University of East London

Ain Shams University

Faculty of Engineering

AIN SHAMS UNIVERSITY

FACULTY OF ENGINEERING

CREDIT HOURS ENG. PROGRAM

Computer engineering and

1839 2019 180 YEARS Engineering Science Leaders since 1839

CSE483

CV Project

QR Code Detector

Submitted to:

Prof. Mahmoud Khalil

Tut. Ahmed Salama

Submitted by:

Mohammed Elhag Mohammed Mahmoud	19P1472
Ali Mohamed Hesham	19P3462
Mohamed Hesham El Said Zidan	20p7579

CONTENTS

INTRODUCTION

OVERVIEW OF QR CODE TECHNOLOGY

QR (Quick Response) codes are a type of two-dimensional barcode that consists of black squares arranged on a white square grid. Developed in 1994 by Denso Wave, a subsidiary of Toyota, QR codes were initially designed for tracking automotive parts during manufacturing. However, their versatility and ability to store large amounts of data quickly led to widespread adoption across various industries and applications.

STRUCTURE OF QR CODES:

QR codes can store information horizontally and vertically, allowing for efficient data encoding. They typically contain three square-shaped corner patterns that enable scanners to detect and align the code correctly. The main components of a QR code include:

- Finder Patterns: These are large square patterns located in three corners of the QR code, used for locating and aligning the code during scanning.
- Alignment Patterns: Small square patterns located near the center of the QR code, used for correcting distortions and ensuring accurate scanning.
- Timing Patterns: These are thin black and white lines surrounding the QR code modules, used for synchronizing the scanning process.

ENCODING DATA

QR codes can encode various types of data, including URLs, text, contact information, Wi-Fi credentials, and more. The amount of data that a QR code can store depends on its version and error correction level. Higher versions of QR codes can store more data but require larger physical dimensions.

SCANNING AND DECODING

Scanning a QR code involves using a smartphone camera or a dedicated QR code scanner to capture an image of the code. The scanning device then processes the image, detects the QR code pattern, and decodes the encoded data. QR codes offer a quick and convenient way to access information, enabling seamless interactions with digital content, products, and services.

PURPOSE AND SCOPE OF THE REPORT

The purpose of this report is to provide a detailed overview of QR code processing algorithms and their implementation. The report aims to:

- ✓ Describe QR Code Processing Techniques: It will explain the methodologies and techniques used for preprocessing images, detecting QR code contours, perspective transformation, masking, and data extraction.

- ✓ Discuss Implementation Details: The report will delve into the implementation details of the QR code processing algorithm, including code structure, organization, and libraries/tools used.

- ✓ Present Results and Analysis: It will showcase results obtained from processing sample input images containing QR codes. Additionally, it will discuss the strengths, limitations, and potential areas for improvement of the algorithm.

BACKGROUND

IMPORTANCE OF QR CODE PROCESSING

QR code processing plays a crucial role in extracting information from QR codes embedded in images or documents. Efficient processing techniques are essential for accurately detecting and decoding QR codes, especially in scenarios where the codes may be distorted, partially obscured, or embedded within complex backgrounds.

The ability to process QR codes has significant implications quickly and accurately across various industries and applications. For example, in retail and marketing, QR codes enable seamless access to product information, promotions, and online content. In logistics and inventory management, QR codes facilitate tracking, authentication, and inventory control. Moreover, QR codes are increasingly used in digital payments, ticketing, authentication, and identity verification processes.

PREVIOUS APPROACHES TO QR CODE DECODING

Over the years, numerous approaches have been developed for decoding QR codes, ranging from simple algorithms to advanced computer vision techniques. Traditional methods often rely on detecting QR code patterns and applying predefined rules to extract the encoded data. These methods may include thresholding, edge detection, contour analysis, and template matching.

In recent years, advancements in computer vision, image processing, and machine learning have led to the development of more robust and accurate QR code decoding algorithms. Deep learning techniques, such as convolutional neural networks (CNNs), have shown promising results in detecting and decoding QR codes from images with varying levels of complexity, noise, and distortion.

Despite the progress in QR code decoding techniques, challenges remain, particularly in handling distorted or damaged codes, dealing with low-quality images, and ensuring real-time processing capabilities for applications requiring rapid response times.

METHODOLOGY

QR codes store information in a grid of black and white squares. To unlock this hidden data, our devices follow a clever multi-step process. First, they clean up the image and straighten it out. Then, they use special patterns within the code to identify how the data is hidden and protected from errors. Finally, they decode the information, transforming those black and white squares into readable text, website links, or other useful content.

PREPROCESSING STEPS

Image Acquisition: The QR code image is obtained from a camera or loaded from a file.

WHY GRAYSCALE CONVERSION MATTERS?

- ✓ ELIMINATING IRRELEVANT INFORMATION: QR code readers are primarily interested in the darkness levels of modules, not their colors. Grayscale conversion removes any color information present in the image, effectively discarding irrelevant data and simplifying the processing for the algorithm.
- ✓ FOCUS ON DATA-CARRYING ELEMENTS: With color information removed, the algorithm can focus solely on the variations in darkness within the grayscale image. This allows for a more accurate analysis of the modules and their corresponding data bits (0 or 1)

BINARIZATION: SHARPENING THE IMAGE FOR DECODING

THRESHOLDING FOR CLEAR DISTINCTION: Binarization takes the grayscale image and sets a specific darkness threshold. Any pixel darker than this threshold is converted to black (1), while anything lighter becomes white (0). This creates a high-contrast binary image where each module is definitively black or white.

ENHANCED ACCURACY: Binarization improves the clarity of the image by removing ambiguity between light and dark shades of gray. This ensures the decoding algorithm can accurately interpret each module's darkness level and correctly assign the corresponding data bit (0 or 1).

ROBUSTNESS TO LIGHTING VARIATIONS: In real-world scenarios, lighting conditions can vary. Binarization helps overcome these variations by focusing on the relative darkness difference between modules. As long as the black modules remain significantly darker than the background under varying light, the threshold can effectively separate them, ensuring accurate data extraction.

COUNTER DETECTION AND PERSPECTIVE TRANSFORMATION

CONTOUR DETECTION: The algorithm identifies the largest contour in the image, assuming it represents the outer border of the QR code.

CORNER DETECTION: Corners (typically three distinct points) are identified within the detected contour. These corners mark the reference points for perspective transformation.

PERSPECTIVE TRANSFORMATION: Based on the corner locations, the image is warped to create a de-skewed and straightened version, correcting for potential tilting of the QR code during capture.

MASK PATTERN AND ERROR CORRECTION LEVEL (ECL) DETECTION

QUIET ZONE REMOVAL: The outermost black border (Quiet Zone) is removed as it doesn't contain data.

MODULE SIZE DETERMINATION: The size of individual data modules within the QR code grid is calculated based on the first row or column.

DATA AND FUNCTION PATTERNS: The code analyzes specific patterns within the grid to identify the used Mask Pattern (for data masking correction) and Error Correction Level (for error correction during decoding)

MASKING AND DATA EXTRACTION:

MASKING REMOVAL: Based on the identified Mask Pattern, a bitwise XOR operation is applied to specific data modules in the grid to remove the masking applied during QR code generation.

FORMAT INFORMATION EXTRACTION: Dedicated format information bits are identified and decoded to reveal additional information about the code, like character encoding used for textual data.

DATA REGION IDENTIFICATION: Based on the QR code version and format information, the code locates the valid data regions within the grid, excluding areas like alignment patterns and timing markers used for error correction.

DATA DECODING PROCESS

ERROR CORRECTION: The Reed-Solomon error correction code is applied to any detected errors within the extracted data bits, ensuring data integrity.

DATA INTERPRETATION: Based on the format information, the code interprets the data bits according to the encoded data type (text, URL, contact information, etc.). This might involve character set conversion or specific formatting based on the data type.

IMPLEMENTATION DETAILS

LIBRARIES AND TOOLS

OPENCV (CV2): This library provides functions for image processing tasks like grayscale conversion, binarization, contour detection, and image manipulation.

NUMPY (NP): This library offers numerical operations and data structures useful for image processing calculations and data manipulation.

MATPLOTLIB (PLT) (potentially): This library might be used for visualization purposes, allowing the code to display intermediate results like identified contours or processed images.

CODE STRUCTURE AND ORGANIZATION

IMAGE PREPROCESSING FUNCTIONS: Functions like average_intensity, find_intersection, and visualize_intersections might be used for initial image analysis tasks.

IMAGE PIPLINING

In our QR code processing pipeline, we implemented a robust approach to image thresholding that adapts to varying lighting conditions. This section details the key steps involved (refer to the code snippet for function details):

GRAYSCALE CONVERSION (IMPLICIT):

As a first step, we implicitly assumed the input image is converted to grayscale. This simplifies the thresholding process as grayscale images deal with a single intensity channel (0-255).

HISTOGRAM ANALYSIS:

We employed a function to identify the unique pixel intensity values present in the grayscale image (`gray1`). This information helps us understand the distribution of brightness within the image.

We then calculated the image's histogram using OpenCV's `cv2.calcHist` function. The histogram provides a visual representation of how many pixels have each intensity value, aiding in selecting an appropriate threshold.

ADAPTIVE THRESHOLDING STRATEGY:

To handle images with different lighting conditions, we implemented an adaptive thresholding strategy. A function `is_Dark` (not shown here), analyzes the histogram or a specific threshold to classify the image as "dark" or not.

DARK IMAGES**:** If the image is classified as dark, we hypothesize that the QR code represents the darkest element. Therefore, we identified the darkest pixel intensity (`Darkest`) using NumPy's `np.min` function.

Subsequently, we applied thresholding using OpenCV's `cv2.threshold`. Pixels with intensity lower than (`Darkest`) were set to black (0), and everything else to white (255).

NON-DARK IMAGES: If the image is not classified as dark, we assumed a more balanced distribution of brightness with the QR code potentially having a distinct intensity. In this case, we calculated the average intensity value using a function `average_intensity` (not shown here) based on the histogram data.

Thresholding was then applied using `cv2.threshold` with the calculated average intensity as the threshold. Pixels below the average were set to black, and above to white.

INVERTED IMAGE HANDLING:

We incorporated error handling to account for potentially inverted images where the foreground objects appear black and the background white. This might occur due to uneven lighting or camera settings.

A function, *`is_inv`* (not shown here), likely checks for this inversion. If detected, the image was flipped using bitwise negation (`255 - gray1`). This ensured a consistent format (dark QR code on a light background) for subsequent processing stages in our pipeline.

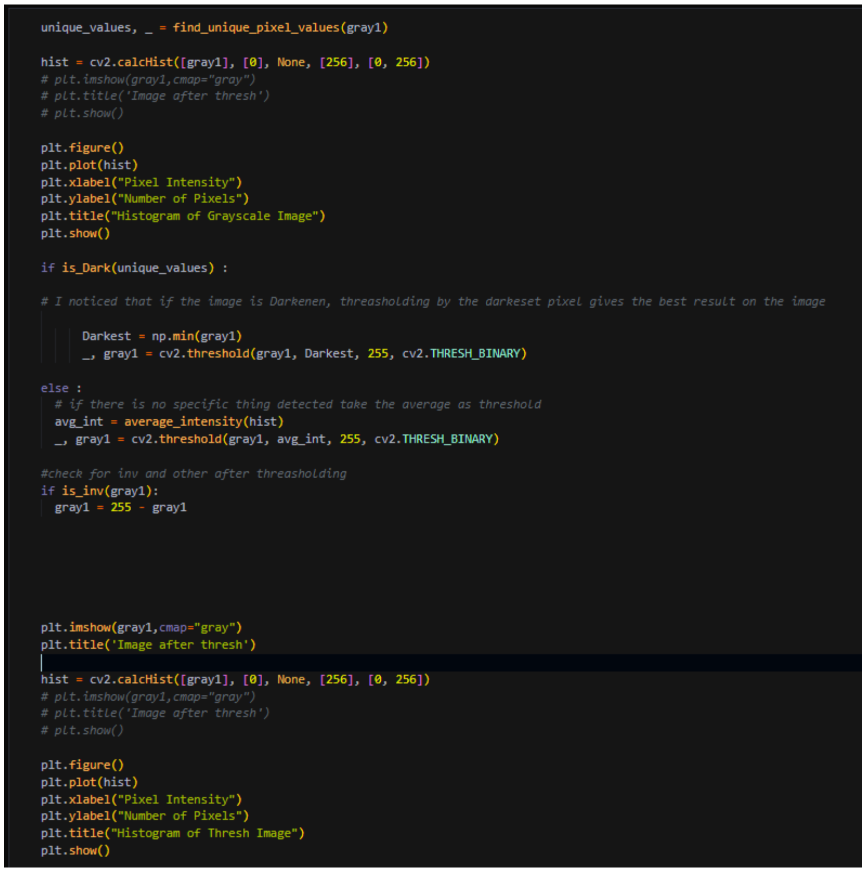
BENEFITS OF PIPELINED THRESHOLDING:

This pipelined approach to image thresholding offers several advantages:

ADAPTABILITY**:** By employing an adaptive strategy, our code can handle images with varying lighting conditions, improving robustness.

FLEXIBILITY**:** The code can adjust its thresholding approach based on the image characteristics (dark vs. balanced).

ERROR HANDLING**:** The inclusion of checks for inverted images ensures consistent data format throughout the pipeline.

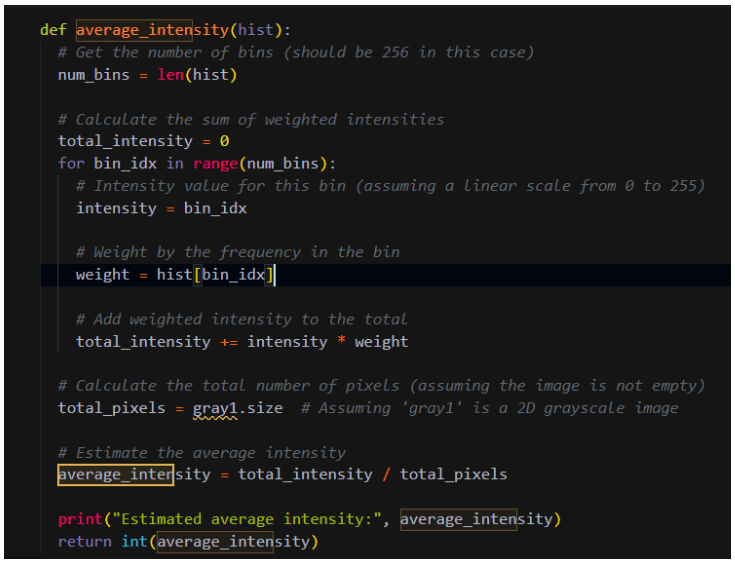


```python
unique_values, _ = find_unique_pixel_values(gray1)

hist = cv2.calcHist([gray1], [0], None, [256], [0, 256])
# plt.imshow(gray1,cmap="gray")
# plt.title('Image after thresh')
# plt.show()

plt.figure()
plt.plot(hist)
plt.xlabel("Pixel Intensity")
plt.ylabel("Number of Pixels")
plt.title("Histogram of Grayscale Image")
plt.show()

if is_Dark(unique_values) :

# I noticed that if the image is Darkenen, threasholding by the darkeset pixel gives the best result on the image

      Darkest = np.min(gray1)
      _, gray1 = cv2.threshold(gray1, Darkest, 255, cv2.THRESH_BINARY)

else :
  # if there is no specific thing detected take the average as threshold
  avg_int = average_intensity(hist)
  _, gray1 = cv2.threshold(gray1, avg_int, 255, cv2.THRESH_BINARY)

#check for inv and other after threasholding
if is_inv(gray1):
  gray1 = 255 - gray1





plt.imshow(gray1,cmap="gray")
plt.title('Image after thresh')

hist = cv2.calcHist([gray1], [0], None, [256], [0, 256])
# plt.imshow(gray1,cmap="gray")
# plt.title('Image after thresh')
# plt.show()

plt.figure()
plt.plot(hist)
plt.xlabel("Pixel Intensity")
plt.ylabel("Number of Pixels")
plt.title("Histogram of Thresh Image")
plt.show()
```

Figure 1

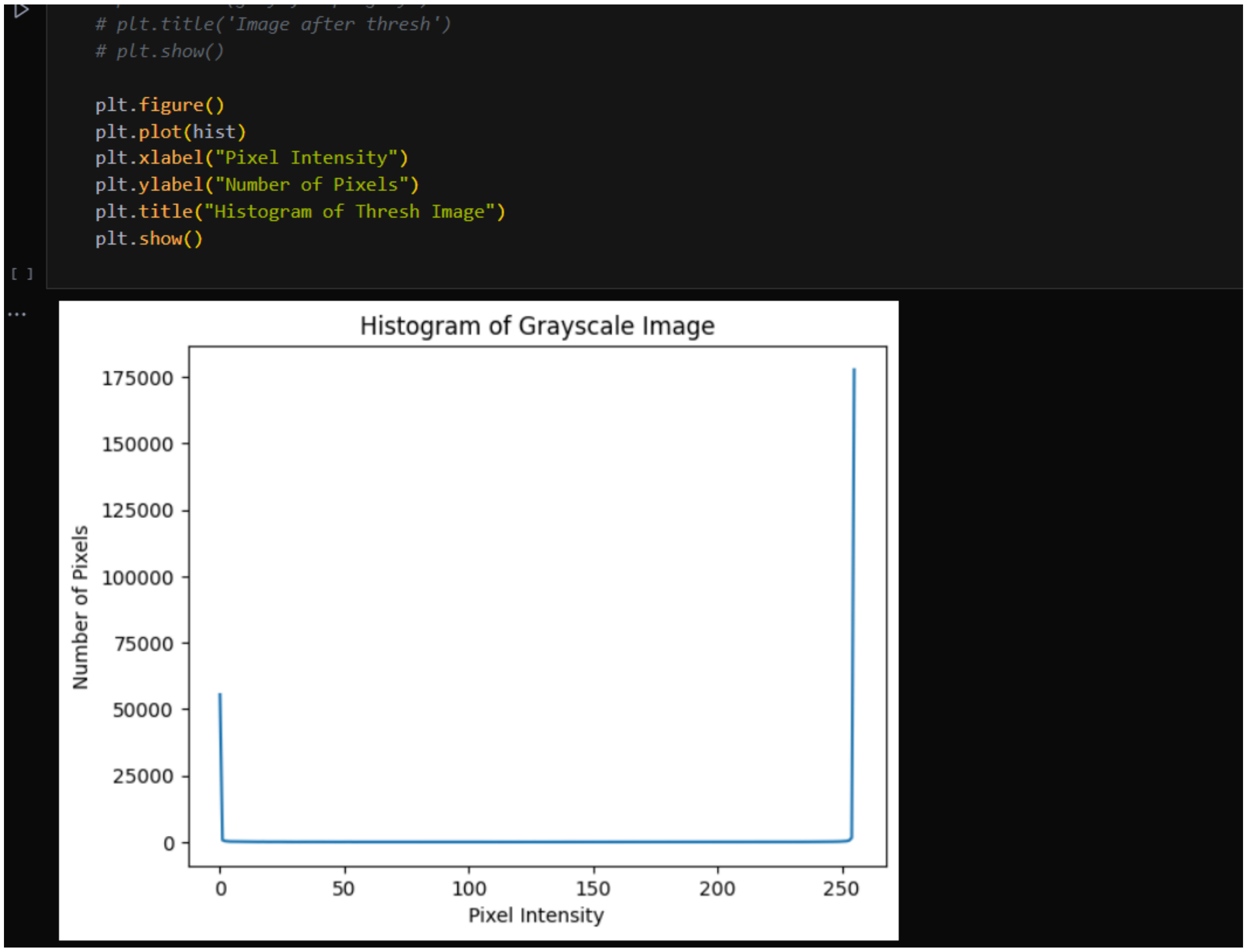
AVERAGE_INTENSITY

This function, average_intensity, calculates the average intensity of a grayscale image represented by its histogram. It operates by iterating through each intensity level in the histogram, weighting each intensity by its frequency (number of pixels with that intensity), and then summing up these weighted intensities. Finally, it divides this sum by the total number of pixels in the image to obtain the average intensity value. This provides a measure of the overall brightness or intensity level present in the image.



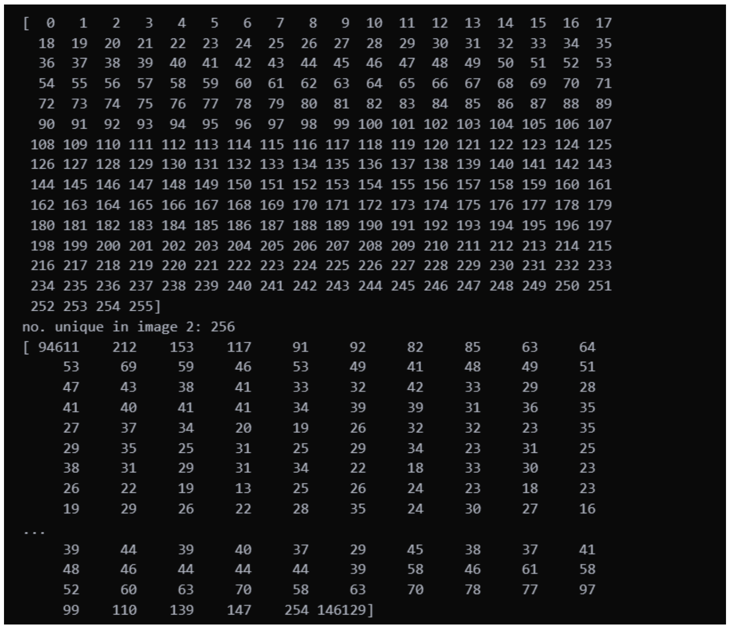
```python
def average_intensity(hist):
    # Get the number of bins (should be 256 in this case)
    num_bins = len(hist)

    # Calculate the sum of weighted intensities
    total_intensity = 0
    for bin_idx in range(num_bins):
        # Intensity value for this bin (assuming a linear scale from 0 to 255)
        intensity = bin_idx

        # Weight by the frequency in the bin
        weight = hist[bin_idx]

        # Add weighted intensity to the total
        total_intensity += intensity * weight

    # Calculate the total number of pixels (assuming the image is not empty)
    total_pixels = gray1.size  # Assuming 'gray1' is a 2D grayscale image

    # Estimate the average intensity
    average_intensity = total_intensity / total_pixels

    print("Estimated average intensity:", average_intensity)
    return int(average_intensity)
```

Figure 2



```
# plt.title('Image after thresh')
# plt.show()

plt.figure()
plt.plot(hist)
plt.xlabel("Pixel Intensity")
plt.ylabel("Number of Pixels")
plt.title("Histogram of Thresh Image")
plt.show()
```

Figure 3



```
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251
 252 253 254 255]
no. unique in image 2: 256
[ 94611    212    153    117     91     92     82     85     63     64
     53     69     59     46     53     49     41     48     49     51
     47     43     38     41     33     32     42     33     29     28
     41     40     41     41     34     39     39     31     36     35
     27     37     34     20     19     26     32     32     23     35
     29     35     25     31     25     29     34     23     31     25
     38     31     29     31     34     22     18     33     30     23
     26     22     19     13     25     26     24     23     18     23
     19     29     26     22     28     35     24     30     27     16
...
     39     44     39     40     37     29     45     38     37     41
     48     46     44     44     44     39     58     46     61     58
     52     60     63     70     58     63     70     78     77     97
     99    110    139    147    254 146129]
```

Figure 4

INTERESTING HISTOGRAM AND UNIQUE VALUES FOUND

As a team, we've come across some interesting histograms in our image analysis Project. These histograms showcase unique aspects of the images we've examined. Here are a few examples:

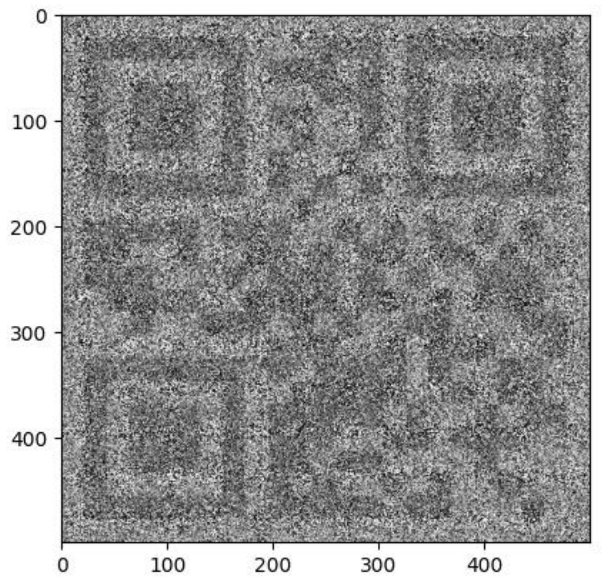


Figure 6

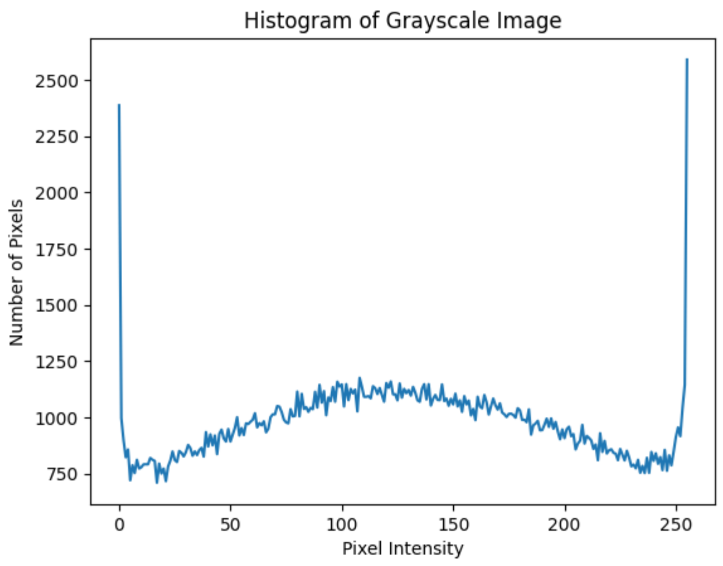


Figure 5

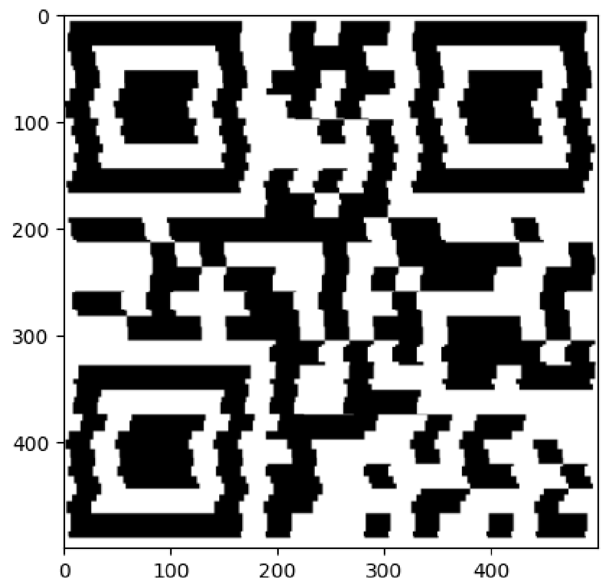


Figure 7

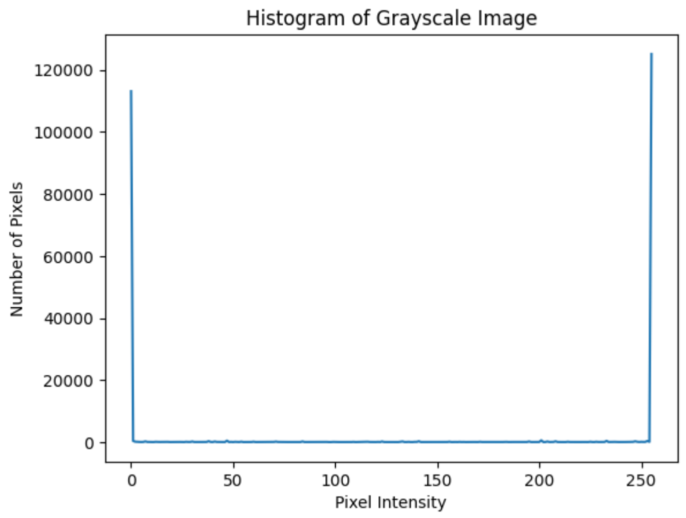


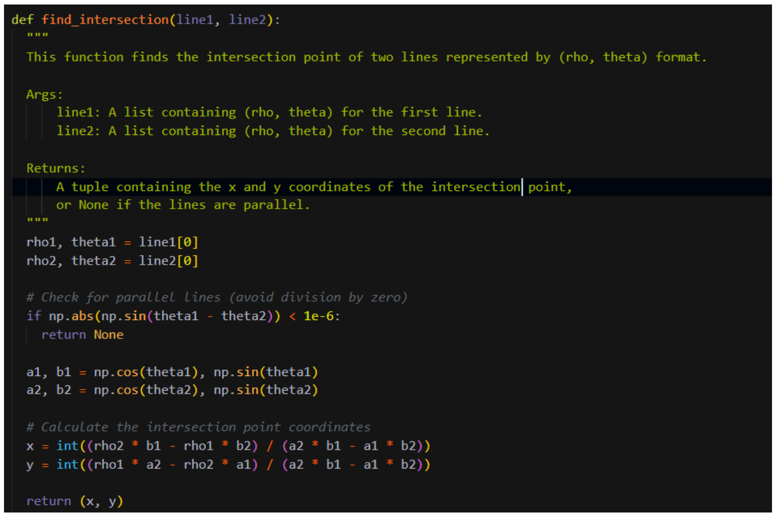
Figure 8

FIND_INTERSECTION

This function, calculates the intersection point of two lines represented in polar coordinate form, specifically in the (ρ, θ) format. It first extracts the ρ and θ values from each input line. Then, it checks if the lines are parallel by comparing the sine of the difference between their θ values. If the lines are nearly parallel (within a small threshold), it returns None to indicate no intersection exists.

If the lines are not parallel, it calculates the coefficients (a1, b1, a2, b2) corresponding to the cosine and sine of the angles θ1 and θ2 for each line. These coefficients are then used to compute the x and y coordinates of the intersection point using linear equation principles derived from the lines' equations in polar form.

The resulting intersection point, represented as a tuple of (x, y) coordinates, is returned if the lines intersect. If not, indicating parallel lines, None is returned.



```python
def find_intersection(line1, line2):
  """
  This function finds the intersection point of two lines represented by (rho, theta) format.

  Args:
      line1: A list containing (rho, theta) for the first line.
      line2: A list containing (rho, theta) for the second line.

  Returns:
      A tuple containing the x and y coordinates of the intersection point,
      or None if the lines are parallel.
  """
  rho1, theta1 = line1[0]
  rho2, theta2 = line2[0]

  # Check for parallel lines (avoid division by zero)
  if np.abs(np.sin(theta1 - theta2)) < 1e-6:
    return None

  a1, b1 = np.cos(theta1), np.sin(theta1)
  a2, b2 = np.cos(theta2), np.sin(theta2)

  # Calculate the intersection point coordinates
  x = int((rho2 * b1 - rho1 * b2) / (a2 * b1 - a1 * b2))
  y = int((rho1 * a2 - rho2 * a1) / (a2 * b1 - a1 * b2))

  return (x, y)
```

Figure 9

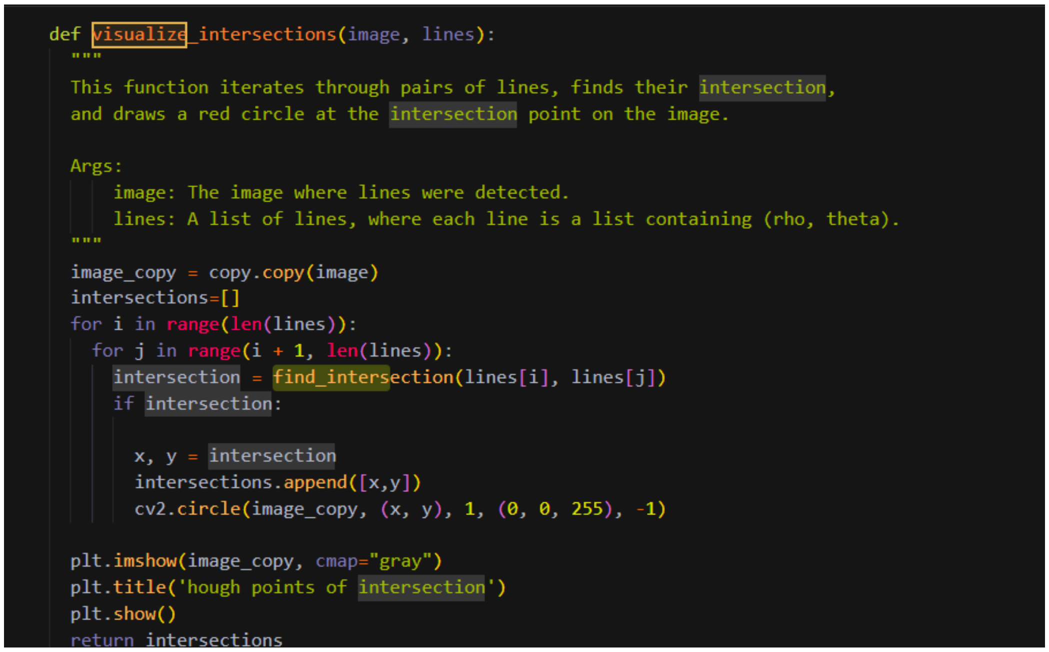
VISUALIZE_INTERSECTIONS

This function, takes an image and a list of lines detected within that image as input. It iterates through pairs of lines, finding their intersection points using the previously defined find_intersection function. If an intersection point is found, it draws a red circle at that point on a copy of the input image.

The function first creates a copy of the input image to avoid modifying the original. Then, it iterates through pairs of lines using nested loops. For each pair of lines, it calls the find_intersection function to determine if they intersect. If an intersection point is returned (indicating non-parallel lines), it extracts the x and y coordinates and draws a red circle at that point on the copied image using OpenCV's cv2.circle function.

After processing all pairs of lines, the function displays the modified image with the intersection points highlighted as red circles using Matplotlib's plt.imshow function.

Finally, it returns a list of intersection points for further analysis or use outside the function.



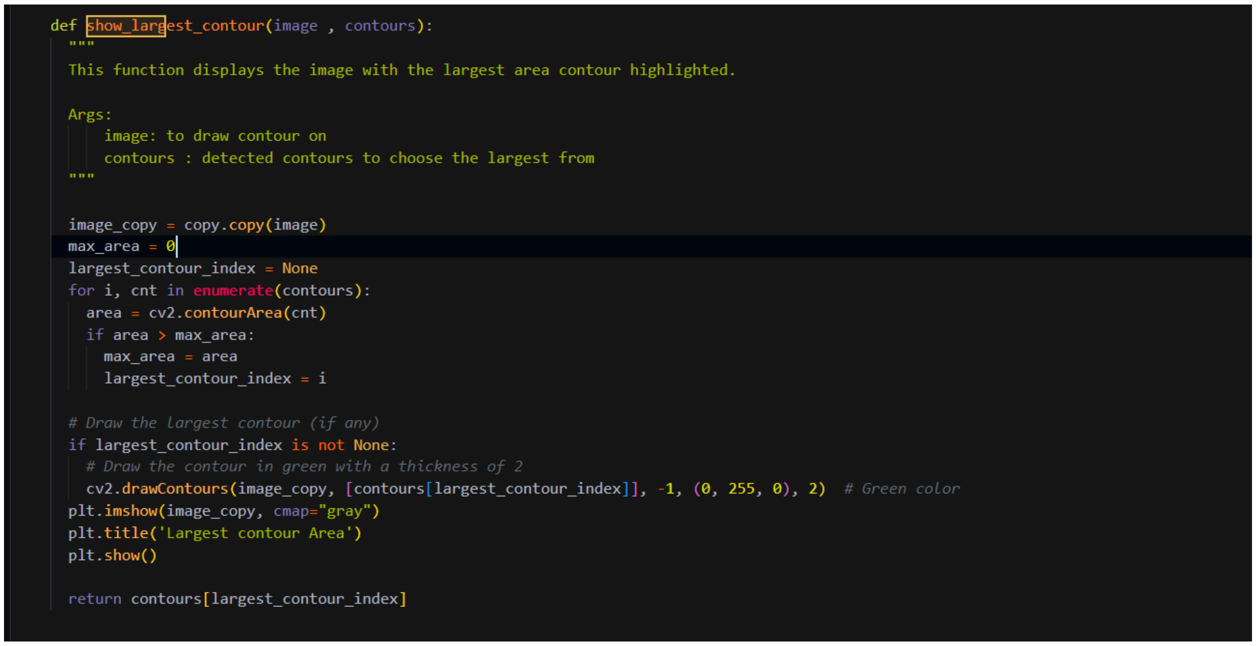
```python
def visualize_intersections(image, lines):
    """
    This function iterates through pairs of lines, finds their intersection,
    and draws a red circle at the intersection point on the image.

    Args:
        image: The image where lines were detected.
        lines: A list of lines, where each line is a list containing (rho, theta).
    """
    image_copy = copy.copy(image)
    intersections=[]
    for i in range(len(lines)):
      for j in range(i + 1, len(lines)):
        intersection = find_intersection(lines[i], lines[j])
        if intersection:

          x, y = intersection
          intersections.append([x,y])
          cv2.circle(image_copy, (x, y), 1, (0, 0, 255), -1)

    plt.imshow(image_copy, cmap="gray")
    plt.title('hough points of intersection')
    plt.show()
    return intersections
```

Figure 10

SHOW_LARGEST_CONTOUR

This function, show_largest_contour, is designed to display an image with the largest area contour highlighted. It takes two arguments: the original image on which contours are to be drawn and a list of contours detected in the image.

The function first creates a copy of the input image to avoid modifying the original image. Then, it initializes variables to keep track of the maximum contour area found (max_area) and the index of the largest contour (largest_contour_index). These variables are initially set to zero and None, respectively.Next, the function iterates through each contour in the provided list (contours). For each contour, it calculates its area using the cv2.contourArea function. If the calculated area is greater than the current maximum area (max_area), it updates max_area with the new value and records the index of the largest contour (largest_contour_index).After iterating through all contours, the function checks if a largest contour was found (i.e., if largest_contour_index is not None). If a largest contour exists, it draws this contour on the copied image using the cv2.drawContours function. The contour is drawn in green with a thickness of 2 pixels.Finally, the modified image, with the largest contour highlighted, is displayed using Matplotlib's plt.imshow function. The function also returns the largest contour found, which can be useful for further analysis or processing outside the function.



```python
def show_largest_contour(image , contours):
  """
  This function displays the image with the largest area contour highlighted.

  Args:
      image: to draw contour on
      contours : detected contours to choose the largest from
  """

  image_copy = copy.copy(image)
  max_area = 0
  largest_contour_index = None
  for i, cnt in enumerate(contours):
    area = cv2.contourArea(cnt)
    if area > max_area:
      max_area = area
      largest_contour_index = i

  # Draw the largest contour (if any)
  if largest_contour_index is not None:
    # Draw the contour in green with a thickness of 2
    cv2.drawContours(image_copy, [contours[largest_contour_index]], -1, (0, 255, 0), 2)  # Green color
  plt.imshow(image_copy, cmap="gray")
  plt.title('Largest contour Area')
  plt.show()

  return contours[largest_contour_index]
```

Figure 11

IMAGE MANIPULATION FUNCTIONS

Functions like rotate_image and is_flipped could be used for correcting potential skew or rotation in the captured image.

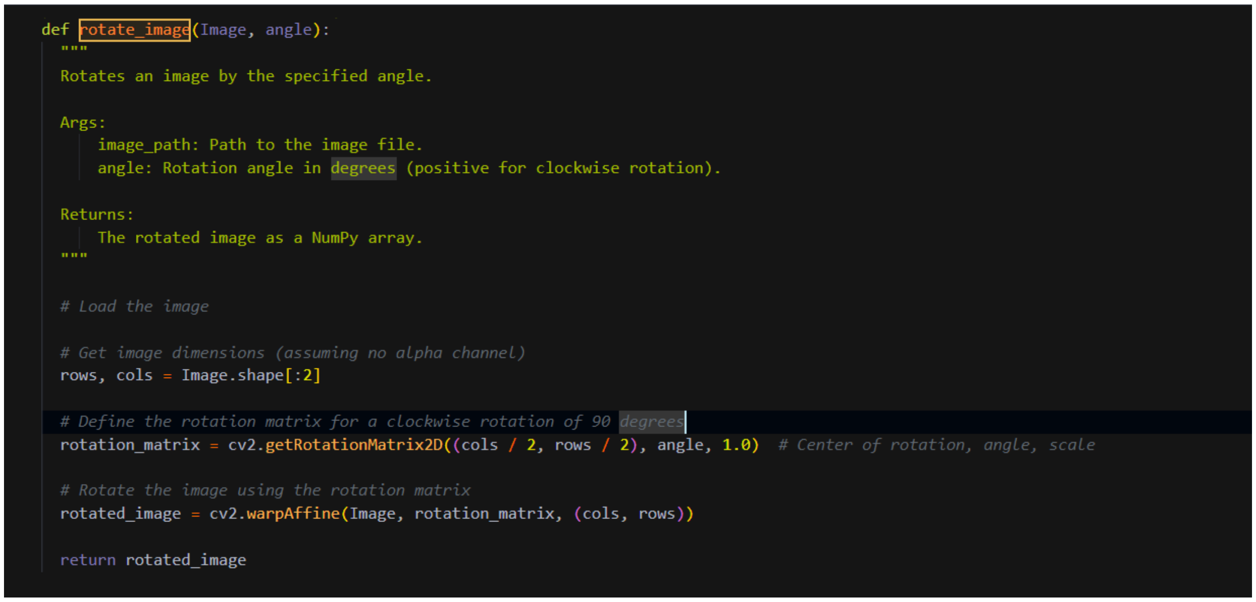
ROTATE_IMAGE

This function, rotate_image, rotates an input image by a specified angle. It takes two arguments: the input image (Image) represented as a NumPy array and the rotation angle (angle) in degrees.

First, it obtains the dimensions of the input image assuming there is no alpha channel, i.e., it extracts the number of rows and columns using the .shape attribute of the NumPy array.

Then, it constructs a rotation matrix using cv2.getRotationMatrix2D. This function requires three arguments: the center of rotation (which is typically the center of the image), the rotation angle (angle), and the scale factor (set to 1.0 here, indicating no scaling). The rotation matrix obtained will describe the transformation required to rotate the image.

After obtaining the rotation matrix, the function applies the rotation to the input image using cv2.warpAffine. This function takes three main arguments: the input image, the rotation matrix, and the output dimensions (which are the same as the input dimensions in this case). It returns the rotated image as a NumPy array.



```python
def rotate_image(Image, angle):
    """
    Rotates an image by the specified angle.

    Args:
        image_path: Path to the image file.
        angle: Rotation angle in degrees (positive for clockwise rotation).

    Returns:
        The rotated image as a NumPy array.
    """

    # Load the image

    # Get image dimensions (assuming no alpha channel)
    rows, cols = Image.shape[:2]

    # Define the rotation matrix for a clockwise rotation of 90 degrees
    rotation_matrix = cv2.getRotationMatrix2D((cols / 2, rows / 2), angle, 1.0)  # Center of rotation, angle, scale

    # Rotate the image using the rotation matrix
    rotated_image = cv2.warpAffine(Image, rotation_matrix, (cols, rows))

    return rotated_image
```

Figure 12

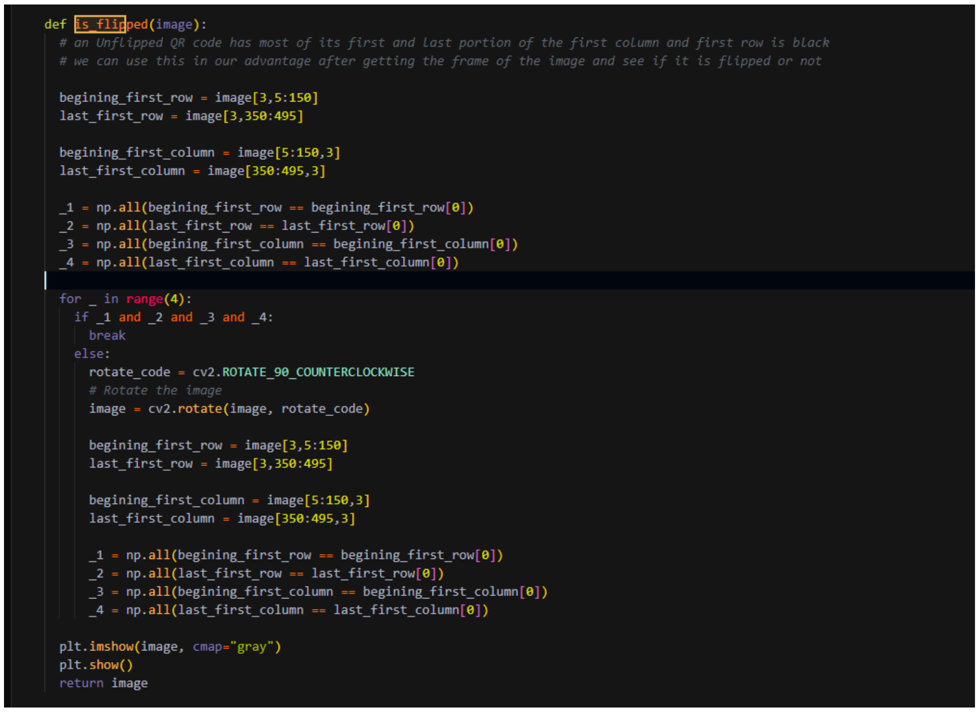
IS_FLIPPED

This function, is_flipped, aims to determine if a QR code image is flipped or not. It employs a heuristic approach based on the observation that an unflipped QR code typically has most of its first and last portions of the first column and first row as black.

Here's how the function works:

- It extracts the beginning and ending portions of the first row and first column of the image.
- It checks if these portions are homogeneous (i.e., contain the same pixel value throughout) using NumPy's np.all function.
- If any of these checks fail, indicating that the image might be flipped, it iteratively rotates the image by 90 degrees counter-clockwise using OpenCV's cv2.rotate function.
- After each rotation, it repeats the checks to see if the image has been correctly oriented.
- It continues this process until either all checks pass or the image has been rotated four times (a full 360-degree rotation).
- Finally, it displays the rotated image using Matplotlib's plt.imshow function and returns the rotated image.

This function provides a practical approach to detect if a QR code image is flipped by examining specific regions of the image for consistency in pixel values. If the image is indeed flipped, it rotates it back to the correct orientation for further processing.



```python
def is_flipped(image):
    # an Unflipped QR code has most of its first and last portion of the first column and first row is black
    # we can use this in our advantage after getting the frame of the image and see if it is flipped or not

    begining_first_row = image[3,5:150]
    last_first_row = image[3,350:495]

    begining_first_column = image[5:150,3]
    last_first_column = image[350:495,3]

    _1 = np.all(begining_first_row == begining_first_row[0])
    _2 = np.all(last_first_row == last_first_row[0])
    _3 = np.all(begining_first_column == begining_first_column[0])
    _4 = np.all(last_first_column == last_first_column[0])

    for _ in range(4):
        if _1 and _2 and _3 and _4:
            break
        else:
            rotate_code = cv2.ROTATE_90_COUNTERCLOCKWISE
            # Rotate the image
            image = cv2.rotate(image, rotate_code)

            begining_first_row = image[3,5:150]
            last_first_row = image[3,350:495]

            begining_first_column = image[5:150,3]
            last_first_column = image[350:495,3]

            _1 = np.all(begining_first_row == begining_first_row[0])
            _2 = np.all(last_first_row == last_first_row[0])
            _3 = np.all(begining_first_column == begining_first_column[0])
            _4 = np.all(last_first_column == last_first_column[0])

    plt.imshow(image, cmap="gray")
    plt.show()
    return image
```

Figure 13

QR CODE PROCESSING FUNCTIONS

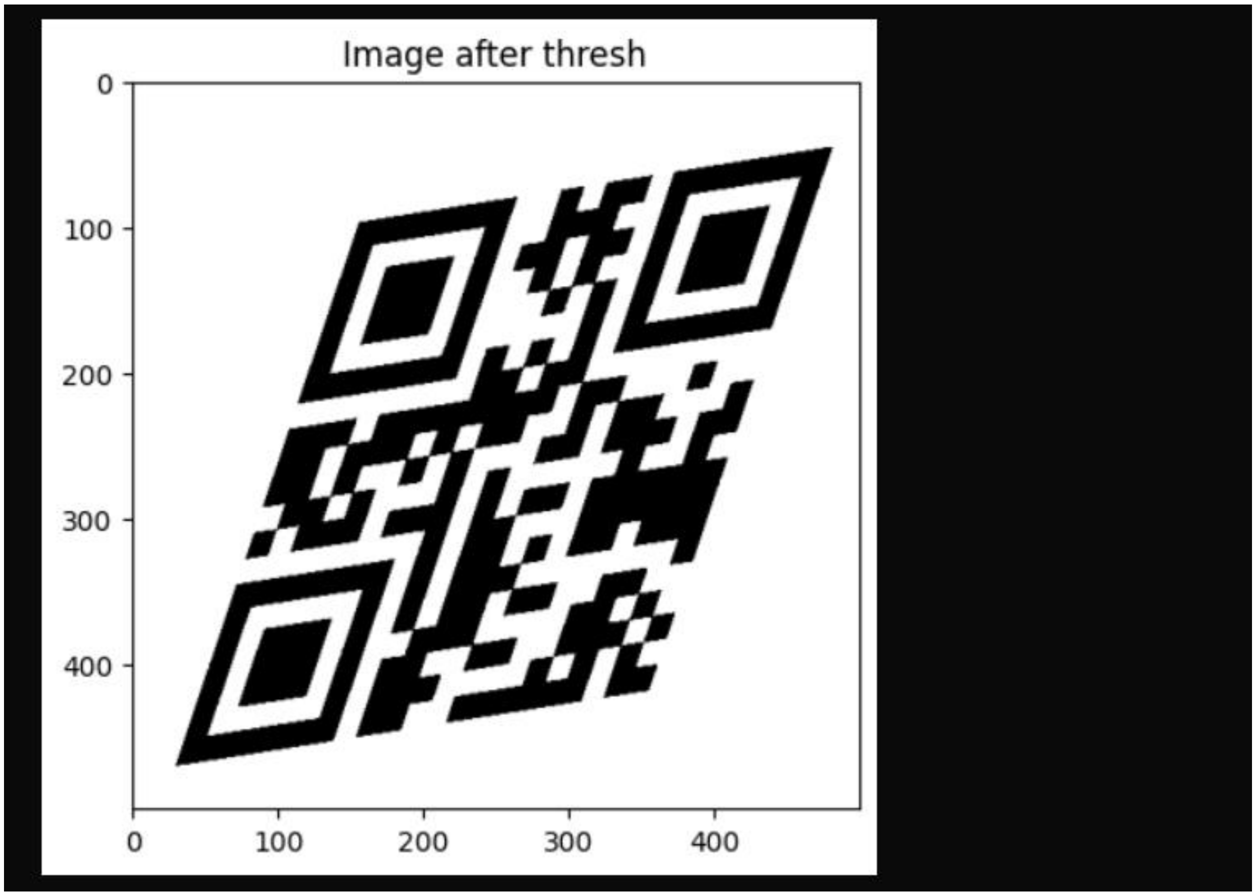
Functions with names like find_unique_pixel_values and is_Dark might be used to analyze the pixel values within the QR code region.

FIND_UNIQUE_PIXEL_VALUES

- FINDING UNIQUE PIXEL VALUES: The code calls a function find_unique_pixel_values(gray1) to obtain the unique pixel values from the grayscale image gray1. It seems to be preparing for further analysis based on the unique pixel values.

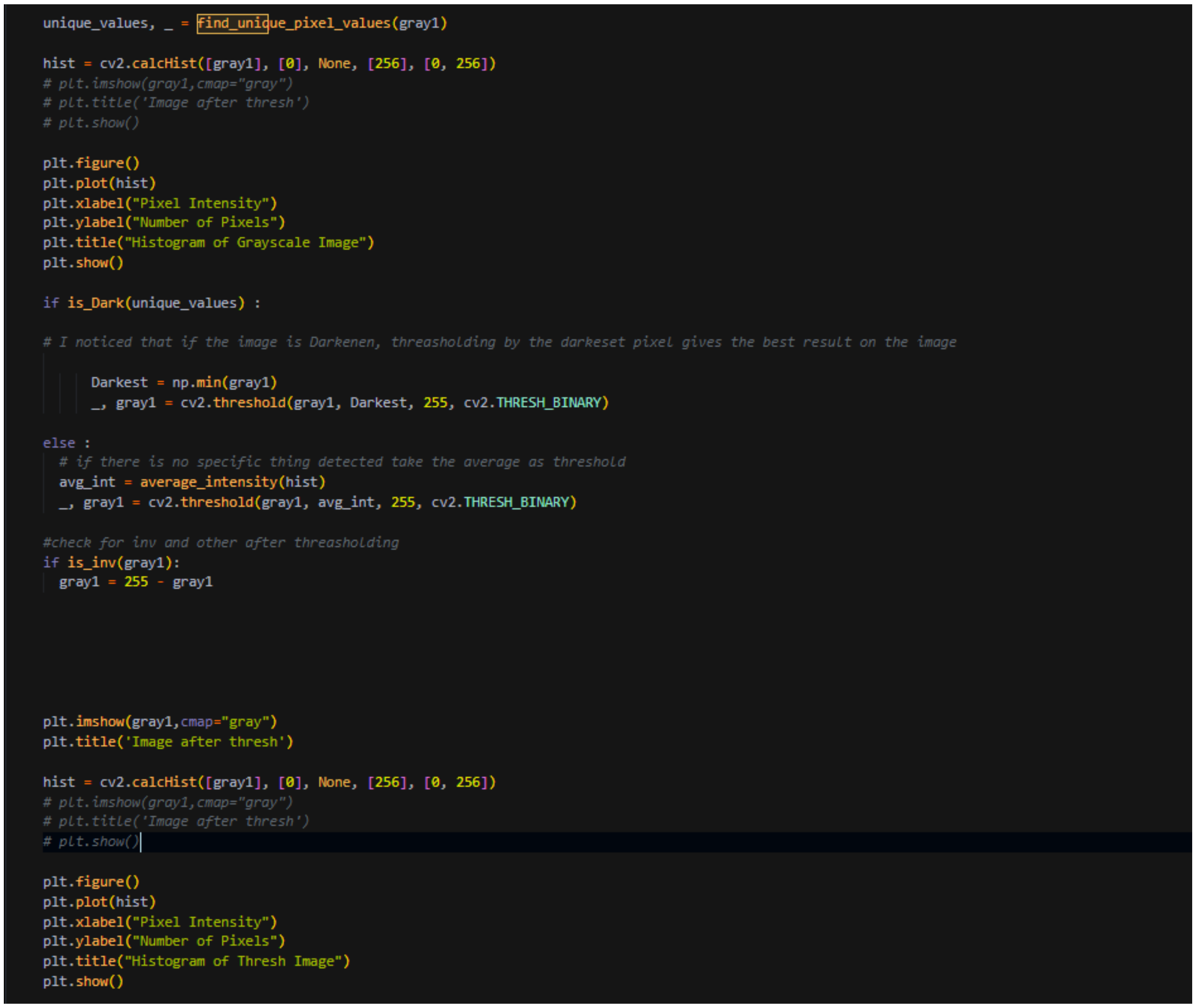
- HISTOGRAM CALCULATION: The code calculates the histogram of the grayscale image gray1 using OpenCV's cv2.calcHist function. The histogram provides a visual representation of the distribution of pixel intensities in the image.

- VISUALIZING HISTOGRAM: It plots the histogram using Matplotlib to visualize the distribution of pixel intensities in the grayscale image.
- THRESHOLDING: Depending on whether the image is considered dark or not (is_Dark(unique_values)), different thresholding methods are applied:



- If the image is deemed dark, thresholding is performed using the darkest pixel value (Darkest = np.min(gray1)).

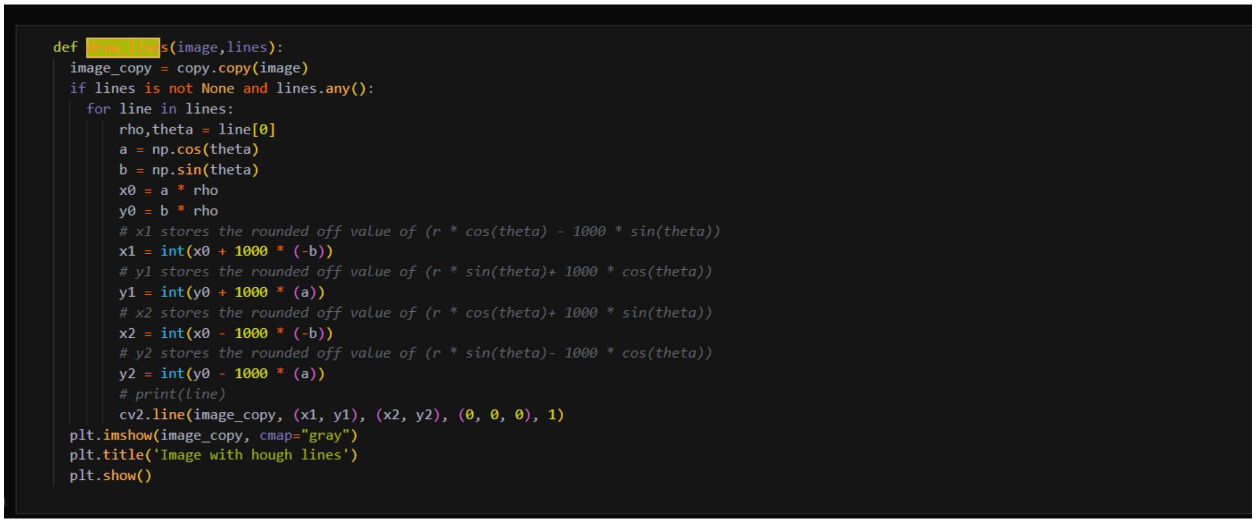
- Otherwise, if the image is not dark, the average intensity value obtained from the histogram is used as the threshold value (avg_int = average_intensity(hist)).
- After thresholding, the image is binarized using OpenCV's cv2.threshold function.
- INVERSE BINARIZATION: After thresholding, if the image is determined to be inverted (is_inv(gray1)), it is inverted back to its original orientation by subtracting the binarized image from 255.
- VISUALIZING THE THRESHOLDED IMAGE AND ITS HISTOGRAM: The script then displays the thresholded image along with its histogram, allowing visual inspection of the effects of thresholding.



```
unique_values, _ = find_unique_pixel_values(gray1)

hist = cv2.calcHist([gray1], [0], None, [256], [0, 256])
# plt.imshow(gray1,cmap="gray")
# plt.title('Image after thresh')
# plt.show()

plt.figure()
plt.plot(hist)
plt.xlabel("Pixel Intensity")
plt.ylabel("Number of Pixels")
plt.title("Histogram of Grayscale Image")
plt.show()

if is_Dark(unique_values) :

# I noticed that if the image is Darkenen, threasholding by the darkeset pixel gives the best result on the image

      Darkest = np.min(gray1)
      _, gray1 = cv2.threshold(gray1, Darkest, 255, cv2.THRESH_BINARY)

else :
  # if there is no specific thing detected take the average as threshold
  avg_int = average_intensity(hist)
  _, gray1 = cv2.threshold(gray1, avg_int, 255, cv2.THRESH_BINARY)

#check for inv and other after threasholding
if is_inv(gray1):
  gray1 = 255 - gray1




plt.imshow(gray1,cmap="gray")
plt.title('Image after thresh')

hist = cv2.calcHist([gray1], [0], None, [256], [0, 256])
# plt.imshow(gray1,cmap="gray")
# plt.title('Image after thresh')
# plt.show()

plt.figure()
plt.plot(hist)
plt.xlabel("Pixel Intensity")
plt.ylabel("Number of Pixels")
plt.title("Histogram of Thresh Image")
plt.show()
```

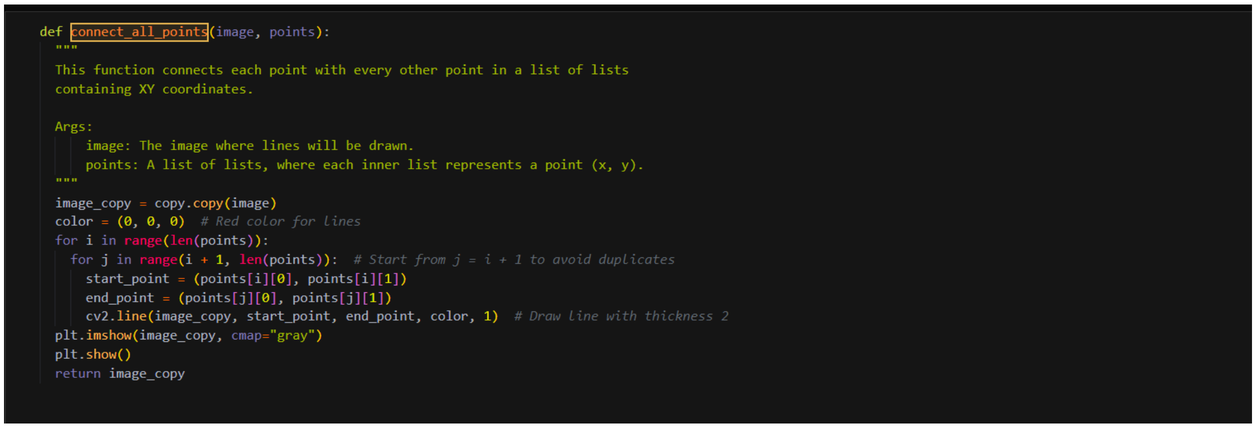
Figure 14

Function	Purpose
DRAW_LINES	Draws lines based on specific representations, potentially used for visualizing detected lines or contours.
CONNECT_ALL_POINTS	Connects all points in a list with lines, possibly for debugging or visualizing identified contours.
SHOW_LARGEST_CONTOUR	Highlights the largest contour, likely representing the outer border of the QR code.
ROTATE_IMAGE	Rotates an image by a specified angle, potentially used for correcting skew or alignment issues.
IS_DARK	Checks if the average intensity is less than a threshold, indicating a potentially dark QR code region.
IS_INV	Checks if the first and last rows/columns are mostly black, suggesting an inverted QR code needing correction.



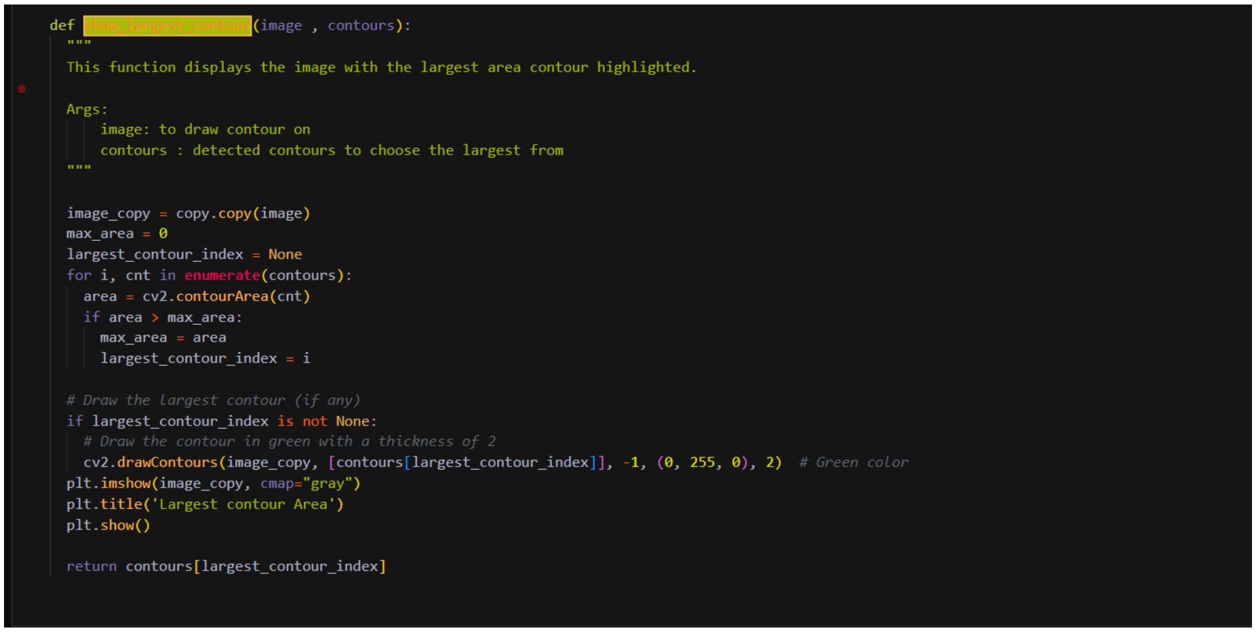
```
def draw_lines(image,lines):
  image_copy = copy.copy(image)
  if lines is not None and lines.any():
    for line in lines:
        rho,theta = line[0]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        # x1 stores the rounded off value of (r * cos(theta) - 1000 * sin(theta))
        x1 = int(x0 + 1000 * (-b))
        # y1 stores the rounded off value of (r * sin(theta)+ 1000 * cos(theta))
        y1 = int(y0 + 1000 * (a))
        # x2 stores the rounded off value of (r * cos(theta)+ 1000 * sin(theta))
        x2 = int(x0 - 1000 * (-b))
        # y2 stores the rounded off value of (r * sin(theta)- 1000 * cos(theta))
        y2 = int(y0 - 1000 * (a))
        # print(line)
        cv2.line(image_copy, (x1, y1), (x2, y2), (0, 0, 0), 1)
  plt.imshow(image_copy, cmap="gray")
  plt.title('Image with hough lines')
  plt.show()
```

Figure 15



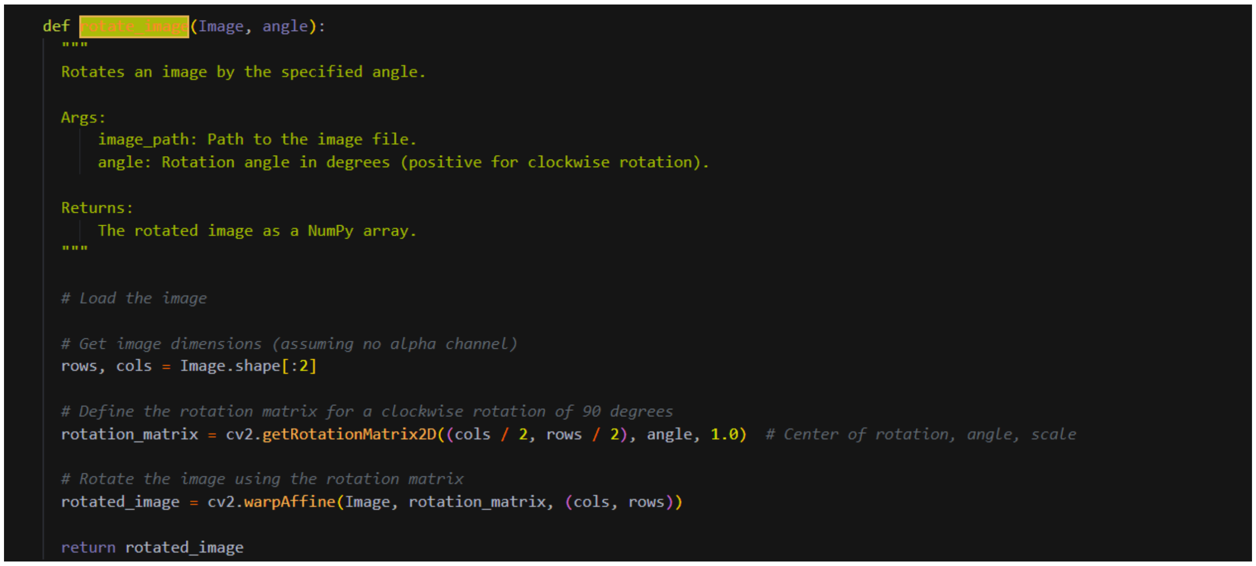
```python
def connect_all_points(image, points):
    """
    This function connects each point with every other point in a list of lists
    containing XY coordinates.

    Args:
        image: The image where lines will be drawn.
        points: A list of lists, where each inner list represents a point (x, y).
    """
    image_copy = copy.copy(image)
    color = (0, 0, 0)  # Red color for lines
    for i in range(len(points)):
      for j in range(i + 1, len(points)):  # Start from j = i + 1 to avoid duplicates
        start_point = (points[i][0], points[i][1])
        end_point = (points[j][0], points[j][1])
        cv2.line(image_copy, start_point, end_point, color, 1)  # Draw line with thickness 2
    plt.imshow(image_copy, cmap="gray")
    plt.show()
    return image_copy
```

Figure 16



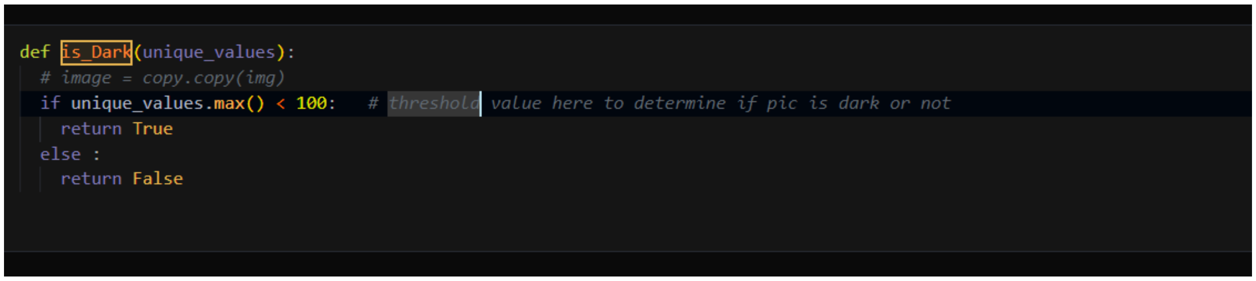
```python
def show_largest_contour(image , contours):
    """
    This function displays the image with the largest area contour highlighted.

    Args:
        image: to draw contour on
        contours : detected contours to choose the largest from
    """

    image_copy = copy.copy(image)
    max_area = 0
    largest_contour_index = None
    for i, cnt in enumerate(contours):
      area = cv2.contourArea(cnt)
      if area > max_area:
        max_area = area
        largest_contour_index = i

    # Draw the largest contour (if any)
    if largest_contour_index is not None:
      # Draw the contour in green with a thickness of 2
      cv2.drawContours(image_copy, [contours[largest_contour_index]], -1, (0, 255, 0), 2)  # Green color
    plt.imshow(image_copy, cmap="gray")
    plt.title('Largest contour Area')
    plt.show()

    return contours[largest_contour_index]
```

Figure 17



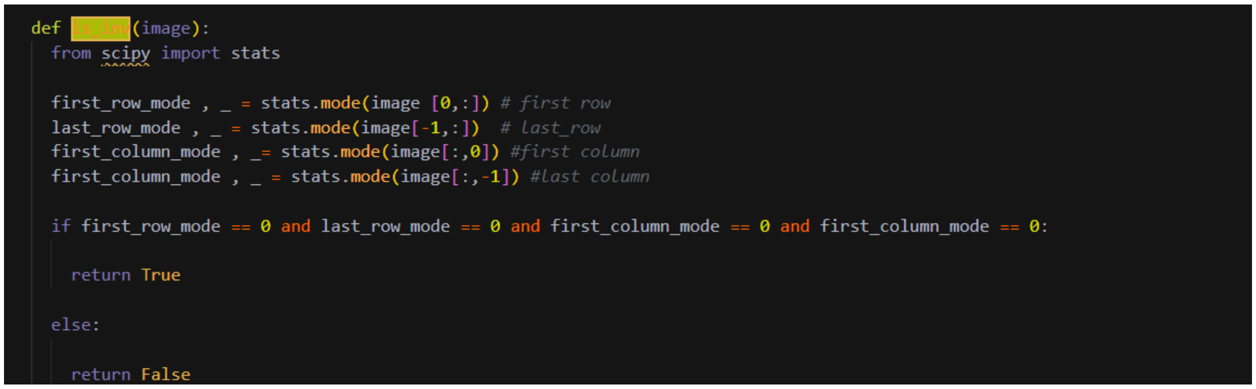
```python
def rotate_image(Image, angle):
    """
    Rotates an image by the specified angle.

    Args:
        image_path: Path to the image file.
        angle: Rotation angle in degrees (positive for clockwise rotation).

    Returns:
        The rotated image as a NumPy array.
    """

    # Load the image

    # Get image dimensions (assuming no alpha channel)
    rows, cols = Image.shape[:2]

    # Define the rotation matrix for a clockwise rotation of 90 degrees
    rotation_matrix = cv2.getRotationMatrix2D((cols / 2, rows / 2), angle, 1.0)  # Center of rotation, angle, scale

    # Rotate the image using the rotation matrix
    rotated_image = cv2.warpAffine(Image, rotation_matrix, (cols, rows))

    return rotated_image
```

Figure 18



```python
def is_Dark(unique_values):
  # image = copy.copy(img)
  if unique_values.max() < 100:   # threshold value here to determine if pic is dark or not
    return True
  else :
    return False
```

Figure 19



```python
def is_inv(image):
  from scipy import stats

  first_row_mode , _ = stats.mode(image [0,:]) # first row
  last_row_mode , _ = stats.mode(image[-1,:])  # last_row
  first_column_mode , _= stats.mode(image[:,0]) #first column
  first_column_mode , _ = stats.mode(image[:,-1]) #last column

  if first_row_mode == 0 and last_row_mode == 0 and first_column_mode == 0 and first_column_mode == 0:

    return True

  else:

    return False
```

Figure 20

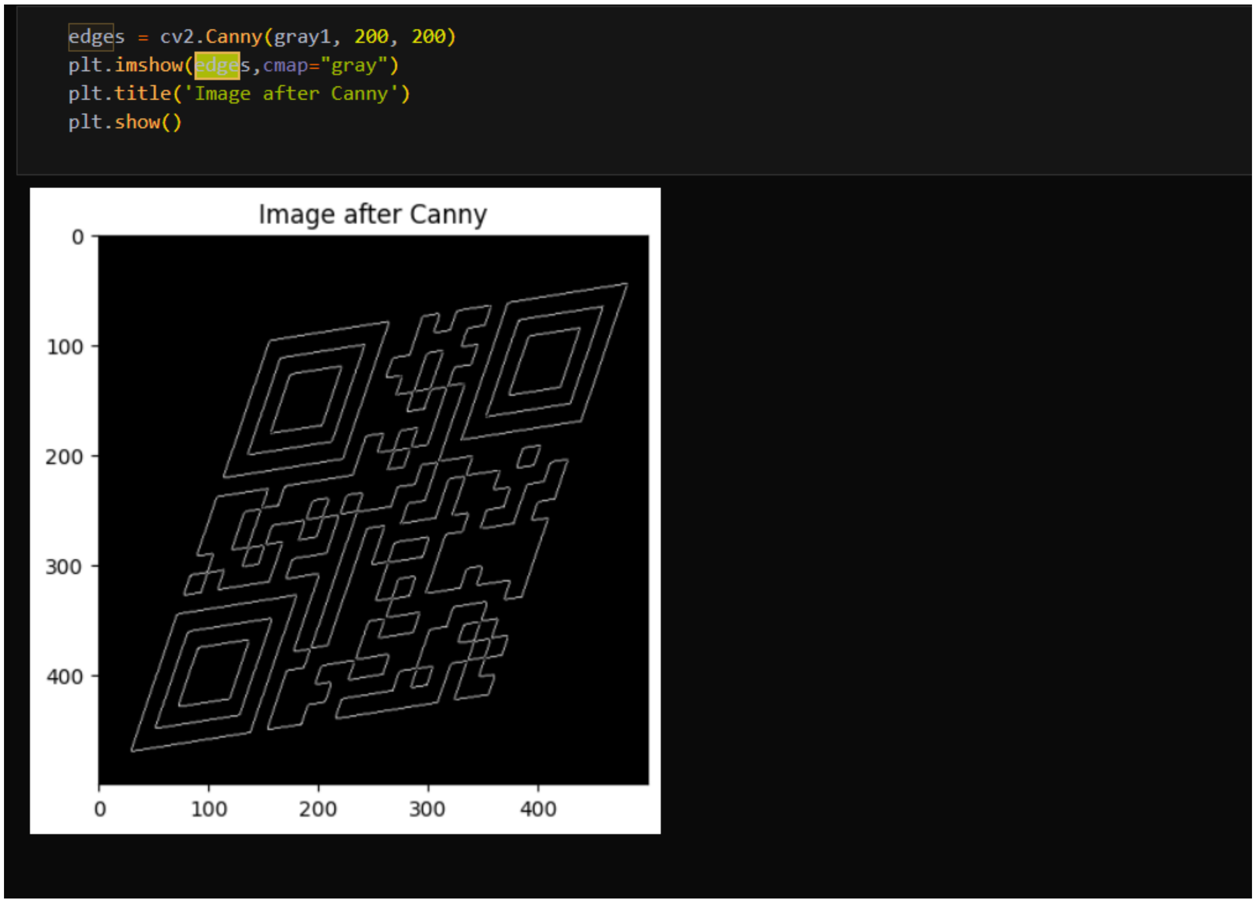
CANNY EDGE DETECTION AND HOUGH LINE TRANSFORM FOR QR CODES

In our QR code processing code, we utilized two powerful image processing techniques to isolate the QR code from its background and identify its key features: Canny edge detection and Hough Line Transform.

CANNY EDGE DETECTION

We first converted the image to grayscale, simplifying the analysis for Canny. Since QR codes typically rely on sharp black and white contrast, grayscale representation was ideal. We might have also incorporated noise reduction techniques before applying Canny to further enhance the accuracy of edge detection.

The Canny edge detector played a critical role in isolating the dark squares and their boundaries within the QR code. It works by identifying pixels with a significant change in intensity. We employed two thresholds: a high threshold for strong, reliable edges, and a low threshold for weaker edges that might connect the strong ones. This two-pronged approach ensured a complete representation of the QR code's borders.



```
edges = cv2.Canny(gray1, 200, 200)
plt.imshow(edges,cmap="gray")
plt.title('Image after Canny')
plt.show()
```

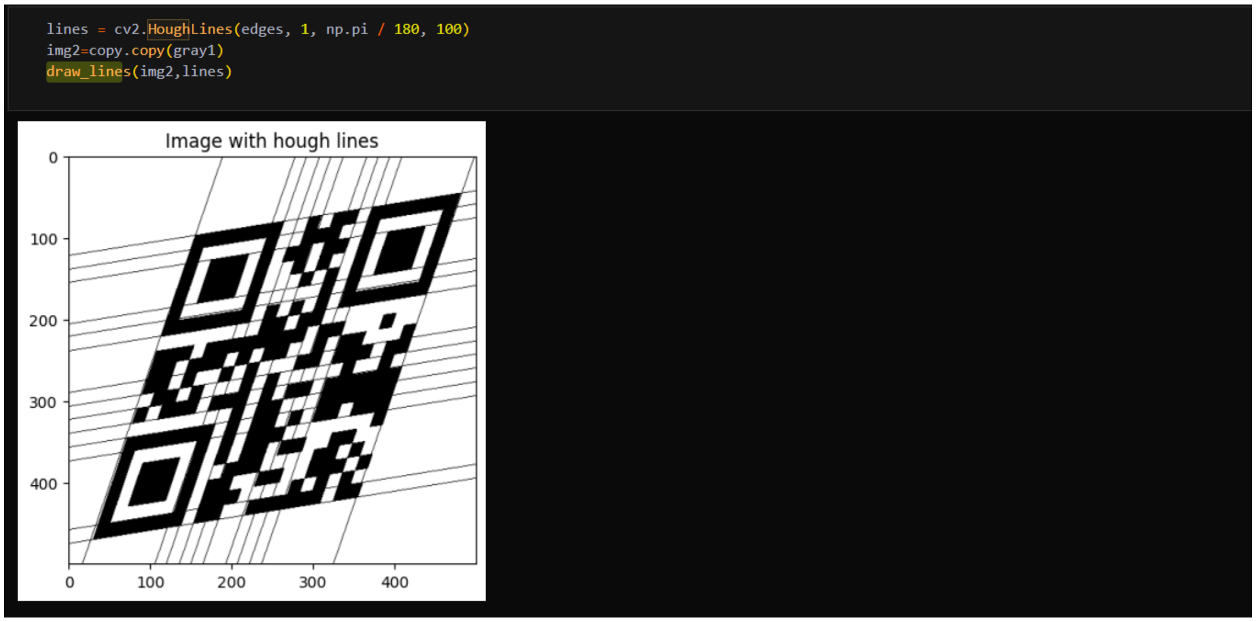
Figure 21

HOUGH LINE TRANSFORM

Once Canny successfully identified the edges, we leveraged the Hough Line Transform to locate straight lines within the image. These lines could potentially correspond to the edges of the QR code's square grid.

We used the edge points detected by Canny as input for the Hough Transform. This technique maps these points onto a parameter space representing lines using distance (ρ) and angle (θ). Essentially, each edge point "votes" for potential lines in the image.

We then analyzed the parameter space. Points corresponding to collinear edges accumulated votes in specific cells, forming peaks. By applying a threshold, we filtered out weak lines and focused on those with a high number of votes. Finally, based on these identified peaks, we extracted the corresponding lines (represented by ρ and θ) back in the original image space.



```
lines = cv2.HoughLines(edges, 1, np.pi / 180, 100)
img2=copy.copy(gray1)
draw_lines(img2,lines)
```

Figure 22

INTEGRATION AND TEAMWORK

The lines extracted by the Hough Transform were instrumental in finding potential corners of the QR code. Intersections between these lines often indicate the code's boundaries. By isolating the largest contour (likely corresponding to the QR code) and filtering out other lines that might not be part of the grid, we were able to refine the line data for further processing.

This combined approach using Canny edge detection and Hough Line Transform showcases the power of teamwork in image processing. By combining these techniques, we achieved a robust method for QR code detection, paving the way for successful decoding in the next steps of our code.

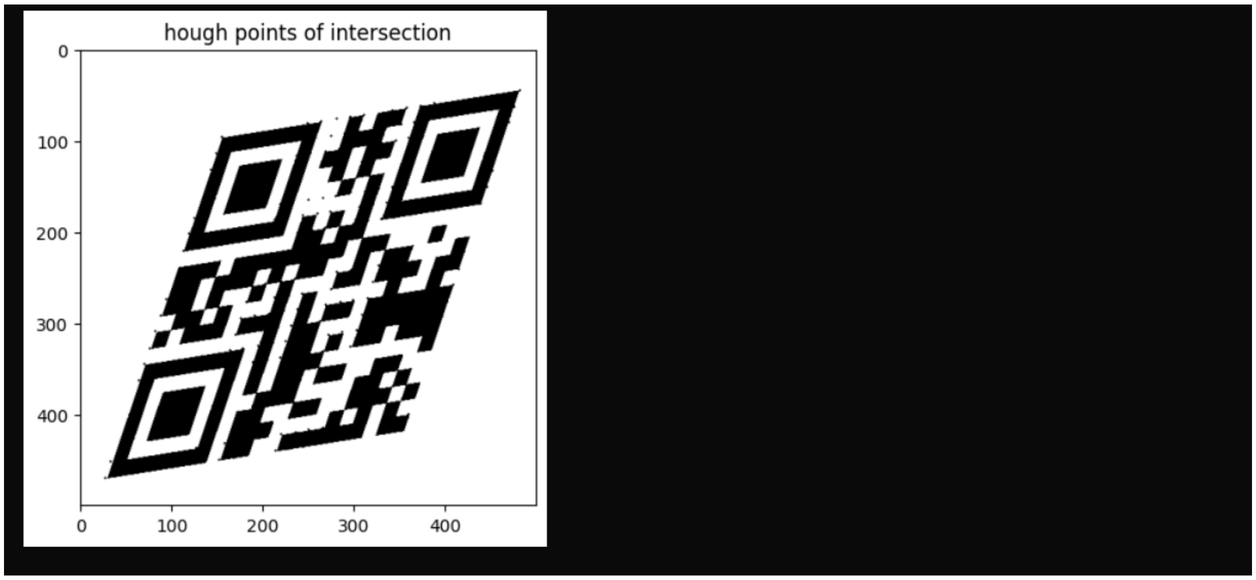


Figure 23

PERSPECTIVE TRANSFORM FOR QR CODE PROCESSING

In our QR code processing code, we leverage perspective transform as a crucial preprocessing step. This technique tackles a common challenge – distortions caused by the camera angle or the orientation of the QR code itself. Perspective transform helps us ensure the code appears undistorted and rectangular, even if the image was captured at an oblique angle.

IMAGINE THIS: You're holding a QR code in front of your phone, but not quite perfectly straight. The captured image might show the code tilted or skewed. This can cause problems when trying to decode the information because QR code readers rely on a specific grid pattern.

PERSPECTIVE TRANSFORM TO THE RESCUE!

Here's how perspective transform works in our code:

1. IDENTIFYING CORRESPONDING POINTS: We typically need four points from the image that would correspond to the corners of a perfectly rectangular QR code if viewed straight-on. These points define the desired, undistorted shape.
2. WARPING THE IMAGE: Mathematically, we calculate a transformation function based on the identified points and the desired rectangular shape. This function essentially maps each pixel in the distorted image to its corresponding location in the undistorted, rectangular version.
3. RESHAPING THE IMAGE: Finally, we apply this transformation function to every pixel in the original image. This effectively warps the image to create a new version where the QR code appears flat and undistorted.

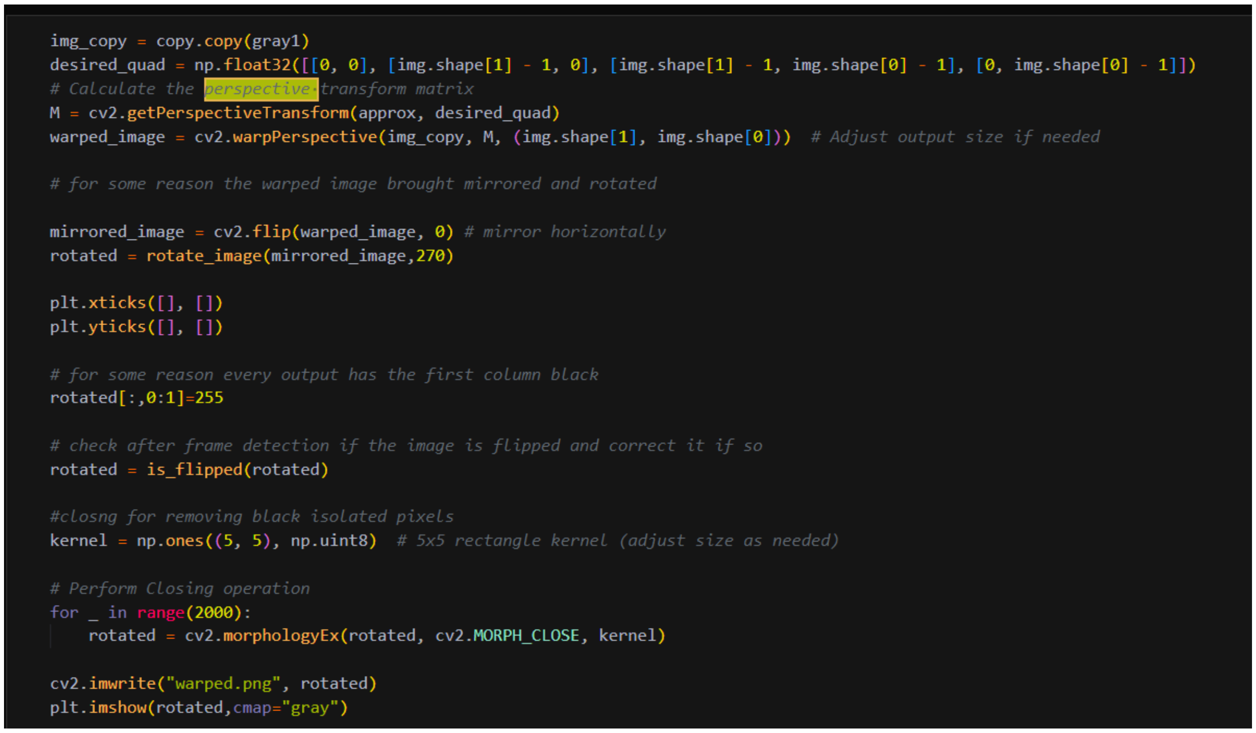
BENEFITS OF PERSPECTIVE TRANSFORM:

By applying perspective transform, we achieve several advantages:

IMPROVED EDGE DETECTION: As mentioned earlier, Canny edge detection performs better on images with straight lines. Perspective transform ensures the QR code's grid lines are straight, leading to more accurate edge detection in the subsequent step.

ROBUST DECODING: A straightened QR code allows for more reliable decoding in later stages of our code. The decoder can accurately identify the data cells and extract the encoded information without errors caused by distortions.

WIDER APPLICABILITY: Perspective transform is a versatile technique applicable to various situations. It can handle QR codes printed on uneven surfaces, captured from unusual angles, or even slightly crumpled.



```python
img_copy = copy.copy(gray1)
desired_quad = np.float32([[0, 0], [img.shape[1] - 1, 0], [img.shape[1] - 1, img.shape[0] - 1], [0, img.shape[0] - 1]])
# Calculate the perspective transform matrix
M = cv2.getPerspectiveTransform(approx, desired_quad)
warped_image = cv2.warpPerspective(img_copy, M, (img.shape[1], img.shape[0]))  # Adjust output size if needed

# for some reason the warped image brought mirrored and rotated

mirrored_image = cv2.flip(warped_image, 0) # mirror horizontally
rotated = rotate_image(mirrored_image,270)

plt.xticks([], [])
plt.yticks([], [])

# for some reason every output has the first column black
rotated[:,0:1]=255

# check after frame detection if the image is flipped and correct it if so
rotated = is_flipped(rotated)

#closng for removing black isolated pixels
kernel = np.ones((5, 5), np.uint8)  # 5x5 rectangle kernel (adjust size as needed)

# Perform Closing operation
for _ in range(2000):
    rotated = cv2.morphologyEx(rotated, cv2.MORPH_CLOSE, kernel)

cv2.imwrite("warped.png", rotated)
plt.imshow(rotated,cmap="gray")
```

Figure 24

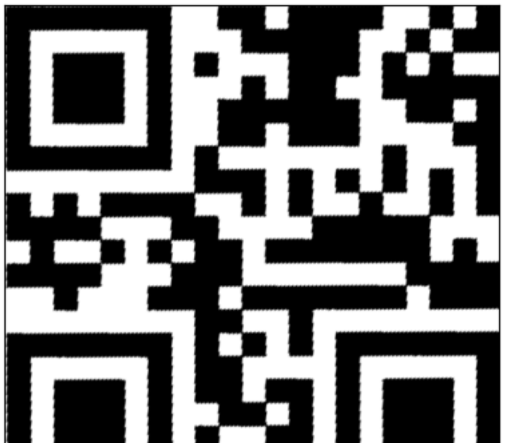


Figure 26

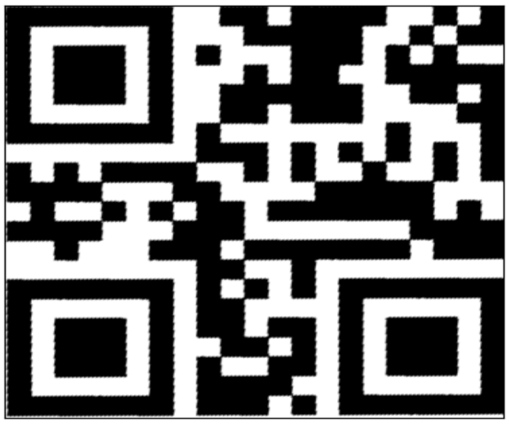


Figure 25

QR DECODING

The initial step of the code involves detecting the QR code's presence and location within the provided image. This is crucial because the subsequent processing relies solely on the data contained within the QR code itself.

Here's a breakdown of the code logic for QR code detection:

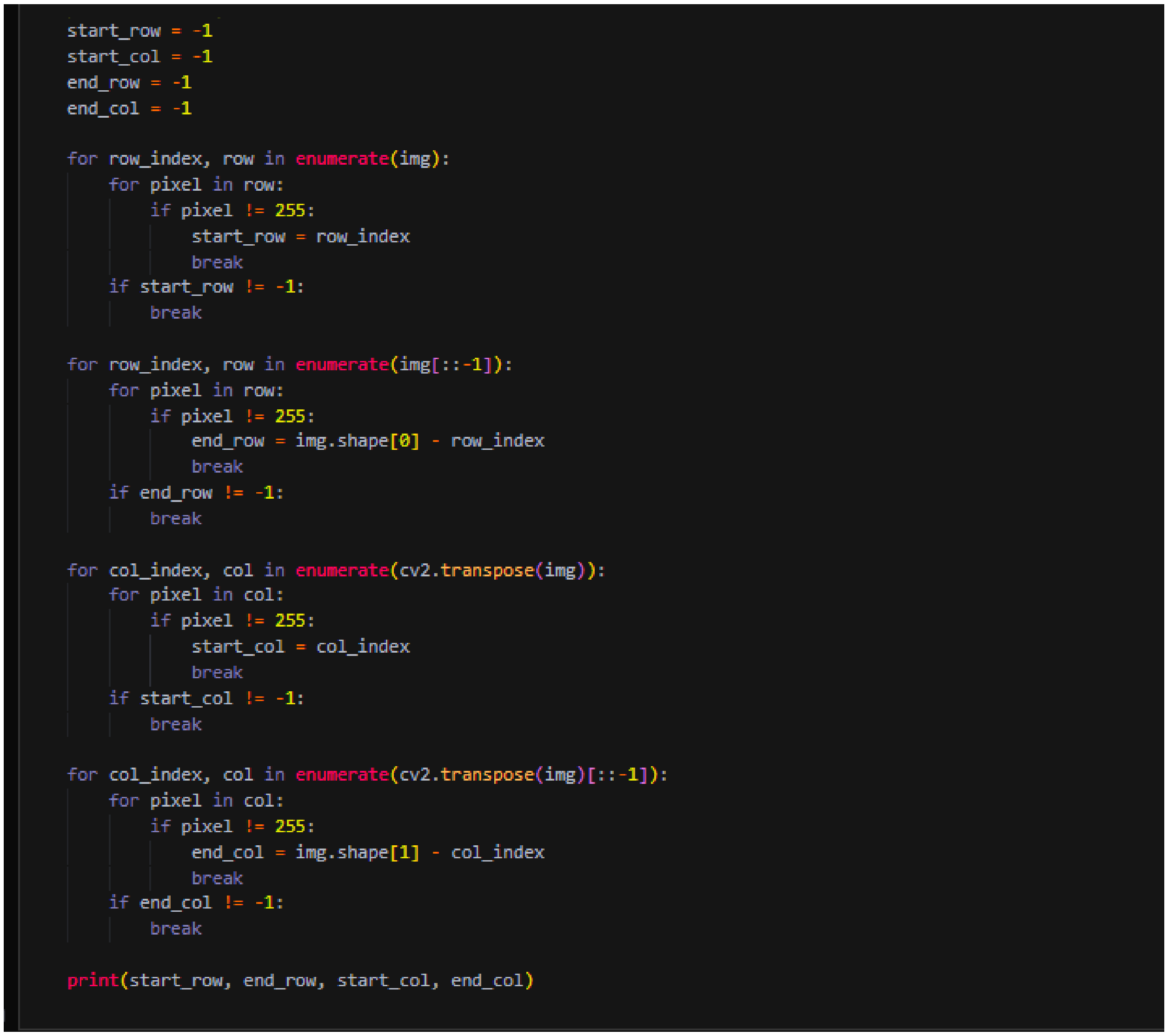
ITERATING THROUGH PIXELS: The code employs nested loops to systematically examine each pixel in the image. The outer loop iterates through each row of the image (`for row_index, row in enumerate(img)`); the inner loop iterates through each pixel within that row (`for pixel in row`).

SEARCHING FOR NON-WHITE PIXELS: Inside the inner loop, the code checks the value of each pixel. In the context of QR codes, a white pixel typically represents a background value (often having a grayscale value of 255). Conversely, a non-white pixel (with a value less than 255) signifies a dark module, which is a fundamental building block of QR codes.

IDENTIFYING STARTING ROW: The code looks for the first non-white pixel encountered in each row. If a non-white pixel is found (`if pixel != 255`), the index of that row (`row_index`) is assigned to the `start_row` variable. This essentially captures the topmost row containing a QR code module. The loop then terminates (`break`) as we've already found the starting position.

REPEATING FOR OTHER BOUNDARIES: The code employs similar logic to identify the remaining boundaries (`start_col`, `end_row`, and `end_col`) of the QR code. It iterates through columns, searching for the first non-white pixels to mark the starting and ending columns. It also repeats the process for all rows to find the bottommost row containing a QR code module.

COMPLETE BOUNDING BOX: By finding the starting and ending positions for rows and columns, the code effectively identifies the bounding box that encloses the entire QR code within the image.



```python
start_row = -1
start_col = -1
end_row = -1
end_col = -1

for row_index, row in enumerate(img):
    for pixel in row:
        if pixel != 255:
            start_row = row_index
            break
    if start_row != -1:
        break

for row_index, row in enumerate(img[::-1]):
    for pixel in row:
        if pixel != 255:
            end_row = img.shape[0] - row_index
            break
    if end_row != -1:
        break

for col_index, col in enumerate(cv2.transpose(img)):
    for pixel in col:
        if pixel != 255:
            start_col = col_index
            break
    if start_col != -1:
        break

for col_index, col in enumerate(cv2.transpose(img)[::-1]):
    for pixel in col:
        if pixel != 255:
            end_col = img.shape[1] - col_index
            break
    if end_col != -1:
        break

print(start_row, end_row, start_col, end_col)
```

Figure 27

QUITE ZONE REMOVAL

A quiet zone is a blank margin or border intentionally incorporated into QR codes. It's essentially a region of white pixels surrounding the actual data-carrying modules of the QR code. The quiet zone serves several important purposes:

ENSURES ACCURATE SCANNING**:** The quiet zone provides a clear and uncluttered area around the QR code. This helps QR code scanners precisely locate the code's boundaries and differentiate it from the background or other image elements. Without a clear quiet zone, nearby objects or patterns could interfere with scanner readings, potentially leading to decoding errors.

IMPROVES ERROR CORRECTION: QR codes employ error correction mechanisms to rectify potential errors introduced during the encoding or reading process. The presence of a quiet zone aids in this error correction functionality. By having a well-defined boundary, the scanner can reliably identify the data region and utilize error correction techniques more effectively.

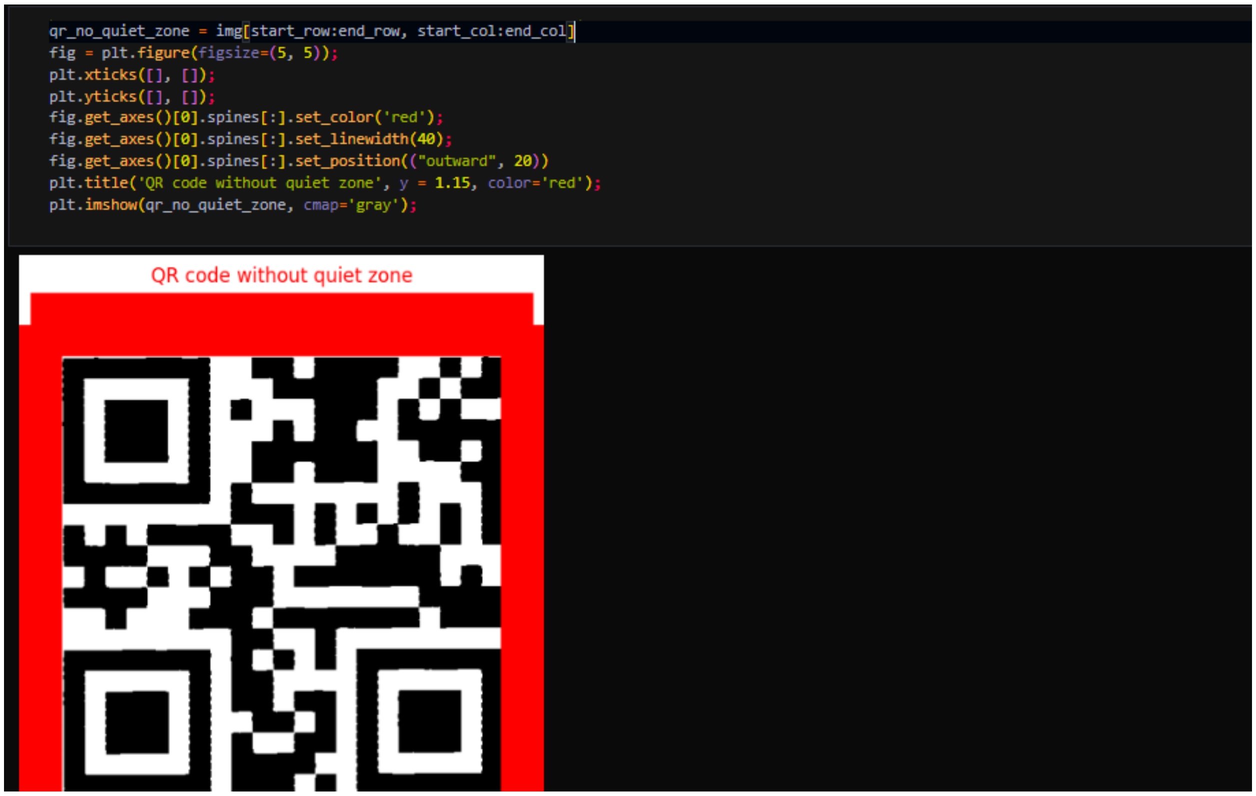
PROVIDES SCANNING FLEXIBILITY: QR codes can be scanned from various orientations and distances. The quiet zone helps scanners adapt to these variations. It offers a buffer zone that allows for slight misalignment or cropping during scanning without compromising the core data area.

WHY REMOVE THE QUIET ZONE?

While the quiet zone plays a vital role during the QR code scanning process, it's not essential for the subsequent decoding stage. In fact, removing the quiet zone simplifies further processing steps within the code:

REDUCES PROCESSING REQUIREMENTS: By eliminating the quiet zone, the code only needs to focus on the data-carrying portion of the QR code. This reduces the amount of image data that needs to be processed, potentially improving performance and efficiency.

STREAMLINES GRID IDENTIFICATION AND DATA EXTRACTION: The removal of the quiet zone ensures the code is working with the core QR code structure containing the data grid and its modules. This simplifies the process of identifying the grid size, extracting individual data cells, and ultimately decoding the information encoded within the QR code.



```python
qr_no_quiet_zone = img[start_row:end_row, start_col:end_col]
fig = plt.figure(figsize=(5, 5));
plt.xticks([], []);
plt.yticks([], []);
fig.get_axes()[0].spines[:].set_color('red');
fig.get_axes()[0].spines[:].set_linewidth(40);
fig.get_axes()[0].spines[:].set_position(("outward", 20))
plt.title('QR code without quiet zone', y = 1.15, color='red');
plt.imshow(qr_no_quiet_zone, cmap='gray');
```

Figure 28

GRID IDENTIFICATION IN QR CODES

The following code snippet incorporates a mechanism to identify the size of the data grid within the decoded QR code. This grid size information is crucial for subsequent steps that involve extracting and decoding the actual data bits encoded within the QR code.

QR CODE STRUCTURE AND GRID

A QR code is comprised of a grid of square modules arranged in a specific pattern. The data to be encoded is embedded within these modules, with their on/off states (dark or light) representing the encoded information. The size of this data grid, or the number of modules along each dimension, determines the data capacity of the QR code. Larger grids can accommodate more information compared to smaller ones.

VERSION INFORMATION AND GRID SIZE

The initial portion of a QR code, specifically the finder patterns and format information areas, encodes crucial information about the QR code's structure, including its version. The version number directly relates to the size of the data grid. Here's the connection:

VERSION NUMBERS**:** QR codes employ various versions, each with a predefined data grid size. These versions range from Version 1 (smallest) to Version 40 (largest).

VERSION AND GRID SIZE RELATIONSHIP: A higher version number signifies a larger data grid. For instance, Version 1 has a 21x21 grid, while Version 40 boasts a 177x177 grid.

CODE'S APPROACH TO GRID IDENTIFICATION

The code you provided utilizes a simplified approach to estimate the grid size:

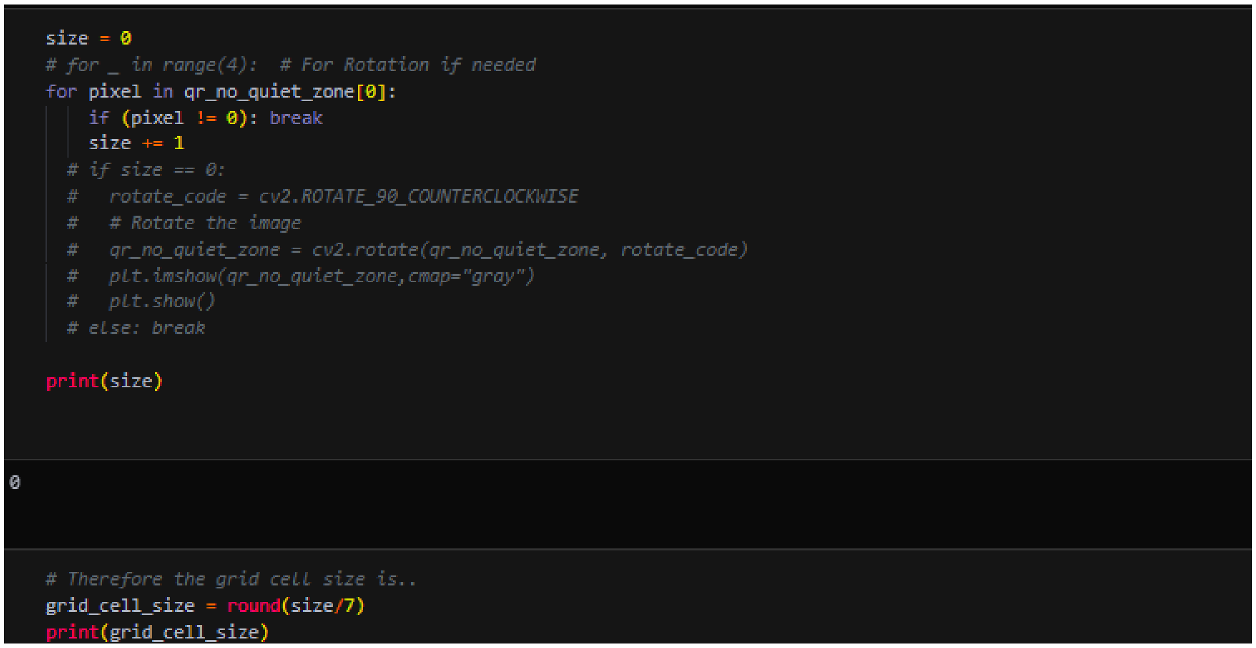
- ANALYZING LEADING WHITE PIXELS**:** It focuses on the top row of the QR code extracted after quiet zone removal (`qr_no_quiet_zone`). It iterates through the pixels in the first row, counting the number of leading white pixels (`size`) before encountering a black pixel.
- ESTIMATION BASED ON A CONSTANT**:** This initial count of leading white pixels (`size`) is then divided by a constant value (typically 7 for QR codes). This constant value is related to the specific structure of function patterns present in QR codes.
- GRID SIZE APPROXIMATION: The resulting value (`size / 7`) provides an approximate estimate of the individual data cell size within the grid.

LIMITATIONS OF THIS APPROACH:

- ACCURACY**:** This method offers a rough estimate and might not be very precise for all QR code versions. The actual relationship between leading white pixels and grid size can vary slightly depending on the specific QR code version and error correction level.
- ALTERNATIVE APPROACHES**:** More sophisticated techniques involve analyzing dedicated grid size information encoded within the format information area of the QR code. This can provide a more accurate determination of the grid size.
-

OVERALL SIGNIFICANCE:

Despite its limitations, the approach used in the code snippet offers a basic mechanism to estimate the data grid size within a QR code. This estimated size is then employed for subsequent steps like resizing the image and segmenting it into individual data cells for further processing and decoding.



```
size = 0
# for _ in range(4):  # For Rotation if needed
for pixel in qr_no_quiet_zone[0]:
    if (pixel != 0): break
    size += 1
  # if size == 0:
  #   rotate_code = cv2.ROTATE_90_COUNTERCLOCKWISE
  #   # Rotate the image
  #   qr_no_quiet_zone = cv2.rotate(qr_no_quiet_zone, rotate_code)
  #   plt.imshow(qr_no_quiet_zone,cmap="gray")
  #   plt.show()
  # else: break

print(size)
```

```
0
```

```
# Therefore the grid cell size is..
grid_cell_size = round(size/7)
print(grid_cell_size)
```

Figure 29

GRID IDENTIFICATION

We've previously discussed how the code identifies the grid size within a QR code. Let's delve deeper into the concept of grid size and explore alternative, more accurate methods for determining it.

QR CODE GRID - THE FOUNDATION OF DATA STORAGE

A QR code's core functionality relies on its data grid. This grid is a two-dimensional arrangement of square modules, where each module can be either dark (on) or light (off). The specific on/off state of these modules encodes the desired information.

GRID SIZE AND DATA CAPACITY

The size of the data grid, or the number of modules along each dimension, directly impacts the amount of data a QR code can store. Larger grids offer more modules to encode information, leading to a higher data capacity. Conversely, smaller grids have limited space and can only accommodate less data.

VERSIONING SYSTEM AND GRID SIZE CORRELATION

QR codes employ a versioning system to categorize different grid sizes. Each version number signifies a specific grid size. Here's a breakdown:

VERSION RANGE**:** Versions range from 1 (smallest) to 40 (largest).

VERSION-GRID SIZE RELATIONSHIP: A higher version number corresponds to a larger grid size. For example, Version 1 has a 21x21 grid, while Version 40 boasts a significantly larger 177x177 grid.

CHALLENGES OF CODE'S APPROACH

The provided code snippet utilizes a simplified method to estimate the grid size:

- ANALYZING LEADING WHITE PIXELS**:** It focuses on the initial white pixels in the first row of the QR code data. It counts the number of leading white pixels before encountering a black pixel.
- ESTIMATION BASED ON CONSTANT**:** This count is then divided by a constant value (usually 7) related to the structure of QR code function patterns.
- LIMITATIONS: While offering a basic idea, this approach has limitations:
    - ACCURACY**:** It might not be very precise, especially for all versions due to variations in the leading white pixel count depending on the specific version and error correction level.
    - ALTERNATIVE APPROACHES: More reliable methods involve decoding dedicated grid size information encoded within the QR code itsel

DATA CELL EXTRACTION IN QR CODES

Following grid size identification, the next step involves extracting individual data cells from the QR code image. These data cells are the fundamental building blocks that hold the encoded information.

UNDERSTANDING DATA CELLS

Recall that a QR code's data is stored within a two-dimensional grid of square modules.

Each data cell encompasses a specific group of these modules. The size of a data cell is directly related to the overall grid size.

The on/off state (dark or light) of the modules within a data cell contributes to the encoded information.

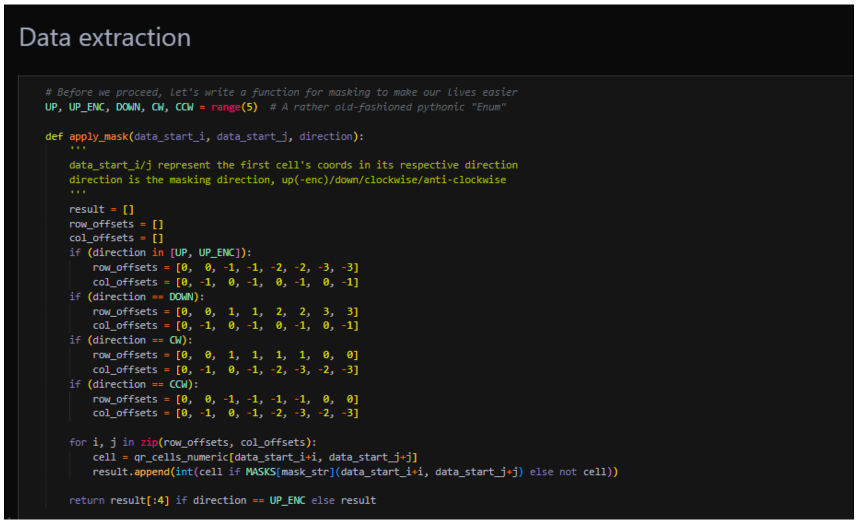
CODE'S APPROACH TO DATA CELL EXTRACTION

The provided code snippet implements a straightforward approach to data cell extraction:

ESTIMATED GRID SIZE: As discussed earlier, the code utilizes the estimated grid size (`grid_cells_num`) obtained from analyzing leading white pixels.

IMAGE RESIZING**:** It resizes the image containing the QR code data (`qr_no_quiet_zone`) to ensure each data cell occupies a square block within the image. This simplifies subsequent processing.

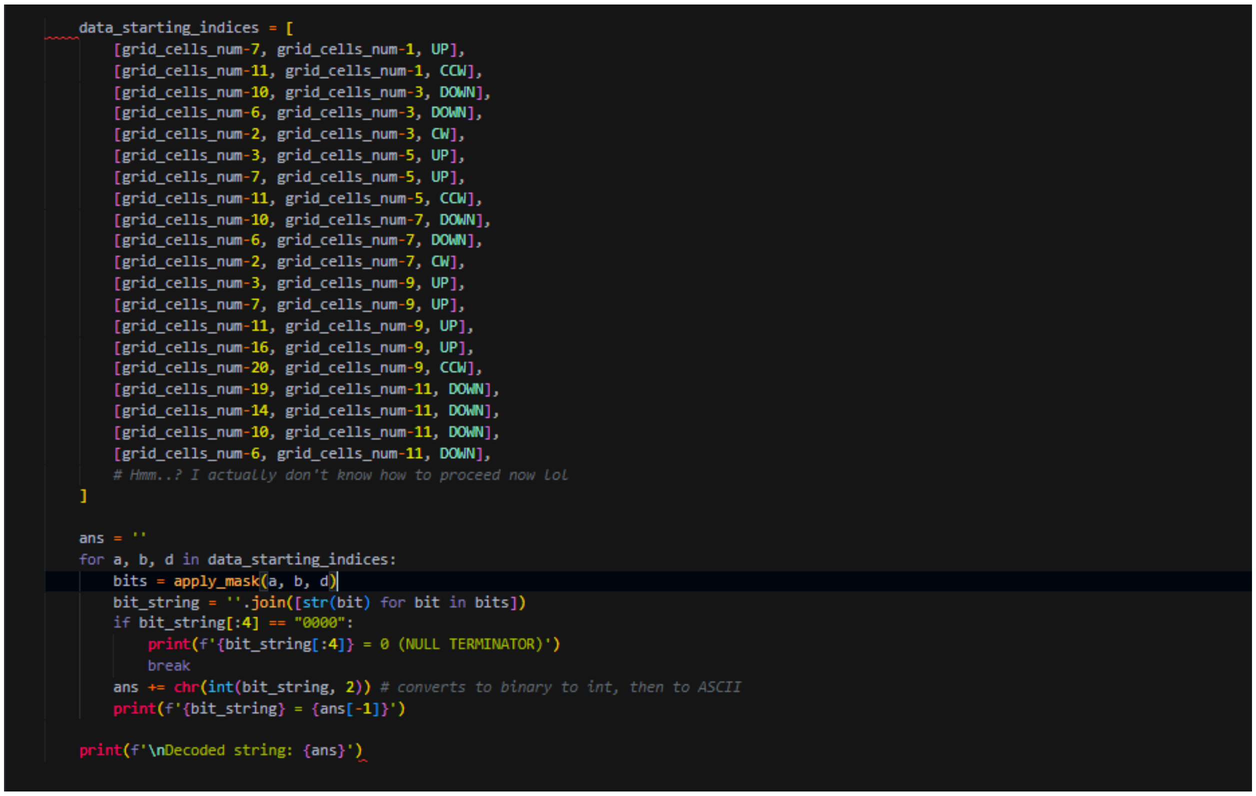
DATA CELL SEGMENTATION**:** The code reshapes the resized image into a 4D array (`qr_cells`). This essentially segments the image data into smaller sub-arrays, where each sub-array represents a single data cell containing the module states within that cell.



Data extraction

```python
# Before we proceed, let's write a function for masking to make our lives easier
UP, UP_ENC, DOWN, CW, CCW = range(5)  # A rather old-fashioned pythonic "Enum"

def apply_mask(data_start_i, data_start_j, direction):
    '''
    data_start_i/j represent the first cell's coords in its respective direction
    direction is the masking direction, up(-enc)/down/clockwise/anti-clockwise
    '''
    result = []
    row_offsets = []
    col_offsets = []
    if (direction in [UP, UP_ENC]):
        row_offsets = [0,  0, -1, -1, -2, -2, -3, -3]
        col_offsets = [0, -1,  0, -1,  0, -1,  0, -1]
    if (direction == DOWN):
        row_offsets = [0,  0,  1,  1,  2,  2,  3,  3]
        col_offsets = [0, -1,  0, -1,  0, -1,  0, -1]
    if (direction == CW):
        row_offsets = [0,  0,  1,  1,  1,  1,  0,  0]
        col_offsets = [0, -1,  0, -1, -2, -3, -2, -3]
    if (direction == CCW):
        row_offsets = [0,  0, -1, -1, -1, -1,  0,  0]
        col_offsets = [0, -1,  0, -1, -2, -3, -2, -3]

    for i, j in zip(row_offsets, col_offsets):
        cell = qr_cells_numeric[data_start_i+i, data_start_j+j]
        result.append(int(cell if MASKS[mask_str](data_start_i+i, data_start_j+j) else not cell))

    return result[:4] if direction == UP_ENC else result
```

Figure 30



```
data_starting_indices = [
    [grid_cells_num-7, grid_cells_num-1, UP],
    [grid_cells_num-11, grid_cells_num-1, CCW],
    [grid_cells_num-10, grid_cells_num-3, DOWN],
    [grid_cells_num-6, grid_cells_num-3, DOWN],
    [grid_cells_num-2, grid_cells_num-3, CW],
    [grid_cells_num-3, grid_cells_num-5, UP],
    [grid_cells_num-7, grid_cells_num-5, UP],
    [grid_cells_num-11, grid_cells_num-5, CCW],
    [grid_cells_num-10, grid_cells_num-7, DOWN],
    [grid_cells_num-6, grid_cells_num-7, DOWN],
    [grid_cells_num-2, grid_cells_num-7, CW],
    [grid_cells_num-3, grid_cells_num-9, UP],
    [grid_cells_num-7, grid_cells_num-9, UP],
    [grid_cells_num-11, grid_cells_num-9, UP],
    [grid_cells_num-16, grid_cells_num-9, UP],
    [grid_cells_num-20, grid_cells_num-9, CCW],
    [grid_cells_num-19, grid_cells_num-11, DOWN],
    [grid_cells_num-14, grid_cells_num-11, DOWN],
    [grid_cells_num-10, grid_cells_num-11, DOWN],
    [grid_cells_num-6, grid_cells_num-11, DOWN],
    # Hmm..? I actually don't know how to proceed now lol
]

ans = ''
for a, b, d in data_starting_indices:
    bits = apply_mask(a, b, d)
    bit_string = ''.join([str(bit) for bit in bits])
    if bit_string[:4] == "0000":
        print(f'{bit_string[:4]} = 0 (NULL TERMINATOR)')
        break
    ans += chr(int(bit_string, 2)) # converts to binary to int, then to ASCII
    print(f'{bit_string} = {ans[-1]}')

print(f'\nDecoded string: {ans}')
```

Figure 31

CONVERTING GRID TO NUMBERS

The provided code snippet effectively converts the QR code's data cells into a numeric representation suitable for further processing and decoding the embedded information. Let's break down the code and understand the conversion process:

1. QR_CELLS_NUMERIC ARRAY CREATION:

Python

```
qr_cells_numeric = np.ndarray((grid_cells_num, grid_cells_num),
dtype=np.uint8)
```

This line creates a new NumPy array (`qr_cells_numeric`) to store the numerical representation of the data cells.

The array has dimensions `(grid_cells_num, grid_cells_num)`, matching the number of rows and columns in the grid of data cells (`qr_cells`).

The data type is set to `uint8` (unsigned 8-bit integer), which can efficiently represent the binary values (0 or 1) extracted from each data cell.

2. LOOPING THROUGH DATA CELLS:

Python

```
for i, row in enumerate(qr_cells):
  for j, cell in enumerate(row):
    qr_cells_numeric[i, j] = (np.median(cell) // 255)
```

This nested loop iterates through each data cell within the `qr_cells` array.

The outer loop (`for i, row in enumerate(qr_cells)`) iterates over each row (`row`) of the `qr_cells` array, with `i` representing the current row index.

The inner loop (`for j, cell in enumerate(row)`) iterates over each cell (`cell`) within the current row, with `j` representing the current column index within that row.

3. CONVERTING CELL TO NUMERIC VALUE:

Python

```
qr_cells_numeric[i, j] = (np.median(cell) // 255)
```

Inside the inner loop, this line calculates a numeric value for each data cell and stores it in the corresponding position of the `qr_cells_numeric` array.

`np.median(cell)` calculates the median value of the pixel intensities within the current data cell (`cell`).

The median is used because it's less sensitive to outliers compared to the mean, especially in scenarios where there might be slight variations in pixel intensities within a cell due to noise or uneven illumination.

Dividing by 255 essentially scales the median value (which typically lies between 0 and 255) down to a value between 0 and 1. This conversion works because QR code data modules are typically represented by either dark pixels (closer to 0) or light pixels (closer to 255).

4. UNDERSTANDING THE RESULT (QR_CELLS_NUMERIC):

The resulting `qr_cells_numeric` array now holds a binary representation of the QR code's data.

A value close to 0 represents a dark module (interpreted as binary 1).

A value close to 1 represents a light module (interpreted as binary 0).

5. EXTRACTING ERROR CORRECTION LEVEL (ECL):

Python

```
print(qr_cells_numeric[8]) # We want row #8

ecl = [int(not(c)) for c in qr_cells_numeric[8, 0:2]]

# Why "not"? Because the standard uses '1's for black and '0's for white

print(ecl)
```

This code snippet focuses on extracting the Error Correction Level (ECL) from the `qr_cells_numeric` array.

It prints the contents of row 8 (`qr_cells_numeric[8]`), which likely contains the data related to ECL and mask information within the QR code structure (refer to QR code documentation for specific row/column placements).

It then extracts the first two elements (`[8, 0:2]`) from this row, which typically hold the bits corresponding to the ECL.

The list comprehension `[int(not(c)) for c in ...]` inverts the bits because QR code standards define dark modules as binary 1 and light modules as binary 0. In image processing, we often use the opposite convention. The `not()` operation flips the bits (0 becomes 1 and vice versa).

Finally, it prints the `ecl` list, which now contains the two binary digits representing the ECL level (e.g., `[1, 1]` for Low level, `[1, 0]` for Medium level, and so on).

By converting the data cells into numeric form and extracting the ECL, the code prepares the data for subsequent decoding steps. These steps might involve applying error correction algorithms based on the identified ECL level and interpreting the

MASKING IN QR CODES: ADDING REDUNDANCY FOR ROBUSTNESS

QR codes, despite their compact design, incorporate masking patterns as an error correction technique. Here's how masking works and why it's crucial for reliable data encoding:

THE MASKING PROCESS:

DATA ENCODING**:** The information to be encoded is first converted into binary data (a sequence of 0s and 1s). This binary data forms the core of the QR code's message.

ERROR CORRECTION: QR codes employ error correction mechanisms to ensure data integrity during transmission or scanning. These mechanisms add redundant data bits based on the original data, allowing the decoder to rectify potential errors.

MASKING PATTERNS**:** After error correction is applied, the QR code structure undergoes masking. Different predefined mask patterns, represented by binary strings, are applied to the data and error correction bits. Each mask pattern follows a specific rule that determines whether a data bit should be flipped (inverted) or left unchanged.

MASKING FUNCTION**:** The masking process involves iterating through each data cell (a small square grid within the QR code) and applying the chosen mask pattern. The mask pattern dictates whether the corresponding bit in the data cell should be inverted based on the cell's location and the mask pattern's rule.

BENEFITS OF MASKING:

IMPROVED ERROR DETECTION: Masking helps in detecting errors during the decoding process. By introducing a controlled pattern of bit inversions, masking breaks up large areas of similar data bits (either all 0s or all 1s) that might occur due to the error correction process. This helps the decoder differentiate true data from potential errors.

REDUCED BLURRY DETECTION ISSUES: QR codes might be scanned under less than ideal conditions, such as with blurry images or uneven lighting. Masking patterns can help mitigate these issues by ensuring sufficient transitions between dark and light modules within the code. This improves the scanner's ability to accurately detect the encoded data.

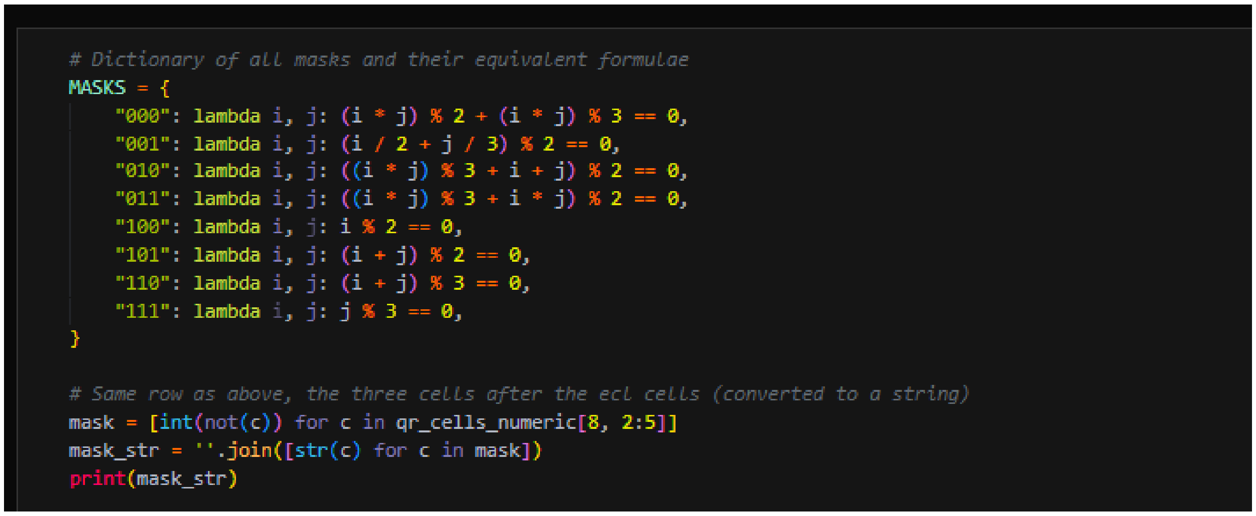
SYNCHRONIZATION AID: Masking patterns also serve as a synchronization mechanism for the decoder. The specific pattern of bit inversions helps the decoder locate the start of the data efficiently, especially if the QR code is partially damaged or scanned at an angle.

MULTIPLE MASK PATTERNS:

QR codes define a set of eight different mask patterns. The selection of the specific mask pattern to be applied is typically based on an optimization process that considers factors like the number of bit transitions within the code and the overall error correction level used.

DECODING AND UNMASKING:

The QR code decoder first identifies the mask pattern used for encoding, often by analyzing dedicated format information areas within the QR code structure. Once the mask pattern is known, the decoder can apply the unmasking process, essentially inverting the bits that were flipped during masking based on the specific mask pattern rule. This reveals the original data and error correction bits for further processing.



```python
# Dictionary of all masks and their equivalent formulae
MASKS = {
    "000": lambda i, j: (i * j) % 2 + (i * j) % 3 == 0,
    "001": lambda i, j: (i / 2 + j / 3) % 2 == 0,
    "010": lambda i, j: ((i * j) % 3 + i + j) % 2 == 0,
    "011": lambda i, j: ((i * j) % 3 + i * j) % 2 == 0,
    "100": lambda i, j: i % 2 == 0,
    "101": lambda i, j: (i + j) % 2 == 0,
    "110": lambda i, j: (i + j) % 3 == 0,
    "111": lambda i, j: j % 3 == 0,
}

# Same row as above, the three cells after the ecl cells (converted to a string)
mask = [int(not(c)) for c in qr_cells_numeric[8, 2:5]]
mask_str = ''.join([str(c) for c in mask])
print(mask_str)
```

Figure 32

REFERENCES

1- https://en.wikipedia.org/wiki/QR_code
2- https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders
3- https://www.thonky.com/qr-code-tutorial