CSE 691: Image and Video Processing

Spring 2020 Assignment 4

Active Contour

Lichen Liang

03/06/2020

Objectives

- Understand the Greedy Snake Algorithm
- Understand each step of the algorithm and what does it do.
- Know how the variables affect the output
- Apply active contour to an image

Method

To run the code, run ***main.m*** and make sure all function files are in same folder.

In the pop up window, click the points in order around the object to form a circle, then press Enter.

The output image for is labeled as :[D]out[image number][variable changed and its value]

For example, Dout3 represents image3, default parameters set in requirement.

1. First, filter the image with Gaussian filter and show the image for user to click points. The user should click points in order to form a contour around the object. These points are recorded in a list called *PointsClicked*. The distance between the points can be too far, so call the function *CreatePoints()* to create more points in between.

2. The *CreatePoints()* function calculates the distance between each consecutive points. If the distance is greater than 5, then we need to add points in between. The number of points that need to be added are calculated by floor(distance/5). For example, if the distance is 21, then we need floor(21/5) = 4 points in between. The coordinates for initial two points and points created will be stored in a list *newL*. *newL* is floored to make sure all coordinate numbers are integers. Later this list *newL* is added to the *PointsList* for processing.
   Note that the last entry of the *PointsList* is a copy of the first entry. For example, if actual points clicked are [ABCDE] then the *PointsList* will be [ABCDEA] for the ease of processing.

3. The Greedy() function is then called. At the beginning I initialized variables α, β, γ to 1 and get the gradient of the image. The *betachange* list is initially empty. The variable iterations determines how many times the loop should run. In each loop, all the points in the *PointsList* are processed. At the beginning of each iteration, an average distance of all the points are calculated to be used later, minimum energy Emin is reset to a large number, and number of points moved and inner iteration counter are reset to 0.

   The points moved in each iteration should be higher than the fraction. The fraction is 10% of total number of points. However, if inner iteration counter is higher than fraction, force break out of this loop and start next iteration. For example, if there are 9 points moved but the minimum to be moved is 10, then I cannot have an 11_{th} or more inner iterations because no points will be able to move.

Before calculating the energy, check if it is not the first iteration and the *betachange* list is not empty. If both are true, then we have to check whether the current points processed is in the *betachange* list, if so, the beta is set to 0 for this iteration this point (refer to steps 9 and 10).

4. We also need to get the neighborhood for this point. The *neighborhood()* function is called and a list of neighborhood values is returned depending on the size required by the user. The function is hard coded and only limit to 3x3 or 5x5 neighborhoods.

5. The *Econtinuity()* function calculates the 1st energy component. The *PointsList*, the point's index, neighborhood, size of the *PointsList*, and average distance are passed into the function. The max distance *dmax* is initially set to 0. We also have to find the previous point *Viprev*. When our current point is the first in the list, then its previous point is size of PointsList-1 ([ABCDEA], A's previous is E). For each point in the neighborhood, use the formula to calculate the energy.

$$Econt = \frac{Daverage - |Vi - Viprev|}{Dmax}$$

Vi is one of the points in the neighborhood. Vi – Viprev is the distance between the two points. This distance is stored in a temporary list, and maximum distance *Dmax* is determined while looping through the neighborhood points. Finally, divide by *Dmax* to normalize all the values to be between [0,1]. Store all the energy values in a list *Econt*. This list size is determined by the neighborhood size. i.e. 3x3 neighborhood size will have 9 points in both neighborhood list and *Econt* list. Same of the next two energy calculations.

6. The *Ecurvature()* functions calculates the 2nd energy component. The *PointsList*, the point's index, neighborhood, and size of the *PointsList* are passed into the function. The max curvature *cmax* is initially set to 0. Similar to continuity, we need the previous point *Viprev* using the same method as above. We also need the next point *Vinext*, calculated by index+1. There is no need to worry about boundary issues since the last point's next is the first point and we have a copy of the first point in our list (E is the last point and E's next in [ABCDEA] is A). For each point in the neighborhood, use the formula to calculate the energy.

$$Ecurv = \frac{||Viprev - 2Vi + Vinext||^2}{cmax}$$

Similar to above, all the calculations are stored in a temporary list then normalized and stored in the final list *Ecurv*.

7. The *Egradient()* functions calculates the 3rd energy component. The image and neighborhood are passed into the function. The image's gradient is calculated again, but we only need the gradient values for that neighborhood. The maximum *M* and minimum *m* gradient values are also recorded. All the gradient values of that neighborhood is stored in a temporary list. Use the formula to calculate the energy component.

$$Egrad = \frac{m - ||gradient||}{M - m}$$

This formula automatically normalizes the values. When the difference between *m* and *M* is less than 5, *m* is set to be *M*-5.

8. After all energy components are calculated, we can use the formula to calculate total energy.

$$E = \int \alpha Econt + \beta Ecuv + \gamma Egrad\ ds$$

As mentioned before, each list for energy component has the size of neighborhood size squared. Then the list for total energy *E* will also have the same size. For each total energy in *E*, find its minimum *Emin* and record its position. If its position in the neighborhood list has the same value as our current index in the *PointsList* (i.e. same pixel coordinate/the center of the neighborhood), then do nothing. Otherwise, update the *PointsList*'s entry of current index to be the new point we found. Update points moved counter and keep the last entry of the *PointsList* updated (to be same as first entry).

9. For the next part, we need to allow corners to fit. In order to do so, the curvature of a point *Ci* should be greater than its neighbor points *Ci-1* and *Ci+1*, it should be greater the curvature threshold (*thresh1* in code), and its gradient should be greater than gradient threshold (*thresh2* in code). The curvature threshold is set to be 10% of the mean curvature values and gradient threshold is set to be 30% of the maximum gradient of the image. I tend to change these two threshold values for different images.

   To calculate curvature of a point, use the formula:

$$Ci = |\frac{U_i}{|U_i|} - \frac{U_{i+1}}{U_{i+1}}|^2$$

   Where

$$U_i = Vi - Viprev$$
$$U_{i+1} = Vinext - Vi$$

   Store all the Ci in a list *C*.

10. For each Ci in *C*, start comparing if it qualifies for corner (refer to step 9). If it does, save this index number to the *betachange* list. The *C* and *PointsList* have same size if we exclude the last entry in *PointsList*, so their indices are mapped one-to-one. In the next iteration, if any index matches with the indices in the *betachange*, the beta will be set to 0 for that point.

    Finally, update iteration counter and loop back.

Results and Discussion

Part 1

I am going to start off by showing images 1 through 8. Using the values required: 3 pixel smoothing, alpha = beta = gamma = 1, and fraction 10%. The input contour is green, two intermediate steps yellow then blue, and final output in red. The white(black for figure 8) dot is where I manually marked the corners are relaxed in the algorithm.

Figure 1. DoutImage1

Figure 2. DoutImage2

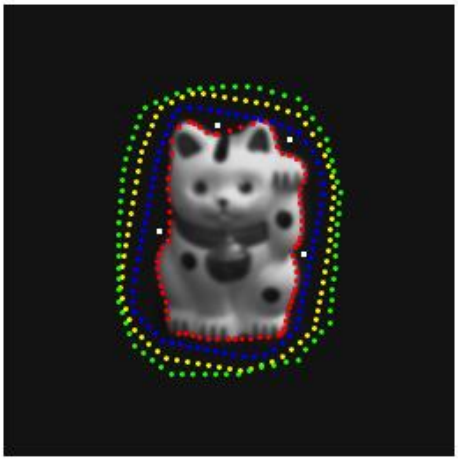

Figure 3. DoutImage3

Figure 4. DoutImage4

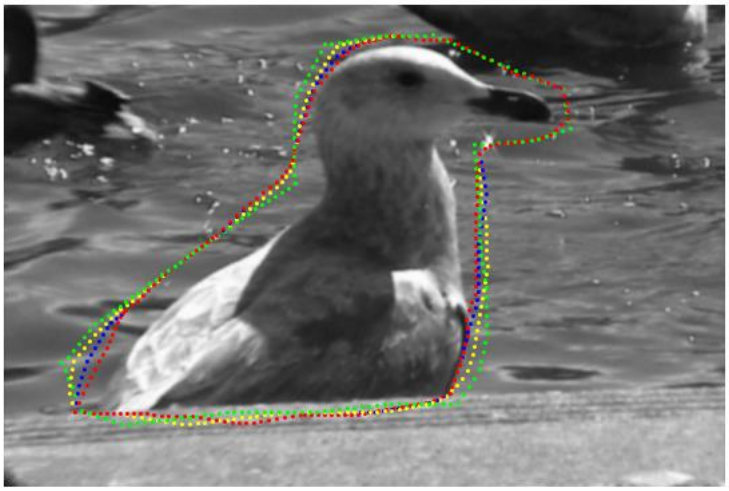

Figure 5. DoutImage5

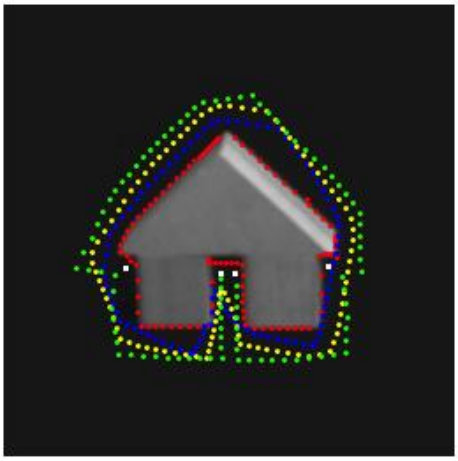

Figure 6. DoutImage6

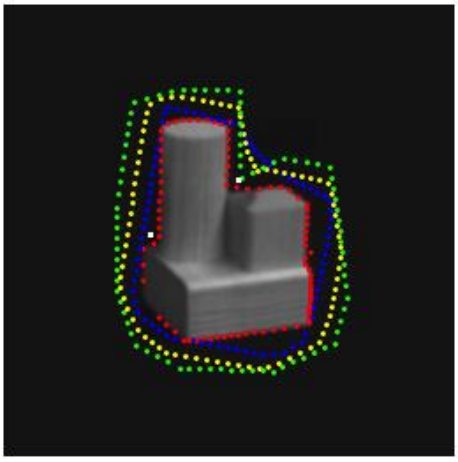

Figure 7. DoutImage7

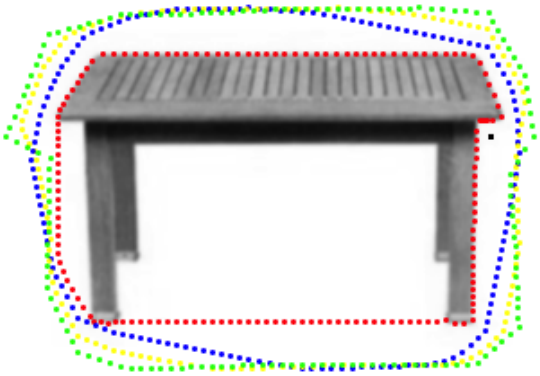

Figure 8. DoutImage8

For Figures 1, 2, 3, 6, and 7, the algorithm seems to do a good job on detecting the contour of the object. All of them have a high contrast with the background with clear boundaries. The corners are relaxed as expected.

However, for figures 4 and 8, the algorithm partially fails. Figure 4 fails at the part where the tripod and the camera men are overlapped. It is hard to decide which contour to take. Also, the camera men does not have his whole body in the image, so it is difficult to draw the contour at the beginning.

Figure 8 fails on the contour under the table are not being drawn as well as the left table leg. If we imaging the contour as a rubber band, then if I draw the initial contour exactly along the table, it would move downwards under the table to relax. If I draw it this way as figure 8, in order for the points to attach under the table, it need to stretch out and the points gets further away from each other, which is the opposite of how the algorithm works.

Similarly, in figure 6, if I do not click the inner part under the object, it would create a horizontal contour or a slight curve at the bottom instead of moving into and attach to the inner surface.

Figure 5 completely fails except a very little part at the bottom right. This failure is mostly caused by constant changes in the gradient and lack of contrast between the duck and the background. In other words, the test for gradient is not affective for this image. This is also expected result since the object is blending in with the background and with the noises (water waves) makes the algorithm mix up between the object and the waves.

Part 2

This part I am going to compared the effects of gaussian filtering, size of neighborhood, alpha, beta, and gamma values, and fraction using images 1, 3 and 6. Refer to Figures 1, 3, 6 for comparison purposes.

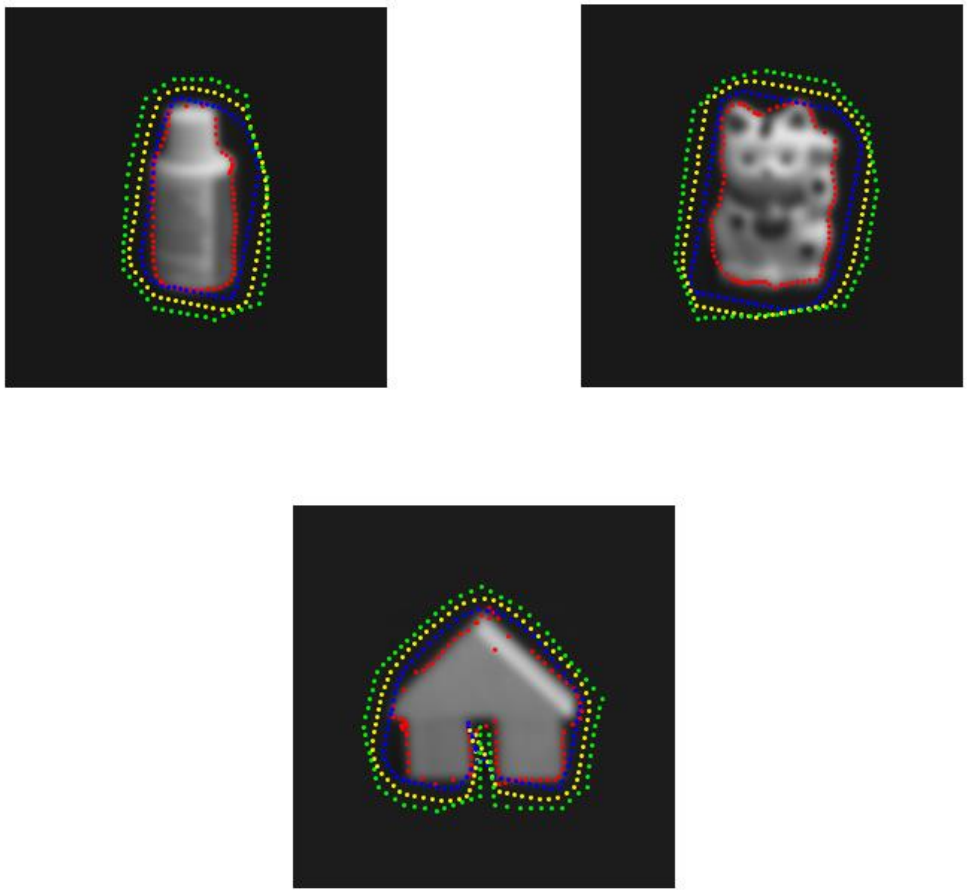

Figure 9. Effect of Gaussian Sigma = 3

It is quite obvious that with a higher Gaussian sigma value, the image is blurred. This results in borders of the of the object being too vague. Consequently, it will affect the gradient values which plays a part in the energy function. The final points are also scattered around the object in an uneven way. For example, the right side of the bottle has more points than the left side, or the concentration of points at the left-most corner for the toy brick.

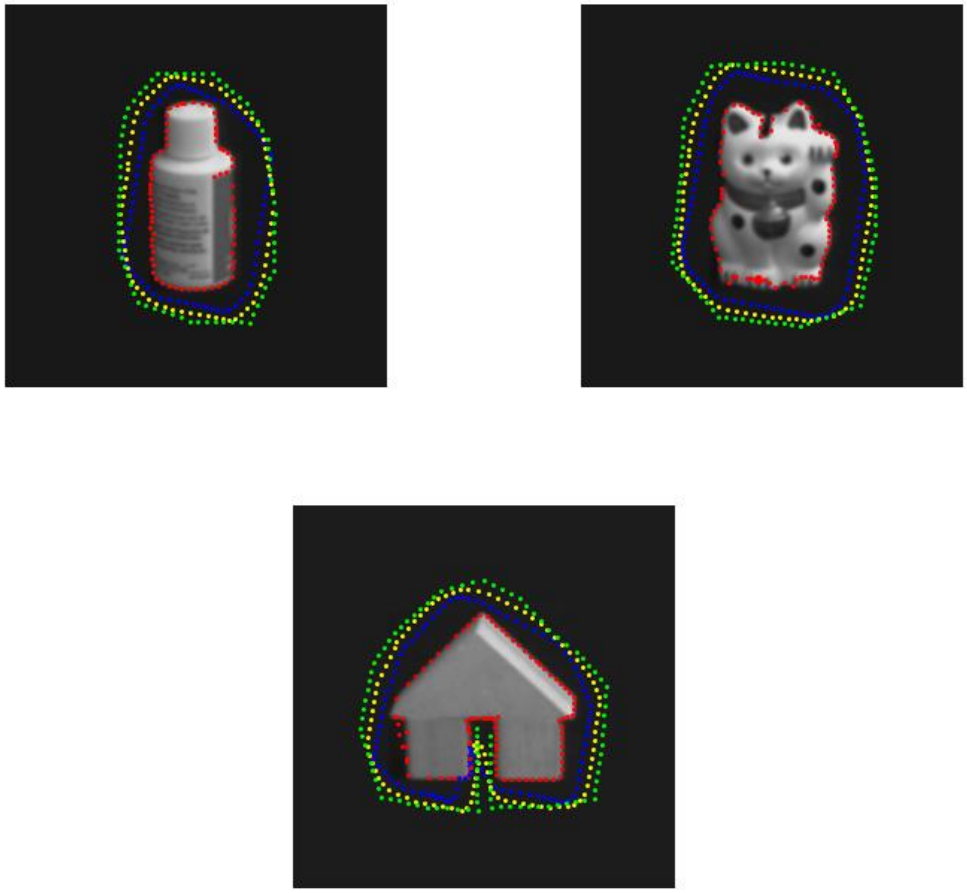

Figure 10. Effect of Neighborhood size = 5x5

With a larger neighborhood, points now have more options to move to. This creates a problem where the locations that contains the object itself are also included in the options. The right side of the bottle has points entered the outer border, this is because when the points are at the border, it sees another large gradient change between the bottle and its label. Similar for the cat statue, points entered the statue at its head and foot of the cat.

Alpha, Beta, and Gamma are the weights of each energy component. By default, the weights are equal. However, if there a need for human interference to get a better output, then changing the weights will do so.

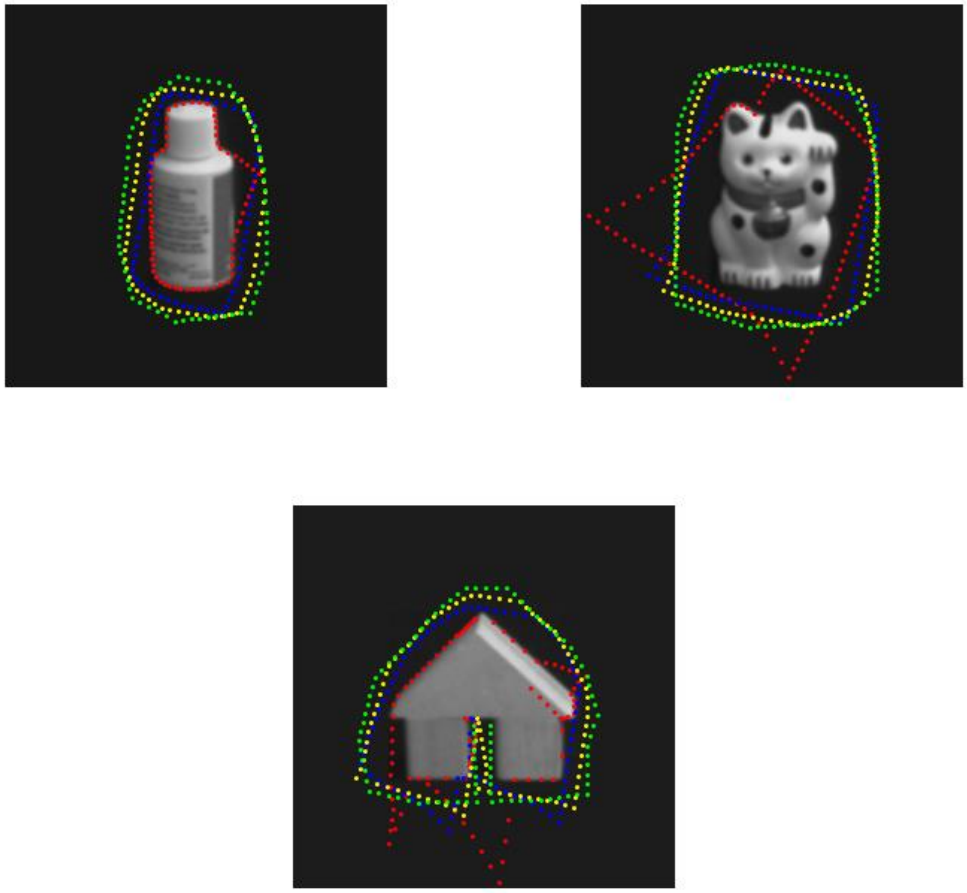

Figure 11. Effect of Alpha = 2

When alpha value is higher, it enforces the continuity term. In other words, the distance between the points will play a major role in the energy for the continuity. Many points are trying to separate them from one another and therefore moving to a location further from the object. This disturbs the calculation of total energy even when the alpha value is set higher.

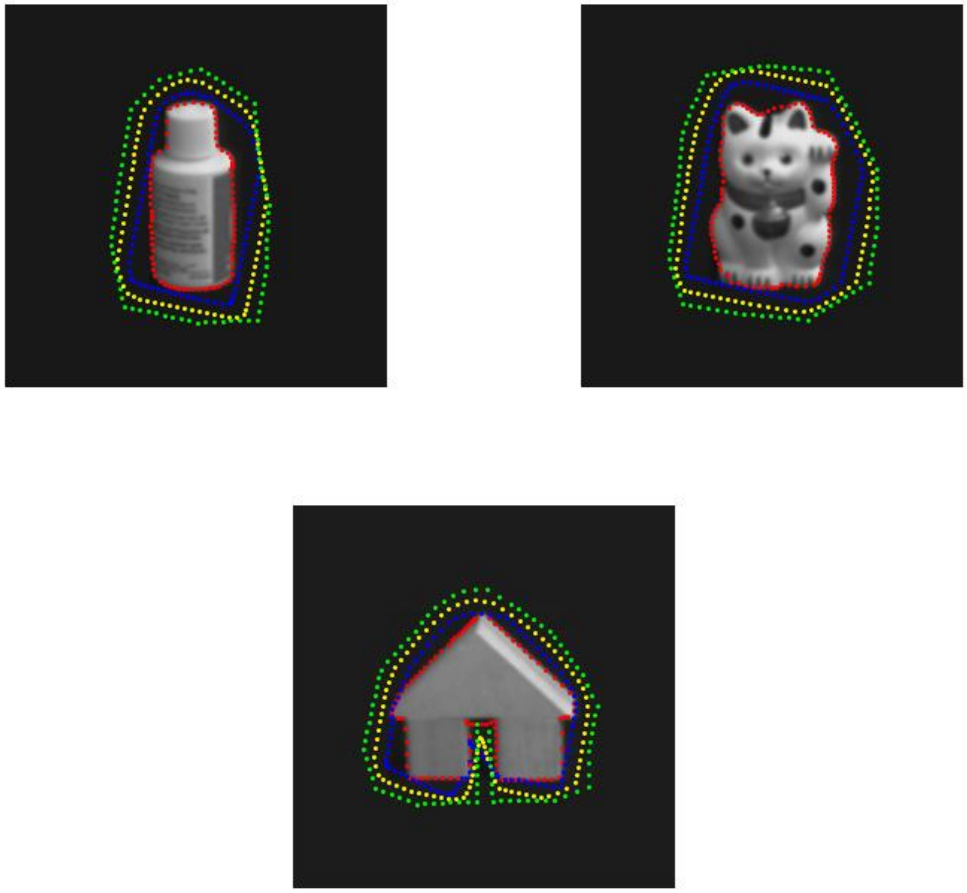

Figure 12. Effect of Beta = 10

Beta affects the curvature weight in the energy function. However, it does not seem to have much of an impact. This is because the gradient change in these three images are so high that curvature factor does not overcome the gradient factor. I tried using beta = 50, the points starts to move into the objects because the other two factors are too small in comparison to curvature. In other words, the contour will keep on getting smaller until no points are able to move. Such as forming into an oval while ignoring the object itself.

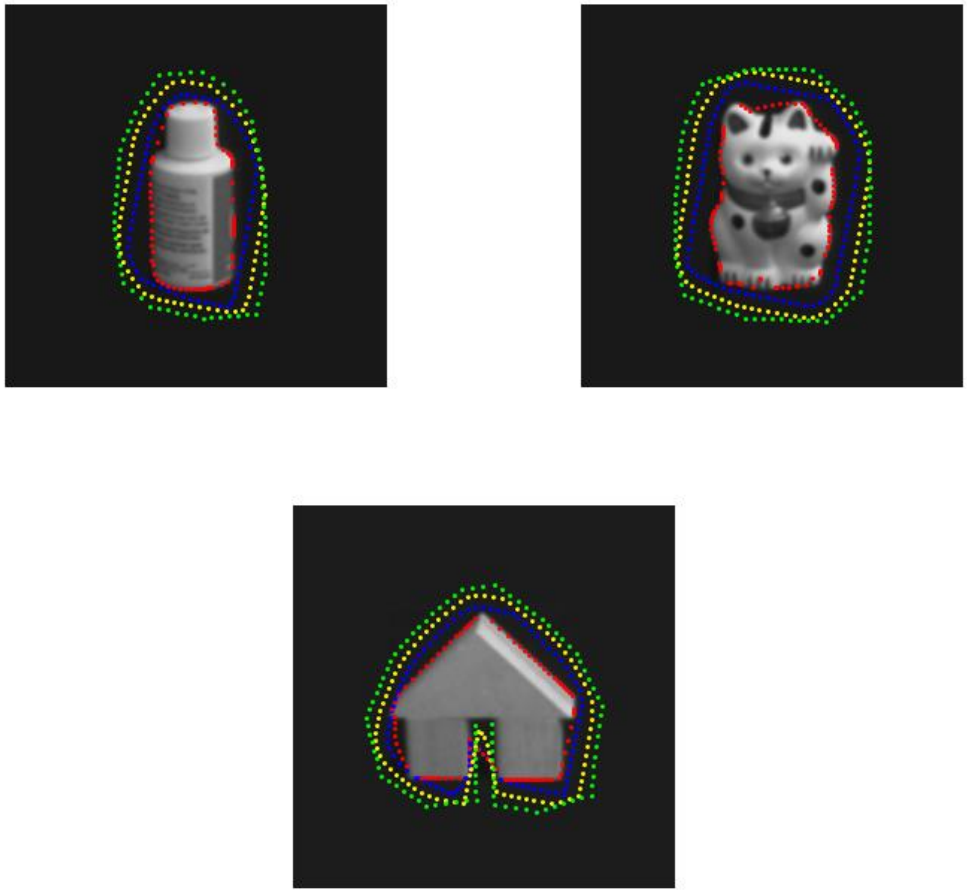

Figure 13. Effect of Gamma = 10

When gamma value is higher, the algorithm puts its weight on the gradient and starts to ignore the continuity, curvature, and even corners. From figure 13, we can see that many points are concentrated on one side of the bottle. This means the continuity energy is being ignored. From all three images, we can also see that corners are no longer being relaxed because of the curvature energy in comparison to the heavily weighted gradient energy.

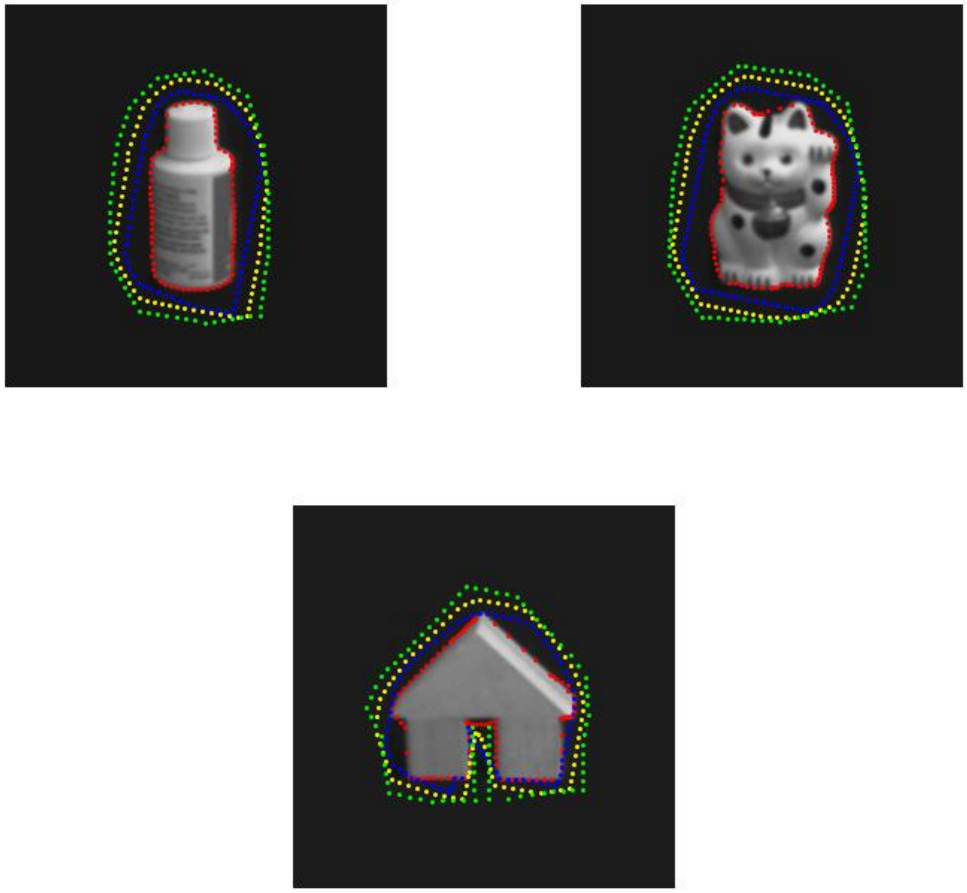

Figure 14. Effect of Fraction = 30%

In this part, the minimum number of points needed to move is increased from 10% of the total points to 30%. This does not seem to have a much of an impact on the output. However, there are more inner iterations in a single iteration. If I did not specify inner iteration counter in the code, the program is very likely to run into an infinite loop. By increase the percentage, theoretically the programs takes longer to run.

Future Improvement

From the results we can see that this algorithm does not work for every image. Images with similar gradients between the object and background, images with irregular shapes, and images with objects that has a boundary outside the image itself behave poorly under this algorithm. Then potential improvements will be solving these issues for these kinds of images. Speed is another factor that can be improved since this algorithm takes too many iterations. If the image is larger and object being more complicated, we can expect an significant slowdown.

Conclusion

In this assignment we implemented the greedy snake algorithm. There are many factors that affects the result:

- Number of iterations (threshold 3 in the pseudocode): this threshold determines how many iterations the algorithm run. If set too small, many points are unable to reach the contour of the object. If set too large, the points may have already reached its final location and the program is still running, causing use of unnecessary resource and results in a longer running time.

- Gaussian Sigma value: this blurs the image as well as the object's borders. Then it becomes very difficult for the algorithm to determine where the contour should stop and therefore creating poor results. It is best to keep sigma value = 1.

- Neighborhood Size: this allows the points to have more locations to move to. Usually it results in points moving into the object which is not what we want. It is best to keep the neighborhood size small.

- Fraction of points (minimum of points need to move). I think this is unnecessary since it increases the number of inner iterations and running time. We also run into the risk of infinite loop. If we feel that there is a need to enforce points to move we can increase the number of the overall iterations.

- Alpha: This affects the continuity and distance between points. A minor change can cause a dramatic response. It is best to be kept at default

- Beta: This puts more weight on the curvature which weakens the effects of continuity and gradient. This may result in gradient detection malfunction and contour entering the object.

- Gamma: This puts more weight on the gradient which weakens the effects of continuity and curvature. This results in concentration of points on the contour and corners or curves being ignored.

- Curvature Threshold (threshold 1 in the pseudocode): this factors affects how many points can be relaxed to form into a corner. When this value is large, many sharp corners will be ignored resulting in wrong output. Depending on visually looking at the object, we should choose a corresponding threshold value. This value differs from image to image, so it cannot be fixed and needs to be constantly changed and tested.

- Gradient Threshold (threshold 2 in the pseudocode): this also affect how many points can be relaxed to form a corner. When the gradient change in the whole image is very small, then this threshold needs to be small in order for corners to be considered. This also depends on visually looking at the image, constant testing and varying between images