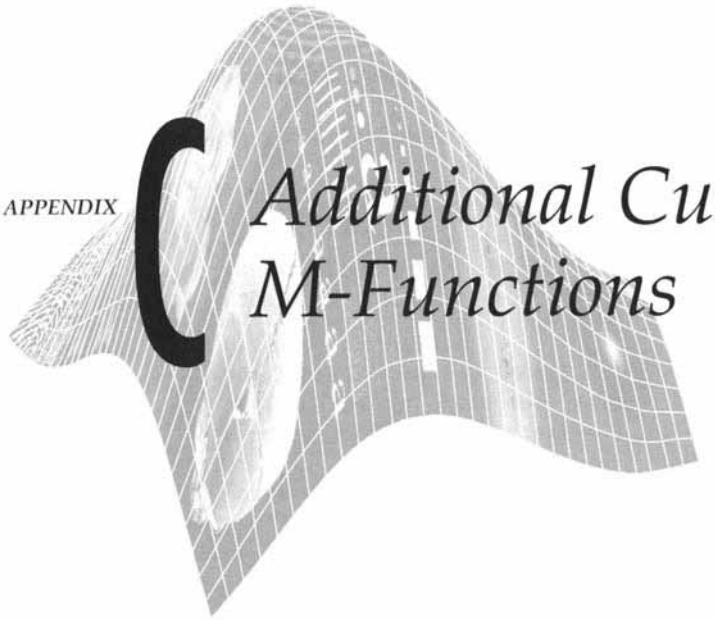

APPENDIX C

Additional Custom 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. Being part of this book, all the following functions are copyrighted and they are intended to be used exclusively by the individual who owns this copy of the book. Any type of dissemination, including copying in any form and/or posting electronically by any means, such as local servers and the Internet, without written consent from the publisher constitutes a violation of national and international copyright law.

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

% 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);
```

```
function av = average(A)
%AVERAGE Computes the average value of an array.
%   AV = AVERAGE(A) computes the average value of input array, A,
%   which must be a 1-D or 2-D array.

% Check the validity of the input. (Keep in mind that
% a 1-D array is a special case of a 2-D array.)
if ndims(A) > 2
   error('The dimensions of the input cannot exceed 2.')
end

% Compute the average
av = sum(A(:))/length(A(:));
```

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 the 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, :)];

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 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 Mmin
%   + 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 [dir, x0 y0] = boundarydir(x, y, orderout)
%BOUNDARYDIR Determine the direction of a sequence of planar points.
%   [DIR] = BOUNDARYDIR(X, Y) determines the direction of travel of
%   a closed, nonintersecting sequence of planar points with
%   coordinates contained in column vectors X and Y. Values of DIR
%   are 'cw' (clockwise) and 'ccw' (counterclockwise). The direction
%   of travel is with respect to the image coordinate system defined
%   in Chapter 2 of the book.
%
%   [DIR, X0, Y0] = BOUNDARYDIR(X, Y, ORDEROUT) determines the
%   direction DIR of the input sequence, and also outputs the
%   sequence with its direction of travel as specified in ORDEROUT.
%   Valid values of this parameter as 'cw' and 'ccw'. The
%   coordinates of the output sequence are column vectors X0 and Y0.
%
%   The input sequence is assumed to be nonintersecting, and it
%   cannot have duplicate points, with the exception of the first
%   and last points possibly being the same, a condition often
%   resulting from boundary-following functions, such as
%   bwboundaries.
```

```
% Preliminaries.
% Make sure coordinates are column vectors.
x = x(:);
y = y(:);

% If the first and last points are the same, delete the last point.
% The point will be restored later.
restore = false;
if x(1) == x(end) && y(1) == y(end)
   x = x(1:end-1);
   y = y(1:end-1);
   restore = true;
end
% Check for duplicate points.
if length([x y]) ~= length(unique([x y],'rows'))
   error('No duplicate points except first and last are allowed.')
end

% The topmost, leftmost point in the sequence is always a convex
% vertex.
x0 = x;
y0 = y;
cx = find(x0 == min(x0));
cy = find(y0 == min(y0(cx)));
x1 = x0(cx(1));
y1 = y0(cy(1));
% Scroll data so that the first point in the sequence is (x1, y1),
% the guaranteed convex point.
I = find(x0 == x1 & y0 == y1);
x0 = circshift(x0, [-(I - 1), 0]);
y0 = circshift(y0, [-(I - 1), 0]);

% Form the matrix needed to check for travel direction. Only three
% points are needed: (x1, y1), the point before it, and the point
% after it.
A = [x0(end) y0(end) 1; x0(1) y0(1) 1; x0(2) y0(2) 1];
dir = 'cw';
if det(A) > 0
   dir = 'ccw';
end

% Prepare outputs.
if nargin == 3
   x0 = x; % Reuse x0 and y0.
   y0 = y;
   if ~strcmp(dir, orderout)
      x0(2:end) = flipud(x0(2:end)); % Reverse order of travel.
      y0(2:end) = flipud(y0(2:end));
   end
   if restore
```

```
      x0(end + 1) = x0(1);
      y0(end + 1) = y0(1);
   end
end

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 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 and that it is
%   ordered in a clockwise or counterclockwise sequence.
%
%   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 inputs.
[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)) + 1;
ymax = max(b(:, 2)) + 1;

% 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 vector of grid coordinates.
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); % See CH 02 regarding function meshgrid.
% Vector of grid all coordinates, arranged as Nunbergridpoints-by-2
% array to match the horizontal dimensions of b. This allows
% computation of distances to be vectorized and thus 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.  See Chapter 13 regarding distance computations.
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;
```

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: 211 $  $Date: 2006-07-31 14:22:42 -0400 (Mon, 31 Jul
2006) $

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 H = cnotch(type, notch, M, N, C, D0, n)
%CNOTCH Generates circularly symmetric notch filters.
%   H = CNOTCH(TYPE, NOTCH, M, N, C, D0, n) generates a notch filter
%   of size M-by-N. C is a K-by-2 matrix with K pairs of frequency
%   domain coordinates (u, v) that define the centers of the filter
%   notches (when specifying filter locations, remember that
%   coordinates in MATLAB run from 1 to M and 1 to N). Coordinates
%   (u, v) are specified for one notch only. The corresponding
%   symmetric notches are generated automatically. D0 is the radius
%   (cut-off frequency) of the notches. It can be specified as a
%   scalar, in which case it is used in all K notch pairs, or it can
%   be a vector of length K, containing an individual cutoff value
%   for each notch pair. n is the order of the Butterworth filter if
%   one is specified.
%
%       Valid values of TYPE are:
%
%           'ideal'     Ideal notchpass filter. n is not used.
%
%           'btw'       Butterworth notchpass filter of order n. The
%                       default value of n is 1.
%
%           'gaussian'  Gaussian notchpass filter. n is not used.
%
%       Valid values of NOTCH are:
%
%           'reject'    Notchreject filter.
%
%           'pass'      Notchpass filter.
%
%       One of these two values must be specified for NOTCH.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).

% Preliminaries.
if nargin < 7
   n = 1; % Default for Butterworth filter.
end

% Define tha largest array of odd dimensions that fits in H. This is
```

```
% required to preserve symmetry in the filter. If necessary, a row
% and/or column is added to the filter at the end of the function.
MO = M;
NO = N;
if iseven(M)
   MO = M - 1;
end
if iseven(N)
   NO = N - 1;
end

% Center of the filter:
center = [floor(MO/2) + 1, floor(NO/2) + 1];

% Number of notch pairs.
K = size(C, 1);
% Cutoff values.
if numel(DO) == 1
        DO(1:K) = DO; % All cut offs are the same.
end

% Shift notch centers  so that they are with respect to the center
% of the filter (and the frequency rectangle).
center = repmat(center, size(C,1), 1);
C = C - center;

% Begin filter computations. All filters are computed as notchreject
% filters. At the end, they are changed to notchpass filters if it
% is so specified in parameter NOTCH.
H = rejectFilter(type, MO, NO, DO, K, C, n);

% Finished. Format the output.
H = processOutput(notch, H, M, N, center);

%--------------------------------------------------------------------%
function H = rejectFilter(type, MO, NO, DO, K, C, n)
% Initialize the filter array to be an "all pass" filter. This
% constant filter is then multiplied by the notchreject filters
% placed at the locations in C with respect to the center of the
% frequency rectangle.
H = ones(MO, NO, 'single');

% Generate filter.
for I = 1:K
   % Place a notch at each location in delta. Function hpfilter
   % returns the filters uncentered. Use fftshit to center the
   % filter at each location. The filters are made larger than
   % M-by-N to simplify indexing in function placeNotches.
   Usize = MO + 2*abs(C(I, 1));
   Vsize = NO + 2*abs(C(I, 2));
```

```
   filt = fftshift(hpfilter(type, Usize , Vsize, D0(I), n));
   % Insert FILT in H.
   H = placeNotches(H, filt, C(I,1), C(I,2));
end

%-----------------------------------------------------------------%
function P = placeNotches(H, filt, delu, delv)
% Places in H the notch contained in FILT.

[M N] = size(H);
U = 2*abs(delu);
V = 2*abs(delv);

% The following calculations are to determine the (common) area of
% overlap between array H and the notch filter FILT.
if delu >= 0 && delv >= 0
   filtCommon = filt(1:M, 1:N); % Displacement is in Q1.
elseif delu < 0 && delv >= 0
   filtCommon = filt(U + 1:U + M, 1:N); % Displacement is in Q2.
elseif delu < 0 && delv < 0
   filtCommon = filt(U + 1:U + M, V + 1:V + N); % Q3
elseif delu >= 0 && delv <= 0
   filtCommon = filt(1:M, V + 1:V + N); % Q4
end

% Compute the product of H and filtCommon. They are registered.
P = ones(M, N).*filtCommon;

% The conjugate notch location is determined by rotating P 180
% degress. This is the same as flipping P left-right and up-down.
% The product of P and its rotated version contain FILT and its
% conjugate.
P = P.*(flipud(fliplr(P)));
P = H.*P; % A new notch and its conjugate were inserted.

%-----------------------------------------------------------------%
function Hout = processOutput(notch, H, M, N, center)
% At this point, H is an odd array in both dimensions (see comments
% at the beginning of the function). In the following, we insert a
% row if M is even, and a column if N is even. The new row and
% column have to be symmetric about their center to preserve
% symmetry in the filter. They are created by duplicating the first
% row and column of H and then making them symmetric.
centerU = center(1,1);
centerV = center(1,2);
newRow = H(1,:);
newRow(1:centerV - 1) = fliplr(newRow(centerV+1:end)); %Symmetric now.
newCol = H(:,1);
newCol(1:centerU - 1) = flipud(newCol(centerU+1:end)); %Symmetric.
% Insert the new row and/or column if appropriate.
```

```
if iseven(M) && iseven(N)
   Hout = cat(1, newRow, H);
   newCol = cat(1, H(1,1), newCol);
   Hout = cat(2, newCol, Hout);
elseif iseven(M) && isodd(N)
   Hout = cat(1, newRow, H);
elseif isodd(M) && iseven(N)
   Hout = cat(2, newCol, H);
else
   Hout = H;
end

% Uncenter the filter, as required for filtering with dftfilt.
Hout = ifftshift(Hout);

% Generate a pass filter if one was specified.
if strcmp(notch, 'pass')
   Hout = 1 - Hout;
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 = 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{:});

function cp = cornerprocess(c, T, q)
%CORNERPROCESS Processes the output of function cornermetric.
%   CP = CORNERPROCESS(C, T, Q) postprocesses C, the output of
%   function CORNERMETRIC, with the objective of reducing the
%   number of irrelevant corner points (with respect to threshold T)
%   and the number of multiple corners in a neighborhood of size
%   Q-by-Q. If there are multiple corner points contained within
%   that neighborhood, they are eroded morphologically to one corner
%   point.
%
%   A corner point is said to have been found at coordinates (I, J)
%   if C(I,J) > T.
%
%   A good practice is to normalize the values of C to the range [0
%   1], in im2double format before inputting C into this function.
%   This facilitates interpretation of the results and makes
%   thresholding more intuitive.

% Peform thresholding.
cp = c > T;

% Dilate CP to incorporate close neighbors.
B = ones(q);
cp = imdilate(cp, B);

% Shrink connnected components to single points.
cp = bwmorph(cp, 'shrink','Inf');

function cv2tifs(y, f)
%CV2TIFS Decodes a TIFS2CV compressed image sequence.
%   Y = CV2TIFS(Y, F) decodes compressed sequence Y (a structure
```

```
%   generated by TIFS2CV) and creates a multiframe TIFF file F.
%
%   See also TIFS2CV.

% Get the number of frames, block size, and reconstruction quality.
fcnt = double(y.frames);
m = double(y.blksz);
q = double(y.quality);

% Reconstruct the first image in the sequence and store.
if q == 0
    r = double(huff2mat(y.video(1)));
else
    r = double(jpeg2im(y.video(1)));
end
imwrite(uint8(r), f, 'Compression', 'none', 'WriteMode', 'overwrite');

% Get the frame size and motion vectors.
fsz = size(r);
mvsz = [fsz/m 2 fcnt];
mv = int16(huff2mat(y.motion));
mv = reshape(mv, mvsz);

% For frames except the first, get a motion conpensated prediction
% residual and add to the proper reference subimages.
for i = 2:fcnt
    if q == 0
        pe = double(huff2mat(y.video(i)));
    else
        pe = double(jpeg2im(y.video(i)) - 255);
    end
    peC = im2col(pe, [m m], 'distinct');

    for col = 1:size(peC, 2)
        u = 1 + mod(m * (col - 1), fsz(1));
        v = 1 + m * floor((col - 1) * m / fsz(1));
        rx = u - mv(1 + floor((u - 1)/m), 1 + floor((v - 1)/m), ...
            1, i);
        ry = v - mv(1 + floor((u - 1)/m), 1 + floor((v - 1)/m), ...
            2, i);

        subimage = r(rx:rx + m - 1, ry:ry + m - 1);
        peC(:, col) = subimage(:) - peC(:, col);
    end

    r = col2im(double(uint16(peC)), [m m], fsz, 'distinct');
    imwrite(uint8(r), f, 'Compression', 'none', ...
        'WriteMode', 'append');
end
```

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. If it does not, an error is issued. See table
%   below.
%
%   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) 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. See Fig. 12.2 for an explanation of the directional 4-
%   and 8-codes.
%
%       ----------------------  ---------------
%       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 bwboundaries.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];

% Check the curve for out-of-order points or breaks.
% 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
   if isempty(find(fcc == 1 || fcc == 3 || fcc == 5 ...
                   || fcc ==7 , 1))
       fcc = fcc./2;
    else
       error('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)
%      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)';
D(J, 1) = 0; % Reserve memory space for loop.
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)
%   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 v = gmean(A)
%GMEAN Geometric mean of columns.
%   V = GMEAN(A) computes the geometric mean of the columns of A.  V
%   is a row vector with size(A,2) elements.
%
%   Sample M-file used in Chapter 3.

m = size(A, 1);
v = prod(A, 1) .^ (1/m);

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
```

function P = i2percentile(h, I)
%I2PERCENTILE Computes a percentile given an intensity value.
```
%   P = I2PERCENTILE(H, I) Given an intensity value, I, and a
%   histogram, H, this function computes the percentile, P, that I
%   represents for the population of intensities governed by
```

```
%   histogram H. I must be in the range [0, 1], independently of the
%   class of the image from which the histogram was obtained. P is
%   returned as a value in the range [0 1]. To convert it to a
%   percentile multiply it by 100. By definition, I = 0 represents
%   the 0th percentile and I = 1 represents 100th percentile.
%
%   Example:
%
%   Suppose that h is a uniform histogram of an uint8 image. Typing
%
%       P = i2percentile(h, 127/255)
%
%   would return P = 0.5, indicating that the input intensity
%   is in the 50th percentile.
%
%   See also function percentile2i.

% Normalized the histogram to unit area. If it is already normalized
% the following computation has no effect.
h = h/sum(h);

% Calculations.
K = numel(h) - 1;
C = cumsum(h); % Cumulative distribution.
if I < 0 || I > 1
    error('Input intensity must be in the range [0, 1].')
elseif I == 0
   P = 0; % Per the definition of percentile.
elseif I == 1
   P = 1; % Per the definition of percentile.
else
   idx = floor(I*K) + 1;
   P = C(idx);
end
```

```
function [X, Y, R] = im2minperpoly(B, cellsize)
%IM2MINPERPOLY Minimum perimeter polygon.
%   [X, Y, R] = IM2MINPERPOLY(B, CELLSIZE) outputs in column vectors
%   X and Y the coordinates of the vertices of the minimum perimeter
%   polygon circumscribing a single binary region or a
%   (nonintersecting) boundary contained in image B. The background
%   in B must be 0, and the region or boundary must have values
%   equal to 1. If instead of an image, B, a list of ordered
%   vertices is available, link the vertices using function
%   connectpoly and then use function bound2im to generate a binary
%   image B containing the boundary.
%
%   R is the region extracted from the image, from which the MPP
%   will be computed (see Figs. 12.5(c) and 12.6(e)). Displaying
%   this region is a good approach to determine interactively a
```

```
%   satisfactory value for CELLSIZE. Parameter CELLSIZE is the size
%   of the square cells that enclose the boundary of the region in
%   B. The value of CELLSIZE must be a positive integer greater than
%   1. See Section 12.2.2 in the book for further details on this
%   parameter, as well as a description and references for the
%   algorithm.

% Preliminaries.
if cellsize <= 1
    error('cellsize must be an integer > 1.');
end
% Check to see that there is only one object in B.
[B, num] = bwlabel(B);
if num > 1
   error('Input image cannot contain more than one region.')
end

% Extract the 4-connected region encompassed by the cellular
% complex. See Fig. 12.6(e) in DIPUM 2/e.
R = cellcomplex(B, cellsize);

% Find the vertices of the MPP.
[X Y] = mppvertices(R, cellsize);

%--------------------------------------------------------------------%
function R = cellcomplex(B, cellsize)
% Computes the cellular complex surrounding a single object in
% binary image B, and outputs in R the region bpounded by the
% cellular complex, as explained in DIPUM/2E Figs. 12.5(c) and
% 12.6(e). Parameter CELLSIZE is as explained earlier.

% Fill the image in case it has holes and compute the 4-connected
% boundary of the result. This guarantees that will be working with
% a single 4-connected boundary, as required by the MPP algorithm.
% Recall that in function bwperim connectivity is with respect to
% the background; therefore, we specify a connectivity of 8 to get a
% connectivity of 4 in the boundary.
B = imfill(B, 'holes');
B = bwperim(B, 8);
[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 the 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.
M1 = K - M;
```

```
N1 = K - N;
B = padarray(B, [M1 N1], 'post'); % B 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);

% Find all the subimages that contain at least one black pixel.
% These will be the cells of the cellular complex enclosing the
% boundary.
I = find(sum(sum(vals(:, :, :)) >= 1));
LI = length(I);
x = r(I);
y = c(I);

% [x', y'] is an LI-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 are the cells are the cellular
% complex.
for k = 1:LI
   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 cell border. This is the
% region, R, around which the MPP will be found.
B = BF & (~B);
R = B(1:M, 1:N); % Remove the padding and output the region.

%--------------------------------------------------------------------%
function [X, Y] = mppvertices(R, cellsize)
%   Outputs in column vectors X and Y the coordinates of the
%   vertices of the minimum-perimeter polygon that circumscribes
%   region R. This is the region bounded by the cellular complex. It
%   is assumed that the coordinate system used is as defined in
%   Chapter 2 of the book, in which the origin is at the top, left,
%   the positive x-axis extends vertically down from the origin and
%   the positive y-axis extends horizontally to the right. No
%   duplicate vertices are allowed. Parameter CELLSIZE is as
%   explained earlier.

% Extract the 4-connected boundary of the region. Reuse variable B.
% It will be a boundary now. See Fig. 12.6(f) in DIPUM 2/e.
B = bwboundaries(R, 4, 'noholes');
B = B{1};
% Function bwboundaries outputs the last coordinate pair equal
% to the first.  Delete it.
```

```
B = B(1:end - 1, :);

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

% Format the vertices in the form required by the algorithm.
L = vertexlist(x, y, cellsize);
NV = size(L, 1); % Number of vertices in L.
count = 1;       % Index for the vertices in the list.
k = 1;           % Index for vertices in the MPP.
X(1) = L(1,1);   % 1st vertex, known to be an MPP vertex.
Y(1) = L(1,2);

% Find the vertices of the MPP.
% Initialize.
cMPPV = [L(1,1), L(1,2)]; % Current MPP vertex.
cV = cMPPV;               % Current vertex.
classV = L(1,3);          % Class of current vertex (+1 for convex).
cWH = cMPPV;              % Current WHITE crawler.
cBL = cMPPV;              % Current BLACK crawler.

% Process the vertices. This is the core of the MPP algorithm.
% Note: Cannot preallocate memory for X and Y because their length
% is variable.
while true
   count = count + 1;
   if count > NV + 1
      break;
   end
   % Process next vertex.
   if count == NV + 1  % Have arrived at first vertex again.
      cV = [L(1,1), L(1,2)];
      classV = L(1,3);
   else
      cV = [L(count, 1), L(count, 2)];
      classV = L(count, 3);
   end
   [I, newMPPV, W, B] = mppVtest(cMPPV, cV, classV, cWH, cBL);
   if I == 1 % New MPP vertex found;
      cMPPV = newMPPV;
      K = find(L(:,1) == newMPPV(:, 1) & L(:,2) == newMPPV(:, 2));
      count = K; % Restart at current location of MPP vertex.
      cWH = newMPPV;
      cBL = newMPPV;
      k = k + 1;
      % Vertices of the MPP just found.
      X(k) = newMPPV(1,1);
      Y(k) = newMPPV(1,2);
```

```
    else
       cWH = W;
       cBL = B;
    end
end
% Convert to columns.
X = X(:);
Y = Y(:);

%-----------------------------------------------------------------%
function L = vertexlist(x, y, cellsize)
%   Given a set of coordinates contained in vectors X and Y, this
%   function outputs a list, L, of the form L = [X(k) Y(k) C(k)]
%   where C(k) determines whether X(k) and Y(k) are the coordinates
%   of the apex of a convex, concave, or 180-degree angle. That is,
%   C(k) = 1 if the coordinates (x(k - 1) y(k - 1), (x(k), y(k)) and
%   (x(k + 1), y(k + 1)) form a convex angle; C(k) = -1 if the angle
%   is concave; and C(k) = 0 if the three points are collinear.
%   Concave angles are replaced by their corresponding convex angles
%   in the outer wall for later use in the minimum-perimeter polygon
%   algorithm, as explained in the book.

% Preprocess the input data. First, arrange the the points so that
% the first point is the top, left-most point in the sequence. This
% guarantees that the first vertex of the polygon is convex.
cx = find(x == min(x));
cy = find(y == min(y(cx)));
x1 = x(cx(1));
y1 = y(cy(1));
% Scroll data so that the first point in the sequence is (x1, y1)
I = find(x == x1 & y == y1);
x = circshift(x, [-(I - 1), 0]);
y = circshift(y, [-(I - 1), 0]);

% Next keep only the points at which a change in direction takes
% place. These are the only points that are polygon vertices. Note
% that we cannot preallocate memory for the loop because xnew and
% ynew are of variable length.
J = 1;
K = length(x);
xnew(1) = x(1);
ynew(1) = y(1);
x(K + 1) = x(1);
y(K + 1) = y(1);
for k = 2:K
   s = vsign([x(k - 1),y(k - 1)], [x(k),y(k)], [x(k + 1),y(k + 1)]);
   if s ~= 0
      J = J + 1;
      xnew(J) = x(k); %#ok<AGROW>
      ynew(J) = y(k); %#ok<AGROW>
```

```
   end
end
% Reuse x and y.
x = xnew;
y = ynew;

% The mpp algorithm works with boundaries in the ccw direction.
% Force the sequence to be in that direction. Output dir is the
% direction of the original boundary. It is not used in this
% function.
[dir, x, y] = boundarydir(x, y, 'ccw');

% Obtain the list of vertices.
% Initialize.
K = length(x);
L(:, :, :) = [x(:) y(:) zeros(K,1)]; % Initialize the list.
C = zeros(K, 1); % Preallocate memory for use in a loop later.

% Do the first and last vertices separately.
% First vertex.
s = vsign([x(K) y(K)], [x(1) y(1)], [x(2) y(2)]);
if s > 0
   C(1) = 1;
elseif s < 0
   C(1) = -1;
   [rx ry] = vreplacement([x(K) y(K)], [x(1) y(1)],...
                          [x(2) y(2)], cellsize);
   L(1, 1) = rx;
   L(1, 2) = ry;
else
   C(1) = 0;
end
% Last vertex.
s = vsign([x(K - 1) y(K - 1)], [x(K) y(K)], [x(1) y(1)]);
if s > 0
   C(K) = 1;
elseif s < 0
   C(K) = -1;
   [rx ry] = vreplacement([x(K - 1) y(K - 1)], [x(K) y(K)], ...
                          [x(1) y(1)], cellsize);
   L(K, 1) = rx;
   L(K, 2) = ry;
else
   C(K) = 0;
end

% Process the rest of the vertices.
for k = 2:K - 1
   s = vsign([x(k - 1) y(k - 1)], [x(k) y(k)], [x(k + 1) y(k + 1)]);
    if s > 0
```

```
      C(k) = 1;
    elseif s < 0
      C(k) = -1;
       [rx ry] = vreplacement([x(k - 1) y(k - 1)], [x(k) y(k)], ...
                        [x(k + 1) y(k + 1)], cellsize);
       L(k, 1) = rx;
       L(k, 2) = ry;
    else
      C(k) = 0;
    end
end

% Update the list with the C's.
L(:, 3)= C(:);

%-------------------------------------------------------------------%
function s = vsign(v1, v2, v3)
%   This function etermines whether a vertex V3 is on the
%   positive or the negative side of straight line passing through
%   V1 and V2, or whether the three points are colinear.  V1, V2,
%   and V3 are 1-by-2 or 2-by-1 vectors containing the [x  y]
%   coordinates of the vertices.  If V3 is on the positive side of
%   the line passing through V1 and V2, then the sign is positive (S
%   > 0), if it is on the negative side of the line the sign is
%   negative (S < 0).  If the points are collinear, then S = 0.
%   Another important interpretation is that if the triplet (V1, V2,
%   V3) form a counterclockwise sequence, then S > 0; if the points
%   form a clockwise sequence then S < 0; if the points are
%   collinear, then S = 0.
%
%   The coordinate system is assumed to be the system is as defined
%   in Chapter 2 of the book.
%
%   This function is based in the result from matrix theory that if
%   we arrange the coordinates of the vertices as the matrix
%
%        A = [V1(1) V1(2) 1; V2(1) V2(2) 1; V3(1) V3(2) 1]
%
%   then, S = det(A) has the properties described above, assuming
%   the stated coordinate system and direction of travel.

% Form the matrix on which the test if based:
A = [v1(1) v1(2) 1; v2(1) v2(2) 1; v3(1), v3(2), 1];
% Compute the determinant.
s = det(A);

%-------------------------------------------------------------------%
function [rx ry] = vreplacement(v1, v, v2, cellsize)
%   This function replaces the coordinates V(1) and V(2) of concave
%   vertex V by its diagonal mirror coordinates [RX, RY]. The values
```

```
%   RX and RY depend on the orientation of the triplet (V1, V, V2).
%   V1 is the vertex preceding V and V2 is the vertex following it.
%   All Vs are 1-by-2 or 2-by-1 arrays containing the coordinates of
%   the vertices. It is assumed that the triplet (V1, V, V2) was
%   generated by traveling in the counterclockwise direction, in the
%   coordinate system defined in Chapter 2 of the book, in which the
%   origin is at the top left, the positive x-axis extends down and
%   the positive y-axis extends to the right. Parameter CELLSIZE is
%   as explained earlier.

% Perform the replacement.

if v(1)>v1(1) && v(2) == v1(2) && v(1) == v2(1) && v(2)>v2(2)
    rx = v(1) - cellsize;
    ry = v(2) - cellsize;
elseif v(1) == v1(1) && v(2) > v1(2) && v(1) < v2(1) && ...
    v(2) == v2(2)
    rx = v(1) + cellsize;
    ry = v(2) - cellsize;
elseif v(1) < v1(1) && v(2) == v1(2) && v(1) == v2(1) &&...
    v(2) < v2(2)
    rx = v(1) + cellsize;
    ry = v(2) + cellsize;
elseif v(1) == v1(1) && v(2) < v1(2) && v(1) > v2(1) &&...
    v(2)== v2(2)
    rx = v(1) - cellsize;
    ry = v(2) + cellsize;
else
   % Only the preceding forms are valid arrangements of vertices.
   error('Vertex configuration is not valid.')
end

%-------------------------------------------------------------------%
function [I, newMPPV, W, B] = mppVtest(cMPPV, cV, classcV, cWH, cBL)
%      This function performs tests for existence of an MPP vertex.
%      The parameters are as follows (all except I and class_c_V) are
%      coordinate pairs of the form [x y]).
%     cMPPV          Current MPP vertex (the last MPP vertex found).
%     cV             Current vertex in the sequence.
%     classcV        Class of current vertex (+1 for convex
%                    and -1 for concave).
%     cWH            The current WHITE (convex) vertex.
%     cBL            The current BLACK (concave) vertex
%     I              If I = 1, a new MPP vertex was found
%     newMPPV        Next MPP vertex (if I = 1).
%     W              Next coordinates of WHITE.
%     B              Next coordinates of BLACK.
%
%   The details of the test are explained in Chapter 12 of the book.
```

```
% Preliminaries
I = 0;
newMPPV = [0 0];
W = cWH;
B = cBL;
sW = vsign(cMPPV, cWH, cV);
sB = vsign(cMPPV, cBL, cV);

% Perform test.
if sW > 0
   I = 1; % New MPP vertex found.
   newMPPV = cWH;
   W = newMPPV;
   B = newMPPV;
elseif sB < 0
   I = 1; % New MPP vertex found.
   newMPPV = cBL;
   W = newMPPV;
   B = newMPPV;
elseif (sW <= 0) && (sB >= 0)
   if classcV == 1
      W = cV;
   else
      B = cV;
   end
end
```

```
function [p, pmax, pmin, pn] = improd(f, g)
%IMPROD Compute the product of two images.
%   [P, PMAX, PMIN, PN] = IMPROD(F, G) outputs the element-by-element
%   product of two input images, F and G, the product maximum and
%   minimum values, and a normalized product array with values in the
%   range [0, 1]. The input images must be of the same size.  They
%   can be of class uint8, unit16, or double. The outputs are of
%   class double.
%
%   Sample M-file used in Chapter 2.

fd = double(f);
gd = double(g);
p = fd.*gd;
pmax = max(p(:));
pmin = min(p(:));
pn = mat2gray(p);
```

```
function cr = imratio(f1, f2)
%IMRATIO Computes the ratio of the bytes in two images/variables.
%   CR = IMRATIO(F1, F2) returns the ratio of the number of bytes in
%   variables/files F1 and F2. If F1 and F2 are an original and
%   compressed image, respectively, CR is the compression ratio.
```

```
error(nargchk(2, 2, nargin));          % Check input arguments
cr = bytes(f1) / bytes(f2);            % Compute the ratio

%-----------------------------------------------------------------------%
function b = bytes(f)
% Return the number of bytes in input f. If f is a string, assume
% that it is an image filename; if not, it is an image variable.

if ischar(f)
   info = dir(f);        b = info.bytes;
elseif isstruct(f)
   % MATLAB's whos function reports an extra 124 bytes of memory
   % per structure field because of the way MATLAB stores
   % structures in memory.  Don't count this extra memory; instead,
   % add up the memory associated with each field.
   b = 0;
   fields = fieldnames(f);
   for k = 1:length(fields)
      elements = f.(fields{k});
      for m = 1:length(elements)
          b = b + bytes(elements(m));
      end
   end
else
   info = whos('f');     b = info.bytes;
end
```

```
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. 12.29). 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 a column vector that contains the linear indices
%   of the locations of the vectors extracted from S.

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

```
% Find the linear indices of the 1-valued elements in MASK. Each
% element of R identifies the location in the M-by-N array of the
% vector extracted from S.
R = find(MASK);

% 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)).

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 );
```

```
function E = iseven(A)
%ISEVEN Determines which elements of an array are even numbers.
%   E = ISEVEN(A) returns a logical array, E, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to even numbers
%   in A, and 0s (FALSE) elsewhere.

%   STEVE: Needs copyright text block. Ralph

E = 2*floor(A/2) == A;
```

```
function D = isodd(A)
%ISODD Determines which elements of an array are odd numbers.
%   D = ISODD(A) returns a logical array, D, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to odd numbers in
%   A, and 0s (FALSE) elsewhere.

D = 2*floor(A/2) ~= A;
```

M

```
function movie2tifs(m, file)
%MOVIE2TIFS Creates a multiframe TIFF file from a MATLAB movie.
%   MOVIE2TIFS(M, FILE) creates a multiframe TIFF file from the
%   specified MATLAB movie structure, M.

% Write the first frame of the movie to the multiframe TIFF.
imwrite(frame2im(m(1)), file, 'Compression', 'none', ...
    'WriteMode', 'overwrite');

% Read the remaining frames and append to the TIFF file.
for i = 2:length(m)
    imwrite(frame2im(m(i)), file, 'Compression', 'none', ...
        'WriteMode', 'append');

end
```

P

```
function I = percentile2i(h, P)
%PERCENTILE2I Computes an intensity value given a percentile.
%       I = PERCENTILE2I(H, P) Given a percentile, P, and a histogram,
%       H, this function computes an intensity, I, representing the
```

```
%       Pth percentile and returns the value in I. P must be in the
%       range [0, 1] and I is returned as a value in the range [0, 1]
%       also.
%
%   Example:
%
%   Suppose that h is a uniform histogram of an 8-bit image.  Typing
%
%        I = percentile2i(h, 0.5)
%
%   would output I = 0.5. To convert to the (integer) 8-bit range
%   [0, 255], we let I = floor(255*I).
%
%   See also function i2percentile.

% Check value of P.
if P < 0 || P > 1
   error('The percentile must be in the range [0, 1].')
end

% Normalized the histogram to unit area. If it is already normalized
% the following computation has no effect.
h = h/sum(h);

% Cumulative distribution.
C = cumsum(h);

% Calculations.
idx = find(C >= P, 1, 'first');
% Subtract 1 from idx because indexing starts at 1, but intensities
% start at 0. Also, normalize to the range [0, 1].
I = (idx - 1)/(numel(h) - 1);
```

```
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 nonintegers.
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 ==y 2
   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

function flag = predicate(region)
%PREDICATE Evaluates a predicate for function splitmerge
%   FLAG = PREDICATE(REGION) evaluates a predicate for use in
%   function splitmerge for Example 11.14 in Digital Image
%   Processing Using MATLAB, 2nd edition. REGION is a subimage, and
%   FLAG is set to TRUE if the predicate evaluates to TRUE for
%   REGION; FLAG is set to FALSE otherwise.

% Compute the standard deviation and mean for the intensities of the
% pixels in REGION.
sd = std2(region);
m = mean2(region);

% Evaluate the predicate.
flag = (sd > 10) & (m > 0) & (m < 125);
```

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);
```

```
function H = recnotch(notch, mode, M, N, W, SV, SH)
%RECNOTCH Generates rectangular notch (axes) filters.
%   H = RECNOTCH(NOTCH, MODE, M, N, W, SV, SH) generates an M-by-N
%   notch filter consisting of symmetric pairs of rectangles of
%   width W placed on the vertical and horizontal axes of the
%   (centered) frequency rectangle. The vertical rectangles start at
%   +SV and -SV on the vertical axis and extend to both ends of the
%   axis. Horizontal rectangles similarly start at +SH and -SH and
%   extend to both ends of the axis. These values are with respect
%   to the origin of the axes of the centered frequency rectangle.
%   For example, specifying SV = 50 creates a rectangle of width W
%   that starts 50 pixels above the center of the vertical axis and
%   extends up to the first row of the filter. A similar rectangle
%   is created starting 50 pixels below the center and extending to
%   the last row. W must be an odd number to preserve the symmetry
%   of the filtered Fourier transform.
%
%       Valid values of NOTCH are:
```

```
%
%           'reject'    Notchreject filter.
%
%           'pass'      Notchpass filter.
%
%
%       Valid values of MODE are:
%
%           'both'          Filtering on both axes.
%
%           'horizontal'    Filtering on horizontal axis only.
%
%           'vertical'      Filtering on vertical axis only.
%
%       One of these three values must be specified in the call.
%
%   H = RECNOTCH(NOTCH, MODE, M, N) sets W = 1, and SV = SH = 1.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).

% Preliminaries.
if nargin == 4
   W = 1;
   SV = 1;
   SH = 1;
elseif nargin ~= 7
   error('The number of inputs must be 4 or 7.')
end
% AV and AH are rectangle amplitude values for the vertical and
% horizontal rectangles: 0 for notchreject and 1 for notchpass.
% Filters are computed initially as reject filters and then changed
% to pass if so specified in NOTCH.
if strcmp(mode, 'both')
   AV = 0;
   AH = 0;
elseif strcmp(mode, 'horizontal')
   AV = 1; % No reject filtering along vertical axis.
   AH = 0;
elseif strcmp(mode, 'vertical')
   AV = 0;
   AH = 1; % No reject filtering along horizontal axis.
end
if iseven(W)
   error('W must be an odd number.')
end

% Begin filter computation. The filter is generated as a reject
% filter. At the end, it are changed to a notchpass filter if it
```

```
% is so specified in parameter NOTCH.
H = rectangleReject(M, N, W, SV, SH, AV, AH);

% Finished computing the rectangle notch filter. Format the
% output.
H = processOutput(notch, H);

%-----------------------------------------------------------------%
function H = rectangleReject(M, N, W, SV, SH, AV, AH)
% Preliminaries.
H = ones(M, N, 'single');
% Center of frequency rectangle.
UC = floor(M/2) + 1;
VC = floor(N/2) + 1;
% Width limits.
WL = (W - 1)/2;
% Compute rectangle notches with respect to center.
% Left, horizontal rectangle.
H(UC-WL:UC+WL, 1:VC-SH) = AH;
% Right, horizontal rectangle.
H(UC-WL:UC+WL, VC+SH:N) = AH;
% Top vertical rectangle.
H(1:UC-SV, VC-WL:VC+WL) = AV;
% Bottom vertical rectangle.
H(UC+SV:M, VC-WL:VC+WL) = AV;

%-----------------------------------------------------------------%
function H = processOutput(notch, H)
% Uncenter the filter to make it compatible with other filters in
% the DIPUM toolbox.
H = ifftshift(H);
% Generate a pass filter if one was specified.
if strcmp(notch, 'pass')
   H = 1 - H;
end
```

S

```
function seq2tifs(s, file)
%SEQ2TIFS Creates a multi-frame TIFF file from a MATLAB sequence.

% Write the first frame of the sequence to the multiframe TIFF.
imwrite(s(:, :, :, 1), file, 'Compression', 'none', ...
    'WriteMode', 'overwrite');

% Read the remaining frames and append to the TIFF file.
for i = 2:size(s, 4)
    imwrite(s(:, :, :, i), file, 'Compression', 'none', ...
        'WriteMode', 'append');
end
```

```
function v = showmo(cv, i)
%SHOWMO Displays the motion vectors of a compressed image sequence.
%   SHOWMO(CV, I) displayes the motion vectors for frame I of a
%   TIFS2CV compressed sequence of images.
%
%   See also TIFS2CV and CV2TIFS.

frms = double(cv.frames);
m = double(cv.blksz);
q = double(cv.quality);

if q == 0
    ref = double(huff2mat(cv.video(1)));
else
    ref = double(jpeg2im(cv.video(1)));
end

fsz = size(ref);
mvsz = [fsz/m 2 frms];
mv = int16(huff2mat(cv.motion));
mv = reshape(mv, mvsz);
v = zeros(fsz, 'uint8') + 128;

% Create motion vector image.
for j = 1:mvsz(1) * mvsz(2)

    x1 = 1 + mod(m * (j - 1), fsz(1));
    y1 = 1 + m * floor((j - 1) * m / fsz(1));

    x2 = x1 - mv(1 + floor((x1 - 1) / m), ...
          1 + floor((y1 - 1) / m), 1, i);
    y2 = y1 - mv(1 + floor((x1 - 1) / m), ...
          1 + floor((y1 - 1) / m), 2, i);

    [x, y] = intline(x1, double(x2), y1, double(y2));
    for k = 1:length(x) - 1
        v(x(k), y(k)) = 255;
    end
    v(x(end), y(end)) = 0;
end

imshow(v);

function [dist, angle] = signature(b, x0, y0)
%SIGNATURE Computes the signature of a boundary.
%   [DIST, ANGLE, XC, YC] = SIGNATURE(B, X0, Y0) computes the
%   signature of a given boundary. A signature is defined as the
%   distance from (X0, Y0) to the boundary, as a function of angle
%   (ANGLE). B is an np-by-2 array (np > 2) containing the (x, y)
%   coordinates of the boundary ordered in a clockwise or
```

```
%   counterclockwise direction. If (X0, Y0) is not included in the
%   input argument, the centroid of the boundary is used by default.
%   The maximum size of arrays DIST 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
%   function bwboundaries.
%
%   If (X0, Y0) or the default centroid is outside the boundary, the
%   signature is not defined and an error is issued.

% 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, result 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 the origin of vector as the centroid, or use the two
% values specified. Use the same symbol (xc, yc) in case the user
% includes (xc, yc) in the output call.
if nargin == 1
   x0 = sum(b(:, 1))/np; % Coordinates of the centroid.
   y0 = sum(b(:, 2))/np;
end

% Check to see that (xc, yc) is inside the boundary.
IN = inpolygon(x0, y0, b(:, 1), b(:, 2));
if -IN
   error('(x0, y0) or centroid is not inside the boundary.')
end

% Shift origin of coordinate 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 (xcart, ycart). The two coordinate
% systems are related as follows:  xcart = y and ycart = -x.
xcart = b(:, 2);
ycart = -b(:, 1);
[theta, rho] = cart2pol(xcart, ycart);
```

```
% 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.

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

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

% Delete duplicate angles.  The unique operation also sorts the
% input in ascending order.
[w, u] = 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);

% Output the length values.
dist = 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;

function s = subim(f, m, n, rx, cy)
%SUBIM Extract subimage.
%   S = SUBIM(F, M, N, RX, CY) extracts a subimage, S, from the input
%   image, F.  The subimage is of size M-by-N, and the coordinates of
%   its top, left corner are (RX, CY).
%
%   Sample M-file used in Chapter 2.

s = zeros(m, n);
rowhigh = rx + m - 1;
colhigh = cy + n - 1;
xcount = 0;
for r = rx:rowhigh
   xcount = xcount + 1;
   ycount = 0;
   for c = cy:colhigh
      ycount = ycount + 1;
      s(xcount, ycount) = f(r, c);
   end
end
```

T

```
function m = tifs2movie(file)
%TIFS2MOVIE Create a MATLAB movie from a multiframe TIFF file.
%   M = TIFS2MOVIE(FILE) creates a MATLAB movie structure from a
%   multiframe TIFF file.

% Get file info like number of frames in the multi-frame TIFF
info = imfinfo(file);
frames = size(info, 1);

% Create a gray scale map for the UINT8 images in the MATLAB movie
gmap = linspace(0, 1, 256);
gmap = [gmap' gmap' gmap'];

% Read the TIFF frames and add to a MATLAB movie structure.
for i = 1:frames
    [f, fmap] = imread(file, i);
    if (strcmp(info(i).ColorType, 'grayscale'))
        map = gmap;
    else
        map = fmap;
    end
```

```
    m(i) = im2frame(f, map);
end

function s = tifs2seq(file)
%TIFS2SEQ Create a MATLAB sequence from a multi-frame TIFF file.

% Get the number of frames in the multi-frame TIFF.
frames = size(imfinfo(file), 1);

% Read the first frame, preallocate the sequence, and put the first
% in it.
i = imread(file, 1);
s = zeros([size(i) 1 frames], 'uint8');
s(:,:,:,1) = i;

% Read the remaining TIFF frames and add to the sequence.
for i = 2:frames
    s(:,:,:,i) = imread(file, i);
end

function [out, revertclass] = tofloat(in)
%TOFLOAT Convert image to floating point
%   [OUT, REVERTCLASS] = TOFLOAT(IN) converts the input image IN to
%   floating-point. If IN is a double or single image, then OUT
%   equals IN.  Otherwise, OUT equals IM2SINGLE(IN).  REVERTCLASS is
%   a function handle that can be used to convert back to the class
%   of IN.

identity = @(x) x;
tosingle = @im2single;

table = {'uint8',   tosingle, @im2uint8
   'uint16',  tosingle, @im2uint16
   'int16',   tosingle, @im2int16
   'logical', tosingle, @logical
   'double',  identity, identity
   'single',  identity, identity};

classIndex = find(strcmp(class(in), table(:, 1)));

if isempty(classIndex)
   error('Unsupported input image class.');
end

out = table{classIndex, 2}(in);

revertclass = table{classIndex, 3};

function [rt, f, g] = twodsin(A, u0, v0, M, N)
%TWODSIN Compare for-loops vs. vectorization.
```

```
%   The comparison is based on implementing the function f(x, y) =
%   Asin(u0x + v0y) for x = 0, 1, 2,..., M - 1 and y = 0, 1, 2,...,
%   N - 1. The inputs to the function are M and N and the constants
%   in the function.
%
%   Sample M-file used in Chapter 2.

% First implement using for loops.

tic   % Start timing.

for r = 1:M
   u0x = u0*(r - 1);
   for c = 1:N
      v0y = v0*(c - 1);
      f(r, c) = A*sin(u0x + v0y);
   end
end

t1 = toc;   % End timing.

%      Now implement using vectorization.  Call the image g.

tic   % Start timing.

r = 0:M - 1;
c = 0:N - 1;
[C, R] = meshgrid(c, r);
g = A*sin(u0*R + v0*C);

t2 = toc;   % End timing.

%      Compute the ratio of the two times.

rt = t1/(t2 + eps); % Use eps in case t2 is close to 0
```

W

```
function w = wave2gray(c, s, scale, border)
%WAVE2GRAY Display wavelet decomposition coefficients.
%   W = WAVE2GRAY(C, S, SCALE, BORDER) displays and returns a
%   wavelet coefficient image.
%
%   EXAMPLES:
%     wave2gray(c, s);                      Display w/defaults.
%     foo = wave2gray(c, s);                Display and return.
%     foo = wave2gray(c, s, 4);             Magnify the details.
%     foo = wave2gray(c, s, -4);            Magnify absolute values.
%     foo = wave2gray(c, s, 1, 'append');   Keep border values.
```

```
%
%   INPUTS/OUTPUTS:
%     [C, S] is a wavelet decomposition vector and bookkeeping
%     matrix.
%
%     SCALE       Detail coefficient scaling
%     ------------------------------------------------------------
%     0 or 1      Maximum range (default)
%     2,3...      Magnify default by the scale factor
%     -1, -2...   Magnify absolute values by abs(scale)
%
%     BORDER      Border between wavelet decompositions
%     ------------------------------------------------------------
%     'absorb'    Border replaces image (default)
%     'append'    Border increases width of image
%
%     Image W:   ------- ------ ------------ ------------------
%                |      |      |            |
%                | a(n) | h(n) |            |
%                |      |      |            |
%                ------- ------     h(n-1)   |
%                |      |      |            |
%                | v(n) | d(n) |            |        h(n-2)
%                |      |      |            |
%                ------- ------ ------------
%                |             |            |
%                |    v(n-1)   |   d(n-1)   |
%                |             |            |
%                ------------- ------------ ------------------
%                |                          |
%                |          v(n-2)          |        d(n-2)
%                |                          |
%
%     Here, n denotes the decomposition step scale and a, h, v, d are
%     approximation, horizontal, vertical, and diagonal detail
%     coefficients, respectively.

% Check input arguments for reasonableness.
error(nargchk(2, 4, nargin));

if (ndims(c) ~= 2) || (size(c, 1) ~= 1)
  error('C must be a row vector.');   end

if (ndims(s) ~= 2) || ~isreal(s) || ~isnumeric(s) || (size(s,2) ~= 2)
  error('S must be a real, numeric two-column array.');   end

elements = prod(s, 2);
if (length(c) < elements(end)) || ...
      ~(elements(1) + 3 * sum(elements(2:end - 1)) >= elements(end))
   error(['[C S] must be a standard wavelet ' ...
```

```
          'decomposition structure.']);
end

if (nargin > 2) && (~isreal(scale) || ~isnumeric(scale))
   error('SCALE must be a real, numeric scalar.');
end

if (nargin > 3) && (~ischar(border))
  error('BORDER must be character string.');
end

if nargin == 2
   scale = 1;  % Default scale.
end

if nargin < 4
   border = 'absorb';  % Default border.
end

% Scale coefficients and determine pad fill.
absflag = scale < 0;
scale = abs(scale);
if scale == 0
   scale = 1;
end

[cd, w] = wavecut('a', c, s);   w = mat2gray(w);
cdx = max(abs(cd(:))) / scale;
if absflag
   cd = mat2gray(abs(cd), [0, cdx]);   fill = 0;
else
   cd = mat2gray(cd, [-cdx, cdx]);   fill = 0.5;
end

% Build gray image one decomposition at a time.
for i = size(s, 1) - 2:-1:1
   ws = size(w);

   h = wavecopy('h', cd, s, i);
   pad = ws - size(h);     frontporch = round(pad / 2);
   h = padarray(h, frontporch, fill, 'pre');
   h = padarray(h, pad - frontporch, fill, 'post');

   v = wavecopy('v', cd, s, i);
   pad = ws - size(v);     frontporch = round(pad / 2);
   v = padarray(v, frontporch, fill, 'pre');
   v = padarray(v, pad - frontporch, fill, 'post');

   d = wavecopy('d', cd, s, i);
   pad = ws - size(d);     frontporch = round(pad / 2);
```

```
   d = padarray(d, frontporch, fill, 'pre');
   d = padarray(d, pad - frontporch, fill, 'post');

   % Add 1 pixel white border.
   switch lower(border)
   case 'append'
      w = padarray(w, [1 1], 1, 'post');
      h = padarray(h, [1 0], 1, 'post');
      v = padarray(v, [0 1], 1, 'post');
   case 'absorb'
      w(:, end) = 1;   w(end, :) = 1;
      h(end, :) = 1;   v(:, end) = 1;
   otherwise
      error('Unrecognized BORDER parameter.');
   end

   w = [w h; v d];                    % Concatenate coefs.
end

if nargout == 0
   imshow(w);                         % Display result.
end
}
```

X

```
function [C, theta] = x2majoraxis(A, B)
%X2MAJORAXIS Aligns coordinate x with the major axis of a region.
%   [C, THETA] = X2MAJORAXIS(A, B) 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]. 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. Output C 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 (pixel) coordinates, which
%   typically 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.
C = imrotate(B, theta, 'bilinear', 'crop');
```

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