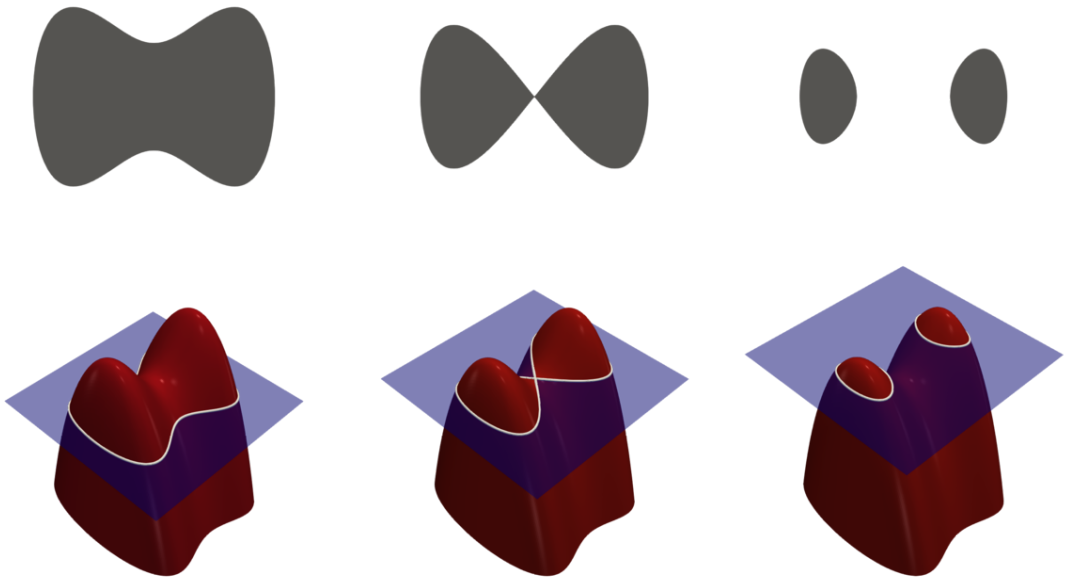
INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

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)

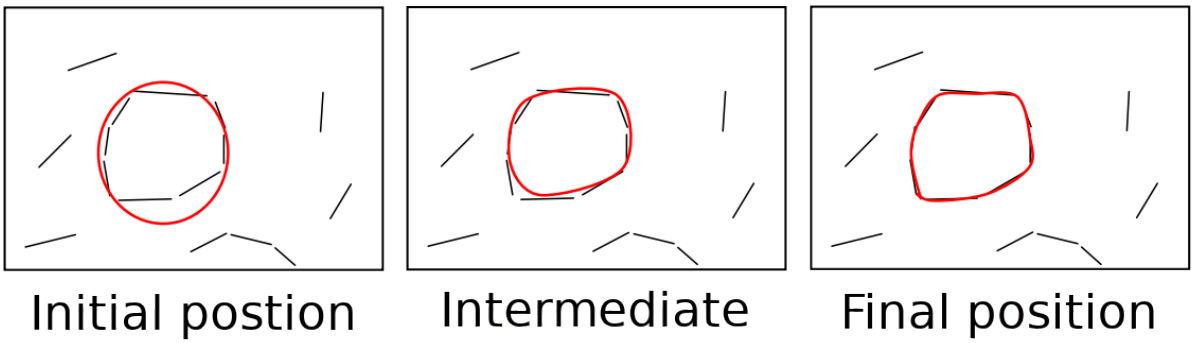


Figure 1: Illustration of the snakes model

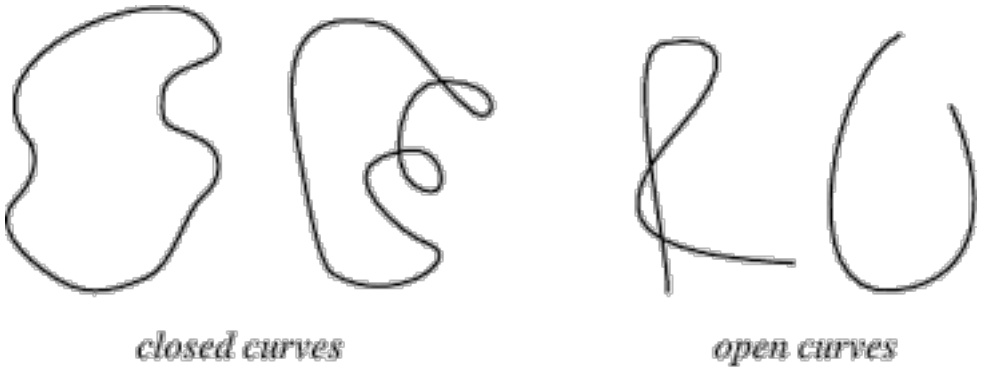
The energy of the snakes is a combination of internal and external energy ("Snakes" 2011)

- **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

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

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 edge-based active contours
- Implicit edge-based active contours
- Region-based 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 $\gamma \in \mathbb{C}^p$.

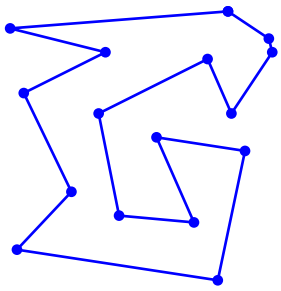
Let's initialize a closed curve which is in the discrete case a polygon.

```
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.

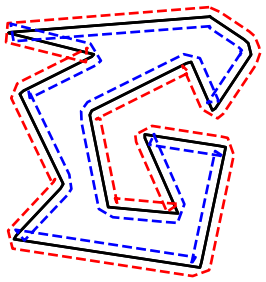
$$n_\gamma(t) = t_\gamma(t)^\perp$$

.

```
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

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

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 $g(\cdot,\cdot)$.

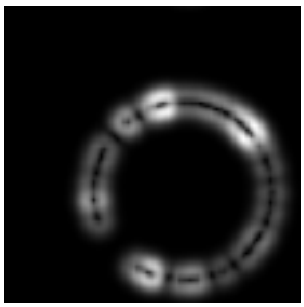
$$W(x) = \sqrt{g(x,x)} \geq 0$$

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
```

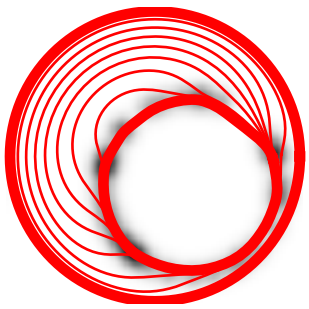
In order to implement a generic geodesic active contour, we should consider the case of open curves.
The evolution of open curves can be performed by imposing boundary conditions.

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

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)
```

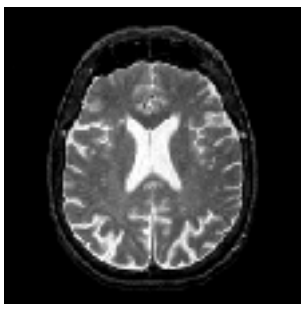


2.4 Medical Image Segmentation

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

We first load the image as $f : [0,1]^2 \to \mathbb{C}$, 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()
```

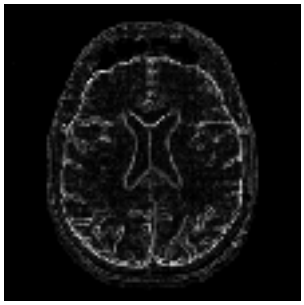


We define the weight as the decreasing function of the gradient magnitude.

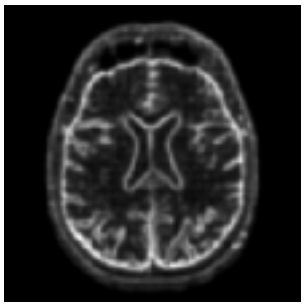
$$W(x) = \psi(d \star h_a(x)) \quad \text{where} \quad d(x) = \|\nabla f(x)\|$$

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

```
# calculate gradient magnitude
G = grad(f)
d0 = np.sqrt(np.sum(G**2, 2))
imageplot(d0)
plt.show()
```



```
# apply gaussian blur
a = 2
d = gaussian_blur(d0, a)
imageplot(d)
plt.show()
```

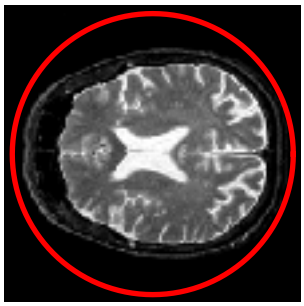


```
# calculate W as decreasing function of d
d = np.minimum(d, .4)
W = rescale(-d, .8, 1)
imageplot(W)
plt.show()
```

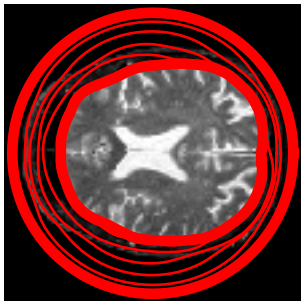
Now that we have our geodesic metric we can apply the method as before.

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()
```



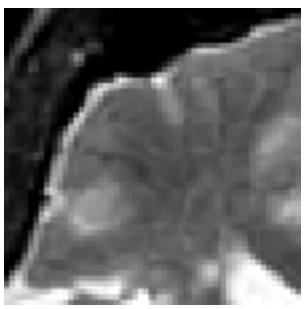
```
geodesic_active_contour(gamma0, W, f, p, dt=2, Tmax=9000, n_plot=10)
```



2.4.1 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()
```

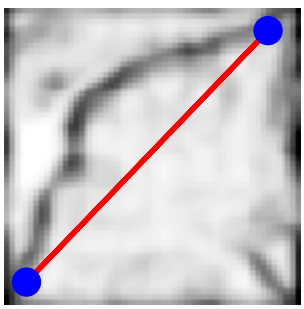


We reapply the same steps to find the geodesic metric.

```
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()
```

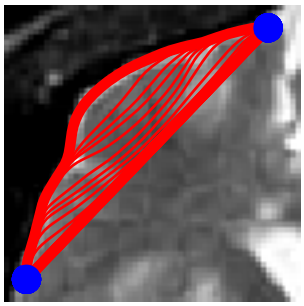
We define our boundary conditions.

```
# 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 Conclusion

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.

"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.