PROJECT 2

Computer Vision & Image Processing

ECE 565

By: Pranati Trivedi

Q1:

Write a global thresholding program in which the threshold is estimated automatically using the procedure discussed in Section 10.3.2. The output of your program should be a segmented (binary) image. Use your program to segment “noisy\_fingerprint.tiff” and produce a segmented image.

clc;

clear all;

img = imread('D:\IITCSem2\CVIP\noisy\_fingerprint.tif');

%img = imread('D:\IITCSem2\CVIP\polymersomes.tif'); %for Q2 (b)

imshow(img); title('original fingerprint image');

figure(); imhist(img); title('histogram of image');

[counts, r] = imhist(img);

i = 1;

mu = cumsum(counts);

T(i) = (sum(r.\*counts))/mu(end);

T(i) = round(T(i));

mu2 = cumsum(counts(1:T(i)));

MBT = sum(r(1:T(i)).\*counts(1:T(i)))/mu2(end);

mu3 = cumsum(counts(T(i):end));

MAT = sum(r(T(i):end).\*counts(T(i):end))/mu3(end);

i = i+1;

T(i) = round((MAT + MBT)/2);

while abs(T(i)-T(i-1))>=1

mu2=cumsum(counts(1:T(i)));

MBT=sum(r(1:T(i)).\*counts(1:T(i)))/mu2(end);

mu3=cumsum(counts(T(i):end));

MAT=sum(r(T(i):end).\*counts(T(i):end))/mu3(end);

i=i+1;

T(i)=round((MAT+MBT)/2);

Threshold=T(i);

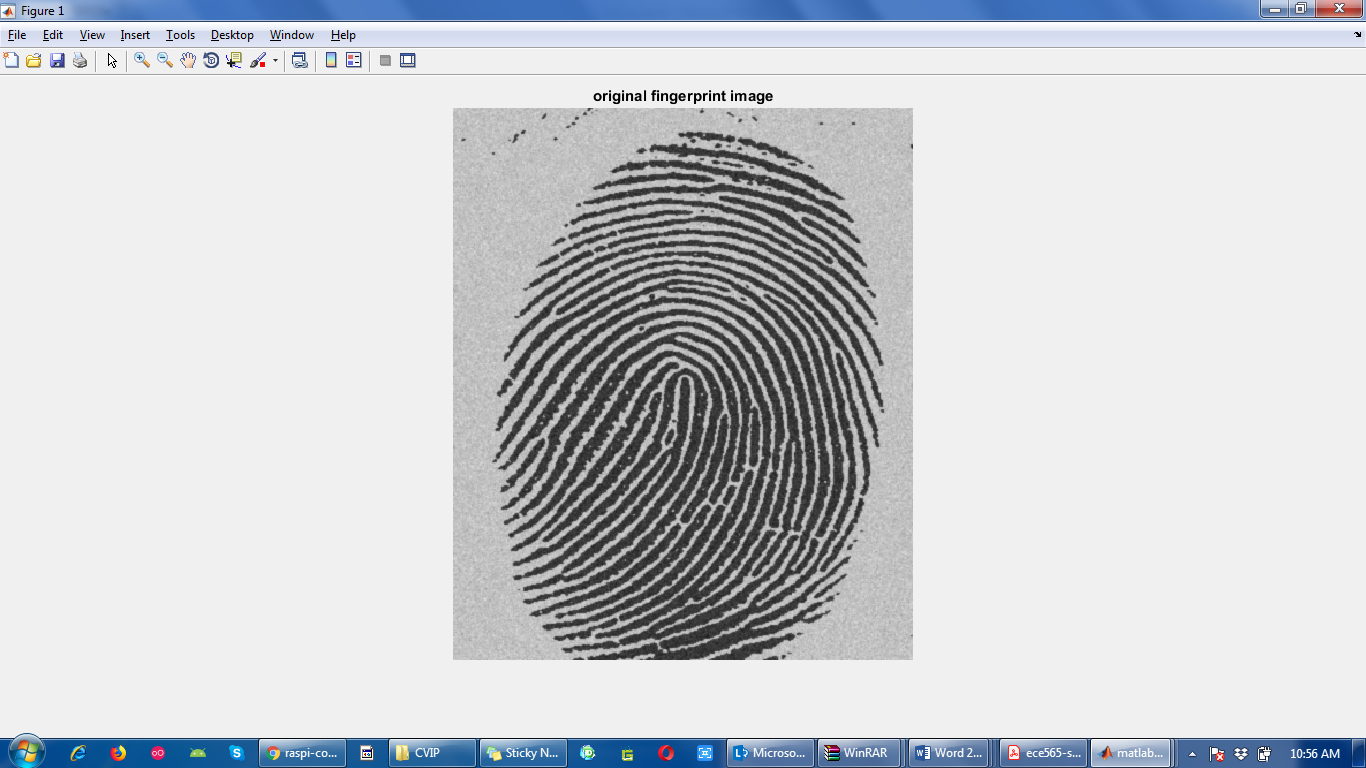
end % Normalize the threshold to the range [i, 1].

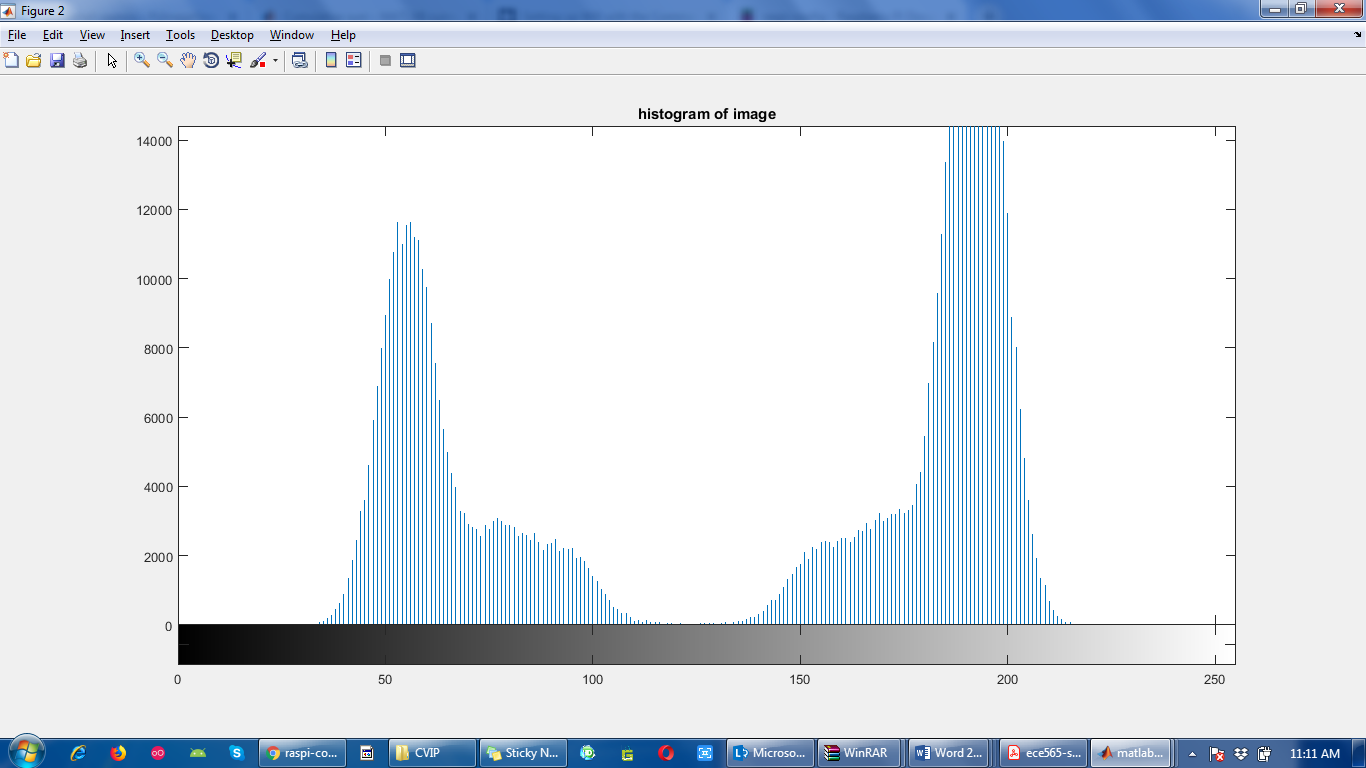
level = (Threshold - 1) / (r(end) - 1);

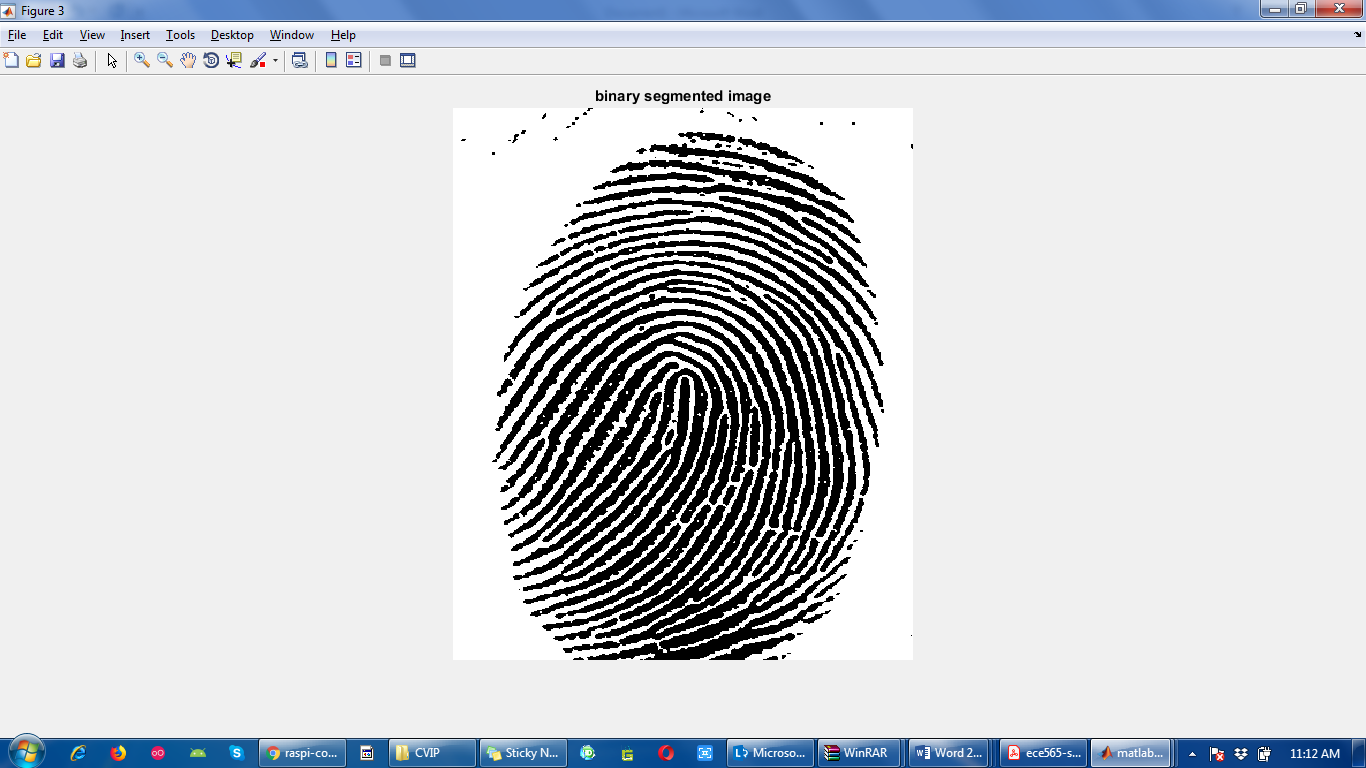
BW = im2bw(img,level);

figure(); imshow(BW); title('binary segmented image');

Output:







Q2: (20 points) **Otsu’s thresholding**

(a) (15 points) Implement Otsu’s optimum thresholding algorithm given in Section 10.3.3 (You should not use **graythresh** for this problem. Please implement the Otsu’s method from scratch by yourself.). Use your implementation of Otsu’s algorithm to segment “polymersomes.tiff”

(b) (5 points) Use the global thresholding algorithm from Problem1 to segment “polymersomes.tiff” and compare the result with the segmented image obtained in Part (a).

clc;

clear all;

img = imread('D:\IITCSem2\CVIP\polymersomes.tif');

imshow(img); title('original polymersomes image');

mean\_g = 0;

mt = 0;

[row,col] = size(img); % obtain row and col values

figure(); imhist(img); title('histogram of image'); % obtain histogram of image

h = imhist(img);

pi = h/(row.\*col);

for i=1:1:256

if pi(i)~=0

lv=i;

break

end

end

for i=256:-1:1

if pi(i)~=0

hv=i;

break

end

end

lh = hv - lv;

for k = 1:256

p1(k)=sum(pi(1:k));

p2(k)=sum(pi(k+1:256)); % obtain cumulative sums

end

for k=1:256

m1(k)=sum((k-1)\*pi(1:k))/p1(k);

m2(k)=sum((k-1)\*pi(k+1:256))/p2(k); % obtain cumulative mean

end

for k=1:256

mean\_g=(k-1)\*pi(k)+mean\_g; % obtain global intensity mean

end

for k =1:256

var(k)=p1(k)\*(m1(k)-mean\_g)^2+p2(k)\*(m2(k)-mean\_g)^2;

end

[y,T]=max(var(:));

T=T+lv;

g=img;

g1=find(g>=T); % thresholding

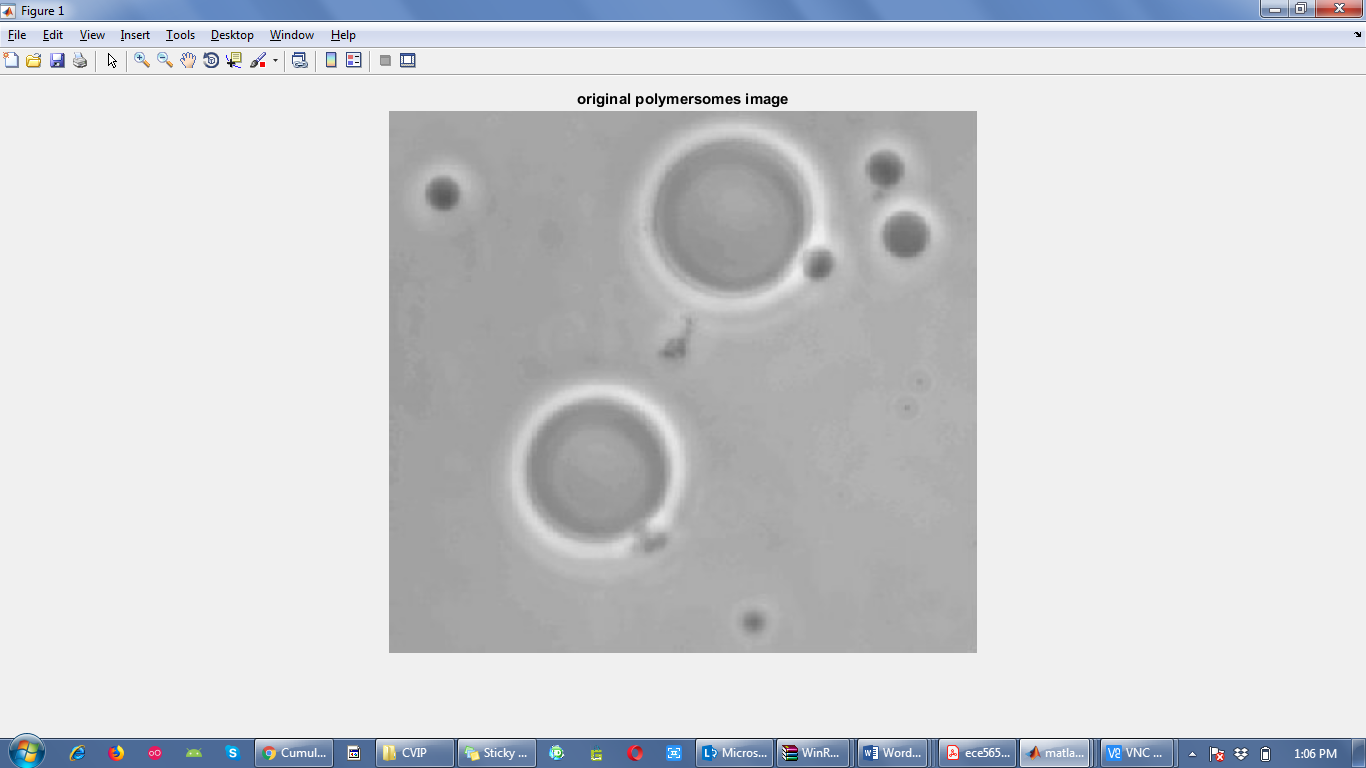
g(g1)=255;

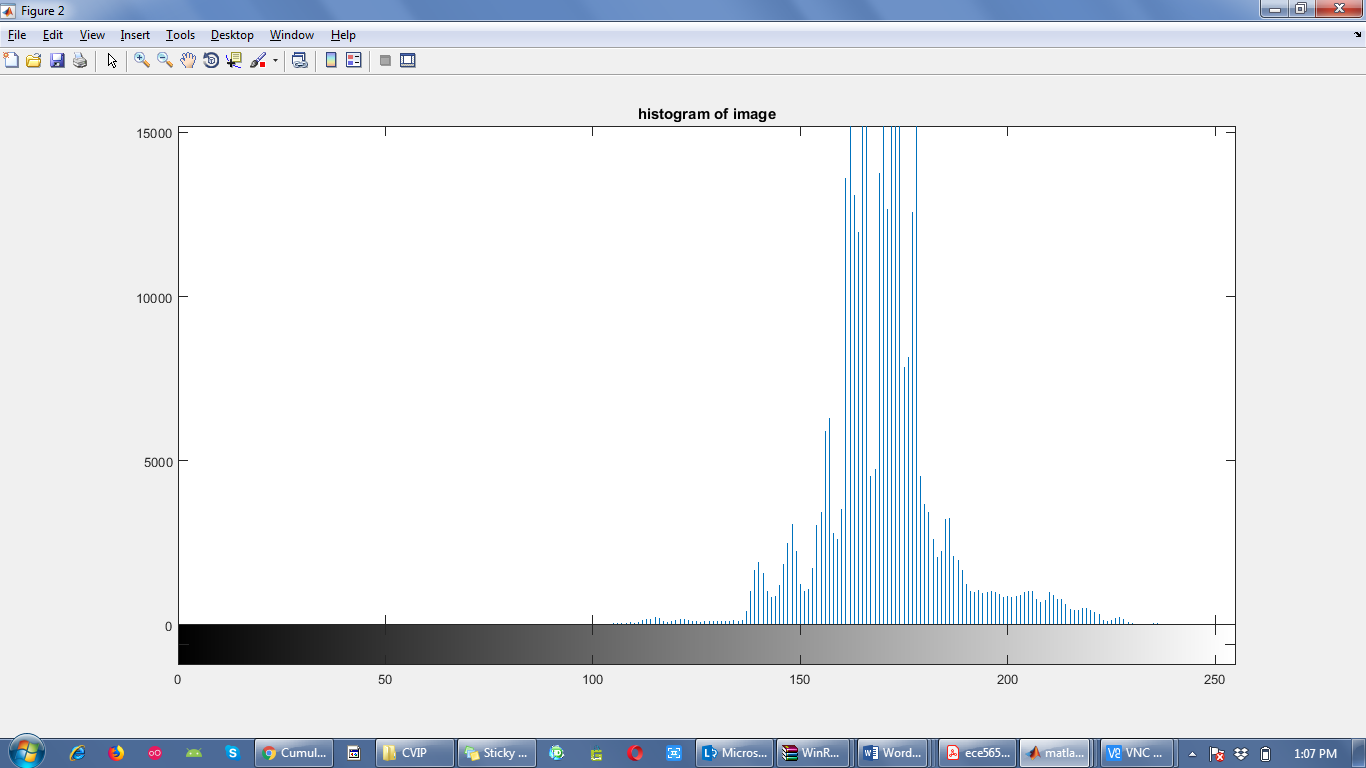
g2=find(g<T);

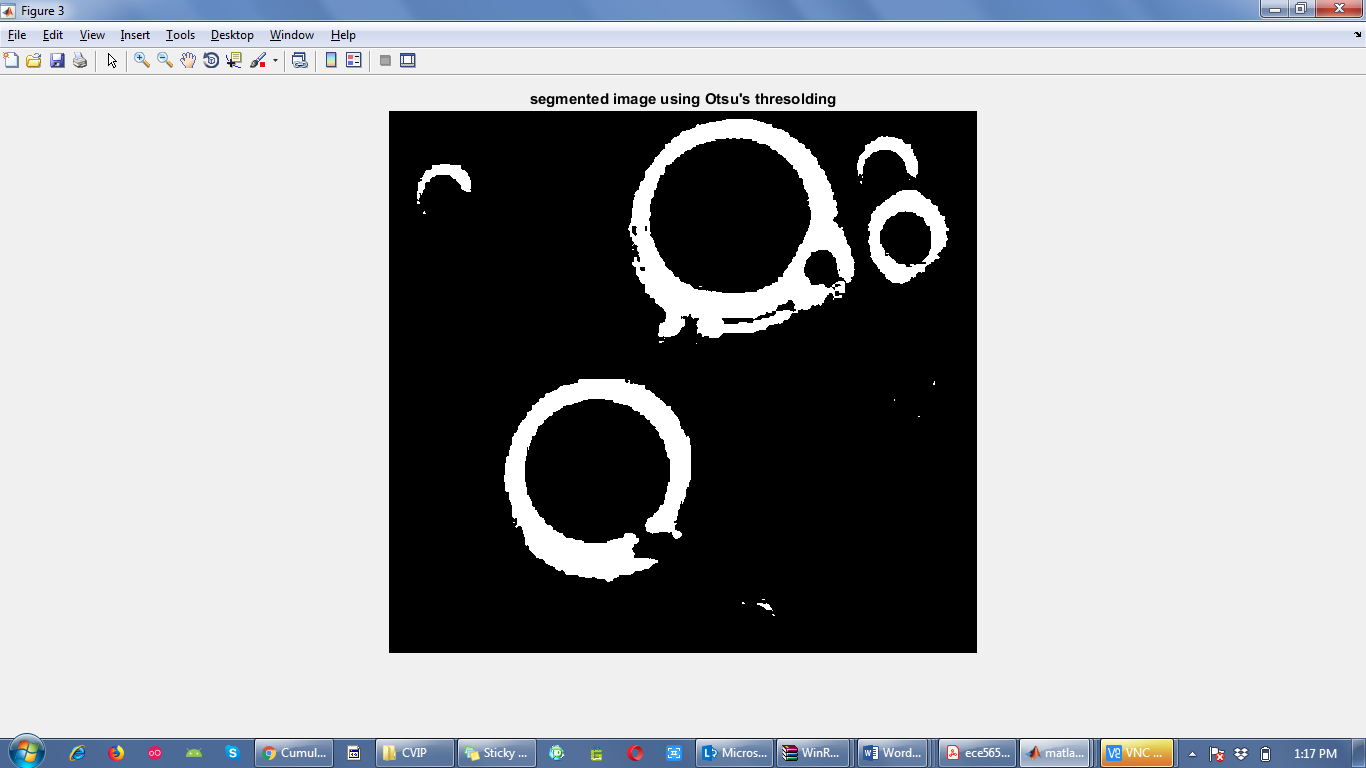
g(g2)=0;

figure(); imshow(g); title('segmented image using Otsu''s thresolding');

Output:







(b)

clc;

clear all;

%img = imread('D:\IITCSem2\CVIP\noisy\_fingerprint.tif');

img = imread('D:\IITCSem2\CVIP\polymersomes.tif'); %for Q2 (b)

imshow(img); title('original fingerprint image');

figure(); imhist(img); title('histogram of image');

[counts, r] = imhist(img);

i = 1;

mu = cumsum(counts);

T(i) = (sum(r.\*counts))/mu(end);

T(i) = round(T(i));

mu2 = cumsum(counts(1:T(i)));

MBT = sum(r(1:T(i)).\*counts(1:T(i)))/mu2(end);

mu3 = cumsum(counts(T(i):end));

MAT = sum(r(T(i):end).\*counts(T(i):end))/mu3(end);

i = i+1;

T(i) = round((MAT + MBT)/2);

while abs(T(i)-T(i-1))>=1

mu2=cumsum(counts(1:T(i)));

MBT=sum(r(1:T(i)).\*counts(1:T(i)))/mu2(end);

mu3=cumsum(counts(T(i):end));

MAT=sum(r(T(i):end).\*counts(T(i):end))/mu3(end);

i=i+1;

T(i)=round((MAT+MBT)/2);

Threshold=T(i);

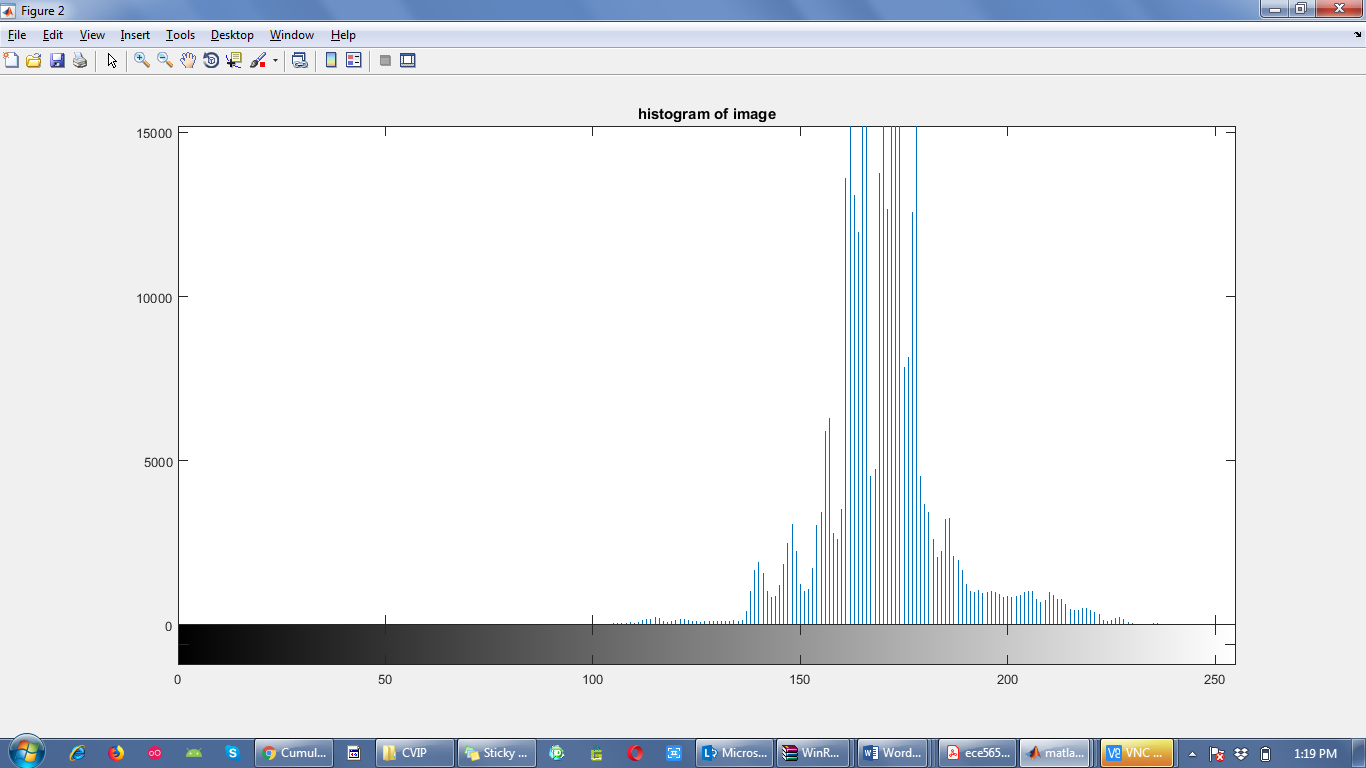
end % Normalize the threshold to the range [i, 1].

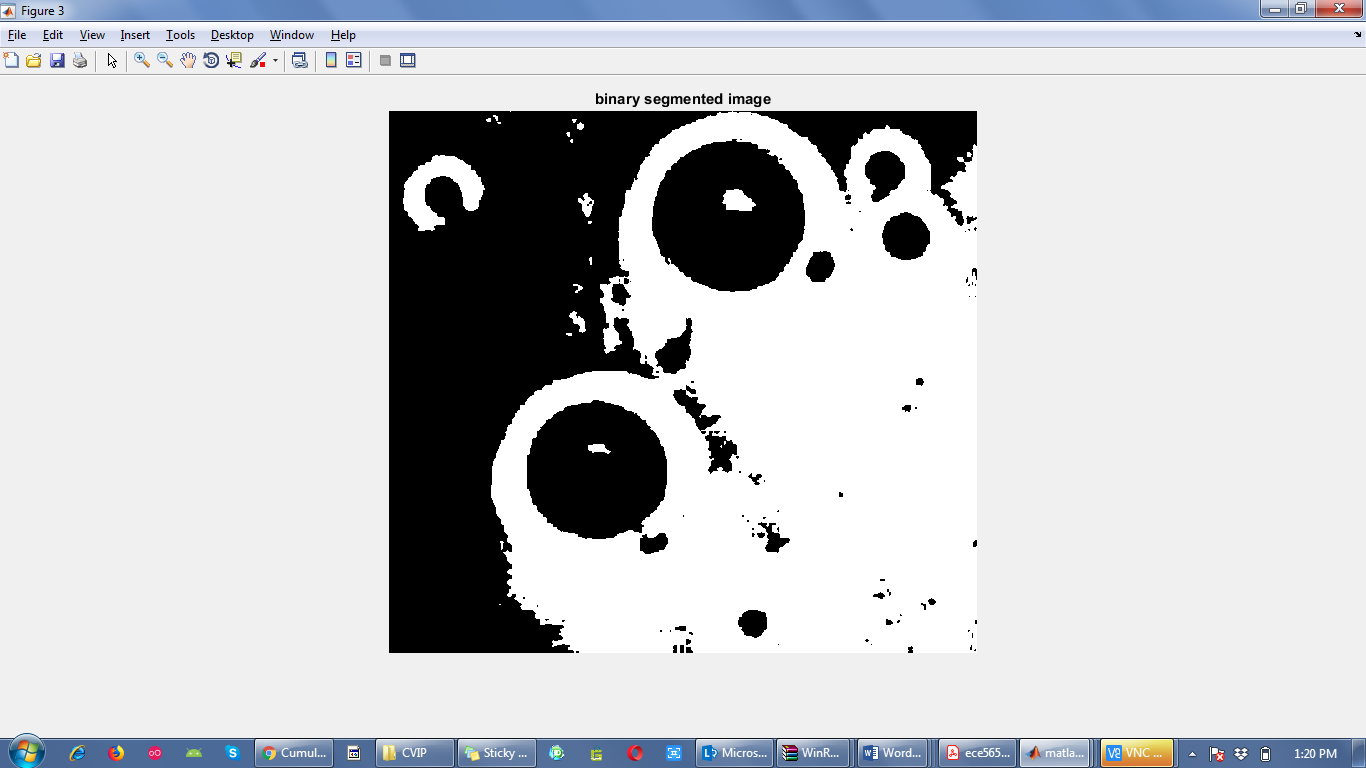
level = (Threshold - 1) / (r(end) - 1);

BW = im2bw(img,level);

figure(); imshow(BW); title('binary segmented image');

Output:





Comparing the image obtained using global thresholding method with Otsu’s method we can observe that the global thresholding method does not give a favorable output for polymersomes.tif image while it gave good output for that fingerprint image. So, it can be said that global thresholding method is not able to accurately decide the threshold value for all images while Otsu’s method was able to find out the perfect value of threshold for polymersome image. Global thresholding method fails in segmenting the object in the image and shows more of white background. While Otsu’s method segmented the image in binary in a better way as it uses histogram to compute the threshold.

Q3:

Figure 3 shows a 570 X 570 image **f** of a circular stroke embedded in specular noise. The objective of the problem is to obtain the Freeman chain code, the first difference of the outer boundary of the largest object, and the integer of minimum magnitude of the code.

(a) (2 points) Generate a smoothed image **g** using 9 X 9 averaging filter.

(b) (2 points) Generate a binary image **gB** by thresholding **g** obtained in Part (a).

(c) (3 points) Extract the outer boundary of **gB** and display the results as a binary image.

(d) (3 points) Subsample the boundary obtained in Part (c) onto a grid whose lines are separated by 50 pixels. Connect the subsampled boundary points with straight line segments. Display the resulting points as a binary image.

(e) (30 points) Write a program that computes the Freeman chain code **c** of a boundary **b** with the code connectivity specified in **CONN** (i.e., **c = fchcode(b, CONN)**). The input **b** is a set of 2-D coordinate pairs for a boundary and **CONN** can be 8 for an 8-connected chain code or 4 for a 4-connected chain code. The output **c** is a structure with the following fields:

c.fcc = chain code (1 X np where np is the number of boundary pixels)

c.diff = First difference of code c.fcc (1 x np)

c.mm = Integer of minimum magnitude from c.fcc (1 x np)

c.diffmm = First difference of code c.mm (1 x np)

c.x0y0 = Coordinates where the code starts (1 x 2)

Generate the chain code, the first difference of the chain code, and the integer of minimum magnitude of the code for the subsampled boundary obtained in Part (d).

%Q3 (a)

clc;

clear all;

img = imread('D:\IITCSem2\CVIP\circular\_stroke.tif');

imshow(img); title('Input image');

[row col] = size(img);

w = (1/81)\*(ones(9)); % 9x9 averaging filter

g = imfilter(img, w, 'conv'); % smooth image

figure(); imshow(g); title('Smooth image'); % display smooth image

%Q3 (b)

gB = im2bw(g,0.5); % Thresholding

figure(); imshow(gB); title('Thresholded image');

%Q3 (c)

b1 = bwboundaries(gB, 'noholes'); % px1 cell array; p is number of objects detected with given parameters

b = b1{1,1}; % npx2 matrix containing coordinates of boundary

gh = bound2im(b, row, col);

figure(); imshow(gh); title('binary image of outer boundary'); % displays boundary converted to image

%Q3 (d)

[s, su] = bsubsamp(b, 50); % subsampling at distance of 50 pixels

gk = bound2im(s, row, col);

figure(); imshow(gk); title('subsampled points');

cn = connectpoly(s(:,1), s(:,2)); % connect subsampled points with straight line segments

cn2 = bound2im(cn, row, col);

figure(); imshow(cn2); title('connected subsampled points')

%Q3 (e)

c= fchcode(su,8); % Freeman chain code for 8 connectivity

disp(c);

Functions:

function image = bound2im(b, M, N)

% BOUND2IM Converts a boundary to an image.

% IMAGE = BOUND2IM(b) converts b, an np-by-2 array containing the

% integer coordinates of a boundary, into a binary image with 1s

% in the locations of the coordinates in b and 0s elsewhere. The

% height and width of the image are equal to the Mmin + H and Nmin

% + W, where Mmin = min(b(:,1)) - 1, N = min(b(:,2)) - 1, and H

% and W are the height and width of the boundary. In other words,

% the image created is the smallest image that will encompass the

% boundary while maintaining the its original coordinate values.

%

% IMAGE = BOUND2IM(b, M, N) places the boundary in a region of

% size M-by-N. M and N must satisfy the following conditions:

%

% M >= max(b(:,1)) - min(b(:,1)) + 1

% N >= max(b(:,2)) - min(b(:,2)) + 1

%

% Typically, M = size(f, 1) and N = size(f, 2), where f is the

% image from which the boundary was extracted. In this way, the

% coordinates of IMAGE and f are registered with respect to each

% other.

% Check input.

if size(b,2) ~= 2

error('The boundary must be of size np-by-2')

end

% Make sure the coordinates are integers.

b = round(b);

% Defaults.

if nargin == 1

Mmin = min(b(:,1)) - 1;

Nmin = min(b(:,2)) - 1;

H = max(b(:,1)) - min(b(:,1)) + 1; % Height of boundary.

W = max(b(:,2)) - min(b(:,2)) + 1; % Width of boundary.

M = H + Mmin;

N = W + Nmin;

end

% Create the image.

image = false(M,N);

linearIndex = sub2ind([M, N], b(:,1), b(:,2));

image(linearIndex) = 1;

function [s, sUnit] = bsubsamp(b, gridsep)

%BSUBSAMP Subsample a boundary.

% [S, SUNIT] = BSUBSAMP(B, GRIDSEP) subsamples the boundary B by

% assigning each of its points to the grid node to which it is

% closest. The grid is specified by GRIDSEP, which is the

% separation in pixels between the grid lines. For example, if

% GRIDSEP = 2, there are two pixels in between grid lines. So, for

% instance, the grid points in the first row would be at (1,1),

% (1,4), (1,6), ..., and similarly in the y direction. The value

% of GRIDSEP must be an even integer. The boundary is specified by

% a set of coordinates in the form of an np-by-2 array. It is

% assumed that the boundary is one pixel thick.

%

% Output S is the subsampled boundary. Output SUNIT is normalized so

% that the grid separation is unity. This is useful for obtaining

% the Freeman chain code of the subsampled boundary. The outputs are

% in the same order (clockwise or counterclockwise) as the input.

% There are no duplicate points in the output.

% Check input.

[np, nc] = size(b);

if np < nc

error('B must be of size np-by-2.');

end

if isinteger(gridsep)

error('GRIDSEP must be an integer.')

end

% Find the maximum span of the boundary.

xmax = max(b(:, 1));

ymax = max(b(:, 2));

% Determine the integral number of grid lines with gridsep points in

% between them that encompass the intervals [1,xmax], [1,ymax].

GLx = ceil((xmax + gridsep)/(gridsep + 1));

GLy = ceil((ymax + gridsep)/(gridsep + 1));

% Form vectors of x and y grid locations.

I = 1:GLx;

J = 1:GLy;

% Vector of grid line locations intersecting x-axis.

X(I) = gridsep\*I + (I - gridsep);

% Vector of grid line locations intersecting y-axis.

Y(J) = gridsep\*J + (J - gridsep);

[C, R] = meshgrid(Y, X);

% Vector of grid all coordinates, arranged as Numbergridpoints-by-2

% array to match the horizontal dimensions of b. This allows

% computation of distances to be vectorized and this be much more

% efficient.

V = [C(1:end) ; R(1:end)]';

% Compute the distance between every element of b and every element

% of the grid.

p = np;

q = size(V, 1);

D = sqrt(sum(abs(repmat(permute(b, [1 3 2]), [1 q 1])...

- repmat(permute(V, [3 1 2]), [p 1 1])).^2, 3));

% D(i, j) is the distance between the ith row of b and the jth

% row of V. Find the min between each element of b and V.

new\_b = zeros(np, 2); % Preallocate memory.

for I = 1:np

idx = find(D(I,:) == min(D(I,:)), 1); % One min in row I of D.

new\_b(I,:) = V(idx, :);

end

% Eliminate duplicates and keep same order as input

[s, m] = unique(new\_b, 'rows');

s = [s, m];

s = fliplr(s);

s = sortrows(s);

s = fliplr(s);

s = s(:, 1:2);

% Scale to unit grid so that can use directly to obtain Freeman

% chain codes. The shape does not change.

sUnit = round(s./gridsep) + 1;

function [x, y] = intline(x1, x2, y1, y2)

%INTLINE Integer-coordinate line drawing algorithm.

% [X, Y] = INTLINE(X1, X2, Y1, Y2) computes an

% approximation to the line segment joining (X1, Y1) and

% (X2, Y2) with integer coordinates. X1, X2, Y1, and Y2

% should be integers. INTLINE is reversible; that is,

% INTLINE(X1, X2, Y1, Y2) produces the same results as

% FLIPUD(INTLINE(X2, X1, Y2, Y1)).

% Copyright 1993-2002 The MathWorks, Inc. Used with permission.

% $Revision: 1.4 $ $Date: 2003/11/21 14:38:20 $

dx = abs(x2 - x1);

dy = abs(y2 - y1);

% Check for degenerate case.

if ((dx == 0) & (dy == 0))

x = x1;

y = y1;

return;

end

flip = 0;

if (dx >= dy)

if (x1 > x2)

% Always "draw" from left to right.

t = x1; x1 = x2; x2 = t;

t = y1; y1 = y2; y2 = t;

flip = 1;

end

m = (y2 - y1)/(x2 - x1);

x = (x1:x2).';

y = round(y1 + m\*(x - x1));

else

if (y1 > y2)

% Always "draw" from bottom to top.

t = x1; x1 = x2; x2 = t;

t = y1; y1 = y2; y2 = t;

flip = 1;

end

m = (x2 - x1)/(y2 - y1);

y = (y1:y2).';

x = round(x1 + m\*(y - y1));

end

if (flip)

x = flipud(x);

y = flipud(y);

end

function c = connectpoly(x, y)

% CONNECTPOLY Connects vertices of a polygon.

% C = CONNECTPOLY(X, Y) connects the points with coordinates given

% in X and Y with straight lines. These points are assumed to be a

% sequence of polygon vertices organized in the clockwise or

% counterclockwise direction. The output, C, is the set of points

% along the boundary of the polygon in the form of an nr-by-2

% coordinate sequence in the same direction as the input. The last

% point in the sequence is equal to the first.

v = [x(:), y(:)];

% Close the polygon.

if ~isequal(v(end,:), v(1,:))

v(end + 1, :) = v(1, :);

end

% Connect vertices.

segments = cell(1, length(v) - 1);

for I = 2:length(v)

[x, y] = intline(v(I - 1, 1), v(I, 1), v(I - 1, 2), v(I, 2));

segments{I - 1} = [x, y];

end

c = cat(1, segments{:});

function c = fchcode(b, conn, dir)

%FCHCODE Computes the Freeman chain code of a boundary.

% C = FCHCODE(B) computes the 8-connected Freeman chain code of a

% set of 2-D coordinate pairs contained in B, an np-by-2 array. C

% is a structure with the following fields:

%

% c.fcc = Freeman chain code (1-by-np)

% c.diff = First difference of code c.fcc (1-by-np)

% c.mm = Integer of minimum magnitude from c.fcc (1-by-np)

% c.diffmm = First difference of code c.mm (1-by-np)

% c.x0y0 = Coordinates where the code starts (1-by-2)

%

% C = FCHCODE(B, CONN) produces the same outputs as above, but

% with the code connectivity specified in CONN. CONN can be 8 for

% an 8-connected chain code, or CONN can be 4 for a 4-connected

% chain code. Specifying CONN=4 is valid only if the input

% sequence, B, contains transitions with values 0, 2, 4, and 6,

% exclusively.

%

% C = FHCODE(B, CONN, DIR) produces the same outputs as above, but,

% in addition, the desired code direction is specified. Values for

% DIR can be:

%

% 'same' Same as the order of the sequence of points in b.

% This is the default.

%

% 'reverse' Outputs the code in the direction opposite to the

% direction of the points in B. The starting point

% for each DIR is the same.

%

% The elements of B are assumed to correspond to a 1-pixel-thick,

% fully-connected, closed boundary. B cannot contain duplicate

% coordinate pairs, except in the first and last positions, which

% is a common feature of boundary tracing programs.

%

% FREEMAN CHAIN CODE REPRESENTATION

% The table on the left shows the 8-connected Freeman chain codes

% corresponding to allowed deltax, deltay pairs. An 8-chain is

% converted to a 4-chain if (1) if conn = 4; and (2) only

% transitions 0, 2, 4, and 6 occur in the 8-code. Note that

% dividing 0, 2, 4, and 6 by 2 produce the 4-code.

%

% ----------------------- ----------------

% deltax | deltay | 8-code corresp 4-code

% ----------------------- ----------------

% 0 1 0 0

% -1 1 1

% -1 0 2 1

% -1 -1 3

% 0 -1 4 2

% 1 -1 5

% 1 0 6 3

% 1 1 7

% ----------------------- ----------------

%

% The formula z = 4\*(deltax + 2) + (deltay + 2) gives the following

% sequence corresponding to rows 1-8 in the preceding table: z =

% 11,7,6,5,9,13,14,15. These values can be used as indices into the

% table, improving the speed of computing the chain code. The

% preceding formula is not unique, but it is based on the smallest

% integers (4 and 2) that are powers of 2.

% Copyright 2002-2004 R. C. Gonzalez, R. E. Woods, & S. L. Eddins

% Digital Image Processing Using MATLAB, Prentice-Hall, 2004

% $Revision: 1.6 $ $Date: 2003/11/21 14:34:49 $

% Preliminaries.

if nargin == 1

dir = 'same';

conn = 8;

elseif nargin == 2

dir = 'same';

elseif nargin == 3

% Nothing to do here.

else

error('Incorrect number of inputs.')

end

[np, nc] = size(b);

if np < nc

error('B must be of size np-by-2.');

end

% Some boundary tracing programs, such as boundaries.m, output a

% sequence in which the coordinates of the first and last points are

% the same. If this is the case, eliminate the last point.

if isequal(b(1, :), b(np, :))

np = np - 1;

b = b(1:np, :);

end

% Build the code table using the single indices from the formula

% for z given above:

C(11)=0; C(7)=1; C(6)=2; C(5)=3; C(9)=4;

C(13)=5; C(14)=6; C(15)=7;

% End of Preliminaries.

% Begin processing.

x0 = b(1, 1);

y0 = b(1, 2);

c.x0y0 = [x0, y0];

% Make sure the coordinates are organized sequentially:

% Get the deltax and deltay between successive points in b. The

% last row of a is the first row of b.

a = circshift(b, [-1, 0]);

% DEL = a - b is an nr-by-2 matrix in which the rows contain the

% deltax and deltay between successive points in b. The two

% components in the kth row of matrix DEL are deltax and deltay

% between point (xk, yk) and (xk+1, yk+1). The last row of DEL

% contains the deltax and deltay between (xnr, ynr) and (x1, y1),

% (i.e., between the last and first points in b).

DEL = a - b;

% If the abs value of either (or both) components of a pair

% (deltax, deltay) is greater than 1, then by definition the curve

% is broken (or the points are out of order), and the program

% terminates.

if any(abs(DEL(:, 1)) > 1) | any(abs(DEL(:, 2)) > 1);

error('The input curve is broken or points are out of order.')

end

% Create a single index vector using the formula described above.

z = 4\*(DEL(:, 1) + 2) + (DEL(:, 2) + 2);

% Use the index to map into the table. The following are

% the Freeman 8-chain codes, organized in a 1-by-np array.

fcc = C(z);

% Check if direction of code sequence needs to be reversed.

if strcmp(dir, 'reverse')

fcc = coderev(fcc); % See below for function coderev.

end

% If 4-connectivity is specified, check that all components

% of fcc are 0, 2, 4, or 6.

if conn == 4

val = find(fcc == 1 | fcc == 3 | fcc == 5 | fcc ==7 );

if isempty(val)

fcc = fcc./2;

else

warning('The specified 4-connected code cannot be satisfied.')

end

end

% Freeman chain code for structure output.

c.fcc = fcc;

% Obtain the first difference of fcc.

c.diff = codediff(fcc,conn); % See below for function codediff.

% Obtain code of the integer of minimum magnitude.

c.mm = minmag(fcc); % See below for function minmag.

% Obtain the first difference of fcc

c.diffmm = codediff(c.mm, conn);

function cr = coderev(fcc)

% Traverses the sequence of 8-connected Freeman chain code fcc in

% the opposite direction, changing the values of each code

% segment. The starting point is not changed. fcc is a 1-by-np

% array.

% Flip the array left to right. This redefines the starting point

% as the last point and reverses the order of "travel" through the

% code.

cr = fliplr(fcc);

% Next, obtain the new code values by traversing the code in the

% opposite direction. (0 becomes 4, 1 becomes 5, ... , 5 becomes 1,

% 6 becomes 2, and 7 becomes 3).

ind1 = find(0 <= cr & cr <= 3);

ind2 = find(4 <= cr & cr <= 7);

cr(ind1) = cr(ind1) + 4;

cr(ind2) = cr(ind2) - 4;

function z = minmag(c)

%MINMAG Finds the integer of minimum magnitude in a chain code.

% Z = MINMAG(C) finds the integer of minimum magnitude in a given

% 4- or 8-connected Freeman chain code, C. The code is assumed to

% be a 1-by-np array.

% The integer of minimum magnitude starts with min(c), but there

% may be more than one such value. Find them all,

I = find(c == min(c));

% and shift each one left so that it starts with min(c).

J = 0;

A = zeros(length(I), length(c));

for k = I;

J = J + 1;

A(J, :) = circshift(c,[0 -(k-1)]);

end

% Matrix A contains all the possible candidates for the integer of

% minimum magnitude. Starting with the 2nd column, succesively find

% the minima in each column of A. The number of candidates decreases

% as the seach moves to the right on A. This is reflected in the

% elements of J. When length(J)=1, one candidate remains. This is

% the integer of minimum magnitude.

[M, N] = size(A);

J = (1:M)';

for k = 2:N

D(1:M, 1) = Inf;

D(J, 1) = A(J, k);

amin = min(A(J, k));

J = find(D(:, 1) == amin);

if length(J)==1

z = A(J, :);

return

end

end

function d = codediff(fcc, conn)

%CODEDIFF Computes the first difference of a chain code.

% D = CODEDIFF(FCC) computes the first difference of code, FCC. The

% code FCC is treated as a circular sequence, so the last element

% of D is the difference between the last and first elements of

% FCC. The input code is a 1-by-np vector.

%

% The first difference is found by counting the number of direction

% changes (in a counter-clockwise direction) that separate two

% adjacent elements of the code.

sr = circshift(fcc, [0, -1]); % Shift input left by 1 location.

delta = sr - fcc;

d = delta;

I = find(delta < 0);

type = conn;

switch type

case 4 % Code is 4-connected

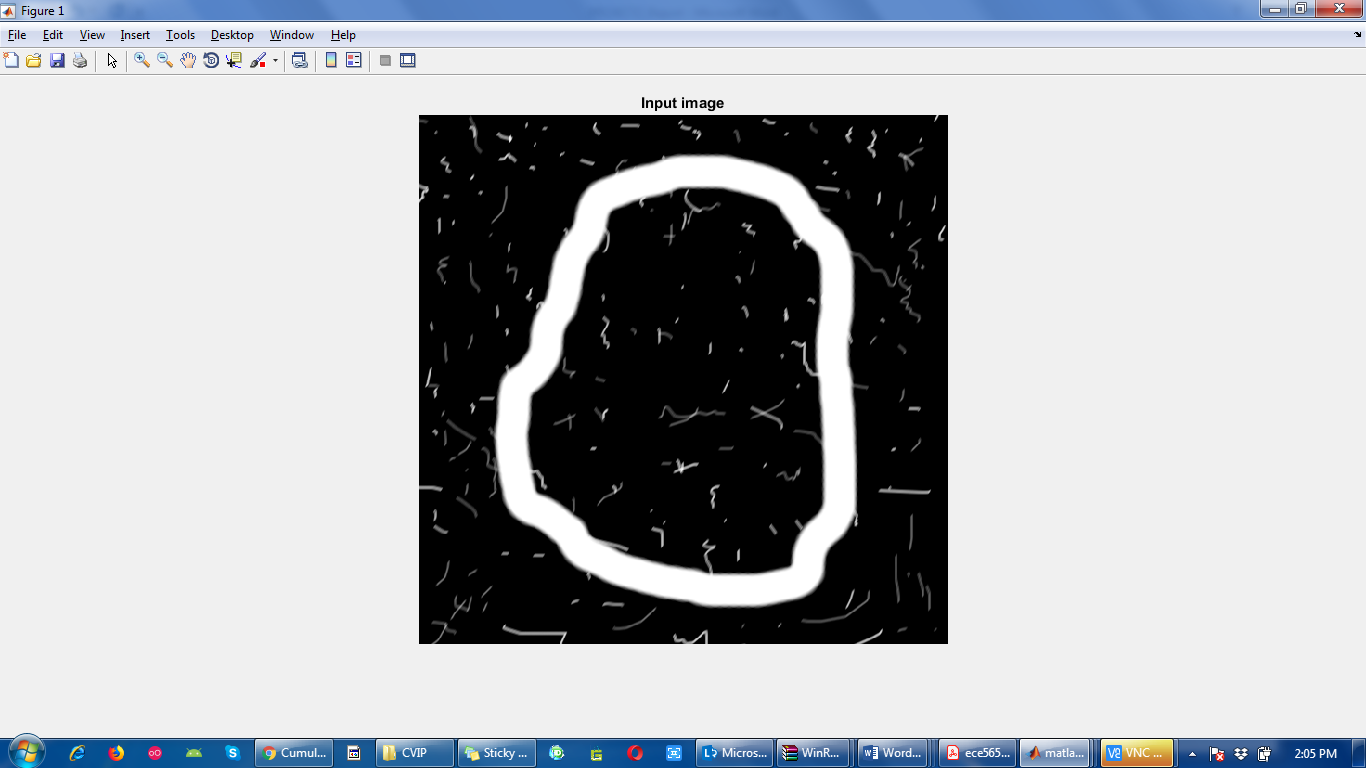
d(I) = d(I) + 4;

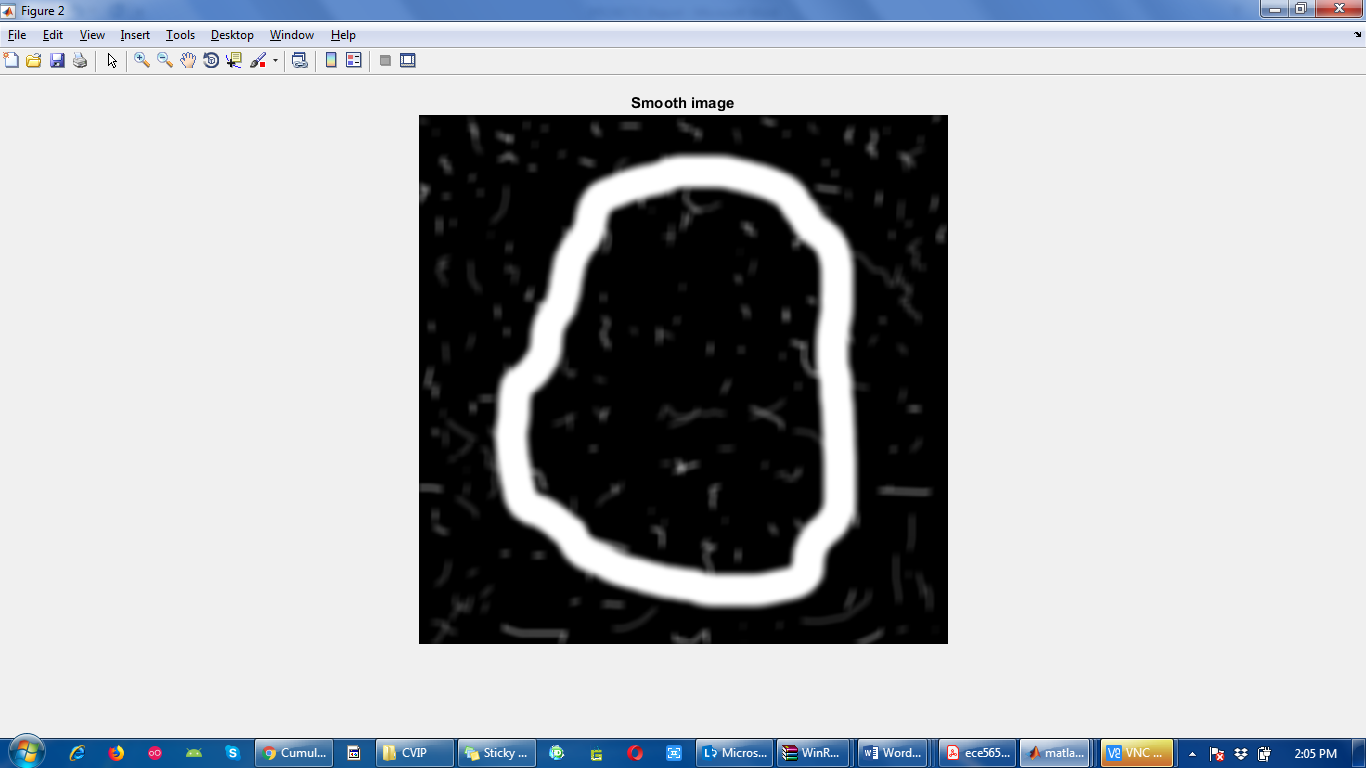
case 8 % Code is 8-connected

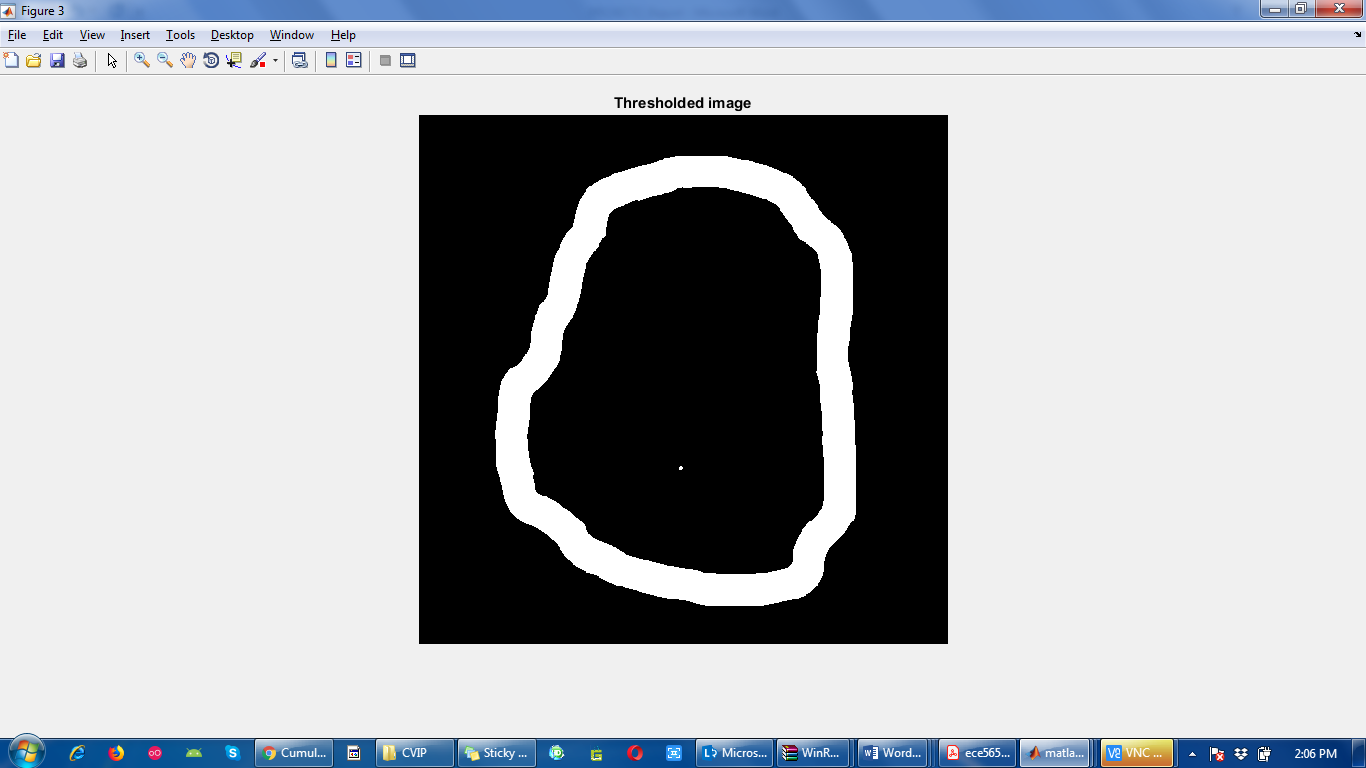
d(I) = d(I) + 8;

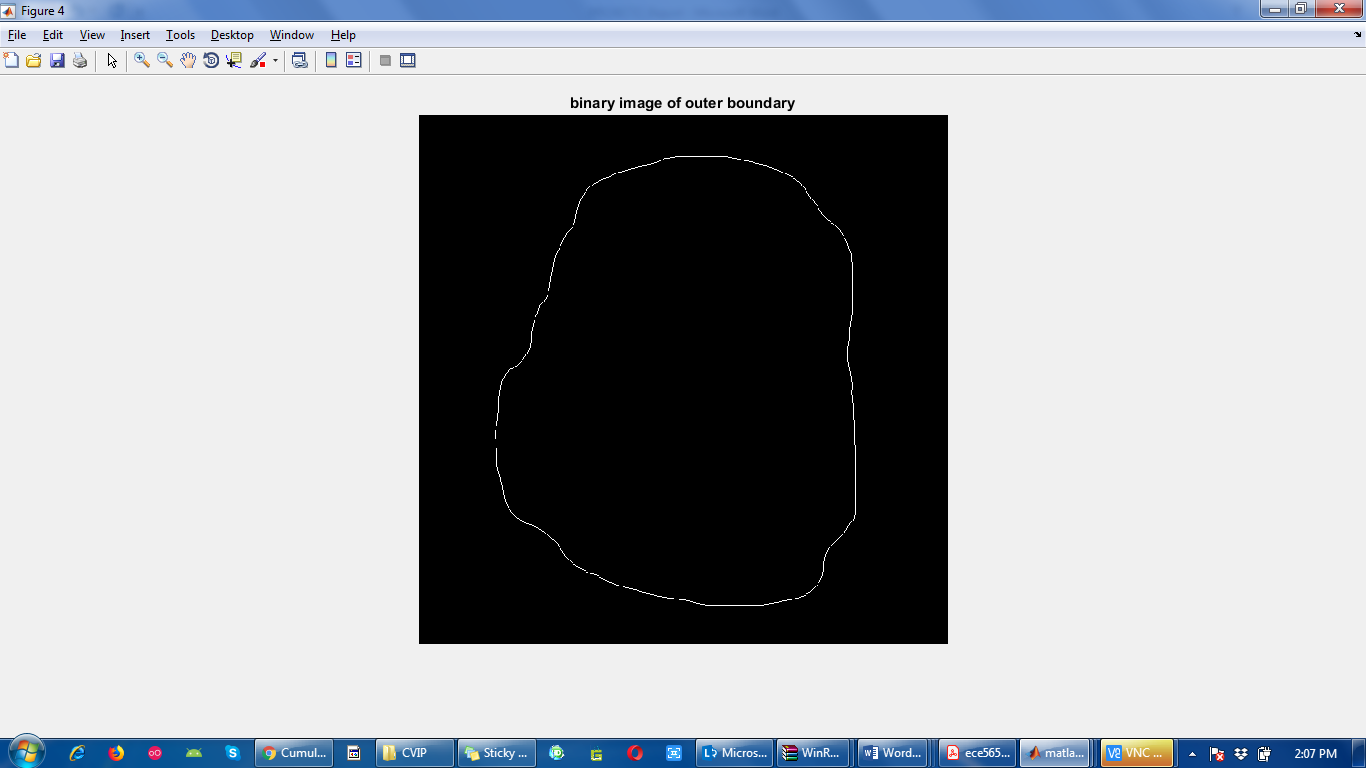
end

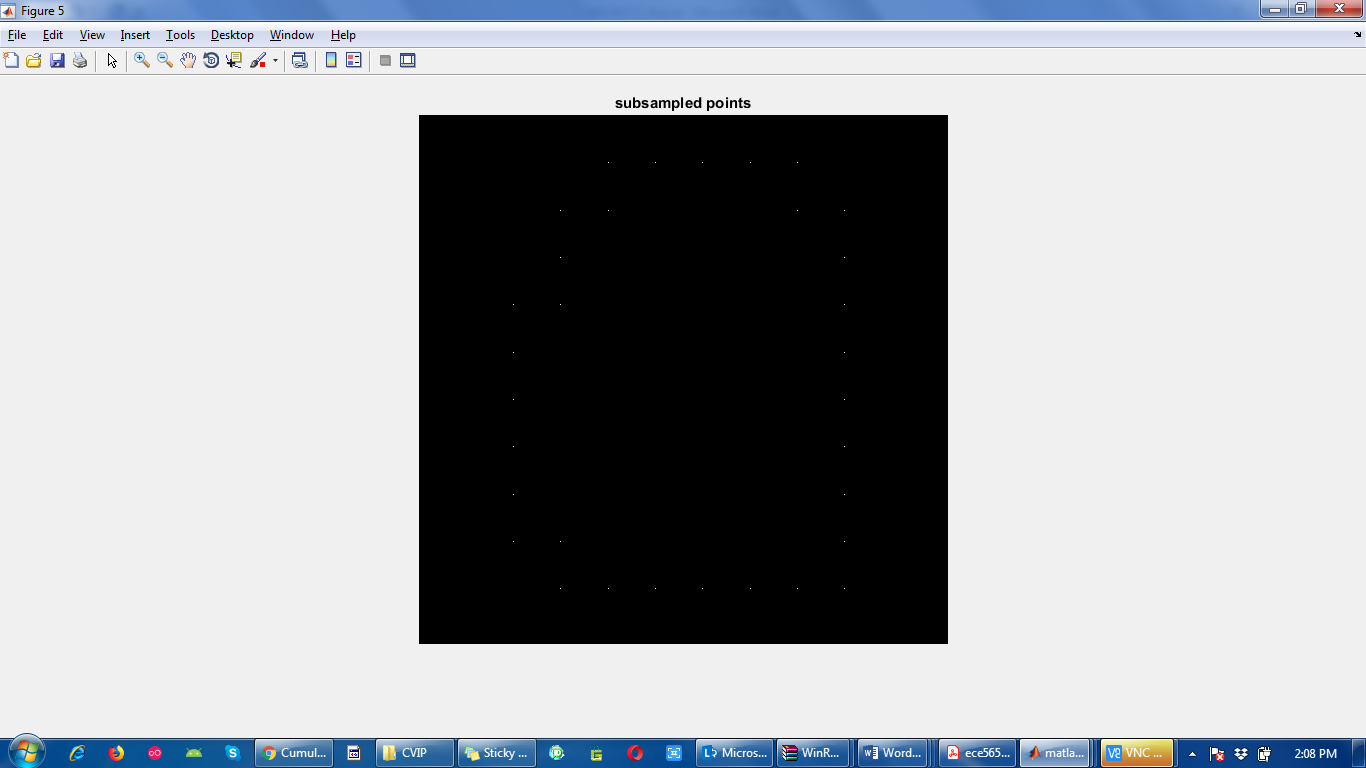
Output:

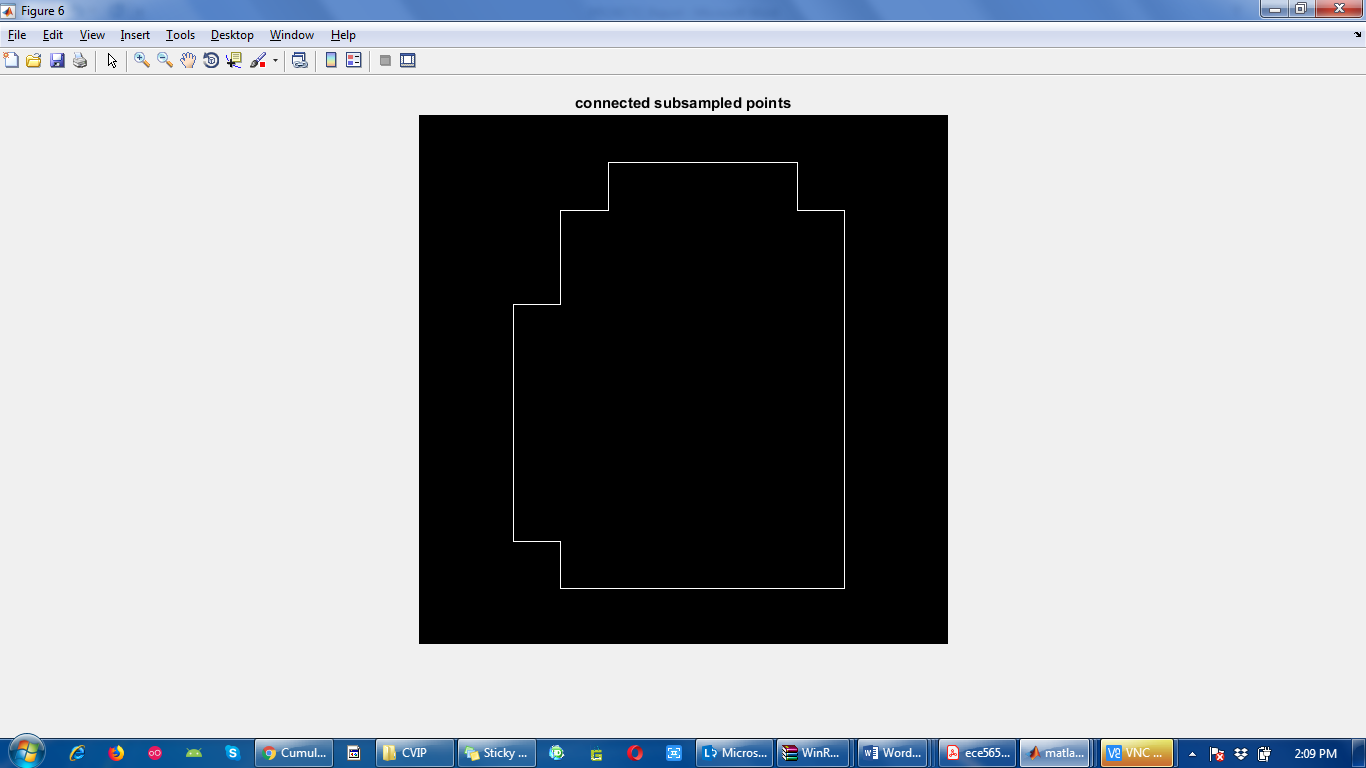












x0y0: [8 3]

fcc: [2 2 2 0 2 2 0 2 0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2]

diff: [0 0 6 2 0 6 2 6 0 0 0 6 2 6 0 0 0 0 0 0 0 6 0 0 0 0 0 6 2 6 0 0]

mm: [0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2 2 2 2 0 2 2 0 2]

diffmm: [0 0 0 6 2 6 0 0 0 0 0 0 0 6 0 0 0 0 0 6 2 6 0 0 0 0 6 2 0 6 2 6]

Q4:

(30 points) **Fourier descriptors**

Figure 4 shows a binary image **f** that shows a human chromosome.

(a) (2 points) Extract the boundary of the chromosome and display the result as a binary image.

(b) (14 points) Write a program to compute the Fourier descriptors of a boundary **s** (i.e., **z = fourierdescp(s)**). The input **s** is an np x 2 sequence of ordered coordinates describing a boundary and the output **z** is a sequence of Fourier descriptors obtained. Compute the Fourier descriptors for the boundary obtained in Part (a).

(c) (14 points) Write a program to compute the inverse Fourier descriptors (i.e., **s = ifourierdescp(z, nd)**). The input **z** is a sequence of Fourier descriptors and **nd** is the number of descriptors used to compute the inverse. **nd** must be an even integer no greater than length(**z**). Reconstruct the boundary using 50% of the total possible descriptors and display the result as a binary image. Then, reconstruct the boundary using 1% of the total possible descriptors and display the result as a binary image.

%Q4 (a)

clc;

clear all;

img = imread('D:\IITCSem2\CVIP\chromosome.tif');

imshow(img); title('Input image');

[row col] = size(img);

b1=bwboundaries(img, 'noholes');

b = b1{1,1};

g=bound2im(b, row, col); % Boundary extraction

figure(); imshow(g); title('Extracted boundary of chromosome as binary image');

%Q4 (b) Computing Fourier descriptors

z=fourierdescp(b);

%Q4 (c) Reconstruct the boundary using a part of the Fourier descriptors

nd\_50 = round(length(z)/2); % for 50% of total fourier descriptors

if(mod(nd\_50,2)) ~= 0 % to check if nd\_50 is even integer

nd\_50 = nd\_50 - 1;

end

nd\_1 = round(length(z)/100); % for 1% of total fourier descriptors

if(mod(nd\_1,2)) ~= 0 % to check if nd\_1 is even integer

nd\_1 = nd\_1 - 1;

end

s1=ifourierdescp(z, nd\_50); % Reconstruction using 50% of total descriptors

s1im=bound2im(s1, row, col);

figure(); imshow(s1im); title('Using 50% fourier descriptors');

s2=ifourierdescp(z, nd\_1); % Reconstruction using 1% of total descriptors

s2im=bound2im(s2, row, col);

figure(); imshow(s2im); title('Using 1% fourier descriptors');

Functions:

function z = fourierdescp(s)

%FRDESCP Computes Fourier descriptors.

% Z = FOURIERDESCP(S) computes the Fourier descriptors of S, which is an

% np-by-2 sequence of image coordinates describing a boundary.

%

% Preliminaries

[np, nc] = size(s);

if nc ~= 2

error('S must be of size np-by-2.');

end

if np/2 ~= round(np/2);

s(end + 1, :) = s(end, :);

np = np + 1;

end

% Create an alternating sequence of 1s and -1s for use in centering

% the transform.

x = 0:(np - 1);

m = ((-1) .^ x)';

% Multiply the input sequence by alternating 1s and -1s to

% center the transform.

s(:, 1) = m .\* s(:, 1);

s(:, 2) = m .\* s(:, 2);

% Convert coordinates to complex numbers.

s = s(:, 1) + i\*s(:, 2);

% Compute the descriptors.

z = fft(s);

function s = ifourierdescp(z, nd)

%IFRDESCP Computes inverse Fourier descriptors.

% S = IFOURIERDESCP(Z, ND) computes the inverse Fourier descriptors of

% of Z, which is a sequence of Fourier descriptor obtained, for

% example, by using function FOURIERDESCP. ND is the number of

% descriptors used to computing the inverse; ND must be an even

% integer no greater than length(Z). If ND is omitted, it defaults

% to length(Z). The output, S, is a length(Z)-by-2 matrix containing

% the coordinates of a closed boundary.

% Preliminaries.

np = length(z);

% Check inputs.

if nargin == 1 | nd > np

nd = np;

end

% Create an alternating sequence of 1s and -1s for use in centering

% the transform.

x = 0:(np - 1);

m = ((-1) .^ x)';

% Use only nd descriptors in the inverse. Since the

% descriptors are centered, (np - nd)/2 terms from each end of

% the sequence are set to 0.

d = round((np - nd)/2); % Round in case nd is odd.

z(1:d) = 0;

z(np - d + 1:np) = 0;

% Compute the inverse and convert back to coordinates.

zz = ifft(z);

s(:, 1) = real(zz);

s(:, 2) = imag(zz);

% Multiply by alternating 1 and -1s to undo the earlier

% centering.

s(:, 1) = m.\*s(:, 1);

s(:, 2) = m.\*s(:, 2);

Output:

