Group Number: 18

Subject Code: MANU2453

Assignment 3 Title: Robot Vision Report

Due Date: October 11th, 2020

Group 18 Member Details:

- Name : Aditya Prawira
  RMIT ID : S3859061
  Email : s3859061@student.rmit.edu.au
- Name : Matthew Andrew
  RMIT ID : S3859586
  Email : s3859586@student.rmit.edu.au

Table of Content

1. Problem 1

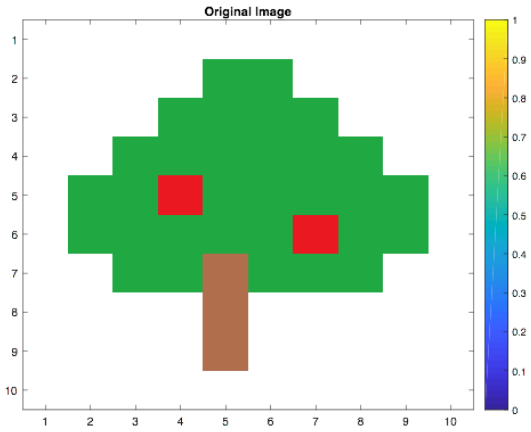


Figure 1. Original AppleTree PNG image 2020

How do you find out the position / coordinates of the red apples?

The input image has the size of 10×10 pixel, where a color fills a pixel of the image. Since the input is a RGB coloured image, therefore it is expected that there will be three 10×10 matrixes for Red (I_{Red}), Green (I_{Green}) and Blue Scale (I_{Blue}). By observation, it is obvious that the apple or red pixel has the lowest intensity/count in this image. Therefore, the key to find the apple's location is to identify the value of color with the lowest intensity.

However, it is sufficient to compute the task by analyzing a single 10×10 matrix. Provided, that the formula to convert the image into grayscale image is

$$I_{Grey} = \frac{I_{Red} + I_{Green} + I_{Blue}}{3}$$

Hence, a 10 x 10 matrix that contains brightness values of the grayscale image is produced. Below are images for I_{Red}, I_{Green}, I_{Blue}, and I_{Grey}, where on the right hand side is the histogram with the x-axis that represents brightness value from the grayscale image, while the y-axis represents intensity/count of that color.

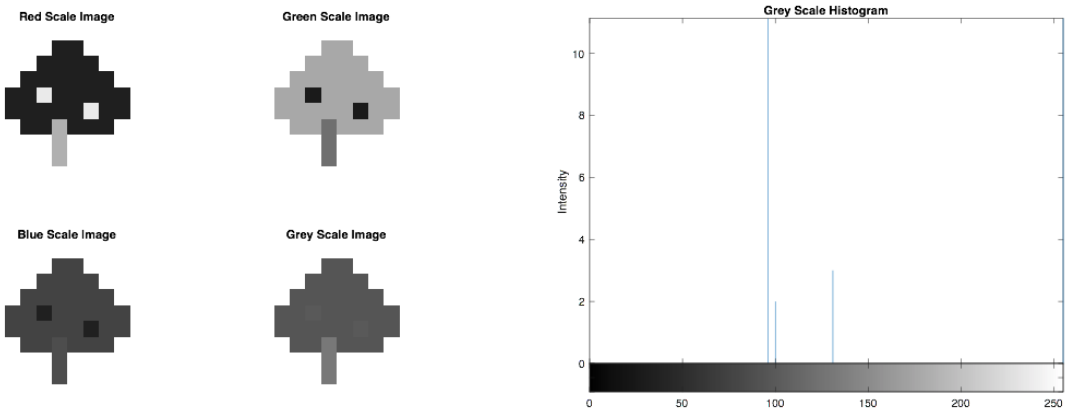
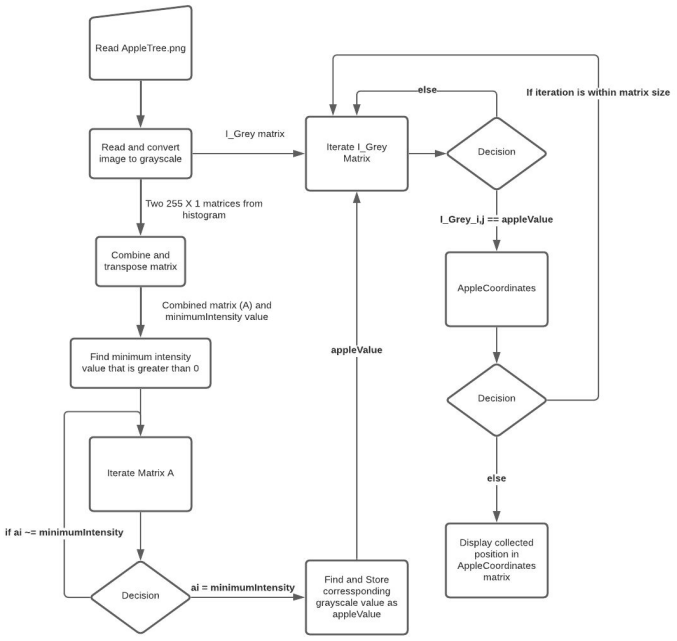


Figure 2. AppleTree in Red, Green, Blue and Grey scale (LHS), & AppleTree Grayscale histogram (RHS) 2020

1.1 Details of the algorithm (Theory / Concept):

Based on the histogram in **Figure 2**, it is identified that the value of 100 in the grayscale image has the lowest intensity of 2. Where 2 is the expected count of apples in the AppleTree image. By utilizing imhist function, other than producing a histogram it is also generating two 255×1 column matrices for the brightness value and its corresponding intensity value. Finally, combine



the two matrices together into 255×2 matrix and transpose it to 2×255 matrix.

Figure 3. Problem 1 algorithm flowchart 2020

Therefore, the algorithm that will be applied is to find the minimum intensity value that is greater than 0. Then, evaluate its corresponding grayscale value and store it. Finally, find values in the 10×10 I_{Grey} matrix that is equal to the value that was stored previously, and stores its positions in a matrix. The workflow of the algorithm will be presented in **Figure 3** above.

1.2 Explanation of the MATLAB codes you wrote:

```
1   % Read Image
2   fileName = 'AppleTree.png';
3   I = imread(fileName);
4   fprintf('Testing Image %s\n', fileName);
5   figure, image(I), colorbar;
6
7   title('Original Image');
8
9   IRed = I(:,:,1);
10  IGreen = I(:,:,2);
11  IBlue = I(:,:,3);
12
13  IGrey = (double(IRed)+double(IGreen)+double(IBlue))/3;
14
15  I = uint8(IGrey);
16
17  %Store grey intensity values coresspond to
18  %its number of count or intensity
19  [y,x] = imhist(I);
20  figure, imhist(I);
21  title('Grey Scale Histogram')
22  ylabel('Intensity');
23
24  A = transpose([y x]);
25  B = A(1,:);
26
27  %Based on the image, an apple fits 1x1 pixel.
28  %Expect 2 apples, or 2 red values.
29  %Find minum count or intensity that is greater than 0
30  minimumIntensity = min(B(B>0));
31
32  appleValue = 0;
33
34  for i = 1:length(A)
35      if A(1,i) == minimumIntensity
36          appleValue = A(2, i);
37      end
38  end
```

```
39
40  [m,n] = size(I);
41  index = 1;
42
43  %Number of x and y coordinates found is equal to
44  %minimumIntensity or the minimum color count
45  X = zeros(0, minimumIntensity);
46  Y = zeros(0, minimumIntensity);
47
48  for i = 1:m
49      for j = 1:n
50          if I(i,j) == appleValue
51              X(index) = j;
52              Y(index) = i;
53              fprintf('Apple Coordinates: (x%d, y%d) = (%d, %d)\n',
54                  index, index, j,i);
55              index = index + 1;
56          end
57      end
58  end
59
60  %Store all coordinates to AppleCoordinates
61  AppleCoordinates = transpose([X; Y]);
62  fprintf('\nApple Coordinates Matrix:\n');
63  disp(AppleCoordinates);
64
65  figure;
66  subplot(2,2,1), imshow(IRed, 'InitialMagnification', 1600);
67  title('Red Scale Image');
68  subplot(2,2,2), imshow(IGreen, 'InitialMagnification', 1600);
69  title('Green Scale Image');
70  subplot(2,2,3), imshow(IBlue, 'InitialMagnification', 1600);
71  title('Blue Scale Image');
72  subplot(2,2,4), imshow(I, 'InitialMagnification', 1600);
73  title('Grey Scale Image');
74
75
```

Figure 4. Problem 1 MATLAB code 2020

In **Figure 4** above,

Line 19, 24, & 25:

The x variable stores all grayscale brightness values ($0 \rightarrow 255$), where the y variable stores all its corresponding intensity level (or count). Hence, as x and y have the same dimension, both variables are then combined and transposed, as shown in line 24, to become a 255×2 matrix (variable A). Finally in line 25, variable B stores the first row of matrix A, where all elements are intensity values.

Line 30:

This line of code will store a value which is the most minimum intensity level that is non-zero (minimumIntensity variable).

Line 34 → 38:

These lines will iterate each element contained in the first row of matrix A. When an element in the first row is equal to minimumIntensity value, then the appleValue variable will store its corresponding grayscale value that is located in the same column and a row lower.

Line 45 & 46:

Initiate 1D row matrix X and Y with the size that is equal to minimumIntensity value (minimum count).

Line 48 - 58:

These lines will iterate I_{Grey} matrix and finds all elements that are equal to appleValue. Then, X and Y will store (x, y) positions of those values.

Line 61:

This line will store and combine all position coordinates of apples $(x_{1,m}, y_{2,m})$, where m is the number of apples spotted by the algorithm.

1.3 The results:

Hence, the result represented in **Figure 5** below.

```
>> Problem1
Testing Image AppleTree.png
Apple Coordinates: (x1, y1) = (4, 5)
Apple Coordinates: (x2, y2) = (7, 6)

Apple Coordinates Matrix:
     4     5
     7     6
```

Figur 2020

Below is the value c e are 2 apples in the image. By observing values of 100 where those values are app t the 4th column and 5th row, where the ow. Therefore, it is obvious that the val l by the I_{Grey} matrix

below.

255	255	255	255	255	255	255	255	255	255
255	255	255	255	96	96	255	255	255	255
255	255	255	96	96	96	96	255	255	255
255	255	96	96	96	96	96	96	255	255
255	96	96	100	96	96	96	96	96	255
255	96	96	96	96	96	100	96	96	255
255	255	96	96	131	96	96	96	255	255
255	255	255	255	131	255	255	255	255	255
255	255	255	255	131	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255

To check and build the confidence for the algorithm's reliability and robustness, 2 case studies are generated by testing 2 images modified from the original AppleTree image. Where the differences are the brightness level. Below are the computed alternative case studies.

1.3.1 Alternative Case Study 1:

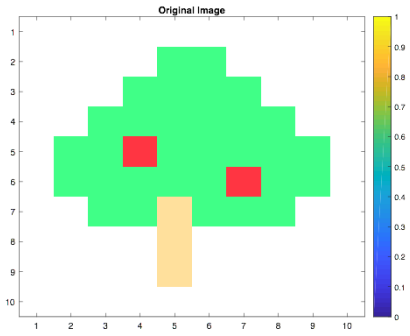


Figure 6. AppleTree2.png images (brighter than original image) 2020

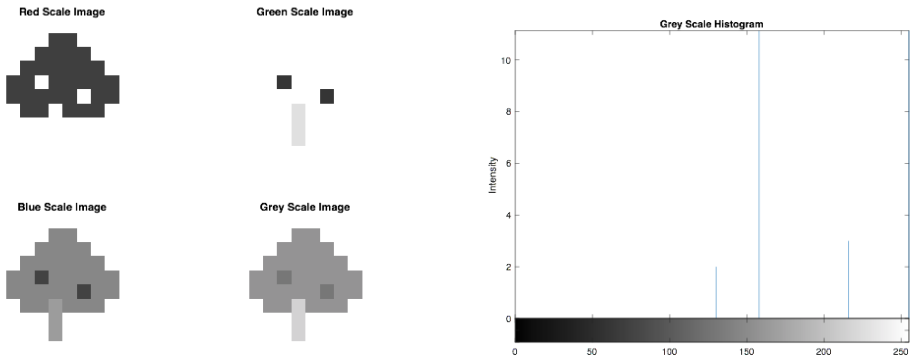


Figure 7. AppleTree2 in Red, Green, Blue and Grey scale (LHS), & AppleTree Grayscale histogram (RHS) 2020

```
>> Problem1
Testing Image AppleTree2.png
Apple Coordinates: (x1, y1) = (4, 5)
Apple Coordinates: (x2, y2) = (7, 6)

Apple Coordinates Matrix:
     4     5
     7     6
```

Figure 8. Result of all apples coordinates in AppleTree2 image 2020

1.3.2 Alternative Case Study 2:

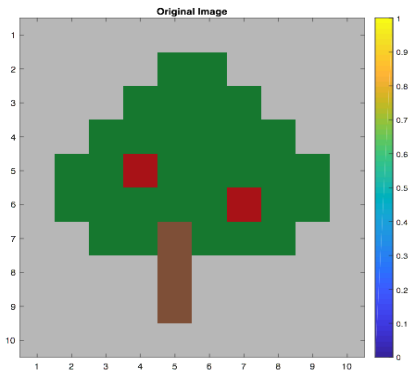


Figure 9. AppleTree3.png images (darker than original image) 2020

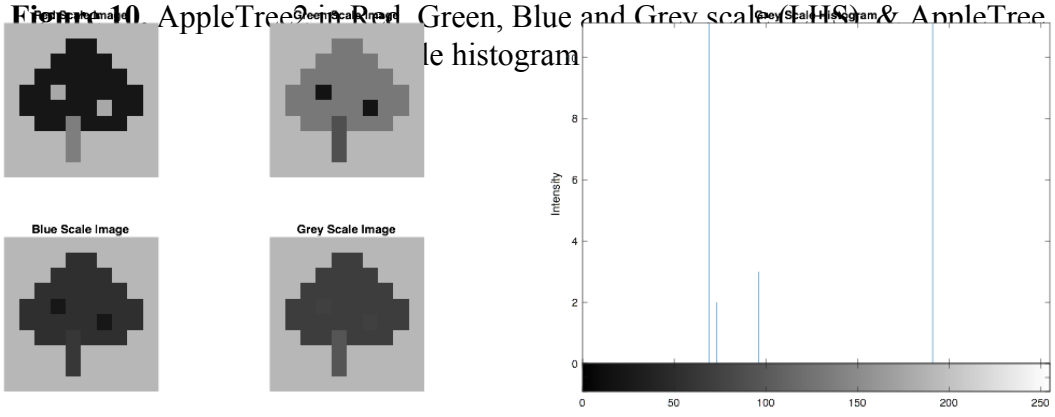


Figure 10. AppleTree3 in Red, Green, Blue and Grey scale (HIS) & AppleTree3 grey scale histogram

```
>> Problem1
Testing Image AppleTree3.png
Apple Coordinates: (x1, y1) = (4, 5)
Apple Coordinates: (x2, y2) = (7, 6)

Apple Coordinates Matrix:
     4     5
     7     6
```

Figure 11. Result of all apples coordinates in AppleTree3 image 2020

In conclusion, by testing 2 new image inputs, it has shown that the algorithm is sufficient to find locations of existing apples despite the change in color or brightness.

2. Problem 2

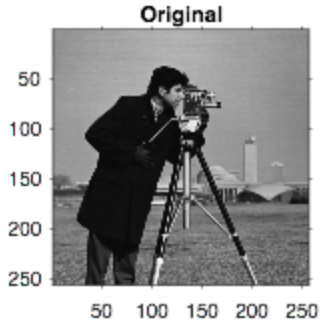


Figure 12. Original ‘cameraman.tif’ image 2020

2.1 Details of the algorithm (Theory / Concept):

The image has the same dimension for height and width. Therefore, it is assumed that the image is a square.

2.1.1 Rotate image by 30°:

To rotate the image, it is essential to ensure that the rotated image stays inside the bounding box with dimension of the original image dimension. It is in order to ensure that the image won’t get cutted off when being rearranged.

Therefore, let's consider the visualisation of the rotated image as shown below.

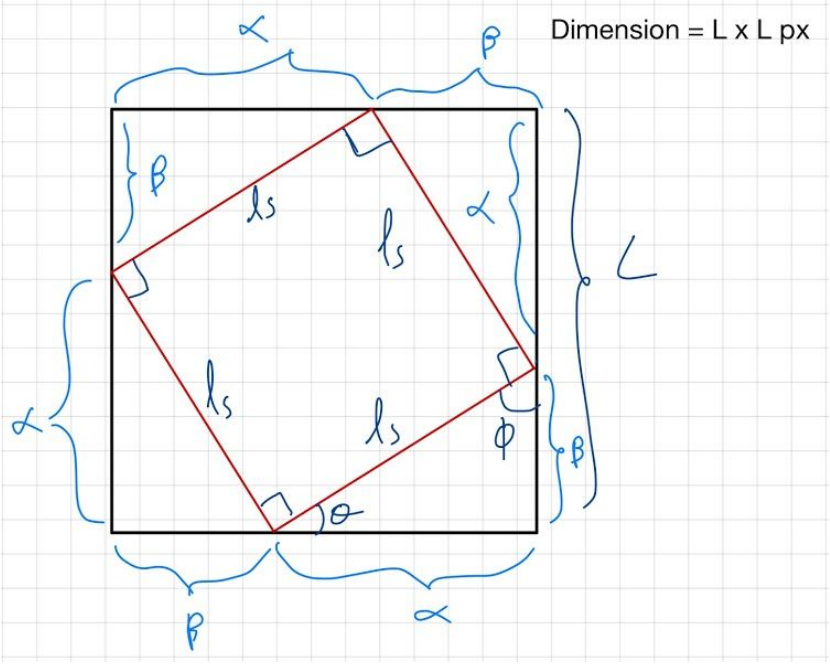


Figure 13. Mathematical modelling for image rotation algorithm 2020

Where the coordinate system of the computer window is as shown below, where y > 0 and x > 0.

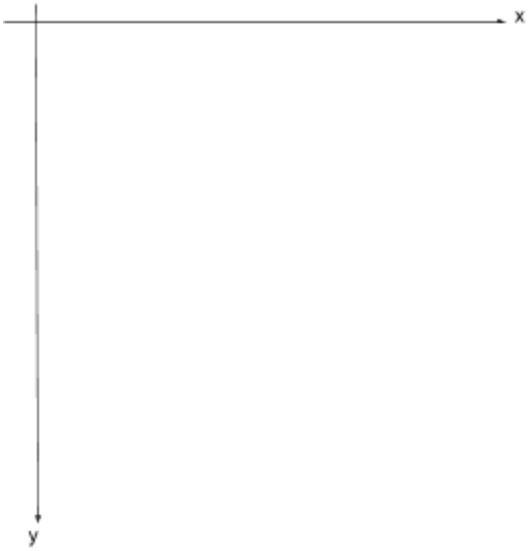


Figure 14. Computer window coordinate system 2020

From **Figure 13**, a mathematical model can be developed.

$$L = \alpha + \beta \text{, where } \alpha \neq \beta$$

$$\Rightarrow cos(\theta) = \frac{\alpha}{l_s};\ sin(\theta) = \frac{\beta}{l_s}$$

$$\Rightarrow \alpha = l_s cos(\theta);\ \beta = l_s sin(\theta)$$

$$\Rightarrow L = l_s cos(\theta) + l_s sin(\theta)$$

$$\Rightarrow l_s = \frac{L}{cos(\theta)+sin(\theta)}$$

Provided that the formulation for manipulating image's dimension, and transformation are shown below:

Scaling: $x' = xS_x;\ y' = yS_y$

Geometric Transformation:

$$x' = (xcos(\theta) + ysin(\theta))S_x + T_x;\ y' = (-xsin(\theta) + ycos(\theta))S_y + T_y$$

In this case, this will ensure that the rotated image will have the size of $l_s < L$, where the scale of the rotated image will be

$S_x = \frac{l_s}{Lx}$, & $S_y = \frac{l_s}{Ly}$, where $L_x = width$, & $L_y = height$

2.1.2 Shrink image by half:

The algorithm to demonstrate this operation is simply utilising the scaling formulation provided, where the new scale of image would be

$$S_x = S_y = \frac{1}{scale}$$

As an example, if it desires to reduce an image by half of its size, therefore the input scale is 2 as the image is expected to be twice smaller. Another example, would be if the input *scale* is equal to 3, then the image will be three times smaller, and etc.

2.1.3 Expand image by double its size:

The algorithm utilized to demonstrate this operation is to double the size of the grayscale image matrix via the scaling formulation above. Furthermore, when the image is being expanded, black pixels will appear within the image as a result of an expanded matrix. The idea proposed to solve this problem is to fill all black pixels by its nearest non-zero color value. In order to compute this idea, a 2-Dimensional convolution method would be utilized in this operation.

2.2 Explanation of the MATLAB codes you wrote:

```
I = imread('cameraman.tif');

figure, imshow(I), axis on; title('Original');

%Task 1: Rotate image by 30 degree.
askAngle = 'Enter angle (Degree):\n';

degree = input(askAngle);

%Store size of image
[Ly, Lx] = size(I);

Irotate = zeros(Ly,Lx);

%Guard for any rotation input that exceed 360
if(degree > 360)
    degree = degree - 360*fix(degree/360);
end

%Guards for negatives angle of rotation or clockwise rotation
if(degree < 0)
    if(degree < -360)
        degree = degree + 360*(fix(abs(degree/360)));
    end
    degree = degree + 360;
end


if degree > 90
    degree2 = degree - 90*fix(degree/90);
    ls = Lx /(cosd(degree2) + sind(degree2));

    %Algorithm for any rotation between 90-180 degree
    if(degree > 90 && degree < 180)
        Tx = abs(ls*cosd(degree));
        Ty = Ly;
    end

    %Algorithm for any rotation between 180-270 degree
    if(degree > 180 && degree <= 270)
        Tx = Lx;
        Ty = abs(ls*sind(90-degree));
    end

    %Algorithm for any rotation between 270-360 degree
    if(degree > 270 && degree <= 360)
        Tx = abs(ls*cosd(90-degree));
        Ty = 0;
    end

%Algorithm for any rotation between 0-90 degree
else
    ls = Lx /(cosd(abs(degree)) + sind(abs(degree)));
    Tx = 0;
    Ty = ls*sind(degree);
end

%New image scale for any rotated
Sx = ls/Lx;
Sy = ls/Ly;

%Loop for any geometric transformation made on the original
%image
for i = 1:Ly
    for j = 1:Lx
        irotate = round(Sy*(-j*sind(degree) + i*cosd(degree)) + Ty);
        jrotate = round(Sx*(j*cosd(degree) + i*sind(degree)) + Tx);

        if(irotate > 0) && (irotate <= Ly) ...
                && (jrotate > 0 ) && (jrotate <= Lx)
            Irotate(irotate, jrotate) =  I(i,j);
        end
    end
end

Irotate = uint8(Irotate);
figure, imshow(Irotate), axis on;
title('Rotated');


%Task 2: Shrink Image by Half

%Ask scale input to shrink image
askShrinkScale = ...
    'How many times you want to shrink the image? ';
scale = input(askShrinkScale);

fprintf('--> Shrinked Image Scale is (1 : %.2f)\n', scale);

%Re-scale image to new smaller scale
Sx = 1/scale;
Sy = 1/scale;

%Size of matrix will change depends on input
Ishrink = zeros(round(Ly/scale),round(Lx/scale));

%Loop for any smaller scale transformation made on the
%original image
for i = 1:Ly
    for j = 1:Lx
        ishrink = round(Sx*i);
        jshrink = round(Sy*j);

        if(ishrink > 0) && (ishrink <= round(Ly*Sy) ...
                && (jshrink > 0 ) && (jshrink <= round(Lx*Sx)))
            Ishrink(ishrink, jshrink) =  I(i,j);
        end
    end
end

Ishrink = uint8(Ishrink);
figure, imshow(Ishrink), axis on;
title('Shrinked');


%Task 3: Expand Image by double

%Ask scale input to expand image
askExpandScale = ...
    'How many times you want to expand the image? ';
scale = input(askExpandScale);
fprintf('--> Expanded Image Scale is (%.2f : 1)\n', scale);

%Size of matrix will change depends on new expanded image size
Iexpand = zeros(scale*Ly, scale*Lx);

%Loop for any scale expansion transformation made on the
%original image
for i = 1:scale*Ly
    for j = 1:scale*Lx
        iexpand = round(scale*i);
        jexpand = round(scale*j);

        if(iexpand > 0) && (iexpand <= scale*Ly) ...
                && (jexpand > 0 ) && (jexpand <= scale*Lx)
            Iexpand(iexpand, jexpand) =  I(i,j);
        end
    end
end

%Loop for any scale expansion transformation made on the
%original image
for i = 1:scale*Ly
    for j = 1:scale*Lx
        iexpand = round(scale*i);
        jexpand = round(scale*j);

        %Guard
        if(iexpand > 0) && (iexpand <= scale*Ly) ...
                && (jexpand > 0 ) && (jexpand <= scale*Lx)
            Iexpand(iexpand, jexpand) =  I(i,j);
        end
    end
end



%Filling Black Pixels at the nearest existing non-zero colour
%neighbourhood

RefilledIexpand = Iexpand;

%Convolution
for i = 2:size(Iexpand,1)-1
    for j = 2:size(Iexpand,2)-1

        %Iterate Iexpand by per 6x6 matrix
        tempMatrix = Iexpand(i-1:i+1,j-1:j+1);

        %Guard to check if the mid value tempMatrix is 0
        %(Black Pixel) and making sure that it is surround
        %by a non-zero color
```

```
173 -           tempSum = sum(tempMatrix(:));
174 -           if(tempMatrix(5)== 0 && tempSum ~= 0 )
175
176                 %Array of non-zero element index position number
177 -               NonZeroIndexPos = find(tempMatrix~=0);
178
179                 %Array contains distances of non-zero elements from
180                 %the mid element's position
181 -               DistanceFromMidElement = abs(NonZeroIndexPos-5);
182
183                 %Ignore the actual distance values and take array of
184                 %its actual index position number
185 -               [~,distanceIndexPost] = sort(DistanceFromMidElement);
186
187                 %Fill the black pixel by the non-zero value from the mid
188                 %element's nearest neighbor.
189 -               RefilledIexpand(i,j) = tempMatrix(NonZeroIndexPos(...
190                     distanceIndexPost(1)));
191 -           end
192 -       end
193 -   end
194
195     %Change values of Iexpand with filled black pixels matrix
196 -   Iexpand = uint8(RefilledIexpand);
197 -   figure, imshow(Iexpand), axis on; title('Expanded');
198
199     %Comparison of output images with the orignal image
200 -   figure;
201 -   subplot(2,2,1), imshowpair(I, Irotate), axis on;
202 -   title('Original Vs Rotated');
203 -   subplot(2,2,2), imshowpair(I, Ishrink), axis on;
204 -   title('Original Vs Shrinked');
205 -   subplot(2,2,[3, 4]), imshowpair(I, Iexpand), axis on;
206 -   title('Original Vs Expanded ');
207
208
```

Figure 15. MATLAB code to rotate, shrink, and expand image 2020

Task 1 (Rotating image): Line 5 → 78

Line 16 → 18:

These lines of code will change any input degree angle that exceed 360° to its equivalent degree that is within $0° \leq \theta \leq 360°$. Where **fix** function means to rundown a value. Therefore, in this computation, if input degree is 405°, the new degree is

$$degree = 405° - 360 * fix(405°/360°) = 405° - 360° * fix(1.125) = 405° - 360° = 45°$$

Another example, would be that is degree input is 750°

$$degree = 750° - 360 * fix(750°/360°) = 750° - 360° * fix(2.08333) = 750° - 360° * 2 = 30°$$

It is make sense, because $360°$ is $0°$ after 1 revolution, and $720°$ is $0°$ after 2 revolutions, and etc.

Line 21 → 26:

This will convert any negative degree angle to its equivalent positive degree angle, and hence allow for clockwise image rotation. For example, $-30°$ is equivalent to $330°$ since it is located in the 4th quadrant. Furthermore, the same algorithm is applied for any negative angles that is less than $-360°$ and convert them to its equivalent angle that is within $-360° \leq \theta \leq 0°$ and convert it to its equivalent angle that is within $0° \leq \theta \leq 360°$.

Line 29 → 56:

These lines will compute value of T_x, T_y, & l_s depending on which quadrant does the input angle lies at.

Task 2 (Shrinking image): Line 81 → 113

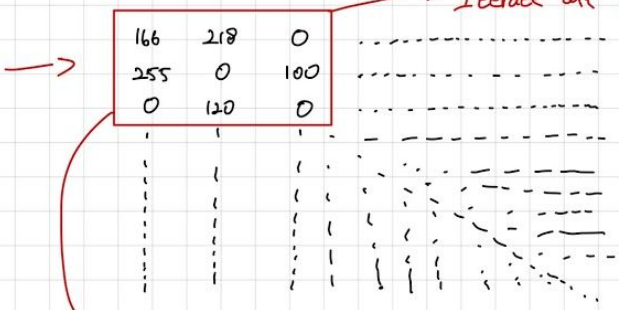
Shrinking image by reading input of how times the image wants to be shrinked. Furthermore, the loop to fill Ishrink matrix will be bounded by the new shrinked image dimension, where the size of Ishrink matrix will adapt to different input values.

Task 3 (Expanding image) : Line 116 → 197

The same process is done for tasking, but the difference is that the size of Iexpand image will be bigger than the original image matrix and will adapt to any input given into the system.

Line 163 - 193:

Convolution process that will refill black pixels in Iexpand matrix by the nearest colour value in its neighbourhood. Below is the diagram to show how the process works. Consider the example below, to understand the systematic operation of the represented code from these lines.



→ Iterate all elements in I expand per 6×6 matrix

Note: 6×6 matrix will be analyse if and only if the middle element of matrix is zero and elements in its neighbourhood are not-all zero

→ let this be tempMatrix = $\begin{bmatrix} 166 & 218 & 0 \\ 255 & 0 & 100 \\ 0 & 120 & 0 \end{bmatrix}$

code: • tempMatrix(5)==0, check if the middle element is 0. where in this case it is 0.

• tempMatrix(:) = $\begin{bmatrix} 166 \\ 218 \\ 0 \\ 255 \\ 0 \\ 100 \\ 0 \\ 120 \\ 0 \end{bmatrix}$, change 6×6 matrix into one dimension

• tempSum = sum(tempMatrix(:))
= 166 + 218 + 0 + 255 + 0 + 100 + 0 + 120 + 0

∴ tempSum ≠ 0

∴ as tempMatrix(5) = 0 and tempSum ≠ 0

→ 6×6 Matrix will be analysed

⇒ NonZeroIndexPos = find(tempMatrix~=0) ⇒ $\begin{bmatrix} 166 \\ 218 \\ 0 \\ 255 \\ 0 \\ 100 \\ 0 \\ 120 \\ 0 \end{bmatrix}$ elements, index, NonZeroIndex Pos.

⇒ NonZeroIndexPos = $\begin{bmatrix} 1 \\ 2 \\ 4 \\ 6 \\ 8 \end{bmatrix}$

Now,

middle element is the 5th element.

DistanceFromMidElement = abs(NonZeroIndexPos − 5)

$$= \left| \begin{bmatrix} 1 \\ 2 \\ 4 \\ 6 \\ 8 \end{bmatrix} - \begin{bmatrix} 5 \\ 5 \\ 5 \\ 5 \\ 5 \end{bmatrix} \right|$$

$$= \begin{bmatrix} 4 \\ 3 \\ 1 \\ 1 \\ 3 \end{bmatrix}$$

element → index

1
2
3
4
5

Now,

[~, distanceIndexPost] = sort(DistanceFromMidElement)

new arrangement original index position.

$$= \begin{bmatrix} 1 \\ 1 \\ 3 \\ 3 \\ 4 \end{bmatrix} \text{ where } \begin{bmatrix} 3 \\ 4 \\ 2 \\ 5 \\ 1 \end{bmatrix}$$

∴ [~, distanceIndexPost] will ignore $\begin{bmatrix} 1 \\ 1 \\ 3 \\ 3 \\ 4 \end{bmatrix}$ and take $\begin{bmatrix} 3 \\ 4 \\ 2 \\ 5 \\ 1 \end{bmatrix}$ instead

=> distanceIndexPost = $\begin{bmatrix} 3 \\ 4 \\ 2 \\ 5 \\ 1 \end{bmatrix}$

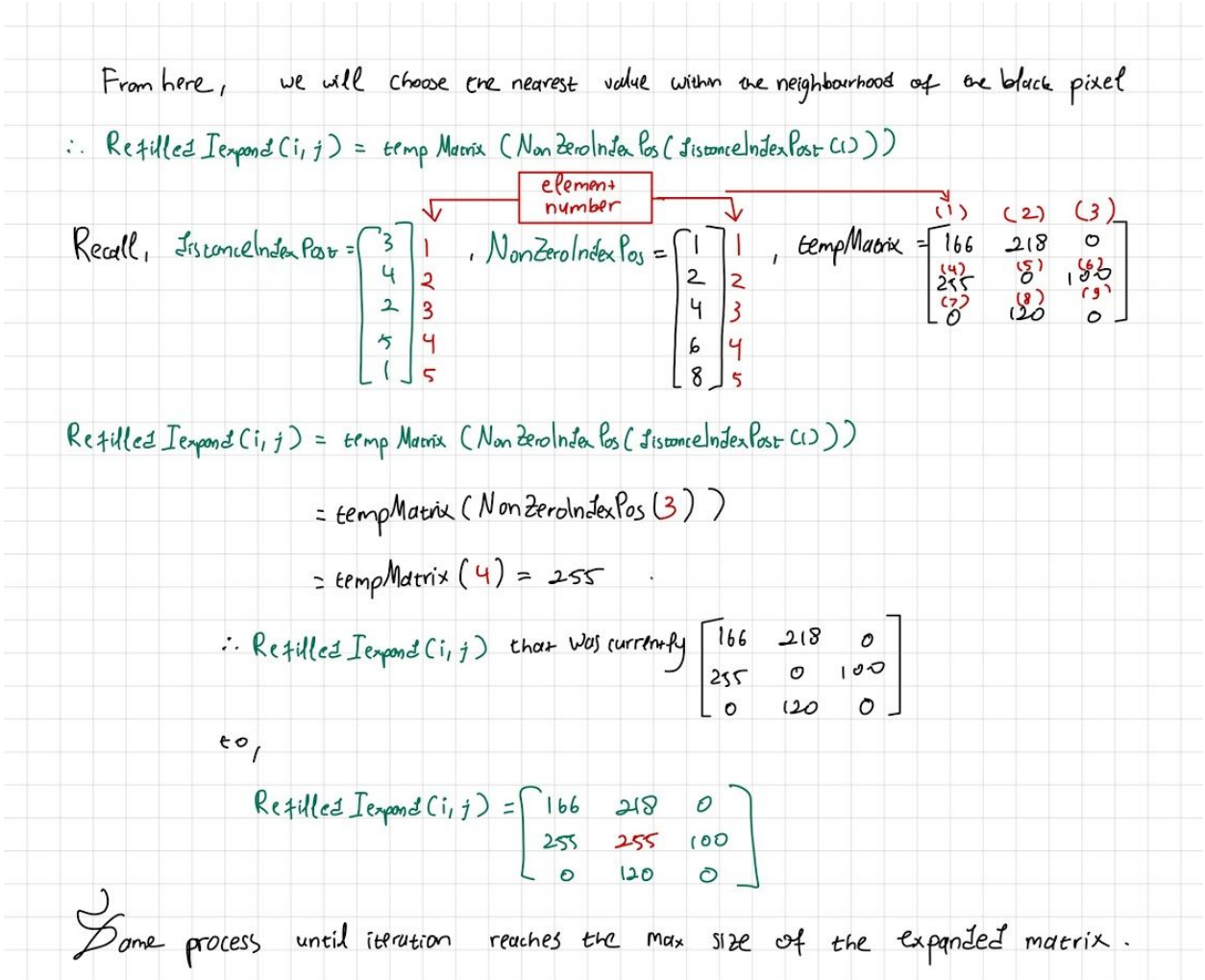
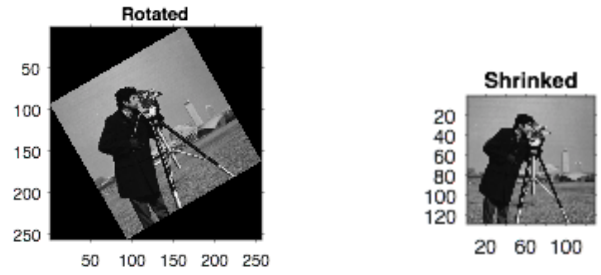


Figure 16. Algorithm or the process of filling the black pixel 2020

The benefit of the algorithm represented in the MATLAB code and in **Figure 16**, is the ability to fill the black pixel with color value that makes the expanded image to look as natural as possible.

2.3 The result



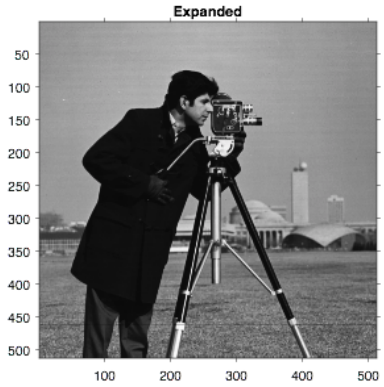


Figure 17. Image is rotated by 30°, shrinked to ½ its size, & expanded to 2 times its size 2020

Note: Result in **Figure 17** is generated from "Problem2_SolvingMainTasks.m" file

2.3.1 Comparison:

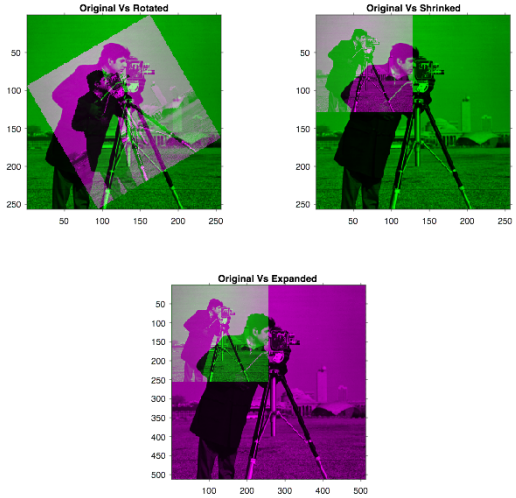


Figure 18. Showing the comparison of transformed and original image 2020

Shown below, are the alternative case studies to check the robustness of the program/algorithm, utilized for all tasks in problem 2. Where results below are generated from "Problem2_TakesMoreInput.m" file.

2.3.2 Alternative case Study 1

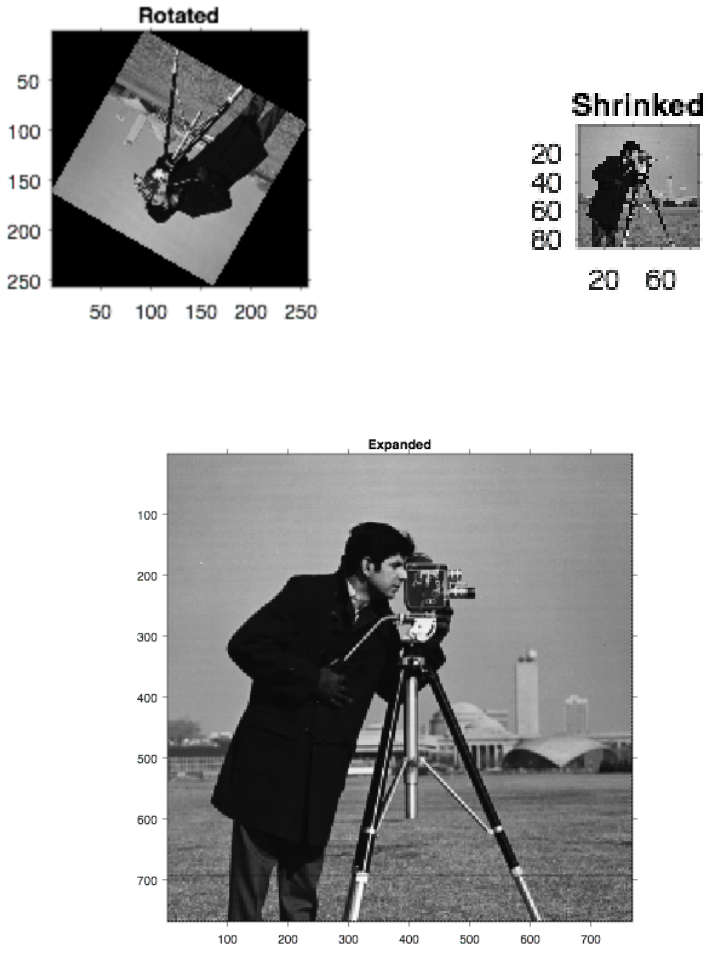


Figure 19. Image is rotated by 150°, shrinked to ⅓ its size, & expanded to 3 times its size 2020

2.3.3 Alternative Case Study 2

```
>> Problem2
Enter angle (Degree):
-30
Shrink image by 2.5
--> Shrinked Image Scale is (1 : 2.50)
Expand image by 2.5
--> Expanded Image Scale is (2.50 : 1)
```

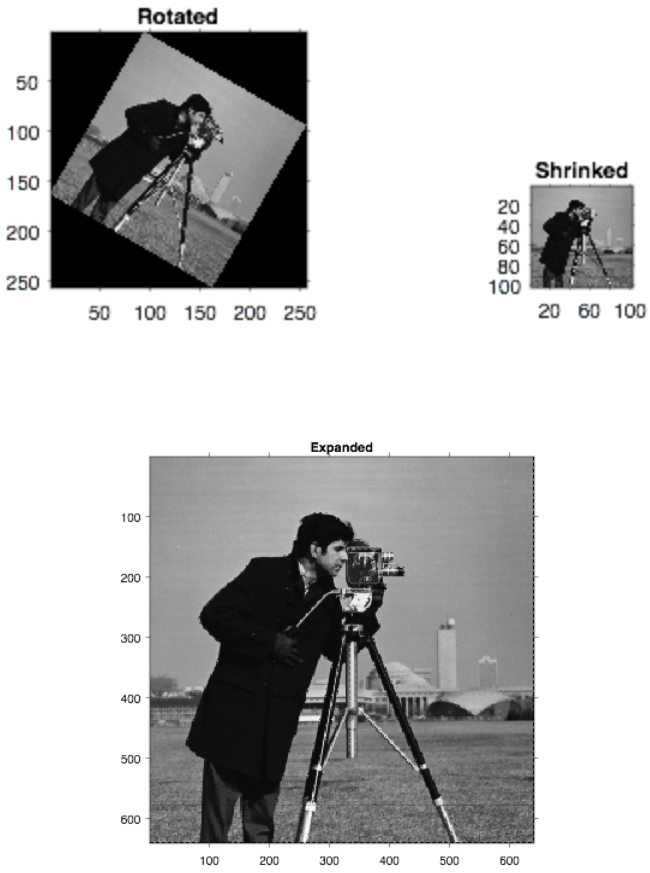


Figure 20. Image is rotated by -30°, shrinked to 1/2.5 its size, & expanded to 2.5 times its size 2020

2.3.4 Alternative case Study 3

```
>> Problem2
Enter angle (Degree):
405
Shrink image by 4
--> Shrinked Image Scale is (1 : 4.00)
Expand image by 4
--> Expanded Image Scale is (4.00 : 1)
```

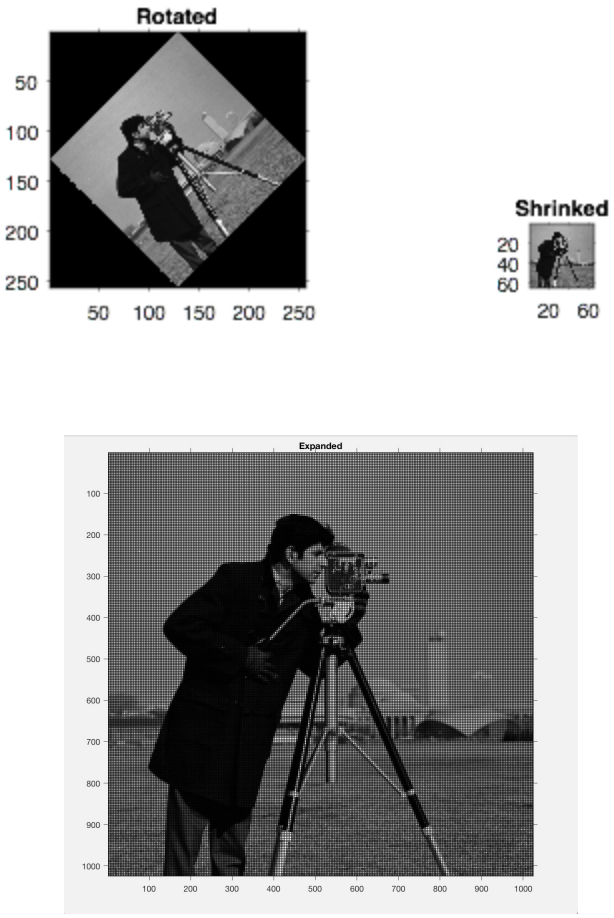


Figure 21. Image is rotated by 405°, shrinked to ¼ its size, & expanded to 4 times its size 2020

3. Problem 3

3.1 Details of the algorithm (Theory / Concept):

This problem required the detection of location of corners, given multiple shape images. Given that the images were essentially binary (each pixel was one of two values; 255 for white and 0 for black), the essence of the problem was in detecting the gradients throughout the image and using the gradients to detect the corners.

Horizontal and vertical edge detection can be done using multiple 3x3 sliding masks and applying it to each window of the image matrix. These masks are:

$$Gx1 = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gx2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$Gy1 = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gy2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

Gx1 identifies gradients from left to right, Gx2 identifies gradients from right to left, Gy1 identifies gradients from top to bottom and Gy2 identifies gradients from bottom to top.

For a square image, this edge detection technique would give us the following output:

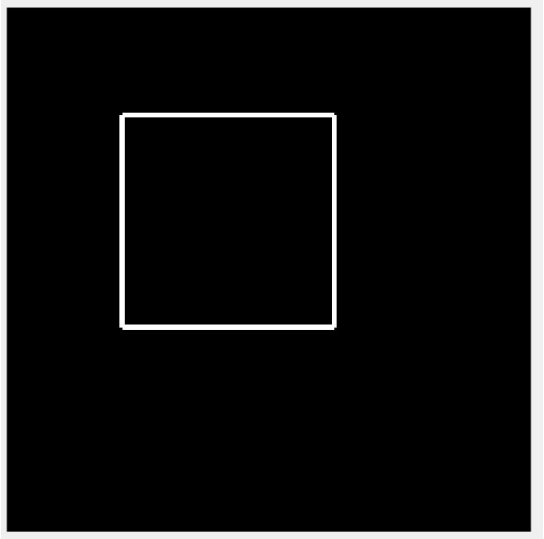


Figure 22. 'WhiteSquare2019.tif' edge detection image 2020

As can be seen, all the horizontal and vertical edges have been detected. Before any arithmetic has been applied, the new matrix shows exact gradients, as shown below:

0	0	255.0000	255.0000	0	0
0	0	255.0000	255.0000	0	0
0	0	255.0000	255.0000	0	0
0	0	255.0000	255.0000	0	0
0	0	255.0000	255.0000	0	0
0	0	255.0000	255.0000	0	0
255.0000	255.0000	360.6245	255.0000	0	0
255.0000	255.0000	255.0000	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

In the matrix segment above, the left corner of the square is shown with the edges clearly shown as the pixels with a number above 0. It can also be seen that there is a single pixel with a value of 360.6245, which is higher than all of its surrounding pixels. This can be easily identified as the corner of the square. So by looping through the entire matrix and finding only the pixels with a value of 360.6245, we can output only the corners. This process works perfectly for squares and rectangles.

However, with shapes that do not have right angle corners, and only 90 degree edges, this process does not work. Using this algorithm on the diamond and triangle shapes, gives the following output:

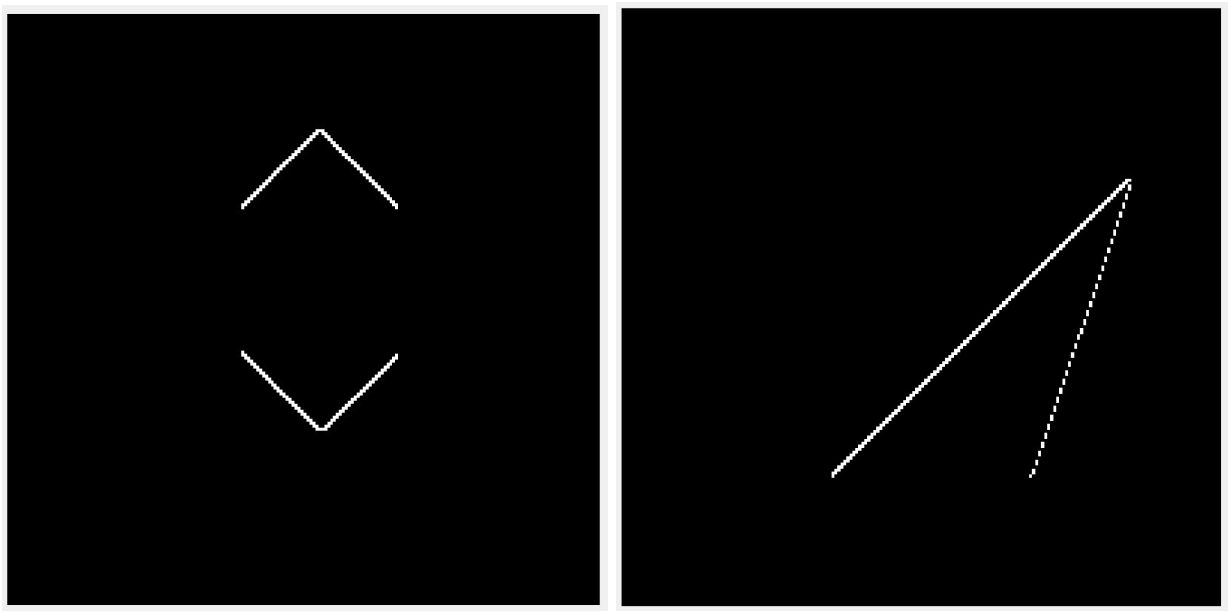


Figure 23. ‘WhiteDiamon2019.tif’ (left hand side) & ‘WhiteTriangle2019.tif’(right hand side) edge detection 2020

This shows that the corners of the shapes have been detected, but so have all the diagonal edges. This is because the diagonal edge pixels have a large vertical and horizontal gradient, and so therefore have a pixel value of over 360.

To combat this, the diagonal edges first need to be detected, using diagonal gradient masks. These masks are:

$$Gxy1 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gxy2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

$$Gxy3 = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gxy4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

By applying these masks to the image matrix along with the vertical and horizontal masks, our output edge detection matrix gives more information about the shape and hence makes it easier to detect edges.

As the edges of each shape are different, we cannot use a single pixel to compare the entire matrix against, as we did previously with the square and rectangle. The large angled diagonal edges also give very high pixel values due to the high gradients in every direction. So, instead of comparing each pixel to a fixed number, each pixel was instead compared to its neighbouring pixels.

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	255.0000	255.0000	255.0000	0
0	0	255.0000	441.6730	624.6199	360.6245	0
0	255.0000	441.6730	441.6730	510.0000	441.6730	0
5.0000	441.6730	441.6730	360.6245	510.0000	360.6245	0
1.6730	441.6730	255.0000	360.6245	510.0000	255.0000	0
1.6730	255.0000	0	441.6730	441.6730	0	0
5.0000	0	255.0000	510.0000	360.6245	0	0
0	0	360.6245	510.0000	255.0000	0	0
0	0	441.6730	441.6730	0	0	0
0	255.0000	510.0000	360.6245	0	0	0
0	360.6245	510.0000	255.0000	0	0	0
0	441.6730	441.6730	0	0	0	0
5.0000	510.0000	360.6245	0	0	0	0
0.6245	510.0000	255.0000	0	0	0	0
1.6730	441.6730	0	0	0	0	0
0.0000	360.6245	0	0	0	0	0
0.0000	255.0000	0	0	0	0	0
1.6730	0	0	0	0	0	0
0.6245	0	0	0	0	0	0
5.0000	0	0	0	0	0	0

Shown above is the top right corner of the triangle shape. It can be seen that the diagonal edges give large pixel values, but the corner pixel (624.6199) is still higher than all the surrounding pixels. In this way, we can identify the corners of the shape by only identifying the pixels which

are larger than every pixel in the 3x3 window surrounding it. In the above matrix, the only pixel that would be identified as a corner is the 624 pixel.

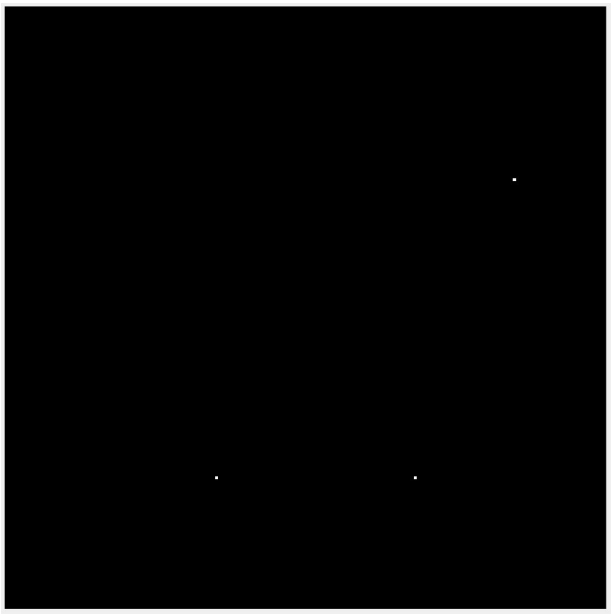


Figure 24. 'WhiteTriangle2019.tif' corner detection 2020

This algorithm works perfectly on the triangle as can be seen above with the three corner pixels identified. It also works on every corner of the diamond shape, except for one corner:

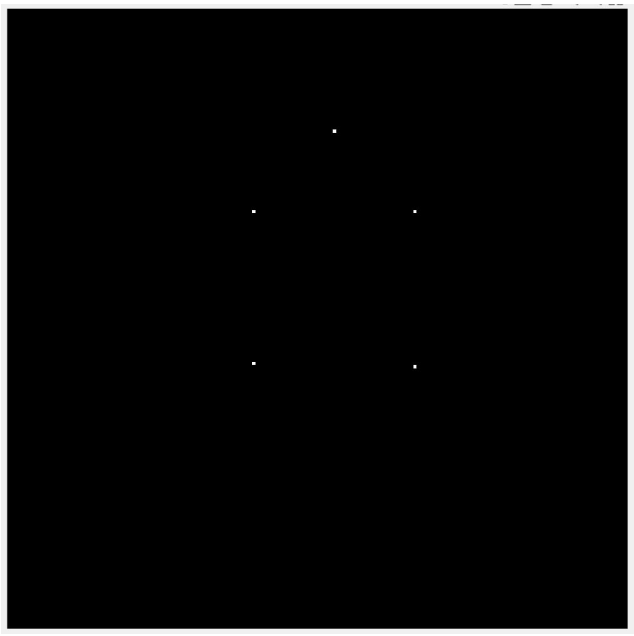


Figure 25. 'WhiteDiamond2019.tif' corner detection 2020

The algorithm doesn't work on the bottom corner of the diamond. Looking closely at the matrix, we can see the bottom corner values as:

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
255.0000	0	0	0	0	255.0000
441.6730	255.0000	0	0	255.0000	441.6730
441.6730	441.6730	255.0000	255.0000	441.6730	441.6730
255.0000	441.6730	510.0000	510.0000	441.6730	255.0000
0	255.0000	360.6245	360.6245	255.0000	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

The corner of the shape is actually two pixels instead of one and is shown here as the two pixels of 510. The algorithm doesn't detect this as a corner because each pixel is not greater than the other pixel and so is not identified. To combat this, we can expand the window to check if each pixel is greater than its surrounding 9x9 pixels, but exclude its direct neighbours. This would then mean that any corner that is made up of two pixels could still be identified as a corner.

3.2 Explanation of the MATLAB codes you wrote:

```
function output = corners(I)

    % Store size of image matrix
    m = size(I,1);
    n = size(I,2);

    % Set up output matrix as same size as input
    output = zeros(m,n);

    % Convert image matrix to double type for mathematical operations
    doubleImage = double(I);

    % Declare edge detection masks
    Gx1 = zeros(m,n);
    Gx2 = zeros(m,n);
    Gy3 = zeros(m,n);
    Gy4 = zeros(m,n);

    Gxy1 = zeros(m,n);
    Gxy2 = zeros(m,n);
    Gxy3 = zeros(m,n);
    Gxy4 = zeros(m,n);

    % Loop through every pixel in image and apply horizontal, vertical and
    % diagonal masks to find areas of high gradient within the image. This
    % will essentially detect all the edges in the image
    for i = 2:m-1
        for j = 2:n-1
            Gx1(i,j) = (-1*doubleImage(i-1,j) + doubleImage(i,j));
            Gx2(i,j) = (-1*doubleImage(i+1,j) + doubleImage(i,j));
            Gy3(i,j) = (-1*doubleImage(i,j-1) + doubleImage(i,j));
            Gy4(i,j) = (-1*doubleImage(i,j+1) + doubleImage(i,j));

            Gxy1(i,j) = (-1*doubleImage(i-1,j-1) + doubleImage(i,j));
            Gxy2(i,j) = (-1*doubleImage(i-1,j+1) + doubleImage(i,j));
            Gxy3(i,j) = (-1*doubleImage(i+1,j-1) + doubleImage(i,j));
            Gxy4(i,j) = (-1*doubleImage(i+1,j+1) + doubleImage(i,j));
        end
    end
```

```
    % Find absolute values of matrices to remove negative gradients
    Gx1 = sqrt(Gx1.^2);
    Gx2 = sqrt(Gx2.^2);
    Gy3 = sqrt(Gy3.^2);
    Gy4 = sqrt(Gy4.^2);

    Gxy1 = sqrt(Gxy1.^2);
    Gxy2 = sqrt(Gxy2.^2);
    Gxy3 = sqrt(Gxy3.^2);
    Gxy4 = sqrt(Gxy4.^2);

    % Find sum of all gradients
    corner = sqrt(Gx1.^2 + Gx2.^2 + Gy3.^2 + Gy4.^2 + Gxy1.^2 + Gxy2.^2 + Gxy3.^2 + Gxy4.^2);
```

```
    % Loop through each pixel of new matrix to identify corners. Corners
    % are areas where the gradient is high in multiple directions. This is
    % where the pixel values are higher than their surrounding pixels
    for i = 5:m-4
        for j = 5:n-4
            % find maximum pixel value in 3x3 window around current pixel
            maxwind = max([corner(i-1,j-1) corner(i-1,j) corner(i-1,j+1) corner(i,j-1) corner(i,j+1) corner(i+1,j-1) corner(i+1,j) corner(i+1,j+1)]);
            if (corner(i,j) > maxwind)
                % if current pixel value is greater than every pixel
                % surrounding it, then it is a corner
                output(i,j) = 255;
            elseif (corner(i,j) == maxwind) % if pixel is equal to the largest pixel of the current window
                maxtotal = 0;
                % find the largest pixel value within each 3x3 window
                % surrounding the current 3x3 window
                for x = i-3:3:i+3
                    for y = j-3:3:j+3
                        if ~(x == i && y == j)
                            maxtotal = max([maxtotal corner(x-1,y-1) corner(x-1,y) corner(x-1,y+1) corner(x,y-1) corner(x,y) corner(x,y+1) corner(x+1,y-1) corner(x+1,y) corner(x+1,y+1)]);
                        end
                    end
                end
                % if the current pixel is larger than every pixel within
                % the current 9x9 window, excluding the 3x3 window it is
                % in, then it is a corner
                if (corner(i,j) > maxtotal)
                    output(i,j) = 255;
                end
            else
                % every pixel that isn't detected as a corner, is made to
                % be black
                output(i,j) = 0;
            end
        end
    end

    % Convert output matrix back to uint8 type to display as image
    output = uint8(output);

end
```

Figure 26. MATLAB code for corner detection function 2020

3.3 The results

Function call:

```
clc
clear

square = imread('WhiteSquare2019.tif');
square = square(:,:,1);
squarecorners = corners(square);

rectangle = imread('WhiteRectangle2019.tif');
rectangle = rectangle(:,:,1);
rectanglecorners = corners(rectangle);

diamond = imread('WhiteDiamond2019.tif');
diamond = diamond(:,:,1);
diamondcorners = corners(diamond);

triangle = imread('WhiteTriangle2019.tif');
triangle = triangle(:,:,1);
trianglecorners = corners(triangle);

figure(1);
subplot(1,2,1);
imshow(square);
title("Original Image");
subplot(1,2,2);
imshow(squarecorners);
title("Corners");

figure(2);
subplot(1,2,1);
imshow(rectangle);
title("Original Image");
subplot(1,2,2);
imshow(rectanglecorners);
title("Corners");

figure(3);
subplot(1,2,1);
imshow(diamond);
title("Original Image");
subplot(1,2,2);
imshow(diamondcorners);
title("Corners");

figure(4);
subplot(1,2,1);
imshow(triangle);
title("Original Image");
subplot(1,2,2);
imshow(trianglecorners);
title("Corners");
```

Figure 27. MATLAB code that call function for corner detection 2020

Output:

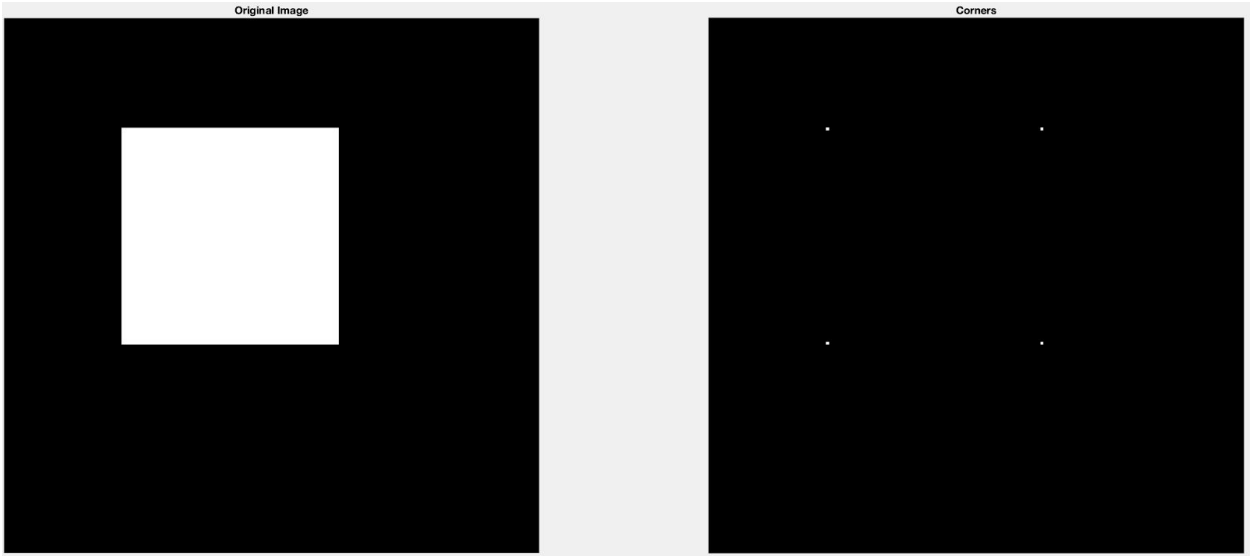


Figure 28. 'WhiteSquare2019.tif' corner detection 2020

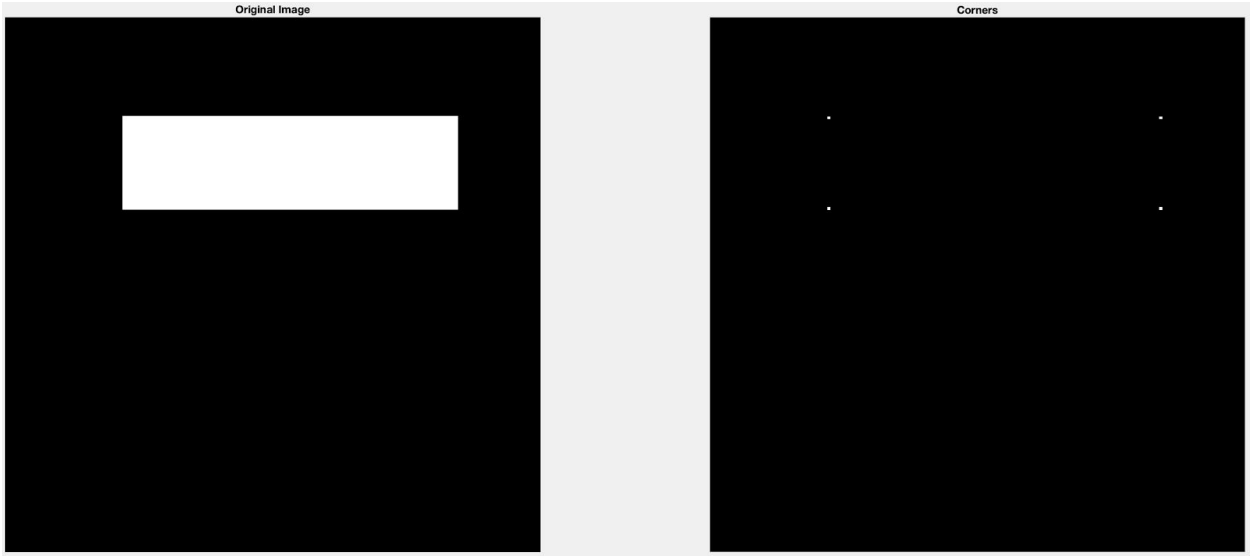


Figure 29. 'WhiteRectangle2019.tif' corner detection 2020

Figure 30. 'WhiteDiamond2019.tif' corner detection 2020

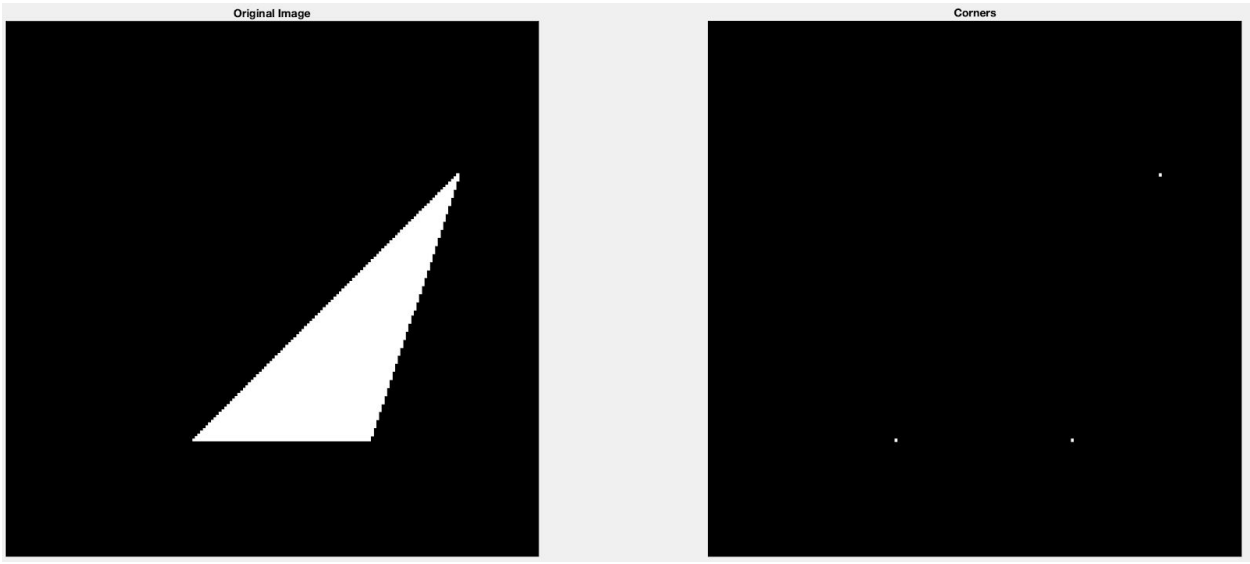


Figure 31. 'WhiteTriangle2019.tif' corner detection 2020

4. Problem 4

4.1 Details of the algorithm (Theory / Concept):

This problem required the detection of multiple specifications of various shapes. These specifications were centroids, bounding boxes, major axes, angle, area, perimeter and circularity.

Centroid:

The centroid of a shape can be found by first finding the moments of the shape. The moments can be found using the below general formula:

$$M_{pq} = \sum_{(x,y)\in Image} x^p y^q I(x,y)$$

For this problem, the moments found were M00, M01, M10, M11, M02 and M20.

To find the centroid of any shape, we can use the below equations:

$$X_c = \frac{M_{10}}{M_{00}} \quad Y_c = \frac{M_{01}}{M_{00}}$$

Bounding Box:

The bounding box of a shape is the smallest rectangle that can fit the entire shape inside it. To do this, the left most, right most, top most and bottom most white pixel needs to be found. This can be done by looping through the image matrix starting from each of the four sides. When the first white pixel is found, the pixel position is stored and the next side of the matrix is looped.

Once the minimum and maximum x and y values are found, the bounding box has the top left corner at (xmin, ymin) and bottom right corner at (xmax, ymax).

Major axes and angle:

There are a few steps in finding the major axes of a shape. The first is to find the moments with respect to the centroid of the shape. The following equations can be used to find them:

$$U_{11} = M_{11} - X_c M_{01}$$
$$U_{20} = M_{20} - X_c M_{10}$$
$$U_{02} = M_{02} - Y_c M_{01}$$

These moments are used to create the inertia matrix of the shape which is:

$$J = \begin{bmatrix} U_{20} & U_{11} \\ U_{11} & U_{02} \end{bmatrix}$$

$\lambda 1$ and $\lambda 2$ are found as the eigenvalues of J, where $\lambda 1$ is the larger eigenvalue, and V is found as the eigenvector of J corresponding to the largest eigenvalue. The axes lengths are found using the equations:

$$a = 2\sqrt{\frac{\lambda_1}{M_{00}}} \quad b = 2\sqrt{\frac{\lambda_2}{M_{00}}}$$

The axes angle is found using:

$$\theta = \arctan\left(\frac{V_y}{V_x}\right)$$

Area:

The area of a shape is equal to the 0 moment of the shape.

$$Area = M_{00}$$

Perimeter:

The perimeter of the shape can be found by first using the same edge detection technique as used in problem 3 where the horizontal, vertical and diagonal gradients are detected using the following masks:

$$Gx1 = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gx2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$Gy1 = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gy2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

$$Gxy1 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gxy2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

$$Gxy3 = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad Gxy4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Once the masks have been applied to the image matrix, they are added together without finding the absolute values. This is so that any negative gradients can be removed in the next step, so that edges are not being counted twice.

The diagonal gradients must be detected as well as the horizontal and vertical gradients to ensure that the length of diagonal edges are calculated accurately.

If using only vertical and horizontal edge detection, diagonal edges end up looking like the image below:

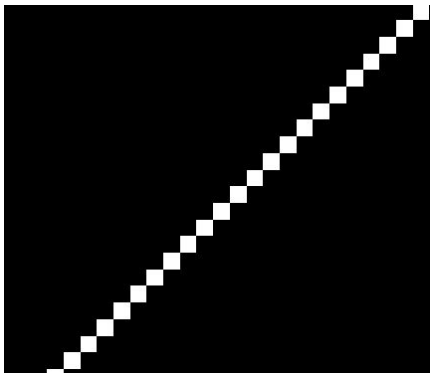


Figure 32. Diagonal edges by utilising vertical and horizontal edge detection 2020

This means that when summing the white pixels to calculate perimeter, the perimeter is hugely underestimated compared to a horizontal edge. When including the diagonal edge detection, we get the diagonal edges more closely reflecting the actual perimeter of the shape, as seen in the below image. This also ensures that the circularity is more accurate in the next step.

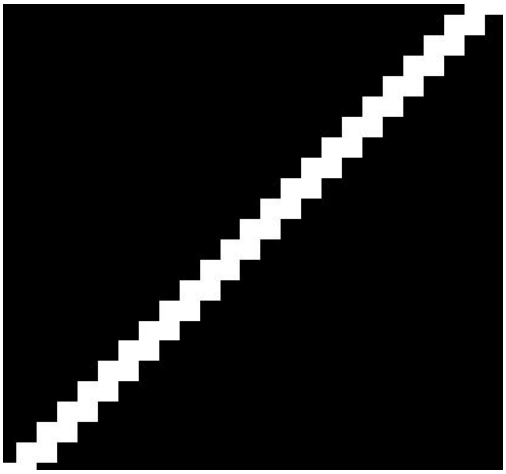


Figure 33. Diagonal edges by utilising diagonal edge detection 2020

Circularity:

The circularity of a shape is essentially how circular the shape is. The more circular a shape is, the larger its area is with respect to its perimeter. A perfect circle should have a circularity of 1, a square has a circularity of $\frac{\pi}{4}$ and a line has a circularity of 0. Calculating the circularity of a shape can help determine what shape it is.

The circularity can be calculated using the below formula:

$$Circularity = \frac{4\pi M_{00}}{p^2}$$

As the images used in the problem are pixelated, the calculated perimeter of each shape is not completely accurate. As such, the circularity of each image is also not completely accurate. However, the circularities calculated are very close to the actual values, and still assist with determining what shape it is.

4.2 Explanation of the MATLAB codes you wrote:

```
function [xc,yc,xmin,xmax,ymin,ymax,a,b,theta1,theta2,area,perim,circularity] = specs(I)

    % Store size of image matrix
    m = size(I,1);
    n = size(I,2);

    % Convert image matrix to double type for mathematical operations
    doubleImage = double(I);

    % Declare max and min edges of shape
    xmin = m;
    xmax = 0;
    ymin = n;
    ymax = 0;

    % Loop through pixels from top to bottom. Once it finds a white square,
    % it stores pixel as minimum y value and breaks loop
    inShape = 0;
    for i = 1:m
        for j = 1:n
            if doubleImage(i,j) == 255
                inShape = 1;
                ymin = i;
                break
            end
            if inShape == 1
                break
            end
        end
        if inShape == 1
            break
        end
    end

    % Loop through pixels from bottom to top. Once it finds a white square,
    % it stores pixel as maximum y value and breaks loop
    inShape = 0;
    for i = m:-1:1
        for j = 1:n
            if doubleImage(i,j) == 255
                inShape = 1;
                ymax = i;
                break
            end
            if inShape == 1
                break
            end
        end
        if inShape == 1
            break
        end
    end

    % Loop through pixels from left to right. Once it finds a white square,
    % it stores pixel as minimum x value and breaks loop
    inShape = 0;
    for j = 1:n
        for i = 1:m
            if doubleImage(i,j) == 255
                inShape = 1;
                xmin = j;
                break
            end
            if inShape == 1
                break
            end
        end
        if inShape == 1
            break
        end
    end
```

```
% Loop through pixels from right to left. Once it finds a white square,
% it stores pixel as maximum x value and breaks loop
inShape = 0;
for j = n:-1:1
    for i = 1:m
        if doubleImage(i,j) == 255
            inShape = 1;
            xmax = j;
            break
        end
        if inShape == 1
            break
        end
    end
    if inShape == 1
        break
    end
end

% Convert image matrix to binary (1 for white, 0 for black)
binaryImage = doubleImage > 0;
binaryImage = double(binaryImage);

% Declare moments
m00 = 0;
m01 = 0;
m10 = 0;
m11 = 0;
m20 = 0;
m02 = 0;

% Loop through binary image matrix to find moments
for i = 1:m
    for j = 1:n
        m00 = m00 + binaryImage(i,j);
        m01 = m01 + i*binaryImage(i,j);
        m10 = m10 + j*binaryImage(i,j);
        m11 = m11 + i*j*binaryImage(i,j);
        m20 = m20 + (j^2)*binaryImage(i,j);
        m02 = m02 + (i^2)*binaryImage(i,j);
    end
end
```

```
% Use moments to find centroid x and y value
xc = m10/m00;
yc = m01/m00;

% Store area value
area = m00;

% Use moment equations to find J matrix
u11 = m11 - xc*m01;
u20 = m20 - xc*m10;
u02 = m02 - yc*m01;
J = [u20 u11;u11 u02];

% Find eigen values of J matrix and store in lamda variables
lamda = eig(J);
lamda1 = max(lamda(:));
lamda2 = min(lamda(:));

% Find axes lengths
a = 2*sqrt(lamda1/m00);
b = 2*sqrt(lamda2/m00);

% Find eigenvector of J matrix
[V,D] = eig(J);

% Find eigenvector for largest eigenvalue
lamda1Index = find(lamda == lamda1);

% Find axis angles
theta1 = atan(V(2,lamda1Index)/V(1,lamda1Index));
theta2 = theta1 + pi/2;

% Declare edge detection masks
Gx1 = zeros(m,n);
Gx2 = zeros(m,n);
Gy3 = zeros(m,n);
Gy4 = zeros(m,n);

Gxy1 = zeros(m,n);
Gxy2 = zeros(m,n);
Gxy3 = zeros(m,n);
Gxy4 = zeros(m,n);
```

```
    % Loop through every pixel in image and apply horizontal, vertical and
    % diagonal masks to find areas of high gradient within the image. This
    % will essentially detect all the edges in the image
    for i = 2:m-1
        for j = 2:n-1
            Gx1(i,j) = (-1*binaryImage(i-1,j) + binaryImage(i,j));
            Gx2(i,j) = (-1*binaryImage(i+1,j) + binaryImage(i,j));
            Gy3(i,j) = (-1*binaryImage(i,j-1) + binaryImage(i,j));
            Gy4(i,j) = (-1*binaryImage(i,j+1) + binaryImage(i,j));

            Gxy1(i,j) = (-1*doubleImage(i-1,j-1) + doubleImage(i,j));
            Gxy2(i,j) = (-1*doubleImage(i-1,j+1) + doubleImage(i,j));
            Gxy3(i,j) = (-1*doubleImage(i+1,j-1) + doubleImage(i,j));
            Gxy4(i,j) = (-1*doubleImage(i+1,j+1) + doubleImage(i,j));
        end
    end

    % Find sum of gradients. The absolute values are not found here so as
    % to remove all negative gradients in the next step. It essentially
    % avoids doubling the perimeter.
    edges = Gx1 + Gx2 + Gy3 + Gy4 + Gxy1 + Gxy2 + Gxy3 + Gxy4;

    % Convert edge matrix to binary
    edges = edges > 0;

    % Declare perimeter variable
    perim = 0;

    % Loop through edge matrix to count edge pixels and find total
    % perimeter
    for i = 1:m-1
        for j = 1:n-1
            if edges(i,j)
                perim = perim + 1;
            end
        end
    end

    % Find circularity
    circularity = (4*pi*m00)./(perim.^2);

end
```

Figure 34. MATLAB code for a function that detects centroids, bounding box, major axes, angle, area, perimeter, and circularity of shapes 2020

4.3 The results

Function Call:

```
clc
clear

image = imread('WhiteDiamond2019.tif');

I=image(:,:,1);

[xc,yc,xmin,xmax,ymin,ymax,a,b,theta1,theta2,area,perim,circularity] = specs(I);

imshow(I)
hold on;
rectangle('Position', [xmin-0.5 ymin-0.5 xmax-xmin+1 ymax-ymin+1], 'EdgeColor', 'r', 'LineWidth', 1)
hold on;
viscircles([xc,yc],0.5)
hold on;
line([xc xc+b*sin(theta1)],[yc yc-b*cos(theta1)])
line([xc xc+a*sin(theta2)],[yc yc-a*cos(theta2)])
```

Figure 35. MATLAB code that calls main function to compute the main task 2020

Circle:

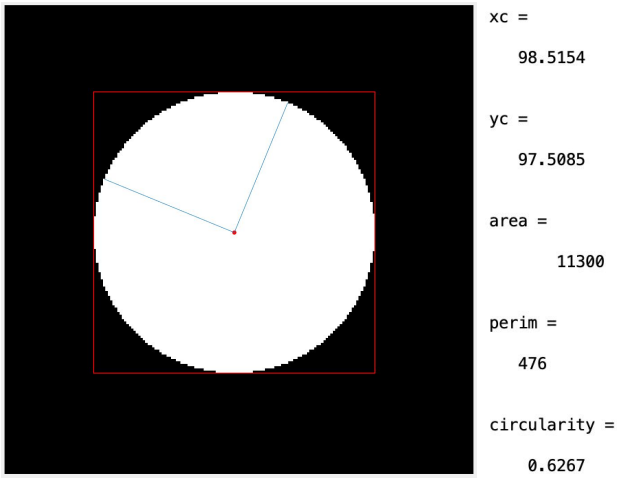


Figure 36. 'WhiteCircle2019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

Ellipse 1:

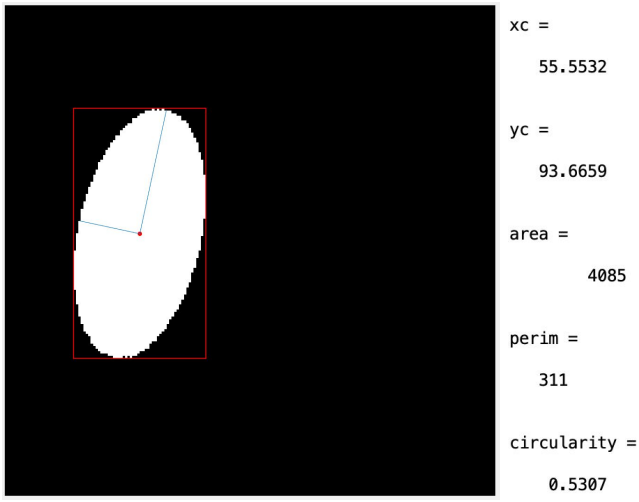


Figure 37. 'WhiteEllipse12019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

Ellipse 2:

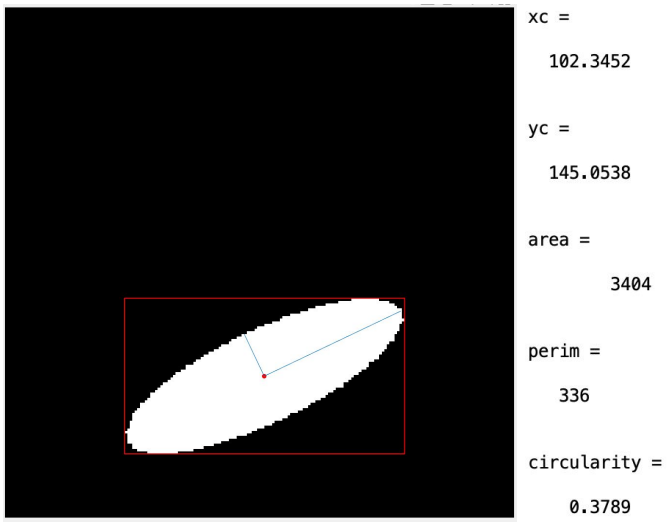


Figure 38. 'WhiteEllipse22019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

Ellipse 3:

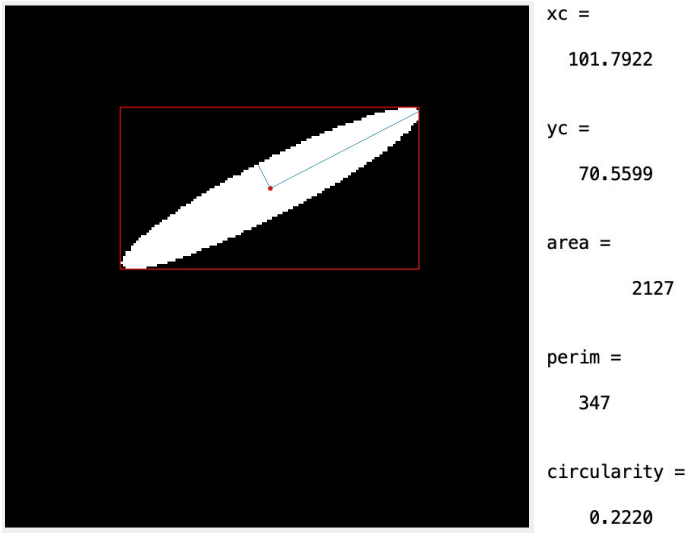


Figure 39. 'WhiteEllipse32019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

Rectangle:

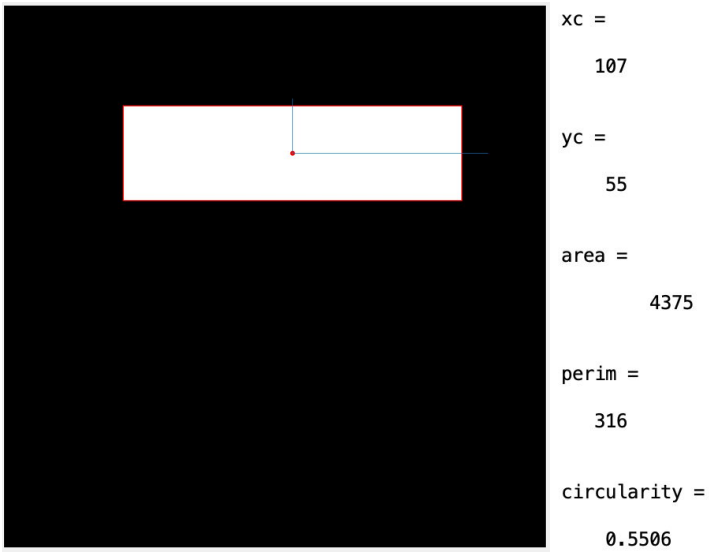


Figure 40. 'WhiteRectangle2019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

Triangle:

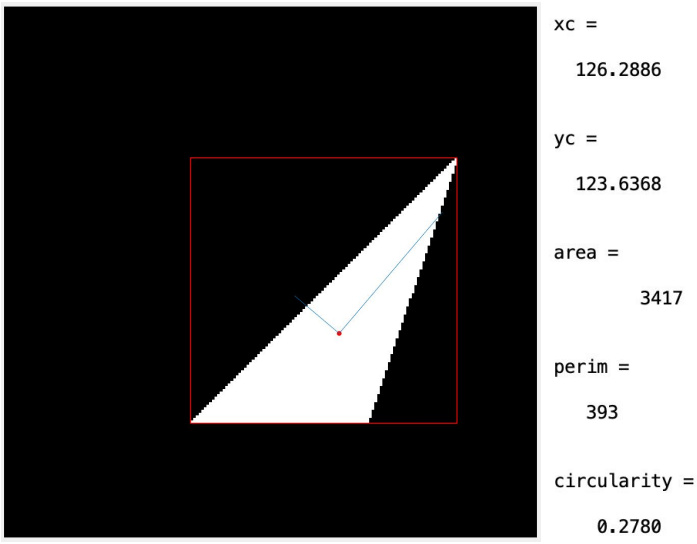


Figure 41. 'WhiteTriangle2019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

Rectangle 2:

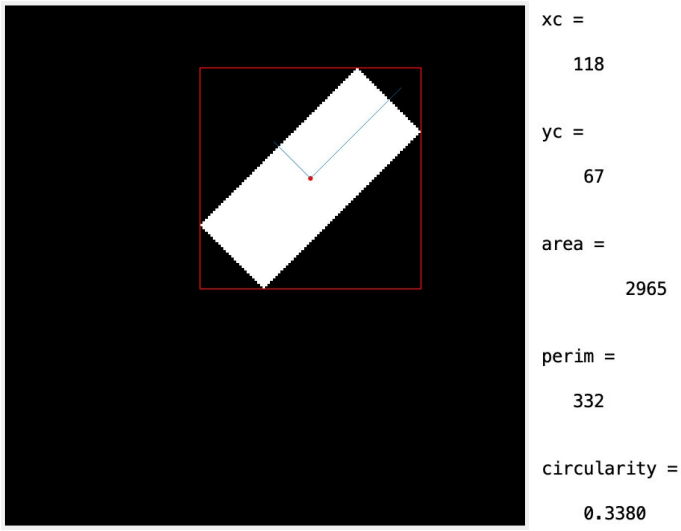


Figure 42. 'WhiteRectangle22019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

Diamond:

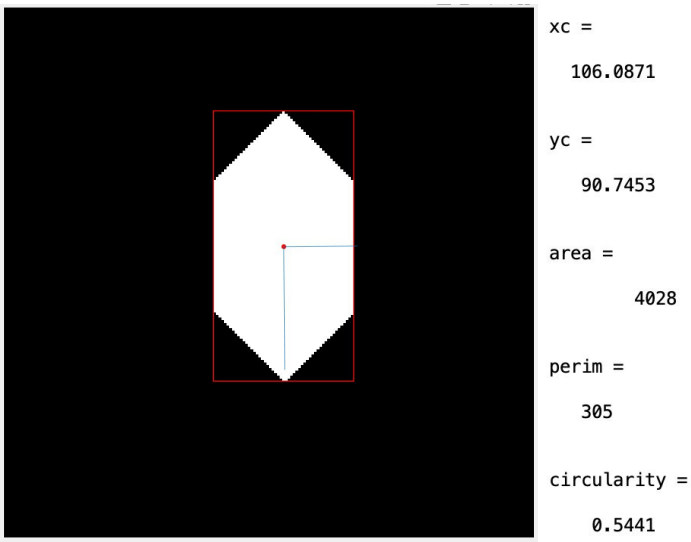


Figure 43. 'WhiteDiamond2019.tif' and its following centroids, bounding box, major axes, angle, area, perimeter, and circularity 2020

5. Problem 5

5.1 Details of the algorithm (Theory / Concept)

In Robotic Vision II lecture, it has been introduced an algorithm to convert image into grayscale and binarized it. The binarized grayscale matrix will only consist of values of 1s and 0s. Below is the input image that will be processed in this task where the desired from this task is to label and plot each shape in this image individually.

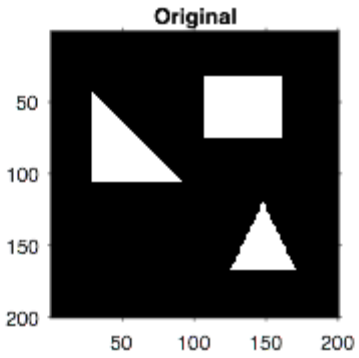


Figure 44. Input image 'MixWhite2019.tif' 2020

Now, let's consider or imagine the matrix (Ibw) of this image in smaller size shown below, since the real matrix is fairly big (200×200 matrix).

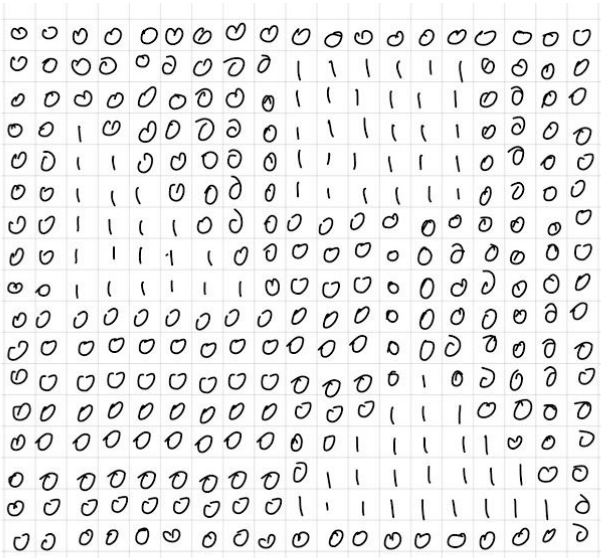


Figure 45. Example of a smaller scale matrix 2020

The goal is to find the first pixel that is equal to 1 as a foreground by checking its "red" neighbourhood, where the "red" neighbourhood will be utilized to iterate the matrix.

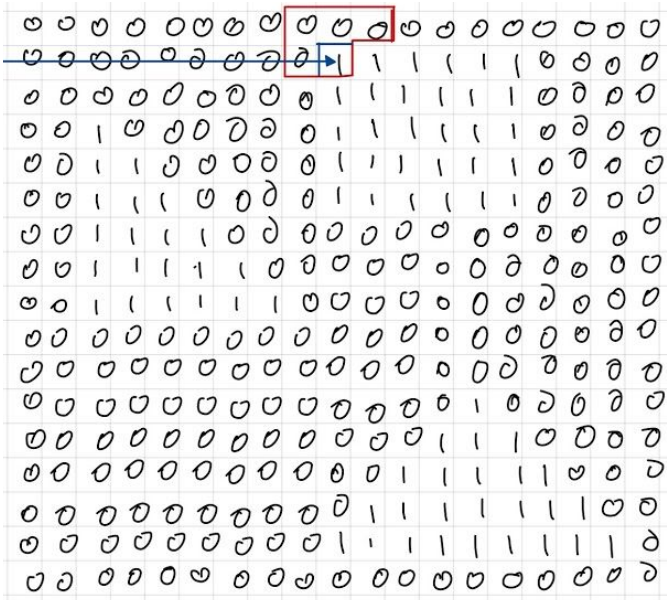


Figure 46. Search for foreground 2020

In this condition, the values inside the foreground's "red" neighbourhood are all zeros, where this condition is set to indicate the start of a new label number as shown in **Figure 47** below.

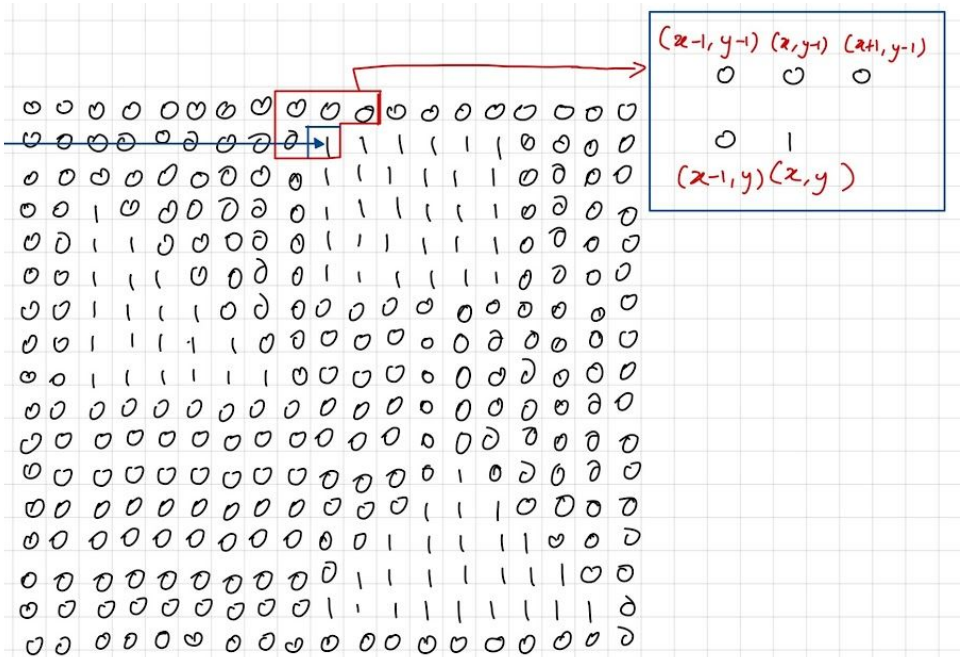


Figure 47. "Red" neighbourhood iterating the matrix 2020

From here, the “red” neighbourhood will move to the next element and check elements inside the neighbourhood, if there is a pixel that has been labelled. In this case, label number 2 is identified which indicates the connectivity of the current foreground with the previous foreground. Hence, change the value of the current foreground to the label number it connects to.

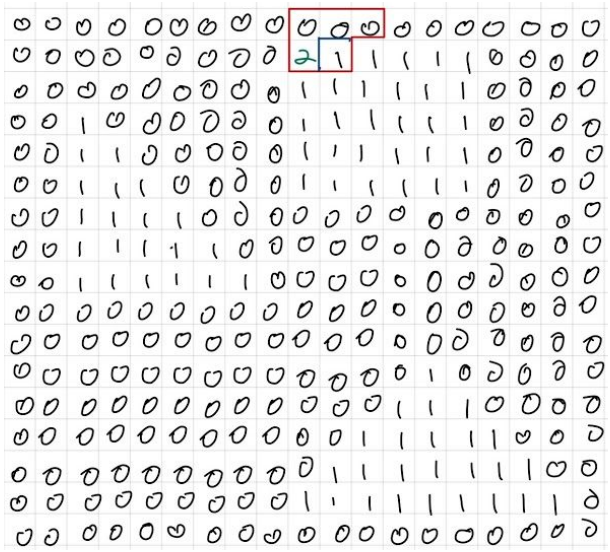


Figure 48. Analysing the next foreground 2020

The same process will be computed as shown in **Figure 49** below. However, everytime the foreground has 0 as its current value, the iteration will be skipped to the next foreground that is non-zero.

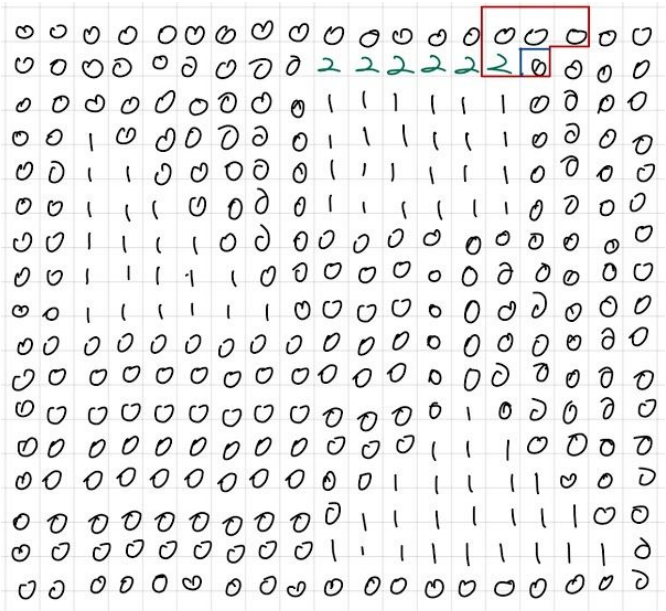


Figure 49. Skip to the next for any zero value 2020

In **Figure 50**, the foreground currently has a value of 1. Similarly, the "red" neighbourhood will check if there is connectivity within it. For the case at hand, 2 values of 2 are identified, and hence foreground is connected to label number 2.

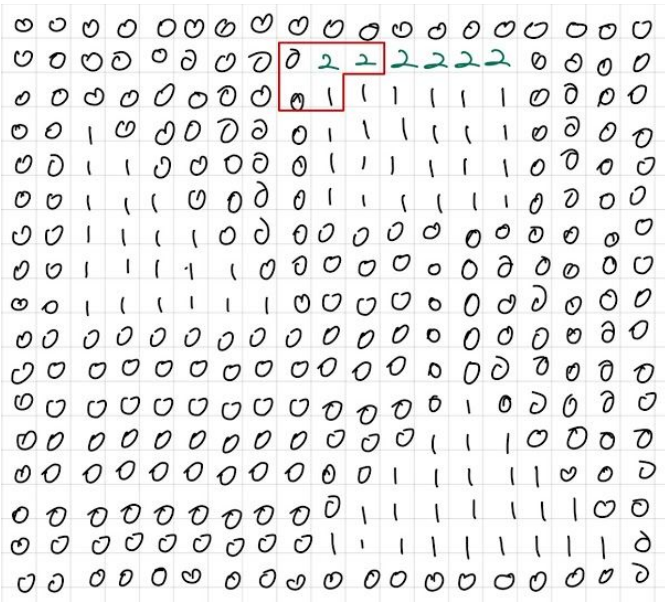


Figure 50. Connected foreground on the next row 2020

Followed by the same process, **Figure 51**, below, shows a foreground where all values within its "red" neighbourhood are all zero, which indicates the start of a new label. Since, label number 2 is taken, the new number for the label would be 3.

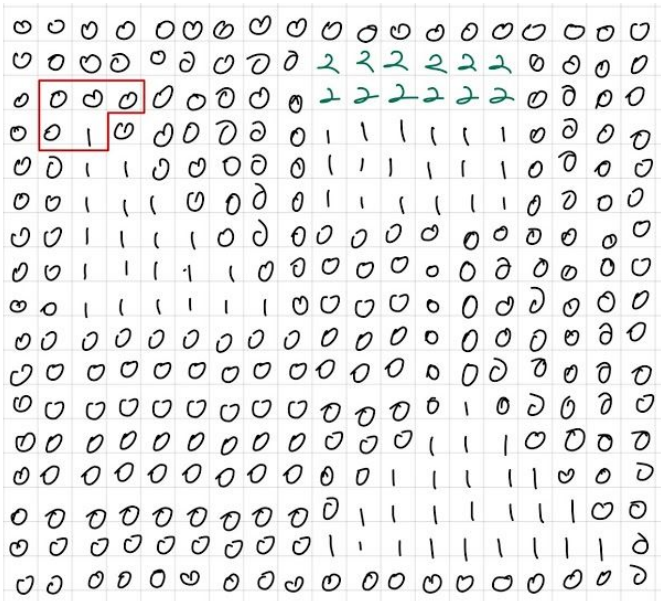


Figure 51. Foreground that meets the condition for a new label 2020

The figure below shows that despite there exist more than 1 type of label in the matrix, it will change any value of 1 to the label number it is connected to.

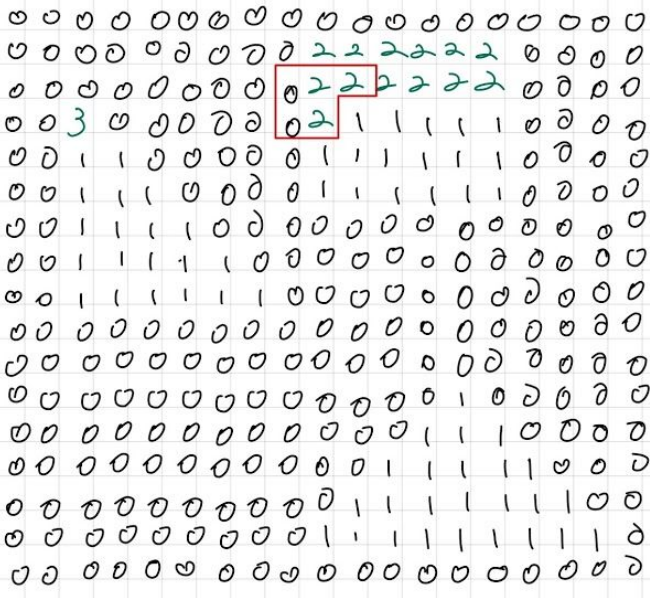


Figure 52. Change foreground to its connected label 2020

Hence, the same procedure will be computed until the iteration has reached its limit which is the size of the matrix/image. The figure below shows the expected output from the process computed above.

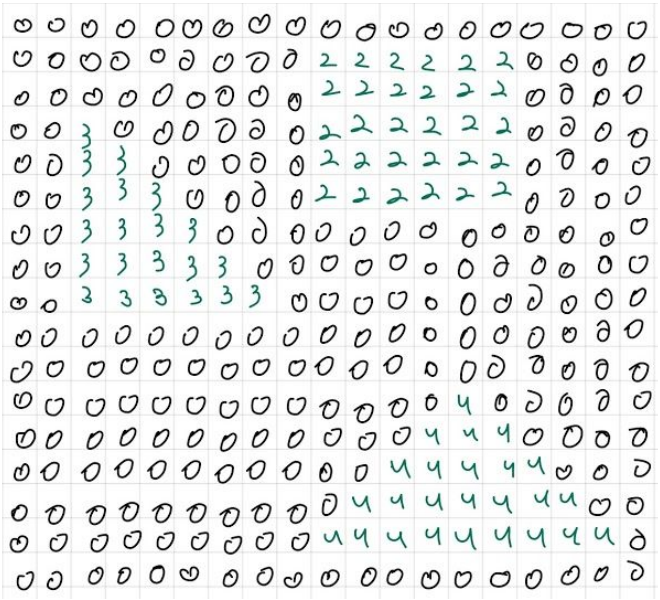


Figure 53. Final expected result from the algorithm 2020

From this point, the label number represents the type of shape/blob identified from the image, and it is desired to plot any identified shapes/blobs individually. Therefore, in this case, it is expected that there would be 3 “black & white” matrices for each individual blob.

5.2 Explanation of the MATLAB codes you wrote

```
%%%%%%%%%%%%%%%%% MAIN FUNCTION %%%%%%%%%%%%%%%%%
I = imread('WhiteMix2019.tif');

%Convert to Grayscale matrix
IRed = I(:,:,1); IGreen = I(:,:,2); IBlue = I(:,:,3);
IGrey = (double(IRed)+double(IGreen)+double(IBlue))/3;
I = uint8(IGrey);
[m,n] = size(I);

%Binarized grayscale matrix
Ibw = uint8(imbinarize(I));

%Label start from 1
label = 1;

%Initialize an empty array to store any taken labels
TAKEN_LABELS  = [];
labelIndex = 1;

%Iterate Ibw matrix
for i = 1:m
    for j = 1:n
        if(Ibw(i,j) == 1)

            %Checking if all  elements surround the
            %foreground are zero.
            if(Ibw(i, j-1) == 0 && Ibw(i-1, j-1) == 0 &&...
                    Ibw(i-1, j) == 0 && Ibw(i-1, j+1) == 0)
                %Start new label number
                label = label + 1;
                Ibw(i,j) = label;

                %Put label in the array that contains any taken label
                TAKEN_LABELS(labelIndex) = label;   %#ok<SAGROW>
                labelIndex = labelIndex + 1;
            end

            %Check for label that is connected to the current
            %foreground
            if(isConnected(TAKEN_LABELS, Ibw(i, j-1)) || ...
                    isConnected(TAKEN_LABELS, Ibw(i-1, j-1))||...
                    isConnected(TAKEN_LABELS, Ibw(i-1, j))||...
                    isConnected(TAKEN_LABELS, Ibw(i-1, j+1)))
```

```
                %Change value of 1 into the specific
                %label it connected to
                Ibw(i,j) = findConnectedLabel(TAKEN_LABELS, ...
                    Ibw(i, j-1), Ibw(i-1, j-1), ...
                    Ibw(i-1, j), Ibw(i-1, j+1));
            end
        end

        %Skip and move to the next iteration when foreground value
        %is zero.
        if(Ibw(i,j) == 0)
            continue;
        end

    end
end

%Call function to show all shapes identified within the image
%and individually labelled. This function will show more or less
%image depending on how many shapes identified within the analysed
%image
plotIndividualBlob(Ibw, TAKEN_LABELS);


%%%% COMBINE IMAGES IN A SINGLE PLOT (FOR THE MAIN CASE STUDY) %%%%
%Matrix for individual blob
Ibw2 = fillBlobMatrix(Ibw, 2);
Ibw3 = fillBlobMatrix(Ibw, 3);
Ibw4 = fillBlobMatrix(Ibw, 4);

%Combined all processed images in a single plot
figure;
subplot(2,2,1), imshow(I), axis on;
title('Original');

subplot(2,2,2), imshow(Ibw2), axis on;
title('Blob Number 2');

subplot(2,2,3), imshow(Ibw3), axis on;
title('Blob Number 3');

subplot(2,2,4), imshow(Ibw4), axis on;
```

```
    title('Blob Number 4');


    %%%%%%% ADDITIONAL FUNCTION CREATED FOR THE TASK %%%%%%%

    %Function to fill matrix of a blob
function blobMatrix = fillBlobMatrix(binaryMatrix, label)
    [m,n] = size(binaryMatrix);
    for i = 1:m
        for j = 1:n
            if(binaryMatrix(i,j) ~= label)
                binaryMatrix(i,j) = 0;
            else
                binaryMatrix(i,j) = 255;
            end
        end
    end
    blobMatrix = binaryMatrix;
end

%A Function that return boolean value if there exist a connectivity
function connectionStatus = isConnected(LABELS, currentNumber)
    l = length(LABELS);
    connectionStatus = 0;
    for i = 1:l
        if(currentNumber == LABELS(i))
            connectionStatus = 1;
            break;
        end
    end
end

%A Function that return the number of label the foreground connected
%to.
function connectedLabel = findConnectedLabel(LABELS, A, B, C, D)
    l = length(LABELS);
    for i = 1:l
        if(A == LABELS(i) || B == LABELS(i) || C == LABELS(i) || ...
                D == LABELS(i))
            connectedLabel = LABELS(i);
            break;
        end
    end
```

```
130 -   end
131
132
133     %Function to plot individual blob
134     function plotIndividualBlob(binarizedMatrix, LABELS)
135 -       l = length(LABELS);
136 -       for i = 1:l
137 -           figure, imshow(fillBlobMatrix(binarizedMatrix, ...
138 -               LABELS(i))), axis on;
139 -           title(['Blob Number ' num2str(LABELS(i)) '']);
140 -       end
141 -   end
142
143
```

Figure 54. MATLAB code for Problem 5 2020

Line 17:

Store label number that has been taken, where the size of the array will change depending on how many new labels are being added to the TAKEN_LABELS variable.

Line 27 → 60:

These lines of code are computing the above algorithm. For example, when foreground is non-zero it has coordinate position (x, y). Then, by utilizing the "red" neighbourhood algorithm, it will check the value of $Ibw(y, x-1)$, $Ibw(y-1, x-1)$, $Ibw(y-1, x)$, & $Ibw(y-1, x+1)$. If the value at $Ibw(y, x-1)$, $Ibw(y-1, x-1)$, $Ibw(y-1, x)$, & $Ibw(y-1, x+1)$ are all 0s, then start a new label. If not all 0, then the program will check by which label the foreground is being connected to.

Line 93 → 105:

A function that will produce a matrix of each identified blob. For example, the first shape is labelled as 2, and hence "Blob Number 2" will have a matrix that has been processed where any value that is not equal to the label number will be changed into 0. Otherwise, 255, which is the brightest color value (White). Then, the same computation will be executed for the rest of the blobs.

Line 108 → 117:

A function that will return a boolean value. If there is a connectivity, it will return 1 (true). Otherwise, 0 (false).

Line 121 → 129:

A function processed $Ibw(y,\ x-1)$, $Ibw(y-1,\ x-1)$, $Ibw(y-1,\ x)$, & $Ibw(y-1,\ x+1)$. If $Ibw(y,\ x-1)$ *or* $Ibw(y-1,\ x-1)$ *or* $Ibw(y-1,\ x)$ *or* $Ibw(y-1,\ x+1)$ is equal to one of the labels in TAKEN_LABELS, it will return the number of that specific label. Hence, the function will change the value of the foreground by which label it is connected to.

Line 134 → 141:

A void function that will show the plot of an individual blob with its label number.

Line 75 → 87:

These lines of code are specifically made for the main case study to show comparison of the original image with the separated blob.

5.3 The results:

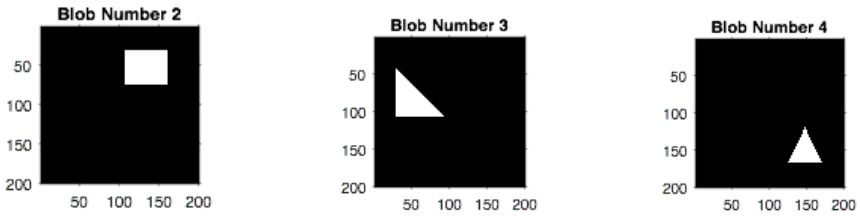


Figure 55. Plotted and labelled individual blob 2020

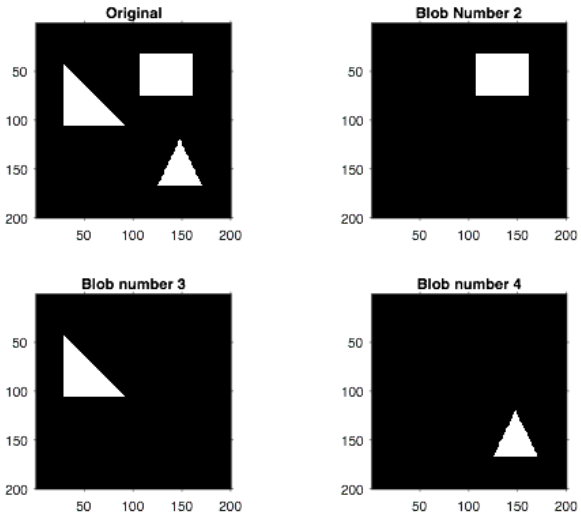


Figure 56. Comparison with the original image 2020

By the result above, it is shown that the algorithm is sufficient to solve this problem. However, the algorithm would be sufficient for basic shapes such as square, circle, triangle etc. The algorithm needs to be improved so that it is sufficient enough to identify and extract any inorganic shapes/blobs.

6. Problem 6

6.1 Details of the algorithm (Theory / Concept)

This problem involved expanding on the code from the previous problem in order to label more complex connected components. The image containing the blobs used for this problem was the following image:

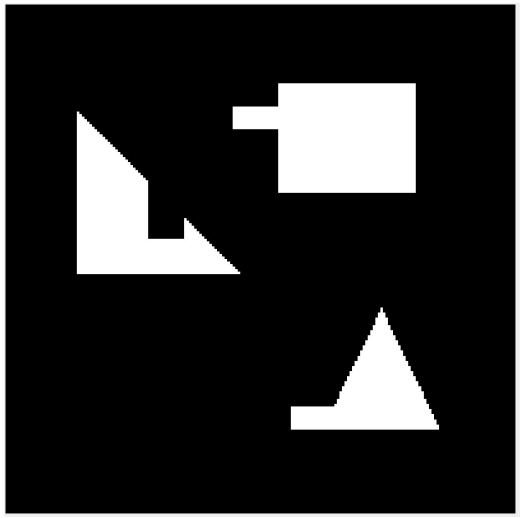


Figure 57. 'WhiteMixComplex2019.tif' image 2020

If we were to use the algorithm from problem 5 to separate the blobs in this image, we would get the following separate components:

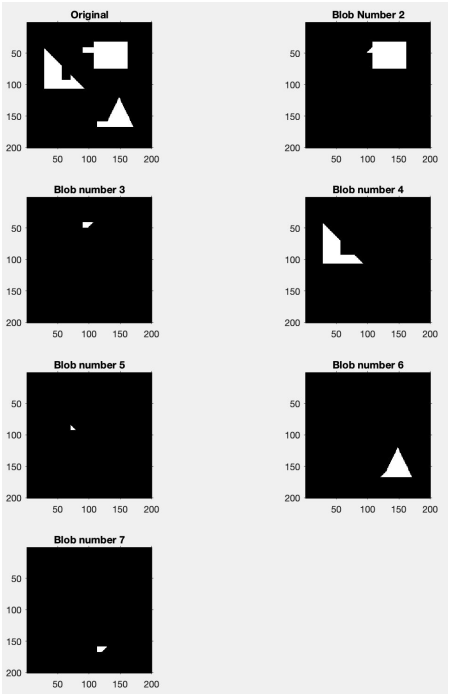
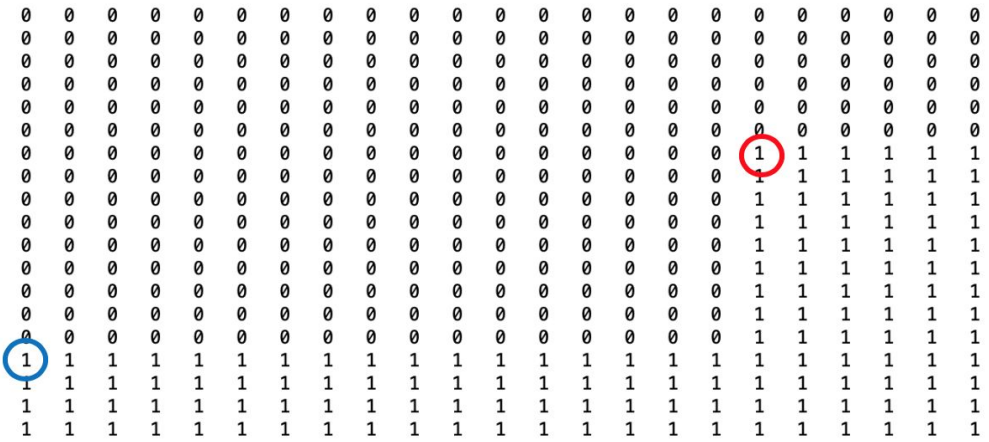


Figure 58. 'WhiteMixComplex2019.tif' by utilising blob detection algorithm from Problem 5 2020

From this, we can see that each of the three complex blobs have been separated into two components each, leading to six output images. This is due to the order of iteration through the image matrix. As the iteration is going row by row from top to bottom and left to right, at certain pixel values, the algorithm is picking up a new blob, when it is actually part of an already detected blob. For example, if we look closely at the blob on the top right of the original image, we can see the following pixel values:



The algorithm will first reach the pixel circled in red, declare it a new blob and label it as 2. From here it will continue moving through row by row from left to right, labelling each 1 value

as 2. Once it gets to the pixel labelled in blue, it will detect it as another new blob, as all the pixels surrounding its top left are 0. It will start labelling all these pixels as 3 from left to right. This results in a matrix that looks like this:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2

From here, all the pixels labelled 3 will be separated from the pixels labelled 2.

To fix this, all the pixels that have been detected as a blob, but have a label different from its surrounding pixels, need to be changed. For example, if the current pixel has a value of 3, but is directly connected to a value of 2, the current pixel needs to be changed to 2. As the initial iteration moves through the matrix from top left to bottom right, the second iteration will need to move from bottom right to top left to fix all the incorrectly labelled pixels.

For example, if we look closely at the bottom of the triangle shaped blob after the first algorithm has been applied, we see:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	6	6	6
0	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	6	6	6	6	6
0	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6
0	7	7	7	7	7	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6
0	7	7	7	7	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	6
0	7	7	7	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	6	6
0	7	7	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	6	6	6
0	7	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	6	6	6	6
0	7	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	6	6	6	6	6
0	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	6	6	6	6	6	6
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The second algorithm will get to the pixel marked in red, and see that it is connected to a pixel to the left and above that is nonzero and has a different label to itself. This will detect that the pixel above and to the left has been labelled incorrectly, and change those labels to its current value of 6. As we are moving from bottom right to top left, we only need to check the pixel directly above and directly to the left. Once this second iteration has moved through the entire matrix, all the pixels will be labelled correctly.

6.2 Explanation of the MATLAB codes you wrote

```
I = imread('WhiteMixComplex2019.tif');

%Convert to Grayscale matrix
IRed = I(:,:,1); IGreen = I(:,:,2); IBlue = I(:,:,3);
IGrey = (double(IRed)+double(IGreen)+double(IBlue))/3;
I = uint8(IGrey);
[m,n] = size(I);

%Binarized grayscale matrix
Ibw = uint8(imbinarize(I));

%Label start from 1
label = 1;

%Initialize an empty array to store any taken labels
TAKEN_LABELS  = [];
labelIndex = 1;

%Iterate Ibw matrix
for i = 1:m
    for j = 1:n
        if(Ibw(i,j) == 1)

            %Checking if all  elements surround the
            %foreground are zero.
            if(Ibw(i, j-1) == 0 && Ibw(i-1, j-1) == 0 &&...
                    Ibw(i-1, j) == 0 && Ibw(i-1, j+1) == 0)
                %Start new label number
                label = label + 1;
                Ibw(i,j) = label;

                %Put label in the array that contains any taken label
                TAKEN_LABELS(labelIndex) = label;   %#ok<SAGROW>
                labelIndex = labelIndex + 1;
            end

            %Check for label that is connected to the current foreground
            if(isConnected(TAKEN_LABELS, Ibw(i, j-1)) || ...
                    isConnected(TAKEN_LABELS, Ibw(i-1, j-1))||...
                    isConnected(TAKEN_LABELS, Ibw(i-1, j))||...
                    isConnected(TAKEN_LABELS, Ibw(i-1, j+1)))

                %Change value of 1 into the specific
                %label it connected to
                Ibw(i,j) = findConnectedLabel(TAKEN_LABELS, ...
                    Ibw(i, j-1), Ibw(i-1, j-1), ...
                    Ibw(i-1, j), Ibw(i-1, j+1));
            end
        end

        %Skip and move to the next iteration when foreground value is zero.
        if(Ibw(i,j) == 0)
            continue;
        end

    end
end

% Initialize an empty array to store any unused labels
UNUSED_LABELS  = [];
unusedLabelIndex = 1;

% Iterate through new Ibw matrix from opposite direction
for i = m:-1:1
    for j = n:-1:1
        % If pixel is part of blob
        if(Ibw(i,j) > 1)
            % If the pixel to the left or above is not zero and not the
            % same as the current pixel, then set it to the current pixel
            if(Ibw(i, j-1) ~= 0 && Ibw(i, j-1) ~= Ibw(i,j))
                Ibw(i,j-1) = Ibw(i,j);
                % Store unused label for later
                if ~any(UNUSED_LABELS == Ibw(i,j-1))
                    UNUSED_LABELS(unusedLabelIndex) = Ibw(i,j-1);
                    unusedLabelIndex = unusedLabelIndex + 1;
                end
            elseif (Ibw(i-1, j) ~= 0 && Ibw(i-1, j) ~= Ibw(i,j))
                Ibw(i-1, j) = Ibw(i,j);
                if ~any(UNUSED_LABELS == Ibw(i-1,j))
                    UNUSED_LABELS(unusedLabelIndex) = Ibw(i-1,j);
                    unusedLabelIndex = unusedLabelIndex + 1;
                end
            end
        end
    end
end

```

```
88      % Initialise output images
89 -    OUTPUTS = zeros(m,n,length(TAKEN_LABELS));
90 -    j = 1;
91
92      %create matrices for each blob
93 -    for i = 1:length(TAKEN_LABELS)
94 -        if ~any(UNUSED_LABELS == i)
95 -            OUTPUTS(:,:,j) = fillBlobMatrix(Ibw, TAKEN_LABELS(i));
96 -            j = j + 1;
97 -        end
98 -    end
99
100     % Plot images
101 -    figure;
102 -    subplot(j,1,1), imshow(I);
103 -    title('Original');
104 -    for i = 1:j-1
105 -        subplot(j,1,i+1), imshow(OUTPUTS(:,:,i));
106 -        title(['Blob Number ' num2str(i)]);
107 -    end
108
109     %Function to fill matrix of a blob
110 -    function blobMatrix = fillBlobMatrix(binaryMatrix, label)
111 -        [m,n] = size(binaryMatrix);
112 -        for i = 1:m
113 -            for j = 1:n
114 -                if(binaryMatrix(i,j) ~= label)
115 -                    binaryMatrix(i,j) = 0;
116 -                else
117 -                    binaryMatrix(i,j) = 255;
118 -                end
119 -            end
120 -        end
121 -        blobMatrix = binaryMatrix;
122 -    end
123
124     %A Function that return boolean value if there exist a connectivity
125 -    function connectionStatus = isConnected(LABELS, currentNumber)
126 -        l = length(LABELS);
127 -        connectionStatus = 0;
128 -        for i = 1:l
129 -            if(currentNumber == LABELS(i))
130 -                connectionStatus = 1;
131 -                break;
132 -            end
133 -        end
134 -    end
135
136     %A Function that return the number of label the foreground connected to.
137 -    function connectedLabel = findConnectedLabel(LABELS, A, B, C, D)
138 -        l = length(LABELS);
139 -        for i = 1:l
140 -            if(A == LABELS(i) || B == LABELS(i) || C == LABELS(i) || ...
141 -                    D == LABELS(i))
142 -                connectedLabel = LABELS(i);
143 -                break;
144 -            end
145 -        end
146 -    end
```

Figure 59. Problem 5 MATLAB code with an extensions that detect irregular shapes 2020

The above code from line 1 to line to line 57 is the same as in problem 5, and the functions from line 109 to line 146 is also the same.

Line 88 to line 107 is used to make the code more robust. This code allows for an input image where the number of blobs is less than or more than 3. The output will show only the number of blobs that exist in the original image, whether it be three, as in this case, or any other number.

6.3 The results

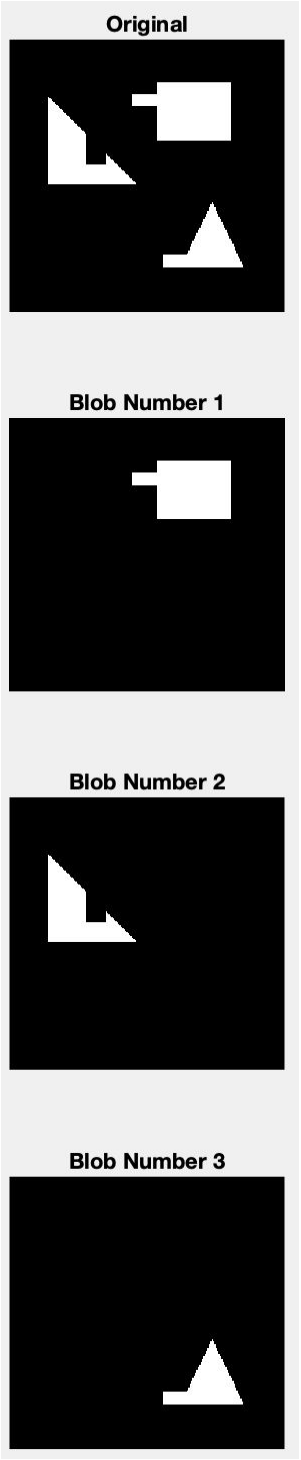


Figure 59. Problem 6 final results 2020