0.1 Boundary descriptors — boundarydescr

We will now show how to calculates the Fourier boundary descriptors w_i (Section 8.2.3 in MainBook). These descriptors are invariant to rotation, translation, and scaling. Their purpose is similar to the region descriptors regiondescr (Section **??**), the main difference being that the input region is characterized by its boundary, not all its pixels.

function w=boundarydescr(xy,n)

Inputs:

xy the object boundary represented as an $M \times 2$ array of point coordinates, each row corresponding to one point. The boundary is expected to be cyclic (the first row equals to the last row).

n *(optional)* number of the descriptors to return. It defaults to 7, values up to 15 are reasonable.

Outputs:

w a row vector of length n containing descriptors $w_2, \ldots, w_{n+1}$

Note that the descriptors w_i are related to Fourier coefficients therefore and decrease (decay) quickly with i, especially for smooth curves. It is therefore advisable to scale them appropriately.

We resample the boundary equidistantly as in resample (Section **??**). The first step is to calculate the distance between neighboring points and then the cumulative arc-length distance from the first point. We obtain the resampling xi, yi using interp1 (*Matlab*). The number N=256 of samples should be a power of two for efficient FFT calculation.

```
x=xy(:,1) ; y=xy(:,2) ;
dx = x(2:end)- x(1:end-1);
dy = y(2:end)- y(1:end-1);
d = sqrt(dx.*dx+dy.*dy);
d = [0;d]; % point 1 to point 1 distance is 0
d=cumsum(d) ; % the arc length distances from point 1
maxd = d(end);

N=256 ;
step=maxd/N ;
si = (0:step:maxd-step)';
xi = interp1(d,x,si);
yi = interp1(d,y,si);
```

Both x and y coordinates are Fourier transformed. Taking the absolute value of the complex coefficients xf, yf brings us invariance with respect to the starting point. Combining them to r (Equation 8.12 in MainBook) using Euclidean distance adds invariance to rotation. Neglecting the first (DC) element r(1) makes the result invariant to shift and normalizing by r(2) completes the effort by yielding w invariant by scaling.

```
xf=fft(xi) ; yf=fft(yi) ;

r=sqrt(abs(xf).^2+abs(yf).^2) ;
```

```
w=r(3:n+2)'/r(2) ;
```

Example

We use the same input images as in Section **??** which we store to structure `imgs`.

```
imgs(1).img=im2double(rgb2gray(imread([ ImageDir 'objectsA1.jpg' ]))) ;
imgs(2).img=im2double(rgb2gray(imread([ ImageDir 'objectsA2.jpg' ]))) ;
imgs(3).img=im2double(rgb2gray(imread([ ImageDir 'objectsA3.jpg' ]))) ;
```

One way of obtaining the boundaries needed for the boundary descriptors would be to apply the function `bwboundaries` (*Matlab*) on the `GraphCut` (Section **??**) segmentation obtained in Section **??**. Here we show an alternative method using snake segmentation `snake` (Section **??**) that finds the boundaries (contours) directly. The segmentation is encapsulated in function `snake_segmentation` (below) . For each object to be segmented we provide the parameters of the initial circular contour within the object and also the parameters κ and λ (see Section **??**) that sometimes need to be adjusted so that the snake stops at the desired boundary. The resulting boundaries are stored in the structure `b(i).o(j).xy`, where `i` is the image number and `j` the object number.

```
b(1).o(1).xy=snake_segmentation(imgs(1).img,80,120,10,0.2,0.05) ;
b(1).o(2).xy=snake_segmentation(imgs(1).img,123,80,1,0.4,0.15) ;
b(1).o(3).xy=snake_segmentation(imgs(1).img,180,100,10,0.2,0.05) ;

b(2).o(1).xy=snake_segmentation(imgs(2).img,110,100,10,0.2,0.05) ;
b(2).o(2).xy=snake_segmentation(imgs(2).img,40,90,10,0.2,0.05) ;
b(2).o(3).xy=snake_segmentation(imgs(2).img,137,54,1,0.4,0.15) ;

b(3).o(1).xy=snake_segmentation(imgs(3).img,150,120,10,0.3,0.05) ;
b(3).o(2).xy=snake_segmentation(imgs(3).img,90,50,10,0.2,0.05) ;
b(3).o(3).xy=snake_segmentation(imgs(3).img,166,66,1,0.65,0.2) ;
```

To save time, the boundarues can be saved using `save boundaries b` and later restored using `load boundaries`, as usual. The top row in Figure **??** shows the boundaries found.

The boundary descriptors are found for all objects in all images and stored into matrix `b(i).phi`, where each row corresponds to one object of image `i`. The descriptors w_i are then normalized by the median value of w_i over all images to compensate for their uneven amplitude. This is improves the classification performance even though it is not strictly necessary for our simple case. Other normalizations (e.g. by variance or maximum) are also possible and likely to work well.

```
phis=[] ;
for i=1:3,
  for j=1:length(b(i).o),
    xy=b(i).o(j).xy ; xy=[xy ; xy(1,:) ] ; % close the contour
    phi=boundarydescr(xy) ;
    b(i).phi(j,:)=phi ; phis= [phis ; phi] ;
  end ;
end ;
```

```
mphi=median(phis) ;

for i=1:3,
    b(i).phi=b(i).phi ./ repmat(mphi,length(b(i).o),1) ;
end ;
```

A casual glance shows that the normalized descriptors characterize the objects well:

```
b(1).phi =

   0.1804   0.8544   0.7838   1.0000   0.8889   1.0944   1.0144
   1.0000   1.0000   0.9761   1.2051   0.2306   1.1735   0.7810
   3.6079   1.1094   3.0756   0.8451   1.6603   0.5796   1.9651

b(2).phi =

   3.5384   1.1063   3.1145   0.8349   1.6322   0.5670   1.9851
   0.2829   0.8659   0.9329   0.9887   1.0000   1.0000   0.7680
   1.2273   1.0157   1.0000   1.1852   0.4457   1.2829   1.0000

b(3).phi =


   3.7690   1.1043   3.0521   0.8078   1.6791   0.5636   1.9840
   0.2695   0.8477   0.8095   1.0160   1.0788   0.9680   0.9596
   0.9206   0.9836   1.0265   1.1791   0.4028   1.2292   0.9844
```

The object matching is performed by a nearest neighbor classifier as in Section **??**. The function nnmatch (Section **??**) calculates for each object in images 2 and 3 the index of the "most similar" object in image 1, where similarity is measured as the Euclidean distance of the descriptor vectors.

```
b(1).ind=1:length(b(1).o) ;
b(2).ind=nnmatch(b(1).phi,b(2).phi);
b(3).ind=nnmatch(b(1).phi,b(3).phi);
```

The final matching can be displayed as follows:

```
for i=1:3,
  imagesc(imgs(i).img) ; colormap(gray) ;
  axis image ; axis off ; hold on ;
  colors='rgbcmyk' ;
  for j=1:length(b(i).o),
    plot(b(i).o(j).xy(:,1),b(i).o(j).xy(:,2),...
      [ colors(b(i).ind(j)) '-' ],'LineWidth',2) ;
  end ;
  hold off
end ;
```

We see that the objects are identified correctly (Figure 1, bottom row), the same contour color belongs to the same object in all three images.

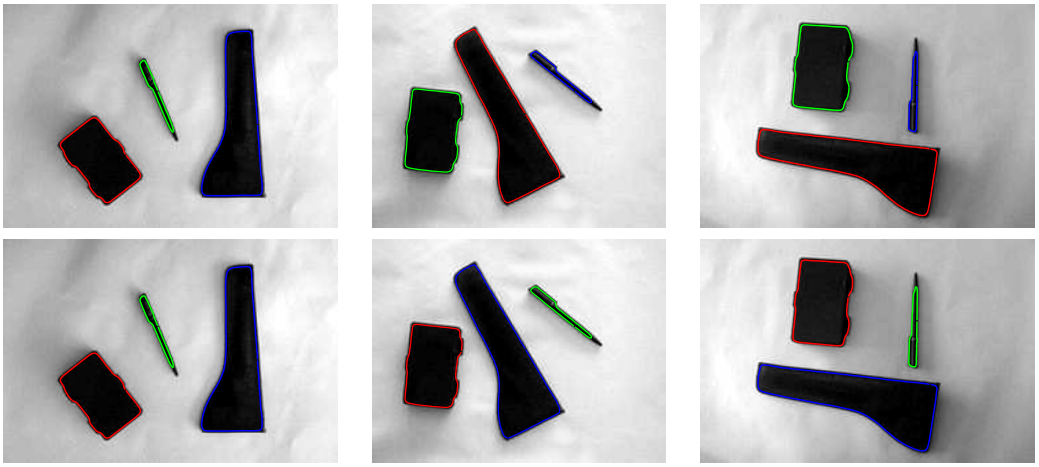

Figure 1: Input images with snake segmentation and initial class labels (top row) and the final classification with class labels (contour colors) determined using boundary descriptors (bottom row).

---

function `xy=snake_segmentation(img,xc,yc,r,kappa,lambda)`

Inputs:

`img`	contains the grayscale image to be segmented
`xc,yc`	x and y coordinates of the center of the initial circular contour
`r`	radius of the initial circular contour
`kappa`	parameter determining the strength of the external force (data term). See also Section **??**.
`lambda`	parameter determining the strength of the balloon force. See also Section **??**.

Outputs:

`xy`	the final contour points as returned by `snake` (Section **??**), packed to an $M \times 2$ array suitable for `boundarydescr` (Section 0.1)

The initial contour is a circle which is expected to lie inside the object to be segmented. Care needs to be taken so that the contour is oriented anticlockwise.

```
t=0:0.5:2*pi ;
xi=xc+cos(t)*r ; yi=yc+sin(t)*r ;
```

The snake will be driven purely by intensity, expanding in dark regions and shrinking in bright regions.

```
[px,py] = gradient(-img);
kappa1=1/max(abs( [px(:) ; py(:)])) ;
[x,y]=snake(xi,yi,0.1,0.01,kappa*kappa1,lambda,px,py);
```

The evolution of the snake can be observed by using the following line instead.

```
[x,y]=snake(xi,yi,0.1,0.01,kappa*kappa1,lambda,px,py,0.4,1,img);
```

Finally, we pack the `x` and `y` arrays together.

```
xy=[x y] ;
```