UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRAU D'ENGINYERIA INFORMÀTICA

Treball final de grau

READING QR CODES ON CHALLENGING SURFACES

Autor: David Martinez Carpena

Director: Ismael Benito Altamirano i Dr. Juan Daniel Prades García

Realitzat a: Departament d'Enginyeria Electrònica i Biomèdica

Barcelona, June 21, 2020

Abstract

QR Codes are a common way of representing information in a machine-readable format, thanks to its fast reading speed and reliability. The increasing use of this technology has led to one remarkable limitation: the standard algorithms for reading QR Codes assume that the surface where it lies is flat. Therefore, QR Codes that lie non-flat surfaces suffer from slow reading speed and high rate of unsuccessful decoding with the standard algorithms.

In this work, we will compare four different correction methods for QR Codes found in challenging surfaces. First, we will implement the method described in the standard of the QR Codes. Secondly, we will study the correction method used in the majority of the commercial readers. Thirdly, we will explore the state of the art for the correction of cylindrical deformations, because it is a common occurrence with some existing solutions. Finally, we will propose using a well-known surface interpolation spline named Thin Plate Spline to target arbitrary deformations. To make this comparison, we will implement a modular library for decoding QR Codes. Using this library we will be able to compare and test the results achieved by each correction method. For making the tests, we will create three different datasets, each with labelled images of QR Codes with different deformations.

Resum

Els codis QR són una tecnologia molt popular per representar informació en un format llegible per ordinadors, gràcies a la seva alta velocitat de lectura i resistència a errors. El creixement en l'ús d'aquestes tecnologies està començant a destacar una de les seves limitacions: l'algoritme estàndard per a la descodificació d'un codi QR assumeix que hi ha una superfície plana sota seu. Com a conseqüència d'aquest fet, els codis QR sobre superfícies adverses presenten un seguit de dificultats com baixes velocitats de lectura i alta probabilitat d'errors de descodificació.

En aquest treball compararem quatre mètodes de correcció de codis QR a superfícies adverses. Primer, considerarem el mètode descrit a l'estàndard dels codis QR. Segon, estudiarem el mètode de correcció més utilitzat pels lectors comercials. Tercer, explorarem els avenços més recents en correcció de deformacions cilíndriques, ja que és un cas molt comú i amb algunes solucions existents. Finalment, proposarem utilitzar un spline interpolador de superfícies anomenat Thin Plate Spline per a abordar superfícies arbitràries. Per fer aquesta comparació necessitarem implementar la nostra pròpia llibreria modular per a la descodificació de codis QR. Utilitzant aquesta llibreria, serem capaços de comparar els resultats aconseguits per cada mètode de correcció. Per computar aquests resultats, construirem tres diferents datasets, amb imatges de codis QR etiquetades segons el tipus de deformació que presentin.

Contents

Chapter 1

Introduction

Nowadays, great part of our daily communications are done using written messages, thanks to our optical character recognition (OCR) abilities as humans. But when we try to replicate this behaviour in machines it gets difficult. Automatic OCR is its own research area, with impressive results in the recent years, but likewise a very complex and expensive process. One of the solutions that tried to solve this problem were barcodes. The barcodes were invented in the 50s as a visual representation of Morse code, and become a way of representing data in a visual and machine-readable format. In the 70s, the barcodes were popularized to automate supermarket checkout systems.

With the barcodes as a successful standard, multiple modifications began to appear and tried to improve in the readability and storage capacity. One of these trends was generating two-dimensional barcodes, also named matrix barcodes. A normal barcode was usually read using specific hardware with a sensor that scanned a line through the code. In the case of 2D barcodes, the code cannot be read using a single line, because it spans the information in the two dimensions of the image. The majority 2D barcodes were designed to be detected and read using computer vision algorithms, instead of specific hardware, sacrificing a part of their greater storage capacity. In this environment the first QR Codes appeared.

The QR Codes are a type of 2D barcode, created by Denso Wave in the decade of the 90s and internationally standardized in the 2000. The standard has been updated in two occasions, with the current standard from 2015. The QR Codes are a freely licensed technology except for the use of the name which is a trademark from Denso Wave. This type of codes were created to track vehicles during manufacturing in an automated way.

The great compromise in QR Codes for fast and easy decoding and great error correction have made them very popular. The usage of this technology has been increasing in the last decades, which has led to more diverse applications and the highlighting of some of its limitations. One of these limitations is the surface where the QR Code lies. The standard assumes that all QR Codes lie in a flat surface and the majority of the reading software also implement this limitation.

Image that a company makes some kind of bottle or cylindrical container, and want to put QR Codes in the bottles as a tracing technology. There is no flat surface in a bottle except for the cap, which normally is too small to contain a QR Code with tracing information. This simple case can lead to the company putting the QR Code in the surface of bottle and breaking the assumption done by majority of QR Code readers. Other more complex examples could be a bag of chips, which can have different types of surfaces depending on how is deformed. The error correction can recover from many errors in these cases, but sacrificing the original purpose of this mechanism. If we expect every QR Code of a product to need the error correction for a normal read, when a photo is taken with bad lightning conditions the error correction will be exhausted. Also, there are cases where the limitations of data lengths and size of the QR Code force to use lower levels of error correction, which could make QR Codes on non-flat surfaces nearly unreadable.

In the recent years, some research has appeared tacking this problem. Beginning in 2013, some papers proposed a solution for the particular case of QR Codes in cylindrical surfaces [Li+13][LWW15][LZ17][Li+19]. The proposed solutions introduce algorithms that work like the standard ones in photos of flat QR codes while being capable of correcting the cylindrical deformation when appears.

Unfortunately, we haven't found any works that address the case of arbitrary deformations. After thinking about the problem, we had an idea for a possible solution: a method named Thin Plate Splines, which is based in a non-linear two-dimensional interpolation with very good approximation properties. The surfaces approximated by this method are very similar to the random surfaces where we would expect a QR Code to lie on.

The primary purpose of this project is to test how all these methods compare against each other. Will the standard method of decoding QR Codes be able to read QR Codes in challenging surfaces? And the cylindrical methods? Finally, how our new proposal will perform in flat and non-flat images from QR Codes?

Memory structure

This work is structured in four main chapters. The first chapter will introduce the problem that we want to tackle and define the goals of the project. The second chapter will set the theoretical foundations for all the concepts used through the project: from the definition and structure of the QR Codes to the mathematical explanation of all the correction methods that we will implement. The third chapter is devoted to the methodology followed during the development of the project and the implementation. Finally, the fourth chapter will describe which experiments have been done and which results can be extracted from them.

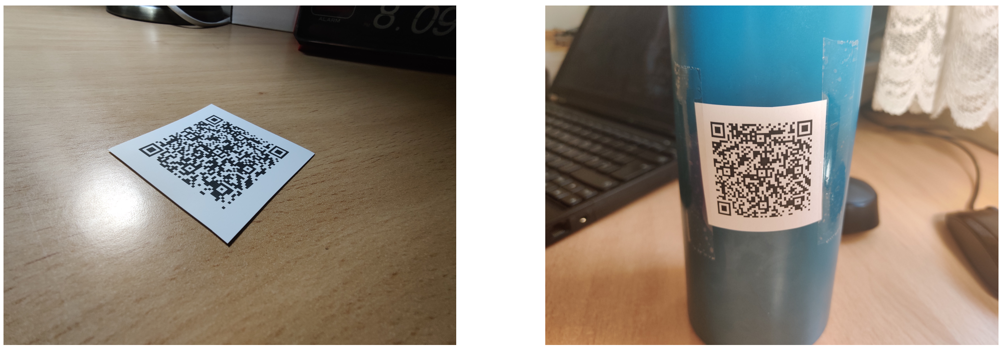
Chapter 2

Objectives

In this first chapter we will define the goals of this project. We will begin by detailing the main goal, which corresponds to solving the problem stated in the introduction. Based on the main goal, we will construct a list of secondary goals, that can help us by restricting our final solution.

2.1 Main goal

The main goal of all this work is to compare different correction methods for QR Codes in challenging surfaces, from existing ones to a new one that we are proposing. To make this comparison we will need to implement all the compared methods. Since the standard of the QR Codes assumes that they will always be laying on a flat surface, any non-flat surface pose a challenge on the common decoding methods and do not work as expected. In a photo taken of a QR Code, we should find two types of deformation: the intrinsic deformation from the surface where the code is over and the perspective deformation caused by the view from the camera.



(a) Perspective deformation from camera

(b) Intrinsic deformation from the surface

Figure 2.1: Types of deformations found in a photo of a QR Code

In this project we will explore different methods to overcome this problem. We will change the geometric correction proposed in the standard by some different ones, trying to revert some deformations. After an initial research, we decided to settle in with fourth different methods of correction:

1. **Affine (AFF):** The simplest and weakest correction that should only work in images with the QR Code perfectly flat. This method allows us to make a first very trivial implementation and should serve as a lower bound to all the others. Using this type of correction is equivalent to the algorithm proposed at the standard.

2. **Projective (PRO):** The most common method of geometric correction for QR Code decoding. The results obtained with this method should be very similar to the ones obtained by a commercial reader. It should be capable of correcting all the deformations generated by the perspective views from a camera, but not the deformations from the surface.

3. **Cylindrical (CYL):** A new approach presented in several papers [Li+13][LWW15] [LZ17][Li+19] in the recent years. Based on the assumption that the deformation has a cylindrical shape, because is a very common case in the real world. The method tries to use the analytical definition of the surface to predict the deformation.

4. **Thin Plate Spline (TPS):** Our new proposal as a correction method. The TPS is a data interpolation and smoothing technique, used in many fields. We decided to use it as a surface-agnostic correction method, thanks to its versatility and non-linear properties. This method give us a similar transformation to CYL, but without any previous assumption on the type of deformation, thanks to the complex interpolation used.

Based on our initial research, we can hypothesize that the four methods that we want to compare will have different behaviours depending on the deformation of the image. In an image with a QR Code in a flat surface, we expect the affine and projective corrections to make the best approximations. In the cases where we have images with QR Codes in cylindrical-like surfaces, the cylindrical correction should achieve the best results. Finally, in all the other random surfaces, the TPS should give the best approximation.

2.2 Secondary goals

Using the main goal as foundation, we want to have a more accurate definition of our objectives. We will choose more secondary goals that will help to constrain and define the type of work we want to deliver.

Custom QR Code decoding library

Our main goal is to replace the geometric correction used in the decoding of a QR Code. Ideally, we would like to use third party code for the rest of process of decoding (location, sampling, decoding, etc.). But all the third party QR Code readers have end to end implementations, where they only give information about the data inside the QR Code, with sometimes other minor extras like a bounding box.

For advanced methods like the cylindrical or TPS corrections we will need a lot more location information than the given by these third party readers. The only alternative is to build a custom and modular library for QR Code decoding, which would let us localize all the necessary features for the different methods.

Although we need this new library, we can still use third party libraries for some steps where they give us what we need. For example, the last step of decoding is reading the QR Code grid of white and black pixels found and corrected in previous steps. This step is normally called decoding, and is a deterministic process. The output of this process is whether the code can read and the data inside, which the third party readers give us. For cases similar to this one, we will try to use third party libraries inside our library.

Extensible implementation

In the design of the software project, we want to create a extensible library that could evolve if we want to do further work in this area. We will apply good software design practises and methodologies, that can help us to manage the code as it grows. To try to avoid overgeneralizing, we will define early the desired public API of our implementation, to ensure we don't lose the track of the coding goals.

Comparable metrics

If we want to do a formal comparison, we will need a set of defined metrics, and our software needs to be capable of measuring them. To correspond to this need, the secondary goal of creating a set of metric and their implementation have to be considered. There are very promising metrics based on the number of wrong pixels obtained, but we will need to recover the full sampled QR Code to compute it. Another valuable metric is whether we can read or not the QR Code.

Variety in the data-sets

Finally, there are a lot of possibilities for building the datasets used to test our hypothesis. To try to maximize the validity of our results, we will create two type of datasets. The first type will have images of QR Codes generated by software and augmented using some data augmentation library, to try to introduce certain deformations. This type has the benefit of being procedural and automatic, we can generate as many images as we can with constant effort. The second type will have photos made to printed QR Codes, which is a lot more difficult to prepare and label, but is a better reflection of the general use case. With this combination, we will have the benefits of both and the ability of detect if we are over-fitting to any of the two types.

Chapter 3

Theoretical foundations

In this chapter we will introduce all the concepts and theory behind the QR Codes and the four methods of correction that we want to compare. We will begin with an introduction to the formal definition of QR Codes, followed by the methods and concepts used for localization. Finally, we will go into details with the mathematics behind each one of the geometric corrections used.

3.1 QR Codes

A QR Code (Quick Response code) is a two-dimensional barcode defined by the international standard ISO/IEC 18004 [ISO15]. The first version of this standard appeared in the year 2000 [ISO00], and was updated in 2005 [ISO06] and 2015 [ISO15], with some minor improvements. In these standards, 3 types of QR Codes are defined:

- **QR Code Model 1:** First design of the QR Code, very similar to the most common one, except for the alignment patterns.
- **QR Code Model 2:** Originally proposed in the first standard, extended in the second by the name QR code 2005 and renamed to just QR Code in the last standard. This is the most used design in the QR Code family and what the majority of people will identify. Improves the computer vision properties of the Model 1 thanks to the addition of alignment patterns. The majority of the current QR Code readers expect this type of design.
- **Micro QR Code:** Alternative design that uses only one finder pattern and no alignment patterns. This implementation tries to cover the use case of very small printing targets, at the expense of worse computer vision helper patterns and less capacity for data.

We will use only the QR Code Model 2 as target, because is the most widely used variant of the standard. From now on, we will refer to a QR Code Model 2 as QR Code.

Breaking down the design, a QR Code can be defined as a square grid of modules. A module is the name given to the "pixels" of a QR Code, which can only be black or white. The number of modules per side of the square grid depends on a integer number from 1 to 40, which is named version. Given a version, we can compute the number of modules per side with the formula:

$$sideLenght = 4 \cdot version + 17$$

In the grid of modules of a QR Codes each module can be classified in one of three disjoint categories based on its purpose:

- **Function patterns:** The section was created only as a helper to the computer vision algorithms. Thanks to it, a reader algorithm can easily find, correct and sample the QR Code, assuming some kind conditions. This section is composed by 3 different types of patterns: finder, alignment and timing.
- **Encoding metadata:** The first modules read by a decoder after successfully recovering the full grid from the image. This section is composed of two encoding areas: format and version. By reading them, the decoder configures some important parameters like the level of error correction or the XOR mask used in the content region. To mitigate some errors, a (15, 5) BCH code error correction is used. The version area only appears in QR Codes of version greater or equal to 7.
- **Content:** The modules containing the data encoded by the QR Code together with the error correction codes (ECC) for this data. The data is encoded in a format that defines several subsections, specifying the encoding, length, etc. The ratio between data and ECC depends on the level of error correction specified in the encoding metadata section. Finally, all the content region is XORed using a certain pattern to decrease the probability of likelihood to a function pattern.

The figure 3.1 presents the geometric realization of the three different sections explained above, using a QR Code of version 7 as an example. The axis of the figure are the number of modules per side, starting by the top left corner. The content section is the biggest one of the three, followed by the function patters. This two sections create a balance between computer vision helpers and it's storage capacity.

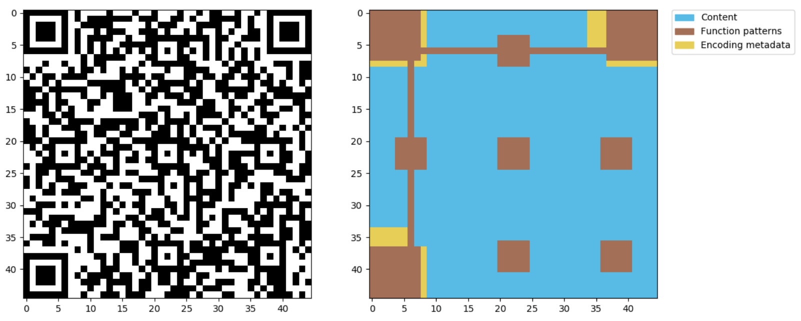


Figure 3.1: High level structure of a QR Code version 7.

The general use of a QR Code begins with the encoding, which is the name of the process of generating a QR Code given the data, error correction and version. After encoding and printing, the next step is reading its contents using a camera and the appropriate software. The process of taking an image of a code and returning its contents can be divided in several steps:

- **Localization:** The first step accepts an image as input and scans it searching for one or more valid QR Codes. A variety of standard computer vision techniques are used for this purpose. The rest of the pipeline will be executed for each QR Code found in this step.
- **Geometric correction:** In an image of a QR Code we can find two types of deformations: the generated when taking a photo and the intrinsic deformation of the surface where the code was printed or placed. In this step we will try to transform the input to a new corrected image that ideally will not have any deformation.
- **Grid sampling:** With a corrected image, the next objective is getting back the square grid of only black or white modules. This steps basically binarizes and subsamples the corrected image to obtain the grid.
- **Decoding:** The last step is the proper decoding of the grid, using the inverse process of the encoding. This process is completely deterministic and doesn't involve any computer vision.

As we explained in the previous chapter, there is no public implementation of the localization step that will give us all the information that we need at the geometric correction step. For this reason, we will explain also the standard approach for localization and feature detection of QR Codes. After that section, we will explain the different geometric correction methods.

For the grid sampling step we will use a trivial down-sampling without interpolation and anti-aliasing. Thanks to the previous step of correction, this simple transformation is enough to obtain a very good QR Code grid. Finally, we will use a public available QR decoder, with the resulting grid as input, to skip the unnecessary work of writing our own decoder for the last step.

3.2 Localization and feature detection

In this section we will introduce the methods used to localize a QR Code and extract all the necessary features for the next steps. Before explaining the method in particular, we need to explain in details what are all the types of function patterns, because them will be key to our localization. As we can see in the figure 3.2, a QR Code has three types of function patterns: finders, alignments and timings.

The finder patterns are the regions which identify the position and orientation of a QR Code. There are three finder patterns placed in three corners, in particular in the top left, top right and bottom left ones. This type of patterns are created to be very easy to find and unique, thanks to their large and structured design, that contrast with the highly

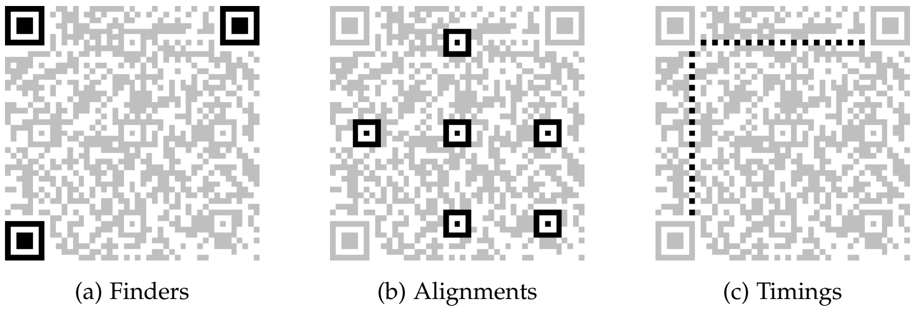


Figure 3.2: The three different function patterns

random patterns inside the QR Code. Although the finder patterns determine a position and orientation, if the image is mirrored, like when we take a photo of a reflection of a QR Code in a mirror, there is no feature in that can be used to identify and correct this diagonal symmetry. As a consequence, in the decoding step the QR Code will be read as is, and if it fails, it will be read applying a diagonal symmetry to its contents and try another read.

The structure of the finder patterns is designed to be easy to find in an image using a simple algorithm about ratios of sequences of pixels in a line. Suppose that we trace a line with any slope on a binarized image which passes over a position pattern crossing the center point. If we count the successive white or black pixels in the line over the position patter, we will see that the count follows a ratio 1 : 1 : 3 : 1 : 1, thanks to the design of the finder pattern. We can see a visual representation of this property in the figure 3.3 This fact enables us to use a very simple method for finding all the finder patterns in an image. We can trace horizontal lines over the image searching for a candidate center that follow the expected ratio. When we find a candidate, we will throw a vertical line over that center, check if it follows the expected ratio and correct the vertical offset. If the candidate follows the vertical and horizontal ratios, we can check some diagonal ratios, to be sure. This method is the one defined in the standard.

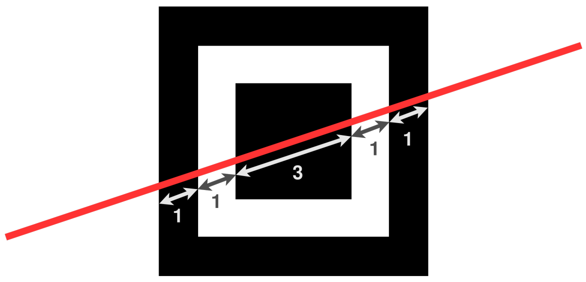


Figure 3.3: Graphical representation of the search for finder patterns using ratios

When searching for QR Codes in an image, first we search all the finder patterns in the image by the method just explained. The next step is to group the finder patterns in groups of three, in a way that we choose which will be our candidate QR Codes in the image. This grouping can be done in multiple ways, a very simple and trivial approach can be doing a K-means clustering algorithm where K is the remainder of the number of finders found by 3.

The next type of function pattern is the alignment patterns. These patterns are complementary to the finder patterns, and their function is to be used as references by the geometric correction step. They span in a grid over the surface of the QR Code, with more alignment pattern the greater the version is. In the shape side, the alignment patterns are smaller than the finders and easier to mistake, because they need to cover the least possible space, and they are not essential to the reading. About the algorithms, we can use again the recommended ones from the standard. The alignments also have a ratios based property, with a ratio of $1:1:1$. The rest of the algorithm is the same that in the case of the finders.

The last type of pattern of this section is the timing patterns. This is another complementary pattern, that is not essential but any QR Code reader could use if they think worth it. The pattern consist in a row between the top finders and a column between the top left and bottom left finder. This row and column are covered using a pattern of alternating black and white modules. The standard defines the possible uses of this pattern can be version guessing or sampling. In the deformed cases that we want to study, these rows and columns don't follow a straight line, and their use is more complex. For this reason, we will not use the timings patterns in our feature extractor.

Given that we will not use the timing patterns, we need to find another way to do the version guessing. For this purpose, we will use a method based used on the papers [LZ17][Li+19], based on the geometrical cross ratio between the corners of two finder patterns. This method assumes that at least one side of the QR Code is close to be straight, but can perform well enough for cases with some deformation. With cases with heavy deformation in all sides this method will probably fail. Another advantage of this method is that only uses the corners of the finders patterns, which are features that we know how to find.

Before explaining the methods, we need a new mathematical definition. The cross ratio $(ABCD)$ of four collinear points A, B, C, D is defined by the formula

$$(ABCD) = \frac{dist(A,C) \cdot dist(B,D)}{dist(A,D) \cdot dist(B,C)}$$

where *dist* denotes the function that gives the distance with the standard metric in the plane. Given two lines r and r', with four points each $A, B, C, D \in r$ and $A', B', C', D' \in r'$, if r' is the image of r under a projective transformation, with A', B', C', D' being the images of A, B, C, D, then the cross ratios of the points at each line are equal:

$$(ABCD) = (A'B'C'D')$$

The exact definition of a projective transformation will be given later in the next chapter, but the only thing we need to understand this definition is that is a transformation that sends one line to another line. In this case, the transformation will map one of the sides of the QR Code in the image to one side of the original grid of the QR Code.

Let A and B be the top left and bottom left corners of the top left finder pattern and C and D be the top left and bottom left corners of the bottom left finder pattern, and A', B', C', D' the respective corners of the original QR Code.

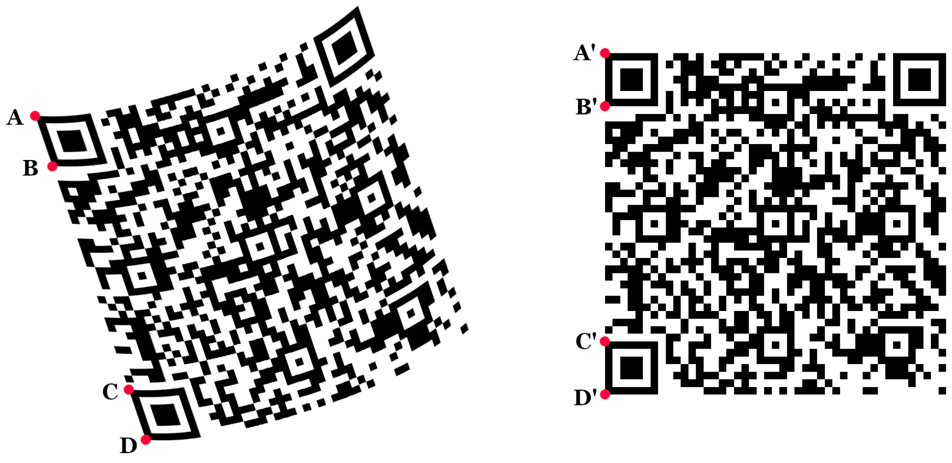


Figure 3.4: Cross ratio method for extracting the version, figure from [Li+19]

Suppose that the QR code has version v, a side length of $s = 17 + 4v$ modules, and that we define the number of modules between finder patterns as $n = s - 14$. In this case, we have:

$$dist(A', C') = n + 7, \quad dist(B', D') = n + 7, \quad dist(A, D) = n + 14, \quad dist(B, C) = n$$

which implies

$$(ABCD) = (A'B'C'D') = \frac{(n+7)^2}{(n+14)n} = \frac{n^2 + 14n + 7^2}{n^2 + 14n} = 1 + \frac{7^2}{n^2 + 14n}$$

We define $\lambda = (ABCD)$ and compute it using the distances between the corners of the finder patterns in the deformed image. In this case, we have:

$$\lambda = 1 + \frac{7^2}{n^2 + 14n} \iff (n+14)n = n^2 + 14n = \frac{7^2}{\lambda - 1}$$

and if we replace in the first expression:

$$\lambda = \frac{(n+7)^2 \cdot \lambda - 1}{7^2} \iff (n+7)^2 = 7^2 \frac{\lambda}{\lambda - 1} \iff n = 7 \cdot \left(\sqrt{\frac{\lambda}{\lambda - 1}} - 1 \right)$$

Once we know n, we can recover the version easily using the formula:

$$v = round\left(\frac{s - 17}{4}\right) = round\left(\frac{n + 14 - 17}{4}\right) = round\left(\frac{n - 3}{4}\right)$$

By all the previous methods, we have localization algorithms for the finders, the alignments and the version of the QR Code, which are all the necessary features for the correction methods that we want to implement.

3.3 Geometric corrections

After finding all the target features in an image and grouping the ones that we consider a QR Code, we need to apply some type of spatial correction to recover its original flat shape.

Before we start explaining each type of correction, we need to define what an image correction is. Beginning with notation, we will use $\mathbb{N}_n$ to denote the set of natural numbers from 0 to n. We can define an image of size (n, m) as a mapping

$$img : \mathbb{N}_n \times \mathbb{N}_m \to [0,1]^3$$

where the points of $\mathbb{N}_n \times \mathbb{N}_m$ are the coordinates of the pixels and the points at $[0,1]^3$ are the RGB coordinates of each pixel. We are assuming all images are RGB ones, but we could generalize to any other number of channels.

An image correction will be a transformation from a deformed image img' of size (n', m') to the original image img of size (n, m). The inputs to any image correction will be the deformed image img' and two k-tuples of points $src \in (\mathbb{N}_{n'} \times \mathbb{N}_{m'})^k$ and $dst \in (\mathbb{N}_n \times \mathbb{N}_m)^k$ that we will call references. The references represent the information obtained from the feature detection step, and we will assume that if we take the i elements of src and dst, they correspond to a point in the deformed image and one in the original one that match. How many and which references will work better depends on the specific correction function used.

Giving that we want to correct a deformed image, we have two ways to go: forward mapping or inverse mapping. A forward mapping is a mapping from coordinates of the deformed image to coordinates of the corrected one. This is the most natural way of thinking about image transformations. On the other hand, an inverse mapping is a mapping from the coordinates of the corrected image to the deformed one.

This two methods will give us the same output in an image correction. If we use forward mapping, we will iterate all the pixels in the deformed image, create their respective images using the mapping, and use this output to interpolate the remaining pixels and dropping the ones that fall outside the destiny size. In the case of inverse mapping, we will iterate by all the pixels of the corrected image to obtain their correspondent pixel of the deformed image, interpolating and extrapolating when falling between pixels or outside the limits of the deformed image.

Although the resulting image will be the same, the inverse mapping method is much more efficient. We are not interested in where all the pixels of the deformed image will land, because will be clip out of the image if they fall outside. Instead, in the inverse mapping technique we only compute the values for every pixel of the corrected image. For this reason, we will use inverse mapping for all the correction methods, with the respective interpolation and extrapolation rules.

Using this definition, we can make our problem a mathematical statement. Suppose that we have a deformed image img' of size (n', m'), and we want to correct it to obtain the original image img of size (n, m). We can define an inverse mapping correction F from the deformed image img' and the original img as a composition of an injection

$i : \mathbb{N}_n \times \mathbb{N}_m \to \mathbb{R}^2$ and a function $f : \mathbb{R}^2 \to \mathbb{R}^2$. The function f is the correction itself:

$$F : \mathbb{N}_n \times \mathbb{N}_m \xrightarrow{i} \mathbb{R}^2 \xrightarrow{f} \mathbb{R}^2$$

To compute the resulting transformation, we just need to apply the inverse mapping correction F to each pixel of original image img. The resulting numbers are points of $\mathbb{R}^2$, but we have the deformed image in the space $\mathbb{N}_{n'} \times \mathbb{N}_{m'}$. To solve this problem, we use interpolation and extrapolation. For each point $(x, y) \in \mathbb{N}_n \times \mathbb{N}_m$, we get their value $F(x, y) \in \mathbb{R}^2$.

- If $F(x, y)$ is outside the square $\mathbb{N}_{n'} \times \mathbb{N}_{m'}$, we use extrapolation to compute a value.
- If $F(x, y)$ is exactly a point in $\mathbb{N}_{n'} \times \mathbb{N}_{m'}$, we take the value at this pixel of the deformed image.
- If $F(x, y)$ is between some points in $\mathbb{N}_{n'} \times \mathbb{N}_{m'}$, we use interpolation to compute the value, based on the nearest pixels of the deformed image.

We can use several strategies for extrapolation and interpolation. For extrapolation, we will mainly use an edge strategy, which assigns to every pixel outside the image space the value of the nearest pixel inside the image. This method of extrapolation is good enough for all our cases, and helps to avoid non continuity generated with other like the constant color strategy. For interpolation, we will use the best strategy for each method of correction. The most common options in image transformations are: nearest-neighbor, bi-linear, bi-quadratic, bi-cubic, bi-quartic and bi-quintic.

With all this settled, we only need to decide how to create the inverse mapping using the references. In this section we will introduce four methods. The first two are linear methods: affine and perspective transformation. The third method is a non-linear analytical method, based on the assumption that the deformation of a surface is cylindrical. Finally, the Thin Plate Splines non-linear method is presented, as a general deformation solution because doesn't need any assumptions about the deformation. For each method, will we introduce briefly the theory behind it and introduce the algorithm used to determine the corresponding f transformation.

3.3.1 Affine

An affine transformation is a function between two affine spaces which acts linearly on vectors between points. All affine transformation can be decomposed in a linear transformation and a translation. The standard example of an affine space is the real plane $\mathbb{R}^2$. As a consequence of its definition, we can derive some geometric properties for this type of functions:

- Preserves collinearity between points.
- Preserves the simple ratios between collinear points.
- Preserves the relation of parallel lines.

Suppose we have a affine transformation $f : \mathbb{R}^2 \to \mathbb{R}^2$, we can decompose it in a linear transformation $h : \mathbb{R}^2 \to \mathbb{R}^2$ and a translation from the origin O to a point B. Because h is a linear two-dimensional transformation, given a linear basis of $\mathbb{R}^2$ exists a 2×2 matrix H, such that $h(\vec{p}) = H \cdot \vec{p}$. Using this information, we can define the affine transformation in a certain affine basis by the equation:

$$f(P) = H \cdot (P - O) + B$$

This equation can be transformed into a matricial expression, but we need to change the way we represent points. Suppose that we have a point $P = (p_1, p_2)$ and that their image has coordinates $f(P) = (q_1, q_2)$, we will write it using the notation

$$P = \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

appending a 1 at the third coordinate of every point. Given the point of translation B with affine coordinates (b_1, b_2) and a matrix H with linear coordinates

$$H = \begin{pmatrix} h_{1,1} & h_{1,2} \\ h_{2,1} & h_{2,2} \end{pmatrix},$$

we can represent f by the matrix

$$A = \left(\begin{array}{cc|c} h_{1,1} & h_{1,2} & b_1 \\ h_{2,1} & h_{2,2} & b_2 \\ \hline 0 & 0 & 1 \end{array}\right)$$

This matrix uniquely determines the affine transformation, and offers a simple way to compute it using matrix multiplication:

$$\left(\begin{array}{cc|c} h_{1,1} & h_{1,2} & b_1 \\ h_{2,1} & h_{2,2} & b_2 \\ \hline 0 & 0 & 1 \end{array}\right) \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix}$$

Using the previous definitions we can enunciate a theorem tells us how many images of points we need to determine an affine transformation:

Theorem 3.1. *Any affine transformation from an affine plane onto itself is determined uniquely by the image of three non-collinear points.*

Proof. Given three non-collinear points and an affine transformation, we can always construct a base with them, and apply the proposition 32.16 of the book [RRR07]. □

This theorem can serve us to have a very visual representation of the type of transformation that we can perform. If we imagine two affine plains, with three points in each of them, an affine transformation can map one set of points to the other, and the images of the rest of points of the plane will be determined by a grid created from the three selected points.

After this general introduction to affine transformations, we need to define how we are going to construct an affine correction of an image. If we want to be able to construct the transformation, we need to find at least 3 features in our image. Suppose that we have $n \geq 3$ features in our tuples $src = ((s_{1,1}, s_{2,1}), \ldots, (s_{1,n}, s_{2,n}))$ and $dst = ((d_{1,1}, d_{2,1}), \ldots, (d_{1,n}, d_{2,n}))$, in this case we can find the matrix A of f by solving the system:

$$\left(\begin{array}{ccc} s_{1,1} & \cdots & s_{1,n} \\ s_{2,1} & \cdots & s_{2,n} \\ \hline 1 & \cdots & 1 \end{array}\right) = \left(\begin{array}{cc|c} h_{1,1} & h_{1,2} & b_1 \\ h_{2,1} & h_{2,2} & b_2 \\ \hline 0 & 0 & 1 \end{array}\right) \left(\begin{array}{ccc} d_{1,1} & \cdots & d_{1,n} \\ d_{2,1} & \cdots & d_{2,n} \\ \hline 1 & \cdots & 1 \end{array}\right)$$

If we have exactly 3 references, the system will be determined, and we can solve it by simple linear algebra. On the other hand, if we have more than 3 references, the system is over-determined, and we can use least squares method to find the affine transformation that minimizes the error in all the references.

Once the matrix is found, applying the transformation is a trivial operation, given a point $P = (p_1, p_2) \in \mathbb{R}^2$ we can find its image by f named $f(P) = (q_1, q_2) \in \mathbb{R}^2$ by the matrix computation

$$A \cdot \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix}$$

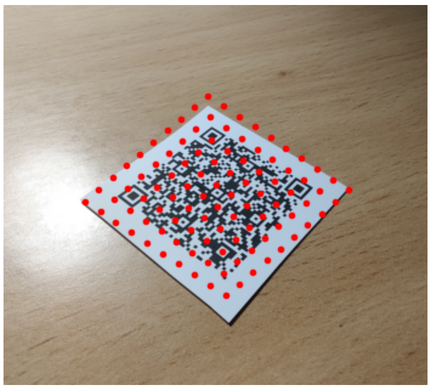


Figure 3.5: Example of sampling a transformation created with the affine method

3.3.2 Projective

The affine method works fine with simple deformations, but begins to fail when we try to correct things like the perspective deformations. When we take a photo we are projecting the view from the camera to the plane of the sensor, which generates a perspective deformation. This type of deformations breaks the geometric property of affine transformation of preservation of parallel lines. But we can generalize the linear transformations in the plane to correct this deformation, using projective transformations.

Before presenting projective transformations we need to define projective spaces. A projective space is defined as a quotient of a vector space over a field K by the equivalence relation

$$v \sim u \iff \exists \lambda \in \mathbb{K}, v = \lambda u.$$

If we take $\mathbb{R}^3$ and make the quotient defined above we obtain the real projective plane $P^2\mathbb{R}$. The main geometric property of $P^2\mathbb{R}$ is that every pair of lines cross at exactly one point.

We can understand $P^2\mathbb{R}$ as a generalization of the affine plane $\mathbb{R}^2$. If we take the affine plane and add a "line of infinity", which we will denote by H_∞, the resulting space is $P^2\mathbb{R}$. The parallel lines from the affine space that don't cross in the standard plane will cross in a point of the line of infinity.

A projective transformation is an isomorphism between projective spaces. Thanks to its definition, a projective transformation can map two parallel lines to crossing lines, breaking one of the main constraints of the affine transformations. Although this seems to break with some intuitive notions of linearity, these types of transformations are linear by definition because they are isomorphism between quotients of vector spaces.

As in the affine case, the projective transformations preserves collinearity between points and preserves the cross ratios between collinear points. When we want to select projective coordinates in $P^2\mathbb{R}$, we can use a restricted set of basis to give compatibility with affine coordinates. A projective basis of $P^2\mathbb{R}$ needs four points, and to get the compatibility we need to pick the last one from the infinity line H_∞. In this case, given a point $P = (x_1, x_2) \in \mathbb{R}^2$, their projective coordinates in this basis will be $(x_1,\ x_2,\ 1)$. All the points in H_∞ will have the third coordinate 0, and the points on the affine plane 1. This representation matches with the extended representation that we used earlier to get the matricial expression of affine transformations.

From this insight, is not surprising to find that the matrix representation of a projection transform will be also very similar. Given a projective transformation f, it will have an associated matrix

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & 1 \end{pmatrix}$$

Given a point $P = (p_1,\ p_2,\ p_3) \in P^2\mathbb{R}$ we can find its image by f named $f(P) = (q_1,\ q_2,\ q_3) \in P^2\mathbb{R}$ using the following matricial computation

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}$$

Using the next theorem, we can end the bridge between the affine transformations and the projective ones:

Theorem 3.2. *Given a projective transformation f with $n \times n$ matrix $A = (a_{i,j})_{i,j}$, this transformation is also affine, i. e. $f(H_\infty) \subseteq H_\infty$, if and only if $a_{n,1} = a_{n,1} = \cdots = a_{n,n} = 0$.*

Proof. The proof can be found in the Lemma 3.5.1 of the book [Cas14]. □

Basically, the theorem tells us that the projective transformations that send H_∞ to H_∞ and the affine space to itself, are exactly the affine transformations, because they correspond to having the last row of the matrix A with zeroes except for the right corner with the one, which is exactly the shape of the affine matrices. Finally, we need a result that help us determine how many references we will need:

Theorem 3.3. *Any projective transformation from a projective plane onto itself is determined uniquely by the image four points not collinear three-by-three.*

Proof. Given four points not collinear three-by-three and a projective transformation, we can apply the theorem 2.8.1 of the book [Cas14]. □

After this general introduction to projective transformations, we need to define how we are going to construct a projective correction of an image. Thanks to the previous theorem, we know that we need to find at least 4 features in our image to build the transformation. Suppose that we have $n \geq 4$ features in our tuples $src = ((s_{1,1}, s_{2,1}), \ldots, (s_{1,n}, s_{2,n}))$ and $dst = ((d_{1,1}, d_{2,1}), \ldots, (d_{1,n}, d_{2,n}))$, in this case we can find the matrix A of f by solving the system:

$$\begin{pmatrix} s_{1,1} & \cdots & s_{1,n} \\ s_{2,1} & \cdots & s_{2,n} \\ 1 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & 1 \end{pmatrix} \begin{pmatrix} d_{1,1} & \cdots & d_{1,n} \\ d_{2,1} & \cdots & d_{2,n} \\ 1 & \cdots & 1 \end{pmatrix}$$

If we have exactly 4 references, the system will be determined, and we can solve it by simple linear algebra. On the other hand, if we have more than 4 references, the system is over-determined, and we can use least squares method to find the projective transformation that minimizes the error in all the references.

Once we found the matrix, we can apply the transformation in the same way that in the affine case, appending 1 to the input point and multiplying by the matrix A.

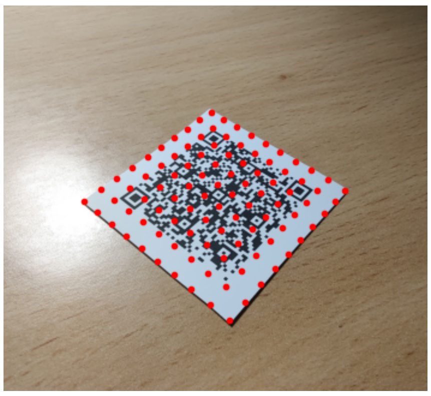


Figure 3.6: Example of sampling a transformation created with the projective method

3.3.3 Cylindrical

The previous two methods were lineal ones, in the affine or projective space. As we said earlier, in a photo of a QR Code we can find two type of deformations: the intrinsic deformation from the surface where the code is over and the perspective deformation caused by the view from the camera. The two previous linear transformations can recover the perspective deformation from the camera, but they can't undo non-linear deformation in the surface.

A very common case in everyday life is the cylindrical deformation. Sometimes a manufacturer needs to use a QR Code in a product that is a bottle or has other cylindrical shapes, and there is no way to put the code flat. Normally, the manufacturer assumes that the error correction from the last decoding step will make up for the geometrical correction errors. But relaying in the error correction reduces the reliability of the QR Codes, and makes the reading process slower. One straightforward solution to this problem is trying to create a non-linear transformation capable of recovering all the contents without errors.

In the recent years, several papers [LZ17][Li+13][Li+19][LWW15] have appeared trying to tackle this problem. All the papers try to solve the problem of QR Codes with cylindrical deformation, and use a common solution: using a non-linear projection to the target surface, in these cases a cylinder, proceeded by a projective transformation. This general idea is refined in each paper with different innovations: new localization methods, different types of cylindrical projections, etc.

As we said, all those methods use a similar approach underneath. The general methods have two main components: a 2D to 3D surface map and a 3D to 2D projective transformation. The 2D to 3D surface map is a non-linear application $G : \mathbb{R}^2 \to \mathbb{R}^3$ such that the 3D surface generated by its graph is the target surface where we expect the QR Code to be in. The 3D to 2D projective transformation is constructed by the same methods explained in the previous section, so we would not detail it again.

Given a surface $G : \mathbb{R}^2 \to \mathbb{R}^3$ and a projective transformation h with matrix H, we can compose those two to define a non-linear transformation f:

$$f : \mathbb{R}^2 \xrightarrow{G} \mathbb{R}^3 \xrightarrow{h} \mathbb{R}^2$$

In the referenced papers, we find different possible G mapping to the cylindrical surface. We decided to take the most natural mapping to the cylinder using a graph of a function. When we take a photo of a QR Code, it can be decoded only if from the camera point of view we can see all its surface. This imposes some restriction in the possible surfaces found. In particular, we need to introduce a new definition:

Definition 3.4. *A surface $G : \mathbb{R}^2 \to \mathbb{R}^3$ is the graph of a continuous function $g : \mathbb{R}^2 \to \mathbb{R}$ if and only if*

$$G(x, y) = (x, y, g(x,y))$$

We can define a cylindrical surface $G : \mathbb{R}^2 \to \mathbb{R}^3$ that is the graph of a function $g : \mathbb{R}^2 \to \mathbb{R}$ using the following function:

$$g(x,y) = \begin{cases} \sqrt{r^2 - (c_0 - x)^2} & \text{if } r^2 - (c_0 - x)^2 \geq 0 \\ 0 & \text{if } r^2 - (c_0 - x)^2 < 0 \end{cases}$$

In our chosen function, we have two parameters to the surface, the radius of the cylinder $r \in \mathbb{R}$ and the first coordinate of any point in the center line of the cylinder $c_0 \in \mathbb{R}$.

Now that we have chosen a target surface function, we need to talk about the process of building the transformation from the references. The standard way to find the cylinder parameters is to find an ellipse that fits one of the two cylindrical deformed sides of the QR Code. The ellipse can be fitted to one of the sides with two position patterns using the least squares method and the corners in that external side. With the fitted ellipse, we will have a radius and a center coordinate. Using some simple scaling, we can get a radius and center coordinate that is scaled to the space of the corrected image. This two values will be the ones used in the application G.

With G already build, we need to find the appropriate values for the matrix H. This time we have a bigger 3×4 projective matrix, counting the number of equations and unknowns, we can deduce that we will need at least 6 points.

Suppose that we have $n \geq 6$ features in our tuples $src = ((s_{1,1}, s_{2,1}), \dots, (s_{1,n}, s_{2,n}))$ and $dst = ((d_{1,1}, d_{2,1}), \dots, (d_{1,n}, d_{2,n}))$. For each point $(d_{1,i}, d_{2,i})$ in dst, we will construct its cylindrical counterpart

$$\forall (d_{1,i}, d_{2,i}) \in dst,\ G(d_{1,i}, d_{2,i}) = (e_{1,i}, e_{2,i}, e_{3,i})$$

Using the cylindrical destiny landmarks and the source ones, we can find the projective matrix solving the system:

$$\begin{pmatrix} s_{1,1} & \cdots & s_{1,n} \\ s_{2,1} & \cdots & s_{2,n} \\ 1 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & 1 \end{pmatrix} \begin{pmatrix} e_{1,1} & \cdots & e_{1,n} \\ e_{2,1} & \cdots & e_{2,n} \\ e_{3,1} & \cdots & e_{3,n} \\ 1 & \cdots & 1 \end{pmatrix}$$

As in the cases of affine and projective transformations, this system can be determined or over-determined, and we will use least squares if needed.

Suppose that we have build a cylindrical transformation f with a surface mapping G and a projective transform h of matrix H. Given a point $P = (p_1, p_2) \in \mathbb{R}^2$, to find $f(P) = (q_1, q_2) \in \mathbb{R}^2$ we will first apply G to P, obtaining $G(P) = (r_1, r_2, r_3) \in \mathbb{R}^3$ and use it to compute the result:

$$H \cdot \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ 1 \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix}$$

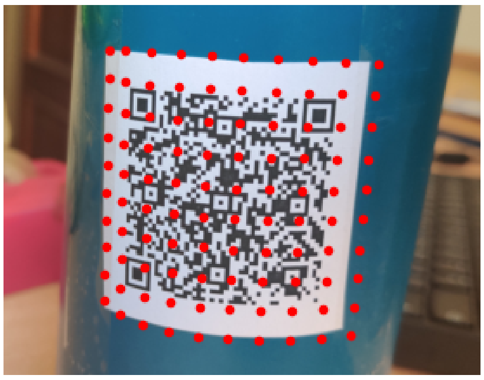


Figure 3.7: Example of sampling a transformation created with the cylindrical method

3.3.4 Thin Plate Splines

As we said in the introduction to the previous method, when a QR Code is in a non-linear surface we will need some non-linear correction to recover all the information. But in the last method, for each kind of surface we will need a new function, which will not correct the other kinds. A very desired improvement is a non-linear method that is independent of the shape of the surface to correct. To solve this problem we decided to try a different idea: using two-dimensional non-linear interpolation of surfaces to construct the transformation that assumes a smooth surface between the reference points.

Based on previous experiences using a two-dimensional interpolator named Thin Plate Spline, we decided to try it as a geometric correction. To introduce this method, we need to begin explaining the concept of radial basis functions.

Definition 3.5. *A radial function is a real-valued function $h : [0, \infty) \to \mathbb{R}$.*

Definition 3.6. *Given some point $c \in \mathbb{R}^n$ and a radial function h, a radial kernel of h in c is a function $h_c : \mathbb{R}^n \to \mathbb{R}$ such that*

$$h_c(x) = h(\|x - c\|)$$

Definition 3.7. *Given a radial function h, h is a radial basis function (RBF) if exists a set of points $\{c_k\}_{k=1}^N \subseteq \mathbb{R}^n$ for which the associated radial kernels $\{h_{c_k}\}_{k=1}^N$ form a basis of the Haar Space, i.e., the matrix*

$$K_h = \begin{pmatrix} h_{c_1}(c_1) & h_{c_1}(c_2) & \cdots & h_{c_1}(c_N) \\ h_{c_2}(c_1) & h_{c_2}(c_2) & \cdots & h_{c_2}(c_N) \\ \vdots & \vdots & \ddots & \vdots \\ h_{c_N}(c_1) & h_{c_N}(c_2) & \cdots & h_{c_N}(c_N) \end{pmatrix}$$

is non-singular.

Radial basis functions are commonly used for function approximation in numerical methods. Another application from this function comes in the field of machine learning, where have been used as an alternative activation function for neural networks [CCG91].

There are many types of RBFs, but we will focus on the polyharmonic radial basis functions (PRBF), which are function from one of the two following shapes:

$$\begin{array}{ll} h(r) = r^k & k = 1, 3, 5, \dots \\ h(r) = r^k \cdot \ln(r) & k = 2, 4, 6, \dots \end{array}$$

We have chosen the PRBF because they provide very good approximation results for many types of smooth surfaces without any tuning parameters. Other types of RBFs have different parameters that have to be tuned to each type of interpolated function.

In particular, we decided to choose the Thin Plate Spline (TPS), which is a PRBF with function:

$$h(r) = r^2 \cdot \ln(r)$$

We will base our use of the TPS on the papers [Boo89][WG06].

The TPS has the property of minimizing the bending energy between references, which is the integral defined by:

$$I_f = \iint \left[\left(\frac{\partial^2 f}{\partial x_1^2} \right)^2 + 2 \left(\frac{\partial^2 f}{\partial x_1 \partial x_2} \right)^2 + \left(\frac{\partial^2 f}{\partial x_2^2} \right)^2 \right] \mathrm{d}x_1 \, \mathrm{d}x_2$$

In one dimensional functions, the cubic spline is the function that has the property of minimizing the bending energy, so the TPS is a natural generalization of the cubic spline to two-dimensional functions.

Using a PRBF, we can define polyharmonic splines. A polyharmonic spline is linear combination of PRBF with a linear affine transformation to achieve function approximation, surface approximation or data interpolation. The PRBF gives a non-linear local deformation around the references used, while the affine transformation created a base linear approximation to the surface used globally. Given a PRBF h, a set of centers $\{ c_k \}_{k=1}^N \subseteq \mathbb{R}^n$ for which the kernels of h form a basis, we can define a polyharmonic spline based on h by the formula

$$f(x) = v^T \begin{pmatrix} x \\ 1 \end{pmatrix} + \sum_{i=1}^N w_i \cdot h(\|x - c_i\|)$$

where $\{ w_k \}_{k=1}^N \subseteq \mathbb{R}$ are the weights of the RBF and $v \in \mathbb{R}^{n+1}$ the weights of the affine transformation.

Until now, we have presented how to build polyharmonic splines for fitting a one-dimensional real function. In the case that we want to approximate more dimensions, we simply approximate various one dimensional functions as polyharmonic splines, which all together are component-wise equal to the multi dimensional one. In other words, given $f : \mathbb{R}^n \to \mathbb{R}^m$, we can consider $f_1, f_2, \dots, f_m : \mathbb{R}^n \to \mathbb{R}$ such that $f = (f_1, f_2, \dots, f_m)$, and approximate each f_i as a polyharmonic spline to determine the approximation of f. We can see examples of this technique applied to TPS in the papers [Boo89][WG06].

Now we have to translate all these concepts to how we can create the image correction. As in the other methods, we have to explain how to build and how to apply the TPS transformation. Based on the paper [Boo89], and given the construction of the system of equations, in this method we only need 1 or more references, although the quality of the results are very dependent of the quality and number of references. Unlike in the previous methods, there are no possibilities of over-determining the system, because there is no number of maximum references, the TPS matrix will grow in proportion to the number of references.

Suppose that we have $N \geq 1$ features in our tuples $src = ((s_{1,1}, s_{2,1}), \ldots, (s_{1,N}, s_{2,N}))$ and $dst = (D_1, D_2, \ldots, D_N) = ((d_{1,1}, d_{2,1}), \ldots, (d_{1,N}, d_{2,N}))$. We will take the points dst points as the centers of our TPS, h will we the PRBF of the TPS given above, and our function will be approximated by components $f = (f_1, f_2)$. Using all this information we can construct the two TPS equations:

$$f_1(p_1, p_2) = v_1^T \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} + \sum_{i=1}^{N} w_{1,i} \cdot h(\|(p_1, p_2) - D_i\|)$$

$$f_2(p_1, p_2) = v_2^T \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} + \sum_{i=1}^{N} w_{2,i} \cdot h(\|(p_1, p_2) - D_i\|)$$

We can translate these equations to a matricial equation:

$$\begin{pmatrix} f_1(p_1, p_2) \\ f_2(p_1, p_2) \\ 1 \end{pmatrix} = \begin{pmatrix} v_1 & v_2 \end{pmatrix}^T \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} + \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,N} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,N} \end{pmatrix} \begin{pmatrix} h_{D_1}((p_1, p_2)) \\ h_{D_2}((p_1, p_2)) \\ \vdots \\ h_{D_N}((p_1, p_2)) \end{pmatrix}$$

We can rename some matrices:

$$A = \begin{pmatrix} v_1 & v_2 \end{pmatrix} \qquad W = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,N} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,N} \end{pmatrix}$$

Which leave the equation as:

$$\begin{pmatrix} f_1(p_1, p_2) \\ f_2(p_1, p_2) \\ 1 \end{pmatrix} = A^T \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} + W \begin{pmatrix} h_{D_1}((p_1, p_2)) \\ h_{D_2}((p_1, p_2)) \\ \vdots \\ h_{D_N}((p_1, p_2)) \end{pmatrix} \tag{3.1}$$

Now that we have put our system as matrices (except for the non-linear application of h), we need to build our transformation finding all the values in the matrices W and A using the references. Let

$$S = \begin{pmatrix} s_{1,1} & \cdots & s_{1,N} \\ s_{2,1} & \cdots & s_{2,N} \end{pmatrix} \qquad D = \begin{pmatrix} d_{1,1} & \cdots & d_{1,N} \\ d_{2,1} & \cdots & d_{2,N} \\ 1 & \cdots & 1 \end{pmatrix}$$

$$K_h = \begin{pmatrix} h_{D_1}(D_1) & h_{D_1}(D_2) & \cdots & h_{D_1}(D_N) \\ h_{D_2}(D_1) & h_{D_2}(D_2) & \cdots & h_{D_2}(D_N) \\ \vdots & \vdots & \ddots & \vdots \\ h_{D_N}(D_1) & h_{D_N}(D_2) & \cdots & h_{D_N}(D_N) \end{pmatrix}$$

we can build the system applying the previous matricial equation to all the references:

$$S = A^T \cdot D + W \cdot K_h$$

But this system is undetermined, we need six more equations to choose a solution. For this reason, we impose that the vectors of coefficients $w_{1,i}$ and $w_{2,i}$ sum to zero, and that their cross-products with the first and second rows of the matrix D also sum to zero. This conditions can be sum up by one matricial equation:

$$0 = D \cdot W^T$$

Finally, we can join the two matricial equations in one:

$$S = A^T \cdot D + W \cdot K_h \implies \left.\begin{array}{r} S^T = D^T \cdot A + K^T \cdot W^T \\ 0 = D \cdot W^T \end{array}\right\} \implies \begin{pmatrix} S^T \\ 0 \end{pmatrix} = \begin{pmatrix} K_h^T & D^T \\ D & 0 \end{pmatrix} \begin{pmatrix} W^T \\ A \end{pmatrix}$$

This last system has the W and A unknowns already isolated, and thanks to our extra equation, is a determined system, so we can solve it using linear algebra.

Once we have found the two matrices W and A, we can apply a TPS transformation to any point $P = (p_1, p_2)$ using the equation 3.1.

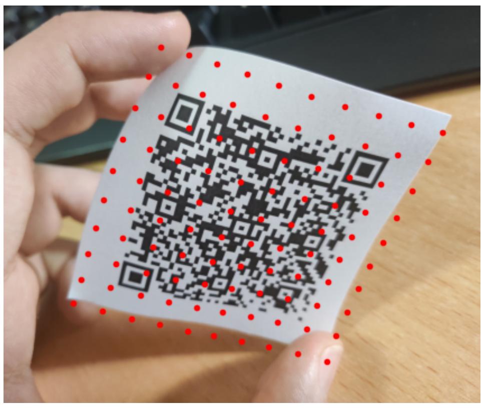


Figure 3.8: Example of sampling a transformation created with the TPS

Chapter 4

Methodology

In this chapter we will give all the details of the development of this project. The first section will explain how we have scheduled project. In the second one, we will introduce the tools and software dependencies used, and the reason behind those choices. Finally, we will introduce the software project, from the patterns used in the design, to a high level specification and the main features of the current implementation.

4.1 Software design

In this section, we will define the functional and non-functional requirements. These requirements are a refinement of the goals to the first chapter to a software specification.

4.1.1 Functional requirements

The functional requirements describe the features or actions of the software system. We have chosen the following ones:

- **Modular library of QR decoding:** In this project we are not interest in creating another end to end QR decoding software, we are interested in creating a modular library which let us build different pipelines for decoding QR Codes. These features are the central point of all the implementation and defines the primary users of the software: developers.
- **Extensible location methods:** In this framework, we want to be able to change the method used to localize the QR Codes in the image in an easy and implementation independent way. That is, using some kind of registration of the methods available and some way to let the user select the one they want to use, like with a collection of identifiers. In this particular project, we will only implement the localization by ratios, but an easy extension will be implementing some advanced localization methods explained in the different papers of the cylindrical algorithm.

- **Extensible correction methods:** As we know, the main point of this project is comparing four different methods of correction. But we don't want to have those four methods hardcoded in four functions. We need to define a common interface for corrections, with a way to easily register new methods, and a way to show the user the available ones. In this project we will have four methods of correction, but we want to easily be able to add a new one with the minimum alteration of code of a user pipeline.

- **Extensible decoding methods:** In the same way that we want extensible localization and correction, we want to be able to choose the decoding method use. Like in the localization case, we will only implement one method for this project, but in future work we need to check if changing the decoding backend we get more successful reads, to ensure that the results about corrections are not conditioned by the decoding backend chosen.

- **Multiple QR support:** Given that we have to build a custom QR Code localizator, is not very demanding to add the requirement to support multiple codes in the same image. This will enable the use of more rich datasets and testing against more real images. The main flow of any QR decoding pipeline will branch after the localization, doing the rest of the pipeline to each QR found.

- **Stackable corrections:** Although we only have considered using one correction as our main experiment, there is a possibility that doing more than one correction gives better results than with only one. Furthermore, we could try to stack different corrections, to add up their advantages and counter their disadvantages. For this reason, is interesting to add the corrections in a way that the creator of a pipeline can add one or more corrections one after another.

- **Integrated plotting:** The challenging part of this project is based in Computer Vision, therefore plotting is essential part of any debugging. For this reason, plotting has to be a first class citizen, easy to use and with sane defaults.

- **Abstraction over datasets:** We want to have an interface over the data in the datasets, with a native structure which gets filled using the labels of the images. This interface can enable making changes over the datasets label format or data without changing the implementation of the experiments.

- **Easy interface for experiments:** We want to have an easy interface to make all the necessary experiments. The interface could be an experiment runner through CLI for example, which enables the execution of code with modifying it.

4.1.2 Non-functional requirements

The non-functional requirements describe how a system is going to be, specifying how is going to do its features or actions. For this software, the non-functional requirements have been:

- **Reproducible:** Since we are trying to compare different methods and extract some conclusions, it's fundamental to have the possibility of reproduce the results by a third party researcher. Reproducible software is a hard goal, but we can easily make some design selection to make our software almost reproducible. Examples of this type of design choices include selecting a package manager that supports basic locking of dependencies or using only free or open source software dependencies.
- **Acceptable performance:** We don't want that the library has a performance comparable to commercial QR Code readers, but we want to be able to run experiments in a medium class laptop without having to let it all night running. This requirement could be achieved by some optimization of the code if needed, until the performance is considered acceptable.
- **Clean and standard compliant code:** We want to produce a legible, commented and standard compliant code, in the sense of the style format of the language used. By making clean code and following the standard style format, we can improve the experience of anyone that tries to read the code, promoting the peer review of our implementation and results.
- **Cross-platform:** We want to use a language and dependencies that are available in the main desktop platforms (Windows, MacOS, GNU/Linux and the BSDs), to being able to reach as many people as possible, and avoid creating unnecessary barriers.

4.2 Development environment

In this section, we will explain each of the choices done in the selection of the development environment for the project. The development environment is composed by all the tools and dependencies used in the development or execution of our software system.

The main choice of any software project is which programming language is going to use. In our case, we wanted a programming language with some of our non-functional requirements (cross-platform, with a reproducible package manager and acceptable performance) and which also has some good base libraries for computer vision. We ended up choosing the Python [Gui90] programming language, because it has a very active community and ecosystem, and it fulfils all our needs.

The next logical choice was the dependencies and libraries of our project. We did some research about the state of the art of computer vision libraries in python. The first decision was that we would use the numerical base from Numpy [VCV11], Scipy [Vir+20] and Matplotlib [Hun07]. This three libraries provide support for efficient multidimensional arrays in Python, common numerical methods using this arrays and a general plotting library. Some time ago they would also offer a general image reader, but this feature has been extracted to a library named ImageIO [Sil+20], which can open and save images from any format to a Numpy array. So we will also use the library ImageIO as our image reader and writer.

After this first selections, we have our core mathematical libraries and our image reader. The next selection is which computer vision library we want to use. In this selection, we have two main options: OpenCV [Its15] or Skimage [Wal+14]. OpenCV is an older and more stable alternative, which was written as a C++ library for computer vision, but has official Python bindings. Skimage is an extension library to Scipy, adding similar functionalities than the ones offered by OpenCV. The two options are based in Numpy arrays, so were compatible with our previous choices. Finally, we ended up choosing Skimage, because is a more modern and clean library, and we think that it has a lot of potential.

After solving our computer vision needs, we need to select which will be our QR Code related dependencies. One first logical dependency was a QR Code generator. For this purpose, we found a library named Qrcode [Loo10], which suited all our needs. Finally, as we said in the first chapters, we aren't interested in redoing all the decoding of a QR Code, and we would want to use a common QR Code reader to make this step. After some research, we found that we have three options in Python: Zbar [Sou09], Zxing [OST08] or a reader from OpenCV. Making some tests and comparisons, the Zbar wrapper Pyzbar [Mus16] shown to be a very lightweight dependency and to have good results in decoding QR Code images, so we decided to choose this library.

Finally, we needed a library for making image augmentation to the generated QR Codes, thus enabling the generation of challenging images. In this space we have a lot of options thanks to the rise of use of image augmentation in machine learning, but we don't want a library that forces us to install a full machine learning library only for the image augmentation. For this reason, we ended up choosing Imgaug [Jun+20], a library that only depends on common computer vision dependencies, making it lightweight compared to the other options.

Now that we have detailed our dependencies, we can explain what are the tools that we have used in the development of the project. All the development of the software has been done using Git as a version control system (VCS), with GitLab as the Git forge. Git is the standard VCS today and GitLab offers very similar features to its competitors while promoting self hosting for large free software project or enterprises.

In the reproducibility aspect, we have different choices right now in Python. Clearly, we have used the Pip package manager which uses PyPI, the official Python package index. But Pip has some problems that make the reproducibilty hard. For solving this necessity, some tools have appeared in the recent years. The first that we have used is Pyenv, which provides a way to create environments with different Python installations. Using this tool we have lock the Python version to 3.7.2, pulling away the dependency from the Python system install, which changes of version as time passes. The second tool that we have used is Pipenv which is a wrapper on top of Pip. It has features like creating virtual pip environments only for this project and automatic locking of all the dependencies.

During the development of the code, we wanted to find any possible bug as early as possible. To cover this need, the recent versions of Python have introduced gradual typing. Using the official Python compiler MyPy [LT12] we can get static analysis of our code. Furthermore, the most used python editors have support to use this compiler in live coding, enabling hints from the compiler directly in the code. This comes at the cost of adding typing definitions at the function headers, but we think that the advantages are far more important.

All these tools have been integrated in the editor that we have used, which is the Pycharm Integrated Development Editor (IDE). This editor has support to all the tools chosen above, and integrates them nicely with simple GUI configuration. In addition to the introduced tools, the editor offers other important features that we have used in the development, like the profiler or the debugger. Although the editor is commercial, they offer a limited version for free to everyone and the premium one without charging to students.

4.3 Implementation

In this section we will give all the details from the implementation done for this project. The Python language organizes code in modules. A module is a file of python code or a folder with a file named `__init__.py`. We can classify all the code development done thanks to the module structure.

First, we will give details of all the high level modules of the implementation. All the code is classified in three main modules: `qrcreator`, `qrdatasetgenerator` and `tfg`.

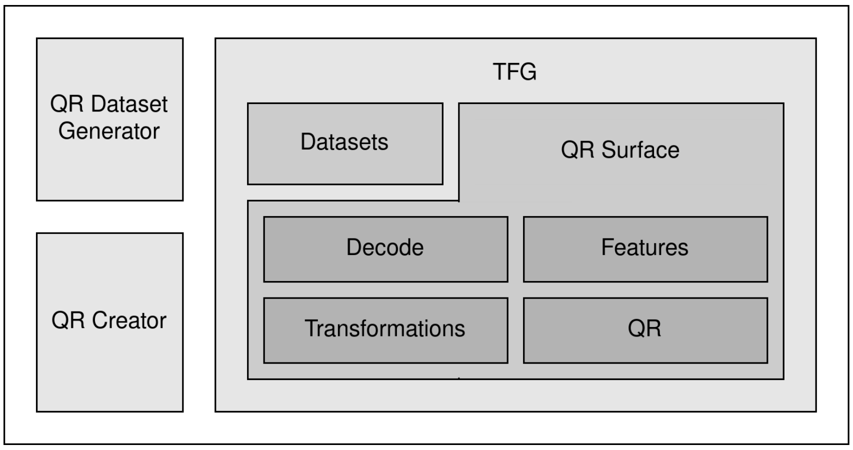


Figure 4.1: Diagram representing the high level Python modules

The `qrcreator` module is build mainly by two files, which are used to generate QR Codes and creating a layout of the generated QR Codes to print many of them in a A4 paper. This module was used to print the QR Code used to create all the datasets that are photos. By preserving this scripts in a module, we can generate more of this QR Code if is needed to compare with existing results in the future.

The `qrdatasetgenerator` module is the generator of the synthetic datasets used for the experiments. The module is composed by two Python files. The first one generates

QR Codes with random parameters like error correction and data. All the metadata from this generated QR Codes is saved alongside them. The second file uses the library Imgaug to augment the generated QR Codes, with some affine and projective transformations. Using this two files we can quickly generate as many QR Codes as needed, but only with affine or projective deformations.

Finally, there is the main module of the project, the module `tfg`. This module contains our library for creating the pipelines of QR Code decoding, which we have named `qrsurface`. Alongside this library, the module also contains a module named `datasets`, which serves as the parser and interface to the datasets, and several files which execute the experiments and compute the results.

The main functionality of the module `tfg` is to be executed using the CLI exposed. If is executed like a module (`python -m tfg`) it will act as a CLI command, with two subcommands `process` and `results`, which makes easier to rerun the processing of the datasets or the computing of the results. The pipeline used to make the comparison of the four correction methods is in the files `tfg/process.py` and `tfg/utils.py`, and can be executed with the subcommand `process`. The other subcommand `results` reads all the metadata of the datasets and the last results for each image, and makes some plots and tables with them.

Since the submodules `qrsurface` and `datasets` contain the majority of the implementation of this project we will make a detailed explanation of each one. From now on we will refer to the `datasets` as the datasets interface module and to the `qrsurface` as the QR Surface module.

4.3.1 Datasets interface module

The datasets interface module serves as a Python representation of all the datasets available. The module only has three public classes `BitmapCollection`, `LabeledQR` and `LabeledImage`, and a public enumeration `Deformation`. We have included diagrams of the fields of the three classes in the figure 4.2.

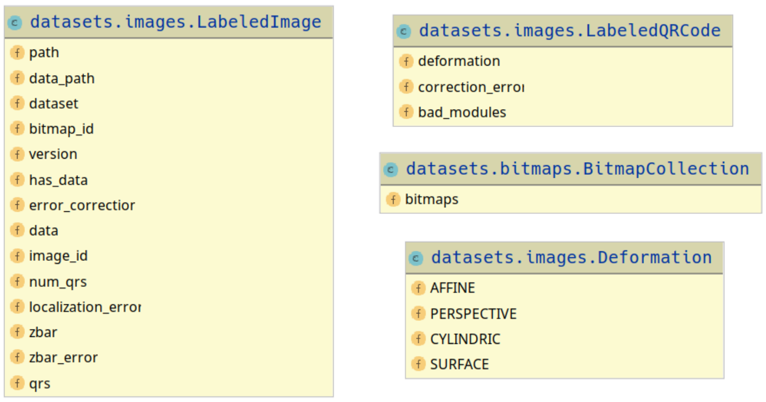


Figure 4.2: Diagram of the public classes of the datasets interface module

The main class of the module is `LabeledImage`, which represents the metadata that we can have for an image of a dataset. At the beginning of execution, the main of the module `tfg` calls a class method from `LabeledImage` that searches for datasets in the given folder, and returns a list of `LabeledImage`. A `LabeledImage` can have 0 or more `LabeledQR`, which represent the metadata of each QR Code in the image. The `LabeledQR` has a field named deformation, which will have one of the values defined at the `Deformation` enumeration.

Apart from the list of images, there is another component of the datasets, the `BitmapCollection`, which represents the collection of original bitmaps of all the QR Codes used in the datasets. Thanks to this collection we can check for errors at the module level after each decoding.

4.3.2 QR Surface module

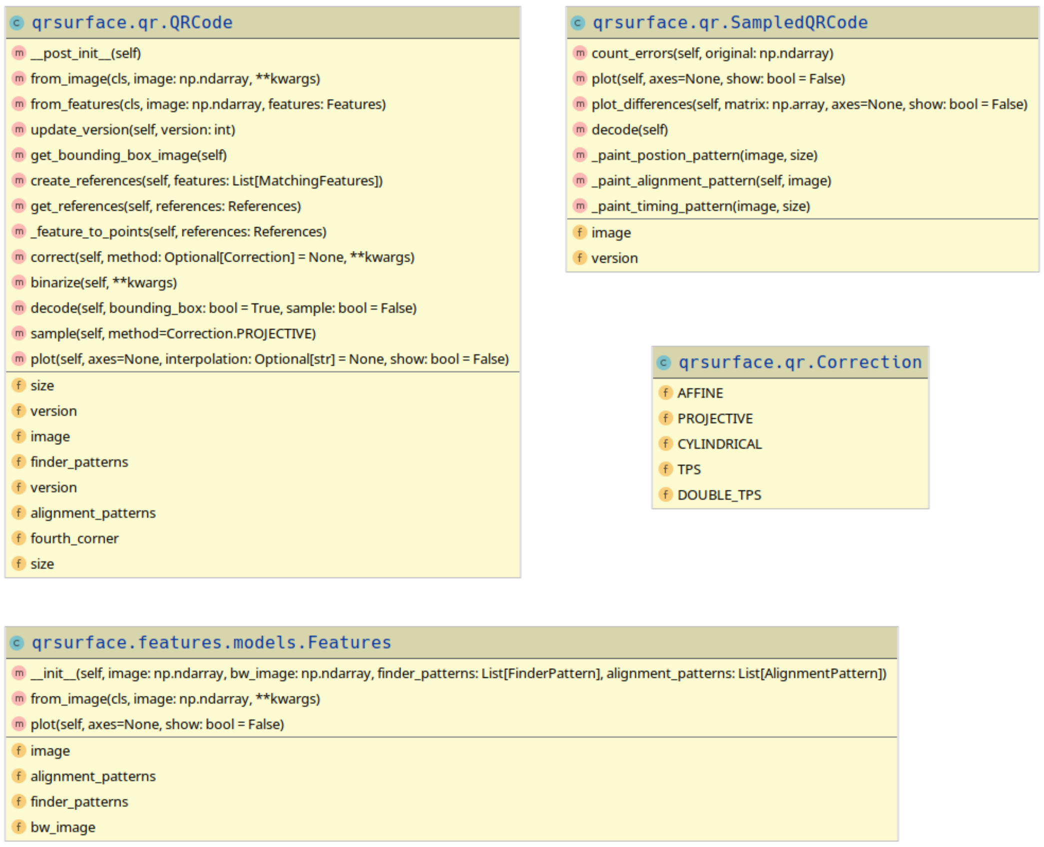
The QR Surface module is the modular library for decoding QR Codes. This library is the most complex module of this project, and is structured in four submodules: `features`, `qr`, `transformations` and `decode`.

The `features` submodule has the abstraction over localization, with an enumeration of the available methods and a main class named `Features`. The `qr` provides the classes that represent valid QR Codes, including `QRCode` and `SampledQRCode`. This two classes have methods that call to the other submodules, to avoid the need to import a lot of functions. The `transformations` enables the abstraction over correction methods, with an enumeration and a main function named `correct`. Finally, the `decode` submodule brings the abstraction over decoding methods, also with an enumeration and a main function named `decode`.

The use of the module can be described using the public classes that exposes. We can see a diagram of classes with fields and methods of each one of the public classes of the QR Surface module in the figure 4.3.

The `Features` class represents all the features that a localizer has found in an image. This is the first step of the pipeline, can be created manually by calling a class method of `Features`, or automatically by the `QRCode` class.

The `QRCode` class represent one QR Code in the image. Can be obtained by calling a method with a `Features` object or with an image, but the return will be a list of `QRCode`, because the image can have zero or more codes. The `QRCode` class represents our pipeline, and have methods like correct, sample and decode. Those methods use the backend submodules and have parameters to choose the desired methods of correction or decoding. Finally, if we do a sample on a `QRCode`, we get a `SampledQRCode`. This class represents the bitmap obtained from the pipeline, and has some utilities to extract the results from the decoding.



```
qrsurface.qr.QRCode
m __post_init__(self)
m from_image(cls, image: np.ndarray, **kwargs)
m from_features(cls, image: np.ndarray, features: Features)
m update_version(self, version: int)
m get_bounding_box_image(self)
m create_references(self, features: List[MatchingFeatures])
m get_references(self, references: References)
m _feature_to_points(self, references: References)
m correct(self, method: Optional[Correction] = None, **kwargs)
m binarize(self, **kwargs)
m decode(self, bounding_box: bool = True, sample: bool = False)
m sample(self, method=Correction.PROJECTIVE)
m plot(self, axes=None, interpolation: Optional[str] = None, show: bool = False)
f size
f version
f image
f finder_patterns
f version
f alignment_patterns
f fourth_corner
f size
```

```
qrsurface.qr.SampledQRCode
m count_errors(self, original: np.ndarray)
m plot(self, axes=None, show: bool = False)
m plot_differences(self, matrix: np.array, axes=None, show: bool = False)
m decode(self)
m _paint_postion_pattern(image, size)
m _paint_alignment_pattern(self, image)
m _paint_timing_pattern(image, size)
f image
f version
```

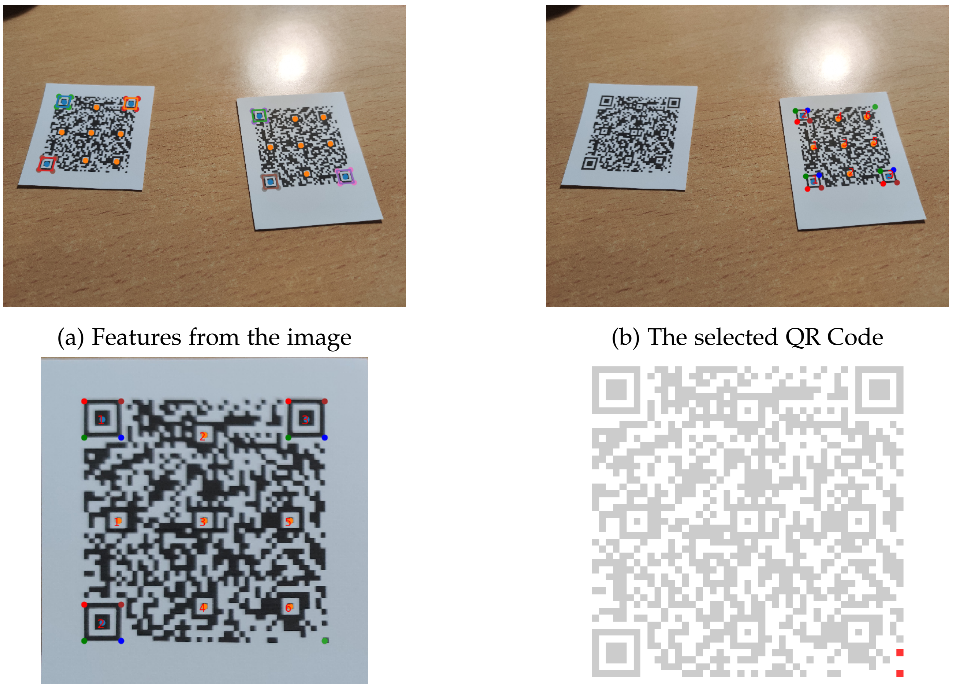
```
qrsurface.qr.Correction
f AFFINE
f PROJECTIVE
f CYLINDRICAL
f TPS
f DOUBLE_TPS
```

```
qrsurface.features.models.Features
m __init__(self, image: np.ndarray, bw_image: np.ndarray, finder_patterns: List[FinderPattern], alignment_patterns: List[AlignmentPattern])
m from_image(cls, image: np.ndarray, **kwargs)
m plot(self, axes=None, show: bool = False)
f image
f alignment_patterns
f finder_patterns
f bw_image
```

Figure 4.3: Diagram of the public classes of the QR surface module

As we can observe in the figure 4.3, each object has its own plot function, and we can pass from one to the other using some methods of the object. This design minimizes the imports and simplifies the user code needed to create a custom pipeline.

In the figure 4.4, we can see the result of calling the plotting methods in different points of a pipeline. The first image represents the output of calling plot over a `Features` object, and shows us all the features that the location algorithm has found in the image. After this step, the method for generating the list of `QRCode` objects from the `Features` has been called, returning a list of two `QRCode` object, because there are two QR Codes in the image. In the second image we can see the output generated by calling the method plot for one of these two `QRCode` objects. We can see how the plot only shows the features inside one of the QR Codes. Just after getting the `QRCode` object we can execute a correction, in this case was a `Correction.TPS`. After the correction, we can use again the plotting method of the `QRCode` object, represented in the third plot, to see the new corrected image and the resulting position of the features. Finally, we have sampled the QR Code. After a sampling, we can use a method for plotting the differences between the original bitmap expected and the one that we obtained using our pipeline. This plot is visualized using the sampled QR with transparency, and overlying red when there is a mismatch between the original and the sampled. This plot is shown in the fourth subfigure.



(a) Features from the image

(b) The selected QR Code

(c) Corrected QR Code

(d) Difference between original and sampled

Figure 4.4: Plots obtained from the integrated plotting methods

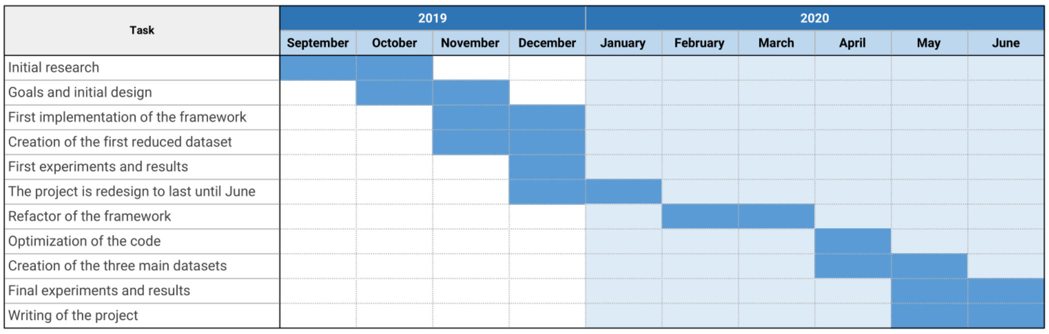
Finally, we would like to remark what optimizations have been done. With the first implementation of the QR Surface module the pipeline was very slow. This confronted our requirement of acceptable performance, and make running experiments impossible in a normal laptop with only 1000 images in the datasets. We decided to find where the code was spending most of the time and optimize it. For that purpose, we used the profiler of PyCharm, which outputs graphs with the percent of time spend by function. Thanks to this analysis, three main bottlenecks were found: the binarization, the parsing of rows before localization by ratios and some contour searching for the corners of finders.

The first issue addressed was the binarization. The process is very slow when applied to all the image, and the code was doing it many times. We solve it by caching the binarization, thus making only one. The second issue was the parsing of rows for the ratios method. To solve this issue we rewrote the function to use Python `itertools` functions, which are more optimized than iterations by a for loop. Finally, the issue of the search of contours was solved by searching only in a "target" area instead on all the image. This three solutions improved greatly the performance, making possible the processing of all the datasets in 5 hours approximately.

We could have followed the optimization easily, by rewriting some functions to use only Numpy methods and by making some functions parallel, like the searching by rows. But in the end, the current performance was good enough to make all the experiments that were needed.

4.4 Temporal planning

The project started in September 2019 and originally was planned to end in January 2020. But as months passed, we began to see that we probably will need more time to finish the goals initially set. Finally, we decided to plan to deliver this project on June 2020, which led to a new temporal planning and the possibility to add some extra features. The final schedule of the project is presented in the table 4.1, with the beginning and end of each one of the high level task done during this project.



Task	2019				2020					
	September	October	November	December	January	February	March	April	May	June
Initial research	🟦	🟦								
Goals and initial design		🟦	🟦							
First implementation of the framework			🟦	🟦						
Creation of the first reduced dataset			🟦	🟦						
First experiments and results				🟦						
The project is redesign to last until June				🟦	🟦					
Refactor of the framework						🟦	🟦			
Optimization of the code								🟦		
Creation of the three main datasets								🟦	🟦	
Final experiments and results									🟦	🟦
Writing of the project									🟦	🟦

Table 4.1: Temporal planning of the project by high level tasks

Chapter 5

Validation

To compare the different methods of correction we will analyse the results of applying all the possible corrections to certain datasets. In this chapter we will explain the experiments done and discuss the results that we can extract from them. First, we will give a description of all the datasets used. Then, we will explain the results obtained from the processing of these datasets.

5.1 Datasets

The first step to making the experiments is constructing the datasets. As we stated in one of our goals, one important requirement is that the datasets represent a diverse sample of photos of QR Codes. As we detailed in that goal, we have two principal ways of achieving the datasets: generating them synthetically or making the photos ourselves. On one hand, if we generate the datasets we can make a large pool of data with constant effort. On the other hand, the synthetic datasets are not representative of the real world, like the photos are, but they require the time of making the photos and labelling. To have the advantages of both, we decided to have datasets of the two types.

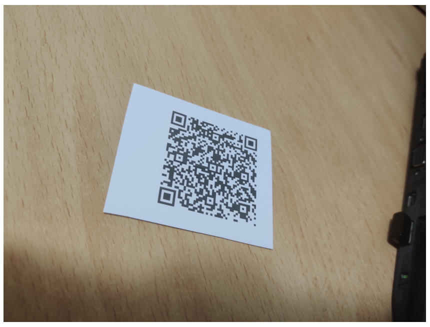
The datasets have been stored following a particular folder structure and labelling format. A dataset is a folder with three subfolders named images, bitmaps and annotations. In images, we have all the images of the dataset. In bitmaps, we have all the original images of the QR Codes that appear in the dataset. The original images consist in images with 1 pixel by QR module and without border. These images enable us to make the module-wise comparison between the result of the decoding and the expected result. Besides the original images, we have a JSON file for each original image, with the information used in the creation of that QR Code, like the version, data and error correction. Finally, in the annotations folder we have one JSON file for each image in the folder images. The JSON file has the same filename stem as the image that represents. Inside the JSON file, we have the filename stem of the original QR Code in that image, the number of QRs in the image and a list objects containing the metadata for each QR in the image, like the type of deformation.

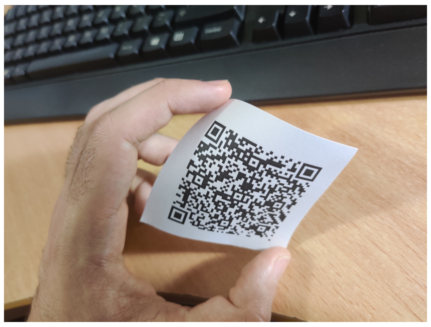
In the end, we have created three datasets:

- **Flat:** The first dataset created, consists in 50 photos of QR Codes in flat surfaces. The QR Code used has been generated using the `qrcreator` module. These photos are the expected targets of a commercial QR decoder, and serve as backward compatibility testing. We also have used this dataset to test the extreme cases, like more than one QR Code by photo, or blurred photos.

- **Random:** The main dataset of this project, which contains 50 photos of QR Codes in cylindrical and random surfaces. This dataset is the one that will enable the majority of our conclusions, because is the one that can show how each correction reacts to challenging deformation. The QR Code used was the same that the one from the Flat dataset.

- **Synthetic:** The synthetic dataset of the project, with 819 images. This dataset was created by three QR Codes for each version from the 1 to the 13, with error correction and data random. Using this 39 QR Codes, we generated the rest of the dataset making 20 augmentations per original image. The augmentations used random affine and projective transformations combined.


(a) Flat


(b) Random

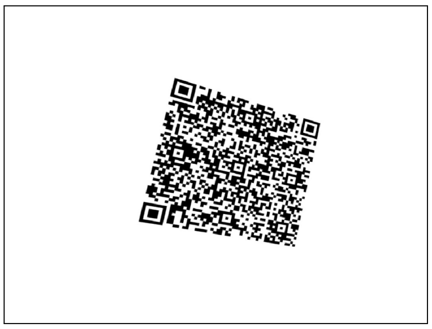

(c) Synthetic

Figure 5.1: Examples of images from the three datasets used

5.2 Results

In this section we will introduce all the results obtained and discuss them. The experiment produced consisted in running all the datasets through a pipeline that tried the four corrections against each image.

First, we need to explain the parameters used for each type of correction. Some tuning parameters were tied to the localization step, and thus affected equally all the correction methods. These parameters were optimized based on maximizing the number of images localized successfully. For the parameters specifics to one correction, the final values were chosen by trying all the candidates and selecting the one which gives the best results in number of total reads. When optimizing the parameters, we tacked into account that the synthetic dataset was much bigger than the other two, and made attempts with only the flat and random datasets to ensure that the parameter was fair. As an example, the order of interpolation for the image inverse mapping was set to 3, i.e. bi-cubic interpolation, to all corrections because maximizes results with all the methods of correction.

Another point to consider before start commenting the results is the final metrics used. After many considerations, we decided to use three metric in the comparison between methods:

- **QR localization:** Whether all the QR Codes in an image where localized successfully, including guessing the versions. This metric is independent of the correction method used, and will only be used to validate that our localizator is good enough,
- **Successful decoding:** Whether each QR Code can be successfully decoded or not. We will also call a successful decoding a read of a QR Code. This is the main metric for comparing correction methods.
- **Ratio of QR pixels failed:** This metric is a float number between 0 and 1. To compute this metric we count how many modules have been guess wrong and divide that count by the total number of modules in the QR Code. This ratio is a more fair metric than the count of bad modules, because the error correction can correct a percent of modules failed in the QR Code, and therefore is not the same failing two modules in a QR of version 2 than in one of version 13. This will be our secondary metric for comparing correction methods

We initially considered another metric, whether the data read from a QR Code was equal to the expected data, but in the experiments, all the images that were read had the expected data. This fact made our data about this metric useless.

On the other hand, we have had one exception in the localization metric. All the images with QR Codes labelled with random deformation skipped the check of the version during the pipeline and the guessed version was substituted by the real one that we have labelled in the metadata. We needed this exception because the algorithm for guessing the version don't work in images with random deformation, because assumes that at least one side of the QR Code is not very deformed. The point of this project was to compare the correction methods, not the localization implementation, and we wanted to have the data about how each method could correct this types of random deformation.

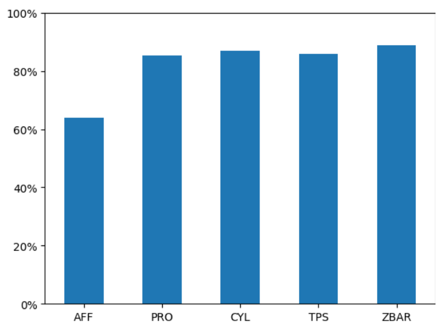
5.2.1 QR localization

The first metric that we want to analyse is the QR localization rate. As we know, this project goal is to compare the correction methods, therefore, the implementation done to the localizer only needed to be good enough to give us all the results needed. We have localized correctly 828 from the 919 images analysed. That is an 89% of success, which is not equally distributed between datasets: the flat and random datasets have 77% and 74% of localization rate, and the synthetic a 91%.

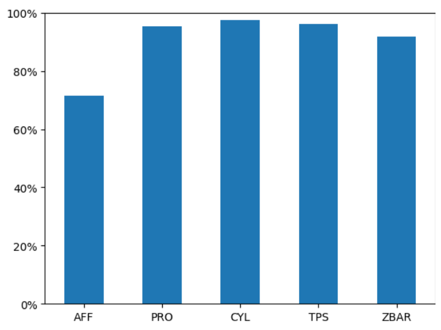
To check if this result was good enough, we checked how many read accomplished the ZBar decoder with the same images. We cannot know which images from ZBar failed from localization and which ones from correction, but their read rate is a lower bound of their localization rate. The ZBar has a read rate of 88%, which can be decomposed in a 60% for the flat dataset, a 16% for the random and a 95% for the synthetic. Clearly, the ZBar localizer performs better in the synthetic images, but seems like it has similar results in the photos. To further proof that point we can compare the read rate of ZBar and our pipeline with projective correction. As we said, ZBar has a read rate of 60% for the flat dataset, while our implementation has a 77%. So, it seems that our localization implementation is good enough for the studied cases, except for the case of guessing the version of QR Codes with random deformation, as we stated earlier.

5.2.2 Successful decoding

Now that we are sure that we have enough data from our localizer, we can begin the comparison of the correction methods. We can compute the read rate using the rate by all the images or by the localized ones. In the figure 5.2 we can see the read rate for all our correction methods and ZBar, by all images and only by the localized ones. We can observe how ZBar performs better in the rate by all images, and worse in the case of the localized ones. This fact suggests that our correction methods are having better results than ZBar. As we said earlier, we don't want to test or compare the performance of our localizer, only of the correction methods. For this reason, we will use the read rate over the images that were localized successfully.



(a) All images



(b) Only images localized by our library

Figure 5.2: Ratio of QR Codes successfully decoded

The second plot of the figure 5.2 we can start the comparison between correction methods. We can see how AFF has very bad results, compared to the other methods, but has achieved more than a 70% of reads, which is pretty good having into account the simplicity of that method. PRO and CYL have very similar results, with CYL having a little better, which corresponds to the cylindrical images. In the end, CYL works very well as an extension of PRO, and has great results in the classic flat case, and very good ones to in cylindrical cases. Finally, TPS has achieved nearly as good results as PRO and CYL, without all the assumptions that these methods have.

Is necessary to remark that because the majority of images were from the synthetic dataset, and with projective deformation, this plots doesn't answer the questions about how each method performs in each deformation. To test if our initial hypothesis was true, we will study the read rates of all the methods based on each one of the considered deformations. In the figure 5.3 we can see the results from this comparison. This figure tries to represent all the combinations of successful decoding between the correction methods used and the deformation labelled at each image. For each combination of deformation and method we are printing a point which is greater when the read rate is greater. This comparison is fairer than the last one, because it doesn't depend on the distribution of deformations on the dataset.

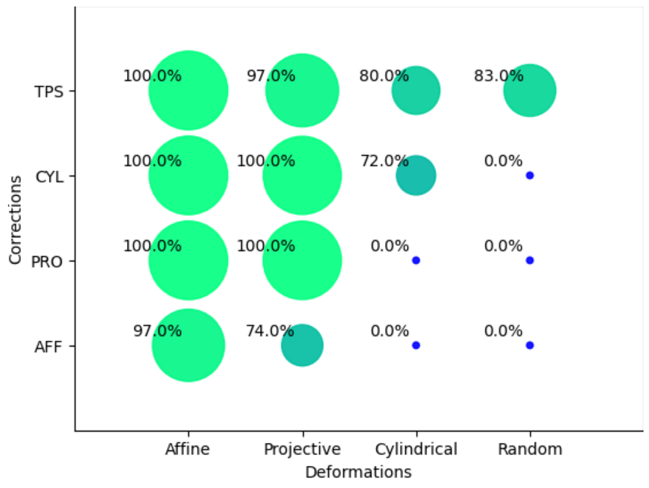


Figure 5.3: Comparison of the read rates by deformation and method

We can analyse the performance of each correction method reading the figure row by row. The AFF method was only capable of decoding with good results the affine and projective deformed images, with 0% of images decoded in cylindrical and random deformations. Also, in the projective deformation obtained the worst read rate of the four methods. The PRO method make similar results to the AFF method, but with perfect results in affine and projective deformations. The CYL method obtains perfect results in affine and projective deformations and an acceptable decoding rate in the cylindrical deformations, but a 0% in the random. Finally, the TPS gets a perfect decoding in affine, nearly perfect in projective, and the best results in cylindrical and random deformations. Surprisingly, TPS has managed to win the CYL method in the cylindrical deformation.

5.2.3 Ratio of QR pixels failed

The next metric that we can study is the ratio of QR pixels, also named modules, failed in each QR Code by method. The first thing that we can study about this metric is which mean and standard deviations has depending on the methods:

	AFF	PRO	CYL	TPS
Mean of the relative error	8.39%	1.43%	0.79%	1.08%
Std. Dev. of the relative error	14.38%	6.81%	5.01%	3.30%

Table 5.1: Metrics of the ratio of modules failed in each QR Code by method

From the table we can discuss how each method has the distribution of errors. The AFF method has more than a 8% of mean of errors and a very high standard deviation too. The PRO and CYL have very good results like in the previous plots, with CYL a little better than PRO. The valuable information comes with the TPS, which has a mean of error a little worse than CYL, but a little better than PRO. Also, it has the smallest standard deviation.

We can study en more details the distribution of errors with the figure 5.4. In this figure we plot the same metric, the ratio of modules failed in each QR Code by method, but using bins and a histogram to show the distribution of errors. This plot help us to explain better the standard deviation of the TPS. The TPS has more QRs with more than a 5% of errors than PRO and CYL, but is the method with less QRs with more than 20% of errors.

From this two results, we can learn that the TPS has more probability of having at least one failed module of the QR Code than PRO and CYL. But the distribution of errors for the TPS is more narrow, and although it fails more in at least one module, it has less probability of having more than 20% of errors. By having this narrow distribution with few errors, the TPS can decode the majority of the images thanks to the Reed Solomon error correction used in the data of the QR Code. This property tells us that the TPS is a very stable method of correction and has very predictable results.

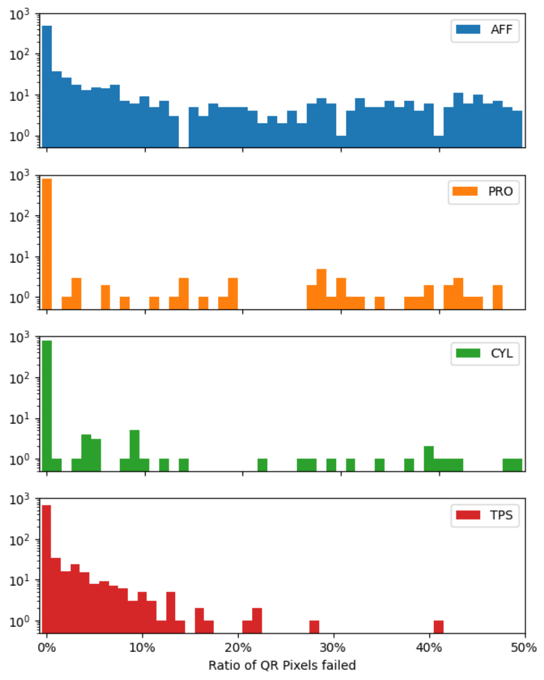


Figure 5.4: Distribution of ratio of modules failed in each QR Code by method

Chapter 6

Conclusions

The problem of reading QR Codes in challenging surfaces keeps being an open problem in many aspects. In this project we have compared several methods for tacking the problem from the geometric correction perspective.

Some of the methods tried have very impressive results. The cylindrical method presented has nearly perfect backward compatibility with the more common projective method, while being able to achieve good results in QR Codes with cylindrical deformation. But the method have several flaws because it can't handle deformations from different shapes.

To solve this problem, we proposed Thin Plate Spline, a general surface independent and non-linear method. The results from this method really overcome our expectations, being close to more linear methods like the projective in flat deformation images, while being able to correct random deformations and cylinders. This method accomplished better results in decoding cylindrical deformation than the cylindrical method while being the only method able to read QR Codes in images labelled with random deformation. We believe that this method is a very safe alternative to the standard ones, because can bring a pretty good read rate to any type of deformation.

Further work

After ending this project, many new paths appear as future works. One clear improvement is trying to solve the problem of localization of QR Code features in random deformations. During all the development of the project, we have had many issues related with the fact that a standard localizer struggles to find the necessary features in images with random deformation. Part of this work was begun by the papers presenting the cylindrical method, and they introduce several techniques to improve the localization.

Another possible extension is trying different decoding backends, with more commercial decoders or a handmade one. By doing this work we could ensure that the decoder use is not influencing in the results found.

Finally, the straightforward extension would be continuing the search of the best correction method for challenging surfaces. We would need to make greater datasets from real photos to be able to extract more meaningful results.

Bibliography

[Boo89] F. L. Bookstein. "Principal warps: thin-plate splines and the decomposition of deformations". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11.6 (1989), pp. 567–585.

[Cas14] E. Casas-Alvero. *Analytic Projective Geometry*. Zürich, Switzerland: European Mathematical Society, 2014.

[CCG91] S. Chen, C. F. N. Cowan, and P. M. Grant. "Orthogonal least squares learning algorithm for radial basis function networks". In: *IEEE Transactions on Neural Networks* 2.2 (1991), pp. 302–309.

[Gui90] Guido van Rossum. *Python programming language*. Python Software Foundation. 1990. URL: `https://www.python.org`.

[Hun07] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[ISO00] ISO Central Secretary. *Information technology — Automatic identification and data capture techniques — Bar code symbology — QR Code*. ISO ISO/IEC 18004:2000. International Organization for Standardization, 2000, pp. 1–114. URL: `https://www.iso.org/standard/30789.html`.

[ISO06] ISO Central Secretary. *Information technology — Automatic identification and data capture techniques — QR Code 2005 bar code symbology specification*. ISO ISO/IEC 18004:2006. International Organization for Standardization, 2006, pp. 1–114. URL: `https://www.iso.org/standard/43655.html`.

[ISO15] ISO Central Secretary. *Information technology - Automatic identification and data capture techniques - QR Code bar code symbology specification*. ISO ISO/IEC 18004:2015. International Organization for Standardization, 2015, pp. 1–117. URL: `https://www.iso.org/standard/62021.html`.

[Its15] Itseez. *Open Source Computer Vision Library*. `https://github.com/itseez/opencv`. 2015.

[Jun+20] Alexander B. Jung et al. *imgaug*. `https://github.com/aleju/imgaug`. Online; accessed 01-Feb-2020. 2020.

[Li+13] Xiaochao Li et al. "Reconstruct argorithm of 2D barcode for reading the QR code on cylindrical surface". In: *2013 International Conference on Anti-Counterfeiting, Security and Identification (ASID)* (2013), pp. 1–5.

[Li+19] Kejing Li et al. "A Correction Algorithm of QR Code on Cylindrical Surface". In: *Journal of Physics: Conference Series* 1237 (June 2019), p. 022006. DOI: 10.1088/1742-6596/1237/2/022006. URL: https://doi.org/10.1088%2F1742-6596%2F1237%2F2%2F022006.

[Loo10] Lincoln Loop. *Pure python QR Code generator*. https://github.com/lincolnloop/python-qrcode. 2010.

[LT12] Jukka Lehtosalo and Mypy Team. *Mypy - An optional static type checker for Python*. http://mypy-lang.org. 2012.

[LWW15] K. Lay, L. Wang, and C. Wang. "Rectification of QR-Code Images Using the Parametric Cylindrical Surface Model". In: *2015 International Symposium on Next-Generation Electronics (ISNE)* (2015), pp. 1–5.

[LZ17] Kuen-Tsair Lay and Ming-Hao Zhou. "Perspective projection for decoding of QR codes posted on cylinders". In: *2017 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)* (2017), pp. 39–42.

[Mus16] London Natural History Museum. *pyzbar - Python wrapper for ZBar*. https://github.com/NaturalHistoryMuseum/pyzbar. 2016.

[OST08] Sean Owen, Daniel Switkin, and ZXing Team. *ZXing ("zebra crossing")*. https://github.com/zxing/zxing. 2008.

[RRR07] Andrés Raya, Alfonso Ríder, and Rafael Rubio. *Álgebra y Geometría lineal*. Barcelona: Editorial Reverté, 2007.

[Sil+20] Steven Silvester et al. *imageio/imageio v2.6.1*. Version v2.6.1. Feb. 2020. DOI: 10.5281/zenodo.3674137. URL: https://doi.org/10.5281/zenodo.3674137.

[Sou09] SourceForge. *ZBar*. http://zbar.sourceforge.net/. 2009.

[VCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), p. 22.

[Vir+20] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.

[Wal+14] Stéfan van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: https://doi.org/10.7717/peerj.453.

[WG06] Mark Whitbeck and Hongyu Guo. "Multiple Landmark Warping Using Thin-plate Splines." In: Jan. 2006, pp. 256–263.