The Report of MP5

I. Basic Concepts

1.1 Canny edge detector

Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various computer vision systems. Canny has found that the requirements for the application of edge detection on diverse vision systems are relatively similar. Thus, an edge detection solution to address these requirements can be implemented in a wide range of situations. The general criteria for edge detection includes:

1. Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible.
2. The edge point detected from the operator should accurately localize on the center of the edge.
3. A given edge in the image should only be marked once, and where possible, image noise should not create false edges.

To satisfy these requirements Canny used the calculus of variations – a technique which finds the function which optimizes a given functional. The optimal function in Canny's detector is described by the sum of four exponential terms, but it can be approximated by the first derivative of a Gaussian. Among the edge detection methods developed so far, Canny edge detection algorithm is one of the most strictly defined methods that provides good and reliable detection. Owing to its optimality to meet with the three criteria for edge detection and the simplicity of process for implementation, it became one of the most popular algorithms for edge detection.

1.2 Algorithm and Implementation

Step 1 Gaussian filter

Since all edge detection results are easily affected by image noise, it is essential to filter out the noise to prevent false detection caused by noise. To smooth the image, a Gaussian filter is applied to convolve with the image. This step will slightly smooth the image to reduce the effects of obvious noise on the edge detector.

Step 2 Determine the intensity gradients

The gradients can be determined by using a Sobel filter where A is the image. An edge occurs when the color of an image changes, hence the intensity of the pixel changes as well.

$$G_X = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} A \text{ , } G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{bmatrix} A$$

Then, calculate the magnitude and angle of the directional gradients:

$$|G| = \sqrt{G_x^2 + G_y^2}$$

$$angle(G) = \arctan\left(\frac{G_x}{G_y}\right)$$

Step 3 Non-maximum suppression

Non-maximum suppression is an edge thinning technique. After applying gradient calculation, the edge extracted from the gradient value is still quite blurred. With respect to criterion 3, there should only be one accurate response to the edge. Thus non-maximum suppression can help to suppress all the gradient values (by setting them to 0) except the local maxima, which indicate locations with the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:

1. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
2. If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (i.e., the pixel that is pointing in the y-direction, it will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

In some implementations, the algorithm categorizes the continuous gradient directions into a small set of discrete directions, and then moves a 3x3 filter over the output of the previous step (that is, the edge strength and gradient directions). At every pixel, it suppresses the edge strength of the center pixel (by setting its value to 0) if its magnitude is not greater than the magnitude of the two neighbors in the gradient direction. In this homework, I also implement a method using interpolation. And I found that interpolation gives a nice result but at the cost of longer run time.

Step 4 Double threshold

After application of non-maximum suppression, remaining edge pixels provide a more accurate representation of real edges in an image. However, some edge pixels remain that are caused by noise and color variation. In order to account for these spurious responses, it is essential to filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value. This is accomplished by selecting high and low threshold values. If an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is smaller than the low threshold value, it will be suppressed. The two threshold values are empirically determined and their definition will depend on the content of a given input image.

Step 5 Edge linking

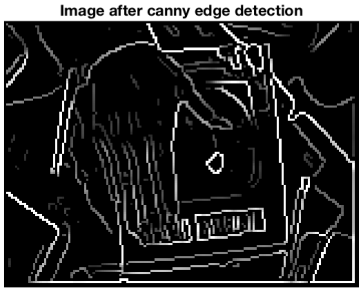
So far, the strong edge pixels should certainly be involved in the final edge image, as they are extracted from the true edges in the image. However, there will be some debate on the weak edge pixels, as these pixels can either be extracted from the true edge, or the noise/color variations. To achieve an accurate result, the weak edges caused by the latter reasons should be removed. Usually a weak edge pixel caused from true edges will be connected to a strong edge pixel while noise responses are unconnected.

1.3 The purpose of the function

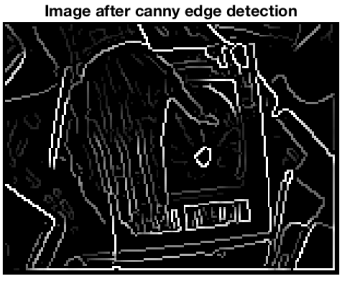
The purpose of this MP is to implement the canny edge detector. In this MP, I use Matlab to finish such assignment.

II. Results and Analysis

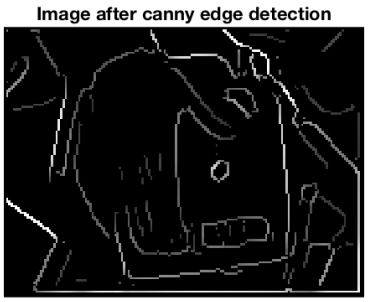
2.1 Test Results



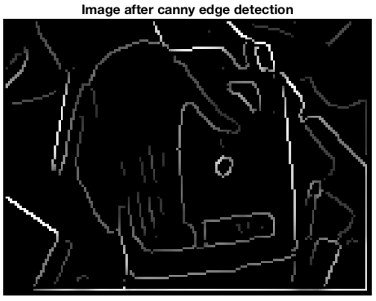
joy1 N=3,sigma=3, Per=70



joy1 N=3,sigma=sqrt(2), Per=50



joy1 N=7,sigma=3, Per=60

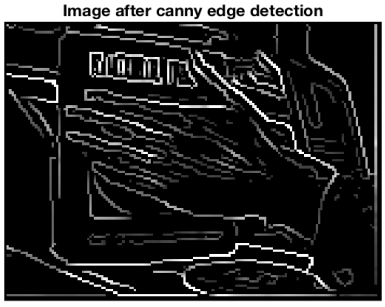


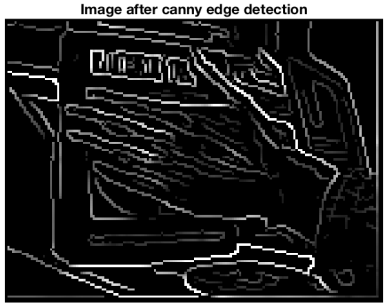
joy1 N=8,sigma=2, Per=60

joy1 Roberts

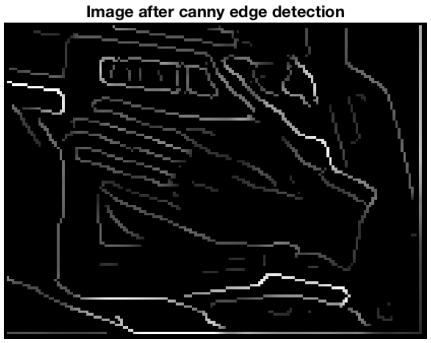
joy1 Sobel

Joy1 Zerocross

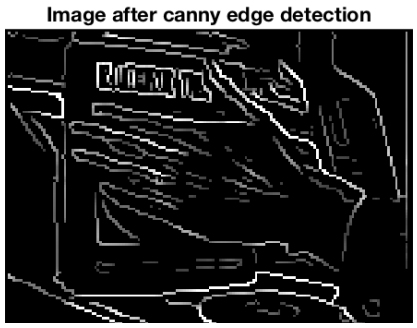




pointer1 N=2,sigma=2, Per=70



pointer1 N=7,sigma=1, Per=50



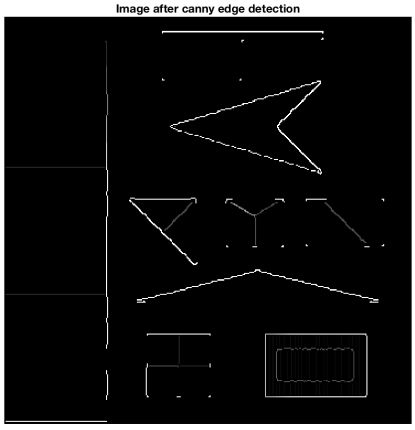
pointer1 N=8,sigma=sqrt(3), Per=60

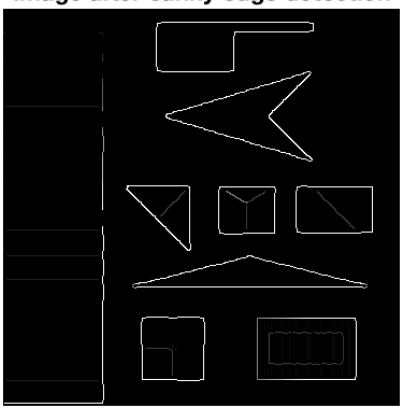
pointer1 N=10,sigma=0.9, Per=80

pointer1 Roberts

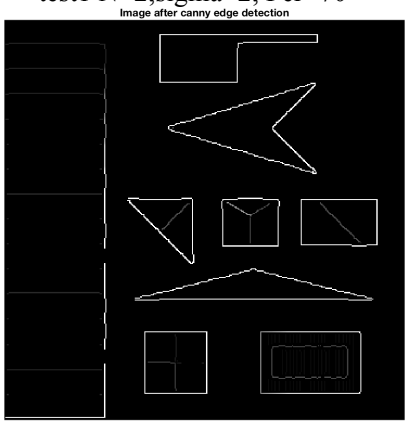
pointer1 Sobel

pointer1 Zerocross

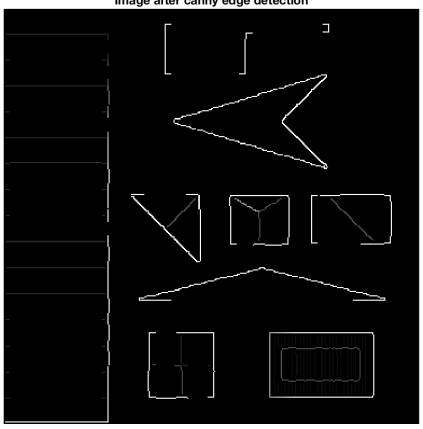




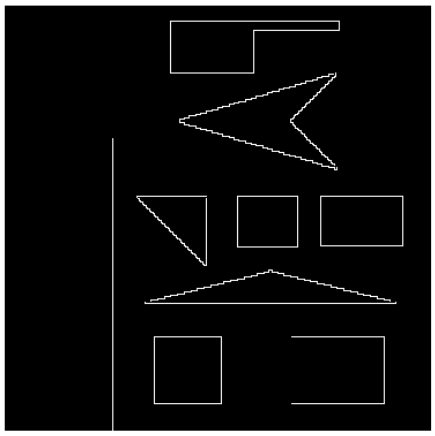
test1 N=2,sigma=2, Per=70



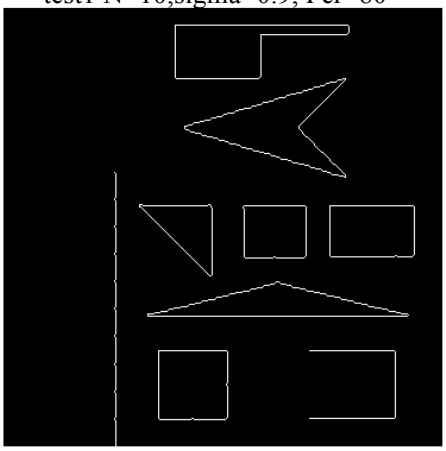
test1 N=7,sigma=sqrt(2), Per=60



test1 N=8,sigma=1, Per=50

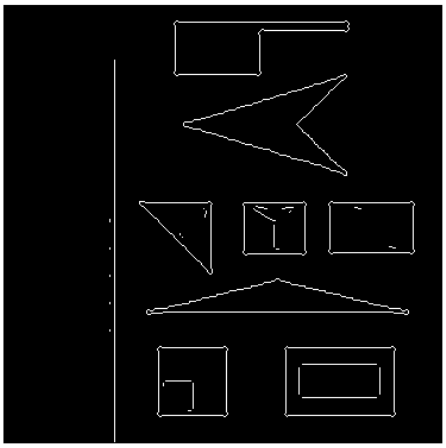


test1 N=10,sigma=0.9, Per=80

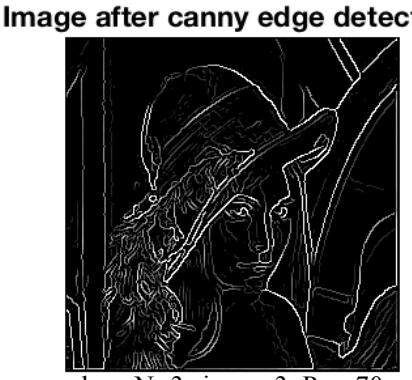


test1 Roberts

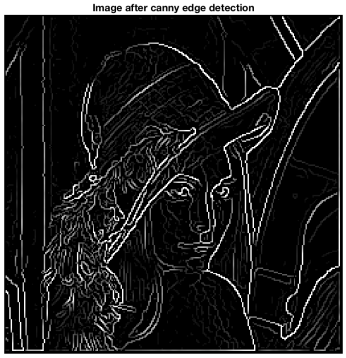
test1 Sobel



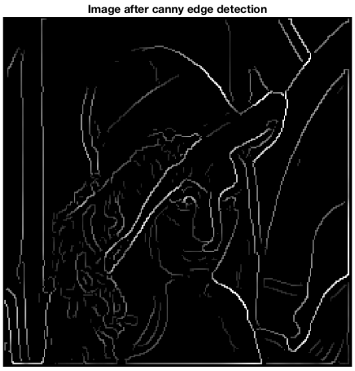
test1 Zerocross



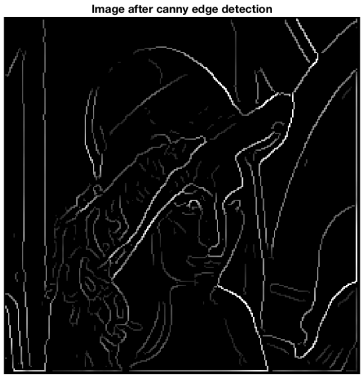
lena N=3,sigma=3, Per=70



lena N=3,sigma=sqrt(2), Per=50



lena N=7,sigma=3, Per=60



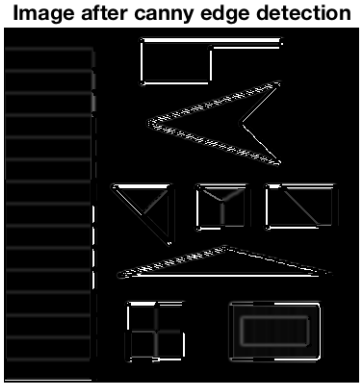
lena N=8,sigma=2, Per=60

lena Roberts

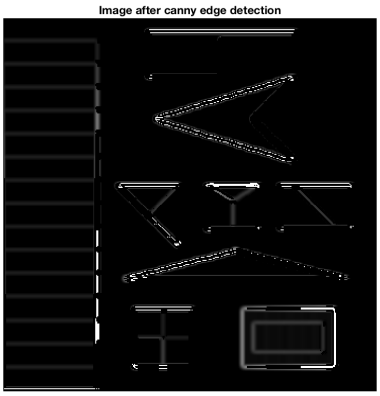
lena Sobel

lena Zerocross

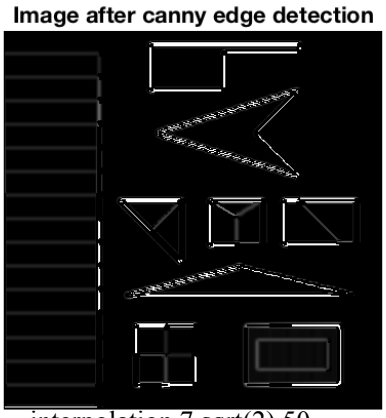
As for NonmaximaSupress function, I also implement it with interpolation method. And I select ‘test1.bmp’ to test it. The results are shown as followed.



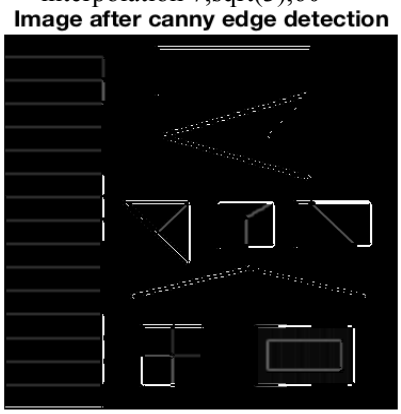
interpolation 7,sqrt(2),60



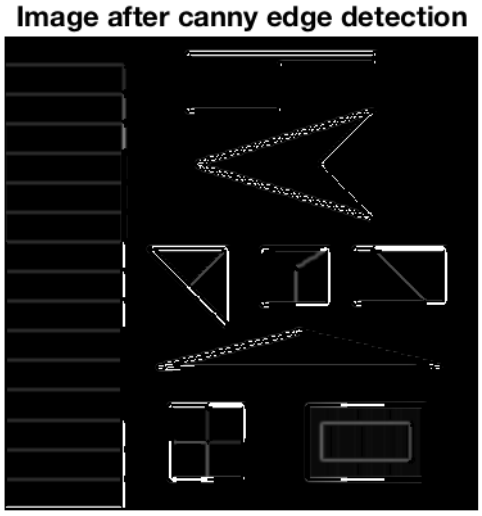
interpolation 7,sqrt(3),60



interpolation 7,sqrt(2),50



interpolation 3,sqrt(2),60



interpolation 8,1,80

2.2 Result Analysis

I used the canny edge detector on the four test images provided with four different input combinations for the size of Gaussian filter, the standard deviation of the filter, and the percent of pixels estimated to be non-edge pixels. From the results above, it could be safely draw conclusion that Canny edge detection performs better in edge detection. Especially when N=7 and sigma = sqrt(2), the results are essentially noise-free.

Roberts Cross operator determines edge more accurately but it is more sensitive to noise. Besides, it is obvious that the edge detecting through Robert operator is not smooth enough. When it comes to Sobel filter, if the noise spots are far away from the edge, the outcome of Sobel filter will become more distinct. And it is the same situation as the Zerocross filter. Compared with such operators, Canny edge detector is better-performed, the outcomes of which is more integrated than other operators. Moreover, it is not sensitive to noise.

2.3 Summary

Through this MP, which is the most time-consuming assignment ever since, I get a better understanding of the core of the Canny edge detector and its contribution to computer vision.

III. Matlab Codes

1. Using Interpolation method

```
function MP5(img_in, N, sigma, percentNonEdge)
    %Enable large recursion.
    set(0,'RecursionLimit',1000);

    % Gaussian smoothing.
    S = GaussSmoothing(img_in, N, sigma);

    % Image gradient.
    [Mag, Theta] = ImageGradient(S);

    % Threshold values.
    [T_low, T_high] = FindThreshold(Mag, percentNonEdge);

    % Nonmaxima suppression.
    Mag = NonmaximaSuppress(Mag, Theta, img_in);

    % Edge linking.
    Mag = EdgeLinking(T_low, T_high, Mag);

    % Display final result.
    figure, imshow(Mag);
    title('Image after canny edge detection');
end


function S = GaussSmoothing(I, N, Sigma)
    % Setup an NxN matrix.
    x = -floor(N / 2):floor(N / 2);
    y = -floor(N / 2):floor(N / 2);
    [X, Y] = meshgrid(x, y);

    % Calculate the Gaussian distribution on the matrix.
    grid = exp(-((X .^ 2) + (Y .^ 2)) / (2 * (Sigma ^ 2)));

    % Weight the distribution with a corner value k.
    k = (1 / grid(N, N));
    kernel_nonnormal = (grid * k);

    % Normalize the distribution.
    kernel = (kernel_nonnormal / (sum(sum(kernel_nonnormal))));

    % Read in the input image into a double matrix.
    mat = im2double(rgb2gray(imread(I)));
```

```
    % Convolute the kernel with the original image.
    S = conv2(mat, kernel, 'same');
end

function [Mag, Theta,Gx,Gy] = ImageGradient(img_in)
    % Setup the output matrices.
    Mag = zeros(size(img_in));
    Theta = zeros(size(img_in));

    % Apply the Sobel operator over the entire image.
    for i = 1:(size(img_in, 1) - 2)
        for j = 1:(size(img_in, 2) - 2)
            % Partial derivatives for pixel (i, j).
            Gx = ((img_in((i + 2), j) + (2 * img_in((i + 2), (j + 1))) + ...
                   img_in((i + 2), (j + 2))) - (img_in(i, j) + ...
                   (2 * img_in(i, (j + 1))) + img_in(i, (j + 2))));
            Gy = ((img_in(i, (j + 2)) + (2 * img_in((i + 1), (j + 2))) + ...
                   img_in((i + 2), (j + 2))) - (img_in(i , j) + ...
                   (2 * img_in((i + 1), j)) + img_in((i + 2), j)));

            % Magnitude of the pixel (i, j).
            Mag(i, j) = sqrt((Gx .^ 2) + (Gy .^ 2));

            % Direction of the pixel (i, j).
            Theta(i, j) = atan(Gy / Gx);
        end
    end

end

function [T_low, T_high] = FindThreshold(Mag, percentageOfNonEdge)

    % Convert the percentage into a decimal.
    p = (percentageOfNonEdge / 100);

    % Get the maximum value of the input matrix.
    mag_max = max(Mag(:));
    if (mag_max > 0)
        % Normalize the input matrix.
        mag_norm = (Mag / mag_max);
    else
        mag_norm = Mag;
    end

    % Create a histogram of the normalized matrix.
    mag_hist = imhist(mag_norm, 64);

    % Calculate the threshold values.
    T_high = (find((cumsum(mag_hist) > (p * size(Mag, 1) * ...
                    size(Mag, 2))), 1, 'first') / 64);
    T_low = (T_high * 0.5);

end

function Mag_out = NonmaximaSuppress(Mag, Theta, im)
    % Perform non-maximum suppression using interpolation
    % Setup the output matrix.
    [m, n] = size(Mag);
    Mag_out = zeros(size(Mag));
    %X=[-1,0,+1 ;-1,0,+1 ;-1,0,+1];
    %Y=[-1,-1,-1 ;0,0,0 ;+1,+1,+1];
    %x = [0 1];
    Gx = SobelFilter(imread(im), 'x');
    Gy = SobelFilter(imread(im), 'y');
    Gx = imgaussfilt(Gx,1);
    Gy = imgaussfilt(Gy,1);

    % Apply nonmaxima suppression over the entire input image.
    for i = 2:(m-1)
        for j = 2:(n-1)
             if (Theta(i,j)>=0 && Theta(i,j)<=45) || ...
                    (Theta(i,j)<-135 && Theta(i,j)>=-180)
                yBot = [Mag(i,j+1) Mag(i+1,j+1)];
                yTop = [Mag(i,j-1) Mag(i-1,j-1)];
                x_est = abs(Gy(i,j)/Mag(i,j)); % y
                if (Mag(i,j) >= ((yBot(2)-yBot(1))*x_est+yBot(1)) && ...
                    Mag(i,j) >= ((yTop(2)-yTop(1))*x_est+yTop(1))) % interpolation
```

```
                    Mag_out(i,j)= Mag(i,j);
                else
                    Mag_out(i,j)=0;
                end
            elseif (Theta(i,j)>45 && Theta(i,j)<=90) || ...
                    (Theta(i,j)<-90 && Theta(i,j)>=-135)
                yBot = [Mag(i+1,j) Mag(i+1,j+1)];
                yTop = [Mag(i-1,j) Mag(i-1,j-1)];
                x_est = abs(Gx(i,j)/Mag(i,j));
                if (Mag(i,j) >= ((yBot(2)-yBot(1))*x_est+yBot(1)) && ...
                    Mag(i,j) >= ((yTop(2)-yTop(1))*x_est+yTop(1)))
                    Mag_out(i,j)= Mag(i,j);
                else
                    Mag_out(i,j)=0;
                end
            elseif (Theta(i,j)>90 && Theta(i,j)<=135) || ...
                    (Theta(i,j)<-45 && Theta(i,j)>=-90)
                yBot = [Mag(i+1,j) Mag(i+1,j-1)];
                yTop = [Mag(i-1,j) Mag(i-1,j+1)];
                x_est = abs(Gx(i,j)/Mag(i,j));
                if (Mag(i,j) >= ((yBot(2)-yBot(1))*x_est+yBot(1)) && ...
                    Mag(i,j) >= ((yTop(2)-yTop(1))*x_est+yTop(1)))
                    Mag_out(i,j)= Mag(i,j);
                else
                    Mag_out(i,j)=0;
                end
            elseif (Theta(i,j)>135 && Theta(i,j)<=180) || ...
                    (Theta(i,j)<0 && Theta(i,j)>=-45)
                yBot = [Mag(i,j-1) Mag(i+1,j-1)];
                yTop = [Mag(i,j+1) Mag(i-1,j+1)];
                x_est = abs(Gx(i,j)/Mag(i,j));
                if (Mag(i,j) >= ((yBot(2)-yBot(1))*x_est+yBot(1)) && ...
                    Mag(i,j) >= ((yTop(2)-yTop(1))*x_est+yTop(1)))
                    Mag_out(i,j)= Mag(i,j);
                else
                    Mag_out(i,j)=0;
                end
            end
        end
    end

end

function E = EdgeLinking(T_low, T_high, Mag)

    % Setup the output matrix and the two threshold matrices.
    [m, n] = size(Mag);
    E = zeros(m, n);
    Mag2 = Mag;
    Mag(Mag < T_low) = 0;
    Mag2(Mag2 < T_high) = 0;

    % Operate on each element of the high threshold matrix.
    for i = 1:m
        for j = 1:n
            if (Mag2(i, j) ~= 0)
                % Recursive call to check 8-neighbors.
                CheckCandidates(i, j, T_low, T_high, Mag, Mag2, E);
            end
        end
    end

    % Set the output to the modified high threshold matrix.
    E = Mag2;

end

function candidate = CheckCandidates(i, j, T_low, T_high, Mag, Mag2, E)

    % Canary value for successful completion.
    candidate = 0;

    % Opeate on each 8-neighbor of the input pixel.
    for k = (i - 1):(i + 1)
        % Ignore pixels outside the image.
        if ((k < 1) || (k > size(Mag, 1)))
            continue;
        end
```

```
        for l = (j - 1):(j + 1)
            % Ignore pixels outside the image, the starting pixel,
            % and pixels that have already been checked.
            if (((l < 1) || (l > size(Mag, 2))) || ...
                ((i == k) && (l == j)) || (E(k, l) == 1))
                continue;
            end

            % Mark pixels above the high threshold, and cascade the match.
            if (Mag(k, l) > T_high)
                Mag2(k, l) = Mag(k, l);
                E(k, l) = 1;
                candidate = 1;
                return;

            % Recursively check pixels above the low threshold.
            elseif (Mag(k, l) > T_low)
                % Mark this pixel as visited.
                E(k, l) = 1;
                % Perform the recurisve call.
                candidate = CheckCandidates(k, l, T_low, T_high, Mag, Mag2, E);

                % Cascade the call stack if a match is found.
                if (candidate == 1)
                    Mag2(k, l) = Mag(k, l);
                    return;
                end
            end
        end
    end

end

function[A] = SobelFilter(A, filterDirection)
    switch filterDirection
        case 'x'
            Gx = [-1 0 +1; -2 0 +2; -1 0 +1];
            A = imfilter(A, double(Gx), 'conv', 'replicate');
        case 'y'
            Gy = [-1 -2 -1; 0 0 0; +1 +2 +1];
            A = imfilter(A, double(Gy), 'conv', 'replicate');
        otherwise
            error('Bad filter direction - try inputs ''x'' or ''y''');
    end
end
```

2. Without Interpolation

```
function mp6(img_in, N, sigma, percentNonEdge)
    %Enable large recursion.
    set(0,'RecursionLimit',1000);

    % Gaussian smoothing.
    S = GaussSmoothing(img_in, N, sigma);

    % Image gradient.
    [Mag, Theta] = ImageGradient(S);

    % Threshold values.
    [T_low, T_high] = FindThreshold(Mag, percentNonEdge);

    % Nonmaxima suppression.
    Mag = NonmaximaSuppress(Mag, Theta);

    % Edge linking.
    Mag = EdgeLinking(T_low, T_high, Mag);

    % Display final result.
    figure, imshow(Mag);
    title('Image after canny edge detection');
end


function S = GaussSmoothing(I, N, Sigma)
    % Setup an NxN matrix.
```

```
    x = -floor(N / 2):floor(N / 2);
    y = -floor(N / 2):floor(N / 2);
    [X, Y] = meshgrid(x, y);

    % Calculate the Gaussian distribution on the matrix.
    grid = exp(-((X .^ 2) + (Y .^ 2)) / (2 * (Sigma ^ 2)));

    % Weight the distribution with a corner value k.
    k = (1 / grid(N, N));
    kernel_nonnormal = (grid * k);

    % Normalize the distribution.
    kernel = (kernel_nonnormal / (sum(sum(kernel_nonnormal))));

    % Read in the input image into a double matrix.
    mat = im2double(rgb2gray(imread(I)));

    % Convolute the kernel with the original image.
    S = conv2(mat, kernel, 'same');
end

function [Mag, Theta,Gx,Gy] = ImageGradient(img_in)
    % Setup the output matrices.
    Mag = zeros(size(img_in));
    Theta = zeros(size(img_in));

    % Apply the Sobel operator over the entire image.
    for i = 1:(size(img_in, 1) - 2)
        for j = 1:(size(img_in, 2) - 2)
            % Partial derivatives for pixel (i, j).
            Gx = ((img_in((i + 2), j) + (2 * img_in((i + 2), (j + 1))) + ...
                   img_in((i + 2), (j + 2))) - (img_in(i, j) + ...
                   (2 * img_in(i, (j + 1))) + img_in(i, (j + 2))));
            Gy = ((img_in(i, (j + 2)) + (2 * img_in((i + 1), (j + 2))) + ...
                   img_in((i + 2), (j + 2))) - (img_in(i , j) + ...
                   (2 * img_in((i + 1), j)) + img_in((i + 2), j)));

            % Magnitude of the pixel (i, j).
            Mag(i, j) = sqrt((Gx .^ 2) + (Gy .^ 2));

            % Direction of the pixel (i, j).
            Theta(i, j) = atan(Gy / Gx);
        end
    end

end

function [T_low, T_high] = FindThreshold(Mag, percentageOfNonEdge)

    % Convert the percentage into a decimal.
    p = (percentageOfNonEdge / 100);

    % Get the maximum value of the input matrix.
    mag_max = max(Mag(:));
    if (mag_max > 0)
        % Normalize the input matrix.
        mag_norm = (Mag / mag_max);
    else
        mag_norm = Mag;
    end

    % Create a histogram of the normalized matrix.
    mag_hist = imhist(mag_norm, 64);

    % Calculate the threshold values.
    T_high = (find((cumsum(mag_hist) > (p * size(Mag, 1) * ...
                    size(Mag, 2))), 1, 'first') / 64);
    T_low = (T_high * 0.5);

end

function Mag_out = NonmaximaSuppress(Mag, Theta)

    % Setup the output matrix.
    [m, n] = size(Mag);
    Mag_out = zeros(size(Mag));

    % Apply nonmaxima suppression over the entire input image.
```

```
    for i = 2:(m-1)
        for j = 2:(n-1)
            pixel = Mag(i, j);
            dir = Theta(i, j);
            % Case 0 from the textbook (-pi/8 to pi/8).
            if ((dir > degtorad(-22.5)) && (dir <= degtorad(22.5)))
                if ((pixel > Mag((i - 1), j)) && (pixel > Mag((i + 1), j)))
                    Mag_out(i, j) = pixel;
                end
            % Case 1 from the textbook (??pi/8 to 3pi/8).
            elseif ((dir > degtorad(22.5)) && (dir <= degtorad(67.5)))
                if ((pixel > Mag((i + 1), (j + 1))) && ...
                    (pixel > Mag((i - 1), (j - 1))))
                    Mag_out(i, j) = pixel;
                end
            % Case 3 from the textbook (-3pi/8 to -pi/8).
            elseif ((dir > degtorad(-67.5)) && (dir <= degtorad(-22.5)))
                if ((pixel > Mag((i + 1), (j - 1))) && ...
                    (pixel > Mag((i - 1), (j + 1))))
                    Mag_out(i, j) = pixel;
                end
            % Case 2 from the textbook (-3pi/8 to -pi/2 and 3pi/8 to pi/2).
            elseif ((dir < degtorad(-67.5)) || (dir > degtorad(67.5)))
                if ((pixel > Mag(i, (j + 1))) && (pixel > Mag(i, (j - 1))))
                    Mag_out(i, j) = pixel;
                end
            end
        end
    end

end

function E = EdgeLinking(T_low, T_high, Mag)

    % Setup the output matrix and the two threshold matrices.
    [m, n] = size(Mag);
    E = zeros(m, n);
    Mag2 = Mag;
    Mag(Mag < T_low) = 0;
    Mag2(Mag2 < T_high) = 0;

    % Operate on each element of the high threshold matrix.
    for i = 1:m
        for j = 1:n
            if (Mag2(i, j) ~= 0)
                % Recursive call to check 8-neighbors.
                CheckCandidates(i, j, T_low, T_high, Mag, Mag2, E);
            end
        end
    end

    % Set the output to the modified high threshold matrix.
    E = Mag2;

end

function candidate = CheckCandidates(i, j, T_low, T_high, Mag, Mag2, E)

    % Canary value for successful completion.
    candidate = 0;

    % Opeate on each 8-neighbor of the input pixel.
    for k = (i - 1):(i + 1)
        % Ignore pixels outside the image.
        if ((k < 1) || (k > size(Mag, 1)))
            continue;
        end

        for l = (j - 1):(j + 1)
            % Ignore pixels outside the image, the starting pixel,
            % and pixels that have already been checked.
            if (((l < 1) || (l > size(Mag, 2))) || ...
                ((i == k) && (l == j)) || (E(k, l) == 1))
                continue;
            end

            % Mark pixels above the high threshold, and cascade the match.
            if (Mag(k, l) > T_high)
```

```
                Mag2(k, l) = Mag(k, l);
                E(k, l) = 1;
                candidate = 1;
                return;

            % Recursively check pixels above the low threshold.
            elseif (Mag(k, l) > T_low)
                % Mark this pixel as visited.
                E(k, l) = 1;
                % Perform the recurisve call.
                candidate = CheckCandidates(k, l, T_low, T_high, Mag, Mag2, E);

                % Cascade the call stack if a match is found.
                if (candidate == 1)
                    Mag2(k, l) = Mag(k, l);
                    return;
                end
            end
        end
    end

end
```