Implementing the Canny Edge Detection Algorithm

Bilal Ahmad

February 16, 2020

1 Introduction

I implemented the Canny edge detection algorithm and applied it on four images. Three of the images: *Flowers*, *Syracuse_01*, and *Syracuse_02* were provided for the exercise. I chose to use an image of the CMG Headquarters building in Beijing, *cctv*, as the fourth image. The purpose of this exercise is to see how different parameters involved in implementing the Canny algorithm affect the final edge detection results.

The algorithm is implemented in three steps. The first step consists of applying a Gaussian filter to the image and calculating the strength and orientation images. In the next step, the strength and orientation images are used to perform non-maximum suppression. Finally, the third step applies hysteresis-thresholding on the suppressed image. All code for this exercise was written using MATLAB and is based on code submitted to the MathWorks File Exchange [1].

2 Canny Edge Detection Implementation

2.1 Producing Strength and Orientation Images

A Gaussian filter is applied to the image to perform initial smoothing. Next, two convolutions are applied to the image using the Sobel operators G_x and G_y to obtain the gradient components J_x and J_x. The Sobel operators, defined in Eq 1 and Eq 2, approximate the derivatives for the horizontal and vertical changes in the image.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{1}$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2}$$

After obtaining the gradient components, the algorithm uses Eq 3 and Eq 4 to estimate the edge strength and orientation of the edge-normal from the component images. The resultant strength and orientation images are E_s and E_o.

$$e_s = \sqrt{J_x^2(i,j) + J_y^2(i,j)} \tag{3}$$

$$e_o = \arctan \frac{J_y}{J_x} \tag{4}$$

The code used to implement these steps is shown in Listing 1. The *artan2* MATLAB function performs a four quadrant inverse tangent on the elements. This will result in values ranging from $-\pi$ to π rather than the typical $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ output range of arctangent. The larger range will be accounted for when performing the non-maximum suppression.

Listing 1: MATLAB Code for Strength and Orientation Images

```
%% Calculate Strength and Orientation Images
lrSigma = arSigma;

% imgaussfilt performs a convolution with a 5x5 Gaussian kernel
larSmoothedImage = imgaussfilt(lanImage, lrSigma);

% Compute the horizontal and vertical gradients
lanGradX = [-1, 0, 1; -2, 0, 2; -1, 0, 1];
lanGradY = [1, 2, 1; 0, 0, 0; -1, -2, -1];
larGradImgX = conv2(larSmoothedImage, lanGradX, 'same');
larGradImgY = conv2(larSmoothedImage, lanGradY, 'same');

% Get the orientation image, convert radians to degrees
larOrImage = atan2 (larGradImgY, larGradImgX);
larOrImage = larOrImage*180/pi;

% Get the strength image
larStrImage = (larGradImgX.^2) + (larGradImgY.^2);
larStrImage = sqrt(larStrImage);
```

2.2 Performing Non-Maximum Suppression

The strength image, E_s, may contain wide ridges around a local maxima. Non-maximum suppression is applied to produce edges that are one pixel wide. The suppression works by first approximating the edge-normal direction from the orientation image, E_o. The

estimated direction, d_k, for each orientation image index is calculated using Eq 5. The MATLAB implementation for calculating d_k is shown in Listing 2.

$$d_k = \begin{cases} 0 & 0 \leq e_o(i,j) < 22.5 \vee 157.5 \leq e_o(i,j) < 202.5 \vee 337.5 \leq e_o(i,j) < 360 \\ 45 & 22.5 \leq e_o(i,j) < 67.5 \vee 202.5 \leq e_o(i,j) < 247.5 \\ 90 & 67.5 \leq e_o(i,j) < 112.5 \vee 247.5 \leq e_o(i,j) < 292.5 \\ 135 & 112.5 \leq e_o(i,j) < 157.5 \vee 292.5 \leq e_o(i,j) < 337.5 \end{cases} \tag{5}$$

Listing 2: Approximating Direction in Orientation Image

```
% Update the orientation image values to be 0, 45, 90, or 135
for i = 1   : lnNumRows
    for j = 1 : lnNumCols
        if ((larOrImage(i,j) >= 0 ) && ...
                (larOrImage(i,j) < 22.5) || ...
            (larOrImage(i,j) >= 157.5) && ...
                (larOrImage(i,j) < 202.5) || ...
            (larOrImage(i,j) >= 337.5) && ...
                (larOrImage(i,j) <= 360))
            larOrImage(i,j) = 0;
        elseif ((larOrImage(i,j) >= 22.5) && ...
                    (larOrImage(i,j) < 67.5) || ...
                (larOrImage(i,j) >= 202.5) && ...
                    (larOrImage(i,j) < 247.5))
            larOrImage(i,j) = 45;
        elseif ((larOrImage(i,j) >= 67.5 && ...
                    larOrImage(i,j) < 112.5) || ...
                (larOrImage(i,j) >= 247.5 && ...
                    larOrImage(i,j) < 292.5))
            larOrImage(i,j) = 90;
        else
            larOrImage(i,j) = 135;
        end
    end
end
```

After the edge normal directions are approximated, non-maximum suppression can be applied to the strength image, E_s. For every element in E_s, if either of the element's two neighbors in the edge-normal direction are greater, update the element to be zero. The MATLAB implementation of the suppression can be seen in Listing 3.

Listing 3: Non-Max Suppression

```
% Perform the non-maximum supression
larNonMaxSup = zeros (lnNumRows, lnNumCols);

for i=2:lnNumRows-1
    for j=2:lnNumCols-1
        if (larOrImage(i,j)==0)
            larNonMaxSup(i,j) = ...
                (larStrImage(i,j) == max([larStrImage(i,j), ...
                larStrImage(i,j+1), ...
                larStrImage(i,j-1)]));
        elseif (larOrImage(i,j)==45)
            larNonMaxSup(i,j) = ...
                (larStrImage(i,j) == max([larStrImage(i,j), ...
                larStrImage(i+1,j-1), ...
                larStrImage(i-1,j+1)]));
        elseif (larOrImage(i,j)==90)
            larNonMaxSup(i,j) = ...
                (larStrImage(i,j) == max([larStrImage(i,j), ...
                larStrImage(i+1,j), ...
                larStrImage(i-1,j)]));
        elseif (larOrImage(i,j)==135)
            larNonMaxSup(i,j) = ...
                (larStrImage(i,j) == max([larStrImage(i,j), ...
                larStrImage(i+1,j+1), ...
                larStrImage(i-1,j-1)]));
        end
    end
end

larNonMaxSup = larNonMaxSup.*larStrImage;
```

2.3 Hysteresis-Thresholding

The final step of this Canny edge detection implementation is performing hysteresis-thresholding. The process works by first applying a lower and upper threshold, τ_l and τ_h, to the resultant image from the non-maximum suppression, I_n. Each element in I_n lower than τ_l is changed to zero and each element greater than τ_h is changed to one. For values in-between the two thresholds, the algorithm then follows the chains of connected local maxima and saves the connected contours found. The implementation

discussed in class only looks for local maxima in the two directions parallel to the edge. For my implementation, I chose to look for local maxima in all eight of the surrounding pixels because it produced better results and is used by researchers [2]. The MATLAB implementation of the hysteresis-thresholding can be seen in Listing 4

Listing 4: Hysteresis-Thresholding

```
%% Perform Hysteresis Thresholding
% upper and lower threshold values
lrThreshLow = arLowThresh;
lrThreshHigh = arHighThresh;
lrThreshLow = lrThreshLow * max(max(larNonMaxSup));
lrThreshHigh = lrThreshHigh * max(max(larNonMaxSup));
larHysThresh = zeros(lnNumRows, lnNumCols);

for i = 1  : lnNumRows
    for j = 1 : lnNumCols
        if (larNonMaxSup(i, j) < lrThreshLow)
            larHysThresh(i, j) = 0;
        elseif (larNonMaxSup(i, j) > lrThreshHigh)
            larHysThresh(i, j) = lrThreshHigh;
        elseif (larNonMaxSup(i+1,j)>lrThreshHigh || ...
                larNonMaxSup(i-1,j)>lrThreshHigh || ...
                larNonMaxSup(i,j+1)>lrThreshHigh || ...
                larNonMaxSup(i,j-1)>lrThreshHigh || ...
                larNonMaxSup(i-1, j-1)>lrThreshHigh || ...
                larNonMaxSup(i-1, j+1)>lrThreshHigh || ...
                larNonMaxSup(i+1, j+1)>lrThreshHigh || ...
                larNonMaxSup(i+1, j-1)>lrThreshHigh)
            larHysThresh(i,j) = 1;
        end
    end
end

% cast the values into integers
larResult = 255 - uint8(larHysThresh.*255);
```

3 Applying the Algorithm to Images

The Canny edge detection algorithm was applied to the images: *Flowers*, *Syracuse_01*, *Syracuse_02*, and *cctv*. Different threshold and sigma values were used in the applications

to see how the parameters affected the algorithm. The resultant images can be seen in Fig 1, Fig 2, Fig 3, and Fig 4.

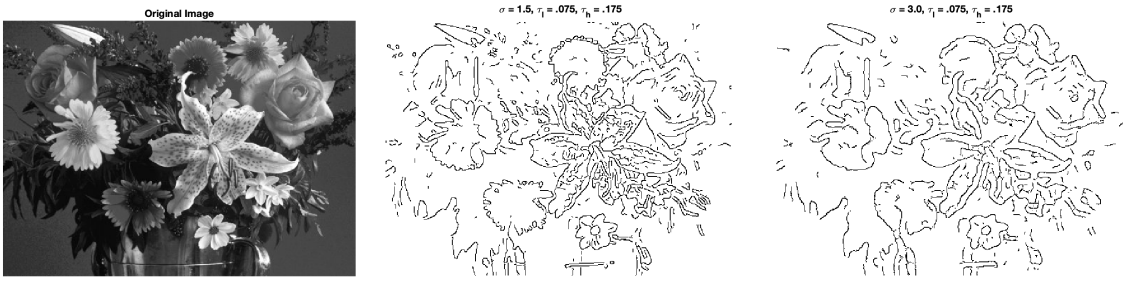

Figure 1: Applying Canny Detection with Different Sigmas to *Flowers*

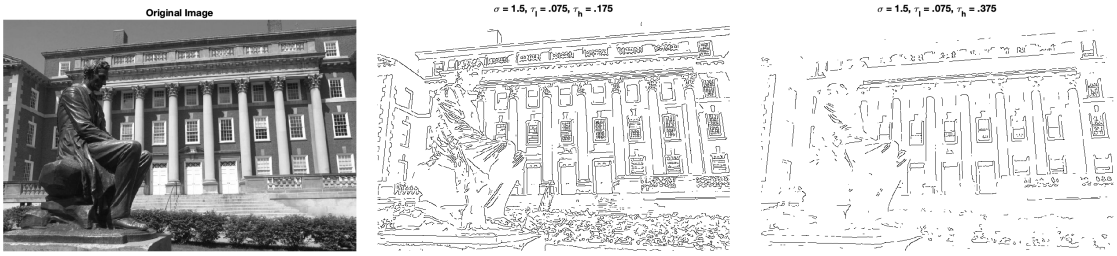

Figure 2: Applying Canny Detection with Different Thresholds to *Syracuse_01*

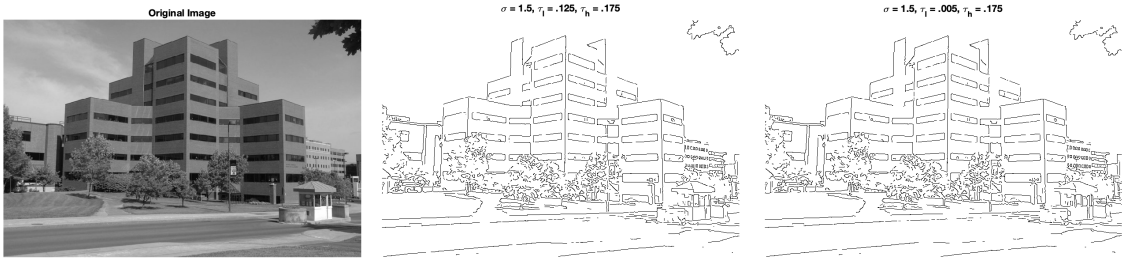

Figure 3: Applying Canny Detection with Different Thresholds to *Syracuse_02*

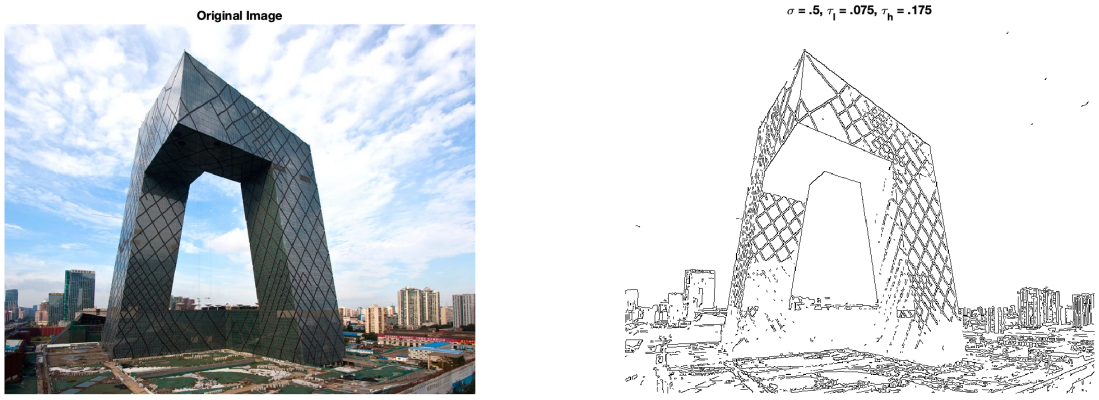

Figure 4: Applying Canny Detection to *cctv*

4 Analysis

4.1 Adjusting Sigma Values

From Fig 1, you can see the results of using two different sigma values during the initial Gaussian filtering. There are fewer edges detected when using a larger sigma value. This is as expected since using a larger sigma value will smooth out and erase more edges in the image. This edge-erasure will make it difficult for the algorithm to detect edges after the initial smoothing.

4.2 Adjusting Threshold Values

In Fig 2 and Fig 3 different threshold values are used for the hysteresis-thresholding step of the Canny edge detection algorithm. Increasing τ_h results in fewer edges in the resultant image. This makes sense because fewer elements of I_n will meet the criteria for connecting contours. Decreasing τ_l did not have as drastic of an impact on the images. Looking closely at Fig 3 you can see that there are more connected elements in the resultant images when using a smaller τ_l. This makes sense because fewer elements of I_n are initially being filtered out with the smaller τ_l.

References

[1] Rachmawan (2020). Canny Edge Detection (matlabcentral/fileexchange/46859-canny-edge-detection), MATLAB Central File Exchange. Retrieved February 16, 2020.

[2] R. R. Rakesh, P. Chaudhuri and C. A. Murthy, "Thresholding in edge detection: a statistical approach," in IEEE Transactions on Image Processing, vol. 13, no. 7, pp. 927-936, July 2004.