Junhua Chen
Image Processing
2018年3月1日

Assignment 3

1. Implement the Greedy Algorithm

a) read an image:

This is very easy to implement, we can choose the image automatically or hardcode the location of the image, and here I just use uigetfile function.

```
%choose the image
[filename, pathname] = uigetfile({'*.jpg'; '*.bmp'; '*.gif';}, 'please choose the image');
%if no images are picked, then the function returns
if filename == 0
    return;
end
```

b) compute the smoothed gradient of the image and find the gradient magnitude at each pixel:

Since we already did it in last 2 assignments, here I just use the built-in function of Matlab to do the smoothing.

```
end
%smooth the image
sigma = 3;
gausFilter = fspecial('gaussian', [3,3], sigma);
img = imfilter(imgsrc, gausFilter, 'replicate');
img = double(img);
figure(1), imshow(imgsrc);
```

c) obtain the position from user

The material of how to use function “getline” is limited, so I searched several related functions and tried to figure out how to use them. In this section, I tried method “getline” and “ginput”, the screenshot below shows how to use “ginput” to get the contour from user.

```
%not, we should add extra points
user_x=[]; user_y=[]; user_count=1; count_max=100; distance_max=5; avg_distance=0;
while user_count<count_max
    [user_xi, user_yi, button] = ginput(1);
    user_xi = round(user_xi);
    user_yi = round(user_yi);
    %make sure the diatance between 2 points is smaller than 5 pixels
```

also, I round the position value to integers.

d) about the distance

Now we should make sure the distance between each point is around 5 pixels, so we need to add extra points if the distance is too big. My way is, if the distance is X, I will put X/5-1 points betweenez that 2 points. For example, if the distance between 2 points is 14 pixels, then I will put 2 extra points in between, and make sure the distance between these points are the same.

```
%make sure the diatance between 2 points is smaller than 5 pixels
%if the distance greater than 5 pixels, add several points in between,
%for instance, a distance of 20 pixels means we need to add 3 points
if(user_count>=2)
    temp_distance = (double((user_xi-user_x(user_count-1))^2+(user_yi-user_y(user_count-1))))^0.5;
    temp_distance = round(temp_distance);
    temp_count = round(temp_distance/5);
    if(temp_count>=2)
        temp_x = (user_xi-user_x(user_count-1))/temp_count;
        temp_y = (user_yi-user_y(user_count-1))/temp_count;
        while(temp_count>=2)
            user_x = [user_x user_x(user_count-1)+temp_x];
            user_y = [user_y user_y(user_count-1)+temp_y];
            temp_count = temp_count-1;
            user_count = user_count+1;
        end
    end
end
```

e) implement the greedy algorithm

Basically, the algorithm is:

1) compute the average distance between points along the snake

2) for each point p:

define the neighborhood

compute the energy E

move p to point with the minimum energy

update the position

and the nerdy is given by the following terms:

$E = alpha * Econt + beta * Ecurv + gamma * Eedge$

where:

Econt is analogous to the first-derivative term in the Kass-Witkin-Terzopoulos formulation, but should seek to keep points spaced equally along the curve. Thus,

Econt = square(d_avg - distance(p[i], p[i-1]))

where d_avg is the average distance between points in the curve.

Ecurv is analogous to a second-derivative term, which seeks to minimize curvature. The finite-difference approximation to the second derivative is

*Ecurv = square(magnitude(p[i-1] - 2*p[i] + p[i+1]))*

Eedge is a term that tries to draw the snake towards edges, based on gradient magnitude. Thus,

Eedge = - magnitude(gradient(I(x,y)))

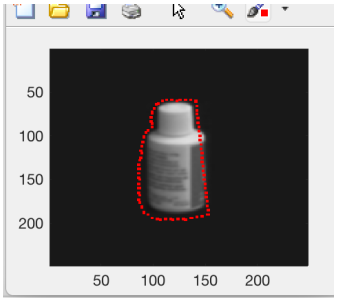
```
avg_distance=0;
for i=2:(user_count-1)
    avg_diatance = avg_distance+((user_x(i)-user_x(i-1))^2+(user_y(i)-user_y(i-1))^2)^0.5;
end
avg_distance = avg_distance/(user_count-1);
for i=2:(user_count-2)
    E_min = 100000;
    j_x = user_x(i);
    j_y = user_y(i);
    for j=1:neighboursize
        current_x = user_x(i)+j-2;
        current_y = user_y(i)+j-2;
        Econt = square(avg_distance - ((user_x(i-1)-current_x)^2+(user_y(i-1)-current_y)^2)^0.5);
        Ecurv = square((user_x(i-1)-2*current_x+user_x(i+1))^2+(user_y(i-1)-2*current_y+user_y(i+1))^2);
        Eedge = -1*sqrt(m(round(current_x), round(current_y)));
        Ej = alpha*Econt + beta*Ecurv + gamma*Eedge;
        if(Ej<E_min)
            E_min = Ej;
            j_x = current_x;
            j_y = current_y;
        end
    end
    if(j_x~=user_x(i)||j_y~=user_y(i))
        move_points=move_points+1;
    end
    user_x(i)=j_x;
    user_y(i)=j_y;
end
```

2. Experiments, evaluation and comments:

a)

standard deviation: 1.2
neighborhood size: 3*3
alpha, beta, gamma=1
fraction=0.1

Image1(the first image is the original contour, the second and the third are the intermediate steps, and the last one is the final contour):



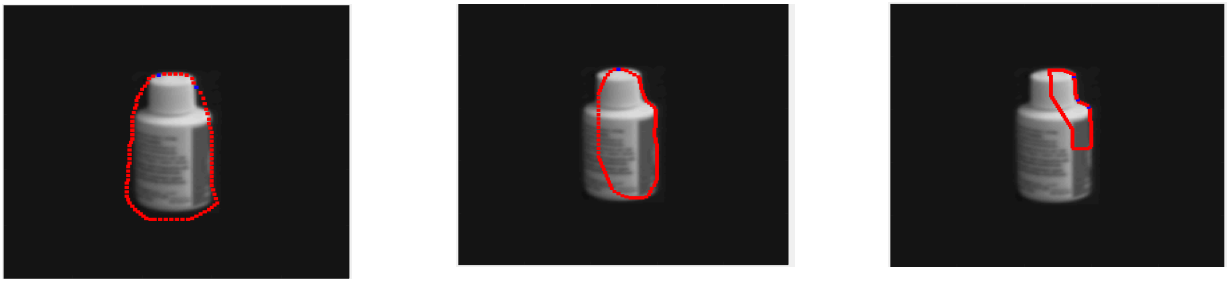


Image 2:

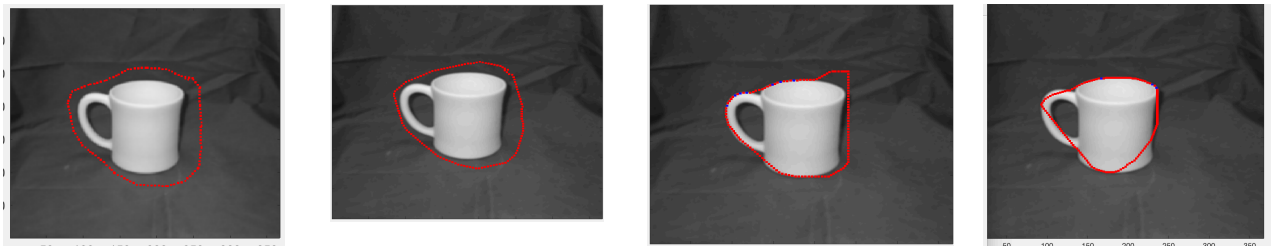


Image 3:

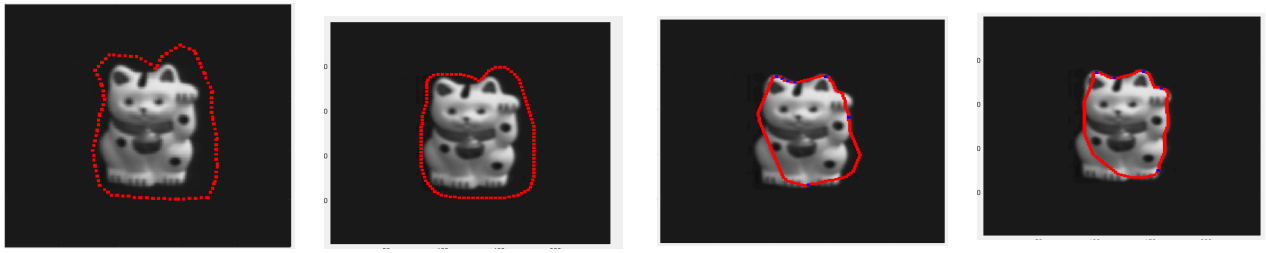


Image 4:

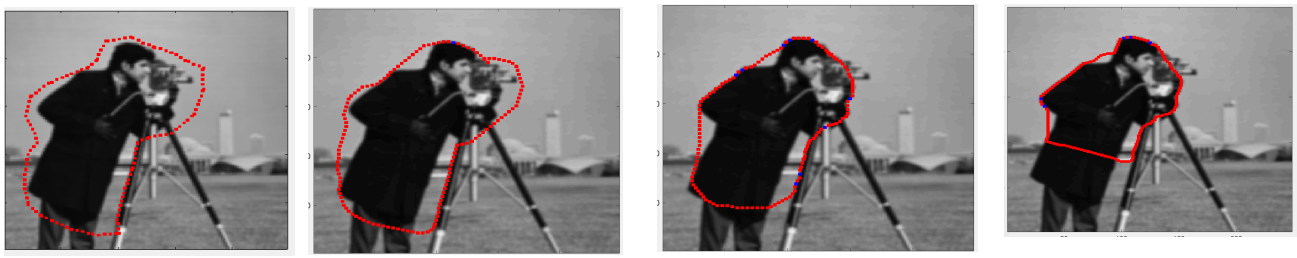


Image 5:

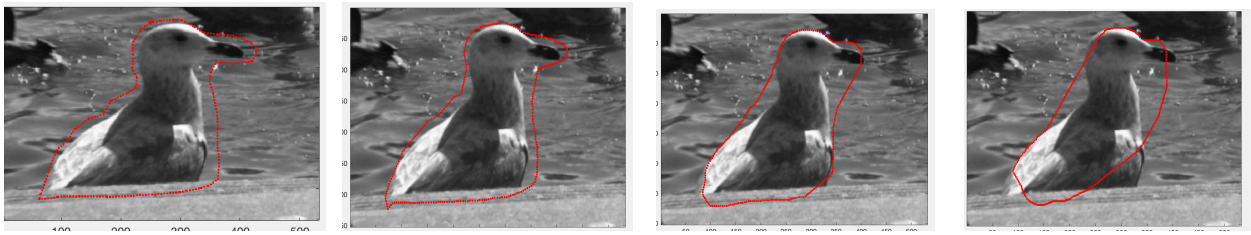


Image 6:

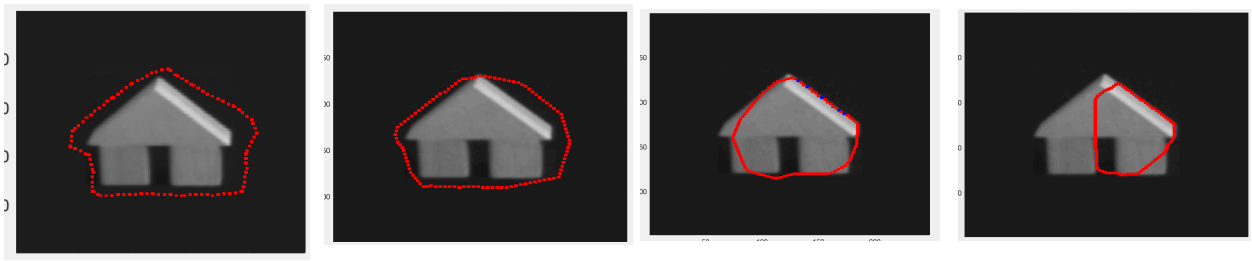


Image 7:

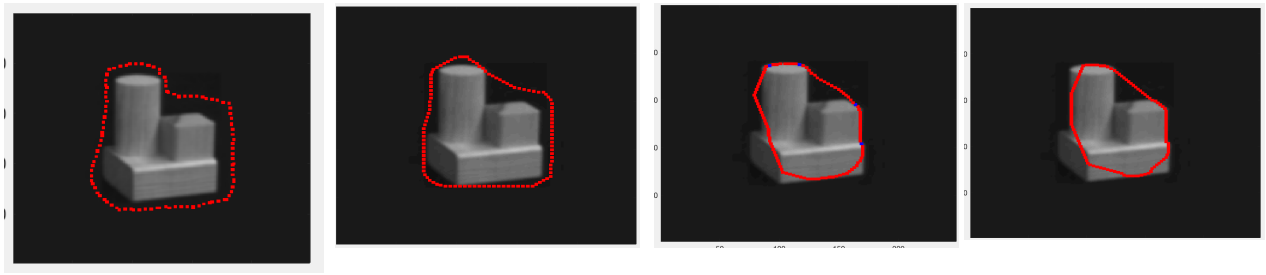
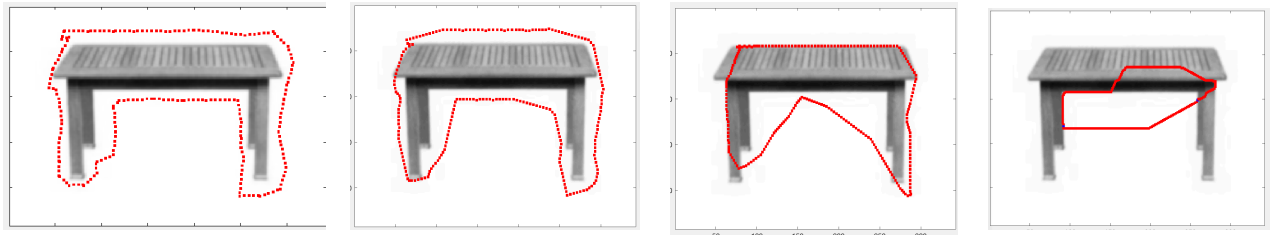


Image 8:



Some results are far from satisfactory, that may related to the improper arguments we set, or the weakness of my code and the algorithm.
Next I will try to change the arguments and do the test again.

b)

first change the width to 15(first line), and then to 3(second line), keep other arguments

Image 1:

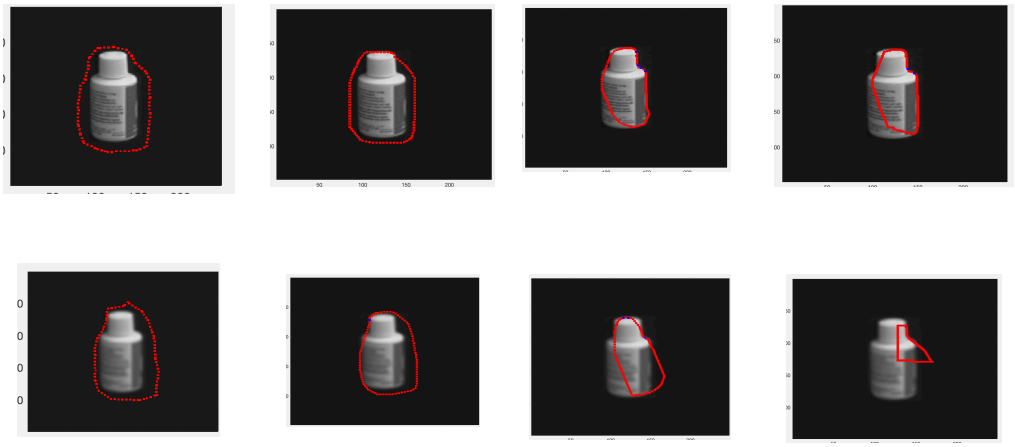


Image 2:

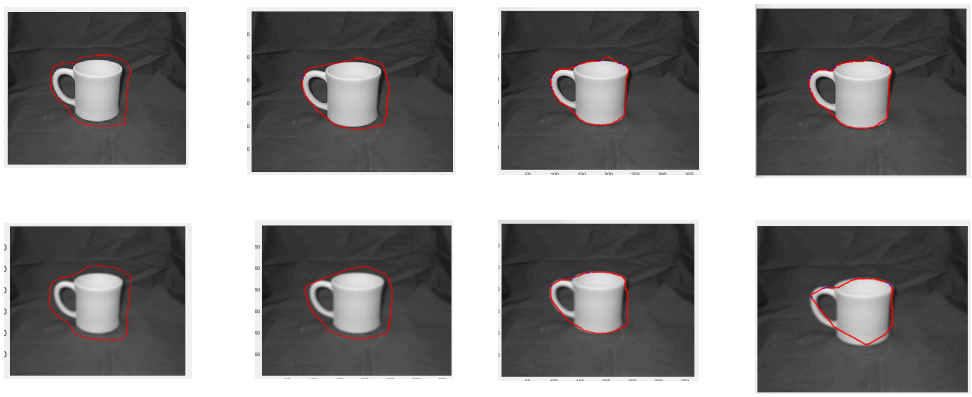


Image 3:

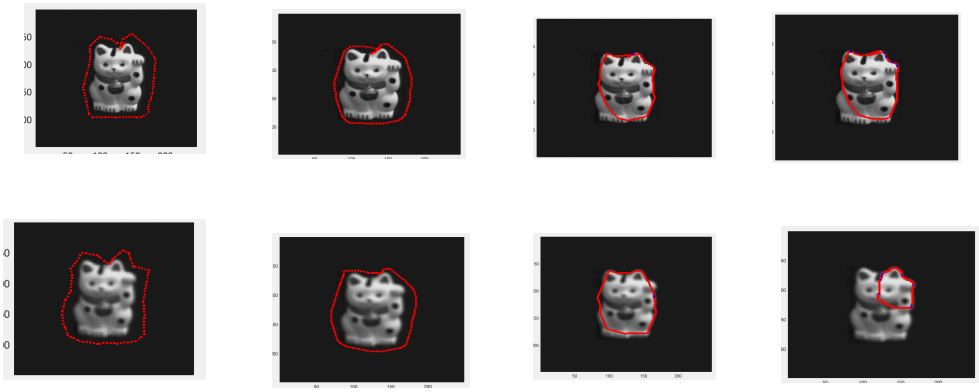
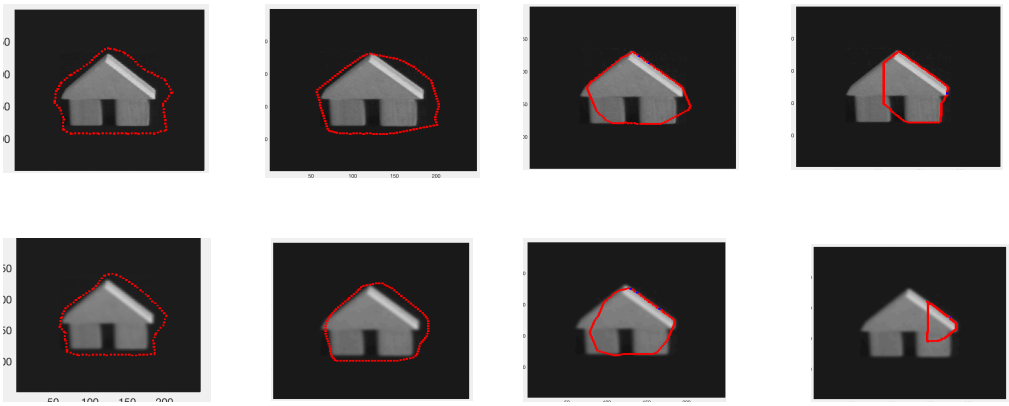


Image 6:



As we can see from the results, larger standard deviation leads to better results of active contour. Since increasing the width of Gaussian makes the image more smoother, that will make the contour more obvious, but also we can see, when we make the width too large, that will make the contour more smoother, as a matter of fact, the image will have no "edges" at all, so the results will become bad.

c) we change the neighborhood size to 5*5(first line) and then 9*9(second line), keep other arguments.

Image 1:

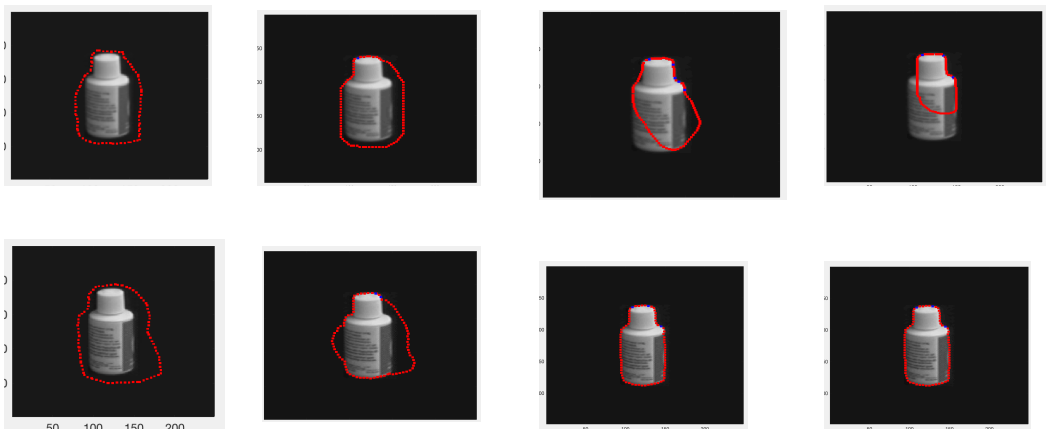
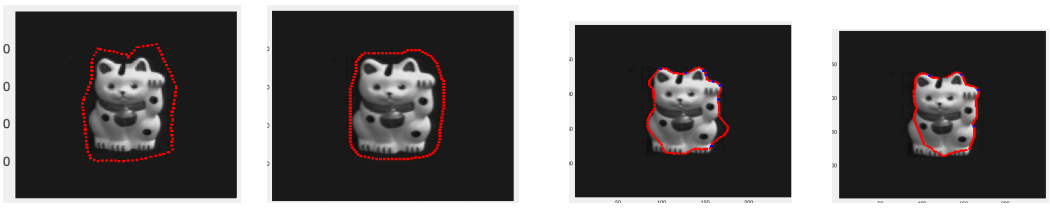


Image 3:



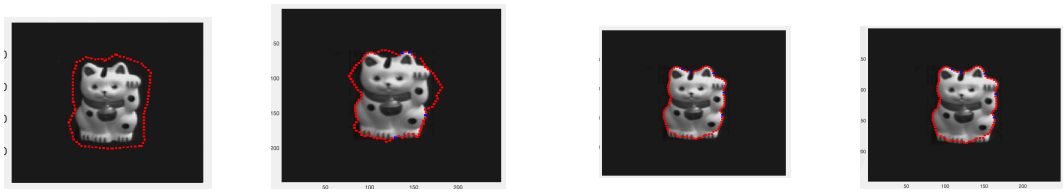
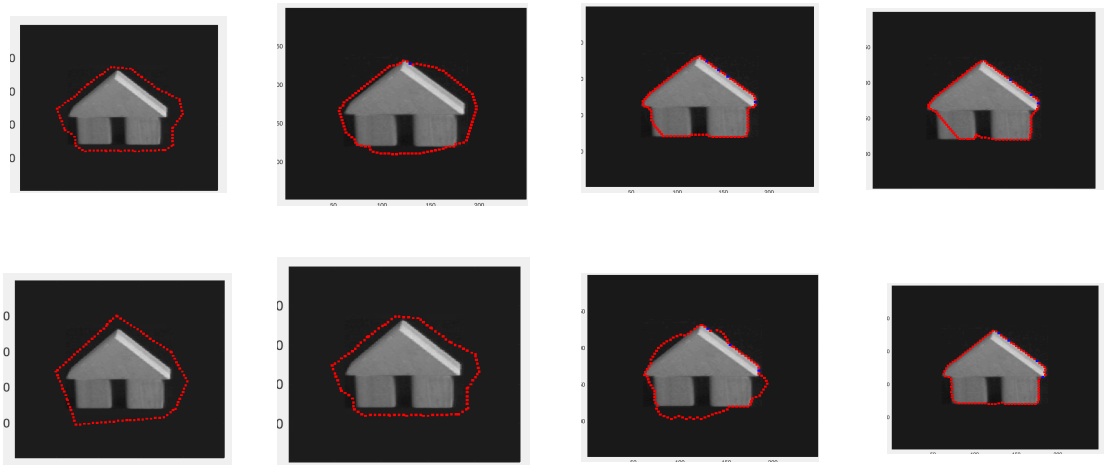


Image 6:



As you can see, bigger neighborhood size leads to better results. But too-large neighbor size will cause more time consuming.

d) first set alpha to 0, then set alpha to 8

Image 1:

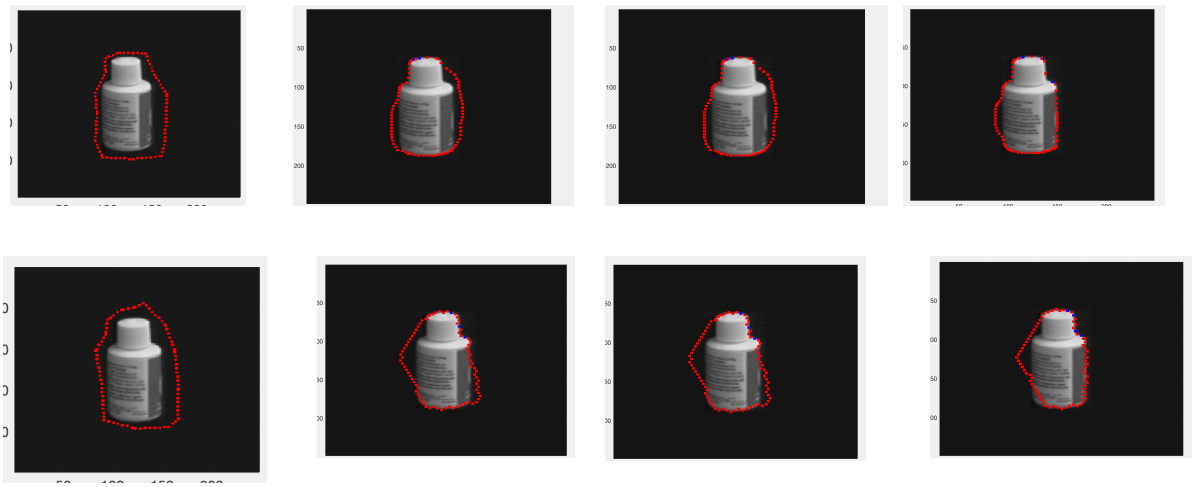


Image 3:

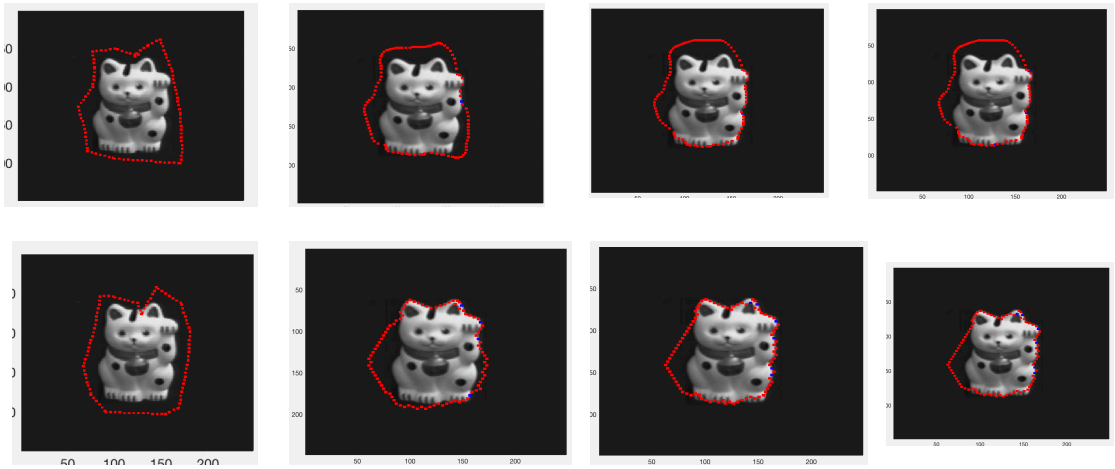
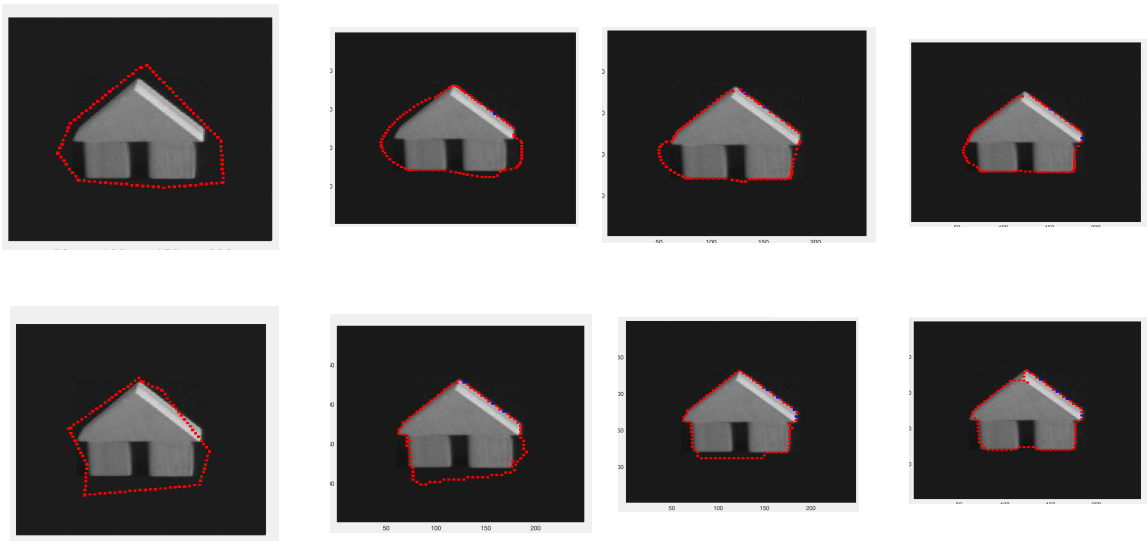


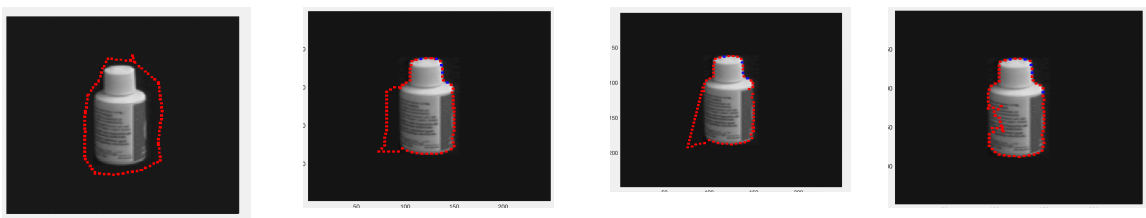
Image 6:



The main functions of alpha are to make the distances closer to the average in neighborhood and reduce the average distance, so when we set a big a, we will get a small contour, vice versa.

e) set beta to 0, and then set beta to 10

Image 1:



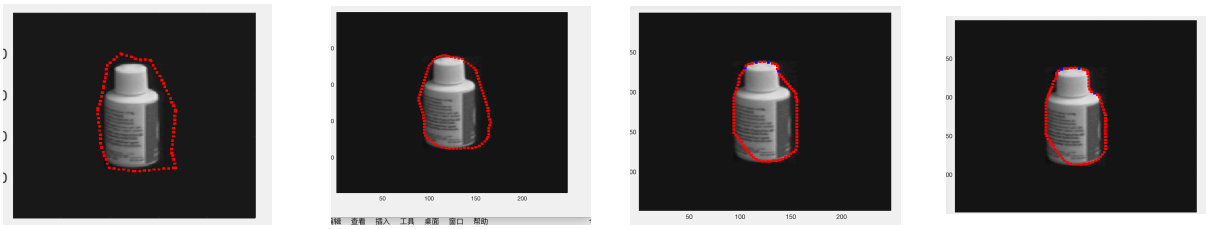
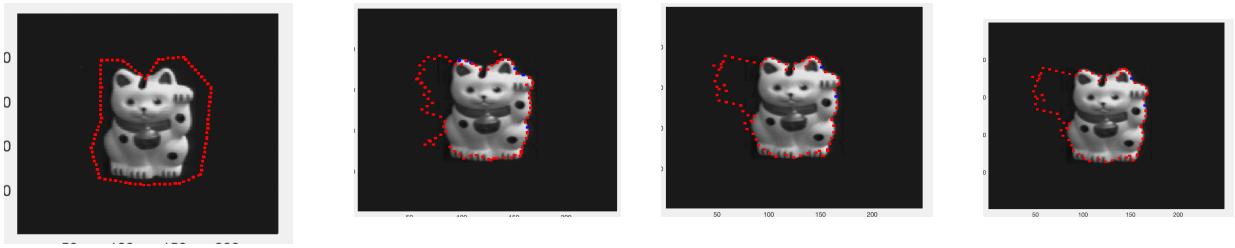


Image 3:



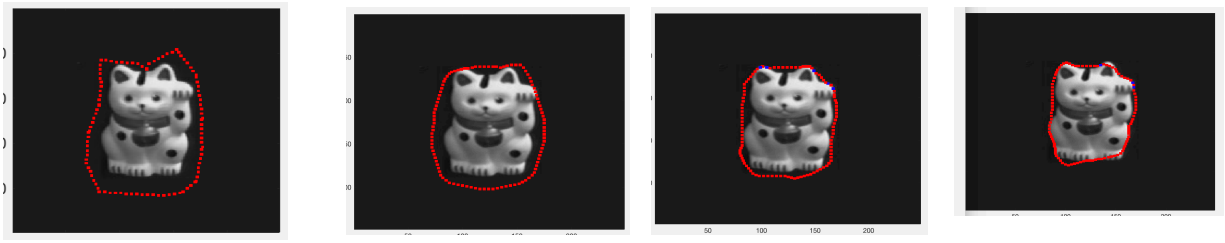
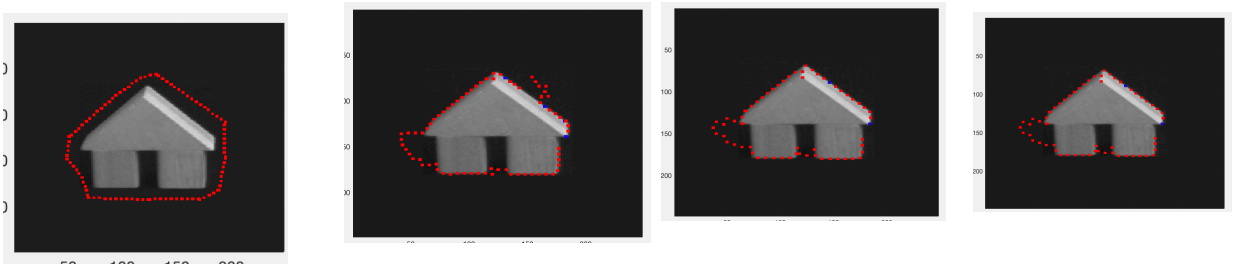
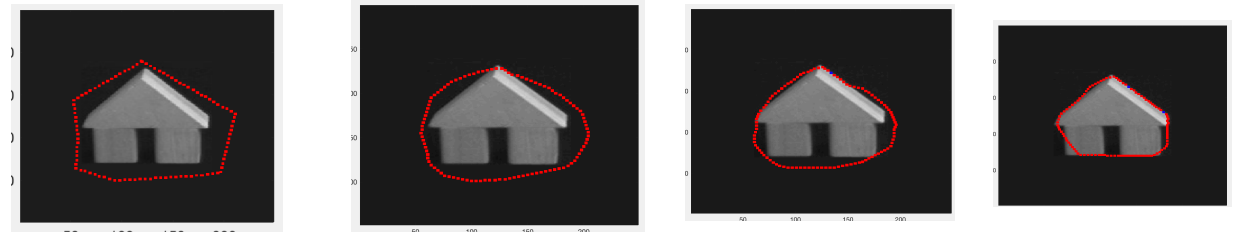


Image 6:





As you can see, bigger beta makes a smoother contour, which corresponds to the function of beta.

f) change gamma to 0 and the change it to 10:

Image 1:

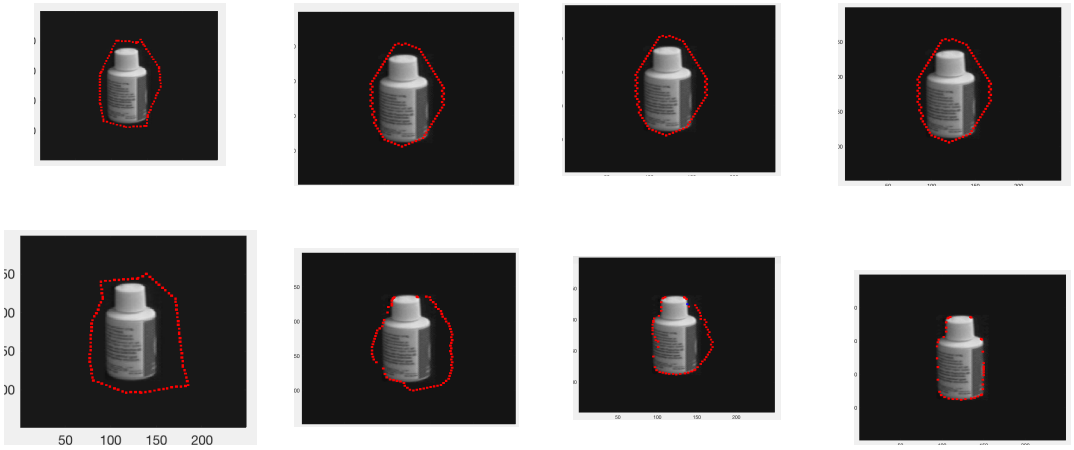


Image 3:

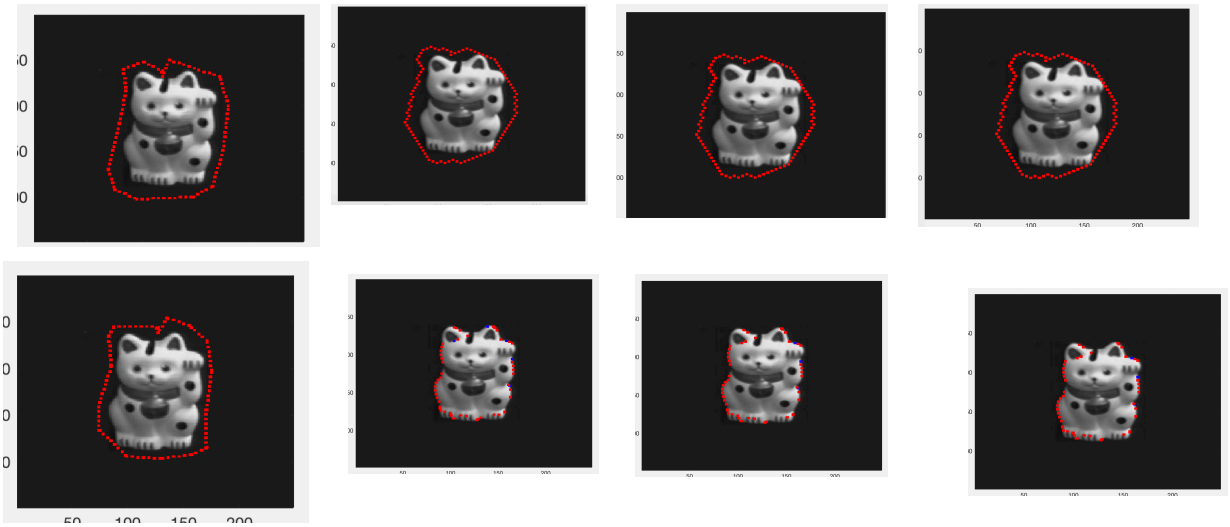
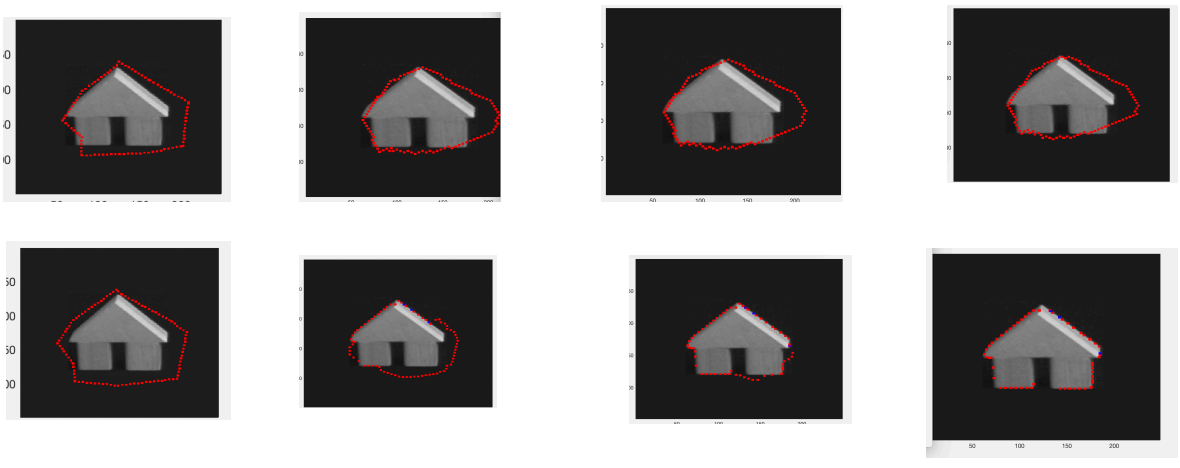


Image 6:



The main function of gamma is to make the contour points moving closer to the edges. If gamma is very large, the contour may not be continuous and obvious , if the value is very small, the contour may goes to somewhere far away from the real contour.

g) First we set fraction to 0.8, keep other arguments and then change it to 0.15

Image 1:

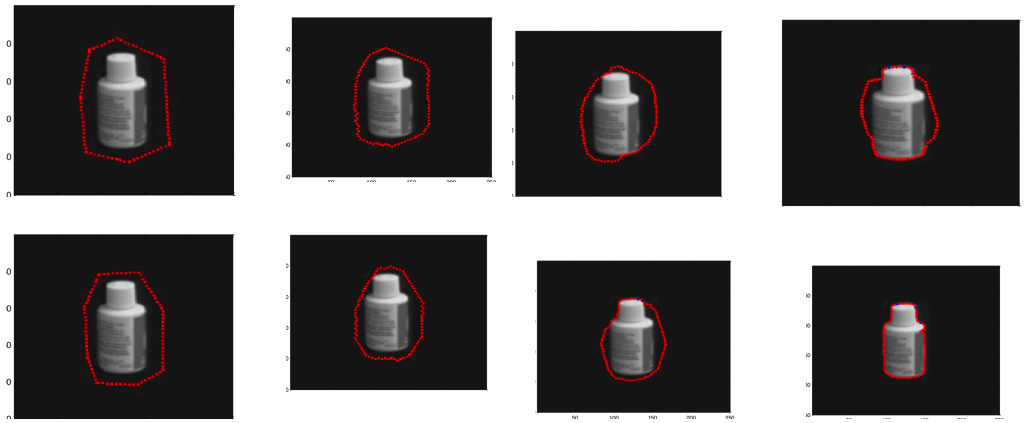


Image 3:

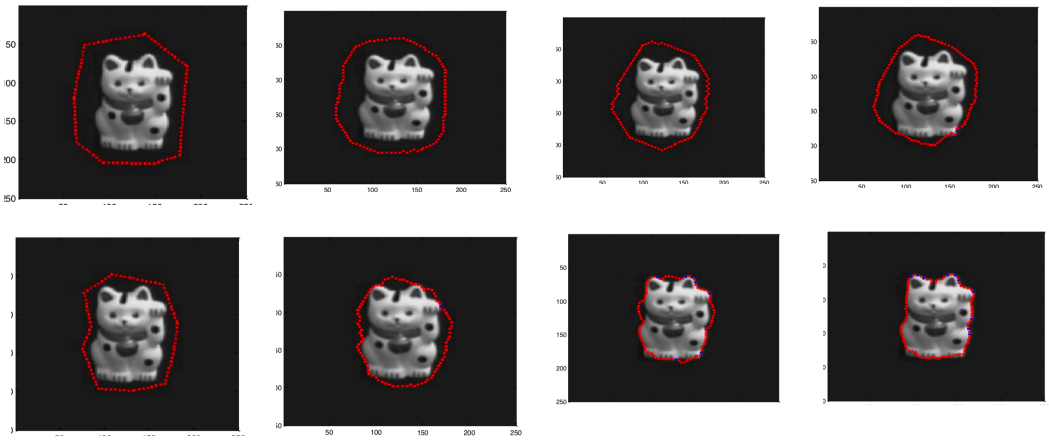
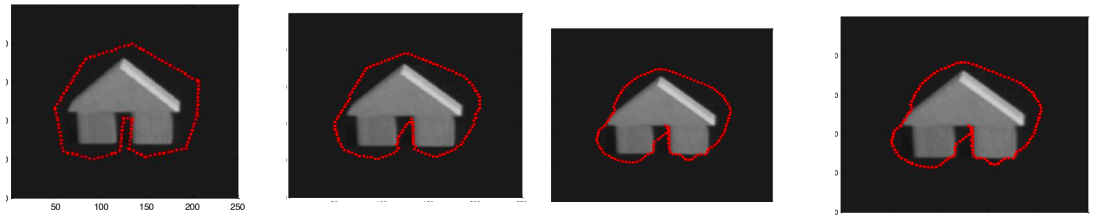
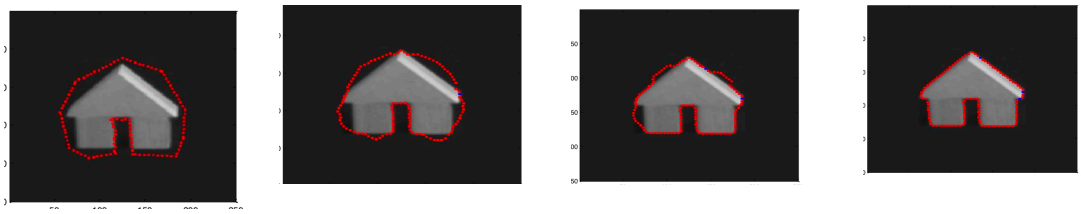


Image 6:





As you can see, small fraction makes the contour more obvious and smooth. But not too small, that may greatly increase the process time.

3)
Sequence1:

Sequence 2: