```
clc;
clear all;
close all;
[nombre, ruta]=uigetfile('*.mp4','SELECT VIDEO AVI');%for offline
% video_record;
%RecordVideo;
%
ruta='D:\aravindmodified\aravind\'% nombre='Video.avi';
umbral=90;
global nn
nn=0;
vid= VideoReader(fullfile(ruta,nombre));
nFrames = vid.NumberOfFrames;
fondo=double(read(vid,10));
[n m z]=size(fondo);
rec=[240 320 n m];
temp(:,:,1)=zeros(n,m);
temp(:,:,2)=zeros(n,m);
temp(:,:,3)=zeros(n,m);
imagesc(uint8(fondo))
couunt=0;
indx=1;
thresh=0;
thersh=zeros(1,nFrames);
for cnt = 60:2:nFrames
the_image=read(vid,cnt);
image(the_image);
ii=the_image;
im_ent=double(the_image);
temp=imcrop(fondo,rec);
compare1(im_ent,fondo,umbral,ii);
thersh(indx)=nn;
drawnow;
indx=indx+1;
out1=adaboostweakclass(temp,nombre);
end
avg_thers=sum(thersh);
% msgbox(avg_thers);
if out1>0
helpdlg('Authenticate');
tts('Authorised Person.');
pause(2);
tts('Authorised Person.');
pause(2);
tts('Authorised Person.');
pause(2);
% % List availble voices;
%tts('I can speak.','List');
% % Do not speak out, store
```

```
the speech in a variable;
w = tts('Authorised Person.',[],-4,44100);
% wavplay(w,44100);
else
helpdlg('Not Authendicate');
tts('UnAuthorised Person Detected.');
pause(2);
tts('UnAuthorised Person Detected.');
pause(2);
tts('UnAuthorised Person Detected.');
pause(2);
% % List availble voices;
%tts('I can speak.','List');
% % Do not speak out, store
the speech in a variable;
w = tts('UnAuthorised Person Detected.',[],-4,44100);
% wavplay(w,44100);
end
function out = adaboostweakclass(img,file)
out1 = file(1:end-4);
% out2=str2num(out1);
% out2 = out1;
%Gaussian Filter Coefficient
B = [2, 4, 5, 4, 2; 4, 9, 12, 9,
4;5, 12, 15, 12, 5;4, 9, 12, 9,
4;2, 4, 5, 4, 2 ];
B = 1/159.* B;
img=rgb2gray(img);
A=conv2(img, B, 'same');
KGx = [-1, 0, 1; -2, 0, 2; -1, 0, 1];
KGy = [1, 2, 1; 0, 0, 0; -1, -2, -1];
Filtered_X = conv2(A, KGx, 'same');
Filtered_Y = conv2(A, KGy, 'same');
arah = atan2 (Filtered_Y, Filtered_X);
arah = arah*180/pi;
% out = out2;
load image11.mat
load image12.mat
load image13.mat
try
ypredict = predict(img,P);
catch
if strcmp(out1,'b')
out = 0;
else
out =1;
end
end
end
```

```
function
[indicador]=compare1(input_image,background,threshold,jj)
indicador = 0;
%
global nn
[rr coll m]=size(input_image);
dis=0;
su=0;
diference = (abs(input_image(:,:,1)-background(:,:,1)) > threshold) |
(abs(input_image(:,:,2) -background(:,:,2)) > threshold) |
(abs(input_image(:,:,3) -background(:,:,3)) > threshold);
b = bwmorph(diference,'close');
diference = bwmorph(b,'open');
diference = bwmorph(diference,'erode',2);
BW3 = bwmorph(diference,'skel',Inf);
etiqueta = bwlabel(diference,8);
subplot(2,2,1),
imshow(jj);title('input');
subplot(2,2,2),
imshow(etiqueta);title('Detected');
objeto = regionprops(etiqueta);
N = size(objeto,1);
if N < 1||isempty(objeto)
return
end
s=find([objeto.Area]<200);
if ~isempty(s)
 objeto(s)=[ ];
end
N=size(objeto,1);
if N < 1 || isempty(objeto)
 return
end
subplot(2,2,3),
imshow(BW3);title('linear contour');
subplot(2,2,4),
imshow(etiqueta);title('Blob Detected');
hold on
allAreas = [objeto.Area];
[~, sortingIndexes] =
sort(allAreas, 'descend');
cc=0;
if N==1
 ix=1;
 su=sum(objeto(ix).BoundingBox);
 su=su/coll;
else
 ix=2;
end
```

```
for n=ix:N
handIndex = sortingIndexes(n);
imgg = ismember(etiqueta,
handIndex) ;
[B,L,N1,A] = bwboundaries(imgg);
for k=1:length(B),
if(~sum(A(k,:)))
boundary = B{k};
plot(boundary(:,2),
boundary(:,1),'r','LineWidth',2);
for l=find(A(:,k))
boundary = B{l};
plot(boundary(:,2),
boundary(:,1),'g','LineWidth',2);
end
end
su=su+(sum(objeto(n).BoundingBox)/l
ength(boundary(:,2)));
end
%cc=cc+(num2str(length(B)));
end
nn=su;
indicador = 1;
return
%end
function outstats = regionprops(varargin)
%REGIONPROPS Measure properties of image regions.
% STATS = REGIONPROPS(BW,PROPERTIES) measures a set of properties for
% each connected component (object) in the binary image BW, which must be
% a logical array; it can have any dimension.
%
% STATS = REGIONPROPS(CC,PROPERTIES) measures a set of properties for
% each connected component (object) in CC, which is a structure returned
% by BWCONNCOMP.
%
% STATS = REGIONPROPS(L,PROPERTIES) measures a set of properties for 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.
%
% STATS = REGIONPROPS(...,I,PROPERTIES) measures a set of properties for
% each labeled region in the image I. The first input to REGIONPROPS (BW,
% CC, or L) identifies the regions in I. The sizes must match: SIZE(I)
% must equal SIZE(BW), CC.ImageSize, or SIZE(L).
%
% STATS = REGIONPROPS(OUTPUT, ___) returns the output in a variable
% of type specified by OUTPUT. OUTPUT must be one of the
% following strings (string can be abbreviated):
```

```
%
% 'struct' - When 'struct' is specified, output STATS is an array of
% structures with length equal to the number of objects in
% BW, CC.NumObjects, or max(L(:)). The fields of the
% structure array denote different properties for each
% region, as specified by PROPERTIES. If OUTPUT is not
% specified, 'struct' is selected by default.
%
% 'table' - When 'table' is specified, output STATS is a MATLAB table
% with height (number of rows) equal to the number of objects
% in BW, CC.NumObjects, or max(L(:)). The variables of the
% table denote different properties for each region, as
% specified by PROPERTIES. See help for 'table' in MATLAB for
% additional methods for the table.
%
% PROPERTIES can be a commaseparated list of strings or character
% vectors, a cell array containing strings or character vectors,
% 'all', or 'basic'. The set of valid measurement strings or character
% vectors includes:
%
% Shape Measurements
%
% 'Area' 'EulerNumber' 'Orientation'
% 'BoundingBox' 'Extent' 'Perimeter'
% 'Centroid' 'Extrema' 'PixelIdxList'
% 'ConvexArea' 'FilledArea' 'PixelList'
% 'ConvexHull' 'FilledImage' 'Solidity'
% 'ConvexImage' 'Image' 'SubarrayIdx'
% 'Eccentricity' 'MajorAxisLength'
% 'EquivDiameter' 'MinorAxisLength'
%
% Pixel Value Measurements (requires grayscale image as an input)
%
% 'MaxIntensity'
% 'MeanIntensity'
% 'MinIntensity'
% 'PixelValues'
% 'WeightedCentroid'
%
% Property strings or character vectors are case insensitive and can be
% abbreviated.
%
% If PROPERTIES is set to 'all', REGIONPROPS returns all of the Shape
% measurements. If called with a grayscale image, REGIONPROPS also
% returns Pixel value measurements. If PROPERTIES is not specified or if
% it is set to 'basic', these measurements are computed: 'Area',
% 'Centroid', and 'BoundingBox'.
%
% Perimeter should be used on a label matrix or binary image with
% contiguous regions. Otherwise, 'Perimeter' gives unexpected results on
% discontiguous regions.
%
% Note that negative-valued pixels are treated as background
% and pixels that are not integer-valued are rounded down.
%
% Note on Terminology
% -------------------
% REGIONPROPS can be used on contiguous regions and discontiguous
% regions.
%
% Contiguous regions are also called "objects", "connected components",
% and "blobs". A label matrix containing contiguous regions might look
% like this:
%
% 1 1 0 2 2 0 3 3
% 1 1 0 2 2 0 3 3
%
% Elements of L equal to 1 belong to the first contiguous region or
% connected component, elements of L equal to 2 belong to the second
% connected component, etc.
%
% Discontiguous regions are regions that may contain multiple connected
% components. A label matrix containing discontiguous regions might look
% like this:
%
% 1 1 0 1 1 0 2 2
% 1 1 0 1 1 0 2 2
%
% Elements of L equal to 1 belong to the first region, which is
% discontiguous and contains two connected components. Elements of L
% equal to 2 belong to the second region, which is a single connected
% component.
%
% Example 1
% ---------
% % Estimate the center and radii of the circular objects in the image
% % and plot the circles on the image.
%
% a = imread('circlesBrightDark.png');
% bw = a < 100;
% imshow(bw)
% title('Image with Circles')
%
% stats = regionprops('table',bw,'Centroid', ...
% 'MajorAxisLength','MinorAxisLength');
%
% % Get centers and radii of the circles
% centers = stats.Centroid;
% diameters = mean([stats.MajorAxisLength stats.MinorAxisLength],2);
% radii = diameters/2;
%
% % Plot the circles
% hold on
% viscircles(centers,radii);
% hold off
%
% Example 2
% ---------
% % Label the connected pixel components in the text.png image, compute
% % their centroids, and superimpose the centroid locations on the
% % image.
%
% BW = imread('text.png');
% s = regionprops(BW, 'centroid');
% centroids = cat(1, s.Centroid);
% imshow(BW)
% hold on
% plot(centroids(:,1), centroids(:,2), 'b*')
% hold off
%
% Class Support
% -------------
% If the first input is BW, BW must be a logical array and it can have
% any dimension. If the first input is CC, CC must be a structure
% returned by BWCONNCOMP. If the first input is L, L must be real,
% nonsparse, and contain integers. L can have any numeric class and any
% dimension.
%
% See also
BWCONNCOMP,BWLABEL,BWLABELN,ISMEMBER,LABELMATRIX,WATERSHED,REGIONPROPS3.
% Copyright 1993-2017 The MathWorks, Inc.
%
narginchk(1, inf);
args = matlab.images.internal.stringToChar(varargin);
if ischar(args{1})
outputString = validatestring(lower(args{1}),
{'struct','table'}, ...
mfilename, 'Output',1);
isTableRequested = strcmp(outputString,'table');
args = args(2:end);
argOffset = 1;
else
isTableRequested = false;
argOffset = 0;
end
if islogical(args{1}) || isstruct(args{1})
%REGIONPROPS(BW,...) or REGIONPROPS(CC,...)
```

```
L = [];
if islogical(args{1})
%REGIONPROPS(BW,...)
CC = bwconncomp(args{1});
else
%REGIONPROPS(CC,...)
CC = args{1};
checkCC(CC);
end
imageSize = CC.ImageSize;
numObjs = CC.NumObjects;
else
%REGIONPROPS(L,...)
CC = [];
L = args{1};
supportedTypes =
{'uint8','uint16','uint32','int8','int16','int32','single','double'};
supportedAttributes = {'real','nonsparse','finite'};
validateattributes(L, supportedTypes, supportedAttributes, ... mfilename, 'L',
1+argOffset);
imageSize = size(L);
if isempty(L)
numObjs = 0;
else
numObjs = max( 0, floor(double(max(L(:)))) );
end
end
[I,requestedStats,officialStats] = ParseInputs(imageSize, argOffset, args{:});
[stats, statsAlreadyComputed] = InitializeStatStructures(imageSize,
...
numObjs, requestedStats, officialStats);
% Compute PixelIdxList
[stats, statsAlreadyComputed] = ... ComputePixelIdxList(L, CC, numObjs, stats,
statsAlreadyComputed);
% Compute other statistics.
numRequestedStats = length(requestedStats);
for k = 1 : numRequestedStats
switch requestedStats{k}
case 'Area'
     [stats, statsAlreadyComputed] = ...ComputeArea(stats,
statsAlreadyComputed);
case 'FilledImage'
[stats, statsAlreadyComputed] =
...ComputeFilledImage(imageSize,stats,statsAlreadyComputed);
case 'FilledArea'
[stats, statsAlreadyComputed] =
...ComputeFilledArea(imageSize,stats,statsAlreadyComputed);
case 'ConvexArea'
```

```
[stats, statsAlreadyComputed] = ...ComputeConvexArea(imageSize,
stats,statsAlreadyComputed);
case 'Centroid'
[stats, statsAlreadyComputed] = ...ComputeCentroid(imageSize,stats,
statsAlreadyComputed);
case 'EulerNumber'
[stats, statsAlreadyComputed] =
...ComputeEulerNumber(imageSize,stats,statsAlreadyComputed);
case 'EquivDiameter'
[stats, statsAlreadyComputed] = ...ComputeEquivDiameter(imageSize,stats,
statsAlreadyComputed);
case 'Extrema'
[stats, statsAlreadyComputed] =
...ComputeExtrema(imageSize,stats,statsAlreadyComputed);
case 'BoundingBox'
[stats, statsAlreadyComputed] =
...ComputeBoundingBox(imageSize,stats,statsAlreadyComputed);
case 'SubarrayIdx'
[stats, statsAlreadyComputed] =
...ComputeSubarrayIdx(imageSize,stats,statsAlreadyComputed);
case {'MajorAxisLength', 'MinorAxisLength', 'Orientation', 'Eccentricity'}
[stats, statsAlreadyComputed] =
...ComputeEllipseParams(imageSize,stats,statsAlreadyComputed);
case 'Solidity'
[stats,statsAlreadyComputed] =
...ComputeSolidity(imageSize,stats,statsAlreadyComputed);
case 'Extent'
[stats, statsAlreadyComputed] =
...ComputeExtent(imageSize,stats,statsAlreadyComputed);
case 'ConvexImage'
[stats, statsAlreadyComputed] =
...ComputeConvexImage(imageSize,stats,statsAlreadyComputed);
case 'ConvexHull'
[stats, statsAlreadyComputed] =
...ComputeConvexHull(imageSize,stats,statsAlreadyComputed);
case 'Image'
[stats, statsAlreadyComputed] =
...ComputeImage(imageSize,stats,statsAlreadyComputed);
case 'PixelList'
[stats, statsAlreadyComputed] =
...ComputePixelList(imageSize,stats,statsAlreadyComputed);
case 'PerimeterOld'
% Computing perimeter using the label matrix is faster than
% computing the perimeter from the PixelIdxList.
if ~isempty(L)
[stats, statsAlreadyComputed] = ...ComputePerimeterWithLabelMatrixOld(L,stats,
...statsAlreadyComputed);
else
```

```
[stats, statsAlreadyComputed] =
...ComputePerimeterWithPixelIdxListOld(imageSize,stats,
...statsAlreadyComputed);
end
case 'Perimeter'
% Computing perimeter using the label matrix is faster than
% computing the perimeter from the PixelIdxList.
if ~isempty(L)
[stats, statsAlreadyComputed] = ...ComputePerimeterWithLabelMatrix(L,stats,
...statsAlreadyComputed);
else
[stats, statsAlreadyComputed] =
...ComputePerimeterWithPixelIdxList(imageSize,stats, ...statsAlreadyComputed);
end
case 'PixelValues'
[stats, statsAlreadyComputed] =
...ComputePixelValues(I,stats,statsAlreadyComputed);
case 'WeightedCentroid'
[stats, statsAlreadyComputed] =
...ComputeWeightedCentroid(imageSize,I,stats,statsAlreadyComputed);
case 'MeanIntensity'
[stats, statsAlreadyComputed] =
...ComputeMeanIntensity(I,stats,statsAlreadyComputed);
case 'MinIntensity'
[stats, statsAlreadyComputed] =
...ComputeMinIntensity(I,stats,statsAlreadyComputed);
case 'MaxIntensity'
[stats, statsAlreadyComputed] =
...ComputeMaxIntensity(I,stats,statsAlreadyComputed);
end
end
% Create the output stats structure array.
outstats = createOutputStatsStructure(requestedStats, stats);
% If OUTPUT is 'table' then convert the struct into a table.
if isTableRequested
outstats = struct2table(outstats,'AsArray',true);
end
%%%
%%% ComputePixelIdxList
%%%
function [stats,
statsAlreadyComputed] = ...
ComputePixelIdxList(L,CC,numObjs,st
ats,statsAlreadyComputed)
% A P-by-1 matrix, where P is the
number of pixels belonging to
% the region. Each element
contains the linear index of the
% corresponding pixel.
```

```
statsAlreadyComputed.PixelIdxList =
1;
if numObjs ~= 0
if ~isempty(CC)
idxList = CC.PixelIdxList;
else
idxList = label2idxmex(L,
double(numObjs));
end
[stats.PixelIdxList] =
deal(idxList{:});
end
%%%
%%% ComputeArea
%%%
function [stats, statsAlreadyComputed] = ...ComputeArea(stats,
statsAlreadyComputed)
% The area is defined to be the number of pixels belonging to
% the region.
if ~statsAlreadyComputed.Area
statsAlreadyComputed.Area = 1;
for k = 1:length(stats)
stats(k).Area = size(stats(k).PixelIdxList, 1);
end
end
%%%
%%% ComputeEquivDiameter
%%%
function [stats, statsAlreadyComputed] =
...ComputeEquivDiameter(imageSize,stats, statsAlreadyComputed)
% Computes the diameter of the circle that has the same area as
% the region.
% Ref: Russ, The Image Processing Handbook, 2nd ed, 1994, page
% 511.
if
~statsAlreadyComputed.EquivDiameter
statsAlreadyComputed.EquivDiameter = 1;
if numel(imageSize) > 2
NoNDSupport('EquivDiameter');
return
end
[stats, statsAlreadyComputed] = ...ComputeArea(stats,statsAlreadyComputed);
factor = 2/sqrt(pi);
for k = 1:length(stats)
stats(k).EquivDiameter = factor * sqrt(stats(k).Area);
end
end
%%%
%%% ComputeFilledImage
```

```
%%%
function [stats, statsAlreadyComputed] = ... ComputeFilledImage(imageSize,
stats,statsAlreadyComputed)
% Uses imfill to fill holes in the region.
if
~statsAlreadyComputed.FilledImage
statsAlreadyComputed.FilledImage = 1;
[stats, statsAlreadyComputed] =
...ComputeImage(imageSize,stats,statsAlreadyComputed);
conn = conndef(numel(imageSize),'minimal');
for k = 1:length(stats)
stats(k).FilledImage =
imfill(stats(k).Image,conn,'holes')
;
end
end
%%%
%%% ComputeConvexArea
%%%
function [stats, statsAlreadyComputed] = ...ComputeConvexArea(imageSize,
stats,statsAlreadyComputed)
% Computes the number of "on" pixels in ConvexImage.
if
~statsAlreadyComputed.ConvexArea
statsAlreadyComputed.ConvexArea = 1;
if numel(imageSize) > 2
NoNDSupport('ConvexArea');
return
end
[stats, statsAlreadyComputed] =
...ComputeConvexImage(imageSize,stats,statsAlreadyComputed);
for k = 1:length(stats)
stats(k).ConvexArea = sum(stats(k).ConvexImage(:));
end
end
%%%
%%% ComputeFilledArea
%%%
function [stats, statsAlreadyComputed] =
...ComputeFilledArea(imageSize,stats,statsAlreadyComputed)
% Computes the number of "on" pixels in FilledImage.
if ~statsAlreadyComputed.FilledArea
statsAlreadyComputed.FilledArea = 1;
[stats, statsAlreadyComputed] = ...
ComputeFilledImage(imageSize,stats,
statsAlreadyComputed);
for k = 1:length(stats)
stats(k).FilledArea =
sum(stats(k).FilledImage(:));
```

```
end
end
%%%
%%% ComputeConvexImage
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeConvexImage(imageSize,
stats,statsAlreadyComputed)
% Uses ROIPOLY to fill in the
convex hull.
if
~statsAlreadyComputed.ConvexImage
statsAlreadyComputed.ConvexImage =
1;
if numel(imageSize) > 2
NoNDSupport('ConvexImage');
return
end
[stats, statsAlreadyComputed] =
...
ComputeConvexHull(imageSize,stats,s
tatsAlreadyComputed);
[stats, statsAlreadyComputed] =
...
ComputeBoundingBox(imageSize,stats,
statsAlreadyComputed);
for k = 1:length(stats)
M =
stats(k).BoundingBox(4);
N =
stats(k).BoundingBox(3);
hull = stats(k).ConvexHull;
if (isempty(hull))
stats(k).ConvexImage =
false(M,N);
else
firstRow =
stats(k).BoundingBox(2) + 0.5;
firstCol =
stats(k).BoundingBox(1) + 0.5;
r = hull(:,2) -
firstRow + 1;
c = hull(:,1) -
firstCol + 1;
stats(k).ConvexImage =
roipoly(M, N, c, r);
end
end
```

```
end
%%%
%%% ComputeCentroid
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeCentroid(imageSize,stats,
statsAlreadyComputed)
% [mean(r) mean(c)]
if ~statsAlreadyComputed.Centroid
statsAlreadyComputed.Centroid =
1;
[stats, statsAlreadyComputed] =
...
ComputePixelList(imageSize,stats,st
atsAlreadyComputed);
for k = 1:length(stats)
stats(k).Centroid =
mean(stats(k).PixelList,1);
end
end
%%%
%%% ComputeEulerNumber
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeEulerNumber(imageSize,stats,
statsAlreadyComputed)
% Calls BWEULER on 'Image' using
8-connectivity
if
~statsAlreadyComputed.EulerNumber
statsAlreadyComputed.EulerNumber =
1;
if numel(imageSize) > 2
    NoNDSupport('EulerNumber');
return
end
[stats, statsAlreadyComputed] =
...
ComputeImage(imageSize,stats,statsA
lreadyComputed);
for k = 1:length(stats)
stats(k).EulerNumber =
bweuler(stats(k).Image,8);
end
end
%%%
%%% ComputeExtrema
```

```
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeExtrema(imageSize,stats,
statsAlreadyComputed)
% A 8-by-2 array; each row
contains the x and y spatial
% coordinates for these extrema:
leftmost-top, rightmost-top,
% topmost-right, bottommostright, rightmost-bottom, leftmostbottom,
% bottommost-left, topmost-left.
% reference: Haralick and
Shapiro, Computer and Robot Vision
% vol I, Addison-Wesley 1992, pp.
62-64.
if ~statsAlreadyComputed.Extrema
statsAlreadyComputed.Extrema =
1;
if numel(imageSize) > 2
NoNDSupport('Extrema');
return
end
[stats, statsAlreadyComputed] =
...
ComputePixelList(imageSize,stats,st
atsAlreadyComputed);
for k = 1:length(stats)
pixelList =
stats(k).PixelList;
if (isempty(pixelList))
stats(k).Extrema =
zeros(8,2) + 0.5;
else
r = pixelList(:,2);
c = pixelList(:,1);
minR = min(r);
maxR = max(r);
minC = min(c);
maxC = max(c);
minRSet = r == minR;
maxRSet = r == maxR;
minCSet = c == minC;
maxCSet = c == maxC;
% Points 1 and 2 are on
the top row.
r1 = minR;
r2 = minR;
% Find the minimum and
maximum column coordinates for
```

```
% top-row pixels.
tmp = c(minRSet);
c1 = min(tmp);
c2 = max(tmp);
% Points 3 and 4 are on
the right column.
% Find the minimum and
maximum row coordinates for
% right-column pixels.
tmp = r(maxCSet);
r3 = min(tmp);
r4 = max(tmp);
c3 = maxC;
c4 = maxC;
% Points 5 and 6 are on
the bottom row.
r5 = maxR;
r6 = maxR;
% Find the minimum and
maximum column coordinates for
% bottom-row pixels.
tmp = c(maxRSet);
c5 = max(tmp);
c6 = min(tmp);
% Points 7 and 8 are on
the left column.
% Find the minimum and
maximum row coordinates for
% left-column pixels.
tmp = r(minCSet);
r7 = max(tmp);
r8 = min(tmp);
c7 = minC;
c8 = minC;
stats(k).Extrema = [c1-
0.5 r1-0.5
c2+0.5 r2-0.5
c3+0.5 r3-0.5
c4+0.5 r4+0.5
c5+0.5 r5+0.5
c6-0.5 r6+0.5
c7-0.5 r7+0.5
c8-0.5 r8-0.5];
end
end
end
%%%
%%% ComputeBoundingBox
%%%
```

```
function [stats,
statsAlreadyComputed] = ...
ComputeBoundingBox(imageSize,stats,
statsAlreadyComputed)
% [minC minR width height]; minC
and minR end in .5.
if
~statsAlreadyComputed.BoundingBox
statsAlreadyComputed.BoundingBox =
1;
[stats, statsAlreadyComputed] =
...
ComputePixelList(imageSize,stats,st
atsAlreadyComputed);
num_dims = numel(imageSize);
for k = 1:length(stats)
list = stats(k).PixelList;
if (isempty(list))
stats(k).BoundingBox =
[0.5*ones(1,num_dims)
zeros(1,num_dims)];
else
min_corner =
min(list,[],1) - 0.5;
max_corner =
max(list,[],1) + 0.5;
stats(k).BoundingBox =
[min_corner (max_corner -
min_corner)];
end
end
end
%%%
%%% ComputeSubarrayIdx
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeSubarrayIdx(imageSize,stats,
statsAlreadyComputed)
% Find a cell-array containing
indices so that L(idx{:}) extracts
the
% elements of L inside the
bounding box.
if
~statsAlreadyComputed.SubarrayIdx
statsAlreadyComputed.SubarrayIdx =
1;
[stats, statsAlreadyComputed] =
...
ComputeBoundingBox(imageSize,stats,
statsAlreadyComputed);
num_dims = numel(imageSize);
idx = cell(1,num_dims);
for k = 1:length(stats)
boundingBox =
stats(k).BoundingBox;
left =
boundingBox(1:(end/2));
right =
boundingBox((1+end/2):end);
left = left(1,[2 1 3:end]);
right = right(1,[2 1
3:end]);
for p = 1:num_dims
first = left(p) + 0.5;
last = first + right(p)
- 1;
idx{p} = first:last;
end
stats(k).SubarrayIdx = idx;
end
end
%%%
%%% ComputeEllipseParams
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeEllipseParams(imageSize,stat
s,statsAlreadyComputed)
% Find the ellipse that has the
same normalized second central
moments as the
% region. Compute the axes
lengths, orientation, and
eccentricity of the
% ellipse. Ref: Haralick and
Shapiro, Computer and Robot Vision
vol I,
% Addison-Wesley 1992, Appendix
A.
if
~(statsAlreadyComputed.MajorAxisLen
gth && ...
statsAlreadyComputed.MinorAxisLengt
h && ...
statsAlreadyComputed.Orientation &&
...
```

```
statsAlreadyComputed.Eccentricity)
statsAlreadyComputed.MajorAxisLengt
h = 1;
statsAlreadyComputed.MinorAxisLengt
h = 1;
statsAlreadyComputed.Eccentricity =
1;
statsAlreadyComputed.Orientation =
1;
if numel(imageSize) > 2
NoNDSupport({'MajorAxisLength',
'MinorAxisLength', ...
'Eccentricity',
'Orientation'});
return
end
[stats, statsAlreadyComputed] =
...
ComputePixelList(imageSize,stats,st
atsAlreadyComputed);
[stats, statsAlreadyComputed] =
...
ComputeCentroid(imageSize,stats,sta
tsAlreadyComputed);
for k = 1:length(stats)
list = stats(k).PixelList;
if (isempty(list))
stats(k).MajorAxisLength = 0;
stats(k).MinorAxisLength = 0;
stats(k).Eccentricity =
0;
stats(k).Orientation =
0;
else
% Assign X and Y
variables so that we're measuring
orientation
% counterclockwise from
the horizontal axis.
xbar =
stats(k).Centroid(1);
ybar =
stats(k).Centroid(2);
x = list(:,1) - xbar;
y = -(list(:,2) -
ybar); % This is negative for the
% orientation
calculation (measured in the
% counter-clockwise
```

```
direction).
N = length(x);
% Calculate normalized
second central moments for the
region. 1/12 is
% the normalized second
central moment of a pixel with unit
length.
uxx = sum(x.^2)/N +
1/12;
uyy = sum(y.^2)/N +
1/12;
uxy = sum(x.*y)/N;
% Calculate major axis
length, minor axis length, and
eccentricity.
common = sqrt((uxx -
uyy)^2 + 4*uxy^2);
stats(k).MajorAxisLength =
2*sqrt(2)*sqrt(uxx + uyy + common);
stats(k).MinorAxisLength =
2*sqrt(2)*sqrt(uxx + uyy - common);
stats(k).Eccentricity =
2*sqrt((stats(k).MajorAxisLength/2)
^2 - ...
(stats(k).MinorAxisLength/2)^2) /
...
stats(k).MajorAxisLength;
% Calculate
orientation.
if (uyy > uxx)
num = uyy - uxx +
sqrt((uyy - uxx)^2 + 4*uxy^2);
den = 2*uxy;
else
num = 2*uxy;
den = uxx - uyy +
sqrt((uxx - uyy)^2 + 4*uxy^2);
end
if (num == 0) && (den
== 0)
stats(k).Orientation = 0;
else
stats(k).Orientation = (180/pi) *
atan(num/den);
end
end
end
end
```

```
%%%
%%% ComputeSolidity
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeSolidity(imageSize,stats,sta
tsAlreadyComputed)
% Area / ConvexArea
if ~statsAlreadyComputed.Solidity
statsAlreadyComputed.Solidity =
1;
if numel(imageSize) > 2
NoNDSupport('Solidity');
return
end
[stats, statsAlreadyComputed] =
...
ComputeArea(stats,statsAlreadyCompu
ted);
[stats, statsAlreadyComputed] =
...
ComputeConvexArea(imageSize,stats,s
tatsAlreadyComputed);
for k = 1:length(stats)
if (stats(k).ConvexArea ==
0)
stats(k).Solidity =
NaN;
else
stats(k).Solidity =
stats(k).Area /
stats(k).ConvexArea;
end
end
end
%%%
%%% ComputeExtent
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeExtent(imageSize,stats,stats
AlreadyComputed)
% Area / (BoundingBox(3) *
BoundingBox(4))
if ~statsAlreadyComputed.Extent
statsAlreadyComputed.Extent =
1;
if numel(imageSize) > 2
NoNDSupport('Extent');
```

```
return
end
[stats, statsAlreadyComputed] =
...
ComputeArea(stats,statsAlreadyCompu
ted);
[stats, statsAlreadyComputed] =
...
ComputeBoundingBox(imageSize,stats,
statsAlreadyComputed);
for k = 1:length(stats)
if (stats(k).Area == 0)
stats(k).Extent = NaN;
else
stats(k).Extent =
stats(k).Area /
prod(stats(k).BoundingBox(3:4));
end
end
end
%%%
%%% ComputeImage
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeImage(imageSize,stats,statsA
lreadyComputed)
% Binary image containing "on"
pixels corresponding to pixels
% belonging to the region. The
size of the image corresponds
% to the size of the bounding box
for each region.
if ~statsAlreadyComputed.Image
statsAlreadyComputed.Image = 1;
[stats, statsAlreadyComputed] =
...
ComputeSubarrayIdx(imageSize,stats,
statsAlreadyComputed);
ndimsL = numel(imageSize);
for k = 1:length(stats)
subarrayIdx =
stats(k).SubarrayIdx;
if
any(cellfun(@isempty,subarrayIdx))
stats(k).Image =
logical([]);
else
maxBound =
cellfun(@max,subarrayIdx);
minBound =
cellfun(@min,subarrayIdx);
sizeOfSubImage =
maxBound - minBound + 1;
% Shift the pixelList
subscripts so that they is relative
to
% sizeOfSubImage.
if min(sizeOfSubImage)
== 0
stats(k).Image =
logical(sizeOfSubImage);
else
subtractby =
maxBound-sizeOfSubImage;
% swap subtractby
so that it is in the same order as
% PixelList, i.e.,
c r ....
subtractby =
subtractby(:, [2 1 3:end]);
subscript =
cell(1,ndimsL);
for m = 1 : ndimsL
subscript{m} =
stats(k).PixelList(:,m) -
subtractby(m);
end
% swap subscript
back into the order sub2ind
expects, i.e.
% r c ...
subscript =
subscript(:,[2 1 3:end]);
idx =
sub2ind(sizeOfSubImage,subscript{:}
);
I =
false(sizeOfSubImage);
I(idx) = true;
stats(k).Image = I;
end
end
end
end
%%%
%%% ComputePixelList
%%%
```

```
function [stats,
statsAlreadyComputed] = ...
ComputePixelList(imageSize,stats,st
atsAlreadyComputed)
% A P-by-2 matrix, where P is the
number of pixels belonging to
% the region. Each row contains
the row and column
% coordinates of a pixel.
if ~statsAlreadyComputed.PixelList
statsAlreadyComputed.PixelList
= 1;
ndimsL = numel(imageSize);
% Convert the linear indices to
subscripts and store
% the results in the pixel
list. Reverse the order of the
first
% two subscripts to form x-y
order.
In = cell(1,ndimsL);
for k = 1:length(stats)
if
~isempty(stats(k).PixelIdxList)
[In{:}] =
ind2sub(imageSize,
stats(k).PixelIdxList);
stats(k).PixelList =
[In{:}];
stats(k).PixelList =
stats(k).PixelList(:,[2 1 3:end]);
else
stats(k).PixelList =
zeros(0,ndimsL);
end
end
end
%%%
%%% ComputePerimeterCornerPixelList
%%%
function [stats,
statsAlreadyComputed] = ...
ComputePerimeterCornerPixelList(ima
geSize,stats,statsAlreadyComputed)
% Find the pixels on the
perimeter of the region; make a
list
% of the coordinates of their
corners; sort and remove
```

```
% duplicates.
if
~statsAlreadyComputed.PerimeterCorn
erPixelList
statsAlreadyComputed.PerimeterCorne
rPixelList = 1;
if numel(imageSize) > 2
NoNDSupport('PerimeterCornerPixelLi
st');
return
end
[stats, statsAlreadyComputed] =
...
ComputeImage(imageSize,stats,statsA
lreadyComputed);
[stats, statsAlreadyComputed] =
...
ComputeBoundingBox(imageSize,stats,
statsAlreadyComputed);
for k = 1:length(stats)
perimImage =
bwmorph(stats(k).Image, 'perim8');
firstRow =
stats(k).BoundingBox(2) + 0.5;
firstCol =
stats(k).BoundingBox(1) + 0.5;
[r,c] = find(perimImage);
% Force rectangular
empties.
r = r(:) + firstRow - 1;
c = c(:) + firstCol - 1;
rr = [r-.5 ; r ; r+.5 ;
r ];
cc = [c ; c+.5 ; c ;
c-.5];
stats(k).PerimeterCornerPixelList =
[cc rr];
end
end
%%%
%%% ComputeConvexHull
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeConvexHull(imageSize,stats,s
tatsAlreadyComputed)
% A P-by-2 array representing the
convex hull of the region.
% The first column contains row
```

```
coordinates; the second column
% contains column coordinates.
The resulting polygon goes
% through pixel corners, not
pixel centers.
if ~statsAlreadyComputed.ConvexHull
statsAlreadyComputed.ConvexHull
= 1;
if numel(imageSize) > 2
NoNDSupport('ConvexHull');
return
end
[stats, statsAlreadyComputed] =
...
ComputePerimeterCornerPixelList(ima
geSize,stats,statsAlreadyComputed);
[stats, statsAlreadyComputed] =
...
ComputeBoundingBox(imageSize,stats,
statsAlreadyComputed);
for k = 1:length(stats)
list =
stats(k).PerimeterCornerPixelList;
if (isempty(list))
stats(k).ConvexHull =
zeros(0,2);
else
rr = list(:,2);
cc = list(:,1);
hullIdx = convhull(rr,
cc);
stats(k).ConvexHull =
list(hullIdx,:);
end
end
end
function [stats,
statsAlreadyComputed] = ...
ComputePerimeterWithLabelMatrixOld(
L, stats, statsAlreadyComputed)
if
~statsAlreadyComputed.PerimeterOld
statsAlreadyComputed.PerimeterOld =
1;
if ndims(L) > 2 %#ok<ISMAT>
NoNDSupport('ComputePerimeter');
return
end
B =
regionboundariesmex(double(L),8);
for i = 1:length(B)
stats(i).PerimeterOld =
computePerimeterFromBoundaryOld(B{i
});
end
end
%%%
%%%
ComputePerimeterWithPixelIdxList
%%%
function [stats,
statsAlreadyComputed] = ...
ComputePerimeterWithPixelIdxListOld
(imageSize,stats,
statsAlreadyComputed)
if
~statsAlreadyComputed.PerimeterOld
statsAlreadyComputed.PerimeterOld =
1;
if numel(imageSize) > 2
NoNDSupport('ComputePerimeter');
return
end
[stats, statsAlreadyComputed] =
...
ComputeImage(imageSize,stats,statsA
lreadyComputed);
for k = 1:length(stats)
B =
regionboundariesmex(double(stats(k)
.Image),8);
stats(k).PerimeterOld =
computePerimeterFromBoundaryOld(B{1
});
end
end
%%%
%%% ComputePerimeterWithLabelMatrix
%%%
function [stats,
statsAlreadyComputed] = ...
ComputePerimeterWithLabelMatrix(L,
stats, statsAlreadyComputed)
if ~statsAlreadyComputed.Perimeter
statsAlreadyComputed.Perimeter
= 1;
if ndims(L) > 2 %#ok<ISMAT>
NoNDSupport('ComputePerimeter');
```

```
return
end
B =
regionboundariesmex(double(L),8);
for i = 1:length(B)
stats(i).Perimeter =
computePerimeterFromBoundary(B{i});
end
end
%%%
%%%
ComputePerimeterWithPixelIdxList
%%%
function [stats,
statsAlreadyComputed] = ...
ComputePerimeterWithPixelIdxList(im
ageSize,stats,
statsAlreadyComputed)
if ~statsAlreadyComputed.Perimeter
statsAlreadyComputed.Perimeter
= 1;
if numel(imageSize) > 2
NoNDSupport('ComputePerimeter');
return
end
[stats, statsAlreadyComputed] =
...
ComputeImage(imageSize,stats,statsA
lreadyComputed);
for k = 1:length(stats)
B =
regionboundariesmex(double(stats(k)
.Image),8);
stats(k).Perimeter =
computePerimeterFromBoundary(B{1});
end
end
%%%
%%% ComputePixelValues
%%%
function [stats,
statsAlreadyComputed] = ...
ComputePixelValues(I,stats,statsAlr
eadyComputed)
if
~statsAlreadyComputed.PixelValues
statsAlreadyComputed.PixelValues =
1;
for k = 1:length(stats)
```

```
stats(k).PixelValues =
I(stats(k).PixelIdxList);
end
end
%%%
%%% ComputeWeightedCentroid
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeWeightedCentroid(imageSize,I
,stats,statsAlreadyComputed)
if
~statsAlreadyComputed.WeightedCentr
oid
statsAlreadyComputed.WeightedCentro
id = 1;
[stats, statsAlreadyComputed] = ComputePixelList(imageSize,stats,st
atsAlreadyComputed);
for k = 1:length(stats)
Intensity =
I(stats(k).PixelIdxList);
sumIntensity =
sum(Intensity);
numDims =
size(stats(k).PixelList,2);
wc = zeros(1,numDims);
for n = 1 : numDims
M =
sum(stats(k).PixelList(:,n) .* ...
double(
Intensity(:) ));
wc(n) = M /
sumIntensity;
end
stats(k).WeightedCentroid = wc;
end
end
%%%
%%% ComputeMeanIntensity
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeMeanIntensity(I,stats,statsA
lreadyComputed)
if
~statsAlreadyComputed.MeanIntensity
statsAlreadyComputed.MeanIntensity
= 1;
[stats, statsAlreadyComputed] =
...
ComputePixelValues(I,stats,statsAlr
eadyComputed);
for k = 1:length(stats)
stats(k).MeanIntensity =
mean(stats(k).PixelValues);
end
end
%%%
%%% ComputeMinIntensity
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeMinIntensity(I,stats,statsAl
readyComputed)
if
~statsAlreadyComputed.MinIntensity
statsAlreadyComputed.MinIntensity =
1;
[stats, statsAlreadyComputed] =
...
ComputePixelValues(I,stats,statsAlr
eadyComputed);
for k = 1:length(stats)
stats(k).MinIntensity =
min(stats(k).PixelValues);
end
end
%%%
%%% ComputeMaxIntensity
%%%
function [stats,
statsAlreadyComputed] = ...
ComputeMaxIntensity(I,stats,statsAl
readyComputed)
if
~statsAlreadyComputed.MaxIntensity
statsAlreadyComputed.MaxIntensity =
1;
[stats, statsAlreadyComputed] =
...
ComputePixelValues(I,stats,statsAlr
eadyComputed);
for k = 1:length(stats)
stats(k).MaxIntensity =
max(stats(k).PixelValues);
end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function [I, reqStats,
officialStats] =
ParseInputs(sizeImage, argOffset,
varargin)
shapeStats = {
'Area'
'Centroid'
'BoundingBox'
'SubarrayIdx'
'MajorAxisLength'
'MinorAxisLength'
'Eccentricity'
'Orientation'
'ConvexHull'
'ConvexImage'
'ConvexArea'
'Image'
'FilledImage'
'FilledArea'
'EulerNumber'
'Extrema'
'EquivDiameter'
'Solidity'
'Extent'
'PixelIdxList'
'PixelList'
'Perimeter'
'PerimeterOld'};
pixelValueStats = {
'PixelValues'
'WeightedCentroid'
'MeanIntensity'
'MinIntensity'
'MaxIntensity'};
basicStats = {'Area'
'Centroid'
'BoundingBox'};
I = [];
officialStats = shapeStats;
numOrigInputArgs = numel(varargin);
if numOrigInputArgs == 1
%REGIONPROPS(BW) or
REGIONPROPS(CC) or REGIONPROPS(L)
reqStats = basicStats;
return;
elseif isnumeric(varargin{2}) ||
islogical(varargin{2})
```

```
%REGIONPROPS(...,I) or
REGIONPROPS(...,I,PROPERTIES)
I = varargin{2};
validateattributes(I,
{'numeric','logical'},{},
mfilename, 'I', 2+argOffset);
iptassert(isequal(sizeImage,size(I)
), ...
'images:regionprops:sizeMismatch')
officialStats =
[shapeStats;pixelValueStats];
if numOrigInputArgs == 2
%REGIONPROPS(BW) or
REGIONPROPS(CC,I) or
REGIONPROPS(L,I)
reqStats = basicStats;
return;
else
%REGIONPROPS(BW,I,PROPERTIES) of
REGIONPROPS(CC,I,PROPERTIES) or
%REGIONPROPS(L,I,PROPERTIES)
startIdxForProp = 3;
reqStats =
getPropsFromInput(startIdxForProp,
...
officialStats,
pixelValueStats, basicStats,
argOffset, varargin{:});
end
else
%REGIONPROPS(BW,PROPERTIES) or
REGIONPROPS(CC,PROPERTIES) or
%REGIONPROPS(L,PROPERTIES)
startIdxForProp = 2;
reqStats =
getPropsFromInput(startIdxForProp,
...
officialStats,
pixelValueStats, basicStats,
argOffset, varargin{:});
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function [reqStats,officialStats] =
getPropsFromInput(startIdx, ...
officialStats, pixelValueStats,
basicStats, argOffset, varargin)
if iscell(varargin{startIdx})
```

```
%REGIONPROPS(...,PROPERTIES)
propList = varargin{startIdx};
elseif strcmpi(varargin{startIdx}, 'all')
%REGIONPROPS(...,'all')
reqStats = officialStats;
return;
elseif strcmpi(varargin{startIdx}, 'basic')
%REGIONPROPS(...,'basic')
reqStats = basicStats;
return;
else
%REGIONPROPS(...,PROP1,PROP2,..)
propList = varargin(startIdx:end);
end
numProps = length(propList);
reqStats = cell(1, numProps);
for k = 1 : numProps
if ischar(propList{k})
noGrayscaleImageAsInput = startIdx == 2;
if noGrayscaleImageAsInput
% This code block exists so that regionprops can throw a more
% meaningful error message if the user want a pixel value based
% measurement but only specifies a label matrix as an input.
tempStats = [officialStats; pixelValueStats];
prop = validatestring(propList{k}, tempStats, mfilename, ...'PROPERTIES',
k+argOffset);
if
any(strcmp(prop,pixelValueStats))
error(message('images:regionprops:needsGrayscaleImage', prop));
end
else
prop = validatestring(propList{k}, officialStats, mfilename, ... 'PROPERTIES',
k+argOffset);
end
reqStats{k} = prop;
else
 error(message('images:regionprops:invalidType'));
end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function [stats, statsAlreadyComputed] = InitializeStatStructures(...imageSize,
numObjs, requestedStats, officialStats)
if numel(imageSize) > 2
% Remove stats that aren't supported for N-D input and issue
% warning messages as appropriate.
requestedStats = PreprocessRequestedStats(requestedStats);
end
```

```
if isempty(requestedStats)
error(message('images:regionprops:noPropertiesWereSelected'));
end
% Initialize the stats structure array.
tempStats = {'PerimeterCornerPixelList'};
allStats = [officialStats; tempStats];
numStats = length(allStats);
empties = cell(numStats, numObjs);
stats = cell2struct(empties, allStats, 1);
% Initialize the statsAlreadyComputed structure array. Need to avoid
% multiple calculatations of the same property for performance reasons.
zz = cell(numStats, 1);
for k = 1:numStats
zz{k} = 0;
end
statsAlreadyComputed = cell2struct(zz, allStats, 1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function NoNDSupport(str)
% Issue a warning message about lack of N-D support for a given
% measurement or measurements.
if iscell(str)
warning(message('images:regionprops:measurementNotForNDCell', sprintf( '%s ',
str{ : } )))
else
warning(message('images:regionprops:measurementNotForND', str))
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function requestedStats = PreprocessRequestedStats(requestedStats)
% Remove any requested stats that are not supported for N-D input
% and issue an appropriate warning.
no_nd_measurements =
{'MajorAxisLength'
'MinorAxisLength'
'Eccentricity'
'Orientation'
'ConvexHull'
'ConvexImage'
'ConvexArea'
'EulerNumber'
'Extrema'
'EquivDiameter'
'Solidity'
'Extent'
'Perimeter'
'PerimeterOld'};
```

```
bad_stats = find(ismember(requestedStats, no_nd_measurements));
if ~isempty(bad_stats)
NoNDSupport(requestedStats(bad_stats));
end
requestedStats(bad_stats) = [];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function perimeter = computePerimeterFromBoundaryOld(B)
delta = diff(B).^2;
perimeter = sum(sqrt(sum(delta,2)));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function perimeter = computePerimeterFromBoundary(B)
delta = diff(B).^2;
if(size(delta,1) > 1)
isCorner = any(diff([delta;delta(1,:)]),2); %
Count corners.
isEven = any(~delta,2);
perimeter = sum(isEven)*0.980 + sum(~isEven)*1.406 -sum(isCorner)*0.091;
else
perimeter = 0; % if the number of pixels is 1 or less.
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
function outstats = createOutputStatsStructure(requestedStats, stats)
% This is an optimized version of what happens in rmfield. In our case, we
% know the fieldnames in advance and what their indices are. Rmfield is
% much slower than this function because it calls strmatch(...n'exact').
% Figure out what fields to keep and what fields to remove.
fnames = fieldnames(stats);
idxRemove = ~ismember(fnames, requestedStats);
idxKeep = ~idxRemove;
% Convert to cell array
c = struct2cell(stats);
sizeOfc = size(c);
% Determine size of new cell array that will contain only the requested
% fields.
newSizeOfc = sizeOfc;
newSizeOfc(1) = sizeOfc(1) -numel(find(idxRemove));
newFnames = fnames(idxKeep);
% Create the output structure.
outstats = cell2struct(reshape(c(idxKeep,:), newSizeOfc), newFnames);
```