Tan Jo-Wayn EE4704 CA2

A2. intermeans.m

```
5
6   function [T,Iout] = intermeans(Iin)
7       %Iin = imread(Iin);
8       T_prev = 0; %initialise prev T
9       T = mean(mean(Iin)); %obtain threshold estimate
10      while T_prev ~= T %loop until T_prev == T
11          T_prev = T;
12          u_1 = mean(mean(Iin(Iin<T_prev))); %mean of image for image <threshold
13          u_2 = mean(mean(Iin(Iin>T_prev))); %mean of image for image >threshold
14          T = ceil((u_1+u_2)/2);
15          %disp(T);
16      end
17      Iout = Iin >= T; %obtain binary matrix
18      %imwrite(Iout,"test2_threshold.bmp");
19      %subplot(1,2,1),imshow(Iin),title('Original Image');
20      %subplot(1,2,2),imshow(Iout),title('Image After Thresholding');
21  end
22
```

Fig 1. intermeans function

1. In my implementation of the intermeans algorithm as per Fig. 1, I first took an initial estimate of the threshold as the average intensity of the image (Line 9). mean() function was called twice as we are operating on a matrix of intensity values when the input is an image.
2. We then enter a loop (Line 10) whereby the terminating condition is when T_prev = T. At the start of the loop, T_prev is assigned the value of T (Line 11). This is to differentiate between the new threshold we are about to calculate and the previous threshold value, T_prev.
3. We split the image into 2 groups and find the mean intensity of the image within each group. The first group selects only the pixels of the image where its intensities are less than T_prev, and the second group selects the pixels of the image where its intensities are greater than T_prev. The respective means of the image intensities of these 2 groups are saved into u_1 and u_2 respectively (Lines 12-13).
4. The new threshold calculated, T, is then assigned to be the average of u_1 and u_2 (Line 14). We also round T up to the nearest integer through the use of *ceil()* so as to avoid running the loop too many times (i.e new threshold and previous threshold don't have to match exactly for the loop to terminate, they just have to round up to the same integer).

```
Command Window
>> intermeans("test1.bmp")
    87

    99

   112

   114

   114


ans =

   114
```

Fig 2. Displaying T after each loop

5. Optionally displaying the values of the threshold calculated (Line 15) yields us Figure 2. We observe that once the threshold value of 114 was printed twice, the algorithm terminates. This shows us that our terminating condition is working as expected because once the new threshold value, T was equivalent to the threshold value calculated in the previous loop, T_prev, we exit the while loop.
6. Lastly, we assign Iout = Iin >= T (Line 17), which thresholds our original image by assigning value of 1 to image pixels above or equal to threshold T, and 0 to image pixels below the threshold T. We observe the correct behaviour of this through plotting the image histogram of Iout (Fig. 4).

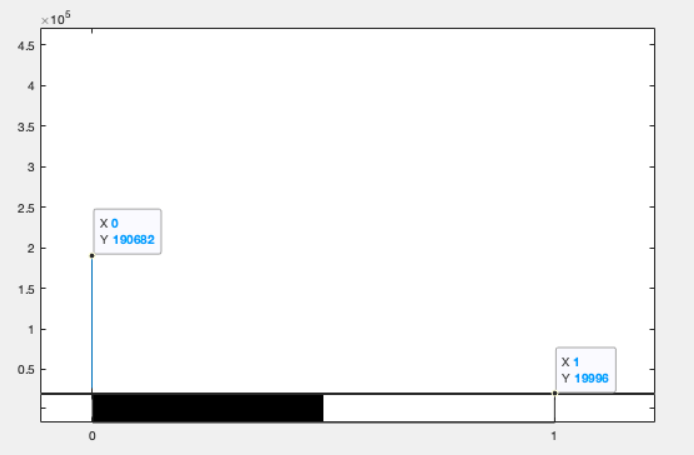


Fig 4. Histogram of test1.bmp after thresholding

7. Plotting the Original Image versus the Image After Thresholding (Fig. 5), we see that we have successfully segmented the image between the object and the background.

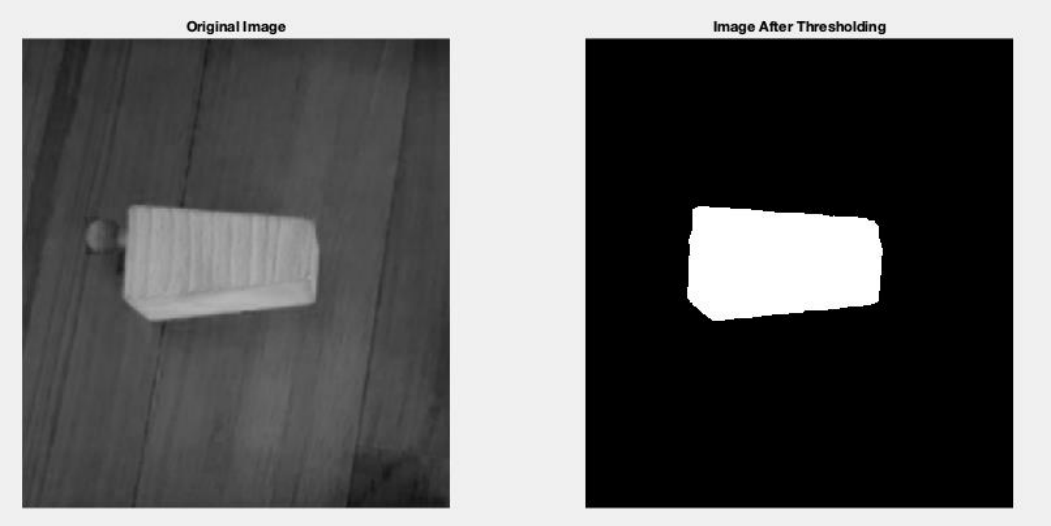


Fig. 5. Original Image vs Image After Thresholding

A3. features.m

```
9       %perimeter
10      P = regionprops(Iin, 'Perimeter').Perimeter;
11      disp("Perimeter: " + P);
12
13      %area - find by filling perimeter, then find sum of pixels
14      boundary = bwperim(Iin,8);
15      I_filled = imfill(boundary,'holes');
16      A = sum(sum(I_filled));
17      disp("Area: " + A);
18
19      %compactness - formula as per lecture notes
20      C = (P^2)/(4*pi*A);
21      disp("Compactness: " + C);
```

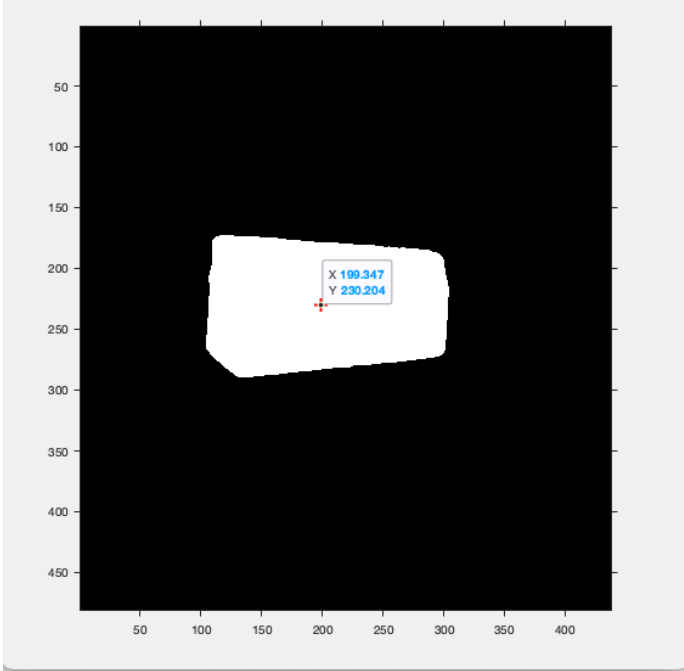
Fig. 6. Perimeter, Area and Compactness of features.m

1. In order to find the object perimeter, I made use of the regionprops() "Perimeter" property, which takes in a binary image and calculates the distance between every adjoining pair of pixels around the border of the object.
2. To calculate Area, I first retrieved only the perimeter pixels of the object in the binary input image through the use of bwperim()function. Then, I filled the perimeter using imfill() and summed up the number of pixels used to filled the perimeter.
3. For compactness, I used the compactness formula, $\gamma = \frac{(perimeter)^2}{4\pi * area}$

```
23      %centroid
24      %m00: no. of pixels in component
25      %size(matrix,2) finds its no. of columns
26      m00=0;
27      m10=0;
28      m01=0;
29      t_cols = size(Iin,2); %total columns
30      t_rows = size(Iin,1); %total rows
31
32      for x = 0:t_cols-1 %iterating across columns
33          for y = 0:t_rows-1 %iterating across rows
34              m00 = m00 + Iin(t_rows-y, x+1);
35              m10 = m10 + (x * Iin(t_rows-y, x+1));
36              m01 = m01 + (y * Iin(t_rows-y, x+1));
37          end
38      end
39
40      xbar = m10/m00;
41      ybar_actual = m01/m00;
42      ybar = t_rows - m01/m00;
43      imshow(Iin);
44      axis on
45      hold on;
46      plot(xbar,ybar, 'r+', 'MarkerSize', 10, 'LineWidth', 2); %plot cross
```

Fig. 7. Centroid of features.m

4. For centroid, I first obtained the values of the number of columns and rows through size() function, and specifying the dimension as the second argument (Lines 29-30). Then, I iterated across each pixel through the use of nested loops, summing up the pixel intensities across each pixel to obtain m00, and summing x*f(x,y) for m10 and y*f(x,y) across each pixel to obtain m10 (first order moment about x) and m01 (first order moment about y) respectively. (where f(x,y) represents the pixel intensity at x,y)
5. Note: (t_rows-y) was used instead of simply using y because we are calculating based on the coordinate system with origin at the bottom left, as opposed to the image-coordinate system of Matlab, with origin at the top left. Also, this leads to a separate calculation of ybar and ybar_actual, whereby ybar_actual is purposed for the use of bottom left as origin and ybar is purposed for the use of top left origin. ybar bottom is returned as output of the function with reference to Matlab's image coordinate system.
6. Additionally, a '+' was plotted as part of the function to demarcate the centroid location on the image. This can be observed in Fig. 8.




```
>> features("test1_threshold.bmp")
Perimeter: 565.46
Area: 19996
Compactness: 1.2725
xbar: 199.3465
ybar: 230.204
phione: 0.19833
```

Fig. 8. features.m on test1.bmp

```
50      %invariant moment
51      % m20 = sumof x^2 * f(x,y)
52      % m02 = sumof y^2 * f(x,y)
53      % u20 = m20 - xbar(m10)
54      % u02 = m02 - ybar(m01)
55      % fi_1 = u20/(u00)^2 + u02/(u00)^2
56
57      u00 = m00;
58      m20 = 0;
59      m02 = 0;
60      for x = 0:t_cols-1
61          for y = 0:t_rows-1
62              m20 = m20 + (x^2 * I_filled(t_rows-y, x+1));
63              m02 = m02 + (y^2 * I_filled(t_rows-y, x+1));
64          end
65      end
66      u20 = m20 - xbar*m10;
67      u02 = m02 - ybar_actual*m01;
68      phione = u20/(u00)^2 + u02/(u00)^2;
```

Fig. 9. Invariant moment of features.m

7. Similarly to centroid, invariant moment was calculated by initializing a nested loop to iterate through every pixel. This time, the sum of x^2*f(x,y) across all x,y was calculated to obtain m20, the second order moment about x, and the sum of y^2*f(x,y) across all x,y was calculated to obtain m02, the second order moment about y. This in turn can be used to calculate u20 and u02 (Lines 66-67), the second order central moments about x, y, through the use of the equation in Fig. 10. Further making use of the equation in Fig. 11, we find that we can obtain the normalised central moments through the second order central moments, and then as per the equation in Fig. 12,

obtain the first invariant central moment through the normalised central moments (Line 68).

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}$$

$$\begin{aligned} \mu_{20} &= m_{20} - \bar{x}m_{10} \\ \mu_{02} &= m_{02} - \bar{y}m_{01} \end{aligned} \qquad \gamma = \frac{p+q}{2} + 1 \qquad \text{for } p+q = 2, 3, \ldots$$

(left) Fig. 10. 2nd Order Central Moment of about x, y
(right) Fig. 11. Normalised Central Moments

$$\phi_1 = \eta_{20} + \eta_{02}$$

Fig. 12. Invariant Central Moment

8. To further test my features.m code, I created input images as per Fig. 13 and 14 as extra test cases for my program.

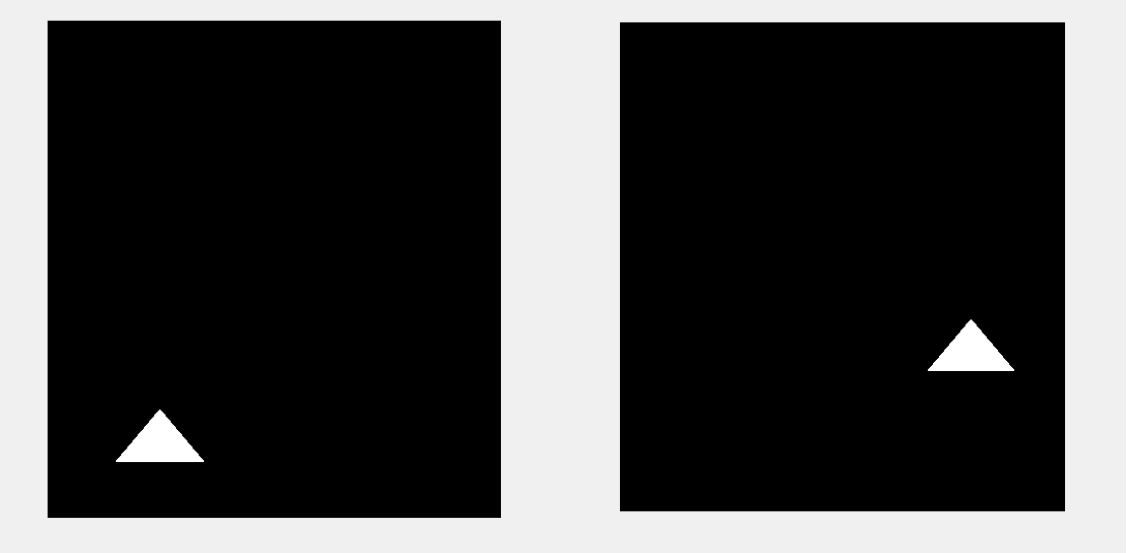


Fig. 13 (left), Fig. 14 (right) Extra test cases for features.m

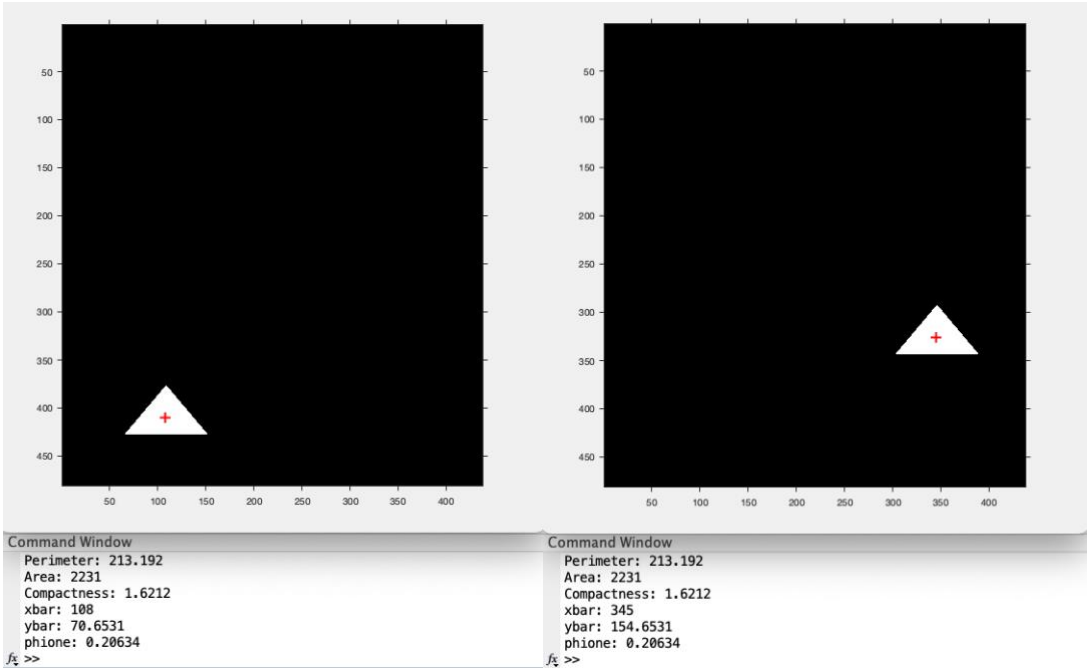


Fig. 15 (left), Fig. 16 (right) Output of features.m

9. Most notably, we find that the value of ϕ_1 remains constant for both images even though the triangle has undergone translation in both the x and y axes, which is expected because ϕ_1 is invariant to translation, scaling and rotation.

B2.

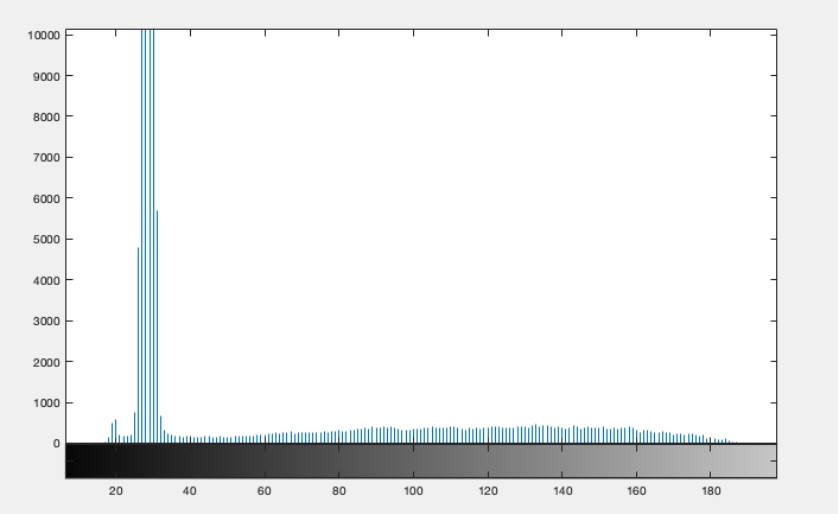
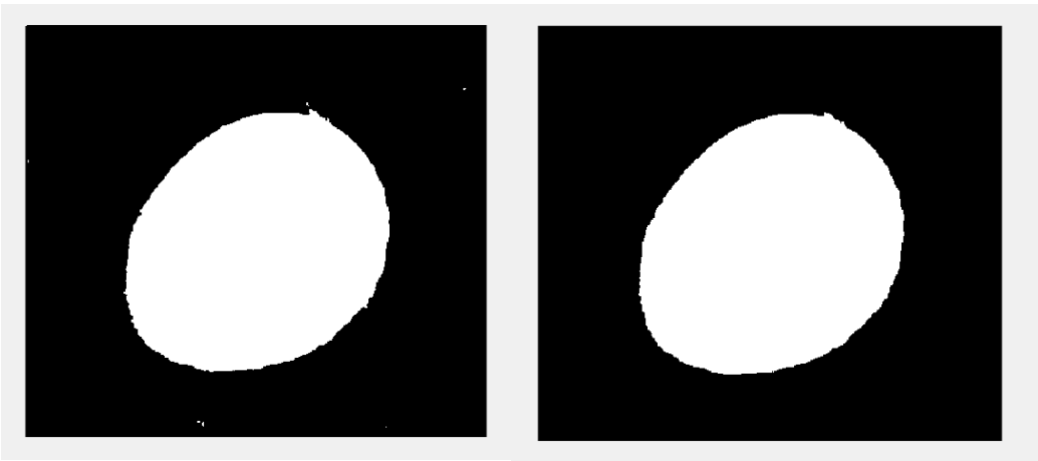


Fig. 17. Histogram plot of test2.bmp

Plotting the image histogram of test2.bmp, we obtain Fig. 17. As observed from the figure, there is a high density of pixels in the range X = [25,32], and these pixels form the darker regions (background) of the image. As such, we will try to set the threshold to be after this pixel range, setting the region of high intensities as the background. Setting the threshold as 33 yields us some salt noise in the image (Fig. 18), while setting the threshold to 34 successfully removes this salt noise (Fig. 19). We will therefore set T_{opt} = 34.



(left) Fig. 18. T = 33
(right) Fig. 19. T = 34

B3.

By intermeans.m program, $T_2 = 80$. Output image is as Fig. 20.

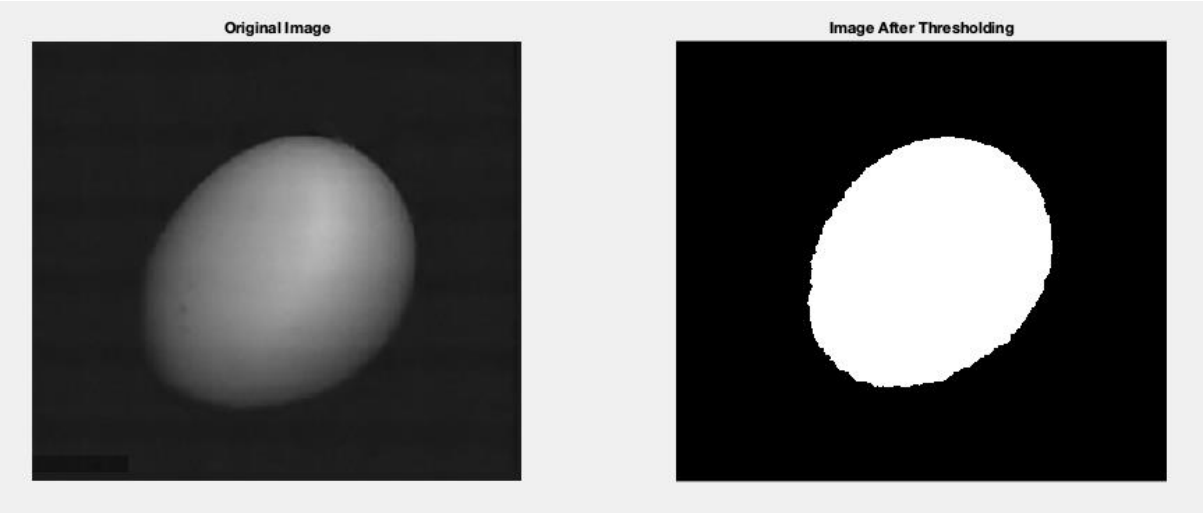


Fig. 20. Result of intermeans thresholding algorithm on test2.bmp

B4.

Thresholding with $T_2 = 80$, we obtain the values of features.m as shown in Fig. 21.

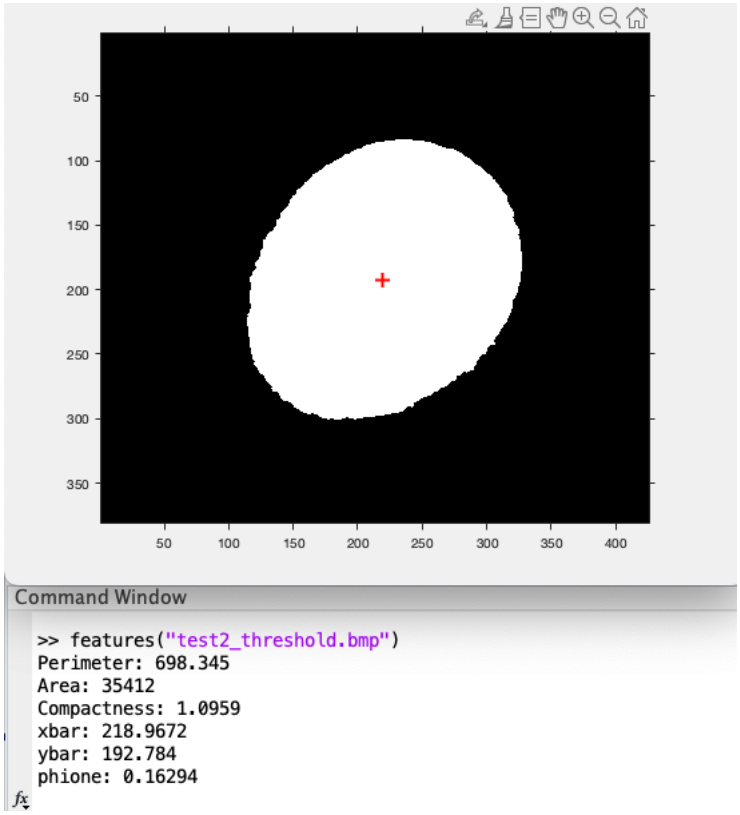


Fig. 21. Measuring features of test2.bmp with $T_2 = 80$

B5.

Thresholding with T_{opt} = 34, we obtain the values of features.m as shown in Fig. 22.

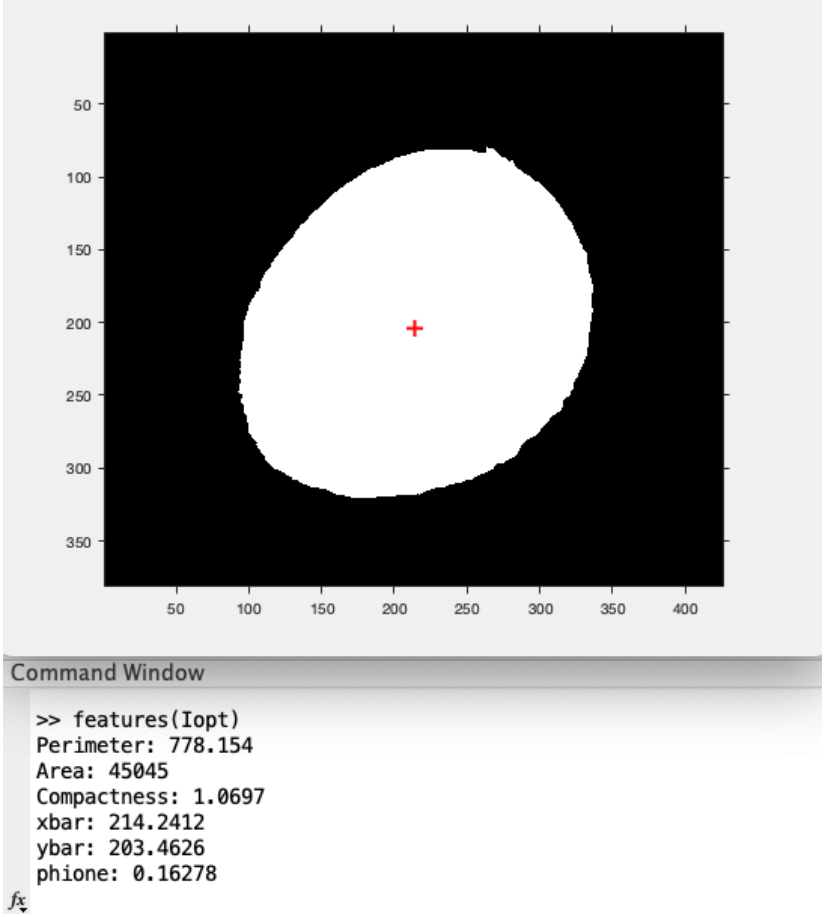


Fig. 22. Measuring features of test2.bmp with Topt = 34

B6.

Comparing the image features where T_2 = 80 and T_{opt}= 34, we find that as we found an optimal threshold value that occurred directly after a region on the histogram (Fig. 17) where the pixel density was especially high, we essentially applied a more effective threshold and managed to remove the background more precisely, obtaining a foreground that encapsulated more of the object. This is apparent in the increase from 698.3 to 778.2 in the perimeter with the change in threshold from 80 to 34, as well as an increase in the area of the object from 35412 to 45045. Also, the compactness fell from 1.0959 to 1.0697 with the change in threshold as well, and this could be attributed to the foreground image appearing rounder and less elongated in Fig. 22 as compared to Fig. 21, thus there is a slight drop in compactness. Most notably, the value of ϕ_1 remained relatively constant throughout the change in threshold. This is because the invariant central moment is invariant to all of translation, **scale** and rotation. Between applying T_2 = 80 and T_{opt}= 34, we can view the key difference as a difference in scale of the object, with the image with T_{opt}= 34 being a larger scale version of the image with T_2 = 80. Since this is the case, it is expected that the ϕ_1 value did not vary too much.

C2.

The edge map of letter.bmp (Fig. 23) is first computed using the bwperim() function, after using intermeans.m to threshold the image.

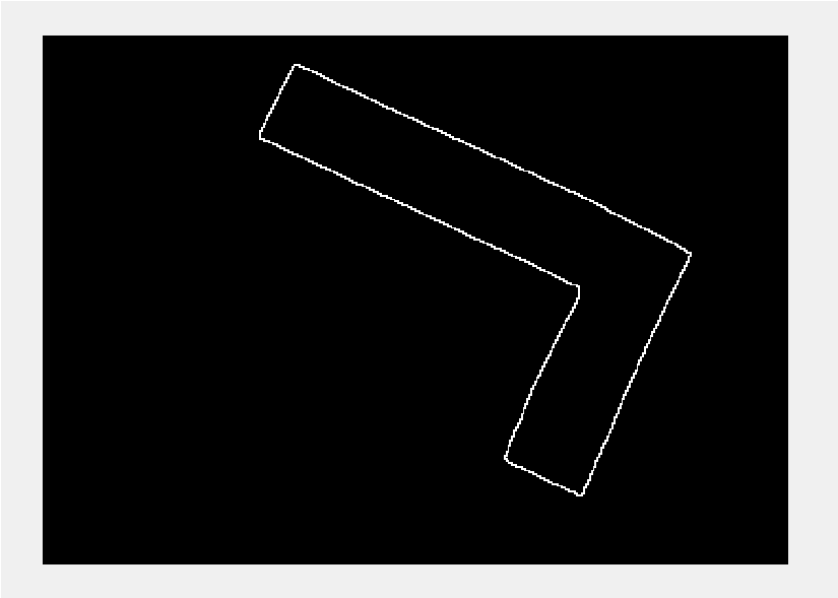


Fig. 23. Edge map of letter.bmp

C3. run_C.m

To perform Hough Transform, we will use the normal representation of a line:

$$tetha_i = \tan^{-1}(y_i/x_i) \quad \ldots(1)$$

$$rho = x_i*\cos(theta) + y_i*\sin(theta) \quad \ldots(2)$$

Where x_i, y_i represent pixels along the edge map.

The range of rho to use was determined by the following formula:

$$rho_i = sqrt(x_i^2 + y_i^2) \quad \ldots(3)$$

```
16  % Populate accumulator
17  t_cols = size(Iout,2); % no. of columns
18  t_rows = size(Iout,1); % no. of rows
19
20  rho_range = round(sqrt(t_rows^2 + t_cols^2)); %transformation to rho
21  acc = zeros(2*rho_range,180); %initialise accumulator
22  edge_pixels = length(x_edges); % no. of perimeter pixels
23
24  for i = 0:edge_pixels-1 % iterate across all edge pixels
25      j = 0;
26      for theta = (-pi/2):pi/180:(pi/2-pi/180) % increment thetha by pi/180
27          rho = round(x_edges(i+1).*cos(theta) + y_edges(i+1).*sin(theta));
28          acc((rho_range+rho),j+1) = acc(rho_range+rho,j+1)+1; %add one vote
29          %for clearer plotting, enable this line only when plotting hough
30          %transform graph
31          %acc((rho_range+rho),j+1) = 1.45*acc((rho_range+rho),j+1);
32          j = j+1;
33      end
34  end
```

Fig. 24 Populating the accumulator

1. In populating the accumulator, the total maximum range of rho was first calculated using the equation described in (3). Rho takes on negative values as well, hence the accumulator was populated with the number of rows as 2*rho_range to avoid negative indexes. The total range of tetha is +pi/2 to -pi/2, hence we intialise 180 columns in the accumulator, similarly to avoid negative indexes.
2. Iterating across each edge pixel (Line 24), we then nest a loop iterating across values of theta from -pi/2 to +pi/2, incrementing by pi/180 each iteration (Line 26). Within this nested loop, we **find the sinusoid rho that corresponds to each edge pixel that we are iterating across, which reflects the main concept behind the Hough Transform.** We then add "votes" to the corresponding accumulator bins that this sinusoid passes through (Line 28).
3. Note that rho_range, half of the maximum range of rho, was added in the first index of acc (Line 28) so as to avoid the issue of negative index. This is later rectified in Line 40 of Fig. 25, where we subtract rho_range to offset the difference when we go back to the spatial domain.

```
36  % Plot edge lines
37  for i = 1:180 %iterate across range of theta
38      for j = 1:rho_range*2 %iterate across range of rho
39          if acc(j,i) >= 37 %set threshold to be 37 votes
40              rho_offset = j - rho_range;
41              x_lines = 0:t_cols;
42              y_lines = (rho_offset - x_lines*cosd(i-90))/sind(i-90);
43              plot(x_lines,y_lines);
44          end
45      end
46  end
47
```

Fig. 25. Code for plotting edge lines on letter.bmp

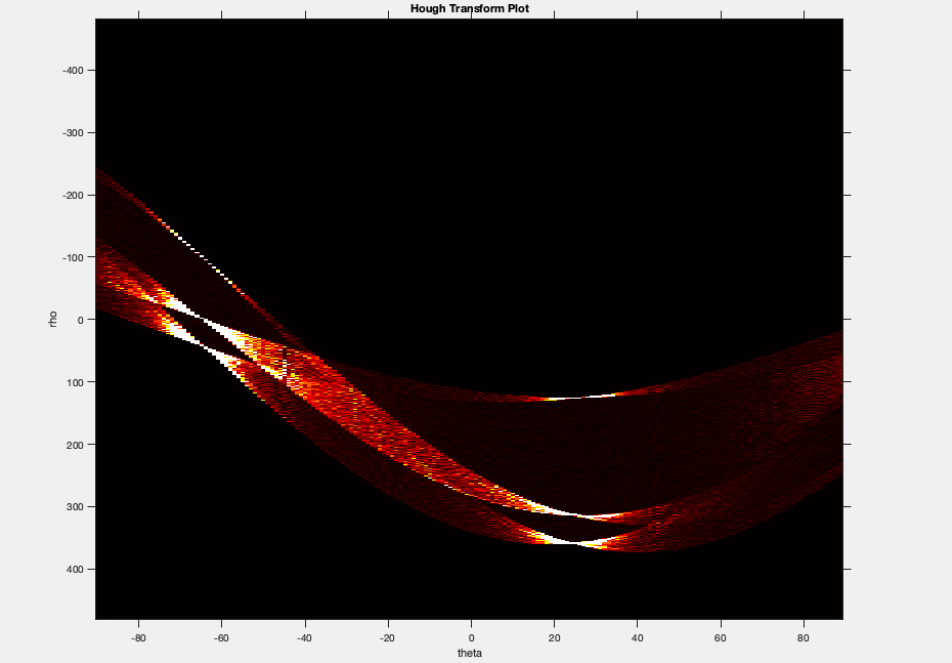


Fig. 26. Plot of Hough Transform

4. Once we have populated the accumulator array for all sinusoids in the row theta domain that correspond to edge pixel points in spatial domain, we are able to plot the hough transform graph in the rho theta domain (Fig. 26). From the plot, it is apparent that are 6 distinct regions where the sinusoids intersect. **These 6 points in the rho theta domain correspond to the 6 edge lines in the spatial domain that make up the letter 'L'.**

C4.

5. We are now ready to plot the edge lines. We iterate across the range of theta and nest a loop which iterates across the range of rho. We set a threshold arbitrarily in order not to flood the original image with edge lines, then tune it such that we minimally obtain one edge line for all the 6 expected edges that make up the letter 'L'. We find out after trial-and-error that the optimal threshold is 37, and correspondingly plot each edge line. Plotting the edge lines yields Fig. 27.

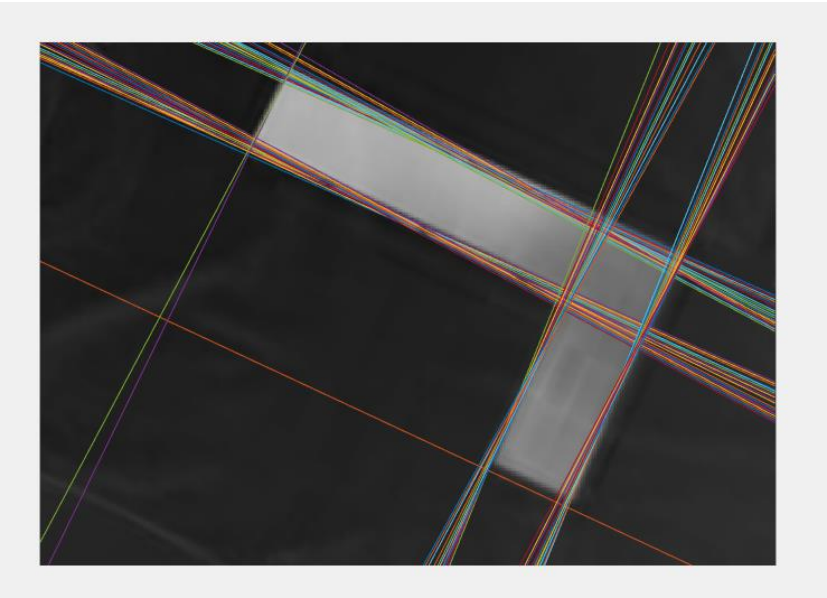


Fig. 27. Edge lines plot of letter.bmp

The image was subsequently saved as letter_line.bmp through the use of saveas() function. For further improvement, we could possibly come up with an algorithm to detect edge lines that have already been plotted, and for edge lines that have multiple plots, to select the most appropriate edge line to be used, so that we end up with only one edge line for each edge and a cleaner edge line plot overall.

D2. rtheta.m

1. The r-θ function can be obtained by plotting the distance of the boundary of the object as a function of angle. For each edge pixel i, the definitions governing r_i and θ_I are as described in Fig. 28. As observed, we will first need to calculate xbar and ybar, which gives us coordinates of the centroid of the boundary object. We have calculated this in features.m, and hence will reuse the earlier centroid calculations to obtain xbar and ybar (Fig. 29)

$$\text{Radial distance:} \quad r_i = \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2}$$
$$\text{Polar angle:} \quad \theta_i = \tan^{-1} \frac{y_i - \bar{y}}{x_i - \bar{x}}$$

Fig. 28. Radial distance and Polar angle formulas used in code

```
9       %calculate xbar, ybar
10      m00 =0;
11      m10 = 0;
12      m01 = 0;
13      t_cols = size(Iin,2); %total columns
14      t_rows = size(Iin,1); %total rows
15
16      for x = 0:t_cols-1 %iterating across columns
17          for y = 0:t_rows-1 %iterating across rows
18              m00 = m00 + Iin(t_rows-y, x+1);
19              m10 = m10 + (x * Iin(t_rows-y, x+1));
20              m01 = m01 + (y * Iin(t_rows-y, x+1));
21          end
22      end
23
24      xbar = m10/m00;
25      ybar = m01/m00;
```

Fig. 29 Calculating xbar, ybar in rhtheta.m

```
27      r = zeros(sum(sum(Iin)),1);
28      theta = zeros(sum(sum(Iin)),1);
29      pos = 0;
30      for x = 0:t_cols-1
31          for y = 0:t_rows-1
32               if Iin(t_rows-y, x+1) == 1 %if pixels are edge pixels
33                  pos = pos + 1;
34                  r(pos) = sqrt((x-xbar)^2 + (y-ybar)^2); %populate r array
35                  theta(pos) = atan2(y-ybar, x-xbar); %populate theta array
36                  if theta(pos)<0
37                      theta(pos) = theta(pos) + 2*pi; %right-shift to make x-axis positive
38                  end
39              end
40          end
41      end
```

Fig. 30. Populating arrays for r and theta

2. To carry out our r-θ plot, we first have to populate our table of r and θ values for each edge pixel. To do this, we initialise arrays for theta and r respectively (Lines 27-28). We then iterate across pixels in the input image, looking for pixels which form the

boundary of the object (i.e pixel intensity == 1). For each of these pixels, we use the equations as described in Fig. 28 to calculate the corresponding radial distance r, and polar angle θ. These calculated values are appended to the arrays (Lines 34-35). Lastly, we add 2*pi to all negative values of theta (Line 37), which has the effect of right-shifting the x-axis to make sure it starts from 0. The resultant r-θ plot is as follows (Fig. 31).

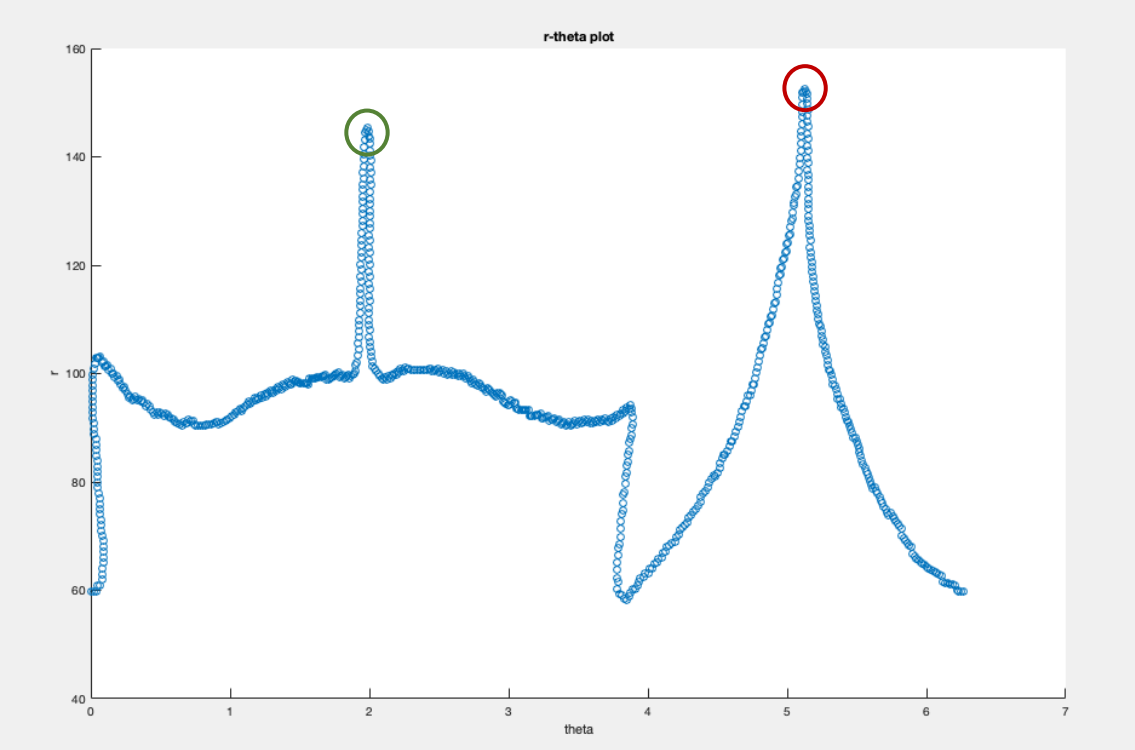


Fig.31. r-θ plot of test3.bmp

3. We can see that the y-axis of the graph corresponds to the distance between the centroid of the object and the edge pixel, as we traverse across the pixels in a circular motion with the centroid as pivot. Most notably, we can observe that the first peak of the r-θ corresponds to the stem portion of the leaf (green circle, Fig. 32), while the second peak of the r-θ plot corresponds to the tip of the leaf (red circle, Fig. 32). This is because the distance between the object's centroid and these these two ends of the leaf have the greatest distance, hence are reflected as peaks in the r-θ plot.

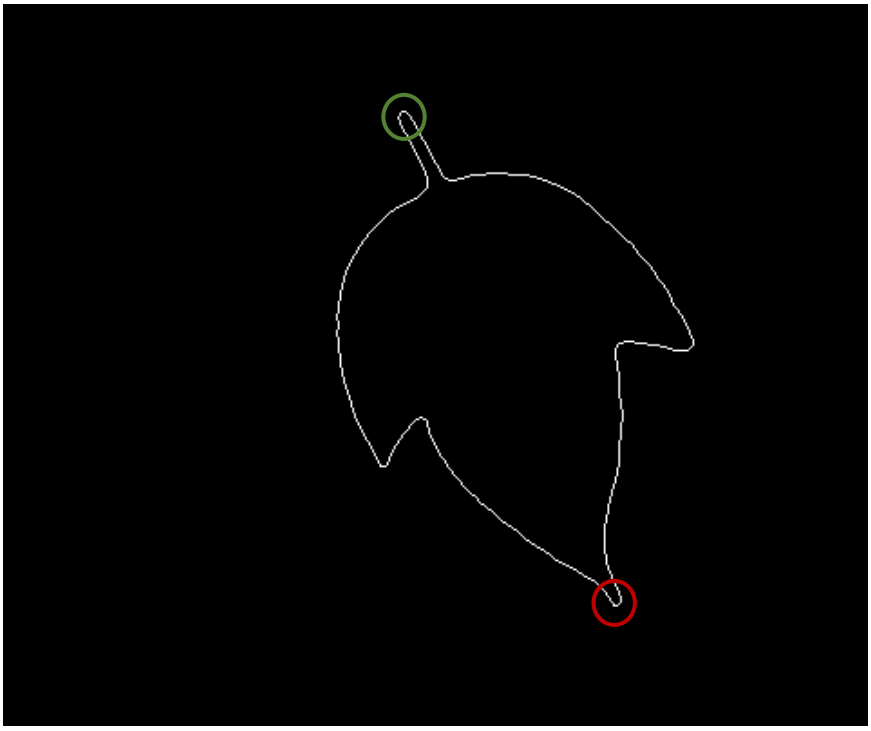


Fig. 32. Original test3.bmp image