling map of this type of processor, i.e. a graph representing how the qubits are connected to each other, is displayed in Figure 4.1.

As can be seen from the schematic, there is limited connectivity between the qubits, which constitutes a constraint on which couples of qubits can be considered when implementing two-qubit gates. That defines a discrepancy between the virtual circuit created from the description done through Qiskit and the actual circuit that can be implemented on the device. In order to solve this, SWAP gates are added

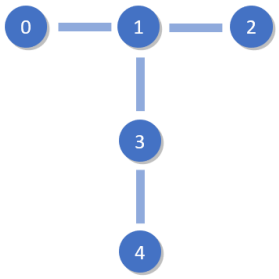


Figure 4.1. Coupling map of the Falcon processor type, version r4T

to the circuit to compensate the lack of connectivity, but increasing the depth of the overall circuit. Furthermore, the performance of this connection coupling maps is sometimes affected to change and not all the connected couples can be effectively used. Because of the fact there is a limited number of free accessible devices available on the IBM Cloud service, their usage is assigned to the users through a queuing system that often implicates having to wait a considerable time before being able to run a test. Other three Falcon devices are available on the Cloud, but the device Quito was chosen because, in the days in which the tests were conducted, it guaranteed the maximum qubit connectivity, allowing all the techniques to be tested on a constant setting.

4.1.2 Ideal and noisy simulators

The main tool used to perform validation tests on the algorithms was Qiskit's Aer backend. Qiskit's Aer contains several backends that implement a variety of simulation methods. Its main one, the Aer Simulator, gives the possibility to run simulations that mimic the execution of an actual device without noise, taking into account its behaviour only from a functional point of view. When its default settings are considered, the simulator returns a *count* dictionary containing the final values of the measurements and the number of times each result has occurred. Moreover while it also provides a number of methods that return, for example, the unitary describing the circuit or its state vector and other ways to represent it. To test the algorithms supported by the library the default configuration was evaluated to be the most appropriate, as the goal was to understand if the images retrieved from the measurements were as expected.
To test the circuit adding a noise model to the simulation, Aer allows the creation of backends starting from actual noise data retrieved from real devices. The data sets available come from IBM quantum computers with different architectures, allowing the simulation of circuits of dimensions that exceed the computational capacities of the free accessible devices. The automatic models generated by the simulator

are an approximation of the real errors that occur on actual devices because they are built from a limited set of input parameters describing the non-ideal behaviour of qubits. Nevertheless, testing circuits on this platforms can overall give a good estimation of the robustness of the algorithms.

4.1.3 Methodology

The main goal of this thesis is to quantitatively and qualitatively compare the different encoding methods and their processing algorithms on real quantum devices, to evaluate their application perspectives on real world scenarios. As previously stated, the limited availability and computational capabilities of accessible quantum hardware, made it necessary to exploit other available tools to ensure that the tests on real devices were as meaningful as possible and to characterize the results obtained by applying the same algorithms on larger images. The methodology followed can therefore be synthesized into three steps:

- First, validation tests were performed to verify the correct implementation of the algorithms, through ideal simulations
- Secondly, functional tests were run to find the optimal number of shots for each encoding method that guaranteed the maximum PSNR, as well as to analyze the depth of the implemented circuits
- At last, tests were conducted on the real devices to make comparisons between the techniques and to see the effect of processing on them

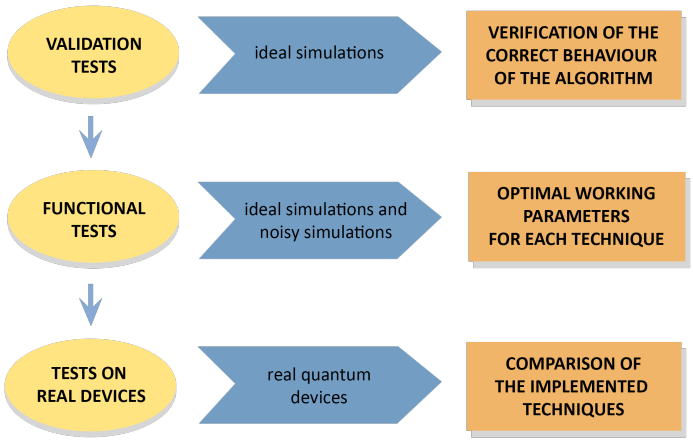


Figure 4.2. Methodology of the tests

4.2 Experimental results

The experimental results presented in this section follow the structure of the methodology presented:

- For what concerns the validation tests, the graphical representation of applying the algorithms to a sample image, depicted in Figure 4.3, will be shown in comparison to the original input image.
- The results of the functional tests will be presented for both the ideal and noisy simulations, grouped in tables that show the varying of the figures of merit previously discussed, while changing the parameters of the simulation and the image.
- The results on real hardware will be shown in both graphical and table form.

The sample image used for experiments is depicted in Figure 4.3.

Figure 4.3. Original sample image

4.2.1 Validation tests

Considering the dimension of the circuits required by the various algorithms varies considerately depending on the specific technique applied, the sample image has been reduced to its 8×8 pixels version. Furthermore, for what concerns the most resource-demanding NEQR algorithms, the intensity range was also reduced to be able to process images of the same size as for the other techniques, but still complying with the 30 qubits limit of the Aer Simulator. The result of these tests are divided in categories that include the encoding methods and the processing type, and they are always shown next to their input image. All of the tests have been run on the Aer backend, executing optimal number of simulations derived with a particular methodology that will be presented in the next section.

FRQI

In this category, it is possible to observe the results of embedding an image with the three techniques supported in the library that are based on the FRQI formalism: standard FRQI, compressed FRQI and MCRQI. In section 4.2.1, the standard FRQI encoding is tested through an ideal simulation and is retrieved with a PSNR=37.65 dB. It is worth noticing that the results obtained from the measurements are then rounded to the nearest integer, in order to interpret them as intensity levels.

Figure 4.4. Original sample and the results obtained from an FRQI encoding

In Figure 4.5, the compressed version of the FRQI encoding gives the same result as the standard version, effectively implementing a lossless compression, with a PSNR=36.91 dB.

Figure 4.5. Original sample and the results obtained from the compressed version of an FRQI encoding

Embedding the RGB version of the sample on an MCRQI circuit, an image with a PSNR=27.12 dB with respect to the original is retrieved. As can be seen from Figure 4.6, the slight differences between the intensities are more visible on an RGB image.

Figure 4.6. Original sample and the results obtained from an MCRQI encoding

NEQR

In this category, the results of embedding an image using one of the three techniques based on the NEQR formalism are shown.

By using the standard NEQR encoding to embed the sample, the retrieved image, depicted in section 4.2.1, has a PSNR=inf dB with respect to the original, due to basis embedding of the technique.

Figure 4.7. Original sample and the results obtained from a NEQR encoding

When considering the compressed version of NEQR though, it modifies the information contained in the sample. The compression applied on the image, heavily depends on how much redundant information there is and where it is stored. As seen in 2.5.2 the minimization of the boolean functions, can create duplicate states for the same pixel, actively modifying its intensity. The PSNR derived from the result of this test are shown in Figure 4.8 is 12.23 dB.

Figure 4.8. Original sample and the results obtained from the compressed version of an NEQR encoding

To embed the RBG version of the sample on a NCQI circuit, while respecting the limitations on the number of qubits, it was necessary to reduce the intensity range of each color channel to $[0,2^3]$. The resulting image has a PSNR=inf dB as NEQR, given the embedding strategy it uses. The image is depicted in Figure 4.9.

Figure 4.9. Original sample and the results obtained from a NCQI encoding

QPIE

Simulating the QPIE circuit encoding the image, the retrieved image has a PSNR=21.30 dB and is shown in Figure 4.10. As for FRQI, the resulting image is an approximation of the original.

4.2.2 Processing algorithms

In this category, the behaviour of the processing algorithms is verified. The PSNR of these techniques is calculated by confronting them with the results of the same processing applied through classical computing on the original image.

Figure 4.10. Original sample and the results obtained from a QPIE encoding

Geometric transformations

The results of tests regarding the geometric transformations functions are shown only for the FRQI encoding. The other versions have identical visual results, with PSNR slightly changing depending on the encoding method's retrieval accuracy.

In Figure 4.11, the image on the right represents the result of applying an axis flip with respect to the X-axis. Its PSNR is of 37.21 dB.

Figure 4.11. Original sample and the results obtained from applying a F_Y operation on a FRQI image

The results of applying a coordinate swap operation is represented in 4.12. The function effectively implements the desired effect, obtaining a PSNR=35.96 dB.

Figure 4.12. Original sample and the results obtained from applying a coordinate swap operation on a FRQI image

In Figure 4.13, it is possible to observe the effect of applying the orthogonal rotation function, with $\theta = 270°$, to an FRQI image. The retrieved image has a PSNR of 36.31 dB.

Figure 4.13. Original sample and the results obtained from applying a orthogonal rotation operation on a FRQI image

In Figure 4.14, the result of a restricted flip operation on the upper half of the image, with respect to the X-axis, is shown. The processed sub-area of the image is highlighted in red, while the PSNR is 35.67 dB.

Figure 4.14. Original sample and the results obtained from applying a restricted F_Y operation on a FRQI image

Applying a coordinate swap operation on the upper right half of the image, produces the result shown in Figure 4.15, as expected. Its PSNR with respect to classical processing is of 34.03 dB.

Figure 4.15. Original sample and the results obtained from applying a restricted coordinate swap operation on a FRQI image

4.2.3 Chromatic transformations

In this category, only the results of applying chromatic transformations to NEQR will be presented, as it is the encoding from which the algorithms of most interest.

In Figure 4.16 the color complement operation is shown. The inversion of the gray-scale information is correctly achieved.

Figure 4.16. Original sample and the results obtained from applying a color complement operation on a NEQR image

If Figure 4.17, it is possible to observe the classification operation applied with a threshold of 128. All of the pixels with higher intensity are displayed in black, while the others are white.

Figure 4.17. Original sample and the results obtained from applying a classification operation on a NEQR image

In order to demonstrate the *add/sub* function, the sample image's intensities were subtracted with the intensities of a copy. Black pixels are equivalent to the intensity level 0, demonstrating the correct behaviour of the algorithm in Figure 4.18.

Figure 4.18. Original sample and the results obtained from applying a SUB operation on two NEQR images

In Figure 4.19 the original sample and its color complemented version are depicted to illustrate the correct functioning of the sort module, which is to compare the two images pixel by pixel and to sort them by intensity value, putting all the brightest in the first image and vice-versa. Instead, on Figure 4.20 it is possible to observe that the two resulting images' pixels are sorted by intensity level.

Figure 4.19. Original sample and color complement version

Figure 4.20. Resulting images from applying a SORT operation on two NEQR images

Quantum Edge Detection

In Figure 4.21, the effect of applying the QHED algorithm on the sample image, as implemented in the library, is depicted. The image selected, in its 8×8 gray-scale version, does not have a considerable differences in terms of intensity between the object and the landscape. Therefore extracted edges of the image do not describe the target accurately.

Figure 4.21. Original sample and the results obtained from applying QHED algorithm on a QPIE images

4.2.4 Functional tests

After the validation tests, the goal was shifted onto characterizing the different techniques in terms of complexity and fidelity by varying the dimension of the images and the number of shots of the simulations. For this purpose, the noise models provided by Qiskit were used as a tool to predict the behaviour of the algorithms in real quantum hardware applications, along with ideal simulations. It is worth noticing that, the many noise models present in Qiskit's toolbox have consistently different behaviours, due to their different data sets, and the results of simulating the implemented algorithms may vary in a non-negligible way when switching between them. In the following section, the Aer backened simulator was used to run ideal simulations, while the FakeMelbourne backend was considered for the noisy simulations. Its noise model is based on data taken from the performances of the IBM quantum computer Melbourne, which has an architecture of 14 qubits. The dimensions of the embedded images used in the tests vary from 2×2 images to 16×16, requiring to reduce the number of intensity levels in order to simulate NEQR images. The number of intensity levels was reduced to 2^3 for all three encoding methods. The tests are organized in the following way:

- First the encoding methods were tested with ideal simulation, in order to understand which are their characteristics in absence of non-idealities, varying the number of shots and the dimensions of the circuit, by changing the size of the image.
- After that, the same process is repeated on the noisy backend, to investigate what are the limitations it introduces.
- The results obtained from processing algorithms are also presented, for a limited number of techniques. The techniques were chosen based on which processing algorithms were also testable on the limited capacities of the available real devices and their PSNR is calculated with respect to the ideal simulations.
- Finally, the same techniques, are tested on the device Quito.

In the reported tables, the main figures of merit describing the QImP algorithms and their performances, are presented for each circuit. In the first experiments , the number of simulation shots are varied to see how the PSNR responds, while the depth and the number of gates are observed in contrast to the image size. In the noisy simulations, the same process is repeated, while observing how the PSNR changes in relation to the depth of the circuit. The number of simulations N_{shots} is given by the formula:

$$N_{shots} = N_{pixels} \times c$$

where N_{pixels} is the number of pixels in the image, and c is a factor varied through the experiments.

FRQI ideal simulation

Here, the results of ideal simulations performed on a FRQI circuit are shown in three tables, in which c varies from 10 to 1000. From these experiments it is possible to see that $c = 1000$ corresponds to the number of measurements that guarantees the highest possible PSNR for FRQI. Switching from the minimum to the maximum number of shots, there is an improvement of more than 20 dB. Test with $c > 1000$ were also run, but the PSNR achievable by the FRQI encoding peaks at about 40 dBs. Increasing c coefficient does not bring additional advantages with respect to the ones shown for $c = 1000$.
The second aspect worth evaluating is the actual increase of depth and gate count when varying the size of the image. The depth and number of gates shown in all of the following tables is extracted from the circuit after the transpiling, which unrolls the circuit in X gates, CX gates and U gates, which are single qubit rotations. The increase of required resources correlated to larger image's sizes is more than what was expected, making the circuits for images of modest size extremely complex.

FRQI tests with c=10, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	91	138	40	14.72 dB
4×4	5	1427	1972	160	13.34 dB
8×8	7	24131	32584	640	16.17 dB
16×16	9	391427	524042	2560	16.65 dB

FRQI tests with c=100, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	91	138	400	23.72 dB
4×4	5	1427	1972	1600	26.85 dB
8×8	7	24131	32584	6400	27.81 dB
16×16	9	391427	524042	25600	29.26 dB

FRQI tests with c=1000, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	91	138	4000	29.73 dB
4×4	5	1427	1972	16000	41.96 dB
8×8	7	24131	32584	64000	38.34 dB
16×16	9	391427	524042	256000	42.10 dB

NEQR ideal simulation

For what concerns ideal simulations on NEQR, only the table corresponding to $c = 10$ is shown, as the PSNR maximum value is already reached. As presented in subsection 2.2.2, in the absence of non-idealities, NEQR is capable of retrieving the exact image. The depth and number of gates observed instead do not match the expected results. In fact, the depth and gate count obtained from transpiling the circuit in CX gates and one-qubit rotations exceed the ones required for FRQI, although the NEQR circuit is reduced to its 3-intensity-qubit version. Slightly different basis of gates were given as inputs for the transpiler, but the result stayed the same.

NEQR tests with c=10, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	5	88	135	40	inf dB
4×4	7	1962	2693	160	inf dB
8×8	9	30117	36839	640	inf dB
16×16	11	461348	279605	2560	inf dB

QPIE ideal simulation

As can be seen from the following tables, QPIE has by far the lowest complexity of all the encoding methods, in line with the theoretical results. As for FRQI, QPIE reaches its highest PSNR value when $c = 1000$, but in this case the PSNR peaks at around 20 dB. It is worth noticing that the QPIE retrieval depends heavily on an estimation of the Root Mean Squared (RMS) value of the pixels intensities, since that information is not embedded in the circuit. By knowing the exact value of the RMS, further tests conducted show that, in an ideal simulation for a QPIE circuit embedding this image, the peak PSNR can arrive at around 27 dB. An image whose RMS value is close to the median one estimated in the retrieval process, will have better results in terms of PSNR. That underlines the fact that QPIE's results can vary a lot depending on the information contained by the image.

QPIE tests with c=10, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	2	5	7	40	7.92 dB
4×4	4	27	33	160	15.39 dB
8×8	6	121	131	640	15.16 dB
16×16	8	503	517	2560	17.21 dB

QPIE tests with c=100, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	2	5	7	400	8.66 dB dB
4×4	4	27	33	1600	18.66 dB
8×8	6	121	131	6400	20.50 dB
16×16	8	503	517	25600	21.21 dB

QPIE tests with c=1000, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	2	5	7	400	8.71 dB
4×4	4	27	33	1600	18.86 dB
8×8	6	121	131	6400	21.37 dB
16×16	8	503	517	25600	22.31 dB

FRQI noisy simulation

When switching to noisy simulations, it is possible to observe that the results of all the encoding methods worsen considerately. Depth and gate count of the circuits have an important impact on the PSNR results, as the parameters describing the errors of gates are introduced, the circuit accumulates error as the critical path of the circuit increases. Even in the noisy simulation, $c = 1000$ represented the optimal working point for the performances of FRQI.

FRQI tests with c=10, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	91	138	40	10.16 dB
4×4	5	1427	1972	160	10.09 dB
8×8	7	24131	32584	640	8.55 dB
16×16	9	391427	524042	2560	8.77 dB

FRQI tests with c=100, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	91	138	400	17.42 dB
4×4	5	1427	1972	1600	10.58 dB
8×8	7	24131	32584	6400	10.49 dB

16×16	9	391427	524042	25600	9.32 dB

FRQI tests with c=1000, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	91	138	4000	21.99 dB
4×4	5	1427	1972	16000	10.30 dB
8×8	7	24131	32584	64000	10.88 dB
16×16	9	391427	524042	256000	10.80 dB

NEQR noisy simulation

The tests on noisy simulations have shown that the encoding method most affected by noise is the NEQR method. With the addition of gate non-idealities, all of the possible 2^{2n} combinations of the basis states are generated, and the results show that the states with the highest probability can be different from the originally encoded states. Over many tests, it was observed that this behaviour is completely aleatory, making NEQR a less robust. The decoding function implemented for NEQR only elaborates the states with the highest probability, which can lead to retrieved information that can be redundant for one pixel and completely absent for another. When the retrieved states are the correct ones, the PSNR reaches the maximum PSNR level, while it can drop significantly when that does not occur. In this case, the number of measurement performed as a impact on the NEQR results, especially for 2×2 images, showing once again the importance of lower depth and gate count.

NEQR tests with c=10, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	5	88	135	40	3.83 dB
4×4	7	1962	2693	160	2.16 dB dB
8×8	9	30117	36839	640	4.15 dB
16×16	11	461348	279605	2560	2.03 dB

NEQR tests with c=100, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	5	88	135	40	inf dB
4×4	7	1962	2693	160	3.30 dB
8×8	9	30117	36839	640	4.02 dB dB

16×16	11	461348	279605	2560	3.04 dB dB

NEQR tests with c=1000, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	5	88	135	40	inf dB
4×4	7	1962	2693	160	10.30 dB
8×8	9	30117	36839	640	4.6 dB dB
16×16	11	461348	279605	2560	4.35 dB

QPIE noisy simulation

The QPIE encoding is the most robust to the effect of noise. The results obtained show PSNR values that are close to the noiseless simulation, for what concerns the 2×2 images, confirming once again the inverse proportionality between the complexity of the circuit and the fidelity of the images retrieved.

QPIE tests with c=10, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	2	5	7	40	8.32 dB
4×4	4	27	33	160	12.72 dB
8×8	6	121	131	640	13.20 dB
16×16	8	503	517	2560	10.15 dB

QPIE tests with c=100, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	2	5	7	400	8.67 dB
4×4	4	27	33	1600	16.22 dB
8×8	6	121	131	6400	15.02 dB
16×16	8	503	517	25600	9.80 dB

QPIE tests with c=1000, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	2	5	7	400	8.66 dB
4×4	4	27	33	1600	17.57 dB
8×8	6	121	131	6400	14.70 dB

16×16	8	503	517	25600	10 dB

Noisy simulation on the Flip Operation

The testing of processing algorithms on FRQI was done through the application of the flip operation along the Y-axis (F_Y). The simulation was repeated for a number of times that was deemed optimal from the previous experiments done on the encoding method, since the flip operation adds a complexity to the circuit of O(1). The results are in line with what was observed from simply testing the encoding method, showing similar PSNR values.

F_Y on FRQI tests with c=1000, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	92	140	4000	18.77 dB
4×4	5	1429	1973	16000	9.70 dB
8×8	7	24134	32586	64000	10.02 dB
16×16	9	391431	524044	256000	9.10 dB

Noisy simulation on the Complement Color technique

For what concerns NEQR, the processing method applied was the color complement technique (CC), which is also a very low complexity algorithm. In the same way as for NEQR, the exact measurement is retrieved when the image is sufficiently small, while in other cases the PSNR is very low, due to the randomness of the retrieved states.

Color complement operation on NEQR tests with c=1000, noisy simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	5	89	138	4000	inf dB
4×4	7	1962	2693	16000	4.54 dB
8×8	9	30117	36839	64000	2.80 dB
16×16	11	461348	279605	256000	3.27 dB

QHED noisy simulation

The QHED simulations showed results consistent to what already observed for QPIE. The added qubit required by the algorithm did not introduce a drop in performances with respect to the simple encoding method.

QHED tests with c=1000, ideal simulation					
Img size	**number of qubits**	**Depth**	**Number of gates**	**Number of shots**	**PSNR**
2×2	3	8	10	4000	9.80 dB
4×4	5	29	34	16000	10. dB
8×8	7	123	132	64000	9.02 dB
16×16	9	503	517	256000	7.97 dB

4.3 Tests on real hardware

As a last step of the testing procedure, the algorithms presented were run on real quantum hardware. The tests were initially performed using the optimal number of shots identified in the previous steps, and later they were varied to verify their accuracy. The number of simulations shown on the tables are the ones that were found to be the optimal ones through various tests on the hardware.

4.3.1 Encoding methods

In this first table, the encoding techniques are closely compared for the same 2×2 image, shown in Figure 4.22, which has a gray-scale range equal to $[0, 2^3 - 1]$. The testing was performed in this way to allow the representation of NEQR on the available hardware, in line with the previous tests performed.

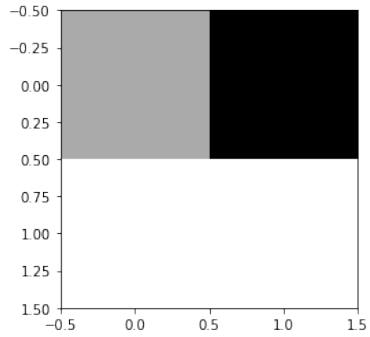


Figure 4.22. 2×2 sample with q=3

Given the redundancy of the sample image chosen, the compression version of the FRQI and NEQR method were also tested. To quantify the amount of compression applied, the Compression Ratio (represented as **CR** in the tables) is also specified. It is defined as

$$Compression\_Ratio = (1 - \frac{Ops\_After\_Compression}{Ops\_Before\_Compression}) \times 100\%$$

This metric varies considerately depending on the embedded image. For the other encoding methods, the corresponding table cell is indicated as Non Classified (NC).

Comparison of embedding techniques on hardware						
Method	N_{qubits}	**Depth**	N_{gates}	N_{shots}	**CR** (%)	**PSNR**
FRQI	3	92	144	4000	NC	6.32 dB
FRQI comp	3	53	76	4000	47.22	9.13 dB
NEQR	5	79	126	400	NC.	6.02dB
NEQR comp	5	6	12	400	90.14	16.90dB
QPIE	2	6	9	4000	NC	10.23dB

From the results obtained, it is possible to observe that the results of FRQI on the Quito device have a substantial drop of performances in terms of PSNR. As it is easily observable from the table, FRQI is by far the encoding that requires the highest depth, followed by NEQR. For both of this encoding methods the compressed versions introduce improvement. As can be seen in Figure 4.3.1, the image retrieved from the compressed version constitutes a recognizable version of the image with respect to the image retrieved from the standard version, which is quantified by an PSNR increase of about 3 dB, due to a $CR = 47.22\%$.

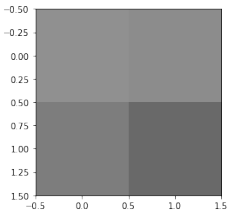


Figure 4.23. FRQI standard

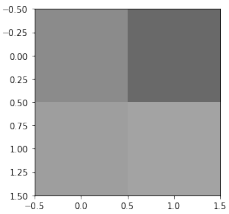


Figure 4.24. FRQI compressed

Figure 4.25. FRQI tested on the IBM Quito device

For what concerns NEQR, the effect of noise is easily observable in Figure 4.3.1 and 4.3.1. Both images have the intensity of one of the pixels of a value unrelated to the original image. In the compressed case, that value is closer to the original one, raising the PSNR from 6.02 dB to 16.90 dB, with $CR = 90.12\%$. These results do not improve by using the maximum number of shots available on the device, which is 8192. It is important to notice that the results obtained from NEQR, vary a lot. Multiple measurements on the Quito device have shown that, depending on the state of the hardware, the NEQR can also give satisfying results with PSNR=inf dB with a certain frequency.

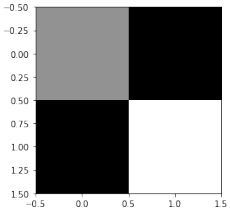


Figure 4.26. NEQR standard

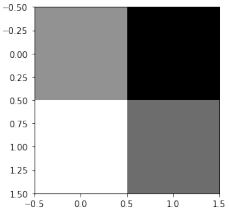


Figure 4.27. NEQR compressed

Figure 4.28. NEQR tested on the IBM Quito device

The image retrieved from testing the QPIE encoding method is depicted in Figure 4.29. By looking at the table 4.3.1 is possible to observe that the result is one of the most encouraging between the other encoding methods. Nevertheless, the results are slightly below the expectations given the very low computational complexity of the algorithm.

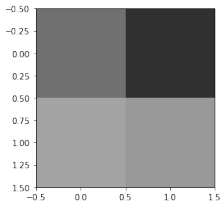


Figure 4.29. QPIE encoding tested on quantum hardware

4.3.2 Processing algorithms

The techniques previously tested on the noisy simulation, were also tested on the hardware. Because all of these processing techniques require the addition of a few simple gates, the results were expected to slightly worsen. The respective PSNR of each resulting image has been calculated with respect to the data obtained from

running the same circuit on an ideal simulation. The corresponding image is also shown next to the one of each device test.

Comparison of processing techniques on hardware					
Processing technique	N_{qubits}	**Depth**	N_{gates}	N_{shots}	**PSNR**
F_Y FRQI	3	92	142	40	8.06 dB
F_Y FRQI comp	3	54	81	4000	9.44 dB
CC	5	80	128	400	13.38 dB
CC comp	5	7	15	400	inf dB
$QHED$	3	19	31	4000	9.26 dB

For what concerns FRQI, by looking at the table **??**, it is possible to observe that the PSNR has slightly improved, but the overall result is line with the expectations. Figures 4.3.2 and 4.3.2 show the results of applying a F_Y operation on the FRQI circuit, where it is noticeable that the compressed circuit delivers an image more recognizable, given its increase of 1.44 dB in terms of PSNR.

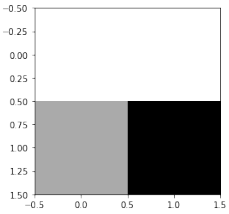


Figure 4.30. Ideal simulation

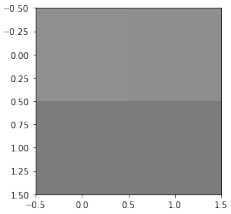


Figure 4.31. FRQI standard

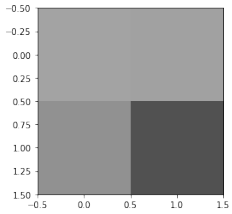


Figure 4.32. FRQI compressed

Figure 4.33. F_Y tested on the IBM Quito device

The result obtained from testing the complement color operation on the circuit show the previously mentioned characteristics of retrieving images embedded with NEQR. For the non-compressed image, pixels intensities are very unrelated to their original value. The compressed circuit instead, gives the exact result, bringing the PSNR to maximum. That is in contrast with the expectations, since the added noise coming from the extra gates was expected to worsen the performance. These underlines the aleatory aspect of retrieving an NEQR image in the presence of non-idealities.

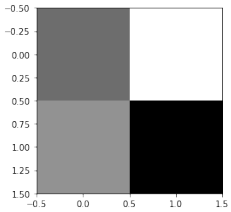


Figure 4.34. Result on ideal simulation on NEQR

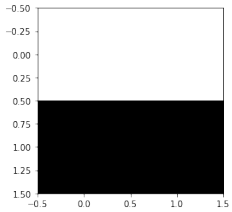


Figure 4.35. NEQR standard

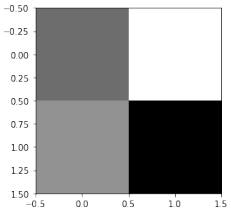


Figure 4.36. NEQR compressed

Figure 4.37. *CC* tested on the IBM Quito device

As a last test, the QHED was also run on quantum hardware. By comparing looking at 4.3.2, it is possible to observe that the edge point of the image is correctly retrieved. Comparing the image with the one in Figure 4.3.2, a PSNR=9.26 dB was observed, in line with the previous results of QPIE.

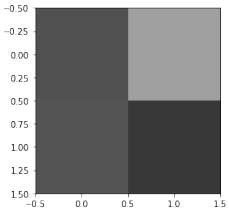


Figure 4.38. QHED tested through ideal simulation

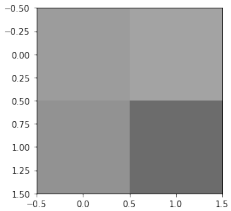


Figure 4.39. QHED tested on IBM Quito device

Chapter 5

Conclusion

The goal of this thesis is to provide a practical way to compare the most promising QImP techniques proposed in the state-of-the-art and to evaluate whether they represent concrete opportunities with respect to their classical counterparts, while considering the limitations of quantum computers available today. For this purpose, a modular Python software library was developed, capable of applying many QImP algorithms. It gives the possibility to flexibly implement encoding methods, geometrical and chromatic transformations, edge detection algorithms and compression techniques, while allowing the characterisation of the different algorithms through a set of test functions. The library makes use of Qiskit, an open-source software-development kit that permits to describe quantum circuits and to run them either on simulators or on real quantum devices, through IBM Cloud quantum computing service and it has been extensively documented considering Jupyter Notebooks.

Once the library development was completed, the implemented techniques were tested using both simulations and real quantum hardware, while particular figures of merit were considered to make quantitative comparisons of the results obtained.

The results observed during the tests highlight the fact that, although the processing algorithms implemented do not contribute to the complexity of the circuit in a substantial way, the depth of the circuits required by NEQR and FRQI only for embedding the information, make the retrieved images non reliable and therefore not suitable for further processing. Future perspectives for these methods should necessarily focus on compressing the information for any kind of image, given how critical depth is to the reliability of the circuit. In this regard, the results obtained by QPIE underline the importance of expanding its processing capabilities both for its low complexity and its suitability to algorithms oriented to computer vision.

In conclusion, the implemented library provides a useful tool for comparing various techniques from the QImP panorama. The analysis and characterisation of

the algorithms, that can be done exploiting the developed functions, represents a starting point to a larger comprehension of practical perspectives of this field of research and the feasibility of its algorithms, which are often not discussed in the available literature.

The modular and flexible nature of the implemented library gives the possibility to enrich it with new techniques in the future and to integrate the ones that are present into more advanced quantum computing algorithms, broadening their initial application into more complex processing tasks. The tests performed on real quantum hardware highlighted the actual advantages and disadvantages of the different encoding methods that were not observable from the theory and simulations.

Bibliography

[1] Salvador E. Vengas-Andraca Fei Yan and Kaoru Hirota. Toward implementing efficient image processing algorithms on quantum computers. 2021.

[2] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum Computing for Scientists*. Cambridge University Press, 2008.

[3] Fangyan Dong Phuc Q. Le and Kaoru Hirota. A flexible representation of quantum images for polynomial preparation, image compression, and processing operations. 2010.

[4] Bo Sun, Phuc Q. Le, Abdullah M. Iliyasu, J. Adrian Garcia Fei Yan, Fangyan Dong, and Kaoru Hirota. Multi-channel representation for images on quantum computers using the rgb α color space. 2018.

[5] Yinghui Gao Yi Zhang, Kai Lu and Mo Wang. Neqr: a novel enhanced quantum representation of digital images. 2013.

[6] Shen Wang Jianzhi Sang and Qiong Li. A novel quantum representation of color digital images. 2016.

[7] Zeyang Liao Xi-Wei Yao, Hengyan Wang and Ming-Cheng Chenr. Quantum image processing and its application to edge detection: Theory and experiment. 2018.

[8] Vivek V. Shende and Igor L. Markov. Quantum circuits for incompletely specified two-qubit operators. 2018.

[9] Sahel Ashhab. Quantum state preparation protocol for encoding classical data into the amplitudes of a quantum information processing register's wave function. 2022.

[10] Fangyan Dongz Phuc Q. Le, Abdullahi M. Iliyasuy and Kaoru Hirotax. Fast geometric transformations on quantum images. 2010.

[11] Fangyan Dong Abdullah M. Iliyasu, Phuc Q. Le and Kaoru Hirota. Watermarking and authentication of quantum images based on restricted geometric transformations. 2011.

[12] Fangyan Dong Phuc Q. Le, Abdullahi M. Iliyasu and Kaoru Hirota. Strategies for designing geometric transformations on quantum images. 2011.

[13] Fangyan Dong Phuc Q. Le, Abdullahi M. Iliyasu and Kaoru Hirota. Efficient color transformations on quantum images. 2010.

[14] Salvador E. Vengas-Andraca Fei Yan. *Quantum Image Processing*. Springer, 2020.

[15] Kai Xu · Yinghui Gao Yi Zhang, Kai Lu and Richard Wilson. Local feature point extraction for quantum images. 2014.

[16] C. McMullen R.K. Brayton, A. Sangiovanni-Vincentelli and G. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[17] Xiande Liu Panchi Li and Hong Xiao. Quantum image median filtering in the spatial domain. 2018.

[18] Nan Jiang Jian Wang and Luo Wang. Quantum image translation. 2014.

[19] Prabh Simran Baweja Alok Anand, Meizhong Lyu and Vinay Patil. Quantum image processing. 2022.

[20] Lu Kai Zhang Yi and Gao YingHui. Qsobel: A novel quantum image edge extraction algorithm. 2014.

[21] Hengyan Wang Xi-Wei Yao and Zeyang Liao. Quantum image processing and its application to edge detection: Theory and experiment. 2018.

[22] Quantum edge detection - qhed algorithm on small and large images. `https://qiskit.org/textbook/ch-applications/quantum-edge-detection.html`.

[23] Qiskit. https://qiskit.org/documentation/.

[24] Ibm quantum. https://quantum-computing.ibm.com/.