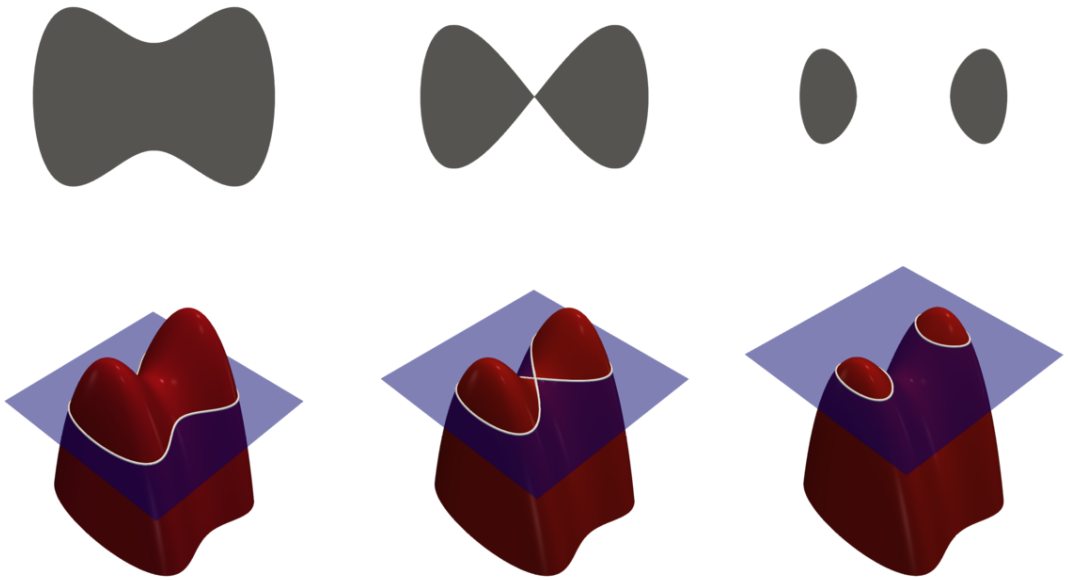
INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Image Processing

Active Contour Models



Rand ASSWAD
Applied Mathematics

Submitted to:
Mr. Christian GOUT

2 March 2020

Contents

1 Introduction

Recognising objects and identifying shapes in images is usually an easy task for human, it is however difficult to automate. The field of **computer vision** is concerned with automating such processes. It aims to extract information from images (or video sequences of images) in order to achieve what a human visual system can. ("Computer Vision" 2020)

Active contour models (also called **snakes**) is a class of algorithms for finding boundaries of shapes. These methods formulate the problem as an optimisation process while attempting to balance between matching to the image and ensuring the result is smooth. ("Snakes" 2011)

In the scope of this project, we will explore a few active contour models with the help of *The Numerical Tours of Signal Processing* (Peyré 2011).

1.1 Snakes method

A snake is a smooth curve, similar to a spline. The concept of a snakes method is to find smooth curves that match the image features by iteratively minimizing the *energy* function of the snakes. ("Active Contour Model" 2020)

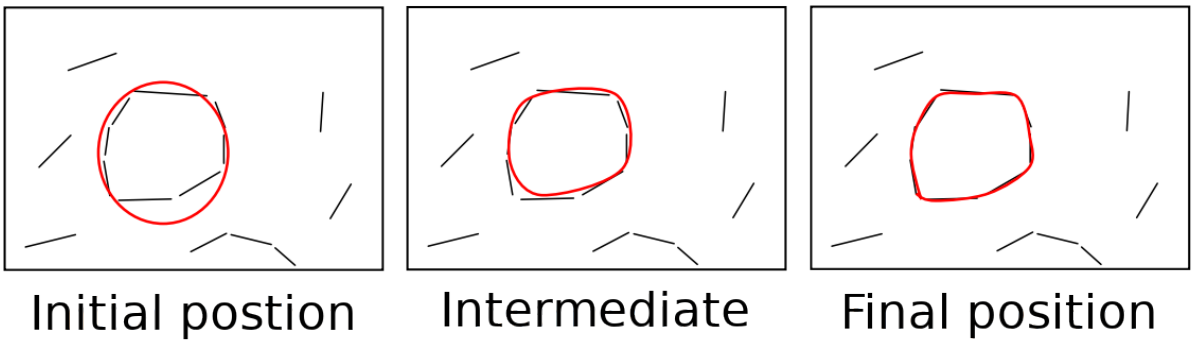


Figure 1: Illustration of the snakes model

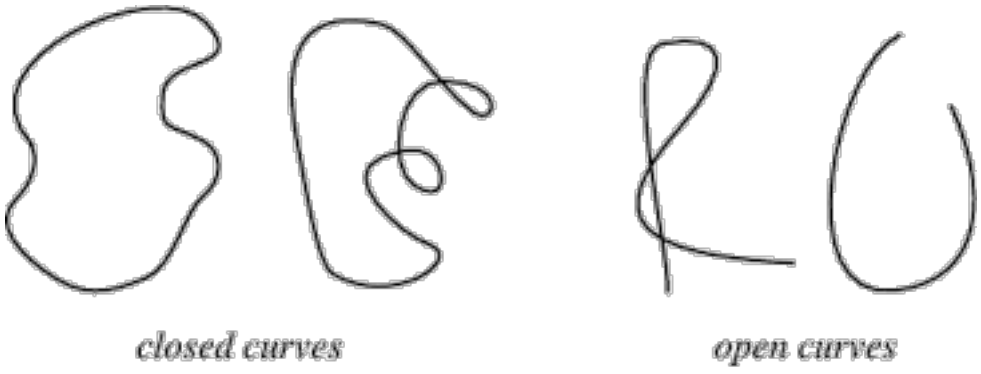
The energy of the snakes is a combination of internal and external energy ("Snakes" 2011)

- **Internal Energy:** a metric that measures the curve's smoothness or regularity.
- **External Energy:** a metric for measuring the data fidelity.

1.2 Classification and representation

Curves can be divided into two types: open curves and closed curves.

- An *open curve* has two distinct ends and does not form a loop.
- A *closed curve* is a curve with no endpoints and which completely encloses an area. (Weisstein n.d.)



A curve has two different representations: **parametric** or **cartesian**.

In the parametric form, the points of the curve are expressed as a function of a real variable, conventionnally denoted t representing *time*.

For instance, the parametric representation of circle in $\mathbb{R}^2$ is given by

$$\gamma : t \mapsto \begin{pmatrix} x_0 + r\cos t \\ y_0 + r\sin t \end{pmatrix}$$

where the points of the curve are defined as $\Gamma = \mathrm{Im}(\gamma) = \{\gamma(t), t \in \mathbb{R}\}$.

The cartesian representation is an equation (or a set of equations) that describes the relations between the coordinates of the points of the curve. Such representation can be explicit $y = f(x)$ or implicit $f(x, y) = 0$.

In the case of a circle, we can express it implicitly by

$$\Gamma = \left\{(x, y) \in \mathbb{R}^2 \,\middle|\, (x - x_0)^2 + (y - y_0)^2 = r\right\}$$

or explicitly

$$\Gamma = \left\{(x, y) \in \mathbb{R}^2 \,\middle|\, y = y_0 \pm \sqrt{r - (x - x_0)^2}\right\}$$

Generally speaking, the parametric representation can be more expressive than cartesian equations. It is also worth noting that tracking a curve's behaviour in a small neighborhood of a point on the curve is much simpler as the derivative of the parametric function $\gamma'(t)$ is easy to calculate and study.

In the following sections we will study active contours with respect to their curve representation, as described by G. Peyré (2011).

- Parametric active contours
- Implicit active contours

2 Parametric Active Contours

2.1 Formulation

In this section we will study an active contour method using a parametric curve mapped into the complex plane as we only manipulate 2D images.

$$\gamma : [0, 1] \mapsto \mathbb{C}$$

To implement our methods, we consider the discrete piecewise affine curve composed of p segments, the parametric function can therefore be considered a vector $\gamma \in \mathbb{C}^p$.

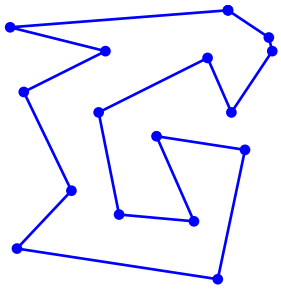
Let's initialize a closed curve which is in the discrete case a polygon.

```
x0 = np.array([.78, .14, .42, .18, .32, .16, .75, .83, .57, .68, .46, .40, .72, .79, .91, .90])
y0 = np.array([.87, .82, .75, .63, .34, .17, .08, .46, .50, .25, .27, .57, .73, .57, .75, .79])
gamma0 = x0 + 1j * y0
```

It would be useful to have a `plot` wrapper for our curve format for the rest of this section. We would want to link the last point with the first $\gamma_{p+1} = \gamma_1$.

```
# close the curve
periodize = lambda gamma: np.concatenate((gamma, [gamma[0]]))

# plot wrapper
def cplot(gamma, s='b', lw=1, show=False):
    gamma = periodize(gamma)
    _ = plt.plot(gamma.real, gamma.imag, s, linewidth=lw)
    _ = plt.axis('tight')
    _ = plt.axis('equal')
```

```
    _ = plt.axis('off')
    if show:
        plt.show()

# plot
cplot(gamma0, 'b.-', show=True)
```



Let's define a few inline functions for resampling the curve according to its length.

```
def resample(gamma, p, periodic=True):
    if periodic:
        gamma = periodize(gamma)
    # calculate segments lengths
    d = np.concatenate(([0], np.cumsum(1e-5 + np.abs(gamma[:-1] - gamma[1:]))))
    # interpolate gamma at new points
    return np.interp(np.linspace(0, 1, p, endpoint=False),  d/d[-1], gamma)
```

Let's initialize $\gamma_1 \in \mathbb{C}^p$ for $p = 256$.

```
# resample gamma
gamma1 = resample(gamma0, p=256)
cplot(gamma1, 'k', True)
```

Define forward and backward finite difference for approximating derivatives.

```
BwdDiff = lambda c: c - np.roll(c, +1)
FwdDiff = lambda c: np.roll(c, -1) - c
```

Thanks to these helper function, we can now define the tangent and the normal of γ at each point. We define the tangent as the *normalized vector* in the direction of the derivative of γ at a given time.

$$t_\gamma(t) = \frac{\gamma'(t)}{\|\gamma'(t)\|}$$

The normal at a given time is simply the orthogonal vector to the tangent.

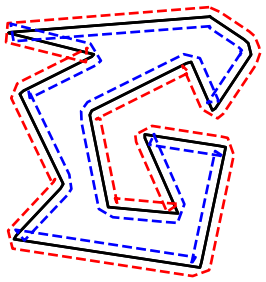
$$n_\gamma(t) = t_\gamma(t)^\perp$$

.

```
normalize = lambda v: v / np.maximum(np.abs(v), 1e-10) # disallow division by 0
tangent = lambda gamma: normalize(FwdDiff(gamma))
normal = lambda gamma: -1j * tangent(gamma)
```

Let's show the curve moved in the normal direction $\gamma_1(t) \pm \Delta n_{\gamma_1}(t)$

```
delta = .03
dn = delta * normal(gamma1)
cplot(gamma1, 'k')
cplot(gamma1 + dn, 'r--')
cplot(gamma1 - dn, 'b--')
plt.show()
```



Now that we have defined our curve and implemented basic functions for manipulating and displaying our curves, let's dive into the method.

2.2 Curve evolution

A curve evolution is a series of curves $s \mapsto \gamma_s$ indexed by an evolution parameter $s \geq 0$. The intial curve γ_0 is evolved by minimizing the curve's energy $E(\gamma_s)$ using the gradient descent algorithm. Which corresponds to minizing the energy flow.

$$\frac{\mathrm{d}}{\mathrm{d}s}\gamma_s = -\nabla E(\gamma_s)$$

The numerical implementation of the method is formulated as

$$\gamma^{(k+1)} = \gamma^{(k)} - \tau_k \cdot \nabla E(\gamma^{(k)})$$

In order to define our energy, we consider a smooth Riemannian manifold equipped with an inner product on the tangent space at each point of the curve.

The inner product along the curve is defined by

$$\langle \mu,\nu \rangle_\gamma = \int_0^1 \langle \mu(t),\nu(t) \rangle \, \|\gamma'(t)\| \, \mathrm{d}t$$

In this section we will consider intrinsic energy functions.

2.3 Intristic curve evolution

Intrinsic energy is defined along the normal, it only depends on the curve itself. We express the curve evolution as the speed along the normal.

$$\frac{\mathrm{d}}{\mathrm{d}s}\gamma_s(t) = \underbrace{\beta\left(\gamma_s(t), n_s(t), \kappa_s(t)\right)}_{\text{speed}} n_s(t)$$

where $\kappa_\gamma(t)$ is the intrinsic curvature of $\gamma(t)$ defined as

$$\kappa_\gamma(t) = \frac{1}{\|\gamma'(t)\|^2} \langle n'(t),\gamma'(t) \rangle$$

The speed term $\beta(\gamma, n, \kappa)$ is defined by the method.

2.3.1 Mean curvature motion

Evolution by mean curvature is based on minimizing the curve length and consequently its curvature. It is in fact the simplest curve evolution method.

The energy is therefore simply the length of the curve

$$E(\gamma) = \int_0^1 \|\gamma'(t)\| \, \mathrm{d}t$$

The energy gradient is therefore is therefore

$$\nabla E(\gamma_s)(t) = -\kappa_s(t) \cdot n_s(t)$$

In the method, the speed function defined simply as $\beta(\gamma, n, \kappa) = \kappa$.

For simplifying calculations, we define the function `curve_step` that calculates a curve step along the normal: $\mathrm{d}\gamma_s(t) = \kappa_s(t) \cdot n_s(t)$.

```
curve_step = lambda gamma: BwdDiff(tangent(gamma)) / np.abs(FwdDiff(gamma))
```

We perform this method on γ_1.

```
# initialize method
dt = 0.001 / 100            # time step
Tmax = 3.0 / 100            # stop time
niter = round(Tmax / dt)    # number of iterations
nplot = 10                  # number of plots
plot_interval = round(niter / nplot)

gamma = gamma1              # initial curve
plot_iter = 0               # plot iterator

for i in range(niter + 1):
    gamma += dt * curve_step(gamma)     # evolve curve
    gamma = resample(gamma, p=256)      # resample curve
    if i == plot_iter:
        lw = 4 if i in [0, niter] else 1
        cplot(gamma, 'r', lw)           # plot curve
        plot_iter += plot_interval      # increment plots
plt.show()
```

2.3.2 Geodesic motion

In a Riemannian manifold, the geodesic distance is the shortest path between two points, which is formulated as the weighted length.

$$L(\gamma) = \int_0^1 W(\gamma(t)) \, \|\gamma'(t)\| \, \mathrm{d}t$$

Where the weight $W(\cdot)$ is the geodesic metric defined as the square root of the quadratic form associated to the geodesic metric tensor $g(\cdot,\cdot)$.

$$W(x) = \sqrt{g(x,x)} \geq 0$$

The speed term of the evolution equation is therefore defined for geodesic motion as

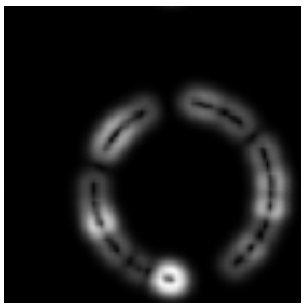
$$\beta(x,n,\kappa) = W \cdot \kappa - \langle \nabla W, n \rangle$$

Let's implement a random synthetic weight $W(x)$.

```
# import math constants
from numpy import pi
tau = pi * 2

# image dimensions
n = 200
nbumps = 40
r = 0.6 * n / 2

# generate random weight
theta = tau * np.random.rand(nbumps,1)
a = np.array([.62*n,.6*n])
x = np.around(a[0] + r * np.cos(theta))
y = np.around(a[1] + r * np.sin(theta))
W = np.zeros([n,n])
for i in np.arange(0,nbumps):
    W[int(x[i]), int(y[i])] = 1
W = gaussian_blur(W, 6.0)
W = rescale(-np.minimum(W, .05), .3,1)

# plot resulting image
imageplot(W)
plt.show()
```

Now that we have our can calculate its gradient.

```
# calculate gradient
G = grad(W)
G = G[:,:,0] + 1j*G[:,:,1]

# display its magnitude
imageplot(np.abs(G))
plt.show()
```



Let's define functions for evaluating $W(\gamma(t))$ and $\nabla W(\gamma(t))$.

```
EvalW = lambda W, gamma: bilinear_interpolate(W, gamma.imag, gamma.real)
EvalG = lambda G, gamma: bilinear_interpolate(G, gamma.imag, gamma.real)
```

Now let's test the method by creating a circular curve.

```
r = .98 * n/2   # radius
p = 128         # number of curve segments
i_theta = np.linspace(0, 2j * pi, p, endpoint=False)
im_center = (1 + 1j) * (n / 2)
gamma0 = im_center + r * np.exp(i_theta)
```

Let's define the dot product for complex vectors.

```
dotp = lambda a, b: a.real * b.real + a.imag * b.imag
```

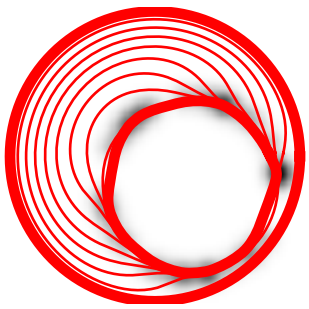
In order to implement a generic geodesic active contour, we should consider the case of open curves.
The evolution of open curves can be performed by imposing boundary conditions.

$$\begin{cases} \gamma(0) = x_0 \\ \gamma(1) = x_1 \end{cases}$$

The algorithm finds therefore the minimal geodesic distance between the two endpoints.

```
def geodesic_active_contour(gamma, W, f, p, dt, Tmax, n_plot, periodic=True, endpts=None):
    # initialize iteration variables
    niter = round(Tmax / dt)
    plot_interval = round(niter / nplot)
    plot_iter = 0

    # calculate gradient
    G = grad(W)
    G = G[:,:,0] + 1j * G[:,:,1]

    imageplot(f.T)
    for i in range(niter + 1):
        N = normal(gamma)                          # calculate normal
        gamma_step = EvalW(W, gamma) * curve_step(gamma) - dotp(EvalG(G, gamma), N) * N
        gamma += dt * gamma_step                   # evolve curve
        gamma = resample(gamma, p, periodic)       # resample curve
        # impose endpoints on open curves
        if not periodic and endpts is not None:
            gamma[0], gamma[-1] = endpts
        if i in [plot_iter, niter]:
```

```
            lw = 4 if i in [0, niter] else 1
            cplot(gamma, 'r', lw)               # plot curve
            plot_iter += plot_interval          # increment plots
            if not periodic:
                # plot endpoints
                _ = plt.plot(gamma[0].real,  gamma[0].imag, 'b.', markersize=20)
                _ = plt.plot(gamma[-1].real, gamma[-1].imag, 'b.', markersize=20)
    plt.show()

# test the method on the random weights
geodesic_active_contour(gamma0, W, W, p=128, dt=1, Tmax=5000, n_plot=10)
```



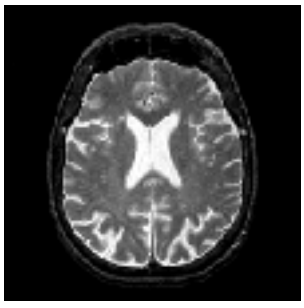
2.4 Medical image segmentation

2.4.1 Evolution of a closed curve

Let's now test the defined method to detect edges in a medical image.

We first load the image as $f : [0,1]^2 \to \mathbb{R}$, where the domain of f is approximated by the discrete space $\{0,\dots,n-1\}^2$.

```
name = 'nt_toolbox/data/cortex.bmp'
n = 256
f = load_image(name, n)
imageplot(f)
plt.show()
```

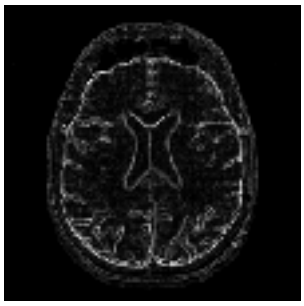


We define the weight as the decreasing function of the gradient magnitude.

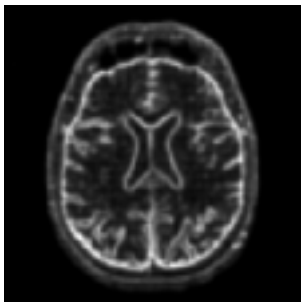
$$W(x) = \psi(d \star h_a(x)) \quad \text{where} \quad d(x) = \|\nabla f(x)\|$$

where h_a is a blurring kernel of width $a > 0$ implemented with the help of the library `nt_toolbox`.

```
# initialize timer variables
from time import time
start = time()

# calculate gradient magnitude
G = grad(f)
d0 = np.sqrt(np.sum(G**2, 2))
param_time = time() - start

# display
imageplot(d0)
plt.show()
```



```
start = time()
# apply gaussian blur
a = 2
d = gaussian_blur(d0, a)
param_time += time() - start

# display
imageplot(d)
plt.show()
```

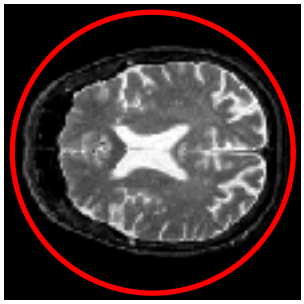


```
start = time()
# calculate W as decreasing function of d
d = np.minimum(d, .4)
W = rescale(-d, .8, 1)
param_time += time() - start

# display
imageplot(W)
plt.show()
```

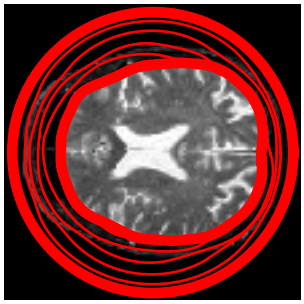
Now that we have our geodesic metric we can apply the method as before.

First let's take a look at our initial curve.

```
r = .95 * n / 2
p = 128
i_theta = np.linspace(0, 2j * pi, p, endpoint=False)
im_center = (1 + 1j) * (n / 2)
gamma0 = im_center + r * np.exp(i_theta)
imageplot(f.T)
cplot(gamma0, 'r', 2)
plt.show()
```



```
start = time()
geodesic_active_contour(gamma0, W, f, p, dt=2, Tmax=9000, n_plot=10)
```



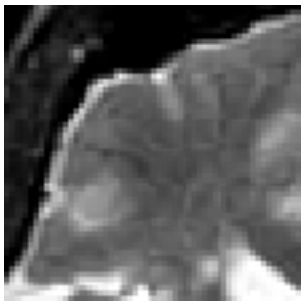
```
param_time += time() - start
print(f"Time elapsed:\t{param_time} seconds")
```

```
Time elapsed:   2.232177972793579 seconds
```

2.4.2 Evolution of an open curve

Let's apply the method on an open curve in a segment of the same image.

```
f = f[45:105, 60:120]
n = f.shape[0]
imageplot(f)
plt.show()
```



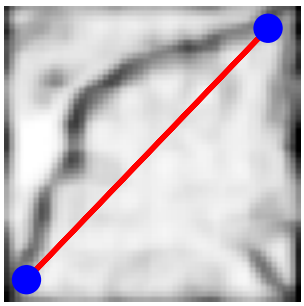
We reapply the same steps to find the geodesic metric.

```
G = grad(f)
G = np.sqrt(np.sum(G**2,2))
sigma = 1.5
G = gaussian_blur(G,sigma)
G = np.minimum(G,.4)
W = rescale(-G,.4,1)
imageplot(W)
plt.show()
```

We define our boundary conditions.

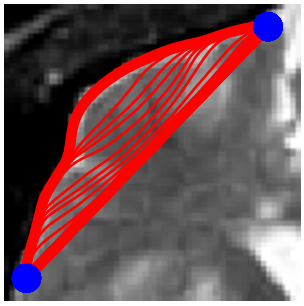
```
# boundary conditions
x0 = 4 + 55j
x1 = 53 + 4j

# initial curve as straight segment
p = 128
t = np.linspace(0, 1, p)
gamma0 = t*x1 + (1-t)*x0
gamma = gamma0

# plot metric
imageplot(W.T)
```

```
cplot(gamma,'r', 2)
_ = plt.plot(gamma[0].real, gamma[0].imag, 'b.', markersize=20)
_ = plt.plot(gamma[-1].real, gamma[-1].imag, 'b.', markersize=20)
plt.show()
```



Now that we have our initial curve, geodesic metric and boundary conditions we can perform the geodesic active contour method.

```
geodesic_active_contour(gamma0, W, f, p=128, dt=0.1, Tmax=8000/7, n_plot=10,
    periodic=False, endpts=(x0, x1))
```



3 Level-Set Method

A level set of a function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$L_c(f) = \{x \in \mathbb{R}^n \,|\, f(x) = c\}$$

In the case of $n = 2$ the level set corresponds to an implicit cartesian representation of a curve, also called *isoline*.

The Level-Set Method (LSM) defines the curve Γ as the zero level set of a function $\varphi : \mathbb{R}^2 \to \mathbb{R}$ called the **level function**.

$$\Gamma = \{x \in \mathbb{R}^2 \,|\, \varphi(x) = 0\}$$

The essence of the method is evolving the curve Γ implicitly through its level function φ.

LSM allows manipulating hypersurfaces in different contexts without having their parametric representation. It can be very useful in tracking changing topologies and in contour detection.

In this part we will consider images defined over a domain $[0, 1]^2$ discretized in an $n \times n$ grid. As a preliminary example, let's consider the level set represenation of a circle of radius r centered in $c \in \mathbb{R}^2$.

$$\varphi_1(x) = \|x - c\|_2 - r$$

where $\|\cdot\|_2$ is the well-known euclidean norm L^2.

The level-set function divides the domain into three sets:

- $\Gamma = \{x \in \mathbb{R}^2 \,|\varphi_1(x) = 0\} = \{x \in \mathbb{R}^2 \,|\|x - c\|_2 = r\}$ = points lying on the circle.
- $L_- = \{x \in \mathbb{R}^2 \,|\varphi_1(x) < 0\} = \{x \in \mathbb{R}^2 \,|\|x - c\|_1 < r\}$ = points lying inside the circle.
- $L_+ = \{x \in \mathbb{R}^2 \,|\varphi_1(x) < 0\} = \{x \in \mathbb{R}^2 \,|\|x - c\|_1 < r\}$ = points lying outside the circle.

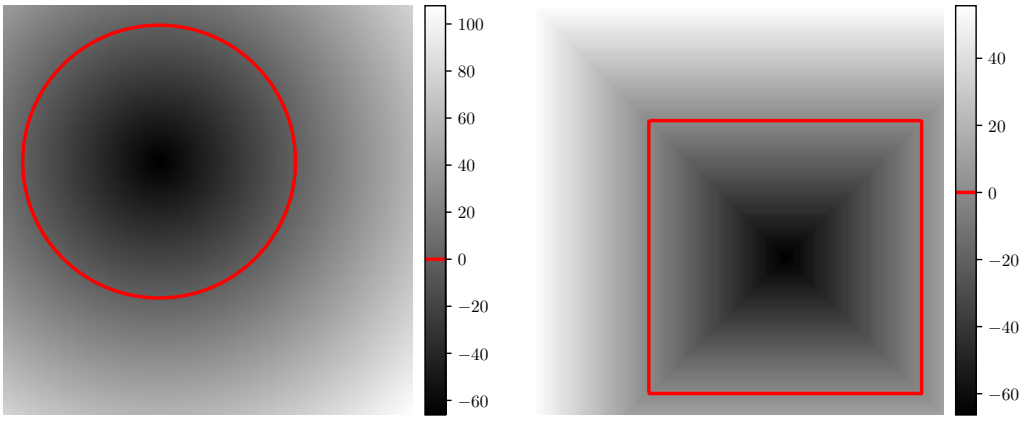
Similarly we can define φ_2 the level-set function of a square centered in c whose side is $2r$.

$$\varphi_2(x) = \|x - c\|_\infty - r$$

where $\|\cdot\|_\infty$ is the maximum norm L^∞.

Let's plot our level functions as a greyscale color mappings.

```
# initialize grid
n = 200
x = y = np.arange(1, n + 1)
X, Y = np.meshgrid(x, y, sparse=True, indexing='xy')

# calculate phi1
r = n / 3
c = np.array([r,r]) + 10
phi1 = np.sqrt((X-c[0])**2 + (Y-c[1])**2) - r

# calculate phi2
c = n - c
phi2 = np.maximum(abs(X-c[0]), abs(Y-c[1])) - r

# plot mappings
from util import plot_levelset
_ = plt.figure(figsize = (10,5))
_ = plt.subplot(121)
plot_levelset(phi1, colorbar=True)
_ = plt.subplot(122)
plot_levelset(phi2, colorbar=True)
plt.show()
```



The level-set representation allows us to compute easily the intersection and the union of two regions.

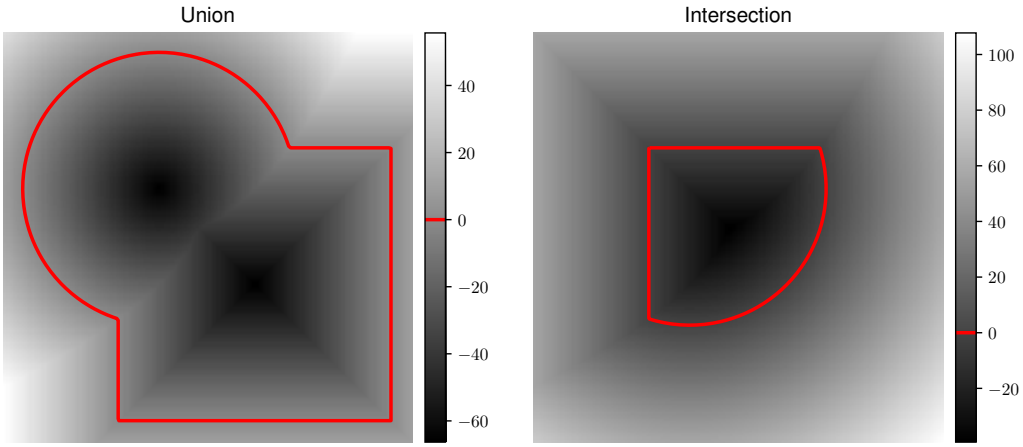
The level-set function of the union of domains is defined as the minimum of the domains' functions.

$$\varphi_\cup = \min(\varphi_1, \varphi_2)$$

Similarly, the level-set function of the intersection of domains is defined by the maximum.

$$\varphi_{\cap} = \max(\varphi_1, \varphi_2)$$

```
_ = plt.figure(figsize = (10,5))

_ = plt.subplot(121)
_ = plt.title("Union")
plot_levelset(np.minimum(phi1, phi2), colorbar=True)

_ = plt.subplot(122)
_ = plt.title("Intersection")
plot_levelset(np.maximum(phi1, phi2), colorbar=True)

plt.show()
```



3.1 Curve evolution

As mentioned before, LSM manipulates curves implicitly through its level-set function. Similarly to parametric curve evolution, LSM curve evolution is a series of level-set functions $s \mapsto \varphi_s$.

The evolution speed was defined for parametric curves in function of the curve function, its normal and its intrinsic curvature. These terms can be redefined for curves expressed as zero-level sets as follows:

- **Normal:** $n(x) = \frac{\nabla\varphi(x)}{\|\nabla\varphi(x)\|}$
- **Curvature:** $\kappa(x) = \operatorname{div}\left(\frac{\nabla\varphi}{\|\nabla\varphi\|}\right)(x)$

The evolution PDE of the level-set function (called the *Level-Set Equation*) is

$$\frac{\mathrm{d}}{\mathrm{d}s}\varphi_s = \beta(\varphi_s(x), n_s(x), \kappa_s(x)) \cdot \|\nabla\varphi_s\|$$

3.2 Mean curvature motion

As for parametric curves, mean curvature motion is based on minimizing the normal energy flow which is characterized by the curve's intrinsic curvature.

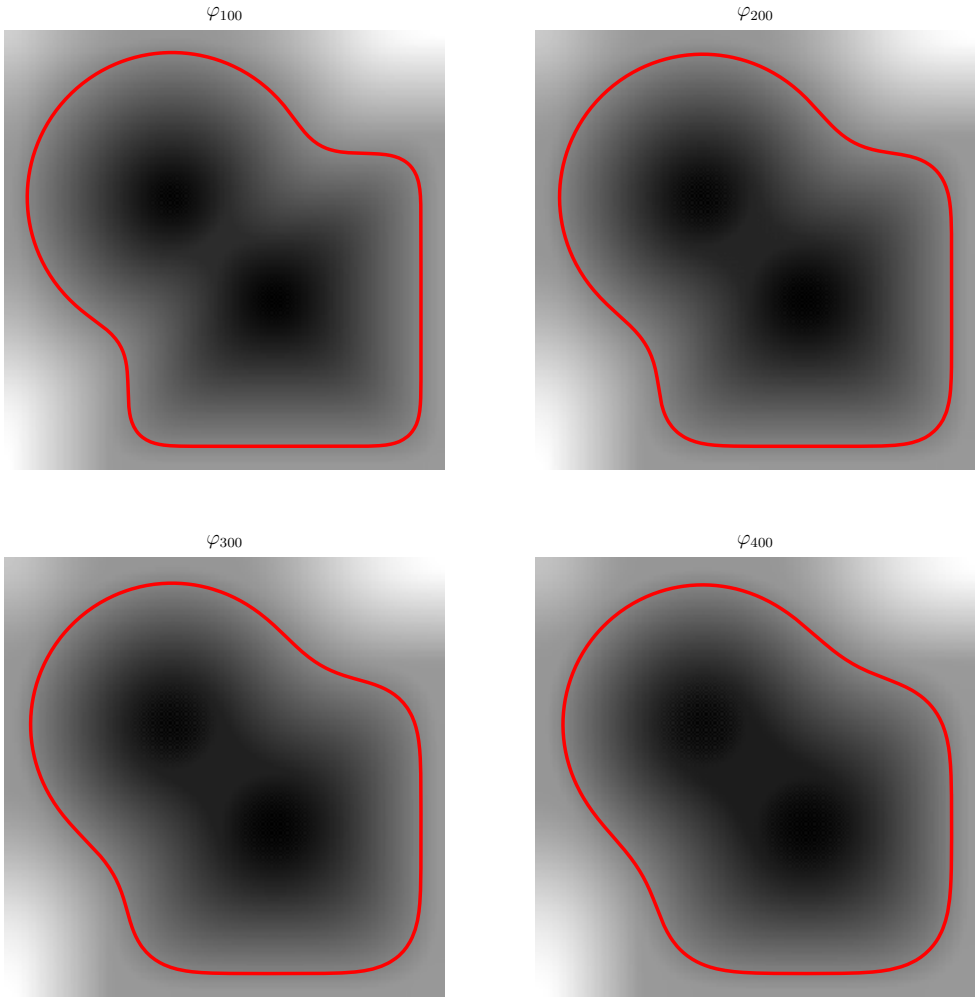
We have seen that the speed function is defined as $\beta(\varphi, n, \kappa) = \kappa$.

The level-set equation becomes

$$\frac{\mathrm{d}}{\mathrm{d}s}\varphi_s = \mathrm{div}\left(\frac{\nabla\varphi_s}{\|\nabla\varphi_s\|}\right)\cdot\|\nabla\varphi_s\|$$

Let's perform this method on the union of the previous curves.

```
from nt_toolbox.grad import *
from nt_toolbox.div import *

dt = 0.5                    # time step
Tmax = 200                  # stop time
niter = round(Tmax / dt)    # number of iterations
nplot = 4                   # number of plots
plot_interval = round(niter / nplot)

phi0 = np.minimum(phi1, phi2)   # union curve
phi = np.copy(phi0)         # initial curve
plot_iter = plot_interval   # plot iterator
subplot = 1                 # subplot counter


_ = plt.figure(figsize=(10,10))
eps = np.finfo(float).eps

for i in range(niter + 1):
    # g0 = grad(phi)
    g0 = grad(phi, order=2)
    # d = |grad(phi)|
    d = np.maximum(np.sqrt(np.sum(g0**2, 2)), eps)
    # g = grad(phi)/|grad(phi)|
    g = g0 / np.repeat(d[:,:,np.newaxis], 2, 2)
    # K = div(g)
    K = div(g[:,:,0], g[:,:,1], order=2)
    # calculate phi step
    G = K * d
    phi += dt * G

    # plot levelset
    if i == plot_iter and subplot <= 4:
        ax = _ = plt.subplot(2, 2, subplot)
        _ = ax.set_title(r'$\varphi_{' + str(i) + '}$')
        plot_levelset(phi)
        subplot += 1
        plot_iter += plot_interval

plt.show()
```



3.3 Level-set redistancing

While the essential property of the level set function φ is the location of its zero isoline, in practice many applications additionally require that it be a signed distance function $\|\nabla\varphi\| = 1$ which ensures that the zero crossing is sufficiently sharp.

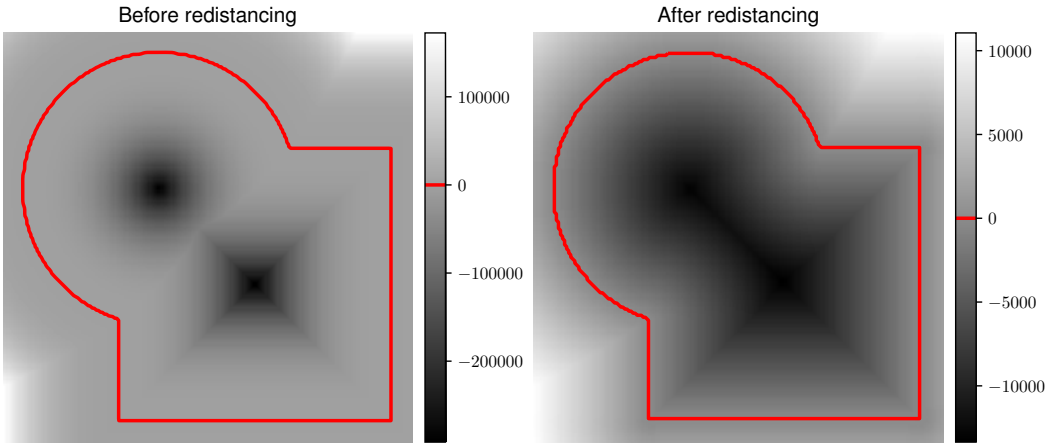
Nevertheless, this property is not generally preserved by the level-set evolution. **Redistancing** is the process of recovering this property without modifying the location of the zero isoline.

Redistancing essentially is computing a signed distance function $\tilde{\varphi}$ from an arbitrary non-signed distance function φ while preserving the zero isoline. Mathematically, this process obeys the *eikonal equation*. (Royston 2017)

$$\begin{cases} \|\nabla\tilde{\varphi}\| = 1 \\ \text{sign}(\tilde{\varphi}) = \text{sign}(\varphi) \end{cases}$$

We can set φ initially to φ_0^3 as $x \mapsto x^3$ preserves the sign (consequently the isoline), we obtain the signed disntance function $\tilde{\varphi}$ from φ (which is a non-signed distance) by performing redistancing with the help of the methods of `nt_toolbox`.

```
phi = phi0**3

from nt_toolbox.perform_redistancing import *
phi1 = perform_redistancing(phi0)

_ = plt.figure(figsize=(10,5))

_ = plt.subplot(1,2,1)
_ = plt.title("Before redistancing")
plot_levelset(phi, colorbar=True)

_ = plt.subplot(1,2,2)
_ = plt.title("After redistancing")
plot_levelset(phi1, colorbar=True)

plt.show()
```



3.4 Geodesic motion

Unlike geodesic motion of parametric curves, the evolution of the zero level-set is computed along *local minima* of the weighted geodesic distance attracting the curve toward the features of the background image.

Given a background image f_0 to segment, we compute the geodesic weight metric W. The geodesic weight should be a decreasing function of the blurred gradient magnitude.

$$W(x) = \psi\underbrace{(d_0 \star h_a)}_{d} \quad \text{where} \quad d_0 = \|\nabla f_0\|$$

given the blurring kernel h_a of size $a > 0$.

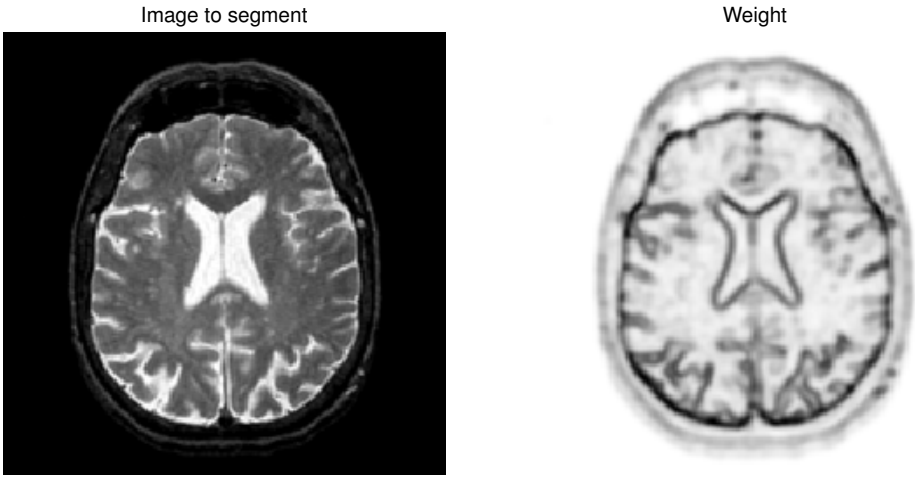
Let's calculate first the blurred gradient magnitude.

```
# load image of size n×n
f0 = rescale(load_image("nt_toolbox/data/cortex.bmp", n))
start = time()

# compute the magnitude of the gradient
g = grad(f0, order=2)
d0 = np.sqrt(np.sum(g**2, 2))
```

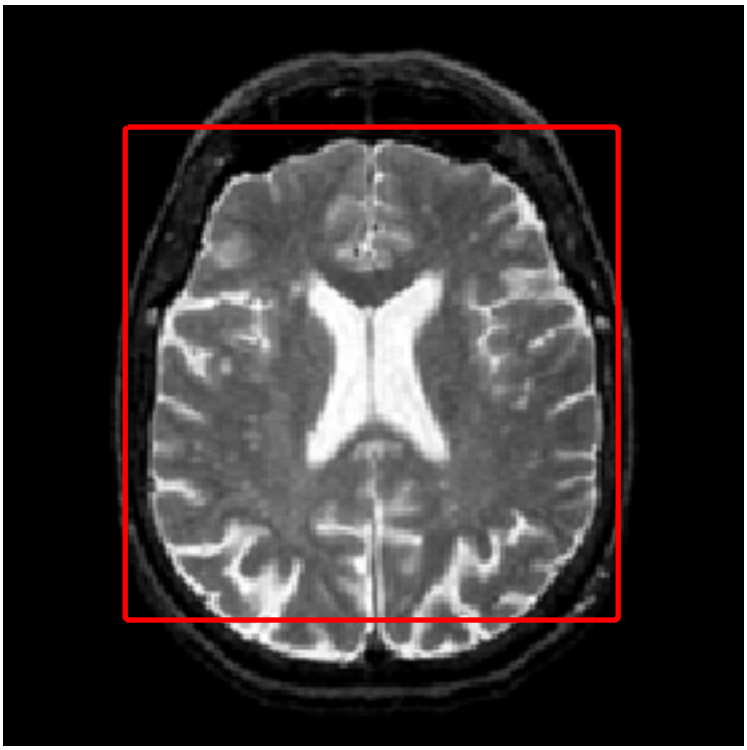
```
# perform blurring
from nt_toolbox.perform_blurring import *
a = 5
d = perform_blurring(d0, np.asarray([a]), bound="per")

lsm_time = time() - start
```

The decreasing function ψ can be defined as $\psi : s \mapsto \alpha + \frac{\beta}{\varepsilon+s}$. Nevertheless, the function `rescale` from the library `nt_toolbox` can adjust the overall values of W as needed without having to adjust the parameters α and β.

```
start = time()
# calculate weight
epsilon = 1e-1
W = 1./(epsilon + d)
W = rescale(-d, 0.1, 1)
lsm_time += time() - start

# display
_ = plt.figure(figsize=(10,5))
imageplot(f0, "Image to segment", [1,2,1])
imageplot(W, "Weight", [1,2,2])
plt.show()
```




```
# initialize centered square contour
r = n / 3
c = np.asarray([n,n]) / 2
phi0 = np.maximum(np.abs(X-c[0]), np.abs(Y-c[1])) - r

# display
_ = plt.figure(figsize=(5,5))
plot_levelset(phi0, img=f0)
plt.show()
```



We remind the weighted length defined for a parametric curve

$$L(\gamma) = \int_0^1 W(\gamma(t))\,\|\gamma'(t)\|\,\mathrm{d}t$$

and the speed term of the evolution equation

$$\beta(x, n, \kappa) = W \cdot \kappa - \langle \nabla W, n \rangle$$

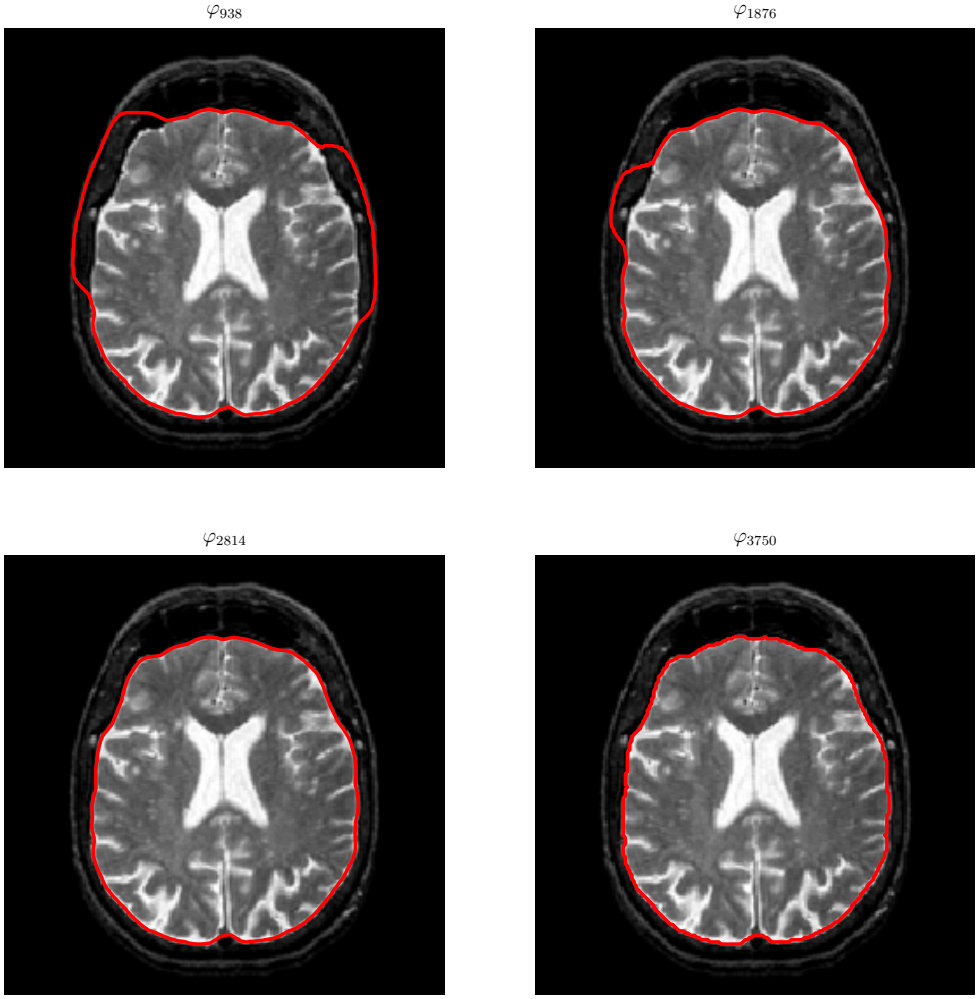
The evolution equation for the level-set function is defined as

$$\frac{\mathrm{d}}{\mathrm{d}s}\varphi_s = \mathrm{div}\left(W\frac{\nabla\varphi_s}{\|\nabla\varphi_s\|}\right)\cdot\|\nabla\varphi_s\|$$

Let's apply the method on our initial square zero isoline.

```
start = time()

dt = 0.4                        # time step
Tmax = 1500                     # stop time
niter = round(Tmax / dt)        # number of iterations
nplot = 4                       # number of plots
plot_interval = round(niter / nplot)
n_redistancing = 30

phi = np.copy(phi0)             # initial curve
gW = grad(W, order=2)           # grad(W)
plot_iter = plot_interval       # plot iterator
subplot = 1                     # subplot counter
```

```
_ = plt.figure(figsize=(10,10))

for i in range(niter + 1):
    # g0 = grad(phi)
    g0 = grad(phi, order=2)
    # d = |grad(phi)|
    d = np.maximum(np.sqrt(np.sum(g0**2, 2)), eps)
    # g = grad(phi)/|grad(phi)|
    g = g0 / np.repeat(d[:,:,np.newaxis], 2, 2)
    # K = div(g)
    K = div(g[:,:,0], g[:,:,1], order=2)
    # calculate phi step
    G = W * d * K + np.sum(gW * g0, 2)
    phi += dt * G

    # perform redistancing
    if i % n_redistancing == 0:
        phi = perform_redistancing(phi)

    # plot levelset
    if i in [plot_iter, niter]:
        ax = _ = plt.subplot(2, 2, subplot)
        _ = ax.set_title(r'$\varphi_{' + str(i) + '}$')
        plot_levelset(phi, img=f0)
        subplot += 1
        plot_iter += plot_interval

plt.show()
```




```
lsm_time += time() - start
print(f"Time elapsed:\t{lsm_time} seconds")
```

```
Time elapsed:   14.876367568969727 seconds
```

3.5 Region-based Chan-Vese segmentation

Chan-Vese active contours corresponds to a region-based energy that looks for a piecewise constant approximation of the image.

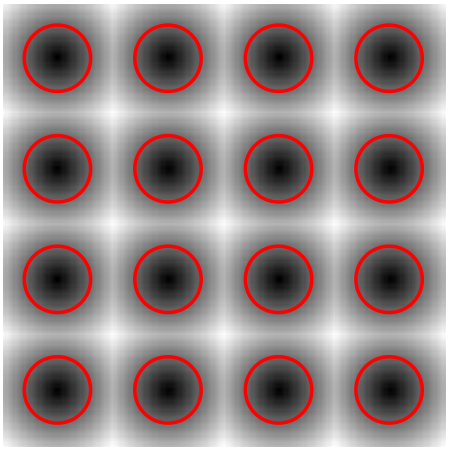
The energy to be minimized is

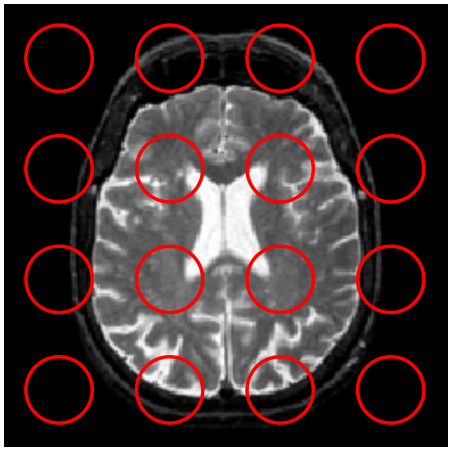
$$\min_{\varphi} \left(L(\varphi) + \lambda \int_{\varphi(x)>0} |f_0(x) - c_1|^2 \, dx + \lambda \int_{\varphi(x)<0} |f_0(x) - c_2|^2 \, dx \right)$$

where L is the length of the zero level set of φ. Note that here $(c_1, c_2) \in \mathbb{R}^2$ are assumed to be known.

Let's initialize our regions.

```
_ = plt.figure(figsize=(10,5))
k = 4                # number of circles
n_k = n / k          # n per region
r = 0.3 * n_k        # circles' radius

# initialize phi to infinity
phi0 = np.zeros([n,n]) + np.float("inf")

# calculate negative regions
for i in range(1, k+1):
    for j in range(1, k+1):
        c = n_k * (np.asarray([i,j]) - 0.5)
        phi0 = np.minimum(phi0, np.sqrt(abs(X-c[0])**2 + abs(Y-c[1])**2) - r)

# display regions
_ = plt.subplot(1,2,1)
plot_levelset(phi0)
_ = plt.subplot(1,2,2)
plot_levelset(phi0, img=f0)
plt.show()
```





The minimizing flow for the Chan-Vese energy can be expressed as

$$\frac{\mathrm{d}}{\mathrm{d}t}\varphi_t = -G(\varphi_t)$$

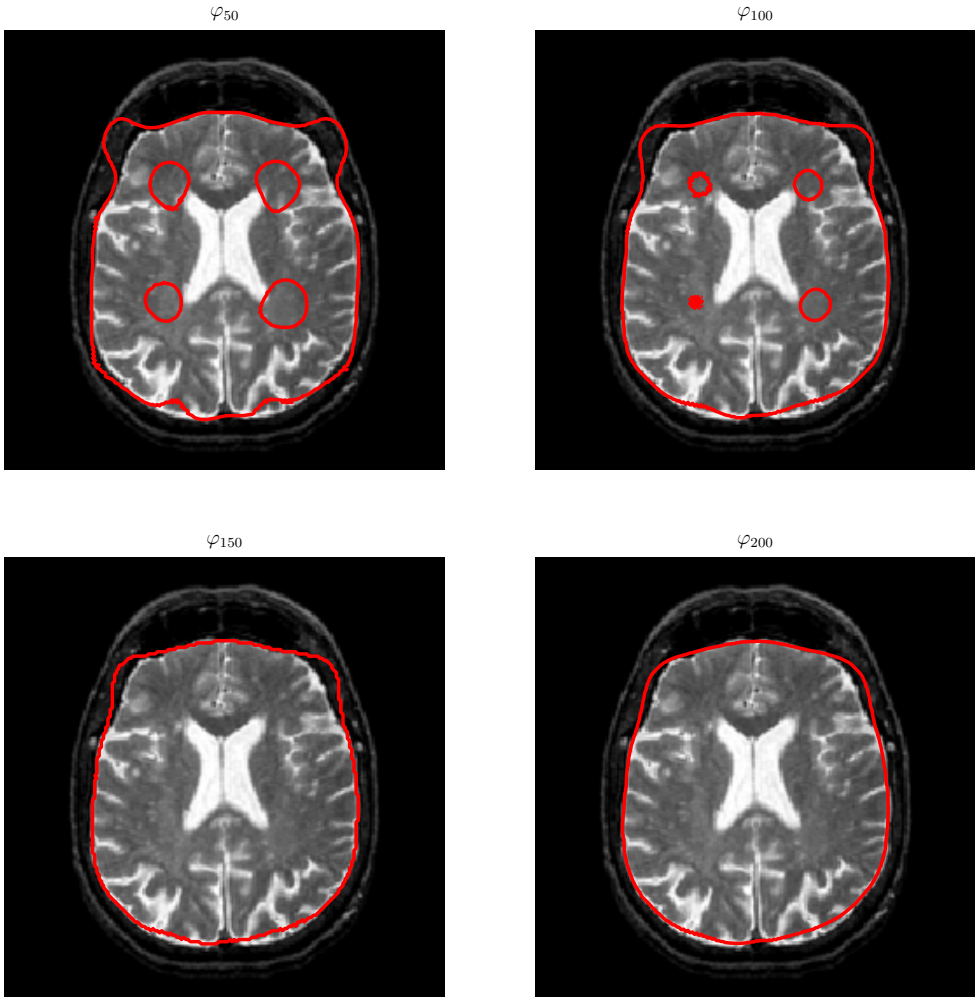
where

$$G(\varphi) = -\left\|\nabla\varphi\right\| \operatorname{div}\left(\frac{\nabla\varphi}{\left\|\nabla\varphi\right\|}\right) + \lambda(f_0 - c_1)^2 - \lambda(f_0 - c_2)^2$$

Now that we have our evolution equation well-defined we can apply our gradient descent method.

```
start = time()

dt = 0.5                    # time step
Tmax = 100                  # stop time
niter = round(Tmax / dt)    # number of iterations
nplot = 4                   # number of plots
plot_interval = round(niter / nplot)
n_redistancing = 30
```

```
lmbda = 2
c1, c2 = (0.7, 0)

phi = np.copy(phi0)          # initial curve
plot_iter = plot_interval    # plot iterator
subplot = 1                  # subplot counter

_ = plt.figure(figsize=(10,10))

for i in range(niter + 1):
    # g0 = grad(phi)
    g0 = grad(phi, order=2)
    # d = |grad(phi)|
    d = np.maximum(np.sqrt(np.sum(g0**2, 2)), eps)
    # g = grad(phi)/|grad(phi)|
    g = g0 / np.repeat(d[:,:,np.newaxis], 2, 2)
    # K = div(g)
    K = div(g[:,:,0], g[:,:,1], order=2)
    # calculate energy term for non-zero phi
    energy = (f0-c1)**2 - (f0-c2)**2
    # calculate phi step
    G = d * K - lmbda * energy
    phi += dt * G

    # perform redistancing
    if i % n_redistancing == 0 and i > 0:
        phi = perform_redistancing(phi)

    # plot levelset
    if i in [plot_iter, niter]:
        ax = _ = plt.subplot(2, 2, subplot)
        _ = ax.set_title(r'$\varphi_{' + str(i) + '}$')
        plot_levelset(phi, img=f0)
        subplot += 1
        plot_iter += plot_interval

plt.show()
```




```
cv_time = time() - start
print(f"Time elapsed:\t{cv_time} seconds")
```

```
Time elapsed:   1.01350998878479 seconds
```

4 Conclusion

Let's compare the execution time of each method for detecting the contour of the medical image.

```
results = "Time elapsed per method (in seconds):\n"
results += f"Parametric edge-based:\t{param_time}\n"
results += f"Implicit edge-based:\t{lsm_time}\n"
results += f"Implicit region-based:\t{cv_time}\n"

print(results)
```

```
Time elapsed per method (in seconds):
```

```
Parametric edge-based:  2.232177972793579
Implicit edge-based:    14.876367568969727
Implicit region-based:  1.01350998878479
```

We have seen two classes of active contour methods, the parametric representation allows simple calculations of evolution functions. However, the implicit representation manipulates 2D functions so the calculations are significantly more complex as they involve gradients and divergences of functions as well as they require more storage. Nevertheless, the implicit representations allows tracking topological changes and can therefore be preferred in analysing sequences of images (videos).

Thanks to *Numerical Tours* (Peyré 2011) we were able to explore a class of contour detection methods that can be applied to different problems. Peyré's tours have allowed us to understand and implement these methods on medical images.

This projet has allowed me to explore the domain of image processing for the first time, and have peaked my interest to seek further understanding of current approaches of contour detection, as well as other problems of image processing.

Références

"Active Contour Model." 2020. *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Active_contour_model&oldid=940601399.

"Computer Vision." 2020. *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Computer_vision&oldid=939131357.

Peyré, Gabriel. 2011. "The Numerical Tours of Signal Processing - Advanced Computational Signal and Image Processing." *IEEE Computing in Science and Engineering* 13 (4): 94–97. https://hal.archives-ouvertes.fr/hal-00519521.

Royston, Michael Wayne. 2017. "A Hopf-Lax Formulation of the Eikonal Equation for Parallel Redistancing and Oblique Projection." PhD thesis, UCLA. https://escholarship.org/uc/item/04f9942g.

"Snakes." 2011. ICBE, University of Manchester. https://web.archive.org/web/20110716113957/http://www.isbe.man.ac.uk/courses/Computer_Vision/downloads/L11_Snakes.pdf.

Weisstein, Eric W. n.d. "Closed Curve." Text. Accessed February 23, 2020. http://mathworld.wolfram.com/ClosedCurve.html.