# Technical Report for CMPT 732, Assignment 1

# Active Contours

Sihui Wang 301474102

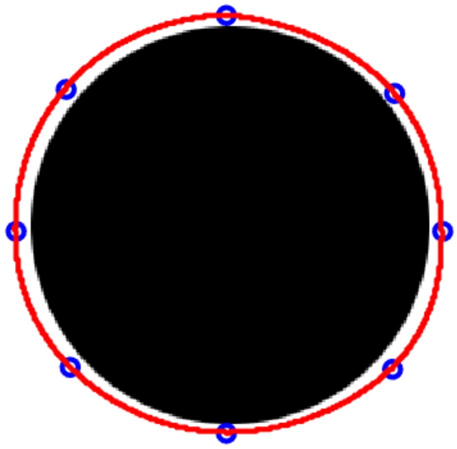
## 1. Main Results

***Note:*** *for images of results of higher resolution, please refer to the images under the folder 'init_snakes' for original images with the initialization, refer to images under the folder 'contour_results' for final snake curves, and refer to images under the folder 'segmentation_results' for the results of segmentations.*

*You can find the information about the preloaded control points under the folder 'control_points'.*

### 1.1 Circle

**The original image with the initialization**: *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

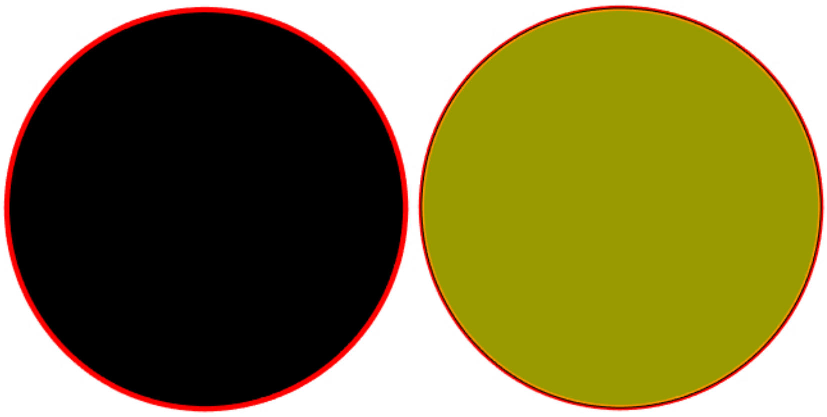


**Parameter Used:**

$\alpha = 100.0, \beta = 10.0, \gamma = 5.0, \kappa = 0.2, Wline = -8.0, Wedge = 1.0, Wterm = 1.0, \sigma = 0.5$

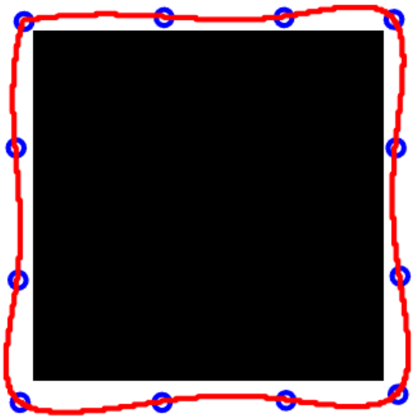
**Number of Iterations:** $N = 60$

**Results:** After 60 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)



### 1.2 Square

**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

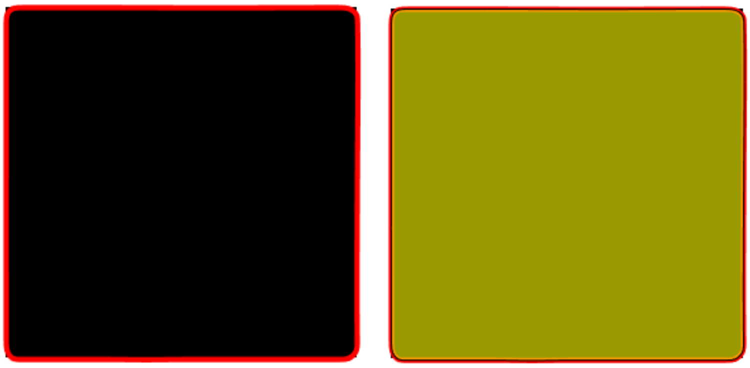


**Parameter Used:**

$\alpha = 3.0, \beta = 3.0, \gamma = 0.45, \kappa = 1.0, Wline = -1.0, Wedge = 1.0, Wterm = 1.0, \sigma = 2.0$

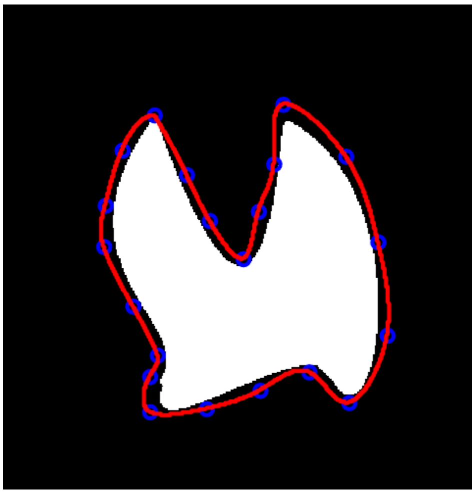
**Number of Iterations:** $N = 160$

**Results:** After 160 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)



### 1.3 Shape

**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.



**Parameter Used:**

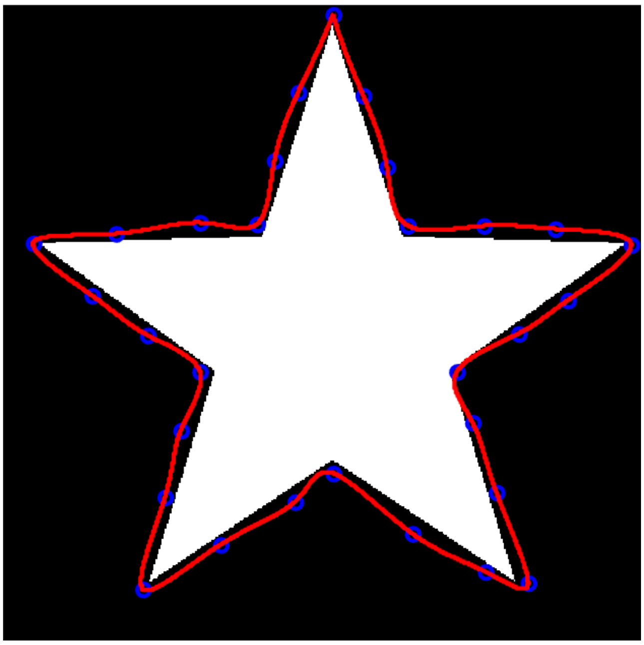
$\alpha = 0.8, \beta = 0.8, \gamma = 0.8, \kappa = 1.1, Wline = 0.8, Wedge = 2.0, Wterm = 2.0, \sigma = 2.0$

**Number of Iterations:** $N = 160$

**Results:** After 160 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)

**1.4 Star**

**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

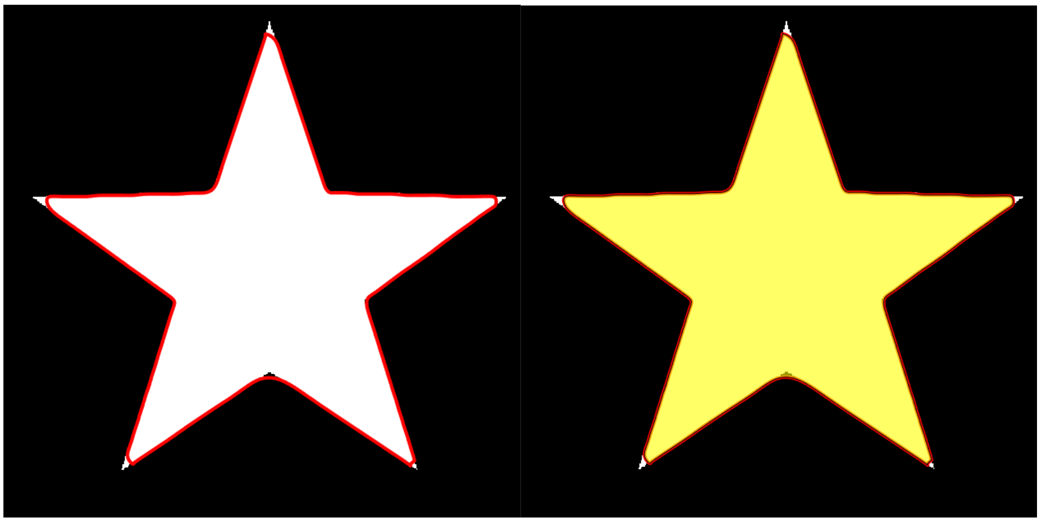


**Parameter Used:**

$\alpha = 0.1, \beta = 50.0, \gamma = 1.0, \kappa = 1.0, Wline = -3.0, Wedge = 3.0, Wterm = 0.0, \sigma = 2.2$

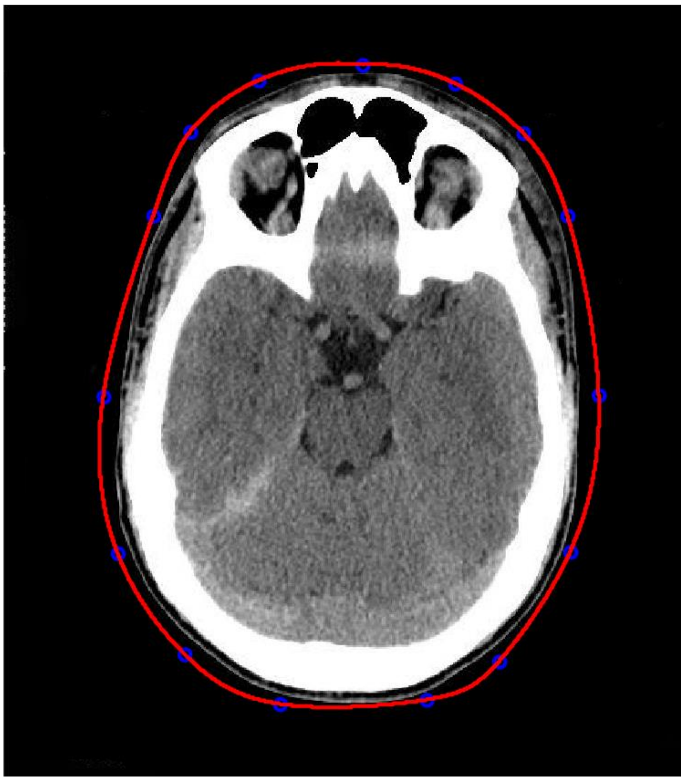
**Number of Iterations:** $N = 150$

**Results:** After 150 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)



**1.5 Brain, Outer Shell of the skull**

**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

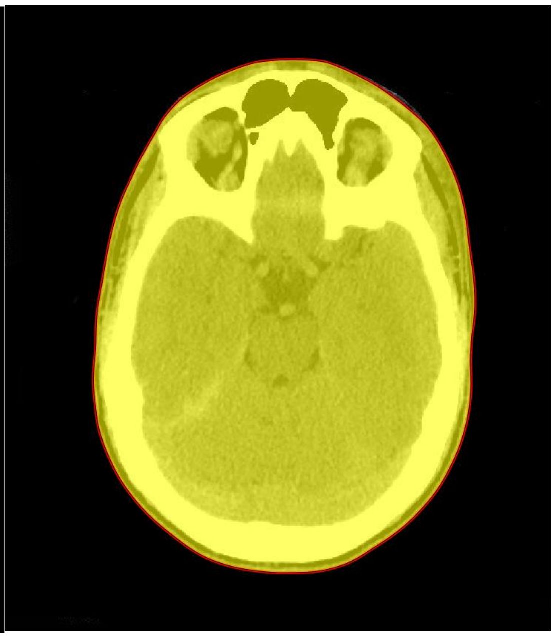


**Parameter Used:**

$\alpha = 20.0, \beta = 20.0, \gamma = 1.0, \kappa = 1.0, Wline = 1.0, Wedge = 1.0, Wterm = 1.0, \sigma = 1.5$

**Number of Iterations:** $N = 200$

**Results:** After 200 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)



**1.6 Brain, Inner contour of the brain matter**

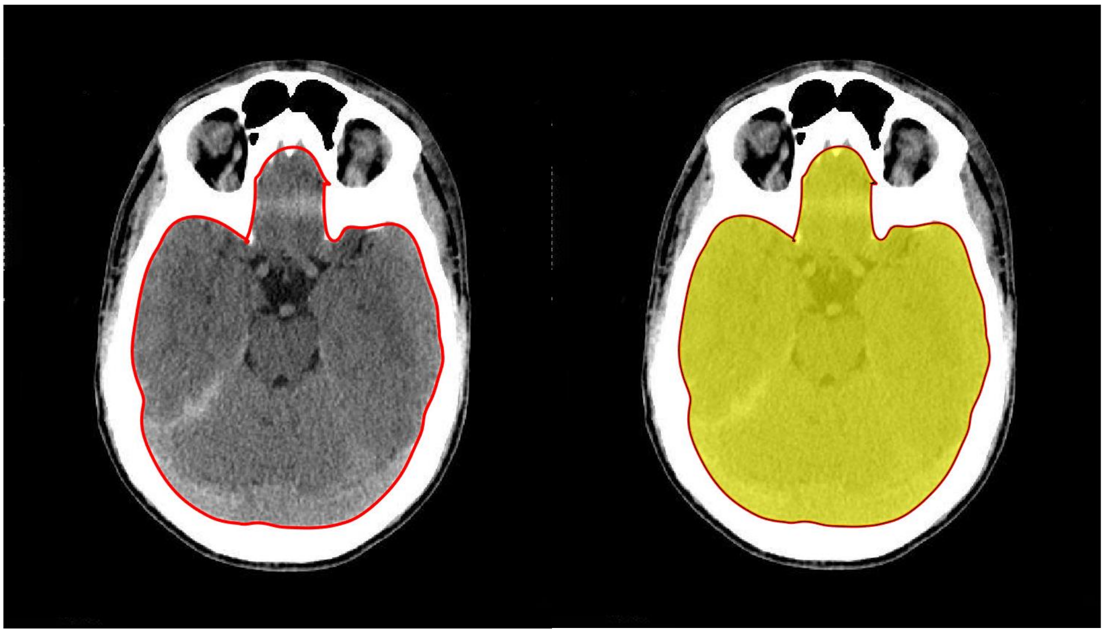
**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

**Parameter Used:**

$\alpha = 1.0, \beta = 20.0, \gamma = 1.0, \kappa = 1.0, Wline = -1.0, Wedge = 8.0, Wterm = 0.0, \sigma = 1.5$

**Number of Iterations:** $N = 200$

**Results:** After 200 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)



**1.7 Brain, the Right eye hole**

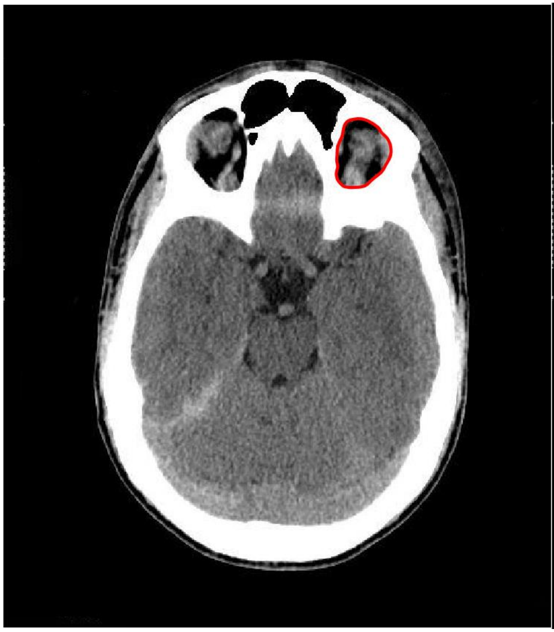
**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

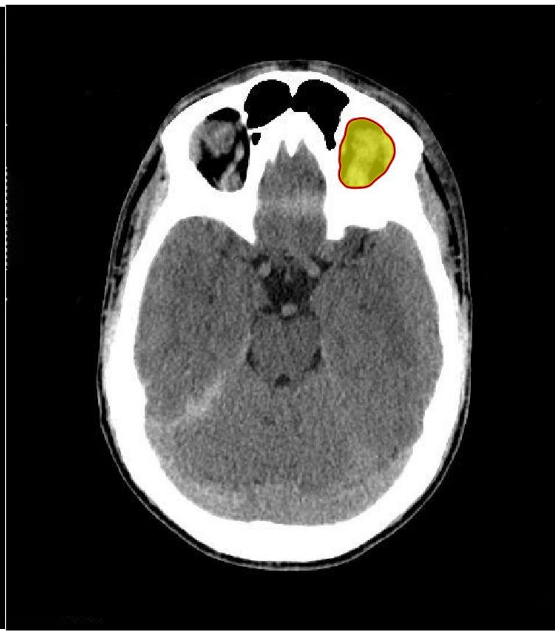
**Parameter Used:**

$\alpha = 1.0, \beta = 1.0, \gamma = 1.0, \kappa = 1.0, Wline = -1.0, Wedge = 1.0, Wterm = 0.0, \sigma = 1.5$

**Number of Iterations:** $N = 100$

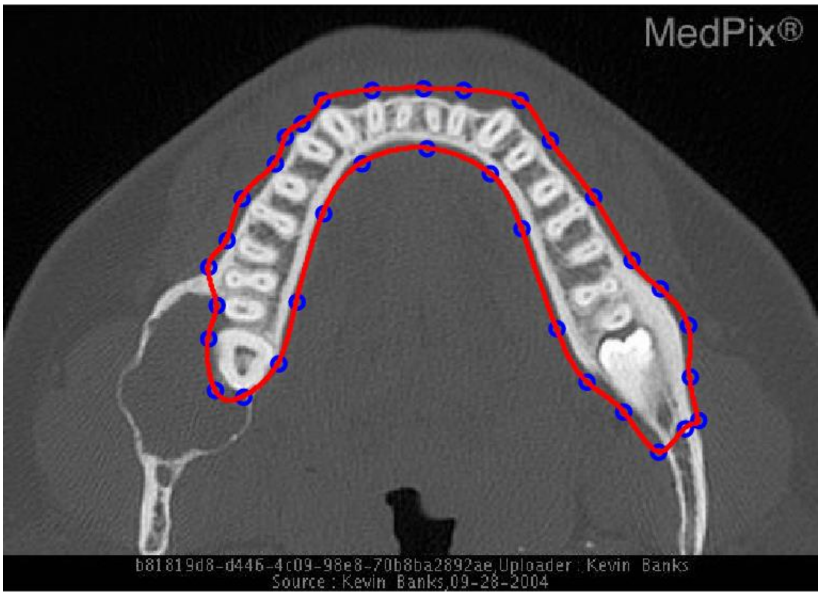
**Results:** After 100 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)





### 1.8 Dental

**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

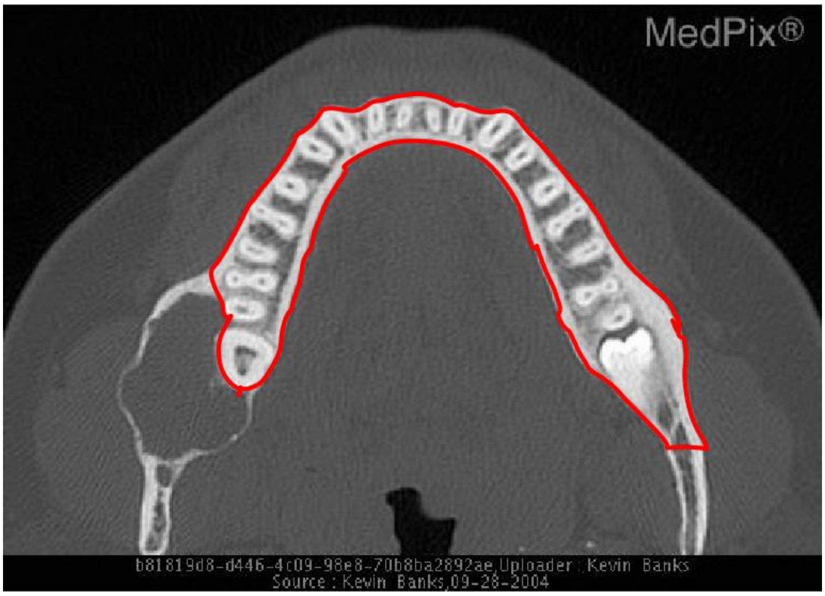


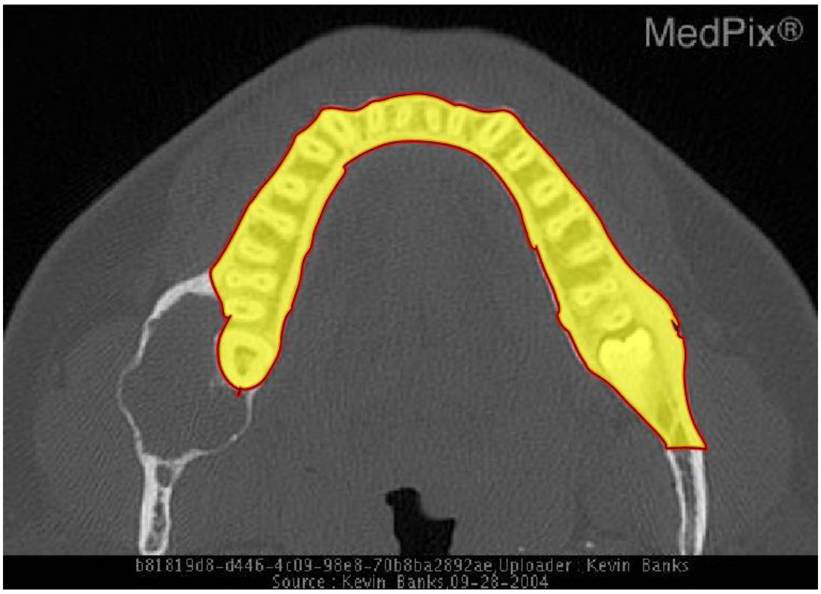

**Parameter Used:**

$\alpha = 0.0, \beta = 1000.0, \gamma = 1.0, \kappa = 1.0, Wline = -20.0, Wedge = 20.0, Wterm = 0.0, \sigma = 2.5$

**Number of Iterations:** $N = 250$

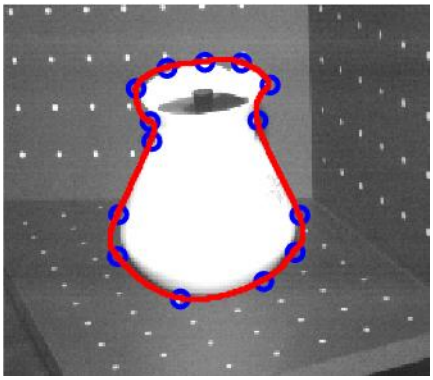
**Results:** After 250 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)




**1.9 Vase**

**The original image with the initialization:** *blue circles* mark the *control points*, and the *red curve* is the *initialized snake* generated by the *spline* function in MATLAB.

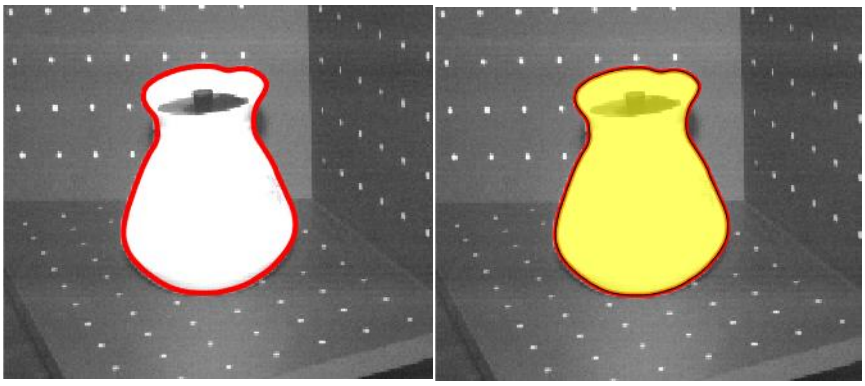


**Parameter Used:**

$\alpha = 1.0, \beta = 1.0, \gamma = 1.0, \kappa = 1.0, Wline = -1.0, Wedge = 1.0, Wterm = 1.0, \sigma = 1.5$

**Number of Iterations:** $N = 50$

**Results:** After 50 iterations, the final result is: (the *red curve* is the *final snake*, and the *semi-opaque yellow region* is the *segmentation* made according to the final snake curve)



**2. Implementation Details:**

**2.1 Initialization:**

**To Collect the Control Points:**

In my implementation there are *two modes*, the *demo mode* and the *manual mode*.

In the **demo mode**, the control points are loaded from the *.mat files* under the folder '*control_points*'. The user doesn't need to specify the control points manually. The codes are in '*initializeSnakeDemo.m*'.

In the **manual mode**, it is up to the user to decide where to place the control points. The codes are in '*initializeSnake.m*'.

In the manual mode, we use the following:

```
[input_x, input_y, input_b]=ginput(1)
```

in a while loop to collect the control points $(input\_x, input\_y)$.

In my implementation, users press '*Q*', '*q*', *ESC* (input_b=27) or *Spacebar* (input_b=32) to finish specifying the control points.

**To Generate Interpolated Curves that Form Closed Loops:**

After obtaining the *control points* $(input\_x, input\_y)$, we use the built-in function '*spline*' in MATLAB to generate the interpolated curves:

```
ctrl_theta=0:2*pi/n_pts:2*pi;
ctrl_pts=[usr_x usr_x(1);usr_y usr_y(1)];
pp=spline(ctrl_theta,ctrl_pts);
yy=ppval(pp,linspace(0,2*pi,step+1));
```

In the above codes, the vector *usr_x* is the collection of *x-coordinates* of the control points, and the vector *usr_y* is the collection of *y-coordinates* of the control points. We add *usr_x(1)* and *usr_y(1)* at the end of each vector *to make sure that the snake forms a closed loop*.

Hence, *pp* is the interpolated curve $(x(\theta), y(\theta))$, and *yy* is its *discretization*.

**To Generate Interpolated Curves with High Accuracy and Fine Spacing:**

In my implementation, I set the variable '*step*' 10 times the sum of the height and the width of the image, which is heuristically accurate enough for the discretization of *pp*.

Then, I use the following codes to generate the final discretization $(x(k), y(k))$ from *pp*:

```
k=1;x(k)=floor(yy(1,1));y(k)=floor(yy(2,1));
for i=1:10*(h+w)
    if floor(yy(1,i+1))~=floor(yy(1,i)) ||
floor(yy(2,i+1))~=floor(yy(2,i))
        k=k+1;x(k)=floor(yy(1,i+1));y(k)=floor(yy(2,i+1));
    end
end
```

This is to make sure that the final discretization of the interpolated curve, $(x(k), y(k))$, is dense enough (since it covers each pixel around the interpolated curve) but not too dense (which is a waste of computational resources).

**To Clamp the Points and Curves:**

To avoid the cases when the control points are outside of the region of the image, in my implementation I use the following codes:

```
usr_x(n_pts)=min(max(input_x,1),size(I,2));
usr_y(n_pts)=min(max(input_y,1),size(I,1));
```

This is to make sure that the *x-coordinate* of the control points designated by users will be within the range of 1 to the width of the image, and the *y-coordinate* of the control points designated by users will be within the range of 1 to the height of the image.

To avoid the cases when some part of the curves should be outside of the region of the image, in my implementation I use the following codes:

```
x=min(max(x,1),w);% w is the width of the image
y=min(max(y,1),h);% h is the height of the image
```

This is to make sure that, in each iteration, the discretization of the interpolated curve, $(x(k), y(k))$ is within the region of the image.

**2.2 External Energies:**

$\boldsymbol{E_{line}}$**:**

We simply set:

```
Eline=I;
```

as long as we have already converted the image to double precision:

```
I=im2double(I);
```

This is to make sure that the program produces the desirable outcome.

$\boldsymbol{E_{edge}}$**:**

Since $E_{edge} = -|\nabla I(x,y)|^2$, we use the following codes to calculate $E_{edge}$:

```
Eedge=imgradient(I); % this is |∇I(x,y)|
Eedge=Eedge.*Eedge;
Eedge=(-1)*Eedge;
```

$E_{term}$:

To calculate $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$, we simply use two convolutional kernels:

```
D_x=[0 0 0; 0 -1 1; 0 0 0];
D_y=[0 0 0; 0 -1 0; 0 1 0];
```

and compute the convolutions $D\_x * I$ and $D\_y * I$ to obtain $C_x = \frac{\partial I}{\partial x}$ and $C_y = \frac{\partial I}{\partial y}$:

```
C_x=conv2(I,D_x,'same');
C_y=conv2(I,D_y,'same');
```

Similarly, we can obtain $C_{xx}, C_{xy}, C_{yy}$ by calculating the convolutions:

```
C_xx=conv2(C_x,D_x,'same');
C_xy=conv2(C_x,D_y,'same');
C_yy=conv2(C_y,D_y,'same');
```

To compute $E_{term} = \frac{C_{yy}C_x^2 - 2C_{xy}C_xC_y + C_{xx}C_y^2}{(1+C_x^2+C_y^2)^{\frac{3}{2}}}$, we just need to remember to *use the element-wise operators*: '**.***', '**./**', '**.^**':

```
Eterm=(C_yy.*C_x.*C_x-
2*C_xy.*C_x.*C_y+C_xx.*C_y.*C_y)./((C_x.*C_x+C_y.*C_y+1).^1.5);
```

Hence, we complete the computation of $E_{term}$.

### 2.3 Internal Energies and Iteration:

**Calculate $\frac{\partial E_{ext}}{\partial x}$ and $\frac{\partial E_{ext}}{\partial y}$:**

This is similar to the computation of $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$. We just need to use convolutional kernels $D\_x$ and $D\_y$ as mentioned above and compute the convolutions $D\_x * E_{ext}$ and $D\_y * E_{ext}$.

```
f_x=conv2(Eext,D_x,'same');
f_y=conv2(Eext,D_y,'same');
```

**Use bilinear interpolation to calculate $f_x$ and $f_y$:**

Since $x$ and $y$ are float values, and $\frac{\partial E_{ext}}{\partial x}$ and $\frac{\partial E_{ext}}{\partial y}$ are only defined at integer pixel values, we will need to perform bilinear interpolation to obtain the exact value of $f_x$ and $f_y$ at the point of $(x, y)$.

For bilinear interpolation, we have:

$$f(x,y) = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(x_1,y_1) & f(x_1,y_2) \\ f(x_2,y_1) & f(x_2,y_2) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}$$

So, when both $x$ and $y$ are not integer, we can obtain $f_x$ and $f_y$ by bilinear interpolation with the following codes:

```
int_x=floor(x(i));
int_y=floor(y(i));
```

```
fx(i)=[int_x+1-x(i) x(i)-int_x]*[f_x(int_y,int_x) f_x(int_y+1,int_x);
f_x(int_y,int_x+1) f_x(int_y+1,int_x+1)]*[int_y+1-y(i); y(i)-int_y];

fy(i)=[int_x+1-x(i) x(i)-int_x]*[f_y(int_y,int_x) f_y(int_y+1,int_x);
f_y(int_y,int_x+1) f_y(int_y+1,int_x+1)]*[int_y+1-y(i); y(i)-int_y];
```

We might want to separately take care of the cases when either $x$ or $y$ is an integer. This is because $floor(x)+1$ *might be out of the region of the image* if the integer *x* is happened to equal to the width of the image, or $floor(y)+1$ *might be out of the region of the image* if the integer *y* is happened to equal to the height of the image.

**Iterations:**

This is simply to implement the equations:

$$x_t = (A+\gamma I)^{-1}(\gamma x_{t-1} + \kappa f_x(x_{t-1}, y_{t-1}))$$
$$y_t = (A+\gamma I)^{-1}(\gamma y_{t-1} + \kappa f_y(x_{t-1}, y_{t-1}))$$

And this could be achieved by the following codes:

```
newX=(Ainv*(gamma*x'+kappa*fx))';
newY=(Ainv*(gamma*y'+kappa*fy))';
```

**Clamp to Image Size:**

Similar to previous discussions, we use the following codes to make sure that the points after iterations are still within the region of the image:

```
newX=min(max(newX,1),w);
newY=min(max(newY,1),h);
```

**2.4 Bonus: Internal Energy Matrix**

In order to find $[Ainv] = (A+\gamma I)^{-1}$, the core task is to find *A*. According to the tutorial, $A$ has the following form:

$$\begin{bmatrix} 2\alpha+6\beta & -\alpha-4\beta & \beta & 0 & \cdots & \cdots & 0 & \beta & -\alpha-4\beta \\ -\alpha-4\beta & 2\alpha+6\beta & -\alpha-4\beta & \beta & 0 & \cdots & \cdots & 0 & \beta \\ \beta & -\alpha-4\beta & 2\alpha+6\beta & -\alpha-4\beta & \beta & 0 & \cdots & \cdots & 0 \\ 0 & \beta & -\alpha-4\beta & 2\alpha+6\beta & -\alpha-4\beta & \beta & 0 & \cdots & 0 \\ & & & & & & & & \\ & & & & & \cdots & & & \\ & & & & & & & & \\ 0 & \cdots & \cdots & 0 & \beta & -\alpha-4\beta & 2\alpha+6\beta & -\alpha-4\beta & \beta \\ \beta & 0 & \cdots & \cdots & 0 & \beta & -\alpha-4\beta & 2\alpha+6\beta & -\alpha-4\beta \\ -\alpha-4\beta & \beta & 0 & \cdots & \cdots & 0 & \beta & -\alpha-4\beta & 2\alpha+6\beta \end{bmatrix}$$

So $A(i,i) = 2\alpha+6\beta (1 \le i \le n)$, $A(i,i+1) = -\alpha-4\beta\ (1 \le i \le n-1)$, $A(i,i-1) = -\alpha-4\beta\ (2 \le i \le n)$, $A(n,1) = A(1,n) = -\alpha-4\beta$, $A(i,i+2) = \beta\ (1 \le i \le n-2)$, $A(i-2,i) = \beta (3 \le i \le n)$, $A(n-1,1) = A(n,2) = A(1,n-1) = A(2,n) = \beta$, and all other elements of $A$ are zeros.

This can be simplified as the following:

$$A(i,j)=\begin{cases}2\alpha+6\beta & i=j\\ -\alpha-4\beta & j=mod(i,n)+1, or\ i=mod(j,n)+1\\ \beta & j=mod(mod(i,n)+1,n)+1, or\ i=mod(mod(j,n)+1,n)+1\end{cases}$$

According to the above discussions, we are able to initialize $A$ as a sparse matrix and compute [Ainv] by the following codes:

```
k=1;
for l=1:nPoints
    i(k)=l;j(k)=l;v(k)=2*alpha+6*beta;k=k+1;
    i(k)=l;j(k)=mod(l,nPoints)+1;v(k)=-alpha-4*beta;k=k+1;
    i(k)=l;j(k)=mod(mod(l,nPoints)+1,nPoints)+1;v(k)=beta;k=k+1;
    i(k)=mod(l,nPoints)+1;j(k)=l;v(k)=-alpha-4*beta;k=k+1;
    i(k)=mod(mod(l,nPoints)+1,nPoints)+1;j(k)=l;v(k)=beta;k=k+1;
end
A=sparse(i,j,v);
Ainv=inv(A+gamma*eye(nPoints));
```

# Technical Report for CMPT 732, Assignment 1

# Image Reconstruction

Sihui Wang 301474102

**1. Main Results:**

Similar to Ground Truth

Globally Brighter

Brighter on Left Side

Brighter on Bottom Side

Brighter on Right Bottom Corner

**2.Explanations:**

**Determine the 4 additional constraints:**

To obtain a reconstructed image that is similar to the ground truth, we just need to add 4 additional constraints as the following:

$$v_{(1,1)} = s_{(1,1)}$$
$$v_{(1,n)} = s_{(1,n)}$$
$$v_{(m,1)} = s_{(m,1)}$$
$$v_{(m,n)} = s_{(m,n)}$$

To obtain a reconstructed image that is globally brighter than the ground truth, we just need to add 4 additional constraints as the following:

$$v_{(1,1)} = 1$$
$$v_{(1,n)} = 1$$
$$v_{(m,1)} = 1$$
$$v_{(m,n)} = 1$$

To obtain a reconstructed image that is brighter on the left side, we just need to add 4 additional constraints as the following:

$$v_{(1,1)} = 1$$
$$v_{(1,n)} = s_{(1,n)}$$
$$v_{(m,1)} = 1$$
$$v_{(m,n)} = s_{(m,n)}$$

To obtain a reconstructed image that is brighter on the bottom side, we just need to add 4 additional constraints as the following:

$$v_{(1,1)} = s_{(1,1)}$$
$$v_{(1,n)} = s_{(1,n)}$$
$$v_{(m,1)} = 1$$
$$v_{(m,n)} = 1$$

To obtain a reconstructed image that is brighter on the right bottom corner, we just need to add 4 additional constraints as the following:

$$v_{(1,1)} = s_{(1,1)}$$
$$v_{(1,n)} = s_{(1,n)}$$
$$v_{(m,1)} = s_{(m,1)}$$
$$v_{(m,n)} = 1$$

# Technical Report for CMPT 732, Assignment 1

# Poisson Blending

Sihui Wang 301474102

**1. Main Results:**

**Example 1:**


**Source**

**Target**

**Cloning**

**Blending**

**Example 2:**

**Source** **Target**

**Cloning** **Blending**

**Example 3:**


Source


Target

Cloning

Blending

**2. Implementations:**

In my implementation, you can use '*main.m*' to see the results of blending three pictures together.

You can use '*demo.m*' to blend *n* pictures together.

Blending RGB images and grayscale images together is enabled in my implementation. Please check with '*output2.png*', '*output3.png*', and '*output4.png*' in the '*PoissonBlending*' folder.