Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingenieria y Agrimensura
Laboratorio de Automatización y Control
Procesamiento Digital de Imágenes

Trabajo Practico Final

Detección de códigos QR aplicando técnicas de procesamiento de imágenes

Integrantes: Emanuel Cima
Diego Sarina
Profesores: Dr. Juan Carlos Gomez
Dr. Gonzalo Sad

Fecha de entrega: 4 de Marzo de 2020
Rosario, Argentina

Resumen

El siguiente informe, refleja el TPF (Trabajo Practico Final) realizado para la cátedra de Procesamiento Digital de Imágenes (DIP). El mismo consta del detalle de la implementación de un software para la detección de Códigos QR aplicando técnicas de procesamiento de imágenes. Se detalla desde la estructura del código QR, cual es el principal método de detección actual y el planteo de nuestra solución.

Cabe destacar que no hemos implementado la decodificación del código, dado que el procedimiento para hacerlo excede los limites del trabajo. Para realizar la decodificación del mismo, nos basamos en una librería de terceros.

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Introducción

Los códigos QR son ampliamente utilizados hoy en día. Por ejemplo, los teléfonos inteligentes y muchos otros dispositivos pueden ser utilizados como escáneres de estos códigos, para poder extraer datos del mismo o para generar un código QR con cierta información propia.

Considerando la diversidad de aplicaciones en las que actualmente se los utiliza, resulta de gran interés el estudio de los mismos.

1.1. Trasfondo del proyecto

La razón para realizar este proyecto es para tener una mejor compresión de la extracción de las características de la imagen y la transformación de la perspectiva. En nuestra investigación durante el desarrollo del proyecto, notamos que según la norma bajo la cual se rige la generación de los QR, los mismos no utilizan técnicas de procesamiento de imágenes para la detección de este (esto se explicara mas adelante).

Entonces, a partir de lo mencionado anteriormente, se propone utilizar métodos vistos en el curso de DIP para lograr detectar los marcadores de un código QR. Finalmente al obtener los mismos y procesarlos, junto con la ayuda de una librería se procede a la decodificación del mismo.

1.2. Consideraciones

Este proyecto esta dividido en dos partes. Primero se abordara el marco teórico y luego se tratara el desarrollo del software en Python.

El marco teórico abordara primero los patrones estándar de codificación y generación de códigos QR utilizados hoy en día, y como se realiza la decodificación de los mismos.

Luego se introducirá nuestra propuesta con la que hemos elaborado el software, que a grandes rasgos consisten en tomar una imagen de la cámara, reconocer los marcadores de la misma (de estar presentes), corregir la perspectiva, binarizarla y finalmente enviar la imagen binaria a la librería de terceros para la extracción del mensaje codificado en el QR. Finalmente se explicaran los algoritmos implementados para una mejor comprensión de los mismos.

2. Código QR

En esta sección, se explica la estructura de los códigos QR y su procedimiento de detección. Esto esta basado en la versión estándar de la norma ISO/IEC 18004-2006.

2.1. Estructura general

El código QR (del ingles “quick response”) es la evolución del código de barras. Si bien los códigos de barras unidimensionales se utilizaron durante décadas, en la década de 1990 surgió la necesidad de un código fácil de leer que pudiera almacenar mas información. Esto llevo al desarrollo de varios códigos bidimensionales. Uno de estos es el código QR desarrollado en Japón. Su principal ventaja sobre sus competidores de la época, era el hecho de que sus creadores pusieron énfasis en la velocidad y precisión de la detección y su lectura.

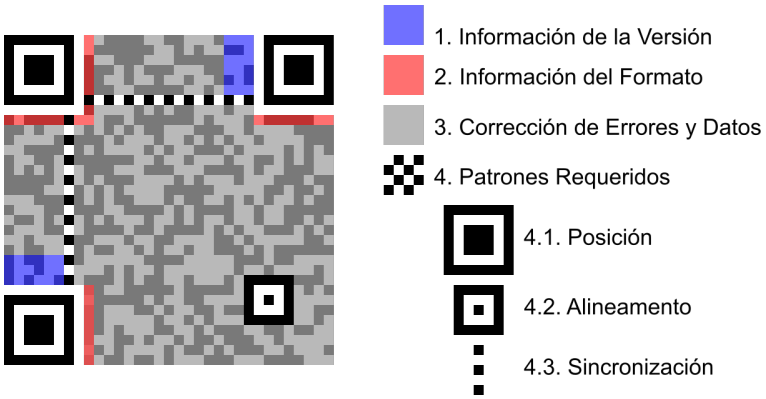

Figura 1: Estructura de un Código QR [4]

Para lograr esto se propusieron desarrollar un patrón de marcador especial que seria fácilmente detectable y no se prestaría a falsos positivos en la detección. Luego de una investigación realizada por los mismos, determinaron que la proporciones menos utilizada de negro y blanco en un material impreso es la que contenía la relación: “1:1:3:1:1”. Así fue como se decidieron los anchos de las áreas blancas y negras contenidas en el patrón de posición. En este sentido, el código QR fue creado de forma que su código pueda ser determinado independientemente del ángulo de escaneo, el cual puede ser cualquiera dentro de los 360º buscando únicamente el patrón (“1:1:3:1:1”).Los marcadores resultantes (vistos en 1) se colocan en las tres esquinas del código QR, lo que al detectarlos nos dará información sobre la posición y la orientación del código.

Además de los patrones de detección de posición, cada código QR contiene varias características para facilitar la detección y decodificación. En el código QR original, un patrón de alineamiento también esta presente. A medida que el QR contiene mas información o se modifica su tamaño (haciéndose mas grande), el patrón de alineamiento se va replicando y estos ayudan a rectificar distorsiones de naturaleza no lineal, como lo es la distorsión cilíndrica. Otras áreas del código están reservadas para la información de la versión, información del formato y corrección de errores y datos. Un patrón de tiempo “timing pattern” compuesto por módulos blancos y negros alternantes esta presente también para ayudar en la detección del tamaño del modulo. La zona alrededor de un código QR tiene una llamada “zona silenciosa” que permite una detección mas fácil.

Por ultimo, los códigos QR tienen varias variantes que difieren por su robustez en términos de

poder leerse incluso con errores de escaneo. En su versión mas robusta, el 30 % del código puede estar corrupto y la información almacenada puede aun ser recuperada. Esto se logra mediante el algoritmo de corrección de errores “Reed-Solomon”. Obviamente,que la información redundante introducida en el código significa que se pueden almacenar menos datos en general. Finalmente, esta capacidad de ser leído y evitar errores en la decodificación de los mismos, ha llevado a los creadores del código QR a incluir elementos artísticos en sus códigos como se visualiza en (vistos en 2)

(a) QR Whatsapp (b) QR Twitter (c) QR Steve

Figura 2: Códigos QR con elementos artísticos, pero todavía legibles.

2.2. Especificación del algoritmo de detección

En la especificación de las norma ISO 18004-2006, el algoritmo para la detección propuesto es:

- Determina un umbral global (“global threshold”) tomando un valor de reflectancia a medio camino entre la máxima reflectancia y la mínima presente en la imagen. Convierte la misma a una imagen binaria aplicando el umbral global.
- Localiza el patrón de posición. El mismo esta formado por tres patrones idénticos ubicados en tres de las cuatro esquinas del símbolo. Como se describe en la figura a continuación, los anchos del modulo en cada patrón de posición forman una secuencia “oscura - clara - oscura -clara - oscura” donde los anchos relativos de cada elemento están en las proporciones 1:1:3:1:1.

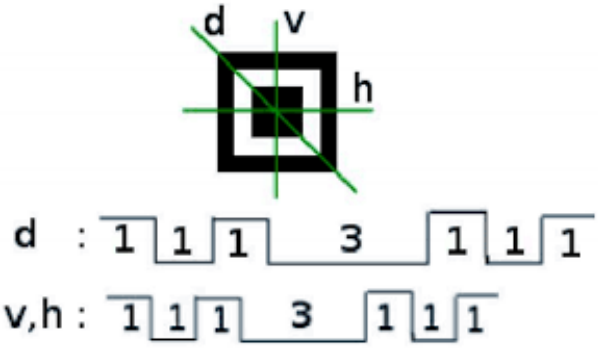

Figura 3: Relación existente en un patrón de posición

1. Cuando se detecta una zona candidata, tome nota de la posición del primer y último punto A y B, respectivamente, en los que una línea de píxeles de la imagen se encuentra con los

bordes exteriores del patrón del posición. Repita esto para las líneas de píxeles adyacentes en la imagen hasta que se hayan identificado todas las líneas que crucen la caja central del patrón de posición en el eje x de la imagen.

2. Repita el paso anterior para las columnas que cruzan al patrón en el eje y de la imagen

3. Localice el centro del patrón. Construya una línea a través de los puntos medios entre los puntos A y B en las líneas de píxeles más externas que cruzan la caja central del patrón del buscador en el eje x. Construya una línea similar a través de los puntos A y B en las columnas de píxeles más externas que cruzan la caja central del patrón del eje y. El centro del patrón se encuentra en la intersección de estas dos líneas.

4. Repita los pasos 1 y 3 para localizar los centros de los otros dos patrones de posición.

5. Si no se detectan zonas candidatas, se invierte la imagen y se comienza desde el principio nuevamente.

3. Planteo de la solución

En esta sección, se explicara la propuesta elegida para la obtención del Código QR aplicando técnicas vistas en la cátedra de Procesamiento de Imágenes, se dará una explicación porque fueron elegidas las mismas y como se implementaron.

3.1. Análisis propuesta implementada. Algoritmo de detección

Cuando empezamos a pensar en como llevar a cabo la resolución de la lectura del código QR, notamos que los algoritmos utilizados comúnmente para realizar esta tarea en su mayoría no aplican técnicas de procesamiento de imágenes. Al darnos cuenta de esta situación, tuvimos que plantear el problema de raíz.

Nuestra propuesta esta basada en tratar a los patrones de posición como contornos en concreto y determinar que los mismos cumplen con una jerarquía determinada. Es decir, que al momento de detectar los contornos de una imagen, nuestro objetivo es encontrar cinco contornos anidados que sean representativos de un posible patrón de posición. Si bien inicialmente un patrón de posición contiene tres contornos anidados; como los limites internos también cuentan, entonces necesitamos buscar cinco. Esto puede verse en la figura a continuación:

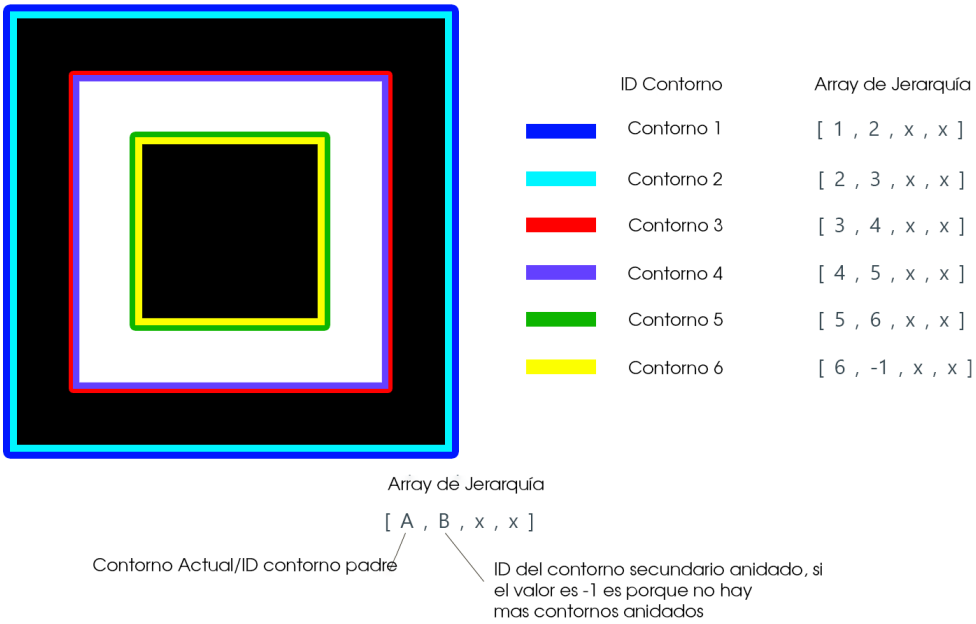

Figura 4: Patrón de Posición de un QR

Continuando con lo mencionado en el párrafo anterior, sabiendo como es la jerarquía de nuestro patrón, se pretende analizar la imagen y encontrar tres posibles candidatos. Tomando como referencia siempre la orientación **Norte** se establecen cuales son los puntos A, B y C. Luego se necesita determinar cual es cada uno correctamente, para ello, se calculan las distancias que unen los puntos entre si. De la comparación de estas distancias, se obtiene cual es el punto distante, es decir el punto C. Esto se puede apreciar en la figura 5:

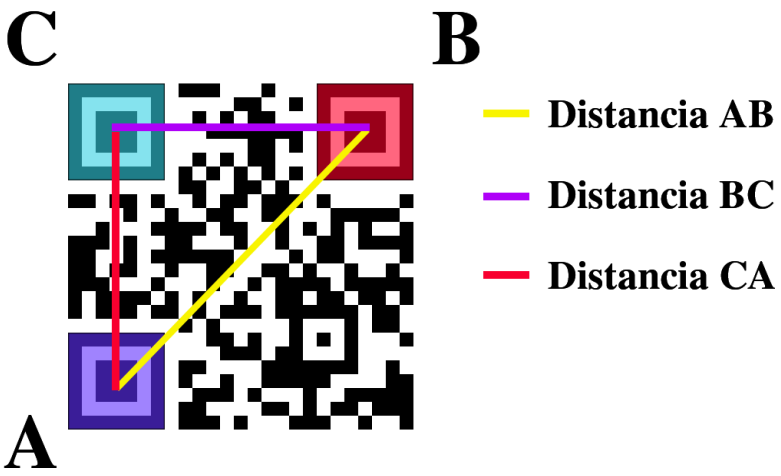

Figura 5: Rectas para calculo de distancias entre puntos A, B y C

Una vez obtenidos la posición correcta de los marcadores, procedemos a calcular la orientación del mismo, dado que consideramos que al momento de leer el QR, este puede estar en cualquiera de sus cuatro orientaciones (Norte, Sur, Este, Oeste) como se visualiza en la figura 6

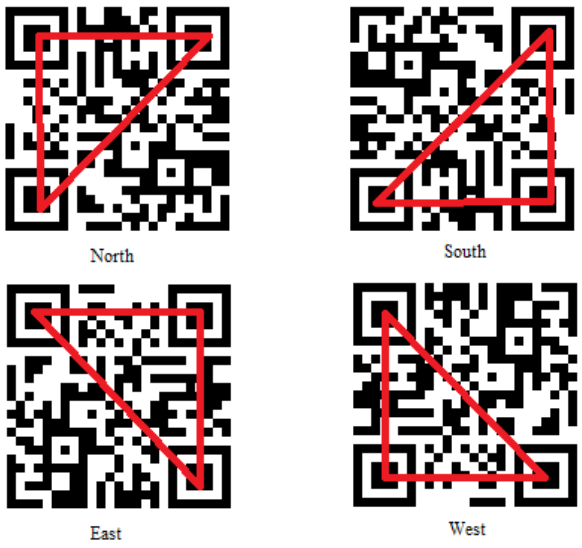

Figura 6: Distintas orientaciones del código QR

Conociendo la orientación del QR, procedemos a obtener los vértices de cada punto (A, B y C) llamaremos a los mismos **L, M, O** para luego estimar el punto restante **N**. Este ultimo punto se calcula a partir de las rectas que se forman entre el punto O y el punto M. 7

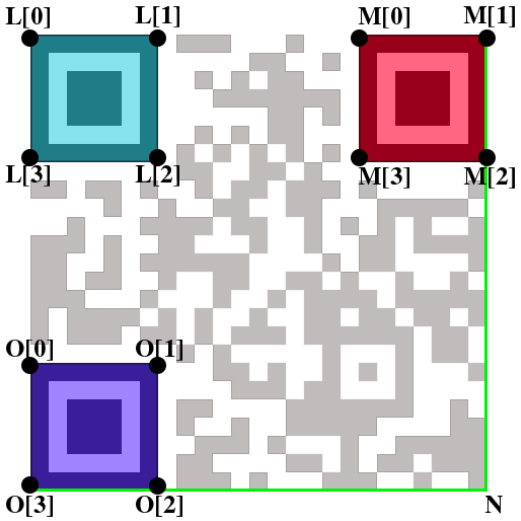

Figura 7: Puntos L, M, O y N.

Ya en este punto con nuestros vértices calculados procedemos a obtener la matriz de transformación y aplicamos una corrección de la perspectiva. Teniendo nuestra imagen con la perspectiva corregida, binarizamos la misma y por ultimo con la ayuda de una librería decodificamos el mensaje que se encuentra en el QR y lo informamos por pantalla al usuario.

3.2. Desarrollo de la propuesta

Como se vio en la sección anterior, la resolución del problema conlleva varias etapas. Para una mejor comprensión de la misma, se elaboro el flujograma correspondiente que se encuentra debajo.

Algo a destacar en el desarrollo, fue que inicialmente realizamos todo el aplicativo para trabajar con una imagen y luego extendimos el código para poder procesar vídeo. Lo que nos trajo ciertas situaciones que tuvimos que resolver, y no habían sido tomadas en cuenta inicialmente.

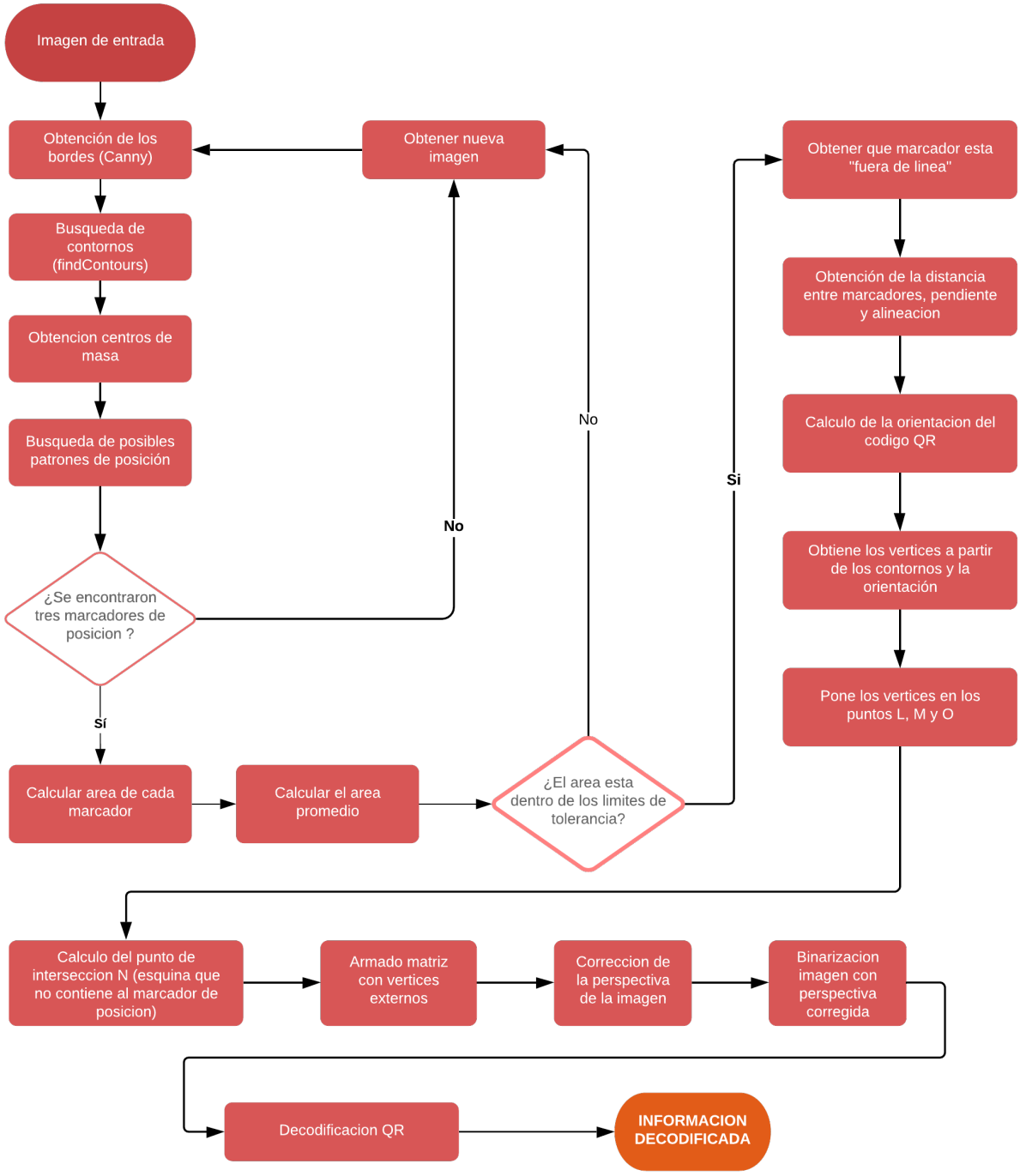

Figura 8: Flujograma solución desarrollada [3]

3.2.1. Interfaz inicial

Por cuestiones de simplicidad, se opto por emplear una interfaz de consola de linea de comandos (CLI). Esto trae consigo ciertas ventajas como lo son:

- Mejor performance al momento de la ejecución

- Implementación mas rápida
- Mayor personalización de los comandos
- Etc.

El software esta diseñado para que trabaje bajo dos modos principales, que son: **Modo Video** o **Modo Imagen**. Luego el modo Video admite utilizar un dispositivo externo (como lo es una cámara web) para procesar en tiempo real, o utilizar un archivo de video que se encuentre en el dispositivo. También se incorporo un modulo "verboso" para realizar el debug correspondiente, y poder tener en pantalla las fases del procesamiento. En la figura 9 se adjuntan la interfaz inicial, los ambos modos correspondientes y sus respectivas opciones.

```
usage: QRScanner [-h] {video,image} ...

qr decode application

positional arguments:
  {video,image}  data input sources

optional arguments:
  -h, --help     show this help message and exit
```

(a) Interfaz Inicial

```
usage: QRScanner video [-h] [-v] [-t THRESH] [-d DEVICE | -f FILENAME] [-c]

input data from video

optional arguments:
  -h, --help        show this help message and exit
  -v, --verbose     verbose mode
  -t THRESH         binarization thresh level
  -d DEVICE         id of the video capturing device (i.e. a camera index)
  -f FILENAME       path of the video file (eg. /home/video.avi)
  -c, --continuous  continuous detection
```

(b) Interfaz Modo Video

```
usage: QRScanner image [-h] [-v] [-t THRESH] [-p PATH]

input data from image

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  verbose mode
  -t THRESH      binarization thresh level
  -p PATH        image path
```

(c) Interfaz Modo Imagen

Figura 9: Interface, Modos y Opciones

Dado que los modos de funcionamiento son similares (salvo ciertas particularidades de implementación que son propias de cada modo), el núcleo del procesamiento es el mismo. Dicho esto, optamos por continuar utilizando el **Modo Imagen** en el resto del informe.

3.2.2. Adquisición imagen inicial

Comenzamos cargando a nuestro aplicativo la imagen con la que trabajaremos, dicha imagen pertenece al conjunto de imágenes disponibles que se encuentran en el directorio "applica-

tion/qr_images". Para este caso utilizamos la imagen **2.jpeg**, la misma se visualiza en la figura 10.

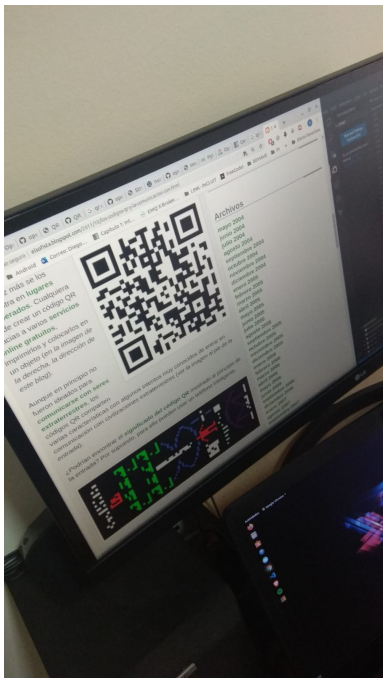

Figura 10: Imagen con código QR a descifrar.

3.2.3. Obtención de bordes

Con nuestra imagen de entrada cargada, procedemos a aplicar una detección de bordes mediante la función **Canny** de openCV. En este punto es importante decidir cuales de todos los bordes son realmente bordes y cuales no. Para realizar esto, tuvimos que configurar dos umbrales que son: *minVal* y *maxVal*. Entonces cualquier borde con un gradiente de intensidad mayor que *maxVal* sera tomado como un borde seguro y aquellos que estén debajo de *minVal* no serán tomados en cuenta. Por lo tanto, quienes se encuentren entre medio de estos umbrales y estén conectados a bordes seguros, serán tomados en cuenta. Esto puede visualizarse en la figura 12

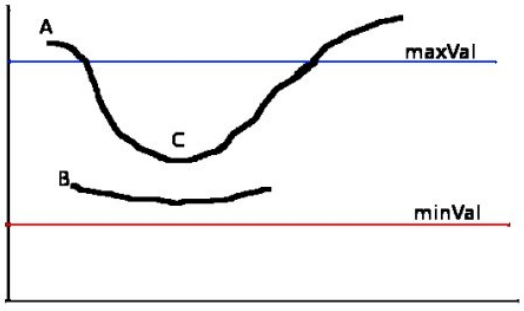

Figura 11: Histéresis detector Canny.

Para nuestro caso, los umbrales elegidos fueron de "20" y "200" tal como se visualiza en el código

de implementación que se adjunta a continuación. Además al utilizar *L2gradient* le indicamos al openCV que utilice la ecuación completa en el calculo del *gradiente de intensidad* en vez de utilizar una aproximación del mismo.

Código 1: Implementación detección de bordes.

```
1               def image_edges(frame):
2                   gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
3                   edges = cv2.Canny(gray, 20, 220, (9, 9), L2gradient=True)
4                   return edges
```

Luego de aplicar esta detección, obtenemos el siguiente resultado:

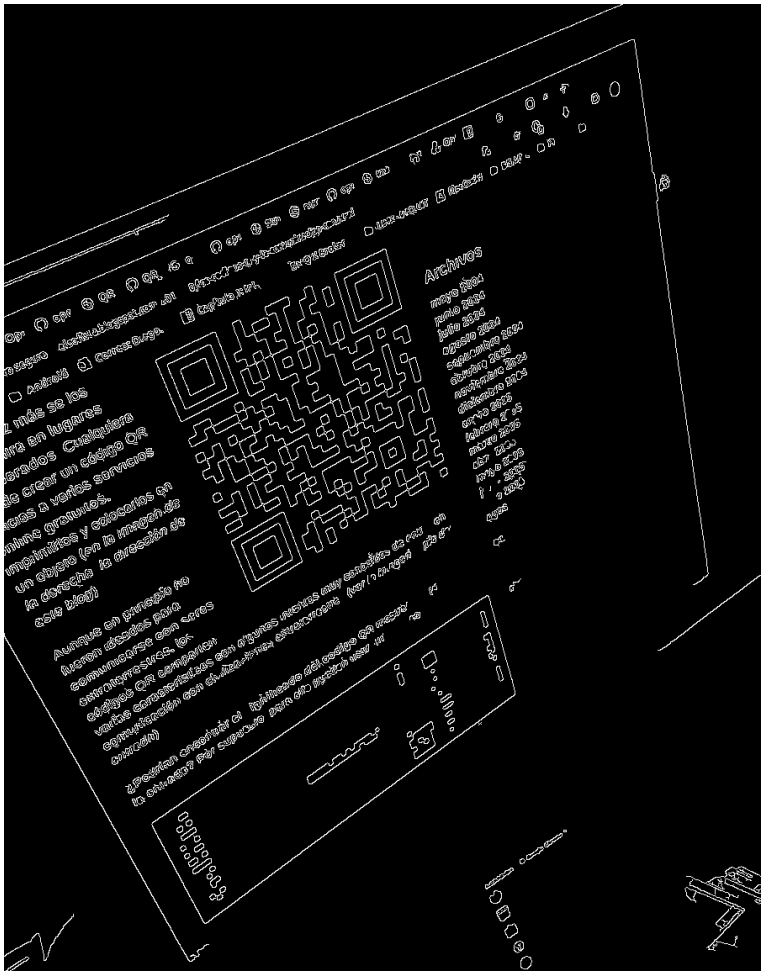

Figura 12: Obtención de bordes.

3.2.4. Búsqueda de contornos

Ahora procedemos a calcular los contornos presentes en la imagen. Para ello utilizamos la función de **findContours**. La ventaja que tenemos en la utilización de esta función es que la misma nos permite generar una matriz de herencia la cual es fundamental al momento de determinar el patrón

de posición.

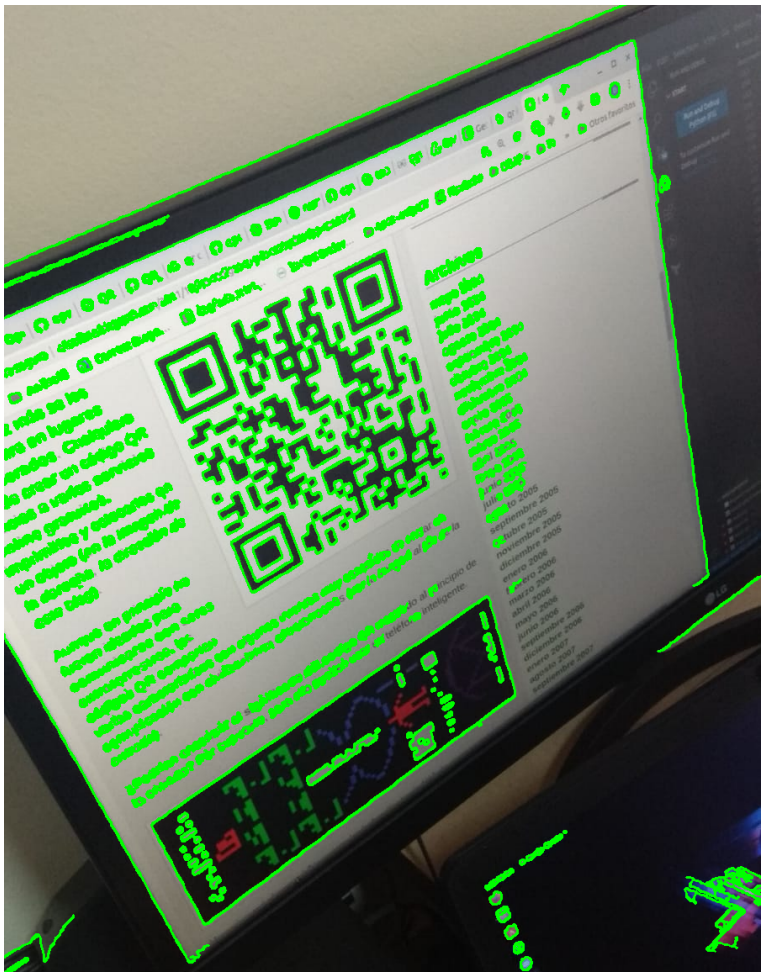

Figura 13: Ploteo de todos los contornos encontrados en la imagen.

Luego de la obtención de los contornos, calculamos los centros de masas de estos. Para llevar a cabo esa tarea, nos valemos de la ayuda de la función **moments**. La misma devuelve un diccionario con todos los valores de los momentos calculados. A partir de estos, se pueden extraer datos útiles como el área, centroide, etc. En nuestro caso calcularemos el centroide cuya relacion es la siguiente:

$$C_x = \frac{M_{10}}{M_{00}} \tag{1}$$

$$C_y = \frac{M_{01}}{M_{00}} \tag{2}$$

3.2.5. Búsqueda de los patrones candidatos

Con la información obtenida a partir de la búsqueda de contornos, procedemos a realizar la búsqueda de los patrones candidatos. Como se planteo al principio de la sección 3.1 nuestro patrón de posición cumple con una jerarquía determinada. Conociendo la misma, se realizo un función

para encontrar que contornos cumplen con la misma y de esa forma catalogarlos como candidatos.

A grandes rasgos la función recibe como parámetros a la matriz de *contornos* y de *jerarquía*, luego se itera sobre cada contorno, realizando las siguientes acciones:

- Aproximación de contorno: se aproxima una forma de contorno a otro con menos vértices, dependiendo de la precisión que le especifiquemos. En nuestro caso, tratamos de encontrar un cuadrado en un imagen, pero podríamos obtener una "mala forma". Con el uso de *approxPolyDP* podemos aproximar la forma. Luego si la misma cumple con tener cuatro vértices se prosigue operando sobre ese contorno.
- Análisis de la jerarquía: si *len(approx) == 4*, iteramos sobre ese contorno preguntando en nuestra matriz de jerarquía si el índice del contorno para el cual estamos trabajando es diferente de **"-1"**, lo que indicaría que ese contorno ya no tiene contornos anidados.
- Finalmente si se cumple la condición de que se han encontrado cinco contornos anidados, procedemos a apendear el ID del contorno que ha cumplido con nuestras reglas.

Código 2: Implementación detección de patrones candidatos.

```
def search_candidate_markers(contours, hierarchy):
    marks_candidates = []
    for i, contour in enumerate(contours):
        approx = cv2.approxPolyDP(
            contour,
            cv2.arcLength(contour, True) * 0.02,
            True
        )
        if len(approx) == 4:
            k = i
            cc = 0
            while (hierarchy[0, k, 2] != -1):
                k = hierarchy[0, k, 2]
                cc += 1
            if (hierarchy[0, k, 2] != -1):
                cc += 1
            if cc >= 5:
                marks_candidates.append(i)
    return marks_candidates
```

3.2.6. Búsqueda del patrón fuera de linea

Luego de obtener nuestros patrones de posición, necesitamos determinar cual de ellos es el que esta alejado de la recta formada por los otros dos patrones. En términos mas sencillos, tomando como referencia la orientación norte, el patrón que queda "fuera de linea" es el patrón **C**, tal como se puede visualizar en la figura 5. Para esto calculamos las distancias de las rectas *AB, BC y CA*. Esta distancia se computa como la diferencia entre los centros de masas de los patrones; finalmente podemos obtener cual no pertenece a la recta.

3.2.7. Calculo de la orientación del QR

Como nuestro QR puede venir en cualquier orientación, necesitamos poder determinarla. Conociendo el valor de la pendiente y el signo de la distancia calculados previamente [1]. Podemos discriminar los cuatro casos posibles: *Norte, Sur, Este, Oeste.*

Del signo de la distancia, sabemos que si la misma posee signo negativo estamos frente a los casos **Norte y Este** y si es positiva **Sur o Oeste**. Luego con la ayuda de la pendiente terminamos de definir la orientación, resultando que la orientación del QR queda determinada bajo estas reglas:

- Orientación Norte: distancia y pendiente ambas negativas
- Orientación Sur: distancia positiva y pendiente negativa
- Orientación Este: distancia negativa y pendiente positiva
- Orientación Oeste: distancia y pendiente ambas positivas

3.2.8. Calculo de los vértices de los patrones de posición

En este punto, por cada patrón de posición localizado, debemos de encontrar sus correspondientes vértices. Para realizar esto, comenzamos calculando el *Bounding Box* de nuestro patrón, cuyos vértices extremos se encuentran representados por color azul en la figura 14. A partir de los datos del mismo, procedemos a calcular cual el centro, es decir la mitad en X y mitad en Y. De esta forma, podemos analizar la imagen como si la misma tuviera cuatro cuadrantes.

Con esto, calculamos para cada punto que pertenece al contorno la distancia del mismo al vértice contenido en su cuadrante. Con la finalidad de obtener que punto se encuentra a la menor distancia. En la figura que se encuentra debajo, los puntos que han sido seleccionados como vértices del patrón son los puntos rojos de mayor tamaño (excluido el centro).

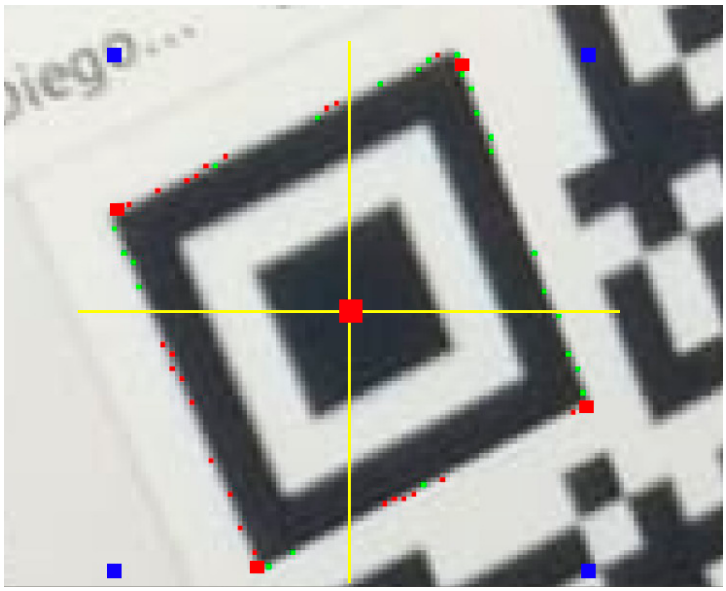

Figura 14: Calculo de los vértices de un patrón de posición.

[1] No se muestra en este informe el calculo de la pendiente, ni de la distancia. Ambas se encuentran respectivamente calculadas en el código

3.2.9. Determinación del punto N (vértice restante del QR)

El punto **N** es nuestro cuarto vértice del código QR. El mismo, fue calculado a partir de la intersección de dos lineas formadas por los puntos:

$$(P_0; P_1) = ((O_2; O_3); (M_1; M_2)) \tag{3}$$

Tal como se visualiza en la figura 7. La implementación del calculo de este, se es:

Código 3: Calculo del punto N.

```
def intersection_between_two_lines(line1_p1, line1_p2, line2_p1, line2_p2):
    delta_x = (line1_p2[0] - line1_p1[0], line2_p2[0] - line2_p1[0])
    delta_y = (line1_p2[1] - line1_p1[1], line2_p2[1] - line2_p1[1])
    def det(a, b):
        return a[0] * b[1] - a[1] * b[0]
    determinant = det(delta_x, delta_y)
    if not determinant:
        raise PointIntersectionError('lines do not intersect')
    d = (det(line1_p2, line1_p1), det(line2_p2, line2_p1))
    x = float(det(d, delta_x)) / determinant
    y = float(det(d, delta_y)) / determinant
    return np.int32([x, y])
```

A continuación se muestra una imagen generada a partir del calculo de los patrones de posición, rectas para el calculo de N y finalmente el punto N. De esta forma podemos tener una primera aproximación si hemos trackeado correctamente la imagen de entrada.

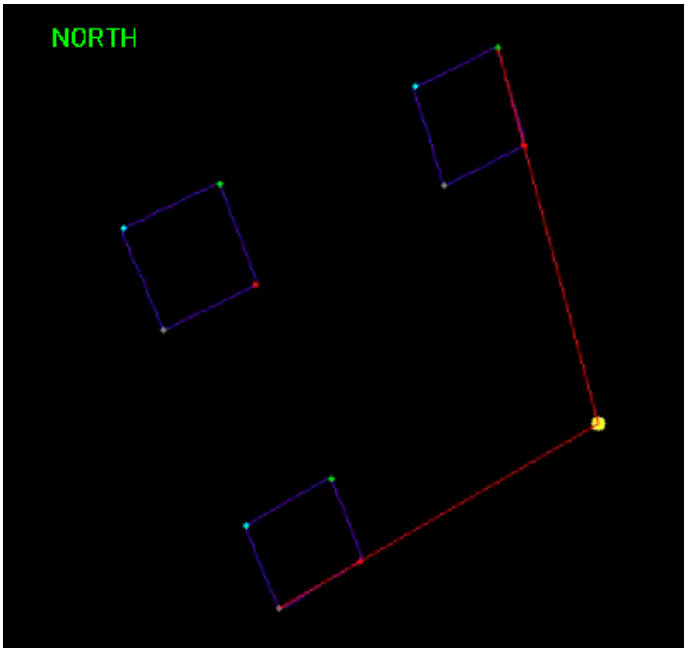

Figura 15: Imagen generada con datos previamente calculados [a]

[a] La generación de la imagen se realiza computando los vértices de cada patrón de posición, las rectas conformadas por los puntos P0 y P1 y el calculo del punto N

3.2.10. Corrección de la perspectiva

Ya sobre el final del algoritmo, necesitamos corregir la perspectiva de la imagen, para poder recortar efectivamente nuestro QR de la imagen original. Para llevar esto a cabo, necesitamos primero calcular la matriz de transformación de perspectiva, esto lo hacemos con la función *cv2.getPerspectiveTransform (src, dst)*, donde ingresamos los vértices externos calculados anteriormente y la matriz de destino donde queremos llevar a cabo la transformación. Y luego mediante *cv2.warpPerspective* aplica la transformación de la imagen de entrada junto con la matriz de transformación obtenida anteriormente. Luego de aplicar la transformación, decidimos enmarcar nuestro código en un recuadro blanco de diez píxeles. Finalmente devolvemos la imagen que se deberá decodificar con la perspectiva corregida.

(a) Imagen de entrada.

(b) Imagen de salida.

Figura 16: Comparación entre imagen de entrada e imagen final luego de ser procesada.

3.2.11. Decodificación de la información

Tal como se planteo anteriormente, por cuestiones que exceden al curso no se realizo la decodificación manual de la información. Para llevar a cabo la misma, nos apoyamos en una librería de terceros llamada **pyzbar**, donde la misma toma como parámetro de entrada la imagen final resultante del procesamiento y nos devuelve la información contenida en la misma.

3.3. Benchmark del aplicativo

Para poder tener una idea si nuestra aplicativo se asemejaba a otros softwars o librerías existentes. Decidimos realizar una pequeña prueba de ejecución, y medir el tiempo que le lleva a la misma poder decodificar el QR.

El escenario de prueba fue nuestro código en el *Modo Imagen* frente a la librería *pyzbar* tomando ambos códigos como referencia la imagen **2.jpeg** que se utilizo en el desarrollo de este informe.

Se procedió a iterar 1000 veces sobre ambos, obteniendo los siguientes resultados:

- Código personal: 76 segundos
- Pyzbar: 114 segundos

El código contra el que se comparo el nuestro es el que se adjunta debajo:

Código 4: Pyzbar implementación.

```
        def qr_decoder(qr_codificado):
            decoded_data = None
            qrs = pyzbar.decode(qr_codificado)
            for qr_ in qrs:
                decoded_data = qr_.data.decode("utf-8")
            return decoded_data
        if __name__ == "__main__":
            str_ = '''
            qr_decoder(image)
            '''
            stp = '''
            from qr_zbar import qr_decoder
            import cv2
            image = cv2.imread("./qr_images/2.jpeg")
            '''
            str_ = textwrap.dedent(str_)
            stp = textwrap.dedent(stp)
            print(timeit.timeit(str_, number=1000, setup=stp))
```

Podemos notar que nuestra implementación ha superado a la libaría convencional, agilizando la detección del QR y logrando un incremento de la rapidez de hasta un **66,6 %**.

4. Conclusiones

4.1. Conclusiones Generales

Se logro llegar a una solución acorde a las exigencias pedidas por la cátedra. Donde se pudieron emplear los métodos vistos en clases. Personalmente nos pareció un practico muy enriquecedor dado que hemos elaborado una solución nueva, que no esta implementada y pudimos establecer la performance de la misma mediante el benchmark realizado en la sección 3.3, la cual fue satisfactoria.

Referencias

[1] Template Informe en LaTeX. *¡Revisa el manual online de este template!*
https://latex.ppizarror.com/informe

[2] Overleaf. *Uno de los mejores editores online para LaTeX, renovado con su versión 2.0.*
https://es.overleaf.com

[3] Lucidchart. *Software de diagramacion en linea y solucion visual.*
https://www.lucidchart.com/pages/es

[4] Wikipedia. *Estructura codigo QR.*
https://es.wikipedia.org/wiki/Código_QR

[5] Canny Edge Detection. *Canny Edge Detection Thory.*
https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html

[6] Moments. *Image Moments.*
https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_contours/py_contour_features/py_contour_features.html?highlight=moments

[7] Emathzone. *Distance of a Point from a Line.*
https://www.emathzone.com/tutorials/geometry/distance-of-a-point-from-a-line.html

[8] Wikipedia. *Intersection of two lines.*
https://en.wikipedia.org/wiki/Line-line_intersection

[9] Argparse. *Parser for command-line options, arguments and sub-commands.*
https://docs.python.org/3/library/argparse.html

[10] Pyzbar. *Pyzbar library.*
https://pypi.org/project/pyzbar/