Implementing the Williams/Shah Greedy Algorithm

Bilal Ahmad

March 09, 2020

1 Introduction

I implemented the greedy algorithm described by Williams and Shah to creative active contours. The algorithm is applied to images and image sequences provided for this exercise. The purpose of this exercise is to see how different parameters involved in implementing the greedy algorithm affect the final results.

The algorithm works by first loading an image and prompting a user to select initial points around an object. If the distance between points selected by the user is too great, the algorithm interpolates additional initial points. The algorithm then searches a neighborhood around each point to find a local minimum of an energy function. The point is then moved to that local minimum. For each point, the energy function is comprised of three components that account for the contour continuity, contour curvature, and surrounding image magnitude. The influence each component has on the energy calculation can be adjusted with configurable parameter constants. The algorithm keeps iterating over the points until a minimum threshold of moved points is met. This paper provides details for the high level elements of the algorithm.

2 Initial Selection of Points

An image is initially displayed and the user is prompted to select points on the image around the object. In order for the algorithm to work efficiently, there need to be enough points selected for the creation of an initial contour. If the user places too much distance between points, the algorithm adds more through interpolation. The line equation between each of the selected points is determined and additional points are added approximately every five pixels between the selected points. The code used to perform this interpolation can be seen in Listing 1. An example of interpolated points being added to an initial contour can be seen in Fig 1. In Fig 1, the user selects initial points around the coffee mug show in *Image2*.

Listing 1: MATLAB Code for Interpolating Points

```
% add extra points if necessary
lanPoints = [];
lnNumSel = length(lanSelectX);

%% use the line equation between points to figure out where
%% the interpolated added ponts go
for i =1:lnNumSel
    if (i == lnNumSel)
        lnNextX = lanSelectX(1);
        lnNextY = lanSelectY(1);
    else
        lnNextX = lanSelectX(i+1);
        lnNextY = lanSelectY(i+1);
    end

    lanPoints = [lanPoints ; lanSelectX(i) lanSelectY(i)];

    % distance between points
    lrDist = sqrt((lanSelectX(i)-lnNextX)^2 + ...
        (lanSelectY(i)-lnNextY)^2);

    % number of points that need to be added
    lnNumAdd = max(0, round(lrDist/5)-1);

    if (lnNumAdd > 0)
        for k=1:lnNumAdd
            lrDisRat = (5*k)/lrDist;
            lanPoints = [lanPoints ; ...
                (1-lrDisRat)*lanSelectX(i) + lnNextX*lrDisRat ...
                (1-lrDisRat)*lanSelectY(i) + lnNextY*lrDisRat];
        end
    end
end
```

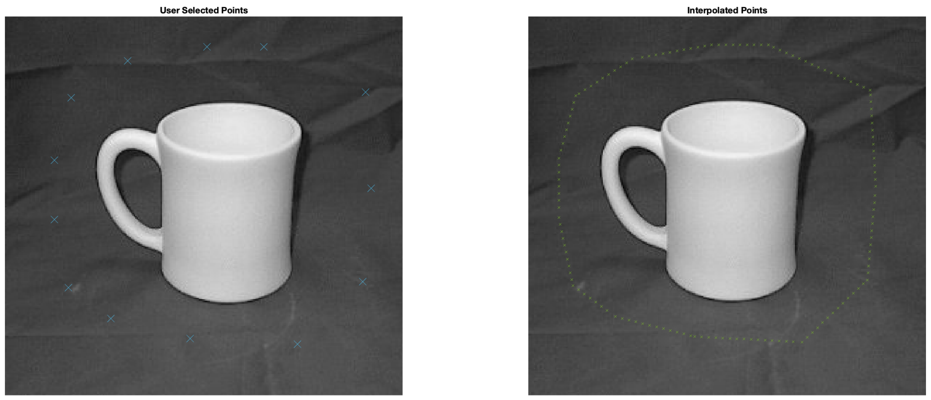

Figure 1: Adding Interpolated Points to Initial Contour

3 Implementing the Greedy Algorithm

3.1 Initial Smoothing and Gradient Calculation

First, a Gaussian filter is applied to the image. This filters our noise that may later affect the magnitude calculation. Next the gradient of the image is calculated using Sobel operators. The gradient magnitude at each pixel, I, is normalized using the maximum, M, and minimum, m, magnitude value in the surrounding neighborhood of the pixel. The normalization calculation is depicted by Eq 1. The difference between M and m is gated to be no less than 5 in Eq 1. The Gaussian smoothing implementation can be seen in Listing 2 and the magnitude normalization implementation of Eq 1 can be seen in Listing 3.

$$NormalizedMagnitude = \frac{m - I}{M - m} \tag{1}$$

Listing 2: MATLAB Code for Initial Smoothing

```
% Calculate the gradient magnitude for each point
% convert image to greyscale if necessary
if (size(lanImage, 3) ~= 1)
    lanImage = rgb2gray(lanImage);
end
% Smooth Image and Obtain Gradient Magnitudes
% imgaussfilt performs a convolution with a 5x5 Gaussian kernel
larSmoothedImage = imgaussfilt(lanImage, arSigma);
larAdjImage = imgradient(larSmoothedImage);
```

Listing 3: MATLAB Code for Normalizing Gradient Magnitude

```
% Calculate the gradient magnitude for each point
for i = 1+lnSize:lnNumRows-lnSize
    for j = 1+lnSize:lnNumCols-lnSize
        lnMaxGrad = max(max(larAdjImage(i-lnSize:i+lnSize,...
            (j-lnSize):(j+lnSize))));
        lnMinGrad = min(min(larAdjImage(i-lnSize:i+lnSize,...
            j-lnSize:j+lnSize)));

        if (lnMaxGrad - lnMinGrad < 5)
            lnMinGrad = lnMaxGrad - 5;
        end
        larMagGradients(i,j) = (lnMinGrad-larAdjImage(i,j)) / ...
            (lnMaxGrad - lnMinGrad);
    end
end
```

3.2 Continuity Component

The continuity component for each contour point find the distance between the current point and the previous point and subtracts that distance from the average distance between points on the contour. The value will me minimized when the distance between points is close to the average distance. This serves to prevent the bunching of points as the contour wraps around the object. The code used to implement these steps is shown in Listing 4.

Listing 4: MATLAB Code for Calculating Continuity Component

```
% Obtain continuity terms
        larContTerm = ones(2*lnSize+1, 2*lnSize+1);
        b = 1;
        for j=-lnSize:lnSize
            a = 1;
            for k = -lnSize:lnSize
                larContTerm(a,b) = abs(lrAvgDist - ...
                 sqrt((lanPoints(i,1)+j-lanPrevPoint(1,1))^2 + ...
                 ((lanPoints(i,2)+k-lanPrevPoint(1,2))^2)));
                a = a + 1;
            end
            b = b + 1;
        end
        larContTerm = larContTerm./max(max(larContTerm));
```

3.3 Curvature Component

The curvature component of the energy calculation uses the current contour point as well as the previous and next contour points to determine the curvature of the section. The calculation exists to prevent sharp corners from arising. In the event that there are corners on the object that the contour needs to wrap around, the algorithm will set the curvature component to zero so it does not affect the energy calculation for the point. The code implementation of calculating the curvature component can be seen in Listing 5 and the implementation for determining whether to allow for corners is shown in Listing 6.

Listing 5: MATLAB Code for Calculating Continuity Component

```
        % Obtain curvature terms
        larCurvTerm = ones(2*lnSize+1 ,2*lnSize+1);
        b = 1;
        for j=-lnSize:lnSize
            a = 1;
            for k = -lnSize:lnSize
                larCurvTerm(a,b) = (lanPrevPoint(1,1) - ...
                        2*(lanPoints(i,1)+j) + ...
                        lanNextPoint(1,1))^2 + ...
                    (lanPrevPoint(1,2) - ...
                        2*(lanPoints(i,2)+k) + ...
                        lanNextPoint(1,2))^2;
                a = a + 1;
            end
            b = b + 1;
        end
        larCurvTerm = larCurvTerm./max(max(larCurvTerm));
```

Listing 6: MATLAB Code for Calculating Continuity Component

```
% Allow for corners
        if larCs(i) > lrPrevC && larCs(i) > lrNextC && ...
                larCs(i) > lrThresh1 && ...
                lrMagV > lrThresh2
            larBetas(i) = 0;
        else
            larBetas(i) = anBeta;
        end
    end
```

3.4 Image Magnitude Component

The magnitude component uses the values from the normalized image gradient calculated earlier to determine the presence of en edge. The code for implementing the calculation of the gradient component for each pixel can be seen in Listing 7

Listing 7: MATLAB Code for Calculating Image Component

```
        % Obtain image energy term
        larImageTerm = ones(2*lnSize+1 ,2*lnSize+1);
        b = 1;
        for j=-lnSize:lnSize
            a = 1;
            for k = -lnSize:lnSize
                lnRow = lanPoints(i,2);
                lnCol = lanPoints(i,1);

                larImageTerm(a,b) = larAdjImage(...
                    lanPoints(i,2)+k, lanPoints(i,1)+j);
                a = a + 1;
            end
            b = b + 1;
        end
```

4 Applying the Algorithm to Images

The greedy algorithm is applied to images provided for this assignment. The results of the algorithm applications can be seen in Fig 2, Fig 3, Fig 4, Fig 5, Fig 6, Fig 7, Fig 8, and Fig 9. After the algorithm is tested on the images with default settings, different parameters are adjusted to determine their effect on the algorithm. The results of these applications can be seen in Fig 10, Fig 11, Fig 12, Fig 13, Fig 14, and Fig 15. Finally, a sequence of images from a video are used to test the algorithm. The result of the final image in each video sequence is shown in Fig 16.

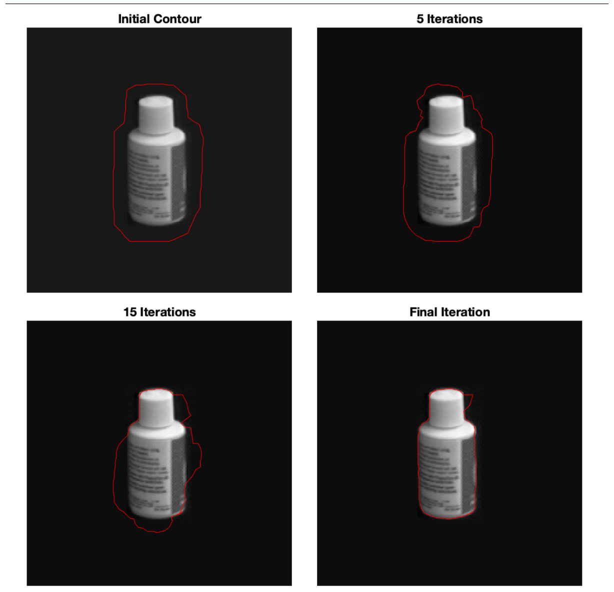

Figure 2: Applying Greedy Algorithm to *Image1*

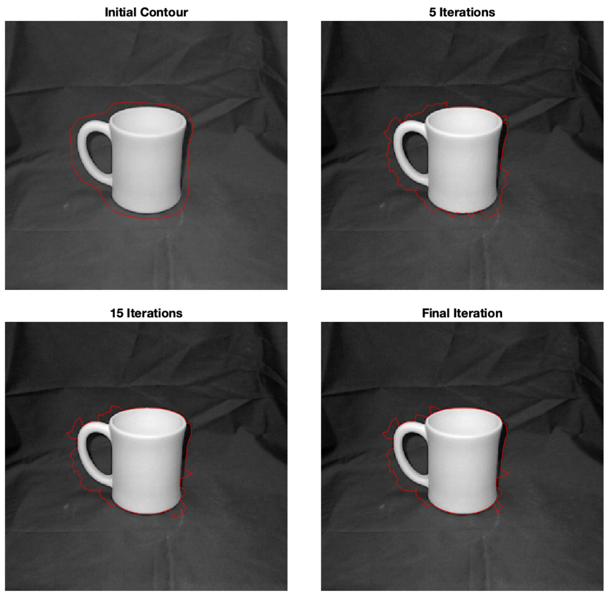

Figure 3: Applying Greedy Algorithm to *Image2*

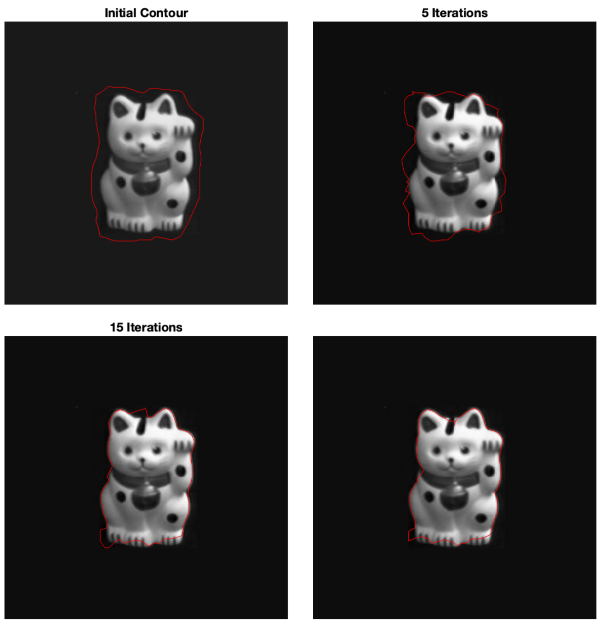

Figure 4: Applying Greedy Algorithm to *Image3*

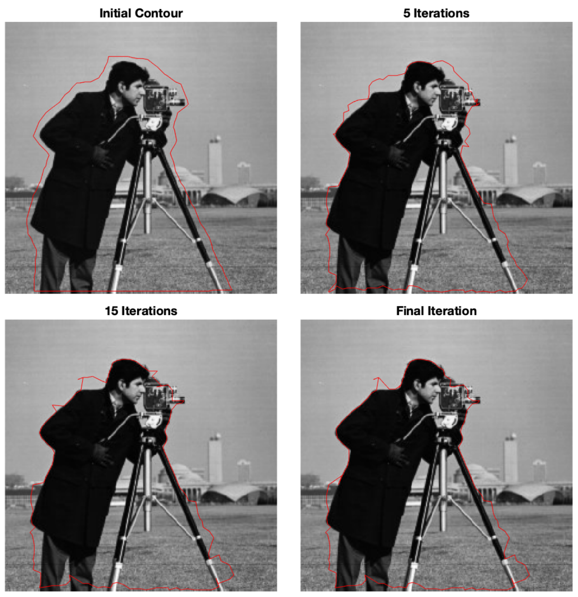

Figure 5: Applying Greedy Algorithm to *Image4*

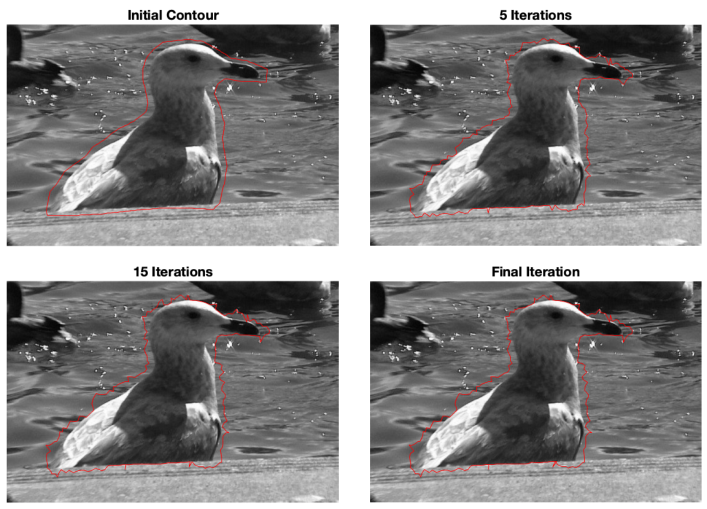

Figure 6: Applying Greedy Algorithm to *Image6*

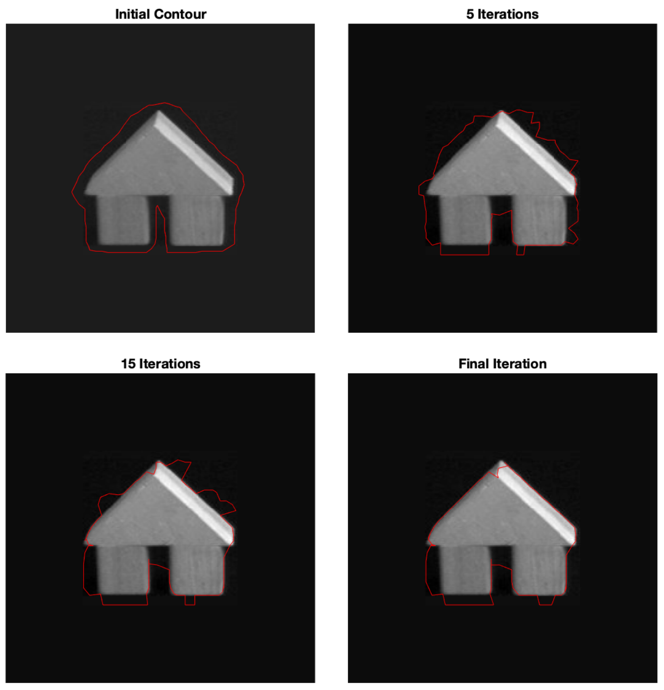

Figure 7: Applying Greedy Algorithm to *Image6*

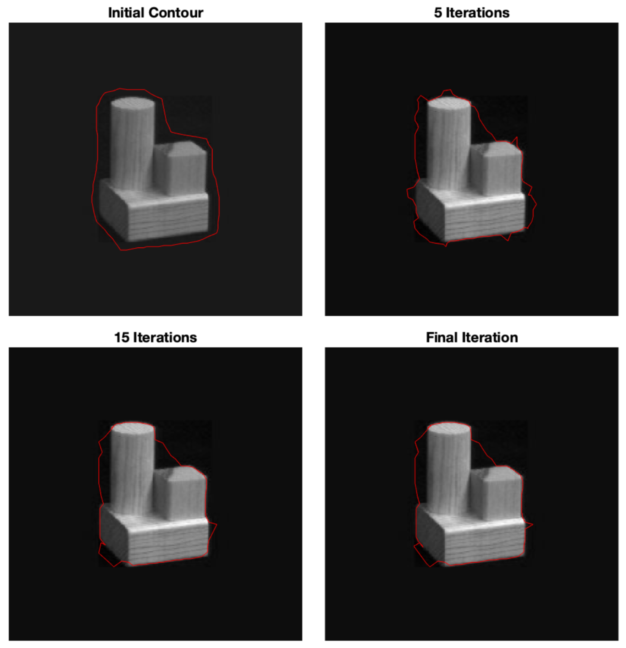

Figure 8: Applying Greedy Algorithm to *Image7*

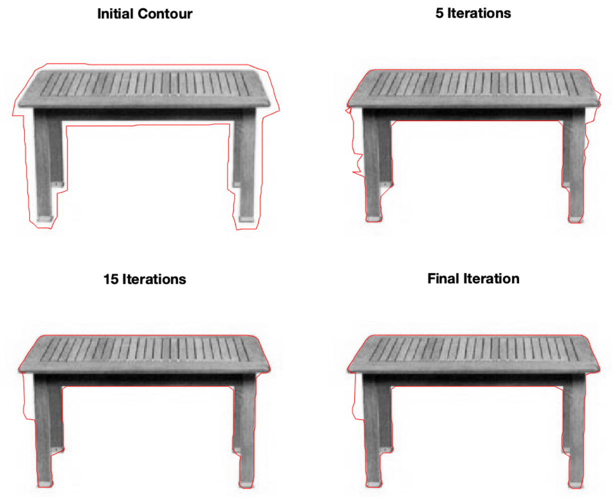

Figure 9: Applying Greedy Algorithm to *Image8*

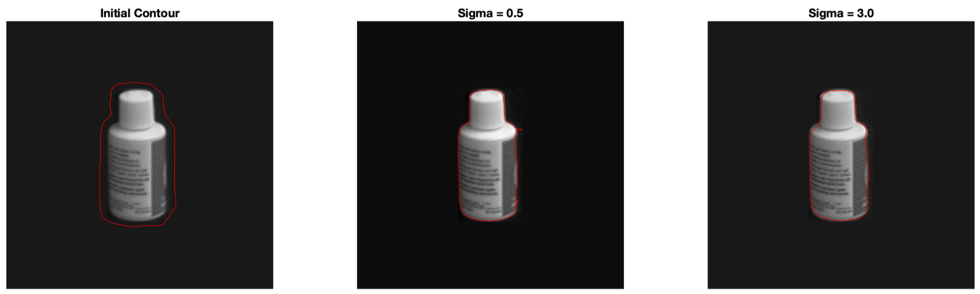

Figure 10: Applying Greedy Algorithm to *Image1* with Different Sigmas

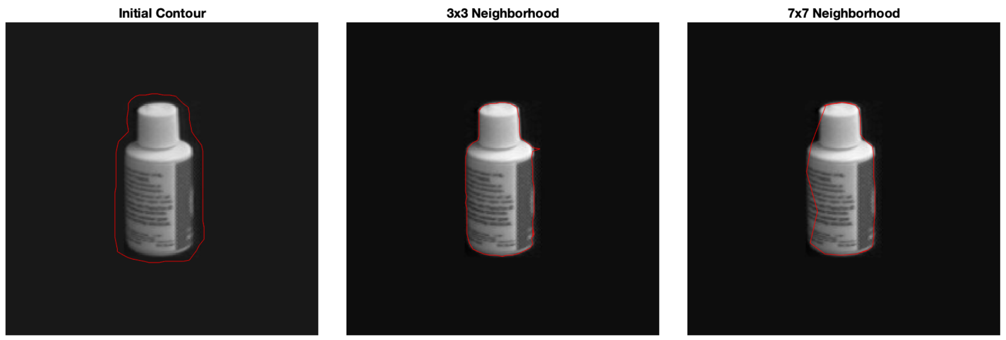

Figure 11: Applying Greedy Algorithm to *Image1* with Different Neighborhoods

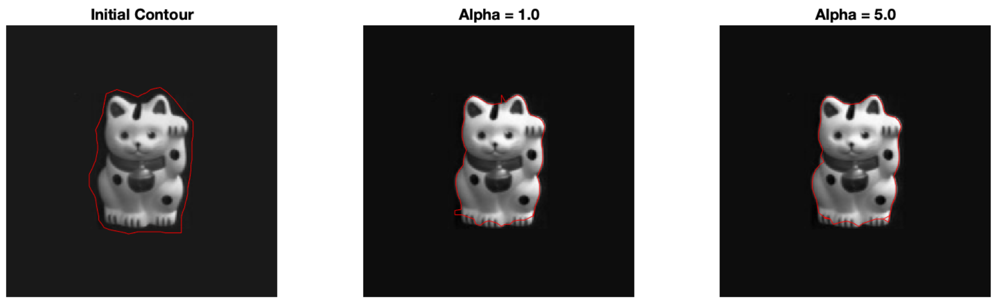

Figure 12: Applying Greedy Algorithm to *Image3* with Different Alphas

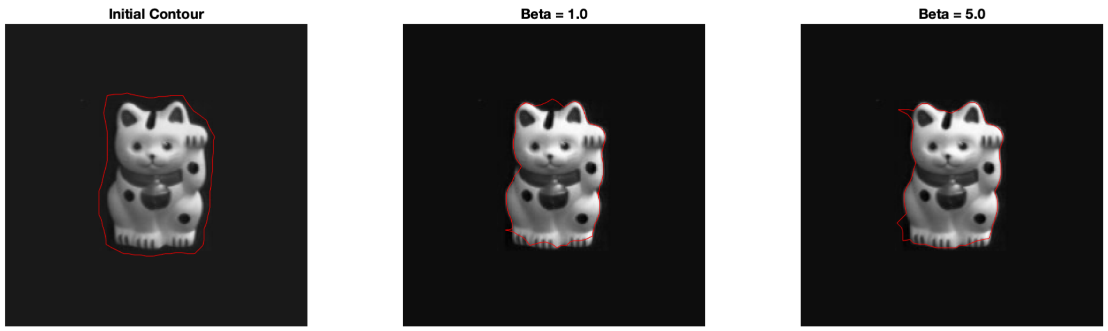

Figure 13: Applying Greedy Algorithm to *Image3* with Different Betas

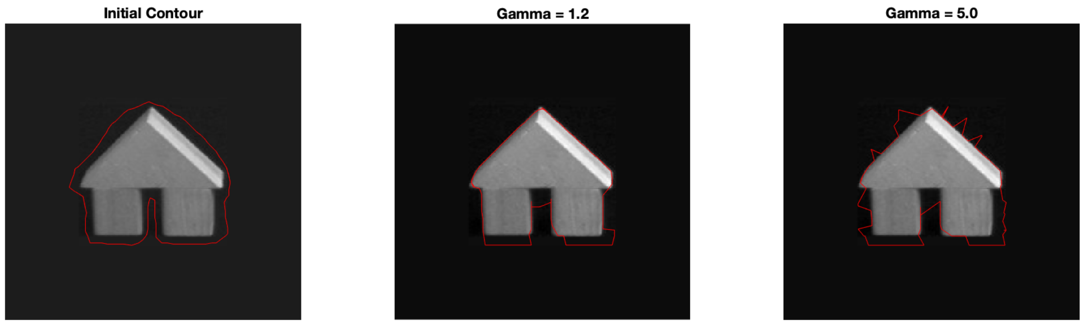

Figure 14: Applying Greedy Algorithm to *Image6* with Different Gammas

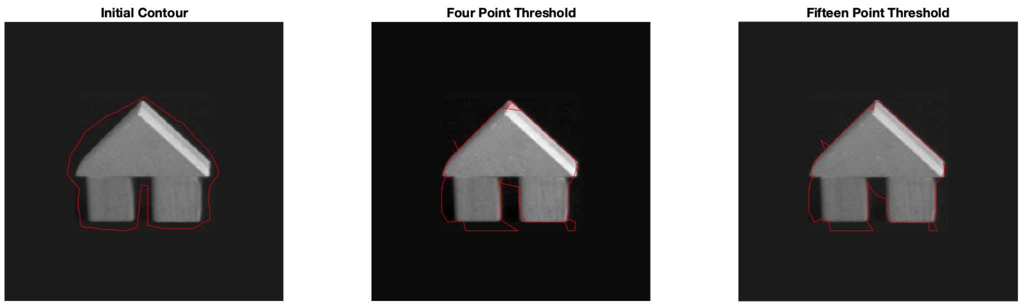

Figure 15: Applying Greedy Algorithm to *Image6* with Different Completion Thresholds

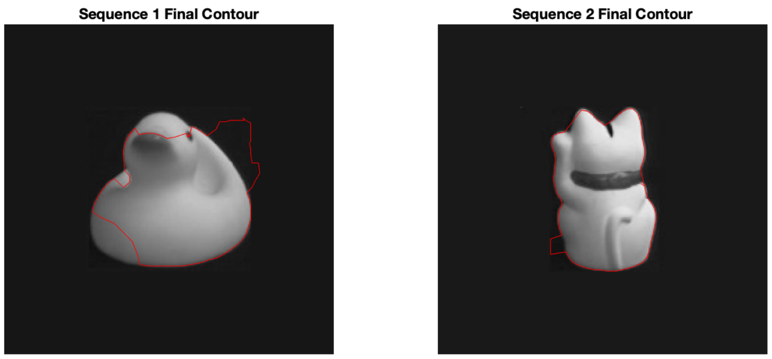

Figure 16: Applying Greedy Algorithm to *Sequence1* and *Sequence2*

5 Analysis

5.1 Adjusting Parameter Values

Increasing the sigma size caused the contour to creep inside the object. This is expected since an increased sigma makes it harder to discern edges and this will effect the image gradient component calculation. Increasing the neighborhood size made the contour shape less smooth. This result makes sense because there are more potential pixels from which each point can move to so there will be more significant shifts in the contour.

Increasing the alpha made the contour points more evenly spaced. This makes sense since a larger emphasis is placed on keeping the points an average distance apart. Increasing the beta value should decrease the amount of false corners in the image. While there are fewer unwanted corners in the image with a higher beta value, the effect wasn't as significant as when increasing other parameters. This is likely due to the fact that the beta value is sometimes zeroed out in the even of real corners anyway. in creasing the gamma threshold made the contours more susceptible to noise. This result is expected since the gradient contour has more influence with a larger gamma value.

Increasing the stop ratio causes the algorithm to perform worse than with a lower stop ratio. This result is expected since the algorithm terminates before all contour points can be moved to their ideal position.

5.2 Effect on Image Sequences

The active contour mapping did a little bit better with the images in *Sequence2* which makes sense since the cat is more rotationally symmetrical. As the duck rotates, the contours tried to expand and contract to match the bill and tail location. Because the car remained a relatively consistent cylinder throughout the rotation, its contours look better.