0.1 Active contours (snakes) — snake

Active contours, informally called *snakes*, are a powerful semi-automatic image segmentation tool. In anticipation of the GVF snakes (Section **??**), we implement here a traditional balloon-force snake (Section 7.2 in MainBook) using a PDE approach (Equation 7.29 in MainBook) with an evolution equation

$$\frac{\partial \mathbf{v}}{\partial t} = \alpha \frac{\partial^2 \mathbf{v}}{\partial s^2} - \beta \frac{\partial^4 \mathbf{v}}{\partial s^4} + \kappa \mathbf{f}_{\mathrm{E}} + \lambda \mathbf{f}_{\mathrm{B}}$$

where $\mathbf{v}(s,t)$ is the snake curve as a function of the arc length s and time t. The constants α and β control the elasticity (stretching) and stiffness (bending) of the snake, and $\mathbf{f}_{\mathrm{E}}$ is the external (image dependent force). The balloon force $\mathbf{f}_{\mathrm{B}}$ is perpendicular to the snake curve and is used to make the snake grow or shrink. We are interested in a steady-state solution $\mathbf{v}(s)$ for $t \to \infty$ which we shall find by Euler integration. This is appropriate, since only low-accuracy is needed. To simplify tuning the parameters α, β, κ, λ, we take unit time steps ($\Delta t = 1$) and assume that the external force $\mathbf{f}_{\mathrm{E}}$ is normalized to have maximum magnitude equal to 1 (pixel/step). The balloon force $\mathbf{f}_{\mathrm{B}}$ has magnitude one everywhere.

The snake $\mathbf{v}(s)$ at time t is represented by two vectors containing the x and y coordinates of a sequence of points on the snake curve. The distance between subsequent points is maintained close to 1 pixel. The snake is supposed to be closed and simple (non-intersecting). If a snake starts to intersect itself, the user is expected to restart the evolution with smaller forces (reduced parameter values).

The code presented here is derived from the work of Xu and Prince [**?**] and their publicly available implementation (http://iacl.ece.jhu.edu/projects/gvf/).

function `[x,y] = snake(x,y,alpha,beta,kappa,lambda,fx,fy,maxstep)`

Inputs:

`x, y` vectors containing the x and y coordinates of the initial snake position. The snake is closed — last point is considered a neighbor of the first one. The points can be far apart and will be interpolated as needed. The contour should be given anticlockwise, otherwise the balloon force will change sign.

`alpha` α controls the elasticity; the higher the value, the less stretching will the snake allow. $\alpha \approx 0.1$ is a good value.

`beta` β controls the rigidity, the resistance to bending. Make it small for the snake to follow a jagged boundary. $\beta = 0 \sim 0.1$ work well.

`kappa` κ weights the contribution of the external force $\mathbf{f}_{\mathrm{E}}$ that usually stops the snake once it arrives at the desired boundary. Make κ bigger for the snake to stop sooner. $\kappa = 0.1 \sim 0.5$ give good results.

`lambda` λ weights the contribution of the balloon force. Positive value makes the snake to expand.

`fx, fy` external force field, normalized not to extend 1 in magnitude. Both `fx` and `fy` are scalar matrices corresponding to the original image.

`maxstep` *(optional)* maximum step-size. Defaults to 0.4 pixels. Reducing it in difficult cases might help to prevent the snake from crossing over.

Outputs:

`x,y` the x and y coordinates of the final position of the snake. The distance between subsequent points is close to 1 pixel.

The snake is represented by its samples $\mathbf{v}(s_i) = [x(s_i)\ y(s_i)]^T$ with a sampling step $\Delta s = 1$, $\|\Delta \mathbf{x}| \approx 1$. The derivatives can be approximated by a discrete convolution:

$$\alpha \frac{\partial^2 \mathbf{v}}{\partial s^2} - \beta \frac{\partial^4 \mathbf{v}}{\partial s^4}\bigg|_{s=s_i} \approx h * \begin{bmatrix} x(s_i) \\ y(s_i) \end{bmatrix}$$

Array `areas` is used in a stopping criterion described later.

```
x=x(:) ; y=y(:) ; % convert to column vectors
h=[ -beta 4*beta+alpha -6*beta-2*alpha 4*beta+alpha -beta ] ;
areas=[] ;
```

We iterate in a `while`-loop until convergence. We resample the snake if needed (Section 0.1) to maintain uniform distances between points. We use interpolation to evaluate the external force $\mathbf{f}_\mathrm{E}$ at the snake points. If the snake leaves the image area, we stop.

```
while true,
  [x,y]=resample(x,y,0.9) ;
  vfx = interp2(fx,x,y);
  vfy = interp2(fy,x,y);
  if any(isnan([vfx(:) ; vfy(:)])), break ; end ;
```

The evaluation of the balloon force $\mathbf{f}_\mathrm{B}$ starts by calculating the tangential vector $\mathbf{q} = \frac{\partial \mathbf{v}}{\partial s}$. The derivative is approximated by finite differences while observing the periodic boundary conditions. We then create unit a vector $\mathbf{p}$, perpendicular to $\mathbf{q}$.

```
  xp = [x(2:end) ; x(1)]; yp = [y(2:end) ; y(1)];
  xm = [x(end) ; x(1:end-1)]; ym = [y(end) ; y(1:end-1)];
  qx = xp-xm; qy = yp-ym;
  pmag = sqrt(qx.*qx+qy.*qy);
  px = qy./pmag;   py = -qx./pmag;
```

Testing for convergence is delicate. Because of the fixed time step used and discretization artifacts, the snake frequently starts to oscillate around the equilibrium position with small amplitude. The oscillations could be avoided be a more elaborate integration scheme, but this would increase the computational cost. An alternative approach is to analyze the snake movement in last few iterations and stop if no progress is being made. As curve distance computation is computationally expensive, we use a trick based on the fact that the snake is always supposed to either grow or shrink. We evaluate the area inside the snake in 10 last iterations and if the change is smaller than one pixel, we stop. Note that the area is quite simple to compute.

```
  area=sum(0.5 * (xp+xm) .* qy) ;
  areas=[areas area] ;
  if length(areas)>10,
    areas=areas(end-9:end) ; % keep last 10 areas
    if max(areas)-min(areas)<1,
        break ;
    end ;
  end ;
```

We proceed to calculate the total force (xd, yd) acting on the snake. Note how the elasticity and stiffness contribution is calculated using a convolution. The function convs, defined below, respects the periodic boundary conditions.

Updating the snake coordinates finishes the loop. As a safeguard, we shorten the step-size (step) if the default size 1 would cause some points to move further than maxstep pixels (0.4 by default). For recommended parameter values, this restriction is normally not applied, only when the snake initialization is noisy or has sharp edges.

```
  xd=convs(x,h)+ kappa*vfx + lambda.*px ;
  yd=convs(y,h)+ kappa*vfy + lambda.*py ;

  maxd=max([xd ; yd]) ;
  step=min(1,maxstep/maxd) ;

  x = x+ step*xd ;
  y = y+ step*yd ;

end ; % while-loop
```

---

function xc=convs(x,h)

convs calculates the convolution of x with h, treating x as periodic. The kernel h is expected to have an odd length and is considered to be centerd.

```
  N=length(h) ; N2=(N-1)/2 ;
  xc=conv([ x(end-N2+1:end) ; x ; x(1:N2) ],h) ;
  xc=xc(N:end-N+1) ;
```

Snake resampling

function [xi,yi]=resample(x,y,step)

resample takes a snake and resamples it if needed so that the distance between points is equal to step, returning the new representation. Note that some implementations do not resample the snake on each iteration, but only each 5 or 10 iterations. This leads to computational savings at the expense of some robustness. Note also that changing the interpoint spacing changes the value of our discrete approximation of the elasticity and stiffness.

The point vectors are made circular and distances between points are calculated. The arc length distances from point 1 to all other points are stored to d.

```
  N = length(x);
  xi=x ; yi=y ;
  x = [x;x(1)]; % make a circular list
  y = [y;y(1)];
  dx = x([2:end])- x(1:N);
```

```
dy = y([2:end])- y(1:N);
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
```

If the length is unchanged, we are done. Otherwise, new snake points are calculated by interpolating from the old ones.

```
maxd = d(end);
if (abs(maxd/step-N)<1),
  return ;
end ;
si = (0:step:maxd)';
xi = interp1(d,x,si);
yi = interp1(d,y,si);
```

Finally, we discard the last point of the list if it is too close to its neighbor, the first point of the list.

```
if (maxd - si(end) <step/2),
  xi = xi(1:end-1);
  yi = yi(1:end-1);
end
```

Examples

First example shows how to use snakes to find the inner boundary of the heart cavity in a magnetic resonance image (Figure 1, left). The initial position of the snake is a small circle located inside the cavity. We will make the snake to expand until it reaches the bright wall.

```
img=imread([ImageDir 'heart.pgm']) ;
t=(0:0.5:(2*pi))' ;
x = 70 + 3*cos(t);
y = 90 + 3*sin(t);
```

To show the initial position, you can use the following code:

```
imagesc(img) ; colormap(gray) ; axis image ; axis off ; hold on ;
plot([x;x(1,1)],[y;y(1,1)],'r') ; hold off ;
```

The external energy is a smoothed version of the image, normalized for convenience (Figure 1, center).

```
h=fspecial('gaussian',20,3) ;
f=imfilter(double(img),h,'symmetric') ;
f=f-min(f(:)) ; f=f/max(f(:)) ;
```

The external force is a negative gradient of the energy. We start the snake evolution with $\alpha = 0.1$, $\beta = 0.01$, $\kappa = 0.2$, $\lambda = 0.05$. Note that the normalization constant is incorporated into κ.

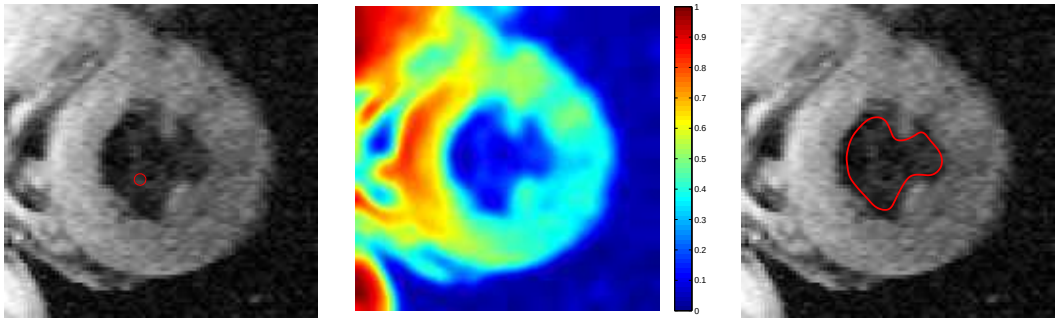

Figure 1: Input image (MRI cross-section of the heart, left) with the initial snake position, the corresponding external energy (middle), and the final boundary found (right).

```
[px,py] = gradient(-f);
kappa=1/max(abs( [px(:) ; py(:)])) ;
[x,y]=snake(x,y,0.1,0.01,0.2*kappa,0.05,px,py);
```

The final position of the snake is shown in Figure 1, right. We can see that the boundary is well recovered. It is instructive to run the snake evolution for different values of the parameters and note how the evolution speed and the final shape changes. Start with small changes first; big changes make the snake behave in unpredictable ways.

The second example deals with segmenting an object (a bird) in a color image (Figure 2, left). This time we set the initial snake position manually around the object using a function `snakeinit` (given below) and let the snake shrink until it hits the object.

```
img=imread([ ImageDir 'bird.png']) ;
[x,y]=snakeinit() ;
```

For convenience, the initial snake position can be saved and reloaded later as follows:

```
save birdxy x y
load birdxy
```

To calculate the external energy (Figure 2, center), the image is first converted into grayscale using a particular linear combination of color channels that emphasizes the difference between the foreground and the background. The result is normalized and small values are suppressed using thresholding. Finally, the energy image is smoothed.

```
f=double(img) ; f=f(:,:,1)*0.5+f(:,:,2)*0.5-f(:,:,3) ;
f=f-min(f(:)) ; f=f/max(f(:)) ;
f=(f>0.25).*f ;
h=fspecial('gaussian',20,3) ;
f=imfilter(double(f),h,'symmetric') ;
```

We calculate the external force from the energy and start the minimization with parameters $\alpha = 0.1$, $\beta = 0.1$, $\kappa = 0.3$. Note the negative value of the balloon force coefficient $\lambda = -0.05$ that makes the snake shrink instead of expand (this depends on the clockwise orientation of the snake points).

```
[px,py] = gradient(-f);
```

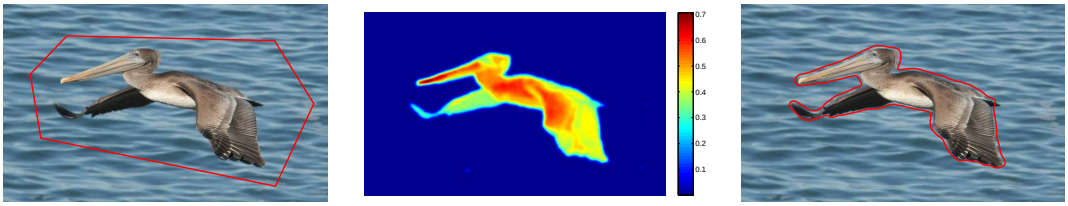

Figure 2: Input image with the initial snake position (left), the corresponding external energy (middle), and the final boundary found (right).

```
kappa=1/(max(max(px(:)),max(py(:)))) ;
[x,y]=snake(x,y,0.1,0.1,0.3*kappa,-0.05,px,py);
```

The final result is shown in Figure 2, right. Observe that the bird is well delineated, although the snake stops a few pixels away from the boundary. This behavior is fairly typical for the simple external energy used. It can be partly eliminated by using less smoothing at the expense of robustness.

---

function `[x,y] = snakeinit()`

`snakeinit` provides a simple interface to initialize the snake position. Display the image you want to segment, e.g. by `imagesc(img)`, call `snakeinit` and use the left mouse button to choose the snake points in a clockwise direction. Right mouse button picks the last point.

```
hold on ;
x = []; y = [];
n =0; but = 1;
while but == 1
      [s, t, but] = ginput(1);
      n = n + 1;
      x(n,1) = s;
      y(n,1) = t;
      plot(x, y, 'r-');
end

plot([x;x(1,1)],[y;y(1,1)],'r-');
hold off
```

0.2 Gradient vector flow snakes — mgvf

Gradient vector flow (Section 7.3.2 in MainBook) is a way to improve a snake segmentation (Section 0.1). The balloon force is no longer needed, as the external force is modified to "know" the direction to the boundary even far away from it. The function `mgvf` — where `m` stands for multiresolution — calculates an external force $\mathbf{f}_E$ from the external energy by solving the PDEs 7.25, 7.26 in MainBook. Multiresolution approach consists of solving the problem first for a reduced version of the image and using the solution as a starting point for the next finer level. It is needed because for large images, the convergence of the force field far away from the edges is quite slow, which makes the snake to stop before reaching the desired boundary.

The code presented here is derived from the work of Xu and Prince [**?**] and their publicly available implementation (http://iacl.ece.jhu.edu/projects/gvf/).

function `[fx,fy]=mgvf(E,mu,tol)`

Inputs:

`E`	external energy. It is normalized to $[0, 1]$.
`mu`	*(optional)* regularization parameter μ. Default value $\mu = 0.2$ works well.
`tol`	*(optional)* absolute stopping tolerance for the ℓ_∞ change of $\mathbf{f}_E$ between two iterations. Defaults to 10^{-3}.

Outputs:

`fx,fy`	the calculated external force

If the energy image is small enough, the solution is found iteratively using function `gvf` (described below). If not, it is scaled down to half the size and `mgvf` is called recursively to find the force field. The force field found is extrapolated to the original resolution and used as a starting guess for an iterative solution in `gvf`.

```
if min(m,n)<64,
  [fx,fy]=gvf(E,mu,tol) ;
else
  Es=imresize(E,0.5,'bilinear')*2 ;
  [fxs,fys]=mgvf(Es,mu,tol) ;
  fx0=imresize(fxs,[m n],'bilinear')*0.5 ;
  fy0=imresize(fys,[m n],'bilinear')*0.5 ;
  [fx,fy]=gvf(E,mu,tol,fx0,fy0) ;
end ;
```

---

function `[fx,fy] = gvf(E, mu, tol,fx0,fy0)`

Inputs:

`E`	external energy
`mu`	*(optional)* the regularization parameter μ. Default value $\mu = 0.2$ works well.
`tol`	the absolute stopping tolerance for the ℓ_∞ change of $\mathbf{f}_E$ between two iterations. Defaults to 10^{-3}.
`fx0,fy0`	*(optional)* initial guess of the force field

Outputs:

`fx,fy`	the calculated external force

Calculate the gradient and use it as an initial solution if no initial guess is provided.

```
[m,n] = size(E);
[gx,gy] = gradient(E);
if nargin<5,
  fx = gx; fy = gy;
else
  fx=fx0 ; fy=fy0 ;
end ;

SqrMagf = gx.*gx + gy.*gy;
```

The solution is found iteratively (Equation 7.27–7.28 in MainBook) for timestep 1. Note that if bigger μ is used, smaller timestep might be needed [**?**]. Convergence is detected by monitoring the maximum amplitude `ampl` of the change `fxd`, `fyd`. We also stop if `ampl` fails to decrease is this signals numerical inaccuracy and further iterations would not be beneficial. As a final safeguard, we stop after 1000 iterations.

```
ampl0=inf ; i=0 ;

while true,

  fxd=mu*4*del2(fx) - SqrMagf.*(fx-gx);
  fyd=mu*4*del2(fy) - SqrMagf.*(fy-gy);

  ampl=max(abs([fxd(:) ; fyd(:)])) ;
  if ampl>ampl0,
    disp(['Numerical␣instability␣detected,␣amplitude␣' num2str(ampl)]) ;
    break ;
  end ;
  ampl0=ampl ;
  fx = fx + fxd ;
  fy = fy + fyd ;
  if ampl<tol || i>1000,
    break ;
  end ;
i=i+1 ;
end ; % while-loop
```

Example

The example shows how to use a GVF snake for segmenting a lung from a CT (computed tomography) image (Figure 3, left).

```
img=dicomread([ImageDir 'ctslice.dcm']) ;
```

An initial position is found manually, using `snakeinit` (Section 0.1).

```
[x,y]=snakeinit() ;
```

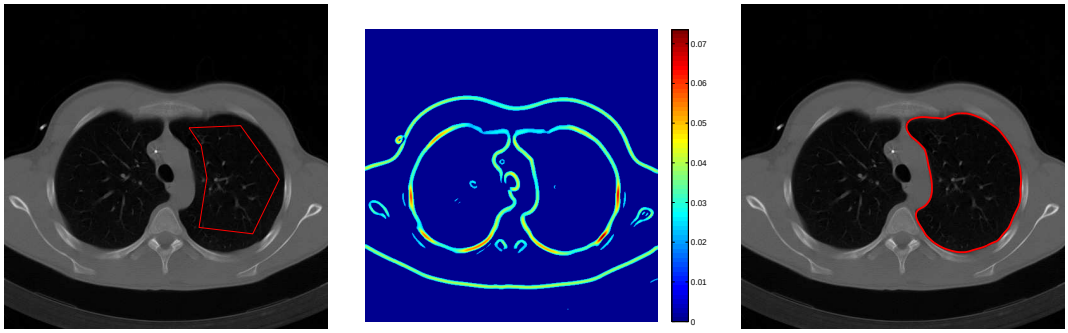

Figure 3: Input image (CT slice) with the initial snake position (left), the corresponding external energy (middle), and the final snake delineating the lung boundary found (right).

For GVF snakes, the external energy is supposed to be an *edge map*, with high values at the locations where we want the snake to be attracted to. We create it by taking a magnitude of the gradient of the smoothed and normalized image, $E = \|\Delta G_\sigma * f\|$. Small values are suppressed by thresholding (Figure 3, center).

```
f=double(img) ; f=f-min(f(:)) ;f=f/max(f(:)) ;
h=fspecial('gaussian',20,3) ;
f=imfilter(f,h,'symmetric') ;
[px,py]=gradient(f) ;
E=sqrt(px.^2+py.^2) ;
E=E.*(E>0.02) ;
```

Applying `mgvf` calculates the force field `ux`, `uy` which is fed into the snake evolution function `snake` (Section 0.1). Note that $\lambda = 0$ as no balloon force is needed. The final result is shown in Figure 2, right.

```
[fx,fy]= mgvf(E) ;
kappa=1/(max(max(fx(:)),max(fy(:)))) ;
[x,y]=snake(x,y,0.1,0.01,0.3*kappa,0,fx,fy,0.4,1,img);
```