Segmentation Workshop
Level Sets (Python)

Tutor: M. Jorge Cardoso

In this workshop, we are going to implement the paper "**Level Set Evolution Without Reinitialization: A New Variational Formulation**" Chunming Li (CVPR'05). If possible, you should read the paper first and try to get an idea of what we are going to do. This should not take more than 30 min.

Here, specifically, we are going to segment the two images available in the following URL http://goo.gl/Ku1U7W

The main equation of the paper is equation (10) and (13). In short, one can describe the level set update function by equation (13)

$$\phi_{i,j}^{k+1} = \phi_{i,j}^{k} + \tau L(\phi_{i,j}^{k})$$

Where *L(ϕ)* is the finite difference approximation of the right hand side of equation (10):

$$\frac{\partial \phi}{\partial t} = \mu[\triangle\ \phi - \mathrm{div}(\frac{\nabla \phi}{|\nabla \phi|})] + \lambda \delta(\phi)\mathrm{div}(g\frac{\nabla \phi}{|\nabla \phi|}) + \nu g \delta(\phi)$$

To do so, lets first load a sample image to python and create the initial level set. To load the image, one can use the command

```
import scipy.ndimage as ndi
imgData = ndi.imread("Obj.bmp")
```

To create the initial level set, generate an image of the same size as the previously loaded image with a rectangle bounding the object. The level set should have a positive value (e.g. equal to 4) inside the rectangle and a negative value (e.g. equal to -4) outside the rectangle. You can use the tool

```
LevelSet = ndi.imread("Start.bmp", np.float64)
LevelSet = ((LevelSet>0)*8)-4
```

to generate the square and than transform that BW image into a level set (`LevelSet`).
It will be necessary to have the gradient of the image throughout the paper. Thus, you should calculate it using the gradient function inside python. As the image in noisy, you should previously convolve the image with a gaussian filter using the functions

```
imgSmooth = ndi.gaussian_filter(imgData, sigma=(2,2), order=0, mode='reflect')
```

Where sigma is a parameter. In this case, make sigma equal to 1.5.
You can then calculate its gradient with

```
imgX = ndi.gaussian_filter(imgSmooth, sigma=(1,0), order=1, mode='reflect')
imgY = ndi.gaussian_filter(imgSmooth, sigma=(0,1), order=1, mode='reflect')
```

Form these gradients, you should now calculate the edge indicator function g (section 3.2 of the paper) and also its derivative. Here,

```
g=1/(1+( imgX**2+ imgY**2) )
```

and

```
Vx = ndi.gaussian_filter(g, sigma=(1,0), order=1, mode='reflect')
Vy = ndi.gaussian_filter(g, sigma=(0,1), order=1, mode='reflect')
```

where Vx and Vy are the gradients of the edge function g.
You should also set all your parameters. These will be changed depending on the image. For now, set them as:

```
epsilon=1.5
sigmaIn=0.8
ts=3.0
Internal_weight=0.05/ts
Length_weight=10.0
Area_weight=-20.0
numiter=500
```

The Area_weight should be negative if the initial level set is outside, and positive if inside. At this point, you should have your initial level set, the gradient of the gaussian filtered image, the edge indicator function, and the gradient of the edge indicator function.

Level Set Evolution

=> Penalising term

Inside a for/while loop, lets calculate each of the components of equation 13 independently. First, calculate the penalising term $\mu[\triangle\ \phi - \mathrm{div}(\frac{\nabla\phi}{|\nabla\phi|})]$

The first part inside the brackets is the laplacian of the current level set. This can be calculated using the

```
Laplacian=ndi.filters.laplace(LevelSet, mode='reflect', cval=0.0)
```

function in Python. The second part can be calculated by first calculating the gradient of the current level set

```
LevelSetGradX = ndi.gaussian_filter(LevelSet, sigma=(1,0), order=1, mode='reflect')
LevelSetGradY = ndi.gaussian_filter(LevelSet, sigma=(0,1), order=1, mode='reflect')
```

and normalising by it's norm (note: the 1e-10 is added in order to avoid a division by zero)

```
normDu = np.sqrt( LevelSetGradX**2 + LevelSetGradY**2 + 1e-10 );
```

resulting in two normalised directional gradients:

```
Nx=LevelSetGradX/normDu
Ny=LevelSetGradY/normDu
```

and then calculating the divergence of this normalised gradients by setting

```
Nxx = ndi.gaussian_filter(Nx, sigma=(1,0), order=1, mode='reflect')
Nyy = ndi.gaussian_filter(Ny, sigma=(0,1), order=1, mode='reflect')
Divergence = Nxx + Nyy;
```

To get the full first term (penalising term) you now only need to calculate

```
PenalisingTerm = Internal_weight * (Laplacian/16.0*13.0 - Divergence)
```

Remove very small penalty terms that are caused by

```
PenalisingTerm=PenalisingTerm * ( (PenalisingTerm>0.01) | (PenalisingTerm<-0.01) )
```

=> Weighted Length term

The second term is the weighted length term: $\lambda\delta(\phi)\mathrm{div}(g\frac{\nabla\phi}{|\nabla\phi|})$

The Dirac can be calculated by equation (11)

```
Dirac = ((1.0/2.0) / epsilon) * ( 1.0 + np.cos((np.pi*LevelSet) / epsilon)) * ( LevelSet
<= epsilon ) * ( LevelSet >= -epsilon )
```

where x is will be the level-set and sigma will be the predefined sigma parameter.
In order to calculate the divergence of the scalar field g and the previously calculated vector field [Nx,Ny], one can use the definition that states

$$\nabla.(gF) = \nabla(g).F + g.\nabla(F)$$

Thus, this divergence can be calculated by setting

```
Divergence2=(Vx*Nx + Vy*Ny + g*Divergence)
```

With Vx and Vy as the gradients of the scalar field g, Nx and Ny as the vector field inside the divergence, g as the scalar field and Divergence as the perviously calculated vector field divergence (see penalising term).
To get the full second term (Weighted Length term) you now only need to calculate

```
WeightedLengthTerm=Length_weight*Dirac*Divergence2
```

=> Weighted Area term
The Third term is the weighted area term: $\cdot \nu g \delta(\phi)$

To get the full third term (Weighted Area term) you only need to calculate

```
WeightedAreaTerm=Area_weight*g*Dirac
```

=> Put it all together

To merge all the terms and update the level-set, one just needs to calculate at each iteration

```
LevelSet = LevelSet + ts*(PenalisingTerm + WeightedLengthTerm + WeightedAreaTerm);
```

=> Neuman boundary conditions

Note that the boundary conditions when using this implementation do not satisfy the Neumann condition. Thus, at each iteration, one needs to apply the function:

```
LevelSet = NeumannBoundCond(LevelSet)
```

This function should enforce the following Neuman conditions

```
LevelSet[0,0] = LevelSet[2,2]
LevelSet[0,-1] = LevelSet[2,-3]
LevelSet[-1,0] = LevelSet[-3,2]
LevelSet[-1,-1] = LevelSet[-3,-3]
LevelSet[0, 1:-2] = LevelSet[2, 1:-2]
LevelSet[-1, 1:-2] = LevelSet[-3, 1:-2]
LevelSet[1:-2, 0] = LevelSet[1:-2, 2]
LevelSet[1:-2, -1] = LevelSet[1:-2, -3]
```

=> Stopping criteria

The stopping criteria can either be “no change” in the value of `LevelSet` (normally a change below a threshold) or just a certain number of iterations. This is up to you.