ECE565 Computer Vision and Image Processing (Spring 2019)
Project 2

(Due: April 25th, 2019)

1. (10 points) **Global thresholding**

   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.

Figure 1. noisy_fingerprint.tiff.

2. (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).

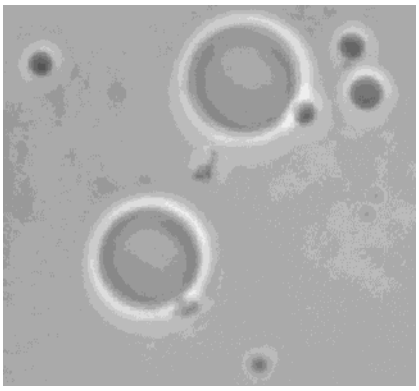

Figure 2. polymersomes.tiff.

3. (40 points) **Chain codes**

Figure 3 shows a 570×570 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.

Figure 3. circular_stroke.tiff.

(a) (2 points) Generate a smoothed image **g** using 9×9 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 ($1 \times np$ where np is the number of boundary pixels)

c.diff = First difference of code c.fcc ($1 \times np$)

c.mm = Integer of minimum magnitude from c.fcc ($1 \times np$)

c.diffmm = First difference of code c.mm ($1 \times np$)

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

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).

4. (30 points) **Fourier descriptors**

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

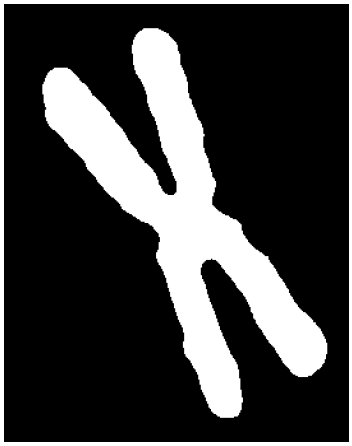

Figure 4. chromosome.tiff.

(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 $np \times 2$ 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.

Appendix

```
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 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 [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)).

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
```