

# Assignment #5 – Geometric Transformations

Faculty of Engineering Science

Dept. of Electrical and Computer Engineering

Introduction to Digital Image Processing, 361-1-4751

Maor Assayag 318550746

Refahel Shetrit 204654891

## 1 QR Code Reader

In this assignment we will learn geometric transformations on the image domain. In this exercise we will learn to read a simplified version of the QR Code. This is the main part of the assignment.

### 1.1 Generating the QR image

Insert your 9-digit ID to the function  $ID2QR(id)$  where ID is a string. Print the QR code on paper. The QR code is a matrix of 6x6 binary values.

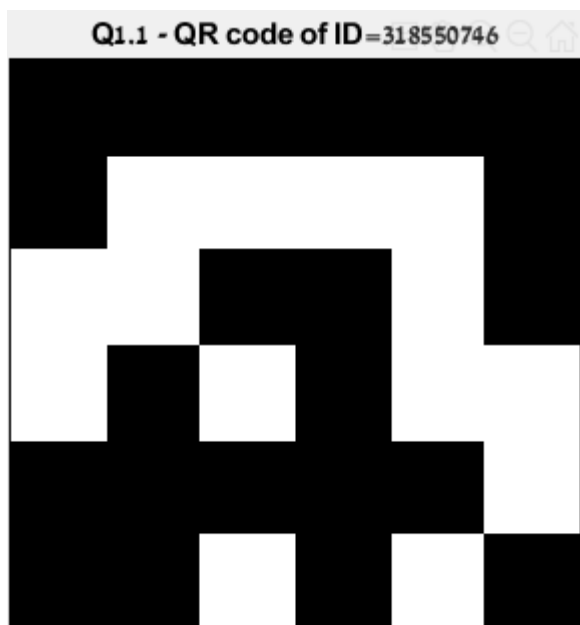


Figure 1.1.1

**Explanation :** The initial matrix is a 6x6 binary matrix, the outputted QR is a stretch out 258x258 grayscale image.

## 1.2 Generating 3 photos of the QR

Take three photos of the QR code from three different angles: easy, intermediate, hard. The hard photo should be one that is on the limit where you can no longer properly interpret the QR code.

**Note :** in page 13 we took 3 photos of the printed QR and showed the results of our algorithm.

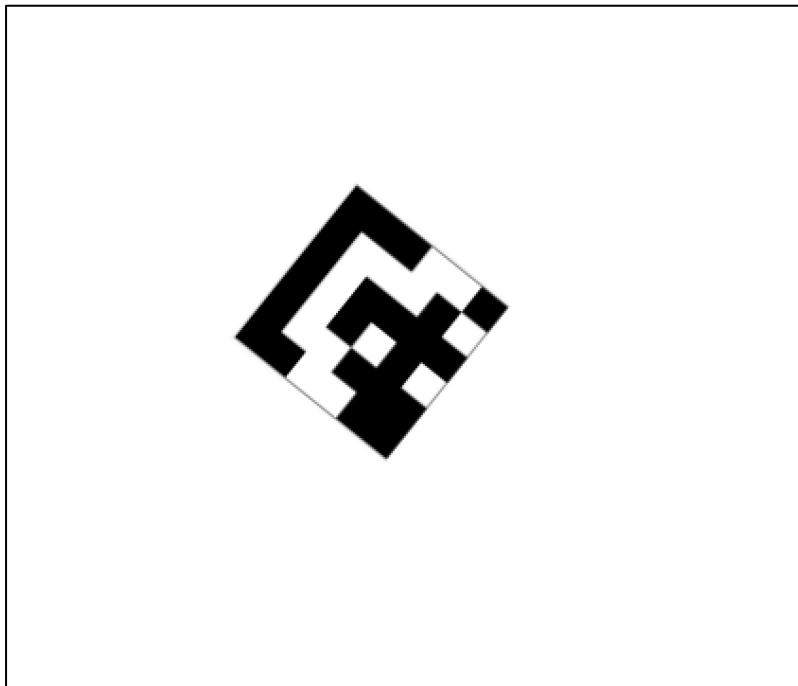


Figure 1.2.1 – image 1

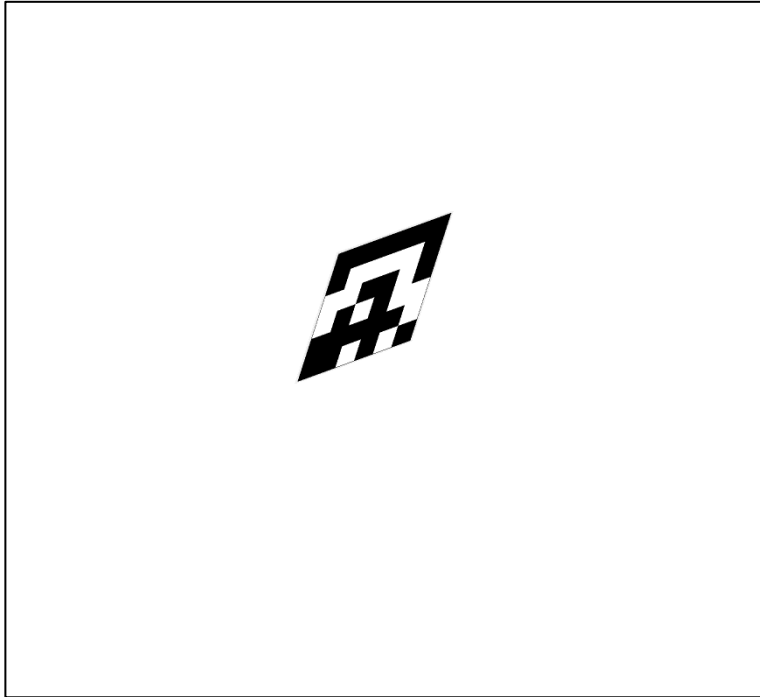


Figure 1.2.2 – image 2

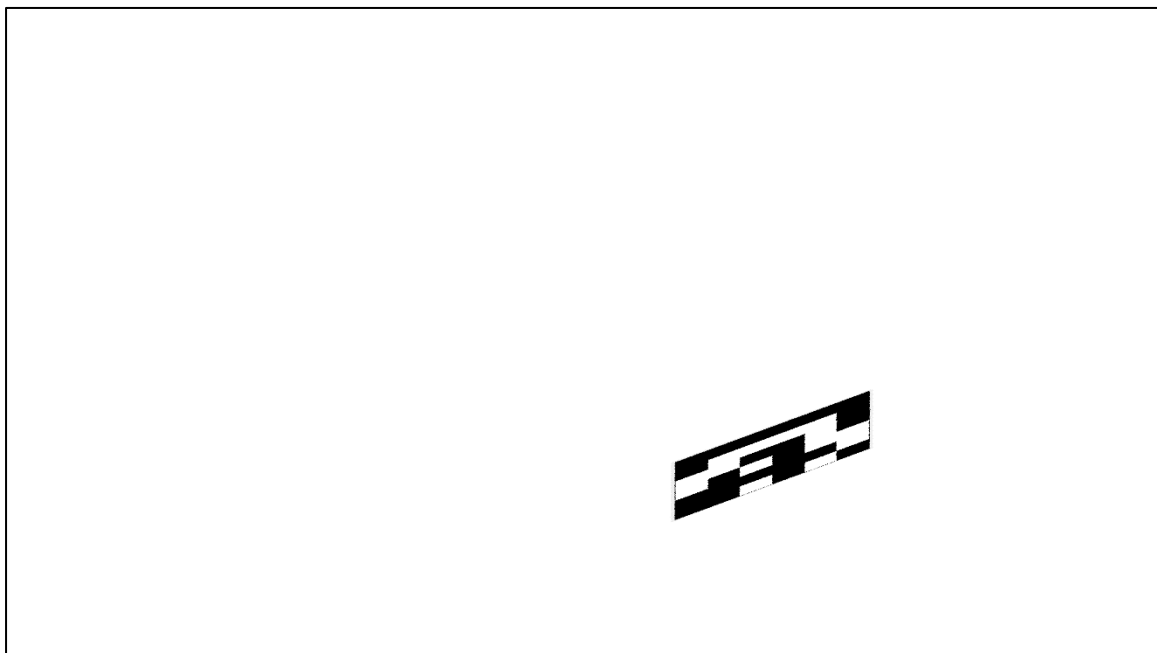


Figure 1.2.3 – image 3

**Explanation :** We generated 3 difference images representing the QR code captured at 3 different perspectives. Those equivalent to take photos of a printed QR code. We read the images and convert them to grayscale using built-in MATLAB function `'rgb2gray()`'. A photo can be RGB (e.g. the example in the instruction) and we want to work in the normalize Grayscale color space.

### 1.3 Generating Corners

Locate the points on the QR code in each image manually. You may use MATLAB's *ginput(4)*.

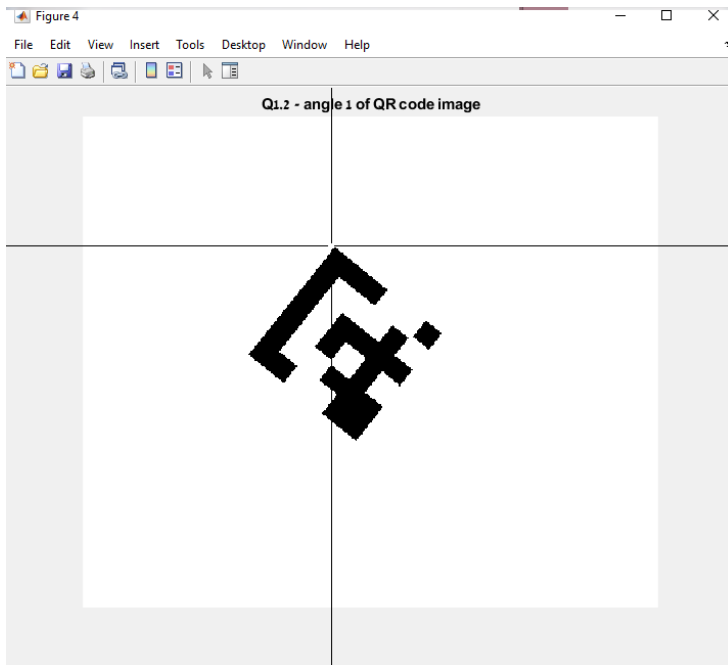


Figure 1.3.1

**Explanation :** We created 4 arrays of 2x4 represent corners coordinates of the original QR code (which will be just the edges of a 258x258 image) and the 3 photos of the QR code. Then we save those into .mat file which will be loaded for now on.

```
%% saving corners as .mat file
save image_corners image_corners
save image1_corners image1_corners
save image2_corners image2_corners
save image3_corners image3_corners
```

Figure 1.3.2

```
% Load corners mat files
load('image_corners.mat');
load('image1_corners.mat');
load('image2_corners.mat');
load('image3_corners.mat');
```

Figure 1.3.3

**Note :** here the original QR image is not needed, we only need the edges of an 258x258 matrix indexes but kept the original image for display purposes.

## 1.4 Transform

Transform the image such that the QR code is straightened. For each image use all the transformations learned in class: Rigid (rotation, translation and scale transformation), Affine (shearing added), Perspective.

### Explanation :

We built a function called `'dip_recoverQR()'` which gets as inputs the following :

*reference\_image* - binary 258x258 image represent QR size

*reference\_image\_corners* - 2x4 double array represent *reference\_image* 4 corners

*image2recover* - grayscale image containing the QR, unknown size

*image2recover\_corners* - 2x4 double array represent *reference\_image* 4 corners

First the function is using the built-in MATLAB function `'fitgeotrans()'`, which fit geometric transformation to control point pairs. Our control points are the corners of the reference image and the taken image (1/2/3).

`tform = fitgeotrans(movingPoints, fixedPoints, transformationType)` takes the pairs of control points, *movingPoints* and *fixedPoints*, and uses them to infer the geometric transformation specified by *transformationType*.

First we will try to use Rigid transformation, specify by `'nonreflectivesimilarity'`. Then we will try `'affine'` for affine and `'projective'` for perspective.

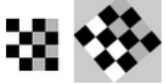



Transformation Type	Description	Minimum Number of Control Point Pairs	Example
'nonreflective similarity'	Use this transformation when shapes in the moving image are unchanged, but the image is distorted by some combination of translation, rotation, and scaling. Straight lines remain straight, and parallel lines are still parallel.	2	
'similarity'	Same as 'nonreflective similarity' with the addition of optional reflection.	3	
'affine'	Use this transformation when shapes in the moving image exhibit shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles become parallelograms.	3	
'projective'	Use this transformation when the scene appears tilted. Straight lines remain straight, but parallel lines converge toward a vanishing point.	4	

Figure 1.4.1 – source MATLAB

When we tried to categorize the type on transformation (e.g. stop at Rigid) we used the built-in MATLAB functions like 'isRigid()' which determine if transformation is rigid transformation. For our understanding, we see that the transformation matrix isn't as Rigid matrixs defined :

tform_tryRigid.T		
1	2	3
0.8165	1.0144	0
-1.0144	0.8165	0
153.6104	-534.7043	1

Figure 1.4.2

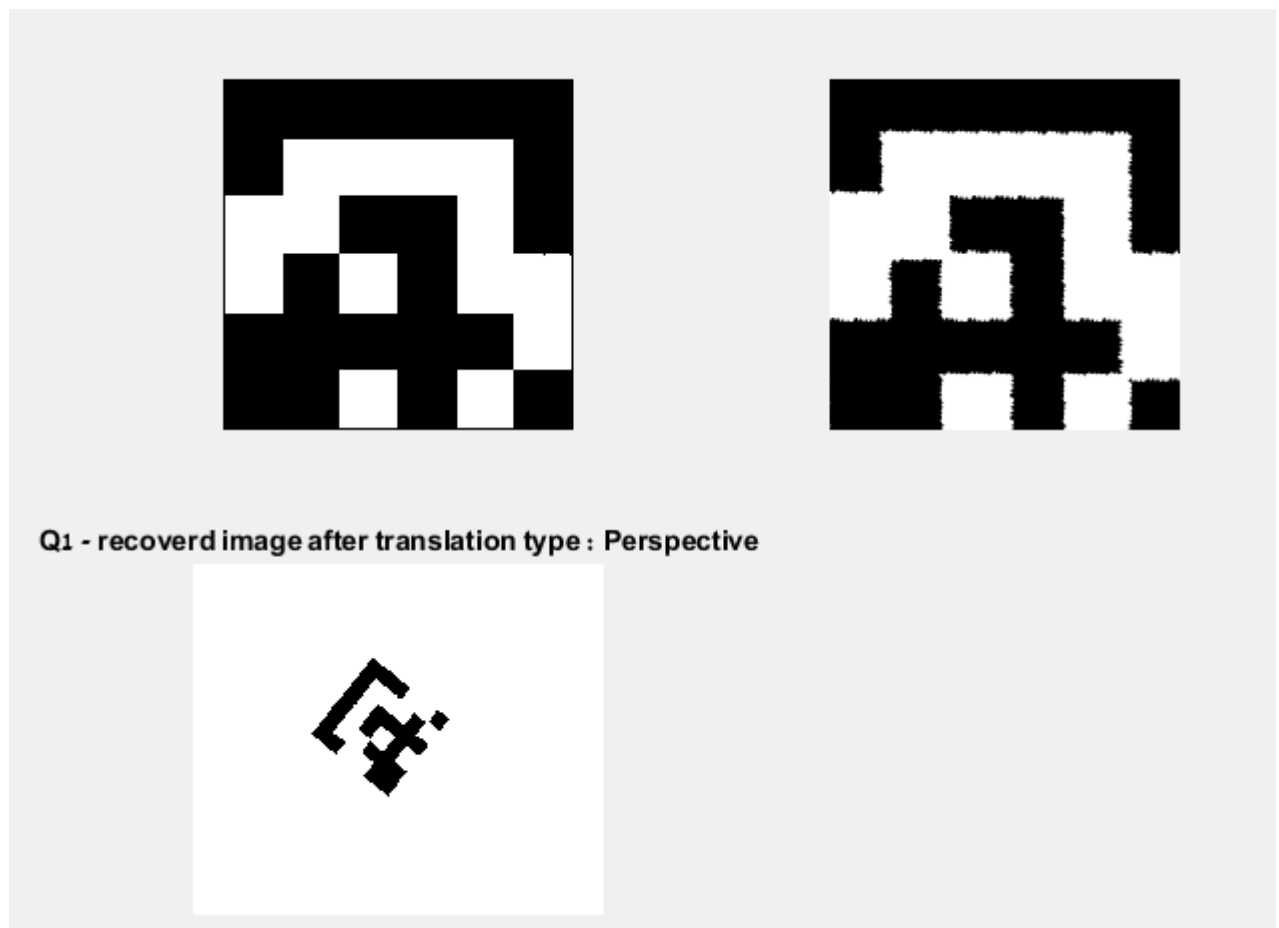


Figure 1.4.3 – image 1 QR

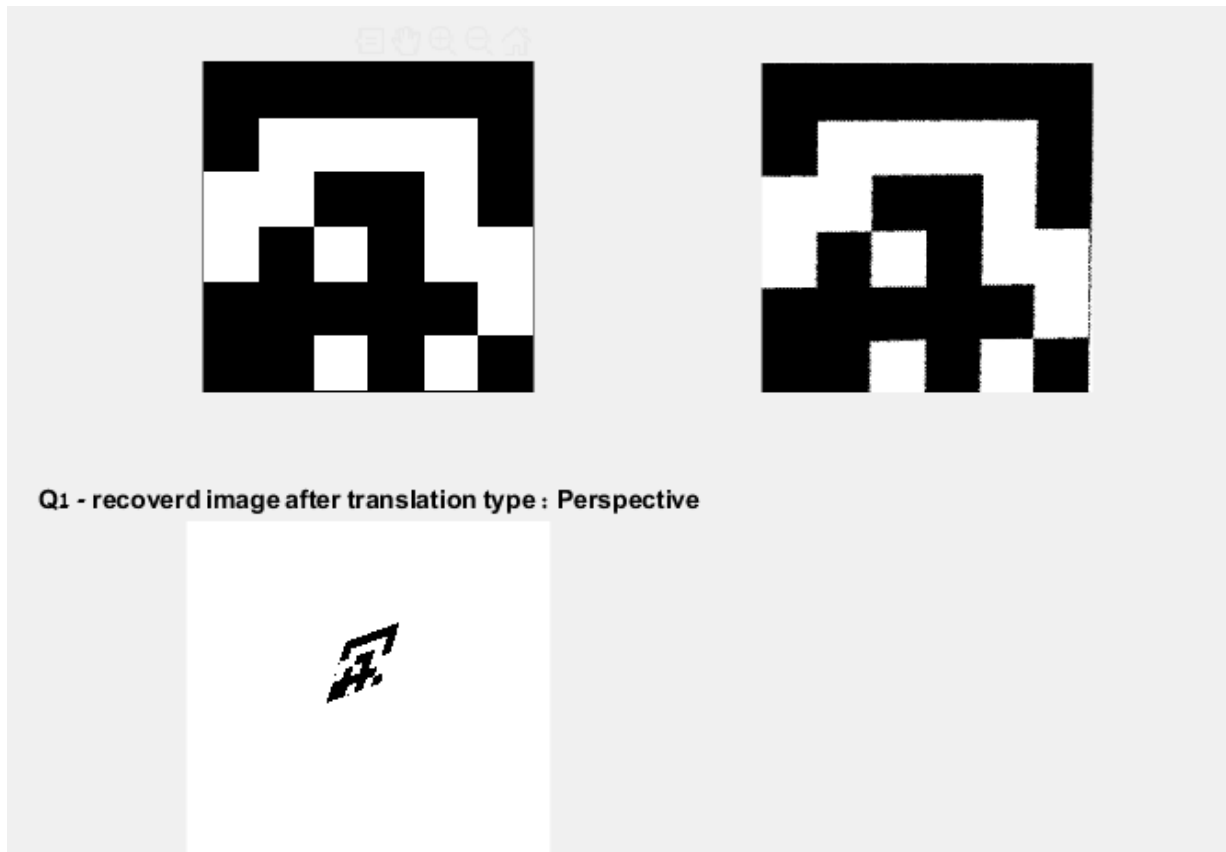


Figure 1.4.4 – image 2 QR

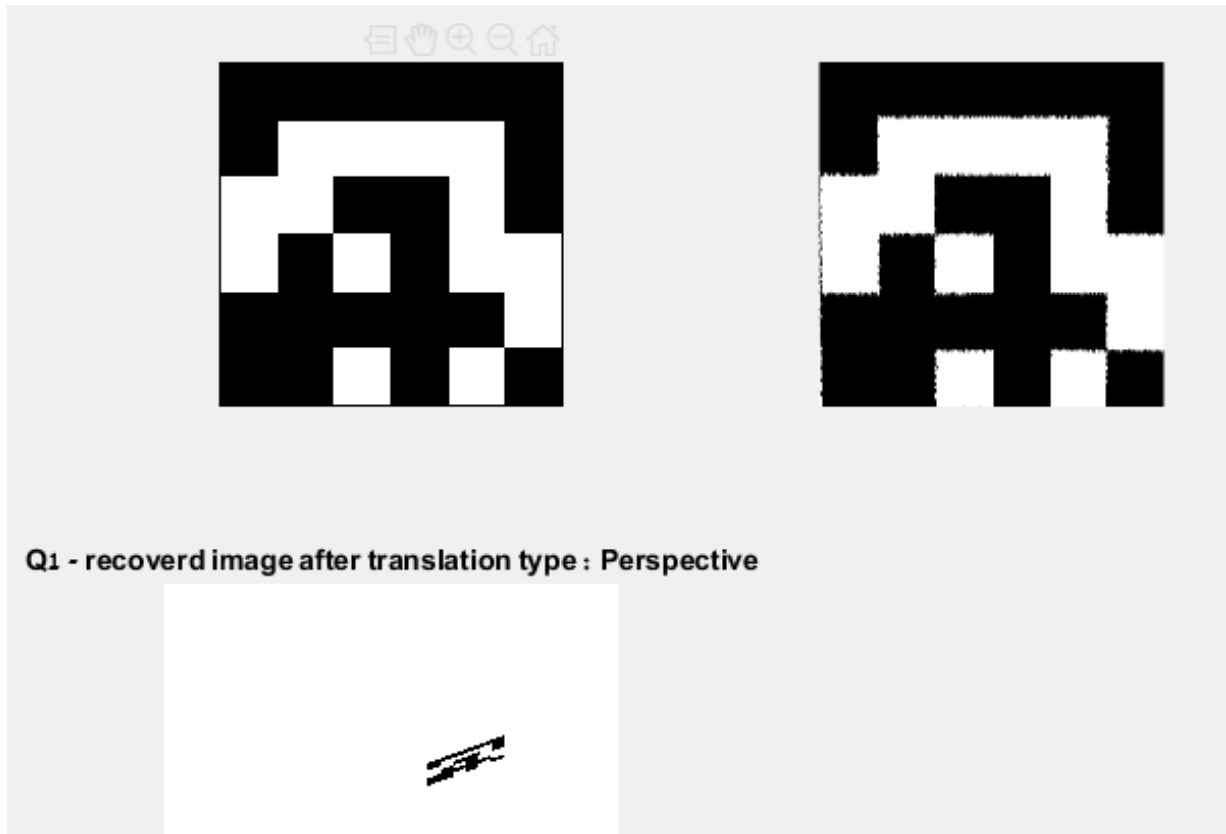


Figure 1.4.5 – image 3 QR

## 1.5 Explain

Explain the result of each transformation and why does it work/fail for the given image.

First we will try Rigid transformation on all images :

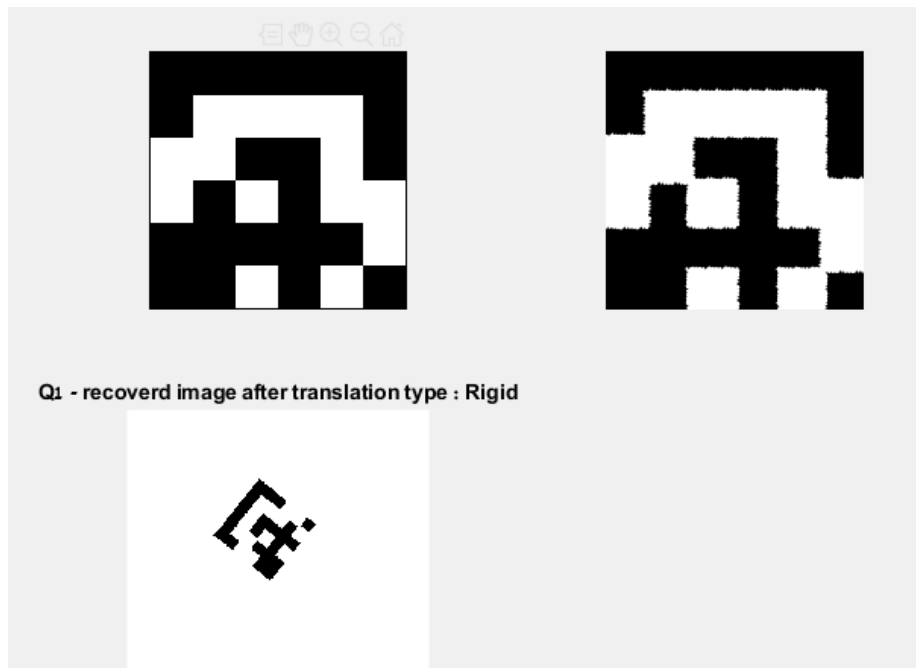


Figure 1.5.1 – image 1 Rigid transformation – success

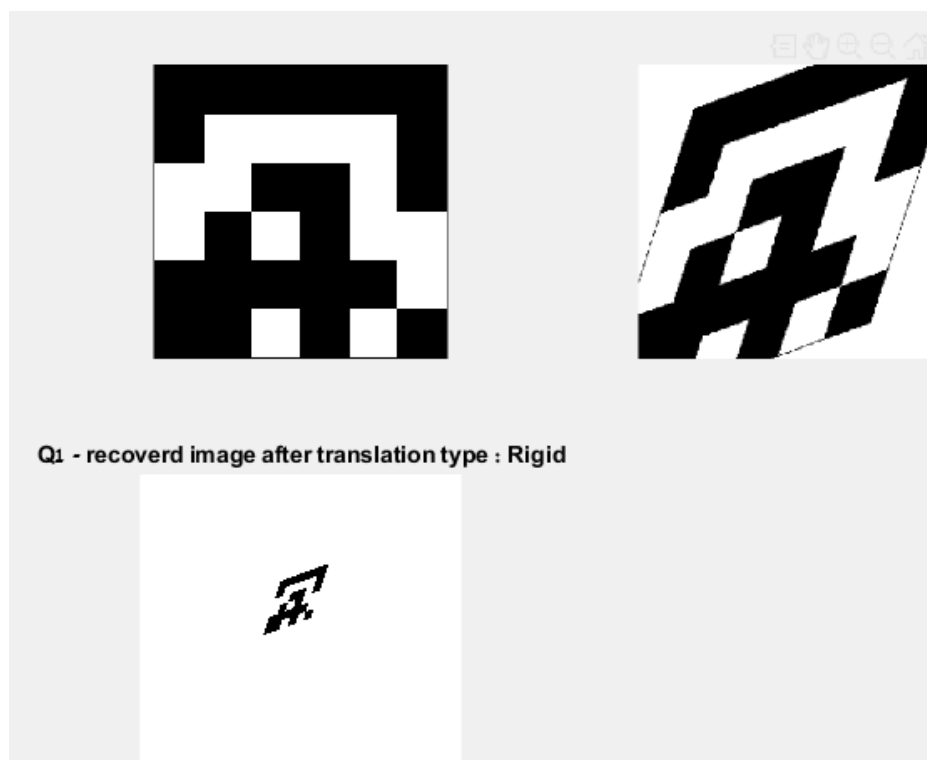


Figure 1.5.2 – image 2 Rigid transformation – failure



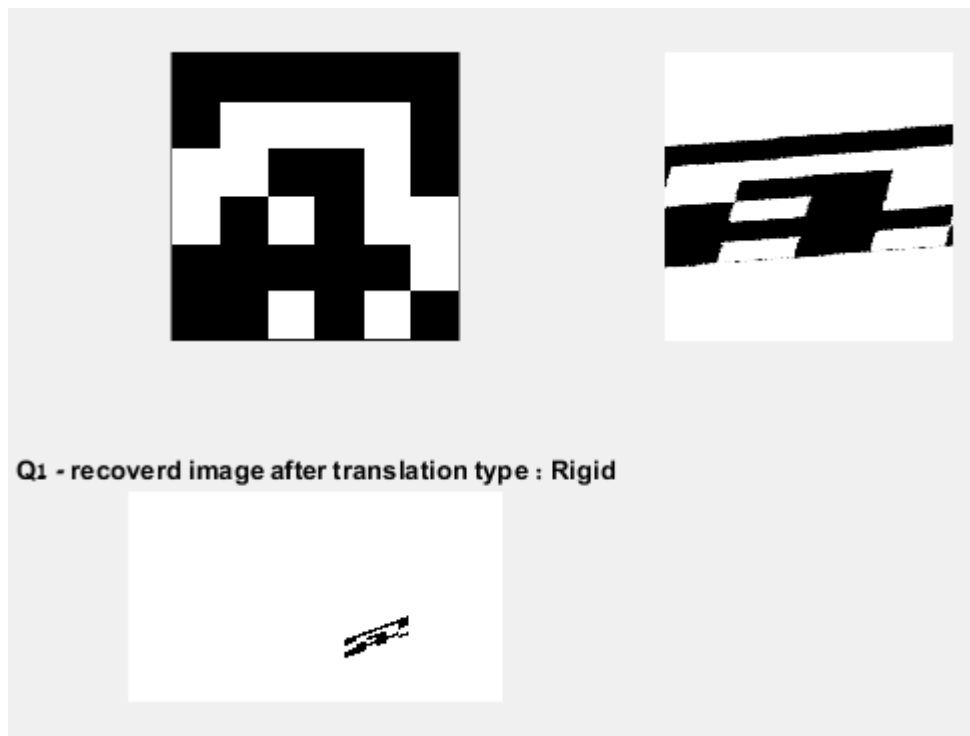


Figure 1.5.3 – image 3 Rigid transformation – failure

**Explanation:** The first image make sense to results a straightened QR because we can see that the image is consist of translation, rotation and scaling of the original image. Unlike image 1 - image 2 and 3 have more components of transformation to them (e.g. sheer and perspective), hence Rigid transformation will not suffice for them.

After that, we tried Affine transformation and got suffice results for all 3 images As shown at Figures 1.4.3-5.

## 1.6 Extract binary values

Extract the binary values from the straightened QR. The matrix is row major meaning that the correct order is (1,1), (2,1), ..., (6,1), (1,2), (2,2),....., (6,6).

```
QR_binary =

    0    0    0    0    0    0
    0    1    1    1    1    0
    1    1    0    0    1    0
    1    0    1    0    1    1
    0    0    0    0    0    1
    0    0    1    0    1    0
```

Figure 1.6.1 – binary matrix extracted from a QR image

### Explanation :

We built a function called '*dip\_QR2Binary\_raw 0*' which gets as inputs the following :  
*straightened\_QR - 258x258 grayscale image represent straightene recoverd QR.*

First the function crops the 2-pixel wide frame the the algorithm ID2QR added to the QR code. Then we divided the QR image to 36 sections, each section size is (scale x scale), when scale is 256/6 resulting from using *imresize()* on the binary map created in the function ID2QR.

For each section we computed the mean value and rounded it to a binary value (0 or 1). The round function helps us avoid the error margin from the transformation process, angle of the photo etc.'

Finally, we used the built-in MATLAB function '*reshape*' to convert the 6x6 binary matrix to an outputted 1x36 binary vector, represents the digits of the id.

## 1.7 Extract ID from binary

Convert every 4 bits to an integer: e.g. 0110 > 6, 1001 > 9.

### Explanation:

We built a function called '*dip\_binaryQR2str()*' which gets as inputs the following :

*binary\_vector* - 6x6 binary row vector represents a QR

For each digit we are extracting the next 4 binary values from the input and converting it to integer digit using the built-in MATLAB function '*bin2dec()*'.

The output is a string of the recovered ID.

```
if (strcmp(id,id_result_1)==1)
    display('Q1 is accurate for image 1');
end
if (strcmp(id,id_result_2)==1)
    display('Q1 is accurate for image 2');
end
if (strcmp(id,id_result_3)==1)
    display('Q1 is accurate for image 3');
end
```

Figure 1.7.1 – we checked the results for the 3 images

```
Q1 is accurate for image 1
Q1 is accurate for image 2
Q1 is accurate for image 3
```

Figure 1.7.2 – results

## 1.8 Algorithm flow

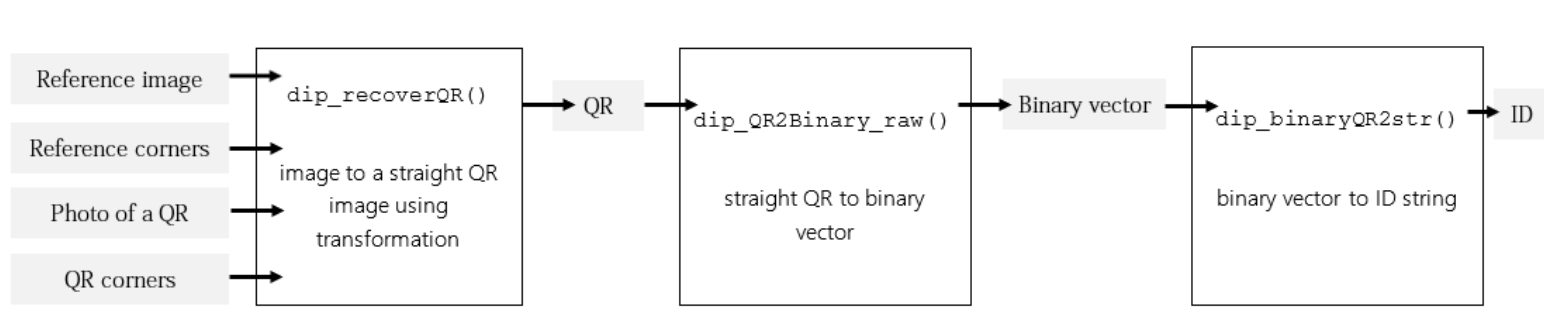


Figure 1.8.1

### Explanation:

'*dip\_imageQR2ID()*' compute the algorithm as a whole, calling all the above explained functions.

Function inputs :

*reference\_image* - binary 258x258 image represent QR size

*reference\_image\_corners* - 2x4 double array represent *reference\_image* 4 corners

*image2recover* - grayscale image containing the QR, unknown size

*image2recover\_corners* - 2x4 double array represent *reference\_image* 4 corners

Function outputs :

*id* - string ID extracted from the binary vector input

## 1.9 Trial with 3 photos

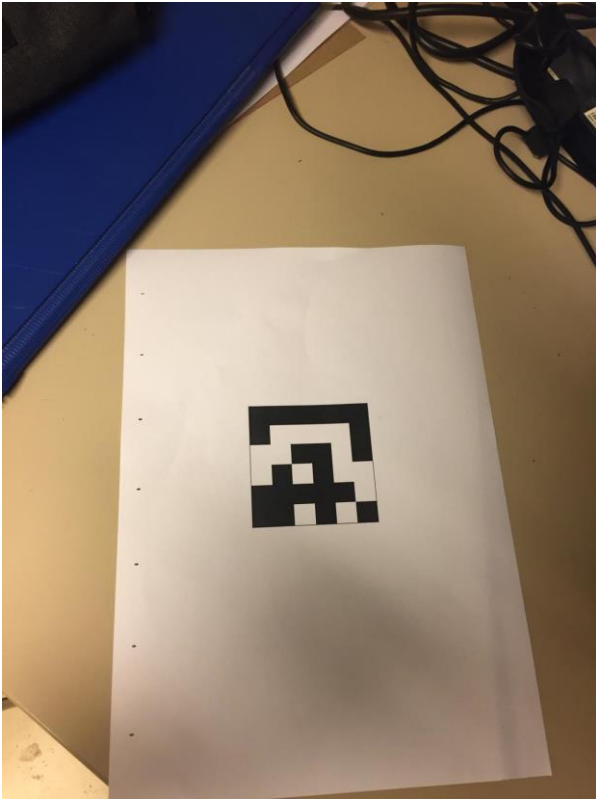


Figure 1.9.1 – original photo 1

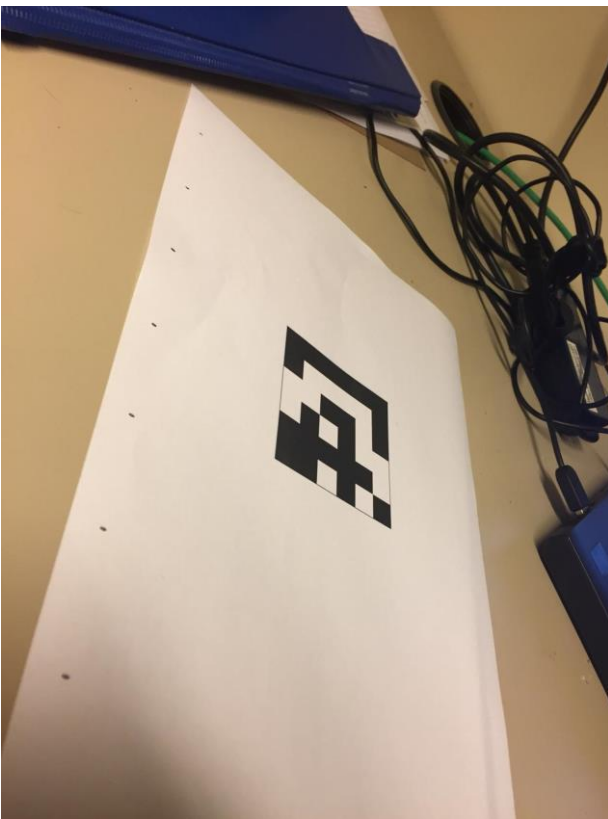


Figure 1.9.2 – original photo 2

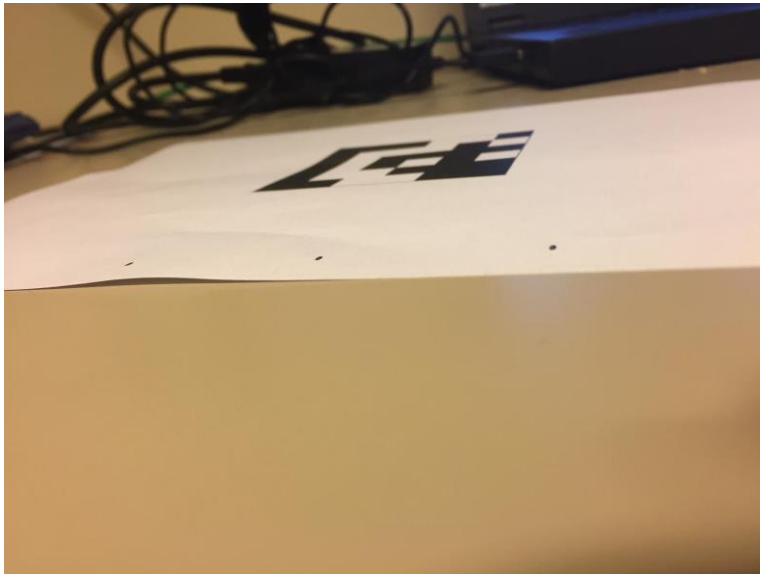


Figure 1.9.3 – original photo 3

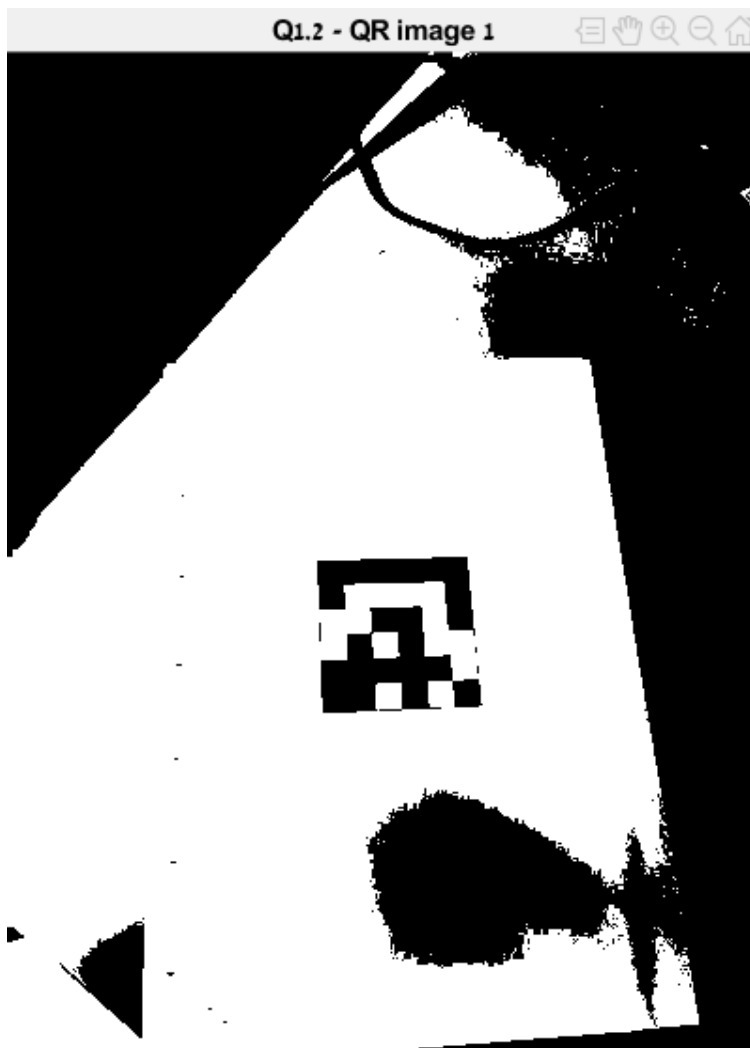


Figure 1.9.4 – read photo 1 – grayscale

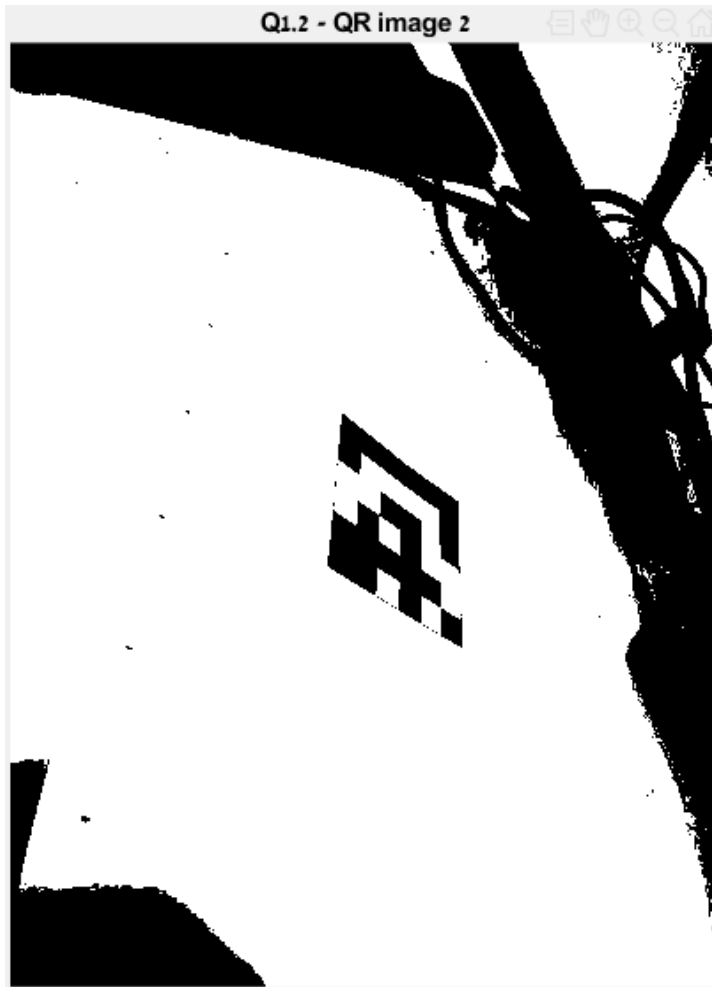


Figure 1.9.5 – read photo 2 - grayscale

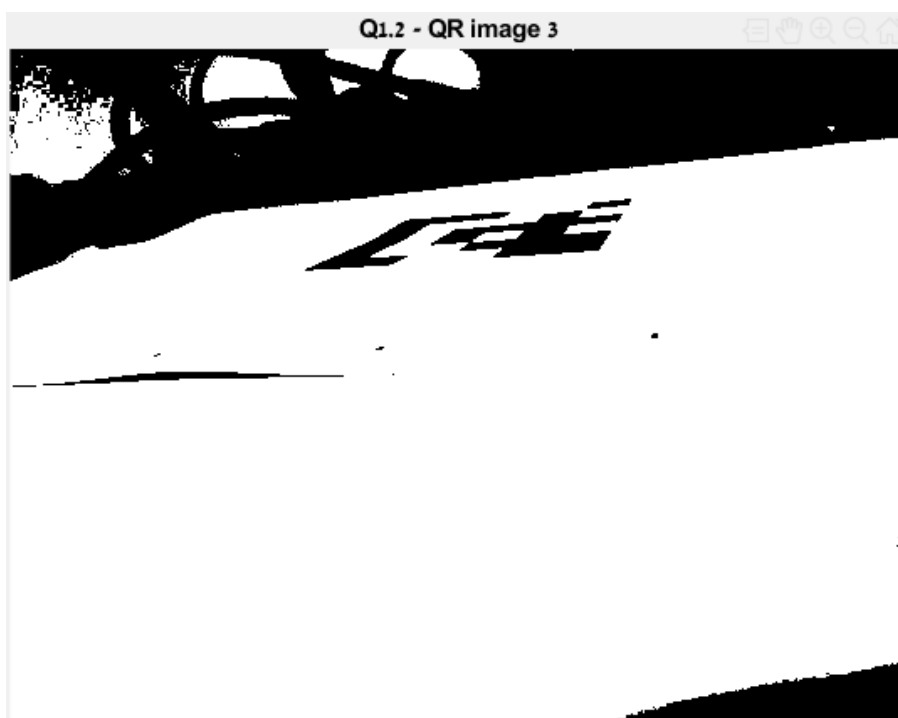


Figure 1.9.6 – read photo 3 - grayscale

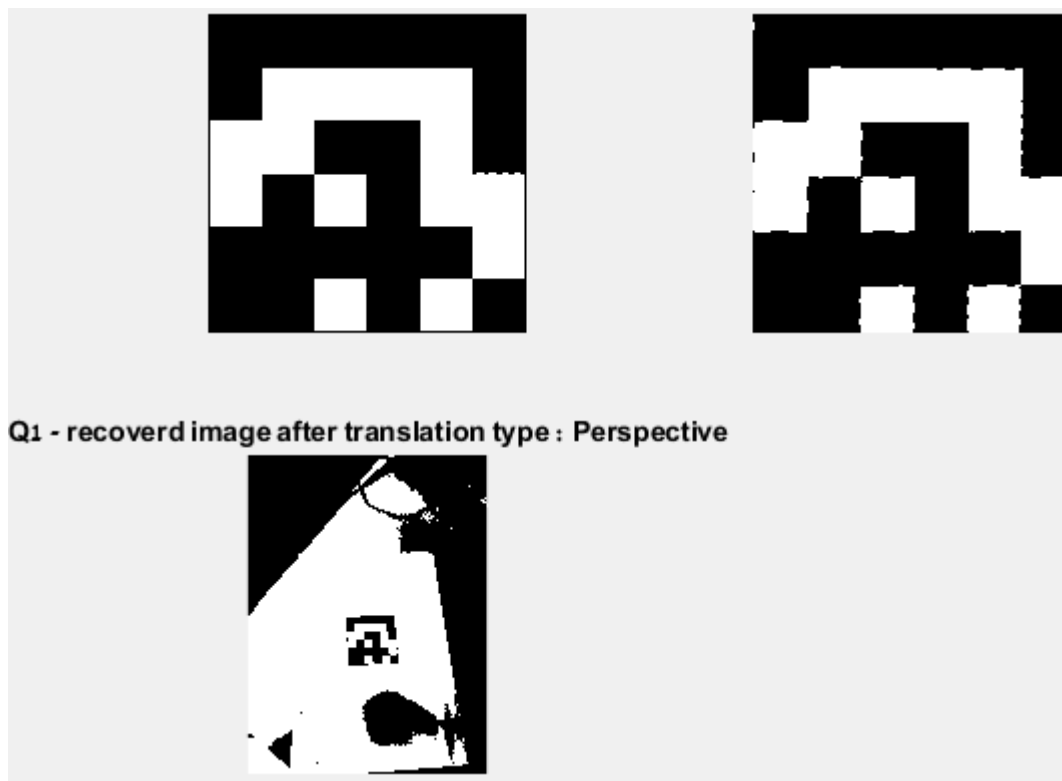


Figure 1.9.7 – photo 1 transformation

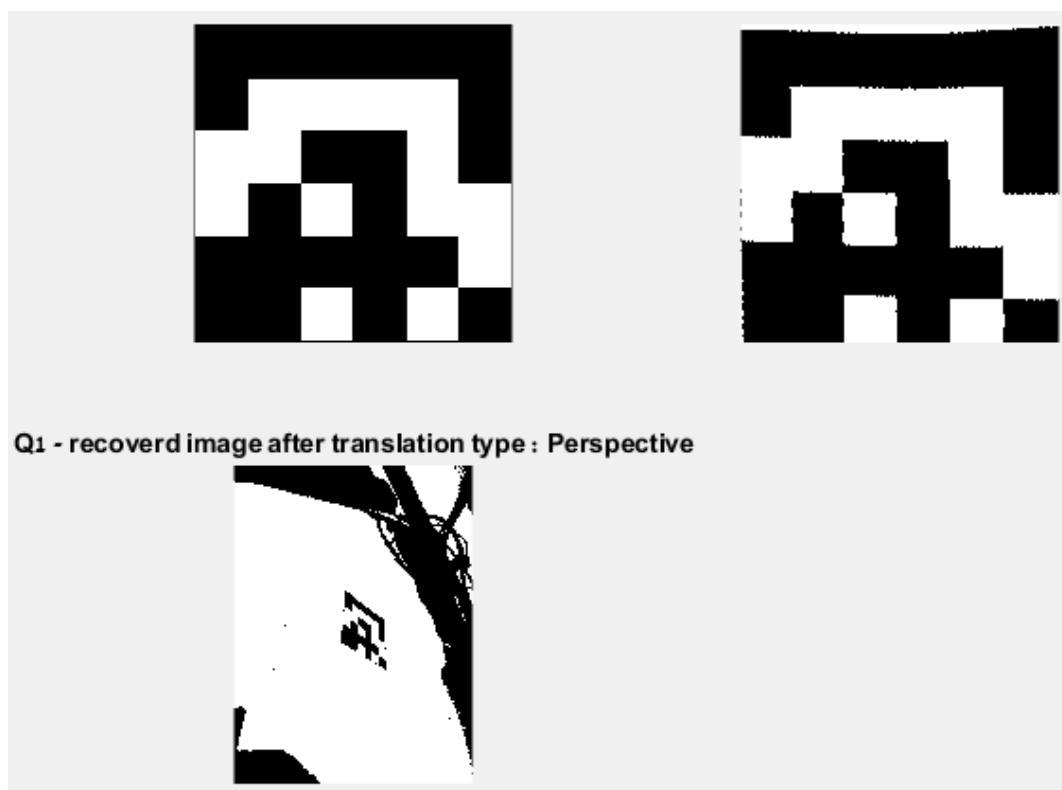


Figure 1.9.8 – photo 2 transformation



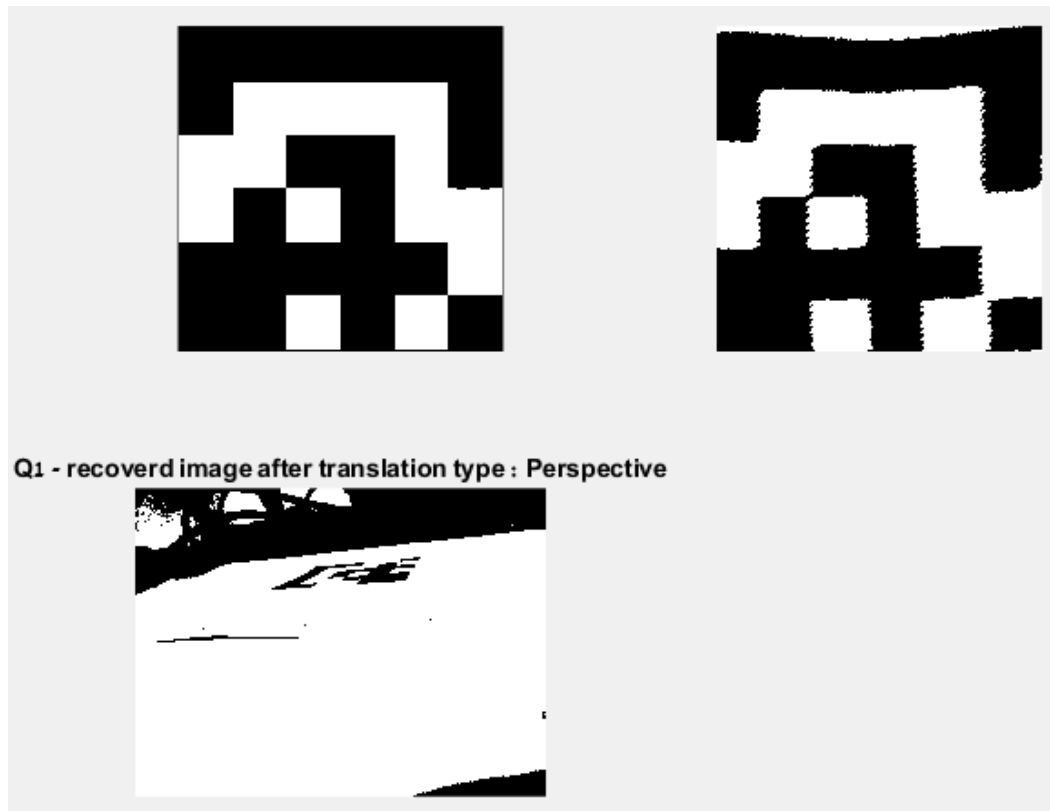


Figure 1.9.9 –

photo 3 transformation

Q1 is accurate for image 1, the result ID for 318550746 is 318550746  
 Q1 is accurate for image 2, the result ID for 318550746 is 318550746  
 Q1 is accurate for image 3, the result ID for 318550746 is 318550746

Figure 1.9.10 – results of the 3 photos

## 2 Automatic Corner Detector

In this exercise you will use previously learned subjects to design an automatic corner detection algorithm for the QR code reader.

This detector will be used to replace the manual input of the previous exercise.

### 2.1 Algorithm flow

**Explanation:**

'*dip\_detecCorners(image\_org)*' detect the corners of the QR image.

Function inputs :

*Image\_org* – any RGB image (need rgb for displaying result only)

Function outputs :

*corners* – detected QR object corners in the image, starting clockwise from upper left corner.

**Note :** in a regular QR we have 3 squares corners that helps us detect the rotation of the image and rotate it to the correct order. In our QR generator this attribute is missing, hence we need to check for the correct order of the corners that have been detected (explained how later).



Figure 2.1.1 – regular QR

## Step 1 - Preparation for boundaries

### 1. Original image as input

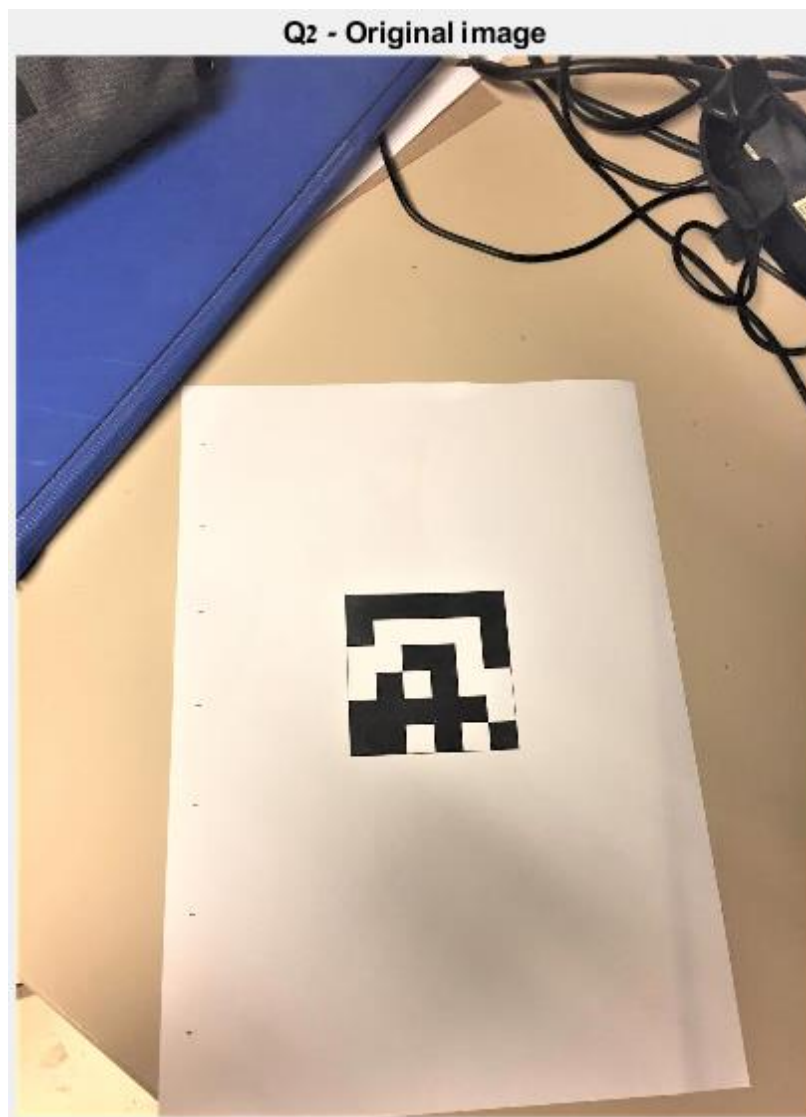


Figure 2.1.1.1

## 2. Convert the original image to Grayscale image

Using  $BW = im2bw(I, level)$  converts the grayscale image  $I$  to binary image  $BW$ , by replacing all pixels in the input image with luminance greater than  $level$  with the value 1 (white) and replacing all other pixels with the value 0 (black).

This is the first 'filter' to noise in our image.

We checked several levels and found 0.65 working well in our light environment.

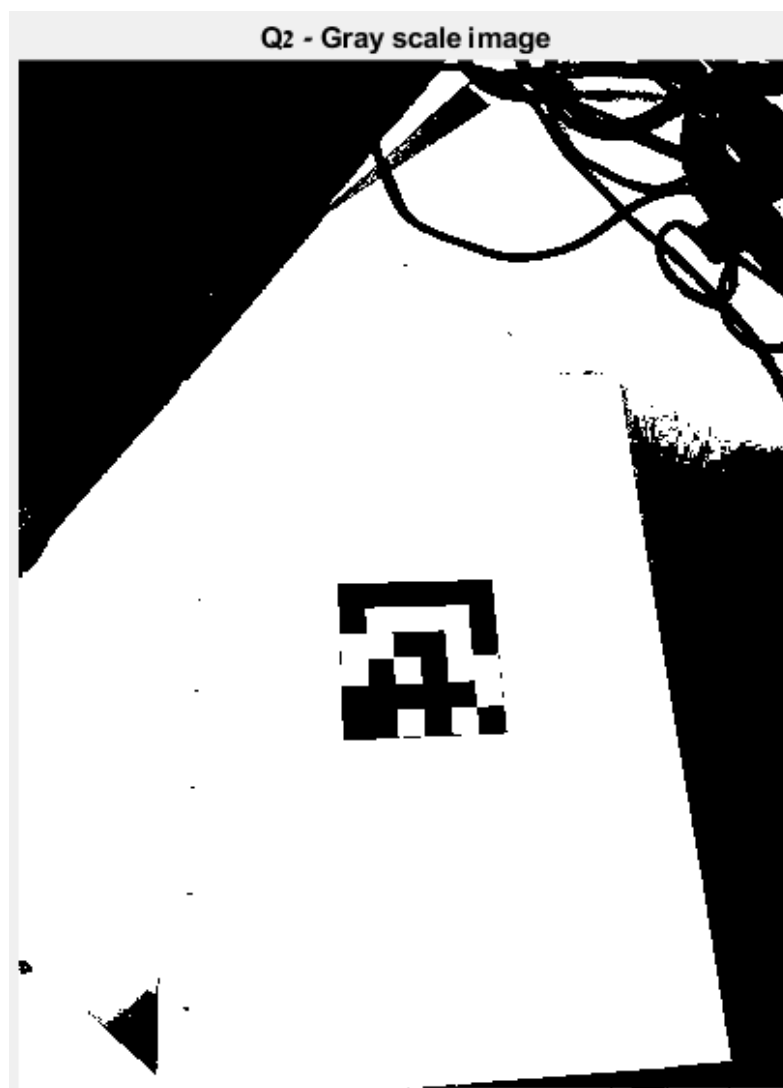


Figure 2.1.1.2

### 3. Canny's edge detector

By using  $BW = \text{edge}(\text{grayscale\_I})$  that returns a binary image BW containing 1s where the function finds edges in the input image I and 0s elsewhere using Canny method. The Canny method is less likely than the other methods to be fooled by noise, and more likely to detect true weak edges.

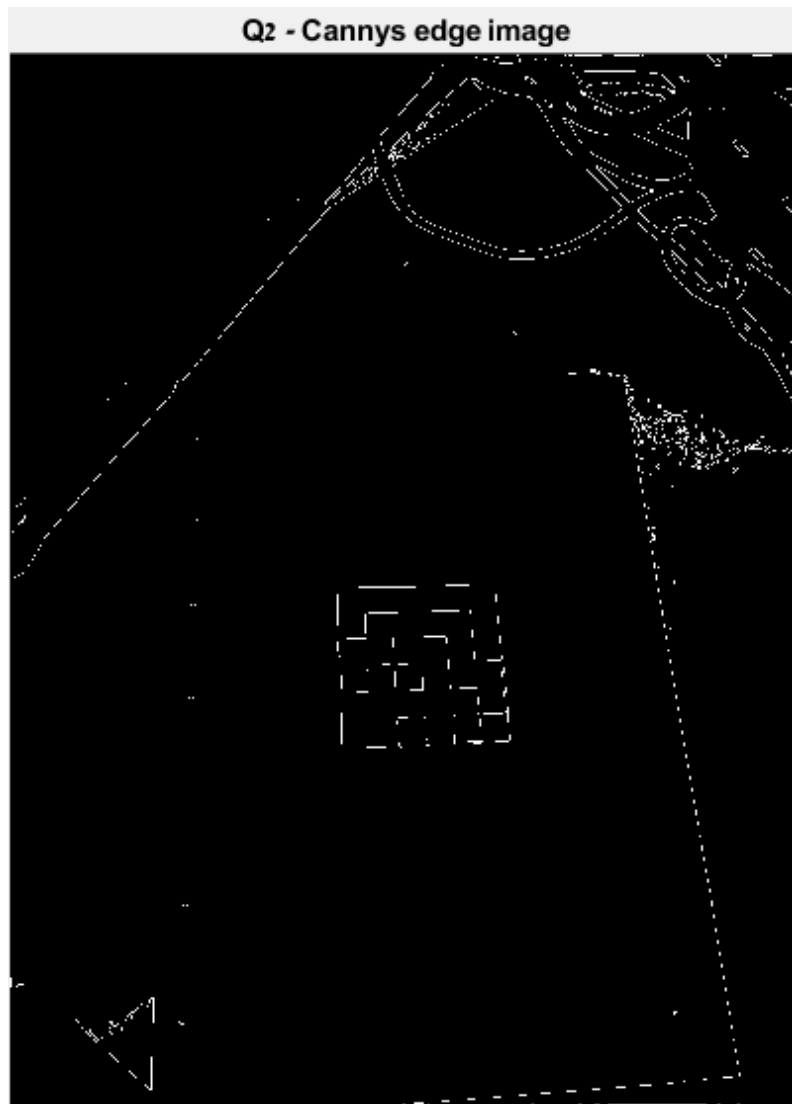


Figure 2.1.1.3

#### 4. Entropy filter

By using  $J = \text{entropyfilt}(I)$  that returns the array J, where each output pixel contains the entropy value of the 9-by-9 neighborhood around the corresponding pixel in the input image I.

This step will help up to define the QR shape in more details, which will help us fill the QR shape next.



Figure 2.1.1.4

### 5. Fill holes in the binary image

By using  $BW2 = \text{imfill}(BW, 'holes')$  that fills holes in the input binary image  $BW$ . In this syntax, a hole is a set of background pixels that cannot be reached by filling in the background from the edge of the image.

By doing so, we are avoiding the inner logic of the QR (black and white squares) to interfere with the boundary's lookup & more operation that we will perform next.

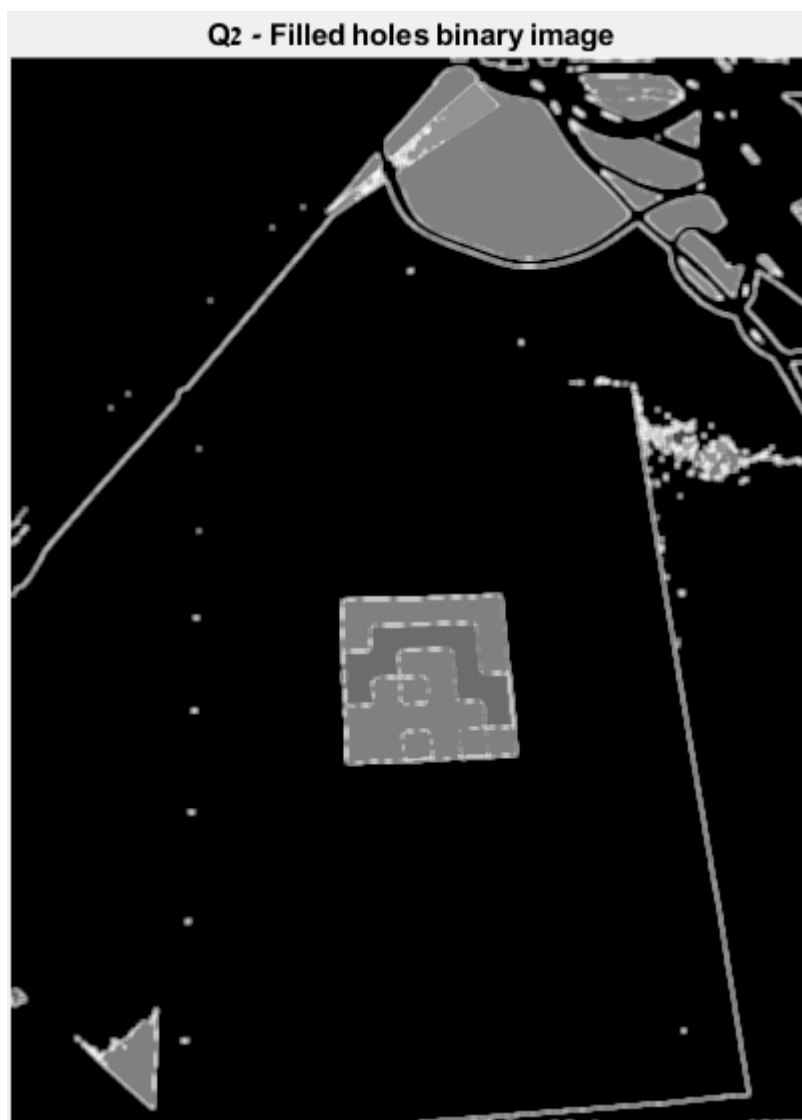


Figure 2.1.1.5

Step 2 - Find the boundaries of all objects in the binary image

1. Boundaries - get outlines of each object in the image

By using `B = bwboundaries(BW,'noholes')` traces the exterior boundaries of objects, where 'noholes' is specifying that we don't want to include the boundaries of holes inside other objects (we are looking for the frame of the QR object).

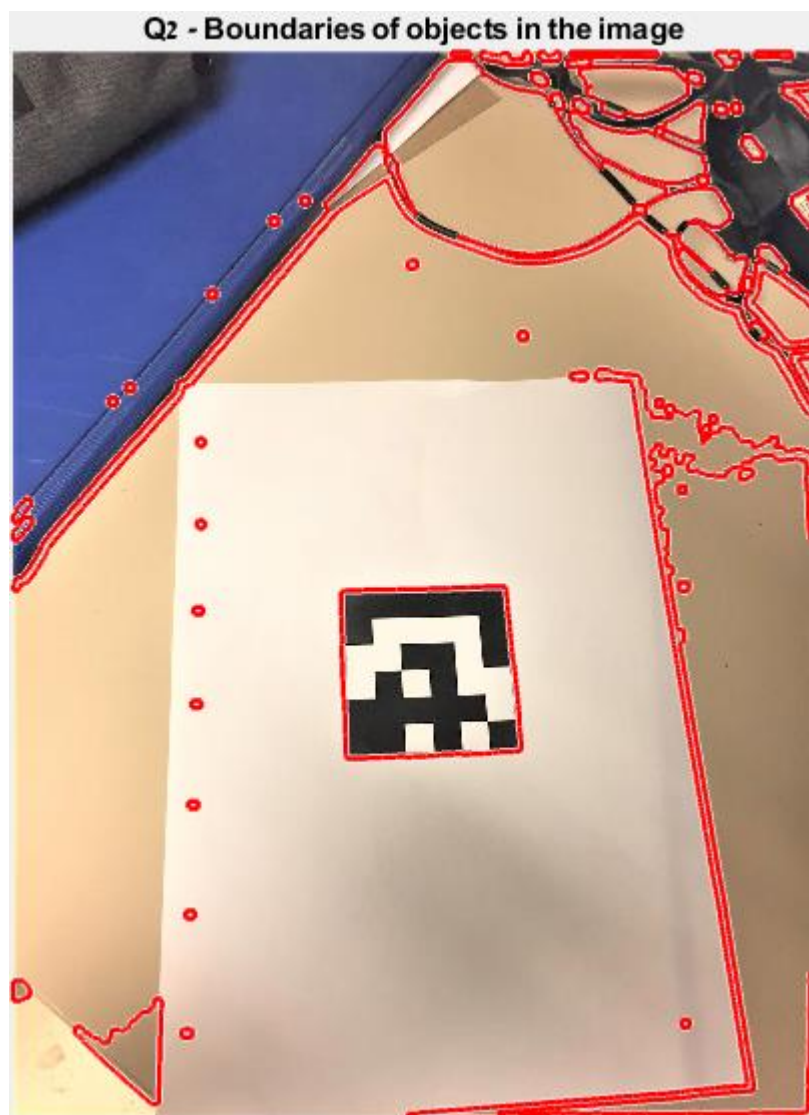


Figure 2.1.2.1



## 2. Classify each object boundaries shape

In this step we used the function *minboundparallelogram* (attached) written by :

<https://www.mathworks.com/matlabcentral/fileexchange/34767-a-suite-of-minimal-bounding-objects?focused=3820654&tab=function>

Also, to classify each object we used the following implementation :

<https://www.mathworks.com/matlabcentral/answers/116793-how-to-classify-shapes-of-this-image-as-square-rectangle-triangle-and-circle>

This implementation is using the function *stats = regionprops(BW, 'Centroid', 'Area', 'Perimeter')* that returns measurements for the set of properties specified by properties for each 8-connected component (object) in the binary image, BW. Stats are structing array containing a struct for each object in the image.

Then for each boundary we are using *minboundparallelogram(boundary)* to Compute the minimal bounding parallelogram of points in the plane.

All of this is helping us classify each object shape by its boundary's stats, which then enable us to find the most parallelogram (in the code represent as square or rectangle) object in the image (the QR).

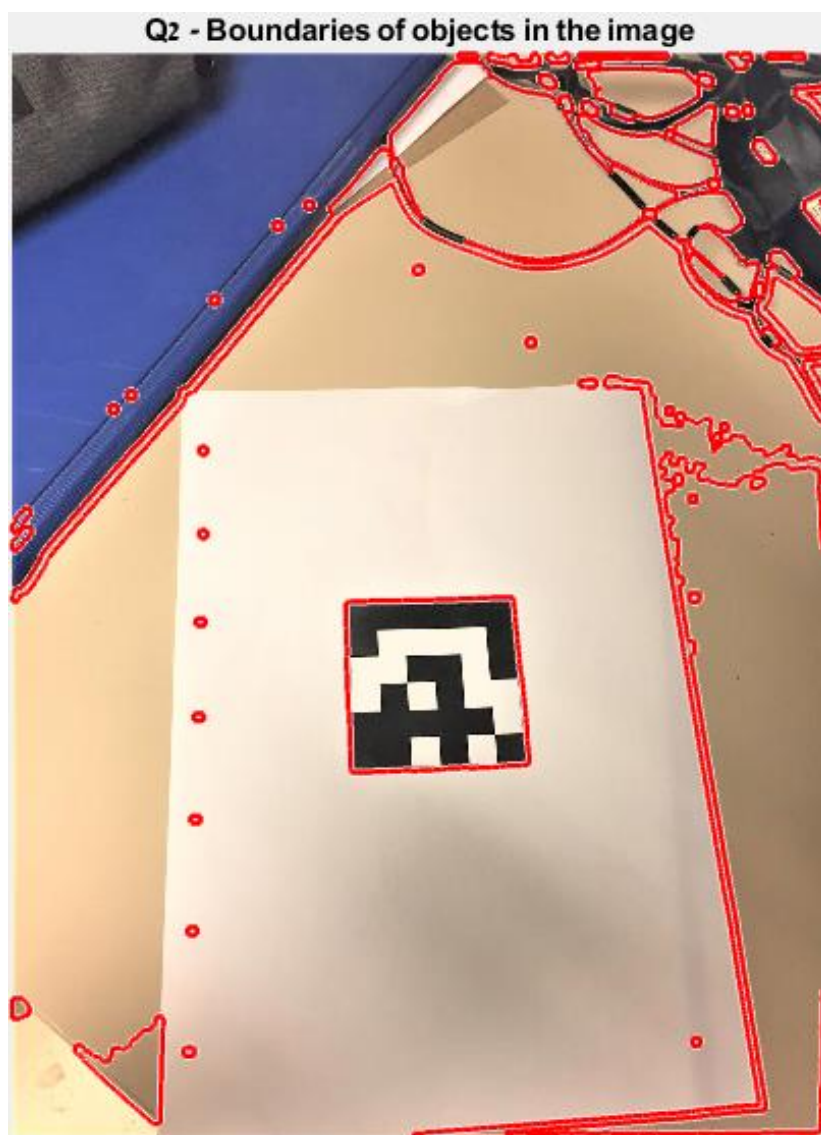


Figure 2.1.2.2

### Step 3 - Find the parallelogram (the QR) boundary

We found the max square/rectangle/parallelogram index boundary by the logic  $\max(\text{Area} * (\text{isSquare} + \text{isRectangle}))$  and then extracted this QR boundary cell by  $B\{\text{index\_max}\}$ .

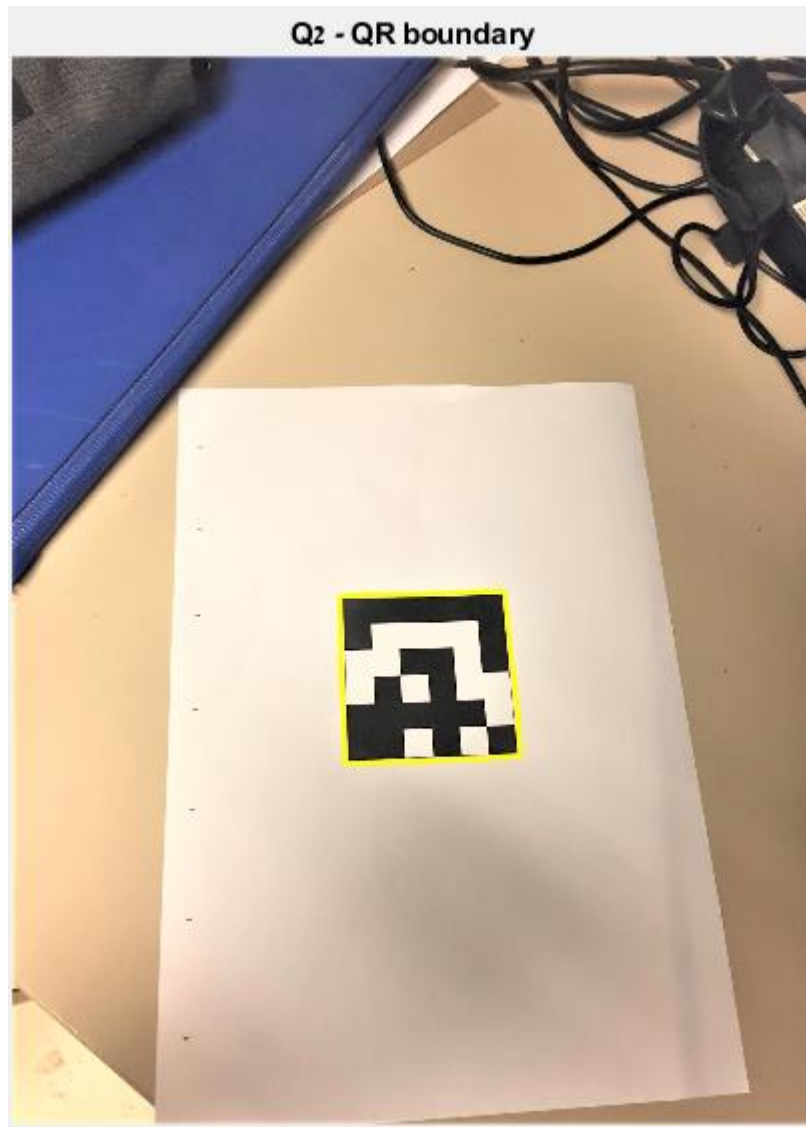


Figure 2.1.3.1

#### Step 4 - Create a binary mask of the QR boundary

After finding the QR boundary, we created a binary mask by using *poly2mask(xi,yi,m,n)* that computes a binary region of interest mask of size m-by-n, from an ROI polygon with vertices at coordinates xi and yi. *poly2mask* closes the polygon automatically, if the polygon is not already closed. The *poly2mask* function sets pixels that are inside the polygon to 1 and sets pixels outside the polygon to 0.

As we can see, currently we been able to take any image of the QR code and extract the exact binary mask of its shape.

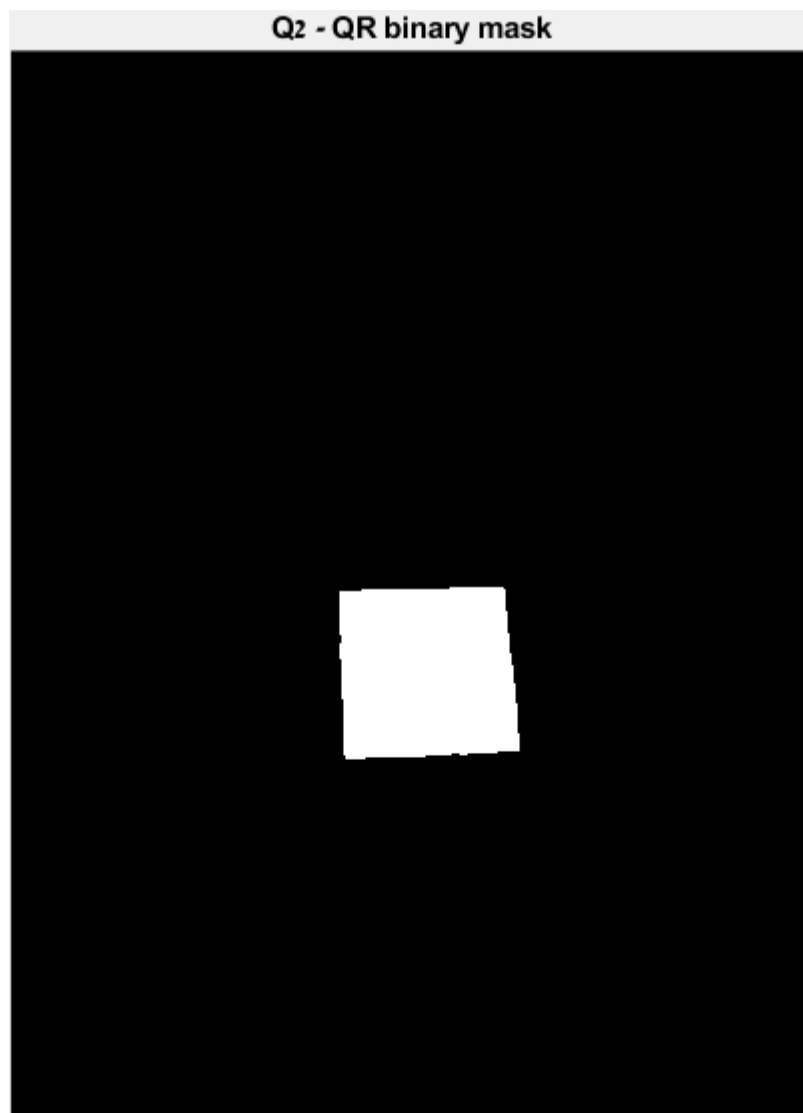


Figure 2.1.4.1

## Step 5 - Find Hough lines of the QR

### 1. Canny's edge detector

By using `edge(binarymaske)` that returns a binary image containing 1s where the function finds edges in the input image I and 0s elsewhere using Canny method. The Canny method is less likely than the other methods to be fooled by noise, and more likely to detect true weak edges.

### 2. Hough lines

`[H,theta,rho] = hough(canny_binarymask)` computes the Standard Hough Transform (SHT) of the binary image BW. The Hough function is designed to detect lines.

### 3. Find 4 peaks of the Hough matrix

`peaks = houghpeaks(H,numpeaks)` locates peaks in the Hough transform matrix, H, generated by the Hough function. numpeaks specifies the maximum number of peaks to identify. The function returns a matrix that holds the row and column coordinates of the peaks.

After playing with the threshold we choose the minimum value to be considered a peak to be `ceil(0.1 * max(H(:)))`.

### 4. Normalize returning values & compute the lines equations matrix

By using the description of the Hough function, we can see that the function uses the parametric representation of a line:  $\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$ . Then we used `p = polyfit(x,y,n)` that returns the coefficients for a polynomial  $p(x)$  of degree n that is a best fit and insert it to the `lines_matrix` as a line equation.

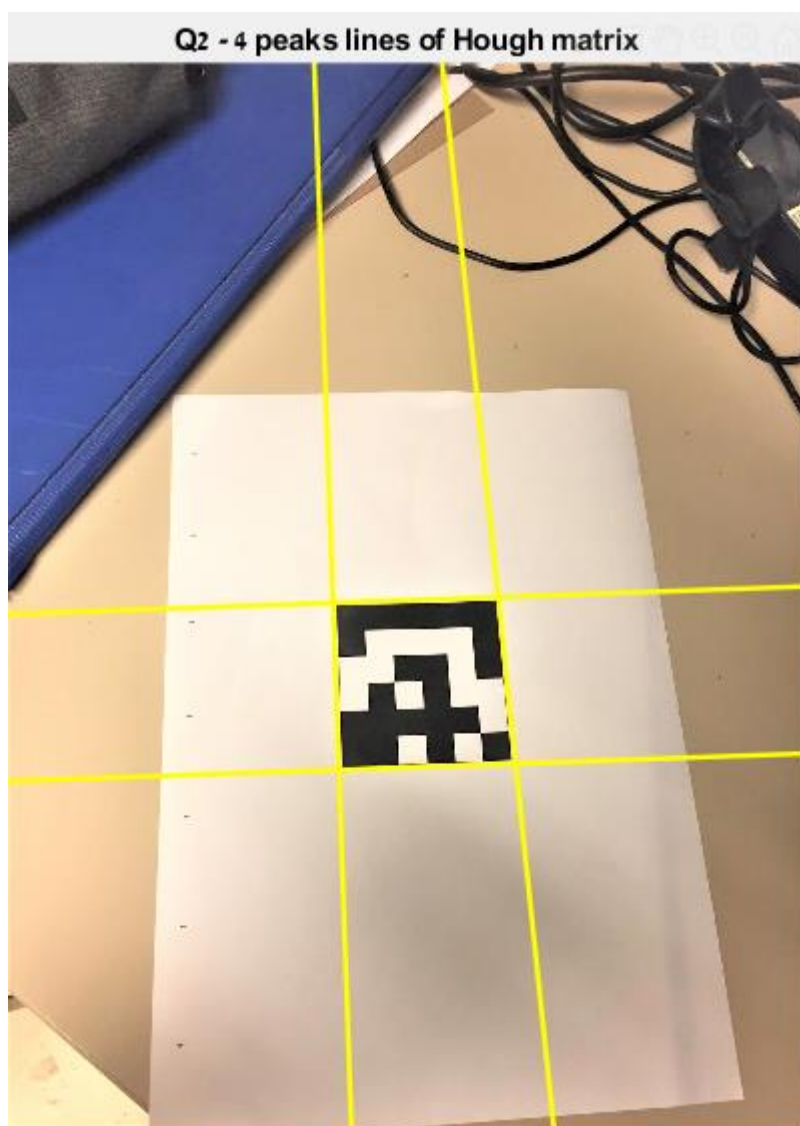


Figure 2.1.5.1

## Step 6 - Find intersections of those lines - the corners

### 1. Find the potential intersection points of the 4 lines

In general, the number of intersections of  $n$  lines would be

$$\#intersection = \frac{n(n-1)}{2} \rightarrow \frac{4(3)}{2} = 6$$

If we have 2 lines :

$$y = a * x + b$$

$$y = c * x + d$$

The intersection point will be :

$$a * x + b = c * x + d$$

$$x_{intersection} = \frac{d-b}{a-c}$$

$$y_{intersection} = a * \frac{d-b}{a-c} + b$$

We calculated the 6 option of potential intersection points (e.g. line 1 with line 2, line 1 with line 3 etc.)

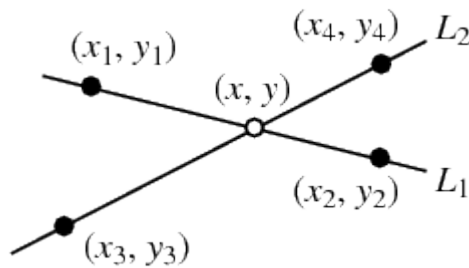


Figure 2.1.6.1

## 2. Remove intersection points that out of the image

Eventually we are looking for 4 intersection points (that will be in the image pixel range). We filter the intersection points that out of range and displayed the 4 points that left.

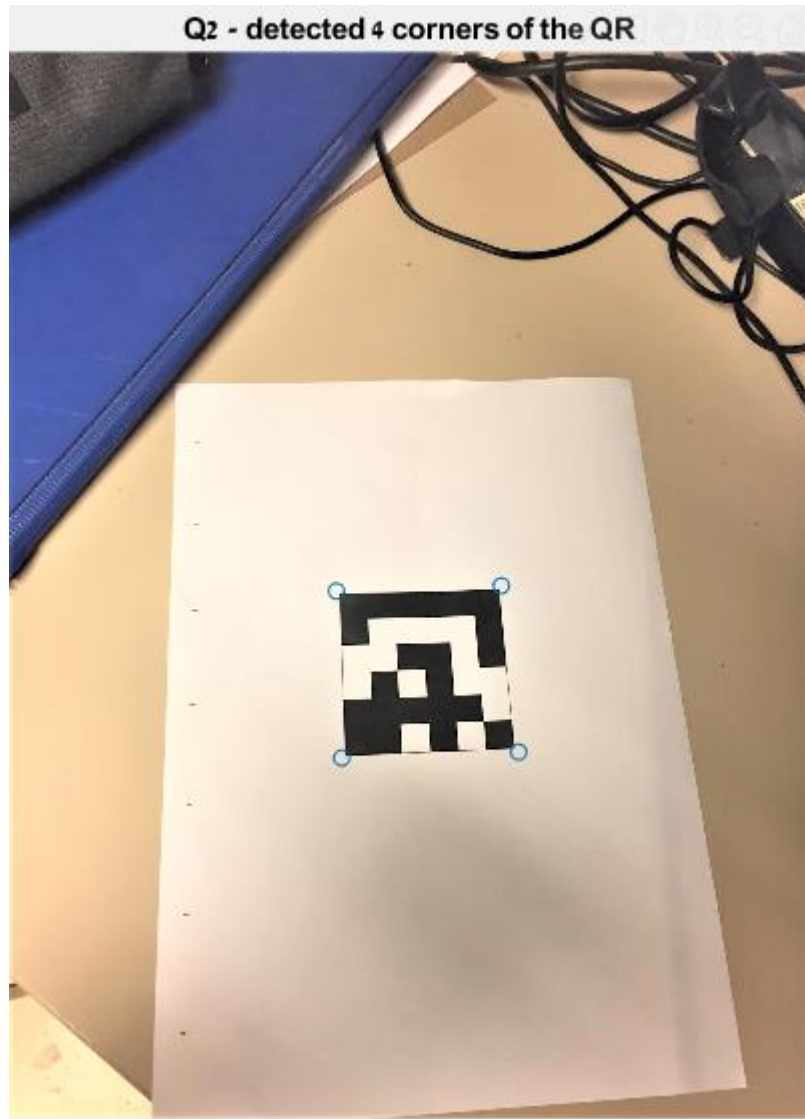
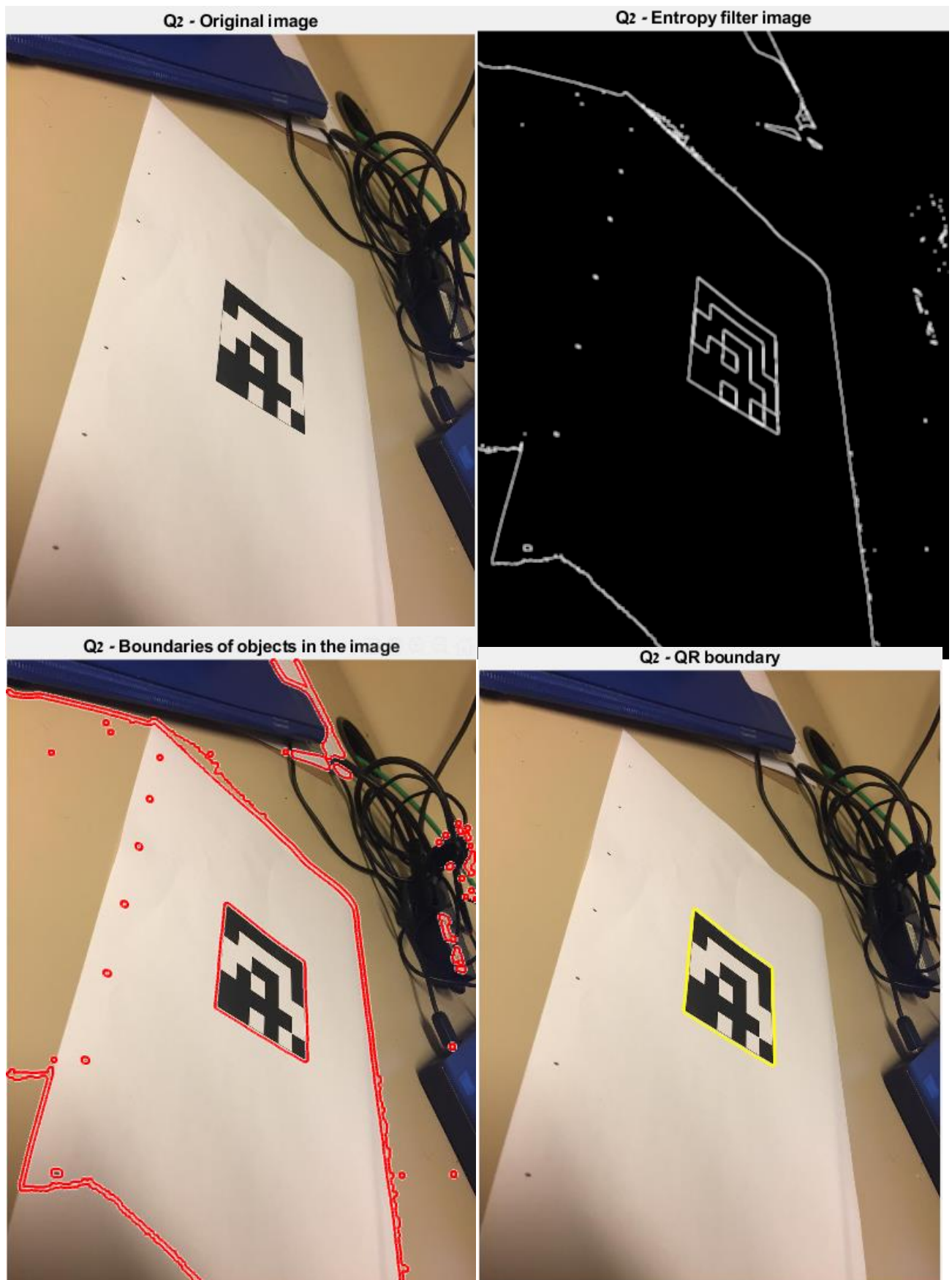


Figure 2.1.6.2

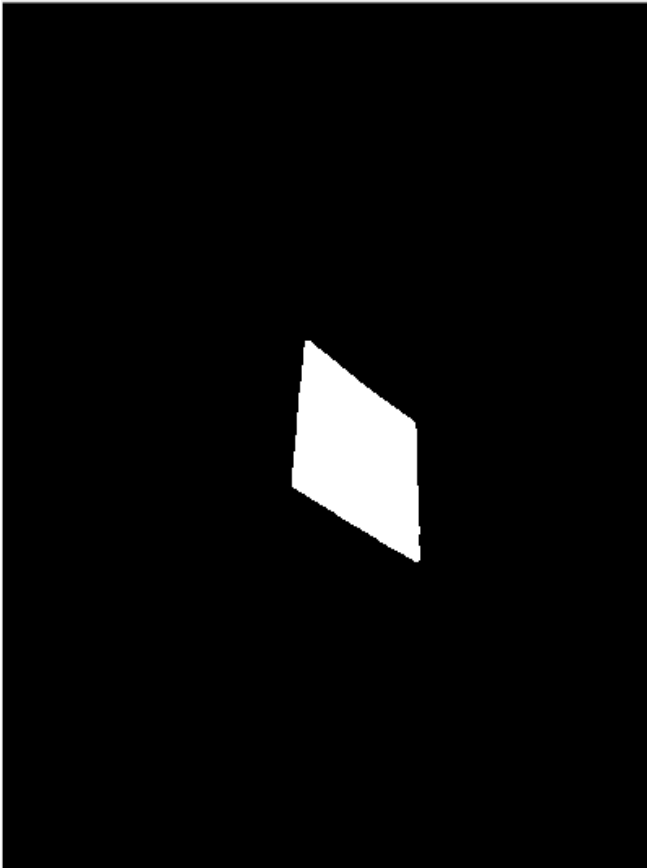


## Additional results of the corner detection algorithm

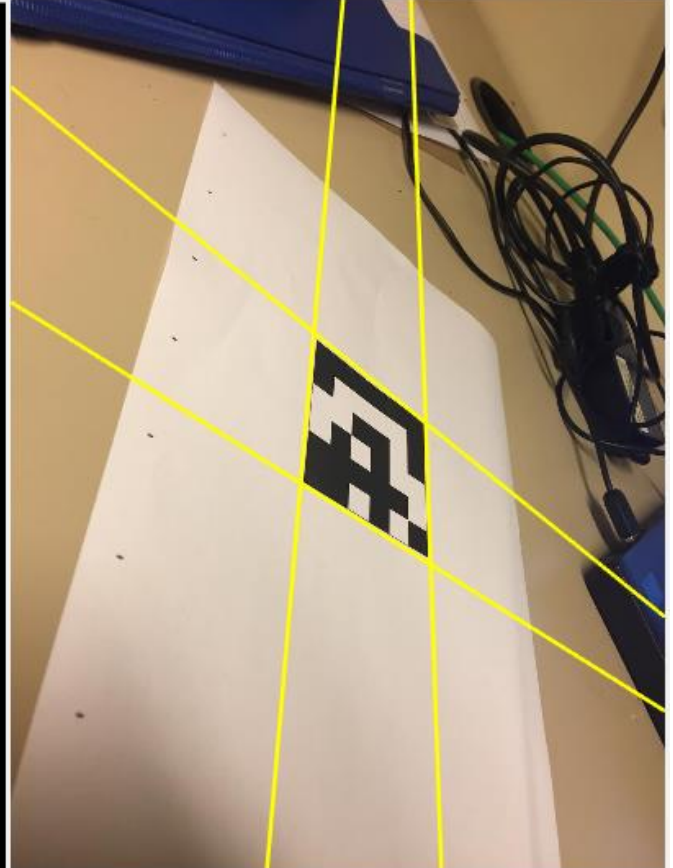
Image 2



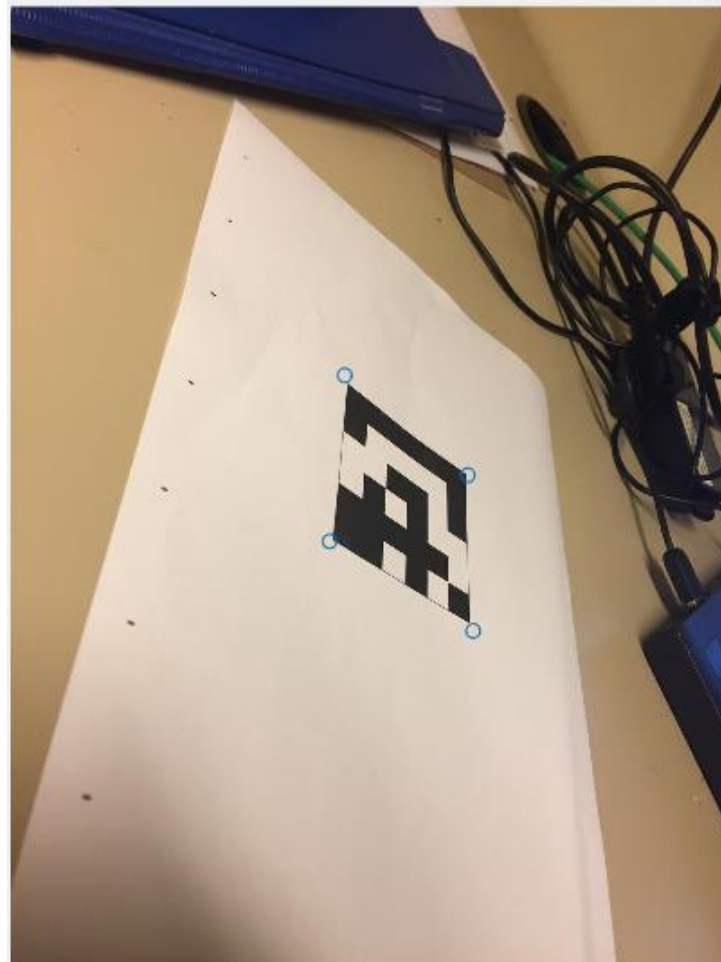
Q2 - QR binary mask



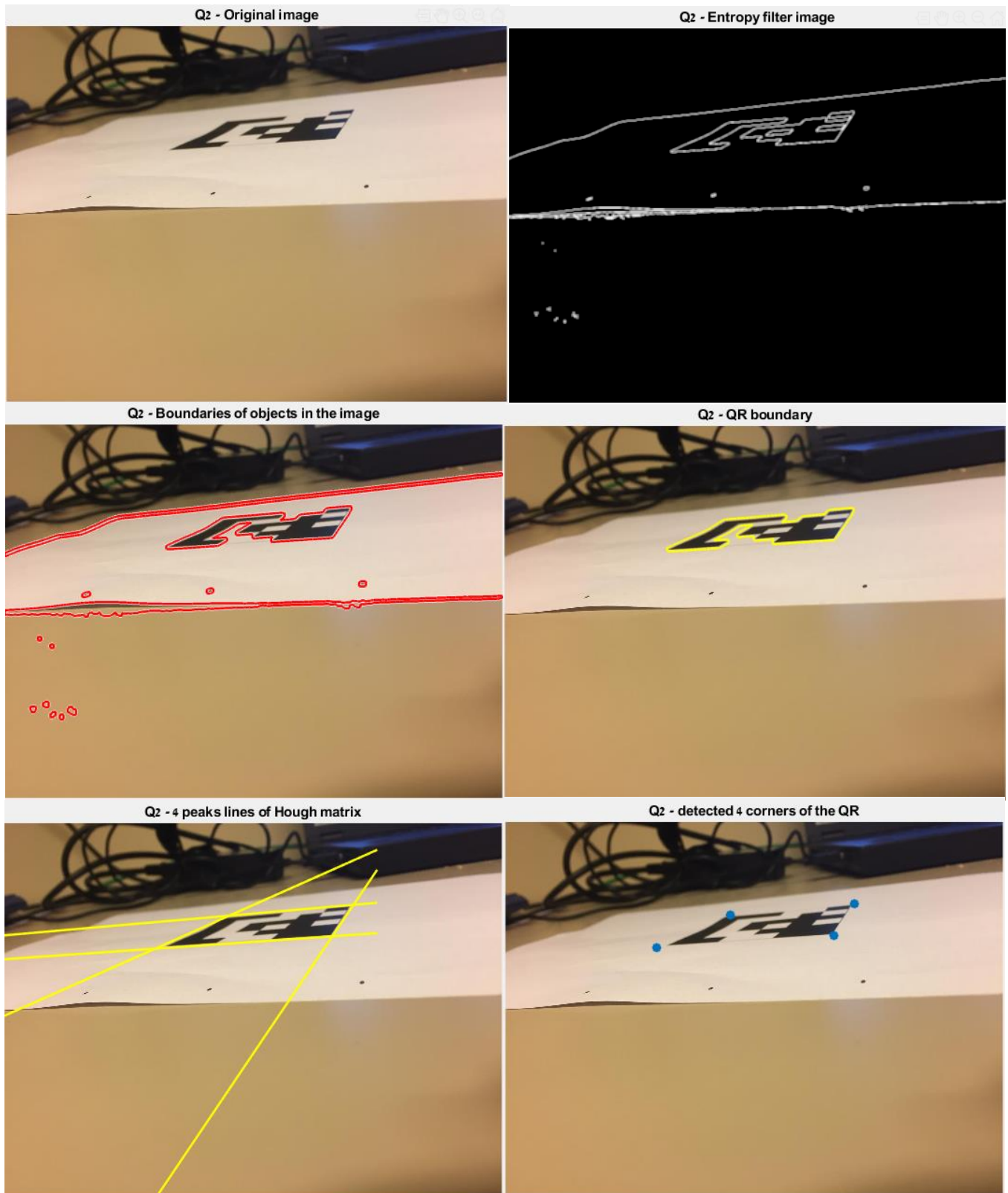
Q2 - 4 peaks lines of Hough matrix



Q2 - detected 4 corners of the QR



## Image 3



## 2.2 Results – feeding the Q1 algorithm

Q2.m file

**Note :** because we don't know what order the Hough matrix will give us the lines, we will get each image a different order of the corners returning from the corner detection algorithm. In the regular QR we have an indication as describe in page 18.

### Image 1

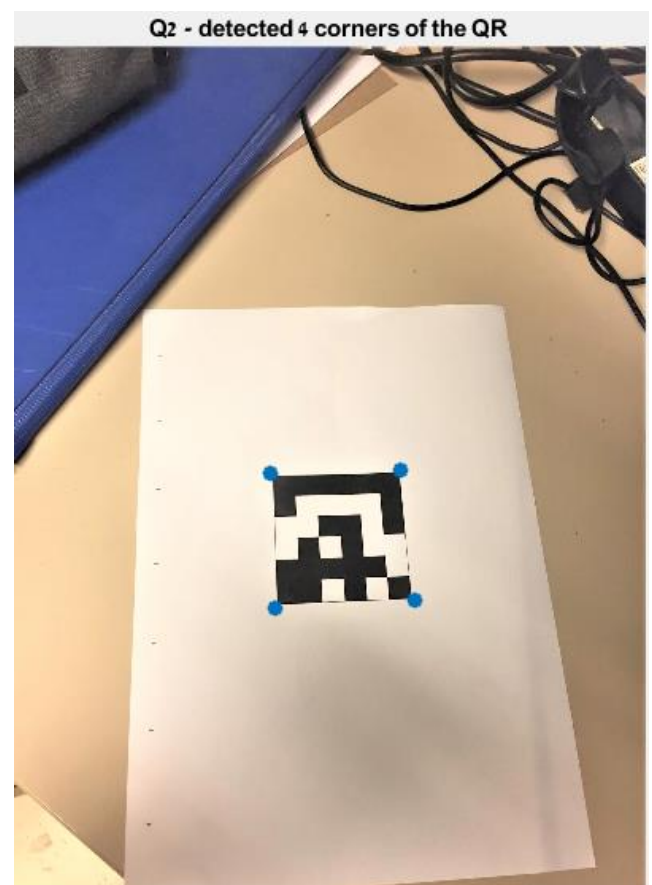
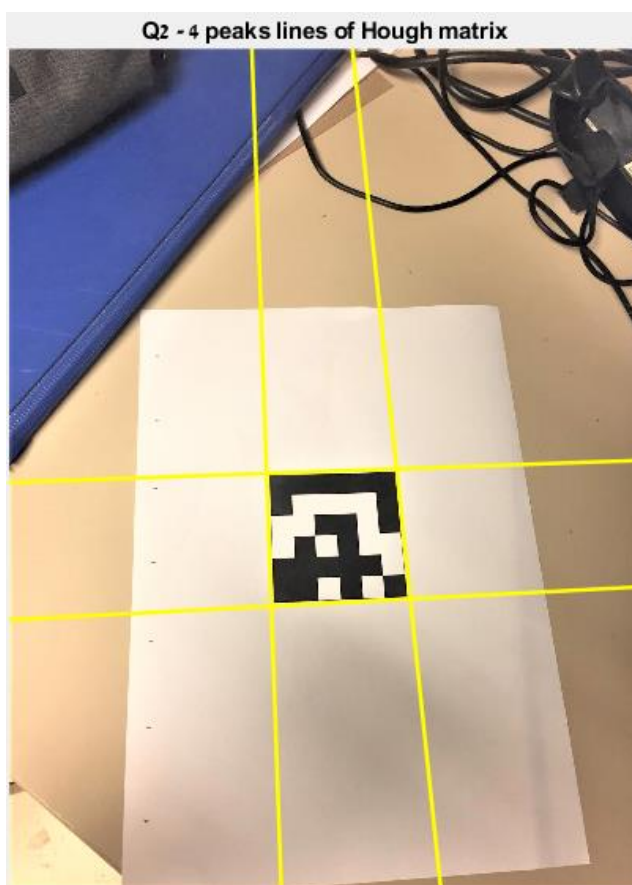
First we will get the corners from our *dip\_detetQR\_corners(image\_org)*, then we will reorder the corners in the vector in a clockwise order from the upper left corner. In the specific image is the following :

```
corners(1,:) = points(1,);
```

```
corners(2,:) = points(2,);
```

```
corners(3,:) = points(4,);
```

```
corners(4,:) = points(3,);
```



Figures 2.2.1 - 2

Then we will feed the function from Q1 `dip_imageQR2ID(image_org, image_corners, qr_angle1, corners)`.

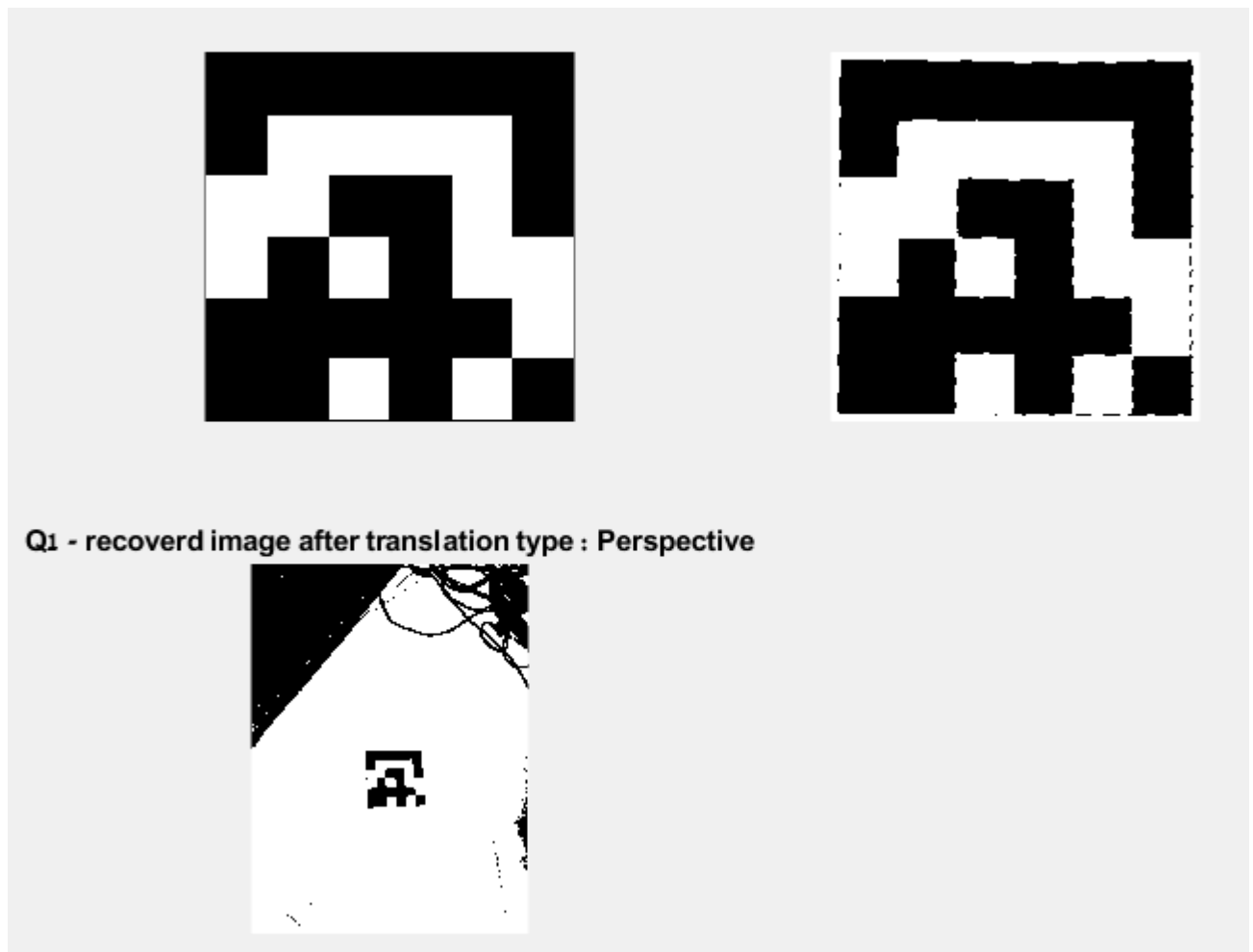


Figure 2.2.3

And we got a correct result :



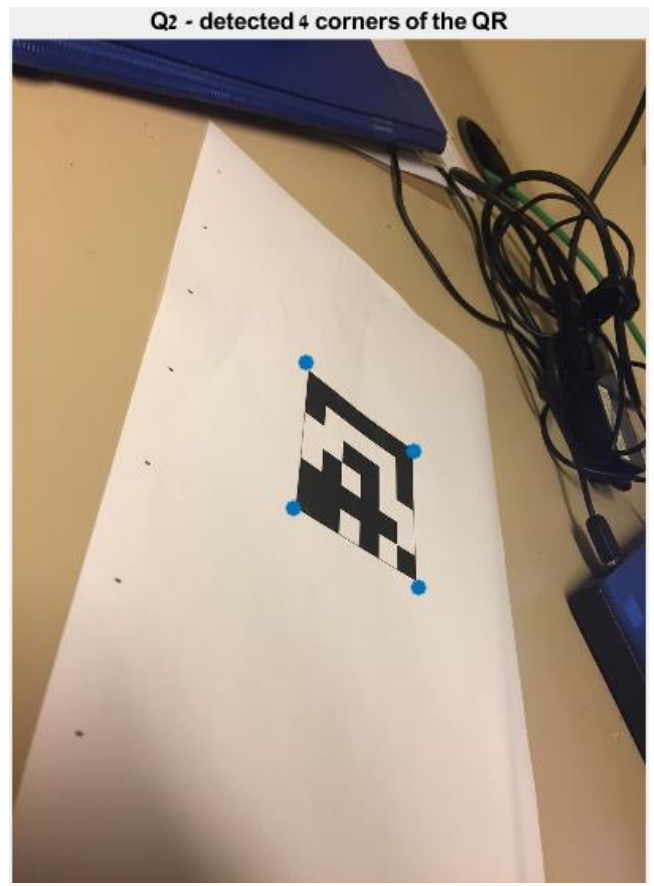
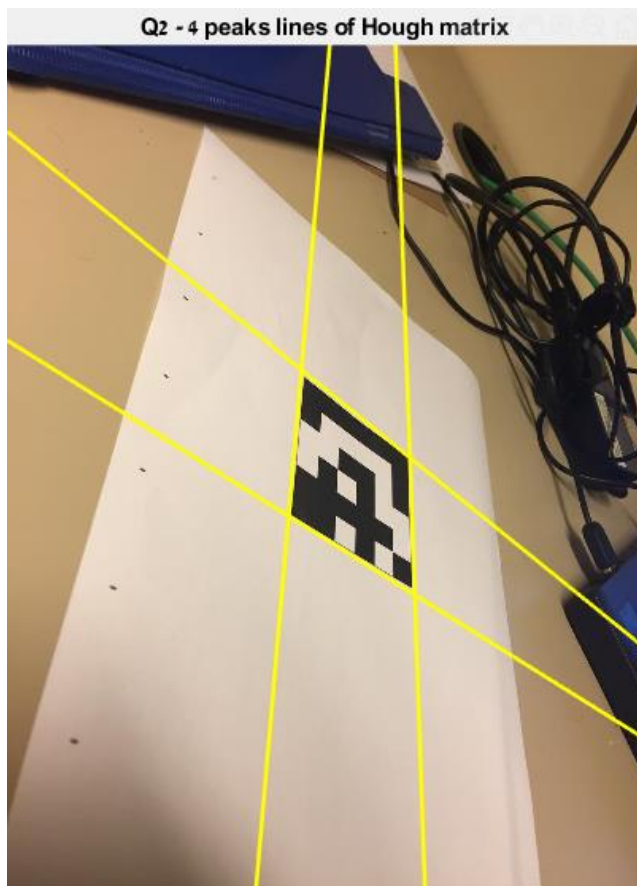
Figure 2.2.4



## Image 2

First we will get the corners from our `dip_detetQR_corners(image_org)`, then we will reorder the corners in the vector in a clockwise order from the upper left corner. In the specific image is the following :

```
corners(1,:) = points(3,);  
corners(2,:) = points(1,);  
corners(3,:) = points(2,);  
corners(4,:) = points(4,);
```



Figures 2.2.4-5

Then we will feed the function from Q1 `dip_imageQR2ID(image_org, image_corners, qr_angle1, corners)`.

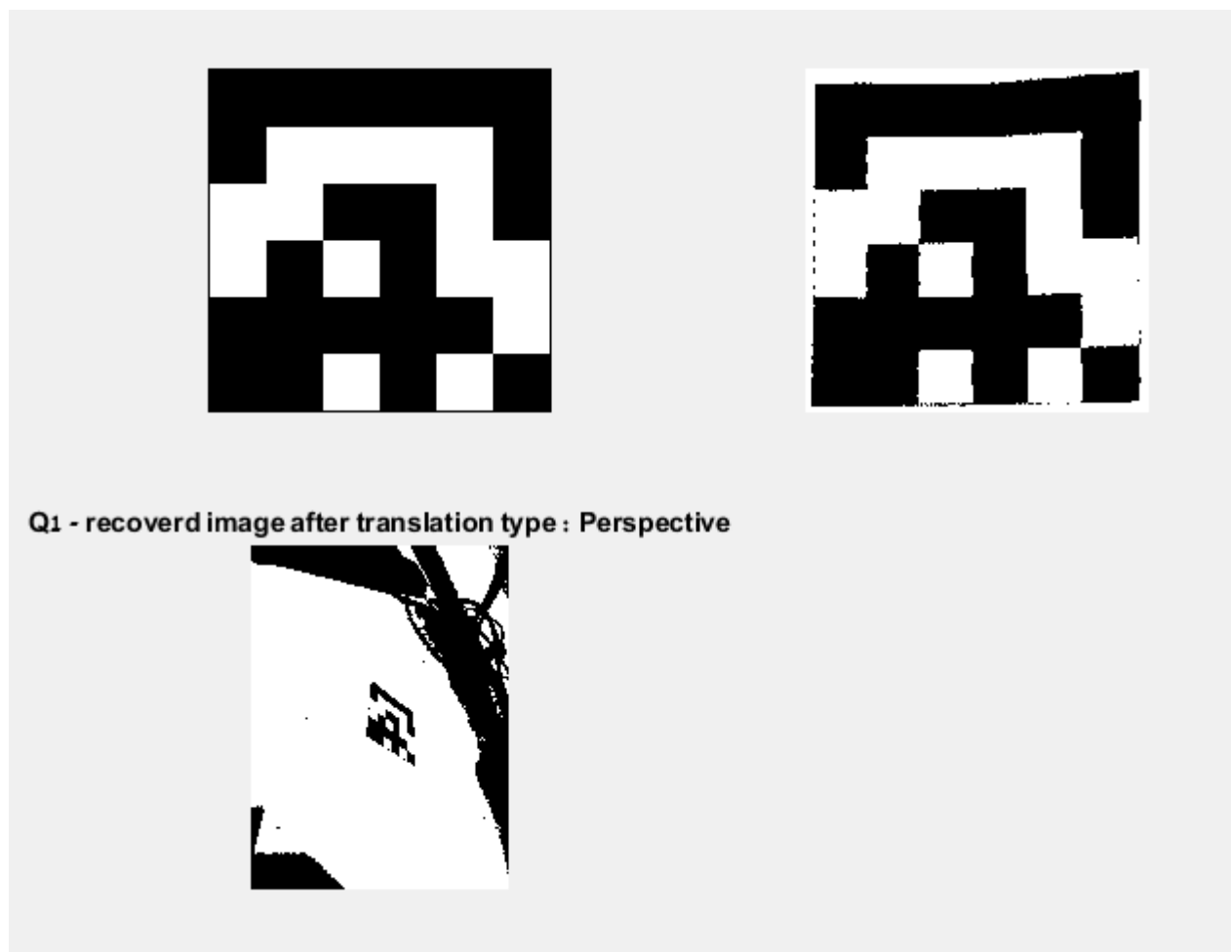


Figure 2.2.6

And we got a correct result :


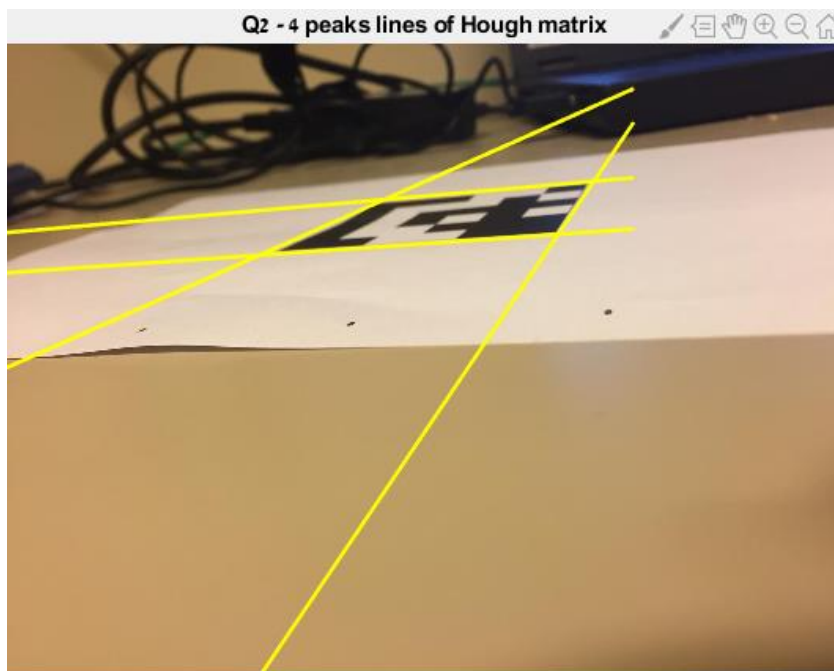
 id	'318550746'
 id_result_1	'318550746'
 id_result_2	'318550746'

Figure 2.2.7

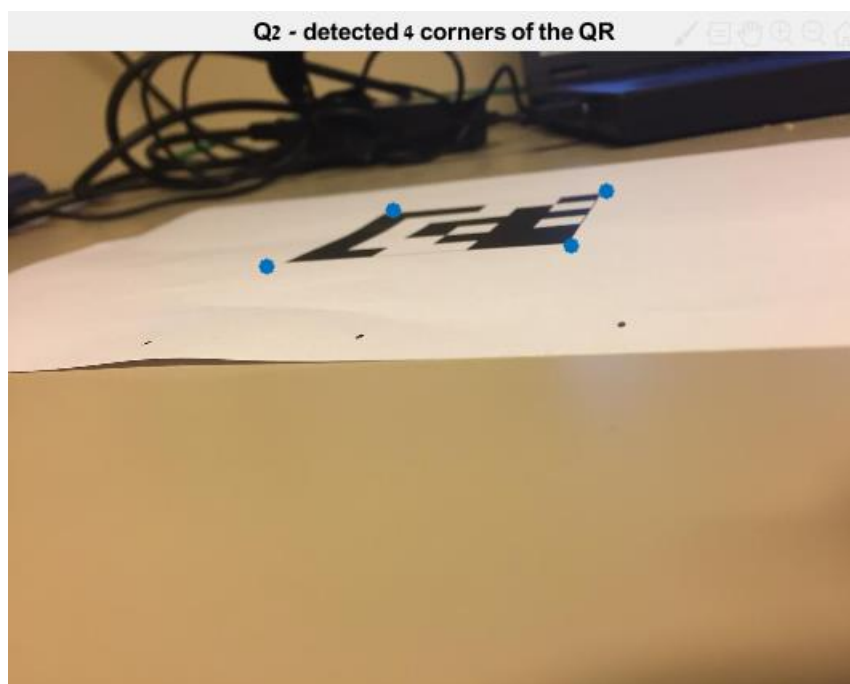
### Image 3

First we will get the corners from our `dip_detetQR_corners(image_org)`, then we will reorder the corners in the vector in a clockwise order from the upper left corner. In the specific image is the following :

```
corners(1,:) = points(1,:);  
corners(2,:) = points(3,:);  
corners(3,:) = points(4,:);  
corners(4,:) = points(2,:);
```



Figures 2.2.8





Figures 2.2.9

Then we will feed the function from Q1 `dip_imageQR2ID(image_org, image_corners, qr_angle1, corners)`.

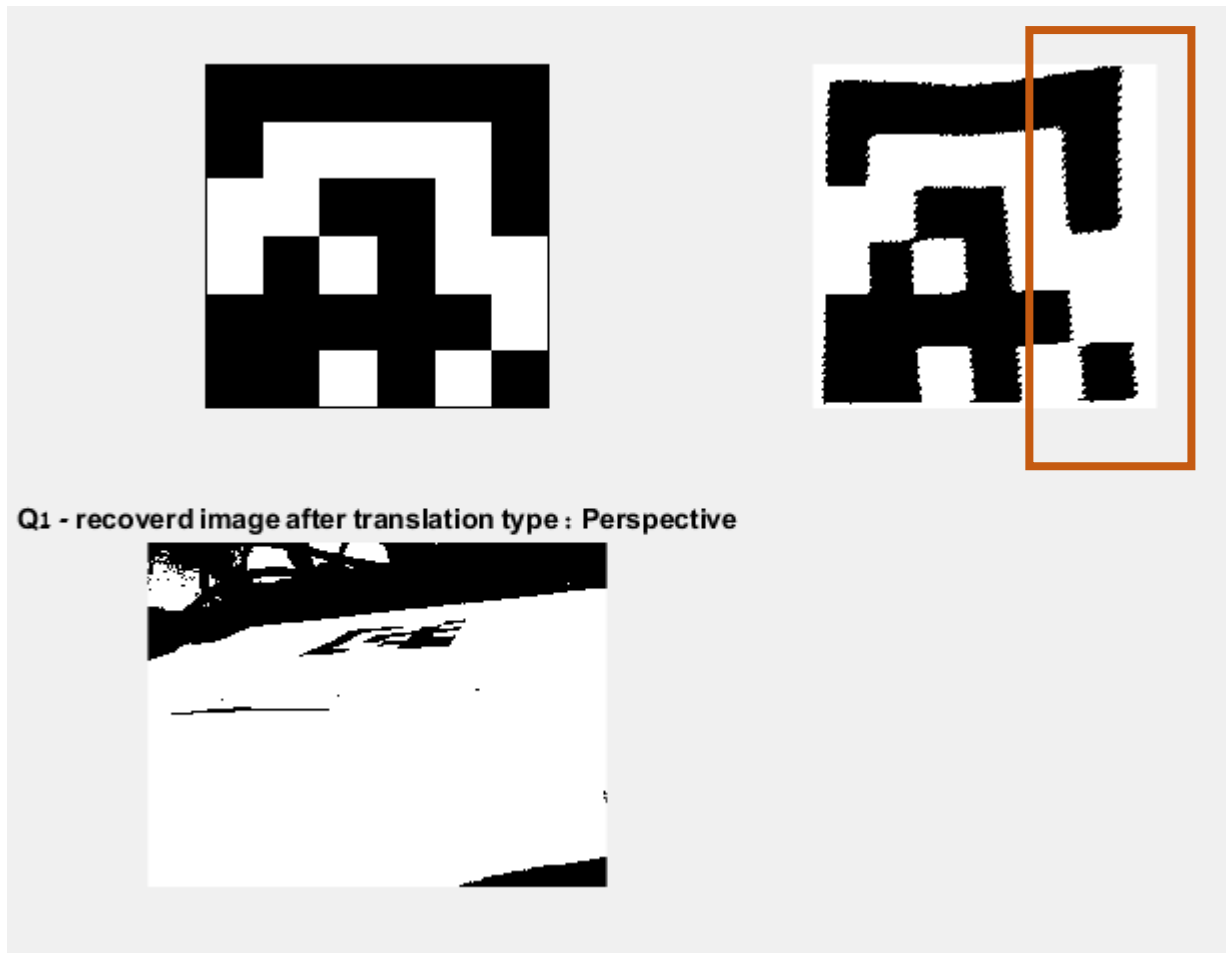


Figure 2.2.10

And we got an incorrect result :

id	'318550746'
id_result_1	'318550746'
id_result_2	'318550746'
id_result_3	'318551211514'

Figure 2.2.11

The reason is that the first corner is a bit off place (the intersection is a little far from the correct corner), which making the stretch QR have bit squares that evaluate as white instead of black (the stretch effect more than the mean of the area in the **right side of the QR**), hence the wrong conversion to the ID.