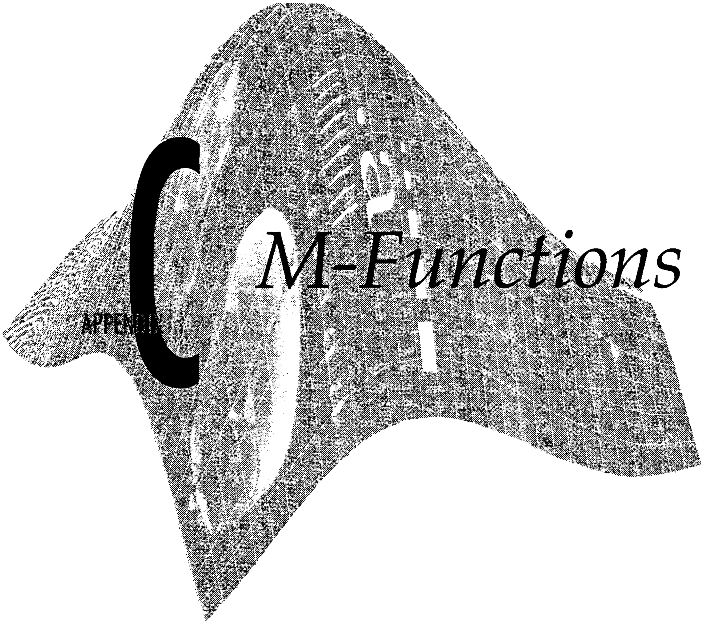

Appendix C M-Functions

Preview

This appendix contains a listing of all the M-functions that are not listed earlier in the book. The functions are organized alphabetically. The first two lines of each function are typed in bold letters as a visual cue to facilitate finding the function and reading its summary description.

A

```
function f = adpmedian(g, Smax)
%ADPMEDIAN Perform adaptive median filtering.
%   F = ADPMEDIAN(G, SMAX) performs adaptive median filtering of
%   image G.  The median filter starts at size 3-by-3 and iterates up
%   to size SMAX-by-SMAX.  SMAX must be an odd integer greater than 1.

% SMAX must be an odd, positive integer greater than 1.
if (Smax <= 1) | (Smax/2 == round(Smax/2)) | (Smax ~= round(Smax))
   error('SMAX must be an odd integer > 1.')
end
[M, N] = size(g);

% Initial setup.
f = g;
f(:) = 0;
alreadyProcessed = false(size(g));

% Begin filtering.
for k = 3:2:Smax
   zmin = ordfilt2(g, 1, ones(k, k), 'symmetric');
   zmax = ordfilt2(g, k * k, ones(k, k), 'symmetric');
   zmed = medfilt2(g, [k k], 'symmetric');
```

```
   processUsingLevelB = (zmed > zmin) & (zmax > zmed) & ...
      ~alreadyProcessed;
   zB = (g > zmin) & (zmax > g);
   outputZxy  = processUsingLevelB & zB;
   outputZmed = processUsingLevelB & ~zB;
   f(outputZxy) = g(outputZxy);
   f(outputZmed) = zmed(outputZmed);

   alreadyProcessed = alreadyProcessed | processUsingLevelB;
   if all(alreadyProcessed(:))
      break;
   end
end

% Output zmed for any remaining unprocessed pixels.  Note that this
% zmed was computed using a window of size Smax-by-Smax, which is
% the final value of k in the loop.
f(~alreadyProcessed) = zmed(~alreadyProcessed);
```

B

```
function rc_new = bound2eight(rc)
%BOUND2EIGHT Convert 4-connected boundary to 8-connected boundary.
%   RC_NEW = BOUND2EIGHT(RC) converts a four-connected boundary to an
%   eight-connected boundary.  RC is a P-by-2 matrix, each row of
%   which contains the row and column coordinates of a boundary
%   pixel.  RC must be a closed boundary; in other words, the last
%   row of RC must equal the first row of RC.  BOUND2EIGHT removes
%   boundary pixels that are necessary for four-connectedness but not
%   necessary for eight-connectedness.  RC_NEW is a Q-by-2 matrix,
%   where Q <= P.

if ~isempty(rc) & ~isequal(rc(1, :), rc(end, :))
   error('Expected input boundary to be closed.');
end

if size(rc, 1) <= 3
   % Degenerate case.
   rc_new = rc;
   return;
end

% Remove last row, which equals the first row.
rc_new = rc(1:end - 1, :);

% Remove the middle pixel in four-connected right-angle turns.  We
% can do this in a vectorized fashion, but we can't do it all at
% once. Similar to the way the 'thin' algorithm works in bwmorph,
% we'll remove first the middle pixels in four-connected turns where
% the row and column are both even; then the middle pixels in all
% the remaining four-connected turns where the row is even and the
% column is odd; then again where the row is odd and the column is
% even; and finally where both the row and column are odd.
```

```
remove_locations = compute_remove_locations(rc_new);
field1 = remove_locations & (rem(rc_new(:, 1), 2) == 0) & ...
         (rem(rc_new(:, 2), 2) == 0);
rc_new(field1, :) = [];

remove_locations = compute_remove_locations(rc_new);
field2 = remove_locations & (rem(rc_new(:, 1), 2) == 0) & ...
         (rem(rc_new(:, 2), 2) == 1);
rc_new(field2, :) = [];

remove_locations = compute_remove_locations(rc_new);
field3 = remove_locations & (rem(rc_new(:, 1), 2) == 1) & ...
         (rem(rc_new(:, 2), 2) == 0);
rc_new(field3, :) = [];

remove_locations = compute_remove_locations(rc_new);
field4 = remove_locations & (rem(rc_new(:, 1), 2) == 1) & ...
         (rem(rc_new(:, 2), 2) == 1);
rc_new(field4, :) = [];

% Make the output boundary closed again.
rc_new = [rc_new; rc_new(1, :)];

%-------------------------------------------------------------------%
function remove = compute_remove_locations(rc)

% Circular diff.
d = [rc(2:end, :); rc(1, :)] - rc;

% Dot product of each row of d with the subsequent row of d,
% performed in circular fashion.
d1 = [d(2:end, :); d(1, :)];
dotprod = sum(d .* d1, 2);

% Locations of N, S, E, and W transitions followed by
% a right-angle turn.
remove = ~all(d, 2) & (dotprod == 0);

% But we really want to remove the middle pixel of the turn.
remove = [remove(end, :); remove(1:end - 1, :)];

if ~any(remove)
   done = 1;
else
   idx = find(remove);
   rc(idx(1), :) = [];
end
```

```
function rc_new = bound2four(rc)
%BOUND2FOUR Convert 8-connected boundary to 4-connected boundary.
%   RC_NEW = BOUND2FOUR(RC) converts an eight-connected boundary to a
%   four-connected boundary.  RC is a P-by-2 matrix, each row of
%   which contains the row and column coordinates of a boundary
%   pixel.  BOUND2FOUR inserts new boundary pixels wherever there is
%   a diagonal connection.

if size(rc, 1) > 1
   % Phase 1: remove diagonal turns, one at a time until they are all gone.
```

```
    done = 0;
    rc1 = [rc(end - 1, :); rc];
    while ~done
       d = diff(rc1, 1);
       diagonal_locations = all(d, 2);
       double_diagonals = diagonal_locations(1:end - 1) & ...
           (diff(diagonal_locations, 1) == 0);
       double_diagonal_idx = find(double_diagonals);
       turns = any(d(double_diagonal_idx, :) ~= ...
                   d(double_diagonal_idx + 1, :), 2);
       turns_idx = double_diagonal_idx(turns);
       if isempty(turns_idx)
          done = 1;
       else
          first_turn = turns_idx(1);
          rc1(first_turn + 1, :) = (rc1(first_turn, :) + ...
                                    rc1(first_turn + 2, :)) / 2;
          if first_turn == 1
             rc1(end, :) = rc1(2, :);
          end
       end
    end
    rc1 = rc1(2:end, :);
end

% Phase 2: insert extra pixels where there are diagonal connections.

rowdiff = diff(rc1(:, 1));
coldiff = diff(rc1(:, 2));

diagonal_locations = rowdiff & coldiff;
num_old_pixels = size(rc1, 1);
num_new_pixels = num_old_pixels + sum(diagonal_locations);
rc_new = zeros(num_new_pixels, 2);

% Insert the original values into the proper locations in the new RC
% matrix.
idx = (1:num_old_pixels)' + [0; cumsum(diagonal_locations)];
rc_new(idx, :) = rc1;

% Compute the new pixels to be inserted.
new_pixel_offsets = [0 1; -1 0; 1 0; 0 -1];
offset_codes = 2 * (1 - (coldiff(diagonal_locations) + 1)/2) + ...
   (2 - (rowdiff(diagonal_locations) + 1)/2);
new_pixels = rc1(diagonal_locations, :) + ...
   new_pixel_offsets(offset_codes, :);

% Where do the new pixels go?
insertion_locations = zeros(num_new_pixels, 1);
insertion_locations(idx) = 1;
insertion_locations = ~insertion_locations;

% Insert the new pixels.
rc_new(insertion_locations, :) = new_pixels;
```

```
function B = bound2im(b, M, N, x0, y0)
%BOUND2IM Converts a boundary to an image.
%   B = BOUND2IM(b) converts b, an np-by-2 or 2-by-np array
%   representing the integer coordinates of a boundary, into a binary
%   image with 1s in the locations defined by the coordinates in b
%   and 0s elsewhere.
%
%   B = BOUND2IM(b, M, N) places the boundary approximately centered
%   in an M-by-N image. If any part of the boundary is outside the
%   M-by-N rectangle, an error is issued.
%
%   B = BOUND2IM(b, M, N, X0, Y0) places the boundary in an image of
%   size M-by-N, with the topmost boundary point located at X0 and
%   the leftmost point located at Y0. If the shifted boundary is
%   outside the M-by-N rectangle, an error is issued. X0 and X0 must
%   be positive integers.

[np, nc] = size(b);
if np < nc
   b = b'; % To convert to size np-by-2.
   [np, nc] = size(b);
end

% Make sure the coordinates are integers.
x = round(b(:, 1));
y = round(b(:, 2));

% Set up the default size parameters.
x = x - min(x) + 1;
y = y - min(y) + 1;
B = false(max(x), max(y));
C = max(x) - min(x) + 1;
D = max(y) - min(y) + 1;

if nargin == 1
   % Use the preceding default values.
elseif nargin == 3
   if C > M | D > N
      error('The boundary is outside the M-by-N region.')
   end
   % The image size will be M-by-N. Set up the parameters for this.
   B = false(M, N);
   % Distribute extra rows approx. even between top and bottom.
   NR = round((M - C)/2);
   NC = round((N - D)/2); % The same for columns.
   x = x + NR; % Offset the boundary to new position.
   y = y + NC;
elseif nargin == 5
   if x0 < 0 | y0 < 0
      error('x0 and y0 must be positive integers.')
   end
   x = x + round(x0) - 1;
   y = y + round(y0) - 1;
```

```
    C = C + x0 - 1;
    D = D + y0 - 1;
    if C > M | D > N
       error('The shifted boundary is outside the M-by-N region.')
    end
    B = false(M, N);
else
    error('Incorrect number of inputs.')
end

B(sub2ind(size(B), x, y)) = true;
```

```
function B = boundaries(BW, conn, dir)
%BOUNDARIES Trace object boundaries.
%   B = BOUNDARIES(BW) traces the exterior boundaries of objects in
%   the binary image BW.  B is a P-by-1 cell array, where P is the
%   number of objects in the image. Each cell contains a Q-by-2
%   matrix, each row of which contains the row and column coordinates
%   of a boundary pixel.  Q is the number of boundary pixels for the
%   corresponding object.  Object boundaries are traced in the
%   clockwise direction.
%
%   B = BOUNDARIES(BW, CONN) specifies the connectivity to use when
%   tracing boundaries.  CONN may be either 8 or 4.  The default
%   value for CONN is 8.
%
%   B = BOUNDARIES(BW, CONN, DIR) specifies the direction used for
%   tracing boundaries.  DIR should be either 'cw' (trace boundaries
%   clockwise) or 'ccw' (trace boundaries counterclockwise).  If DIR
%   is omitted BOUNDARIES traces in the clockwise direction.

if nargin < 3
   dir = 'cw';
end

if nargin < 2
   conn = 8;
end

L = bwlabel(BW, conn);

% The number of objects is the maximum value of L.  Initialize the
% cell array B so that each cell initially contains a 0-by-2 matrix.
numObjects = max(L(:));
if numObjects > 0
   B = {zeros(0, 2)};
   B = repmat(B, numObjects, 1);
else
   B = {};
end

% Pad label matrix with zeros.  This lets us write the
% boundary-following loop without worrying about going off the edge
% of the image.
Lp = padarray(L, [1 1], 0, 'both');
```

```
% Compute the linear indexing offsets to take us from a pixel to its
% neighbors.
M = size(Lp, 1);
if conn == 8
   % Order is N NE E SE S SW W NW.
   offsets = [-1, M - 1, M, M + 1, 1, -M + 1, -M, -M-1];
else
   % Order is N E S W.
   offsets = [-1, M, 1, -M];
end

% next_search_direction_lut is a lookup table.  Given the direction
% from pixel k to pixel k+1, what is the direction to start with when
% examining the neighborhood of pixel k+1?
if conn == 8
   next_search_direction_lut = [8 8 2 2 4 4 6 6];
else
   next_search_direction_lut = [4 1 2 3];
end

% next_direction_lut is a lookup table.  Given that we just looked at
% neighbor in a given direction, which neighbor do we look at next?
if conn == 8
   next_direction_lut = [2 3 4 5 6 7 8 1];
else
   next_direction_lut = [2 3 4 1];
end

% Values used for marking the starting and boundary pixels.
START    = -1;
BOUNDARY = -2;

% Initialize scratch space in which to record the boundary pixels as
% well as follow the boundary.
scratch = zeros(100, 1);

% Find candidate starting locations for boundaries.
[rr, cc] = find((Lp(2:end-1, :) > 0) & (Lp(1:end-2, :) == 0));
rr = rr + 1;

for k = 1:length(rr)
   r = rr(k);
   c = cc(k);
   if (Lp(r,c) > 0) & (Lp(r - 1, c) == 0) & isempty(B{Lp(r, c)})
      % We've found the start of the next boundary.  Compute its
      % linear offset, record which boundary it is, mark it, and
      % initialize the counter for the number of boundary pixels.
      idx = (c-1)*size(Lp, 1) + r;
      which = Lp(idx);

      scratch(1) = idx;
      Lp(idx) = START;
      numPixels = 1;
      currentPixel = idx;
      initial_departure_direction = [];
```

```
done = 0;
next_search_direction = 2;
while ~done
   % Find the next boundary pixel.
   direction = next_search_direction;
   found_next_pixel = 0;
   for k = 1:length(offsets)
      neighbor = currentPixel + offsets(direction);
      if Lp(neighbor) ~= 0
         % Found the next boundary pixel.

         if (Lp(currentPixel) == START) & ...
                isempty(initial_departure_direction)
            % We are making the initial departure from
            % the starting pixel.
            initial_departure_direction = direction;

         elseif (Lp(currentPixel) == START) & ...
                (initial_departure_direction == direction)
            % We are about to retrace our path.
            % That means we're done.
            done = 1;
            found_next_pixel = 1;
            break;
         end

         % Take the next step along the boundary.
         next_search_direction = ...
             next_search_direction_lut(direction);
         found_next_pixel = 1;
         numPixels = numPixels + 1;
         if numPixels > size(scratch, 1)
            % Double the scratch space.
            scratch(2*size(scratch, 1)) = 0;
         end
         scratch(numPixels) = neighbor;

         if Lp(neighbor) ~= START
            Lp(neighbor) = BOUNDARY;
         end

         currentPixel = neighbor;
         break;
      end

      direction = next_direction_lut(direction);
   end

   if ~found_next_pixel
      % If there is no next neighbor, the object must just
      % have a single pixel.
      numPixels = 2;
      scratch(2) = scratch(1);
      done = 1;
```

```
            end
        end

        % Convert linear indices to row-column coordinates and save
        % in the output cell array.
        [row, col] = ind2sub(size(Lp), scratch(1:numPixels));
        B{which} = [row - 1, col - 1];
    end
end

if strcmp(dir, 'ccw')
    for k = 1:length(B)
        B{k} = B{k}(end:-1:1, :);
    end
end
```

```
function [s, su] = bsubsamp(b, gridsep)
%BSUBSAMP Subsample a boundary.
%   [S, SU] = 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 SU is normalized so
%   that the grid separation is unity.  This is useful for obtaining
%   the Freeman chain code of the subsampled boundary.

% Check input.
[np, nc] = size(b);
if np < nc
    error('B must be of size np-by-2.');
end
if gridsep/2 ~= round(gridsep/2)
    error('GRIDSEP must be an even integer.')
end

% Some boundary tracing programs, such as boundaries.m, end with
% the beginning, resulting in a sequence in which the coordinates
% of the first and last points are the same. If this is the case
% in b, eliminate the last point.
if isequal(b(1, :), b(np, :))
    np = np - 1;
    b = b(1:np, :);
end

% Find the max x and y spanned by the boundary.
xmax = max(b(:, 1));
ymax = max(b(:, 2));
```

```
% Determine the number of grid lines with gridsep points in
% between them that can fit in the intervals [1,xmax], [1,ymax],
% without any points in b being left over. If points are left
% over, add zeros to extend xmax and ymax so that an integral
% number of grid lines are obtained.
% Size needed in the x-direction:
L = gridsep + 1;
n = ceil(xmax/L);
T = (n - 1)*L + 1;

% Zx is the number of zeros that would be needed to have grid
% lines without any points in b being left over.
Zx = abs(xmax - T - L);
if Zx == L
   Zx = 0;
end
% Number of grid lines in the x-direction, with L pixel spaces
% in between each grid line.
GLx = (xmax + Zx - 1)/L + 1;

% And for the y-direction:
n = ceil(ymax/L);
T = (n - 1)*L + 1;
Zy = abs(ymax - T - L);
if Zy == L
   Zy = 0;
end
GLy = (ymax + Zy - 1)/L + 1;

% Form vectors of x and y grid locations.
I = 1:GLx;
% Vector of grid line locations intersecting x-axis.
X(I) = gridsep*I + (I - gridsep);

J = 1:GLy;
% Vector of grid line locations intersecting y-axis.
Y(J) = gridsep*J + (J - gridsep);

% Compute both components of the cityblock distance between each
% element of b and all the grid-line intersections.  Assign each
% point to the grid location for which each comp of the cityblock
% distance was less than gridsep/2. Because gridsep is an even
% integer, these assignments are unique. Note the use of meshgrid to
% optimize the code.
DIST = gridsep/2;
[XG, YG] = meshgrid(X, Y);
Q = 1;
for k=1:np
   [I,J] = find(abs(XG - b(k, 1)) <= DIST & abs(YG - b(k, 2)) <= ...
                DIST);
   IL = length(I);
   ord = k*ones(IL, 1); % To keep track of order of input coordinates
```

```
   K = Q + IL – 1;
   d1(Q:K, :) = cat(2, X(I), ord);
   d2(Q:K, :) = cat(2, Y(J), ord);
   Q = K + 1;
end

% d is the set of points assigned to the new grid with line
% separation of gridsep. Note that it is formed as d=(d2,d1) to
% compensate for the coordinate transposition inherent in using
% meshgrid (see Chapter 2).
d = cat(2, d2(:, 1), d1); % The second column of d1 is ord.

% Sort the points using the values in ord, which is the last col in
% d.
d = fliplr(d); % So the last column becomes first.
d = sortrows(d);
d = fliplr(d); % Flip back.

% Eliminate duplicate rows in the first two components of
% d to create the output. The cw or ccw order MUST be preserved.
s = d(:, 1:2);
[s, m, n] = unique(s, 'rows');

% Function unique sorts the data--Restore to original order
% by using the contents of m.
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 code.  The shape does not change.
su = round(s./gridsep) + 1;
```

C

```
function image = changeclass(class, varargin)
%CHANGECLASS changes the storage class of an image.
%   I2 = CHANGECLASS(CLASS, I);
%   RGB2 = CHANGECLASS(CLASS, RGB);
%   BW2 = CHANGECLASS(CLASS, BW);
%   X2 = CHANGECLASS(CLASS, X, 'indexed');

%   Copyright 1993–2002 The MathWorks, Inc.  Used with permission.
%   $Revision: 1.2 $  $Date: 2003/02/19 22:09:58 $

switch class
case 'uint8'
   image = im2uint8(varargin{:});
case 'uint16'
   image = im2uint16(varargin{:});
```

```
case 'double'
   image = im2double(varargin{:});
otherwise
   error('Unsupported IPT data class.');
end
```

```
function [VG, A, PPG]= colorgrad(f, T)
%COLORGRAD Computes the vector gradient of an RGB image.
%   [VG, VA, PPG] = COLORGRAD(F, T) computes the vector gradient, VG,
%   and corresponding angle array, VA, (in radians) of RGB image
%   F. It also computes PPG, the per-plane composite gradient
%   obtained by summing the 2-D gradients of the individual color
%   planes. Input T is a threshold in the range [0, 1]. If it is
%   included in the argument list, the values of VG and PPG are
%   thresholded by letting VG(x,y) = 0 for values <= T and VG(x,y) =
%   VG(x,y) otherwise. Similar comments apply to PPG.  If T is not
%   included in the argument list then T is set to 0. Both output
%   gradients are scaled to the range [0, 1].

if (ndims(f) ~= 3) | (size(f, 3) ~= 3)
   error('Input image must be RGB.');
end

% Compute the x and y derivatives of the three component images
% using Sobel operators.
sh = fspecial('sobel');
sv = sh';
Rx = imfilter(double(f(:, :, 1)), sh, 'replicate');
Ry = imfilter(double(f(:, :, 1)), sv, 'replicate');
Gx = imfilter(double(f(:, :, 2)), sh, 'replicate');
Gy = imfilter(double(f(:, :, 2)), sv, 'replicate');
Bx = imfilter(double(f(:, :, 3)), sh, 'replicate');
By = imfilter(double(f(:, :, 3)), sv, 'replicate');

% Compute the parameters of the vector gradient.
gxx = Rx.^2 + Gx.^2 + Bx.^2;
gyy = Ry.^2 + Gy.^2 + By.^2;
gxy = Rx.*Ry + Gx.*Gy + Bx.*By;
A = 0.5*(atan(2*gxy./(gxx - gyy + eps)));
G1 = 0.5*((gxx + gyy) + (gxx - gyy).*cos(2*A) + 2*gxy.*sin(2*A));

% Now repeat for angle + pi/2.  Then select the maximum at each point.
A = A + pi/2;
G2 = 0.5*((gxx + gyy) + (gxx - gyy).*cos(2*A) + 2*gxy.*sin(2*A));
G1 = G1.^0.5;
G2 = G2.^0.5;
% Form VG by picking the maximum at each (x,y) and then scale
% to the range [0, 1].
VG = mat2gray(max(G1, G2));

% Compute the per-plane gradients.
RG = sqrt(Rx.^2 + Ry.^2);
GG = sqrt(Gx.^2 + Gy.^2);
```

```
BG = sqrt(Bx.^2 + By.^2);
% Form the composite by adding the individual results and
% scale to [0, 1].
PPG = mat2gray(RG + GG + BG);

% Threshold the result.
if nargin == 2
   VG = (VG > T).*VG;
   PPG = (PPG > T).*PPG;
end
```

```
function I = colorseg(varargin)
%COLORSEG Performs segmentation of a color image.
%   S = COLORSEG('EUCLIDEAN', F, T, M) performs segmentation of color
%   image F using a Euclidean measure of similarity. M is a 1-by-3
%   vector representing the average color used for segmentation (this
%   is the center of the sphere in Fig. 6.26 of DIPUM). T is the
%   threshold against which the distances are compared.
%
%   S = COLORSEG('MAHALANOBIS', F, T, M, C) performs segmentation of
%   color image F using the Mahalanobis distance as a measure of
%   similarity. C is the 3-by-3 covariance matrix of the sample color
%   vectors of the class of interest. See function covmatrix for the
%   computation of C and M.
%
%   S is the segmented image (a binary matrix) in which 0s denote the
%   background.

% Preliminaries.
% Recall that varargin is a cell array.
f = varargin{2};
if (ndims(f) ~= 3) | (size(f, 3) ~= 3)
   error('Input image must be RGB.');
end
M = size(f, 1); N = size(f, 2);
% Convert f to vector format using function imstack2vectors.
[f, L] = imstack2vectors(f);
f = double(f);
% Initialize I as a column vector.  It will be reshaped later
% into an image.
I = zeros(M*N, 1);
T = varargin{3};
m = varargin{4};
m = m(:)'; % Make sure that m is a row vector.

if length(varargin) == 4
   method = 'euclidean';
elseif length(varargin) == 5
   method = 'mahalanobis';
else
   error('Wrong number of inputs.');
end
```

```
switch method
case 'euclidean'
   % Compute the Euclidean distance between all rows of X and m. See
   % Section 12.2 of DIPUM for an explanation of the following
   % expression. D(i) is the Euclidean distance between vector X(i,:)
   % and vector m.
   p = length(f);
   D = sqrt(sum(abs(f - repmat(m, p, 1)).^2, 2));
case 'mahalanobis'
   C = varargin{5};
   D = mahalanobis(f, C, m);
otherwise
   error('Unknown segmentation method.')
end

% D is a vector of size MN-by-1 containing the distance computations
% from all the color pixels to vector m. Find the distances <= T.
J = find(D <= T);

% Set the values of I(J) to 1.  These are the segmented
% color pixels.
I(J) = 1;

% Reshape I into an M-by-N image.
I = reshape(I, M, N);
```

```
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 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{:});
```

D

```
function s = diameter(L)
%DIAMETER Measure diameter and related properties of image regions.
%   S = DIAMETER(L) computes the diameter, the major axis endpoints,
```

```
%   the minor axis endpoints, and the basic rectangle of each labeled
%   region in the label matrix L. Positive integer elements of L
%   correspond to different regions. For example, the set of elements
%   of L equal to 1 corresponds to region 1; the set of elements of L
%   equal to 2 corresponds to region 2; and so on.  S is a structure
%   array of length max(L(:)). The fields of the structure array
%   include:
%
%     Diameter
%     MajorAxis
%     MinorAxis
%     BasicRectangle
%
%   The Diameter field, a scalar, is the maximum distance between any
%   two pixels in the corresponding region.
%
%   The MajorAxis field is a 2-by-2 matrix.  The rows contain the row
%   and column coordinates for the endpoints of the major axis of the
%   corresponding region.
%
%   The MinorAxis field is a 2-by-2 matrix.  The rows contain the row
%   and column coordinates for the endpoints of the minor axis of the
%   corresponding region.
%
%   The BasicRectangle field is a 4-by-2 matrix.  Each row contains
%   the row and column coordinates of a corner of the
%   region-enclosing rectangle defined by the major and minor axes.
%
%   For more information about these measurements, see Section 11.2.1
%   of Digital Image Processing, by Gonzalez and Woods, 2nd edition,
%   Prentice Hall.

s = regionprops(L, {'Image', 'BoundingBox'});

for k = 1:length(s)
    [s(k).Diameter, s(k).MajorAxis, perim_r, perim_c] = ...
        compute_diameter(s(k));
    [s(k).BasicRectangle, s(k).MinorAxis] = ...
        compute_basic_rectangle(s(k), perim_r, perim_c);
end

%-------------------------------------------------------------------%
function [d, majoraxis, r, c] = compute_diameter(s)
%   [D, MAJORAXIS, R, C] = COMPUTE_DIAMETER(S) computes the diameter
%   and major axis for the region represented by the structure S. S
%   must contain the fields Image and BoundingBox.  COMPUTE_DIAMETER
%   also returns the row and column coordinates (R and C) of the
%   perimeter pixels of s.Image.

% Compute row and column coordinates of perimeter pixels.
[r, c] = find(bwperim(s.Image));
r = r(:);
```

```
c = c(:);
[rp, cp] = prune_pixel_list(r, c);

num_pixels = length(rp);
switch num_pixels
case 0
   d = -Inf;
   majoraxis = ones(2, 2);

case 1
   d = 0;
   majoraxis = [rp cp; rp cp];

case 2
   d = (rp(2) - rp(1))^2 + (cp(2) - cp(1))^2;
   majoraxis = [rp cp];

otherwise
   % Generate all combinations of 1:num_pixels taken two at at time.
   % Method suggested by Peter Acklam.
   [idx(:, 2) idx(:, 1)] = find(tril(ones(num_pixels), -1));
   rr = rp(idx);
   cc = cp(idx);

   dist_squared = (rr(:, 1) - rr(:, 2)).^2 + ...
      (cc(:, 1) - cc(:, 2)).^2;
   [max_dist_squared, idx] = max(dist_squared);
   majoraxis = [rr(idx,:)' cc(idx,:)'];

   d = sqrt(max_dist_squared);

   upper_image_row = s.BoundingBox(2) + 0.5;
   left_image_col = s.BoundingBox(1) + 0.5;

   majoraxis(:, 1) = majoraxis(:, 1) + upper_image_row - 1;
   majoraxis(:, 2) = majoraxis(:, 2) + left_image_col - 1;
end

%-----------------------------------------------------------------%
function [basicrect, minoraxis] = compute_basic_rectangle(s, ...
                                                 perim_r, perim_c)
%   [BASICRECT,MINORAXIS] = COMPUTE_BASIC_RECTANGLE(S, PERIM_R,
%   PERIM_C) computes the basic rectangle and the minor axis
%   end-points for the region represented by the structure S.  S must
%   contain the fields Image, BoundingBox, MajorAxis, and
%   Diameter. PERIM_R and PERIM_C are the row and column coordinates
%   of perimeter of s.Image. BASICRECT is a 4-by-2 matrix, each row
%   of which contains the row and column coordinates of one corner of
%   the basic rectangle.

% Compute the orientation of the major axis.
theta = atan2(s.MajorAxis(2, 1) - s.MajorAxis(1, 1), ...
              s.MajorAxis(2, 2) - s.MajorAxis(1, 2));

% Form rotation matrix.
T = [cos(theta) sin(theta); -sin(theta) cos(theta)];
```

```
% Rotate perimeter pixels.
p = [perim_c perim_r];
p = p * T';

% Calculate minimum and maximum x- and y-coordinates for the rotated
% perimeter pixels.
x = p(:, 1);
y = p(:, 2);
min_x = min(x);
max_x = max(x);
min_y = min(y);
max_y = max(y);

corners_x = [min_x max_x max_x min_x]';
corners_y = [min_y min_y max_y max_y]';

% Rotate corners of the basic rectangle.
corners = [corners_x corners_y] * T;

% Translate according to the region's bounding box.
upper_image_row = s.BoundingBox(2) + 0.5;
left_image_col = s.BoundingBox(1) + 0.5;

basicrect = [corners(:, 2) + upper_image_row - 1, ...
             corners(:, 1) + left_image_col - 1];

% Compute minor axis end-points, rotated.
x = (min_x + max_x) / 2;
y1 = min_y;
y2 = max_y;
endpoints = [x y1; x y2];

% Rotate minor axis end-points back.
endpoints = endpoints * T;

% Translate according to the region's bounding box.
minoraxis = [endpoints(:, 2) + upper_image_row - 1, ...
             endpoints(:, 1) + left_image_col - 1];

%---------------------------------------------------------------------%
function [r, c] = prune_pixel_list(r, c)
%   [R, C] = PRUNE_PIXEL_LIST(R, C) removes pixels from the vectors
%   R and C that cannot be endpoints of the major axis.  This
%   elimination is based on geometrical constraints described in
%   Russ, Image Processing Handbook, Chapter 8.

top = min(r);
bottom = max(r);
left = min(c);
right = max(c);

% Which points are inside the upper circle?
x = (left + right)/2;
y = top;
radius = bottom - top;
inside_upper = ( (c - x).^2 + (r - y).^2 ) < radius^2;
```

```
% Which points are inside the lower circle?
y = bottom;
inside_lower = ( (c - x).^2 + (r - y).^2 ) < radius^2;

% Which points are inside the left circle?
x = left;
y = (top + bottom)/2;
radius = right - left;
inside_left = ( (c - x).^2 + (r - y).^2 ) < radius^2;

% Which points are inside the right circle?
x = right;
inside_right = ( (c - x).^2 + (r - y).^2 ) < radius^2;

% Eliminate points that are inside all four circles.
delete_idx = find(inside_left & inside_right & ...
                  inside_upper & inside_lower);
r(delete_idx) = [];
c(delete_idx) = [];
```

F

```
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.

% 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
```

G

```
function g = gscale(f, varargin)
%GSCALE Scales the intensity of the input image.
%   G = GSCALE(F, 'full8') scales the intensities of F to the full
%   8-bit intensity range [0, 255].  This is the default if there is
```

```
%   only one input argument.
%
%   G = GSCALE(F, 'full16') scales the intensities of F to the full
%   16-bit intensity range [0, 65535].
%
%   G = GSCALE(F, 'minmax', LOW, HIGH) scales the intensities of F to
%   the range [LOW, HIGH]. These values must be provided, and they
%   must be in the range [0, 1], independently of the class of the
%   input. GSCALE performs any necessary scaling. If the input is of
%   class double, and its values are not in the range [0, 1], then
%   GSCALE scales it to this range before processing.
%
%   The class of the output is the same as the class of the input.

if length(varargin) == 0 % If only one argument it must be f.
   method = 'full8';
else
   method = varargin{1};
end

if strcmp(class(f), 'double') & (max(f(:)) > 1 | min(f(:)) < 0)
   f = mat2gray(f);
end

% Perform the specified scaling.
switch method
case 'full8'
   g = im2uint8(mat2gray(double(f)));
case 'full16'
   g = im2uint16(mat2gray(double(f)));
case 'minmax'
   low = varargin{2}; high = varargin{3};
   if low > 1 | low < 0 | high > 1 | high < 0
      error('Parameters low and high must be in the range [0, 1].')
   end
   if strcmp(class(f), 'double')
      low_in = min(f(:));
      high_in = max(f(:));
   elseif strcmp(class(f), 'uint8')
      low_in = double(min(f(:)))./255;
      high_in = double(max(f(:)))./255;
   elseif strcmp(class(f), 'uint16')
      low_in = double(min(f(:)))./65535;
      high_in = double(max(f(:)))./65535;
   end
   % imadjust automatically matches the class of the input.
   g = imadjust(f, [low_in high_in], [low high]);
otherwise
   error('Unknown method.')
end
```

I

```
function [X, R] = imstack2vectors(S, MASK)
%IMSTACK2VECTORS Extracts vectors from an image stack.
%   [X, R] = imstack2vectors(S, MASK) extracts vectors from S, which
%   is an M-by-N-by-n stack array of n registered images of size
%   M-by-N each (see Fig. 11.24). The extracted vectors are arranged
%   as the rows of array X. Input MASK is an M-by-N logical or
%   numeric image with nonzero values (1s if it is a logical array)
%   in the locations where elements of S are to be used in forming X
%   and 0s in locations to be ignored. The number of row vectors in X
%   is equal to the number of nonzero elements of MASK. If MASK is
%   omitted, all M*N locations are used in forming X.  A simple way to
%   obtain MASK interactively is to use function roipoly. Finally, R
%   is an array whose rows are the 2-D coordinates containing the
%   region locations in MASK from which the vectors in S were
%   extracted to form X.

% Preliminaries.
[M, N, n] = size(S);
if nargin == 1
   MASK = true(M, N);
else
   MASK = MASK ~= 0;
end

% Find the set of locations where the vectors will be kept before
% MASK is changed later in the program.
[I, J] = find(MASK);
R = [I, J];

% Now find X.

% First reshape S into X by turning each set of n values along the third
% dimension of S so that it becomes a row of X. The order is from top to
% bottom along the first column, the second column, and so on.
Q = M*N;
X = reshape(S, Q, n);

% Now reshape MASK so that it corresponds to the right locations
% vertically along the elements of X.
MASK = reshape(MASK, Q, 1);

% Keep the rows of X at locations where MASK is not 0.
X = X(MASK, :);

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: 5.11 $  $Date: 2002/03/15 15:57:47 $

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 phi = invmoments(F)
%INVMOMENTS Compute invariant moments of image.
%   PHI = INVMOMENTS(F) computes the moment invariants of the image
%   F. PHI is a seven-element row vector containing the moment
%   invariants as defined in equations (11.3-17) through (11.3-23) of
%   Gonzalez and Woods, Digital Image Processing, 2nd Ed.
%
%   F must be a 2-D, real, nonsparse, numeric or logical matrix.

if (ndims(F) ~= 2) | issparse(F) | ~isreal(F) | ~(isnumeric(F) | ...
                                                    islogical(F))
   error(['F must be a 2-D, real, nonsparse, numeric or logical '...
          'matrix.']);
end
```

```
F = double(F);

phi = compute_phi(compute_eta(compute_m(F)));

%-------------------------------------------------------------------%
function m = compute_m(F)

[M, N] = size(F);
[x, y] = meshgrid(1:N, 1:M);

% Turn x, y, and F into column vectors to make the summations a bit
% easier to compute in the following.
x = x(:);
y = y(:);
F = F(:);

% DIP equation (11.3-12)
m.m00 = sum(F);
% Protect against divide-by-zero warnings.
if (m.m00 == 0)
   m.m00 = eps;
end
% The other central moments:
m.m10 = sum(x .* F);
m.m01 = sum(y .* F);
m.m11 = sum(x .* y .* F);
m.m20 = sum(x.^2 .* F);
m.m02 = sum(y.^2 .* F);
m.m30 = sum(x.^3 .* F);
m.m03 = sum(y.^3 .* F);
m.m12 = sum(x .* y.^2 .* F);
m.m21 = sum(x.^2 .* y .* F);

%-------------------------------------------------------------------%
function e = compute_eta(m)

% DIP equations (11.3-14) through (11.3-16).

xbar = m.m10 / m.m00;
ybar = m.m01 / m.m00;

e.eta11 = (m.m11 - ybar*m.m10) / m.m00^2;
e.eta20 = (m.m20 - xbar*m.m10) / m.m00^2;
e.eta02 = (m.m02 - ybar*m.m01) / m.m00^2;
e.eta30 = (m.m30 - 3 * xbar * m.m20 + 2 * xbar^2 * m.m10) / m.m00^2.5;
e.eta03 = (m.m03 - 3 * ybar * m.m02 + 2 * ybar^2 * m.m01) / m.m00^2.5;
e.eta21 = (m.m21 - 2 * xbar * m.m11 - ybar * m.m20 + ...
           2 * xbar^2 * m.m01) / m.m00^2.5;
e.eta12 = (m.m12 - 2 * ybar * m.m11 - xbar * m.m02 + ...
           2 * ybar^2 * m.m10) / m.m00^2.5;

%-------------------------------------------------------------------%
function phi = compute_phi(e)

% DIP equations (11.3-17) through (11.3-23).
```

```
phi(1) = e.eta20 + e.eta02;
phi(2) = (e.eta20 - e.eta02)^2 + 4*e.eta11^2;
phi(3) = (e.eta30 - 3*e.eta12)^2 + (3*e.eta21 - e.eta03)^2;
phi(4) = (e.eta30 + e.eta12)^2 + (e.eta21 + e.eta03)^2;
phi(5) = (e.eta30 - 3*e.eta12) * (e.eta30 + e.eta12) * ...
         ( (e.eta30 + e.eta12)^2 - 3*(e.eta21 + e.eta03)^2 ) + ...
         (3*e.eta21 - e.eta03) * (e.eta21 + e.eta03) * ...
         ( 3*(e.eta30 + e.eta12)^2 - (e.eta21 + e.eta03)^2 );
phi(6) = (e.eta20 - e.eta02) * ( (e.eta30 + e.eta12)^2 - ...
                                 (e.eta21 + e.eta03)^2 ) + ...
         4 * e.eta11 * (e.eta30 + e.eta12) * (e.eta21 + e.eta03);
phi(7) = (3*e.eta21 - e.eta03) * (e.eta30 + e.eta12) * ...
         ( (e.eta30 + e.eta12)^2 - 3*(e.eta21 + e.eta03)^2 ) + ...
         (3*e.eta12 - e.eta30) * (e.eta21 + e.eta03) * ...
         ( 3*(e.eta30 + e.eta12)^2 - (e.eta21 + e.eta03)^2 );
```

M

```
function [x, y] = minperpoly(B, cellsize)
%MINPERPOLY Computes the minimum perimeter polygon.
%   [X, Y] = MINPERPOLY(F, CELLSIZE) computes the vertices in [X, Y]
%   of the minimum perimeter polygon of a single binary region or
%   boundary in image B. The procedure is based on Slansky's
%   shrinking rubber band approach. Parameter CELLSIZE determines the
%   size of the square cells that enclose the boundary of the region
%   in B. CELLSIZE must be a nonzero integer greater than 1.
%
%   The algorithm is applicable only to boundaries that are not
%   self-intersecting and that do not have one-pixel-thick
%   protrusions.

if cellsize <= 1
   error('CELLSIZE must be an integer > 1.');
end

% Fill B in case the input was provided as a boundary. Later
% the boundary will be extracted with 4-connectivity, which
% is required by the algorithm. The use of bwperim assures
% that 4-connectivity is preserved at this point.
B = imfill(B, 'holes');
B = bwperim(B);
[M, N] = size(B);

% Increase image size so that the image is of size K-by-K
% with (a) K >= max(M,N) and (b)  K/cellsize = a power of 2.
K = nextpow2(max(M, N)/cellsize);
K = (2^K)*cellsize;

% Increase image size to nearest integer power of 2, by
% appending zeros to the end of the image. This will allow
% quadtree decompositions as small as cells of size 2-by-2,
```

```
% which is the smallest allowed value of cellsize.
M = K - M;
N = K - N;
B = padarray(B, [M N], 'post'); % f is now of size K-by-K

% Quadtree decomposition.
Q = qtdecomp(B, 0, cellsize);

% Get all the subimages of size cellsize-by-cellsize.
[vals, r, c] = qtgetblk(B, Q, cellsize);

% Get all the subimages that contain at least one black
% pixel. These are the cells of the wall enclosing the boundary.
I = find(sum(sum(vals(:, :, :)) >= 1));
x = r(I);
y = c(I);

% [x', y'] is a length(I)-by-2 array.  Each member of this array is
% the left, top corner of a black cell of size cellsize-by-cellsize.
% Fill the cells with black to form a closed border of black cells
% around interior points. These cells are the cellular complex.
for k = 1:length(I)
   B(x(k):x(k) + cellsize-1, y(k):y(k) + cellsize-1) = 1;
end

BF = imfill(B, 'holes');

% Extract the points interior to the black border. This is the region
% of interest around which the MPP will be found.
B = BF & (~B);

% Extract the 4-connected boundary.
B = boundaries(B, 4, 'cw');
% Find the largest one in case of parasitic regions.
J = cellfun('length', B);
I = find(J == max(J));
B = B{I(1)};

% Function boundaries outputs the last coordinate pair equal to the
% first.  Delete it.
B = B(1:end-1,:);

% Obtain the xy coordinates of the boundary.
x = B(:, 1);
y = B(:, 2);

% Find the smallest x-coordinate and corresponding
% smallest y-coordinate.
cx = find(x == min(x));
cy = find(y == min(y(cx)));

% The cell with top leftmost corner at (x1, y1) below is the first
% point considered by the algorithm.  The remaining points are
% visited in the clockwise direction starting at (x1, y1).
x1 = x(cx(1));
y1 = y(cy(1));
```

```
% Scroll data so that the first point is (x1, y1).
I = find(x == x1 & y == y1);
x = circshift(x, [-(I - 1), 0]);
y = circshift(y, [-(I - 1), 0]);

% The same shift applies to B.
B = circshift(B, [-(I - 1), 0]);

% Get the Freeman chain code.  The first row of B is the required
% starting point. The first element of the code is the transition
% between the 1st and 2nd element of B, the second element of
% the code is the transition between the 2nd and 3rd elements of B,
% and so on.  The last element of the code is the transition between
% the last and 1st elements of B. The elements of B form a cw
% sequence (see above), so we use 'same' for the direction in
% function fchcode.
code = fchcode(B, 4, 'same');
code = code.fcc;

% Follow the code sequence to extract the Black Dots, BD, (convex
% corners) and White Dots, WD, (concave corners). The transitions are
% as follows: 0-to-1=WD; 0-to-3=BD; 1-to-0=BD; 1-to-2=WD; 2-to-1=BD;
% 2-to-3=WD; 3-to-0=WD; 3-to-2=dot.  The formula t = 2*first - second
% gives the following unique values for these transitions: -1, -3, 2,
% 0, 3, 1, 6, 4.  These are applicable to travel in the cw direction.
% The WD's are displaced one-half a diagonal from the BD's to form
% the half-cell expansion required in the algorithm.

% Vertices will be computed as array "vertices" of dimension nv-by-3,
% where nv is the number of vertices. The first two elements of any
% row of array vertices are the (x,y) coordinates of the vertex
% corresponding to that row, and the third element is 1 if the
% vertex is convex (BD) or 2 if it is concave (WD). The first vertex
% is known to be convex, so it is black.
vertices = [x1, y1, 1];
n = 1;
k = 1;
for k = 2:length(code)
   if code(k - 1) ~= code(k)
      n = n + 1;
      t = 2*code(k-1) - code(k); % t = value of formula.
      if t == -3 | t == 2 | t == 3 | t == 4 % Convex: Black Dots.
         vertices(n, 1:3) = [x(k), y(k), 1];
      elseif t == -1 | t == 0 | t == 1 | t == 6 % Concave: White Dots.
         if t == -1
            vertices(n, 1:3) = [x(k) - cellsize, y(k) - cellsize,2];
         elseif t==0
            vertices(n, 1:3) = [x(k) + cellsize, y(k) - cellsize,2];
         elseif t==1
            vertices(n, 1:3) = [x(k) + cellsize, y(k) + cellsize,2];
         else
            vertices(n, 1:3) = [x(k) - cellsize, y(k) + cellsize,2];
         end
```

```
      else
         % Nothing to do here.
      end
   end
end

% The rest of minperpoly.m processes the vertices to
% arrive at the MPP.

flag = 1;
while flag
   % Determine which vertices lie on or inside the
   % polygon whose vertices are the Black Dots. Delete all
   % other points.
   I = find(vertices(:, 3) == 1);
   xv = vertices(I, 1); % Coordinates of the Black Dots.
   yv = vertices(I, 2);
   X = vertices(:, 1); % Coordinates of all vertices.
   Y = vertices(:, 2);
   IN = inpolygon(X, Y, xv, yv);
   I = find(IN ~= 0);
   vertices = vertices(I, :);

   % Now check for any Black Dots that may have been turned into
   % concave vertices after the previous deletion step. Delete
   % any such Black Dots and recompute the polygon as in the
   % previous section of code. When no more changes occur, set
   % flag to 0, which causes the loop to terminate.
   x = vertices(:, 1);
   y = vertices(:, 2);
   angles = polyangles(x, y); % Find all the interior angles.
   I = find(angles > 180 & vertices(:, 3) == 1);
   if isempty(I)
      flag = 0;
   else
      J = 1:length(vertices);
      for k = 1:length(I)
         K = find(J ~= I(k));
         J = J(K);
      end
      vertices = vertices(J, :);
   end
end

% Final pass to delete the vertices with angles of 180 degrees.
x = vertices(:, 1);
y = vertices(:, 2);
angles = polyangles(x, y);
I = find(angles ~= 180);

% Vertices of the MPP:
x = vertices(I, 1);
y = vertices(I, 2);
```

P

```
function B = pixeldup(A, m, n)
%PIXELDUP Duplicates pixels of an image in both directions.
%   B = PIXELDUP(A, M, N) duplicates each pixel of A M times in the
%   vertical direction and N times in the horizontal direction.
%   Parameters M and N must be integers.  If N is not included, it
%   defaults to M.

% Check inputs.
if nargin < 2
   error('At least two inputs are required.');
end
if nargin == 2
   n = m;
end

% Generate a vector with elements 1:size(A, 1).
u = 1:size(A, 1);

% Duplicate each element of the vector m times.
m = round(m); % Protect against nonintergers.
u = u(ones(1, m), :);
u = u(:);

% Now repeat for the other direction.
v = 1:size(A, 2);
n = round(n);
v = v(ones(1, n), :);
v = v(:);
B = A(u, v);

function angles = polyangles(x, y)
%POLYANGLES Computes internal polygon angles.
%   ANGLES = POLYANGLES(X, Y) computes the interior angles (in
%   degrees) of an arbitrary polygon whose vertices are given in
%   [X, Y], ordered in a clockwise manner.  The program eliminates
%   duplicate adjacent rows in [X Y], except that the first row may
%   equal the last, so that the polygon is closed.

% Preliminaries.
[x y] = dupgone(x, y); % Eliminate duplicate vertices.
xy = [x(:) y(:)];
if isempty(xy)
   % No vertices!
   angles = zeros(0, 1);
   return;
end
if size(xy, 1) == 1 | ~isequal(xy(1, :), xy(end, :))
   % Close the polygon
   xy(end + 1, :) = xy(1, :);
end

% Precompute some quantities.
d = diff(xy, 1);
```

```
v1 = -d(1:end, :);
v2 = [d(2:end, :); d(1, :)];
v1_dot_v2 = sum(v1 .* v2, 2);
mag_v1 = sqrt(sum(v1.^2, 2));
mag_v2 = sqrt(sum(v2.^2, 2));

% Protect against nearly duplicate vertices; output angle will be 90
% degrees for such cases. The "real" further protects against
% possible small imaginary angle components in those cases.
mag_v1(~mag_v1) = eps;
mag_v2(~mag_v2) = eps;
angles = real(acos(v1_dot_v2 ./ mag_v1 ./ mag_v2) * 180 / pi);

% The first angle computed was for the second vertex, and the
% last was for the first vertex. Scroll one position down to
% make the last vertex be the first.
angles = circshift(angles, [1, 0]);

% Now determine if any vertices are concave and adjust the angles
% accordingly.
sgn = convex_angle_test(xy);

% Any element of sgn that's -1 indicates that the angle is
% concave. The corresponding angles have to be subtracted
% from 360.
I = find(sgn == -1);
angles(I) = 360 - angles(I);

%-------------------------------------------------------------------%
function sgn = convex_angle_test(xy)
%   The rows of array xy are ordered vertices of a polygon. If the
%   kth angle is convex (>0 and <= 180 degress) then sgn(k) =
%   1. Otherwise sgn(k) = -1. This function assumes that the first
%   vertex in the list is convex, and that no other vertex has a
%   smaller value of x-coordinate. These two conditions are true in
%   the first vertex generated by the MPP algorithm. Also the
%   vertices are assumed to be ordered in a clockwise sequence, and
%   there can be no duplicate vertices.
%
%   The test is based on the fact that every convex vertex is on the
%   positive side of the line passing through the two vertices
%   immediately following each vertex being considered.  If a vertex
%   is concave then it lies on the negative side of the line joining
%   the next two vertices. This property is true also if positive and
%   negative are interchanged in the preceding two sentences.

% It is assumed that the polygon is closed.  If not, close it.
if size(xy, 1) == 1 | ~isequal(xy(1, :), xy(end, :))
   xy(end + 1, :) = xy(1, :);
end

% Sign convention: sgn = 1 for convex vertices (i.e, interior angle > 0
% and <= 180 degrees), sgn = -1 for concave vertices.
```

```
% Extreme points to be used in the following loop.  A 1 is appended
% to perform the inner (dot) product with w, which is 1-by-3 (see
% below).
L = 10^25;
top_left = [-L, -L, 1];
top_right = [-L, L, 1];
bottom_left = [L, -L, 1];
bottom_right = [L, L, 1];

sgn = 1; % The first vertex is known to be convex.

% Start following the vertices.
for k = 2:length(xy) - 1
   pfirst= xy(k - 1, :);
   psecond = xy(k, :); % This is the point tested for convexity.
   pthird = xy(k + 1, :);
   % Get the coefficients of the line (polygon edge) passing
   % through pfirst and psecond.
   w = polyedge(pfirst, psecond);

   % Establish the positive side of the line w1x + w2y + w3 = 0.
   % The positive side of the line should be in the right side of the
   % vector (psecond - pfirst).  deltax and deltay of this vector
   % give the direction of travel. This establishes which of the
   % extreme points (see above) should be on the + side. If that
   % point is on the negative side of the line, then w is replaced by -w.

   deltax = psecond(:, 1) - pfirst(:, 1);
   deltay = psecond(:, 2) - pfirst(:, 2);
   if deltax == 0 & deltay == 0
      error('Data into convexity test is 0 or duplicated.')
   end
   if deltax <= 0  & deltay >= 0 % Bottom_right should be on + side.
      vector_product = dot(w, bottom_right); % Inner product.
      w = sign(vector_product)*w;
   elseif deltax <= 0 & deltay <= 0 % Top_right should be on + side.
      vector_product = dot(w, top_right);
      w = sign(vector_product)*w;
   elseif deltax >= 0 & deltay <= 0  % Top_left should be on + side.
      vector_product = dot(w, top_left);
      w = sign(vector_product)*w;
   else % deltax >= 0 & deltay >= 0, so bottom_left should be on + side.
      vector_product = dot(w, bottom_left);
      w = sign(vector_product)*w;
   end
   % For the vertex at psecond to be convex, pthird has to be on the
   % positive side of the line.
   sgn(k) = 1;
   if (w(1)*pthird(:, 1) + w(2)*pthird(:, 2) + w(3)) < 0
      sgn(k) = -1;
   end
end
```

```
%-----------------------------------------------------------------------%
function w = polyedge(p1, p2)
%   Outputs the coefficients of the line passing through p1 and
%   p2. The line is of the form w1x + w2y + w3 = 0.

x1 = p1(:, 1);  y1 = p1(:, 2);
x2 = p2(:, 1);  y2 = p2(:, 2);
if x1==x2
   w2 = 0;
   w1 = -1/x1;
   w3 = 1;
elseif y1==y2
   w1 = 0;
   w2 = -1/y1;
   w3 = 1;
elseif x1 == y1 & x2 == y2
   w1 = 1;
   w2 = 1;
   w3 = 0;
else
   w1 = (y1 - y2)/(x1*(y2 - y1) - y1*(x2 - x1) + eps);
   w2 = -w1*(x2 - x1)/(y2 - y1);
   w3 = 1;
end
w = [w1, w2, w3];
%-----------------------------------------------------------------------%
function [xg, yg] = dupgone(x, y)
% Eliminates duplicate, adjacent rows in [x y], except that the
% first and last rows can be equal so that the polygon is closed.

xg = x;
yg = y;
if size(xg, 1) > 2
   I = find((x(1:end-1, :) == x(2:end, :)) & ...
            (y(1:end-1, :) == y(2:end, :)));
   xg(I) = [];
   yg(I) = [];
end
```

R

```
function [xn, yn] = randvertex(x, y, npix)
%RANDVERTEX Adds random noise to the vertices of a polygon.
%   [XN, YN] = RANDVERTEX[X, Y, NPIX] adds uniformly distributed
%   noise to the coordinates of vertices of a polygon. The
%   coordinates of the vertices are input in X and Y, and NPIX is the
%   maximum number of pixel locations by which any pair (X(i), Y(i))
%   is allowed to deviate. For example, if NPIX = 1, the location of
%   any X(i) will not deviate by more than one pixel location in the
%   x-direction, and similarly for Y(i). Noise is added independently
%   to the two coordinates.
```

```
% Convert to columns.
x = x(:);
y = y(:);

% Preliminary calculations.
L = length(x);
xnoise = rand(L, 1);
ynoise = rand(L, 1);
xdev = npix*xnoise.*sign(xnoise - 0.5);
ydev = npix*ynoise.*sign(ynoise - 0.5);

% Add noise and round.
xn = round(x + xdev);
yn = round(y + ydev);

% All pixel locations must be no less than 1.
xn = max(xn, 1);
yn = max(yn, 1);
```

S

```
function [st, angle, x0, y0] = signature(b, varargin)
%SIGNATURE Computes the signature of a boundary.
%   [ST, ANGLE, X0, Y0] = SIGNATURE(B) computes the
%   signature of a given boundary, B, where B is an np-by-2 array
%   (np > 2) containing the (x, y) coordinates of the boundary
%   ordered in a clockwise or counterclockwise direction. The
%   amplitude of the signature as a function of increasing ANGLE is
%   output in ST. (X0,Y0) are the coordinates of the centroid of the
%   boundary. The maximum size of arrays ST and ANGLE is 360-by-1,
%   indicating a maximum resolution of one degree. The input must be
%   a one-pixel-thick boundary obtained, for example, by using the
%   function boundaries. By definition, a boundary is a closed curve.
%
%   [ST, ANGLE, X0, Y0] = SIGNATURE(B) computes the signature, using
%   the centroid as the origin of the signature vector.
%
%   [ST, ANGLE, X0, Y0] = SIGNATURE(B, X0, Y0) computes the boundary
%   using the specified (X0, Y0) as the origin of the signature
%   vector.

% Check dimensions of b.
[np, nc] = size(b);
if (np < nc | nc ~= 2)
   error('B must be of size np-by-2.');
end

% Some boundary tracing programs, such as boundaries.m, end where
% they started, resulting in a sequence in which the coordinates
% of the first and last points are the same. If this is the case,
% in b, eliminate the last point.
if isequal(b(1, :), b(np, :))
   b = b(1:np - 1, :);
```

```
   np = np - 1;
end

% Compute parameters.
if nargin == 1
   x0 = round(sum(b(:, 1))/np); % Coordinates of the centroid.
   y0 = round(sum(b(:, 2))/np);
elseif nargin == 3
   x0 = varargin{1};
   y0 = varargin{2};
else
   error('Incorrect number of inputs.');
end

% Shift origin of coord system to (x0, y0)).
b(:, 1) = b(:, 1) - x0;
b(:, 2) = b(:, 2) - y0;

% Convert the coordinates to polar.  But first have to convert the
% given image coordinates, (x, y), to the coordinate system used by
% MATLAB for conversion between Cartesian and polar cordinates.
% Designate these coordinates by (xc, yc). The two coordinate systems
% are related as follows:  xc = y and yc = -x.
xc = b(:, 2);
yc = -b(:, 1);
[theta, rho] = cart2pol(xc, yc);

% Convert angles to degrees.
theta = theta.*(180/pi);

% Convert to all nonnegative angles.
j = theta == 0; % Store the indices of theta = 0 for use below.
theta = theta.*(0.5*abs(1 + sign(theta)))...
        - 0.5*(-1 + sign(theta)).*(360 + theta);
theta(j) = 0; % To preserve the 0 values.

temp = theta;
% Order temp so that sequence starts with the smallest angle.
% This will be used below in a check for monotonicity.
I = find(temp == min(temp));

% Scroll up so that sequence starts with the smallest angle.
% Use I(1) in case the min is not unique (in this case the
% sequence will not be monotonic anyway).
temp = circshift(temp, [-(I(1) - 1), 0]);

% Check for monotonicity, and issue a warning if sequence
% is not monotonic. First determine if sequence is
% cw or ccw.
k1 = abs(temp(1) - temp(2));
k2 = abs(temp(1) - temp(3));
if k2 > k1
   sense = 1; % ccw
elseif k2 < k1
   sense = -1; % cw
```

```
else
   warning(['The first 3 points in B do not form a monotonic ' ...
            'sequence.']);
end
% Check the rest of the sequence for monotonicity. Because
% the angles are rounded to the nearest integer later in the
% program, only differences greater than 0.5 degrees are
% considered in the test for monotonicity in the rest of
% the sequence.
flag = 0;
for k = 3:length(temp) - 1
   diff = sense*(temp(k + 1) - temp(k));
   if diff < -.5
      flag = 1;
   end
end
if flag
   warning('Angles do not form a monotonic sequence.');
end

% Round theta to 1 degree increments.
theta = round(theta);

% Keep theta and rho together.
tr = [theta, rho];

% Delete duplicate angles.  The unique operation
% also sorts the input in ascending order.
[w, u, v] = unique(tr(:, 1));
tr = tr(u,:); % u identifies the rows kept by unique.

% If the last angle equals 360 degrees plus the first
% angle, delete the last angle.
if tr(end, 1) == tr(1) + 360
   tr = tr(1:end - 1, :);
end

% Output the angle values.
angle = tr(:, 1);

% The signature is the set of values of rho corresponding
% to the angle values.
st = tr(:, 2);
```

```
function [srad, sang, S] = specxture(f)
%SPECXTURE Computes spectral texture of an image.
%   [SRAD, SANG, S] = SPECXTURE(F) computes SRAD, the spectral energy
%   distribution as a function of radius from the center of the
%   spectrum, SANG, the spectral energy distribution as a function of
%   angle for 0 to 180 degrees in increments of 1 degree, and S =
%   log(1 + spectrum of f), normalized to the range [0, 1]. The
%   maximum value of radius is min(M,N), where M and N are the number
%   of rows and columns of image (region) f. Thus, SRAD is a row
%   vector of length = (min(M, N)/2) - 1; and SANG is a row vector of
%   length 180.
```

```
% Obtain the centered spectrum, S, of f. The variables of S are
% (u, v), running from 1:M and 1:N, with the center (zero frequency)
% at [M/2 + 1, N/2 + 1] (see Chapter 4).
S = fftshift(fft2(f));
S = abs(S);
[M, N] = size(S);
x0 = M/2 + 1;
y0 = N/2 + 1;

% Maximum radius that guarantees a circle centered at (x0, y0) that
% does not exceed the boundaries of S.
rmax = min(M, N)/2 - 1;

% Compute srad.
srad = zeros(1, rmax);
srad(1) = S(x0, y0);
for r = 2:rmax
   [xc, yc] = halfcircle(r, x0, y0);
   srad(r) = sum(S(sub2ind(size(S), xc, yc)));
end

% Compute sang.
[xc, yc] = halfcircle(rmax, x0, y0);
sang = zeros(1, length(xc));
for a = 1:length(xc)
   [xr, yr] = radial(x0, y0, xc(a), yc(a));
   sang(a) = sum(S(sub2ind(size(S), xr, yr)));
end

% Output the log of the spectrum for easier viewing, scaled to the
% range [0, 1].
S = mat2gray(log(1 + S));

%-------------------------------------------------------------------%
function [xc, yc] = halfcircle(r, x0, y0)
%   Computes the integer coordinates of a half circle of radius r and
%   center at (x0,y0) using one degree increments.
%
%   Goes from 91 to 270 because we want the half circle to be in the
%   region defined by top right and top left quadrants, in the
%   standard image coordinates.

theta=91:270;
theta = theta*pi/180;
[xc, yc] = pol2cart(theta, r);
xc = round(xc)' + x0; % Column vector.
yc = round(yc)' + y0;

%-------------------------------------------------------------------%
function [xr, yr] = radial(x0, y0, x, y);
%   Computes the coordinates of a straight line segment extending
%   from (x0, y0) to (x, y).
%
```

```
%   Based on function intline.m. xr and yr are
%   returned as column vectors.

 [xr, yr] = intline(x0, x, y0, y);
```

```
function [v, unv] = statmoments(p,  n)
%STATMOMENTS Computes statistical central moments of image histogram.
%   [W, UNV] = STATMOMENTS(P, N) computes up to the Nth statistical
%   central moment of a histogram whose components are in vector
%   P. The length of P must equal 256 or 65536.
%
%   The program outputs a vector V with V(1) = mean, V(2) = variance,
%   V(3) = 3rd moment, . . . V(N) = Nth central moment. The random
%   variable values are normalized to the range [0, 1], so all
%   moments also are in this range.
%
%   The program also outputs a vector UNV containing the same moments
%   as V, but using un-normalized random variable values (e.g., 0 to
%   255 if length(P) = 2^8). For example, if length(P) = 256 and V(1)
%   = 0.5, then UNV(1) would have the value UNV(1) = 127.5 (half of
%   the [0 255] range).

Lp = length(p);
if (Lp ~= 256) & (Lp ~= 65536)
   error('P must be a 256- or 65536-element vector.');
end
G = Lp - 1;

% Make sure the histogram has unit area, and convert it to a
% column vector.
p = p/sum(p); p = p(:);

% Form a vector of all the possible values of the
% random variable.
z = 0:G;

% Now normalize the z's to the range [0, 1].
z = z./G;

% The mean.
m = z*p;

% Center random variables about the mean.
z = z - m;

% Compute the central moments.
v = zeros(1, n);
v(1) = m;
for j = 2:n
   v(j) = (z.^j)*p;
end

if nargout > 1
   % Compute the uncentralized moments.
   unv = zeros(1, n);
   unv(1)=m.*G;
```

```
   for j = 2:n
      unv(j) = ((z*G).^j)*p;
   end
end
```

```
function [t] = statxture(f, scale)
%STATXTURE Computes statistical measures of texture in an image.
%   T = STATXURE(F, SCALE) computes six measures of texture from an
%   image (region) F. Parameter SCALE is a 6-dim row vector whose
%   elements multiply the 6 corresponding elements of T for scaling
%   purposes. If SCALE is not provided it defaults to all 1s.  The
%   output T is 6-by-1 vector with the following elements:
%      T(1) = Average gray level
%      T(2) = Average contrast
%      T(3) = Measure of smoothness
%      T(4) = Third moment
%      T(5) = Measure of uniformity
%      T(6) = Entropy

if nargin == 1
   scale(1:6) = 1;
else % Make sure it's a row vector.
   scale = scale(:)';
end
% Obtain histogram and normalize it.
p = imhist(f);
p = p./numel(f);
L = length(p);

% Compute the three moments. We need the unnormalized ones
% from function statmoments. These are in vector mu.
[v, mu] = statmoments(p, 3);

% Compute the six texture measures:
% Average gray level.
t(1) = mu(1);
% Standard deviation.
t(2) = mu(2).^0.5;
% Smoothness.
% First normalize the variance to [0 1] by
% dividing it by (L-1)^2.
varn = mu(2)/(L - 1)^2;
t(3) = 1 - 1/(1 + varn);
% Third moment (normalized by (L - 1)^2 also).
t(4) = mu(3)/(L - 1)^2;
% Uniformity.
t(5) = sum(p.^2);
% Entropy.
t(6) = -sum(p.*(log2(p + eps)));

% Scale the values.
t = t.*scale;
```

X

```
function [B, theta] = x2majoraxis(A, B, type)
%X2MAJORAXIS Aligns coordinate x with the major axis of a region.
%   [B2, THETA] = X2MAJORAXIS(A, B, TYPE) aligns the x-coordinate
%   axis with the major axis of a region or boundary. The y-axis is
%   perpendicular to the x-axis.  The rows of 2-by-2 matrix A are the
%   coordinates of the two end points of the major axis, in the form
%   A = [x1 y1; x2 y2]. On input, B is either a binary image (i.e.,
%   an array of class logical) containing a single region, or it is
%   an np-by-2 set of points representing a (connected) boundary. In
%   the latter case, the first column of B must represent
%   x-coordinates and the second column must represent the
%   corresponding y-coordinates. On output, B contains the same data
%   as the input, but aligned with the major axis. If the input is an
%   image, so is the output; similarly the output is a sequence of
%   coordinates if the input is such a sequence. Parameter THETA is
%   the initial angle between the major axis and the x-axis. The
%   origin of the xy-axis system is at the bottom left; the x-axis is
%   the horizontal axis and the y-axis is the vertical.
%
%   Keep in mind that rotations can introduce round-off errors when
%   the data are converted to integer coordinates, which is a
%   requirement.  Thus, postprocessing (e.g., with bwmorph) of the
%   output may be required to reconnect a boundary.

% Preliminaries.
if islogical(B)
   type = 'region';
elseif size(B, 2) == 2
   type = 'boundary';
   [M, N] = size(B);
   if M < N
      error('B is boundary. It must be of size np-by-2; np > 2.')
   end
   % Compute centroid for later use. c is a 1-by-2 vector.
   % Its 1st component is the mean of the boundary in the x-direction.
   % The second is the mean in the y-direction.
   c(1) = round((min(B(:, 1)) + max(B(:, 1))/2));
   c(2) = round((min(B(:, 2)) + max(B(:, 2))/2));

   % It is possible for a connected boundary to develop small breaks
   % after rotation. To prevent this, the input boundary is filled,
   % processed as a region, and then the boundary is re-extracted. This
   % guarantees that the output will be a connected boundary.
   m = max(size(B));
   % The following image is of size m-by-m to make sure that there
   % there will be no size truncation after rotation.
   B = bound2im(B,m,m);
   B = imfill(B,'holes');
```

```
else
   error('Input must be a boundary or a binary image'.)
end

% Major axis in vector form.
v(1) = A(2, 1) - A(1, 1);
v(2) = A(2, 2) - A(1, 2);
v = v(:);  % v is a col vector

% Unit vector along x-axis.
u = [1; 0];

% Find angle between major axis and x-axis. The angle is
% given by acos of the inner product of u and v divided by
% the product of their norms. Because the inputs are image
% points, they are in the first quadrant.
nv = norm(v);
nu = norm(u);
theta = acos(u'*v/nv*nu);
if theta > pi/2
   theta = -(theta - pi/2);
end
theta = theta*180/pi;  % Convert angle to degrees.

% Rotate by angle theta and crop the rotated image to original size.
B = imrotate(B, theta, 'bilinear', 'crop');

% If the input was a boundary, re-extract it.
if strcmp(type, 'boundary')
   B = boundaries(B);
   B = B{1};
   % Shift so that centroid of the extracted boundary is
   % approx equal to the centroid of the original boundary:
   B(:, 1) = B(:, 1) - min(B(:, 1)) + c(1);
   B(:, 2) = B(:, 2) - min(B(:, 2)) + c(2);

end
```