EDGE DETECTION WITH FIRST AND SECOND ORDER DERIVATIVE FILTERS

NOVEMBER 2, 2021

TA's: Muhammed Zemzemoğlu and Mehmet Emin Mumcuoğlu

Name of Student: Moses Chuka Ebere
Student Number: 31491
Course: Computer Vision (EE 417)

TABLE OF CONTENTS

INTRODUCTION

In this laboratory, first and second order derivative filters are used for edge detection in images.

Edge Detection

This is an image processing technique applied to pinpoint the boundaries or edges of the objects contained in an image. Edge detection is usually performed using first or second derivative filters. When first derivative filters are used, edges are found at local maxima or minima so simply area where there's a dramatic/sharp variation in intensity values.

Some common first order derivative filters include Prewitt, Sobel, Roberts, etc.

Some common second order derivative filters include Difference of Gaussian (DoG) and Laplacian of Gaussian (LoG).

This report explores the Prewitt, Sobel, and LoG filters for edge detection.

EXPLANATION OF METHODS

First Derivative

Prewitt Filter

This is a first derivative filter developed by Judith M. S. Prewitt in 1970. The filter was purposefully designed to combine smoothing and derivative. The filter is made up of 3 one-dimensional central difference filters stacked together. Prewitt stacked the 1D filters to mitigate their high noise sensitivities. The X and Y kernel filters are:

-1	0	1
-1	0	1
-1	0	1

X Filter

-1	-1	-1
0	0	0
1	1	1

Y Filter

The X-filter performs smoothing in the vertical direction and derivative in the horizontal direction (to detect vertical edges). The Y-filter is simply a transpose of the X-filter.

To obtain a Prewitt filtered image, the Sobel filter is used to scan the image, and, at every instance, the convolution of the image window with Prewitt filter is taken.

In MATLAB, the above is performed using the following function that takes an image and a threshold (to obtain the edges) as inputs, and returns the Prewitt X & Y filtered images, the Prewitt Gradient, and the Prewitt Edges:

```
function [output1, output2, output3, output4] =
lab3prewitt(img, t)

% The row, column, and channels of the image are obtained
% along with the cardinality of the image.
[r, c, ch] = size(img);
Card = r*c;

% This is added in case the image introduced is an RGB
%image.
```

```
% It functions to convert it to a gray-scale image.
if (ch == 3)
    img = rgb2gray(img);
end

%Convert the image to double before performing any
%mathematical operation
I = double(img);

% Create the sobel filters; the x-filter will be used to
%create the vertically filtered image, while the y-filter
% will be used to create the horizontally filtered image.
x_filt = [-1 0 1; -1 0 1; -1 0 1];
y_filt = [-1 -1 -1; 0 0 0; 1 1 1];

% The prewitt filter has a size (2k+1)x(2k+1), where k=1;
k = 1;

% To ensure that we don't end up with images of different
% sizes relative to the original images, we initialize our
% final images with the same number of rows and columns as
% the original ones.
Ihor = zeros(r, c);
Iver = zeros(r, c);
Igrad = zeros(r, c);
Iedge = zeros(r, c);

% Use nested for-loops to create a window for scanning the
% image.
for i=(k+1):1:r-k
    for j=(k+1):1:c-k
        %the window will go from -k to +k
        wp = I(i-k:i+k, j-k:j+k);
        % Perform convolution using the previously created
% masks.
        Iver(i,j) = sum(wp(:).*x_filt(:));
        Ihor(i,j) = sum(wp(:).*y_filt(:));
        Igrad(i,j) = sqrt((Iver(i,j)^2 + Ihor(i,j)^2));

        Iedge(i,j) = Igrad(i,j);
        % Use the threshold to determine which pixels
%qualify as edges.
```

```
        if Iedge(i,j) >= t
            Iedge(i,j) = 255;
        else
            Iedge(i,j) = 0;
        end
    end
end

% Convert the resulting images to unsigned 8-bit images
%and return results.
output1 = uint8(Ihor);
output2 = uint8(Iver);
output3 = uint8(Igrad);
output4 = uint8(Iedge);

end
```

In the main script, the following code calls the Prewitt filter function and applies it on the input image, before displaying the results.

```
%% Prewitt Filter
% Read the image to be preprocessed
a = imread('peppers.png');
% The threshold is a user-defined variable to obtain the
%edges.
thr = 150;

% We call the prewitt filter function. y represents the
%Sobel Y-filtered image, x - Sobel X-fitered image, gr -
%Sobel Gradient, and ed - Sobel Edge
[y, x, gr, ed] = lab3prewitt (a, thr);

%The original image,the Prewitt X & Y filtered images, the
% Prewitt Gradient, and the Prewitt Edges are displayed.
figure
subplot(2, 3, 1)
imshow(a)
title('Original Image')

subplot(2, 3, 2)
imshow(x)
```

```
title('Prewitt X Filtered Image');

subplot(2, 3, 3)
imshow(y)
title('Prewitt Y Filtered Image');

subplot(2, 3, 5)
imshow(gr)
title('Prewitt Gradient')

subplot(2, 3, 6)
imshow(ed)
title('Prewitt Edges')
```

First Derivative

Sobel Filter

The Sobel filter or Sobel-Feldman operator is a first derivative operator that uses special filter kernels there perform simultaneous derivative and smoothing on an image. It does this by the process of convolution. Each filter is made up of 3 one-dimensional central difference filters stacked together. Sobel and Feldman stacked the 1D filters to mitigate their high noise sensitivities. The X and Y kernel filters are:

-1	0	1
-2	0	2
-1	0	1

X Filter

-1	-2	-1
0	0	0
1	2	1

Y Filter

The X-filter performs smoothing in the vertical direction and derivative in the horizontal direction (to detect vertical edges). The Y-filter is simply a transpose of the X-filter.

To obtain a Sobel filtered image, the Sobel filter is used to scan the image, and, at every instance, the convolution of the image window with the Sobel filter is taken.

In MATLAB, the following function was written. The function accepts an image and a threshold (to obtain the edges) as inputs, and returns the Sobel X & Y filtered images, the Sobel Gradient, and the Sobel Edges.

```
function [output1, output2, output3, output4] =
lab3sobel(img, t1)

% The row, column, and channels of the image are obtained
% along with the cardinality of the image.
[r, c, ch] = size(img);
Card = r*c;

% This is added in case the image introduced is an RGB
%image. It functions to convert it to a gray-scale image.
if (ch == 3)
    img = rgb2gray(img);
end

%Convert the image to double before performing any
%mathematical operation
I = double(img);

% Create the sobel filters; the x-filter will be used to
%create the vertically filtered image, while the y-filter
%will be used to create the horizontally filtered image.
x_filt = [-1 0 1; -2 0 2; -1 0 1];
y_filt = [-1 -2 -1; 0 0 0; 1 2 1];

% The sobel filter has a size (2k+1)x(2k+1), where k = 1;
k = 1;

% To ensure that we don't end up with images of different
%sizes relative to the original images, we initialize our
% final images with the same number of rows and columns as
% the original ones.
Ihor = zeros(r, c);
Iver = zeros(r, c);
Igrad = zeros(r, c);
Iedge = zeros(r, c);
```

```
% Use nested for-loops to create a window for scanning the
% image.
for i=(k+1):1:r-k
    for j=(k+1):1:c-k
        %the window will go from -k to +k
        wp = I(i-k:i+k, j-k:j+k);
        % Perform convolution using the previously created
% masks.
        Iver(i,j) = sum(wp(:).*x_filt(:));
        Ihor(i,j) = sum(wp(:).*y_filt(:));
        Igrad(i,j) = sqrt((Iver(i,j)^2 + Ihor(i,j)^2));

        Iedge(i,j) = Igrad(i,j);
        % Use the threshold to determine which pixels
%qualify as edges.
        if Iedge(i,j) >= t1
            Iedge(i,j) = 255;
        else
            Iedge(i,j) = 0;
        end
    end
end

% Convert the resulting images to unsigned 8-bit images
%and return the results.
output1 = uint8(Ihor);
output2 = uint8(Iver);
output3 = uint8(Igrad);
output4 = uint8(Iedge);

end
```

The following code calls the Sobel filter function and applies it on the input image, before displaying the results.

```
%% Sobel Filter
% Read the image to be preprocessed
b = imread('peppers.png');
% The threshold is a user-defined variable to obtain the
%edges.
th1 = 150;
```

```
% We call the Sobel filter function. y1 represents the
%Sobel Y-filtered image, x1 - Sobel X-fitered image, gr1 -
%Sobel Gradient, and ed1 - Sobel Edge
[y1, x1, gr1, ed1] = lab3sobel(b, th1);

%The original image,the Sobel X & Y filtered images, the
%Sobel Gradient, and the Sobel Edges are displayed.
figure
subplot(2, 3, 1)
imshow(b)
title('Original Image')

subplot(2, 3, 2)
imshow(x1)
title('Sobel X Filtered Image');

subplot(2, 3, 3)
imshow(y1)
title('Sobel Y Filtered Image');

subplot(2, 3, 5)
imshow(gr1)
title('Sobel Gradient')

subplot(2, 3, 6)
imshow(ed1)
title('Sobel Edges')
```

Second Derivative

Laplacian of Gaussian

The Laplacian measures the second spatial derivative of an image. It highlights areas of an image that experience a sharp variation in intensity values. Here, the image is first smoothened using a Gaussian filter, and the zero crossings (which correspond to edges) are found by taking the Laplacian of the smoothened image.

$$O(x,y) = \nabla^2[I(x,y) * G(x,y)] \text{ Or } O(x,y) = \nabla^2 G(x,y) * I(x,y)$$

Where $\nabla^2$is the Laplacian (which produces a scalar), I(x,y) is the original image, and G(x,y) is the smoothed image;

The Laplacian operator is:

0	1	0
1	-4	1
0	1	0

In MATLAB, the following function was written for the LoG filtering process. The function accepts an image as an input, and returns the LoG filtered version.

```
function [output1, output2] = lab3log(img)

% First of all, smoothen the image using Gaussian
filtering.
img = lab3gaussfilt(img);

% The row, column, and channels of the image are obtained
along with the cardinality of the image.
[r, c, ch] = size(img);
Card = r*c;

% This is added in case the image introduced is an RGB
image.
% It functions to convert it to a gray-scale image.
if (ch == 3)
    img = rgb2gray(img);
end

%Convert the image to double before performing any
%mathematical operation
I = double(img);

% Create the sobel filters; the x-filter will be used to
create the
% vertically filtered image, while the y-filter will be
used to create the
% horizontally filtered image.
LoG_filt = [0 1 0; 1 -4 1; 0 1 0];
```

```
% The prewitt filter has a size (2k+1)x(2k+1), where k =
1;
k = 1;

% To ensure that we don't end up with images of different
sizes relative
% to the original images, we initialize our final images
with the same number of rows
% and columns as the original ones.
ILoG = zeros(r, c);

% Use nested for-loops to create a window for scanning the
image.
for i=(k+1):1:r-k
    for j=(k+1):1:c-k
        %the window will go from -k to +k
        wp = I(i-k:i+k, j-k:j+k);
        % Perform convolution using the previously created
masks.
        ILoG(i,j) = sum(wp(:).*LoG_filt(:));
    end
end

% Create a row vector having the same number of columns as
our original image.
Inew = zeros(1, c);
ii = 1;

% This is used to select the line segment.
% The following syntax is just a trivial inclusion based
on our observation of where an
% edge will be present.
if mod(r, 2) == 0
    r1 = r/2;
else
    r1 = (r+1)/2;
end

% This for-loop is included just in case we want the
gradient profile along
% the entire width of the image.
```

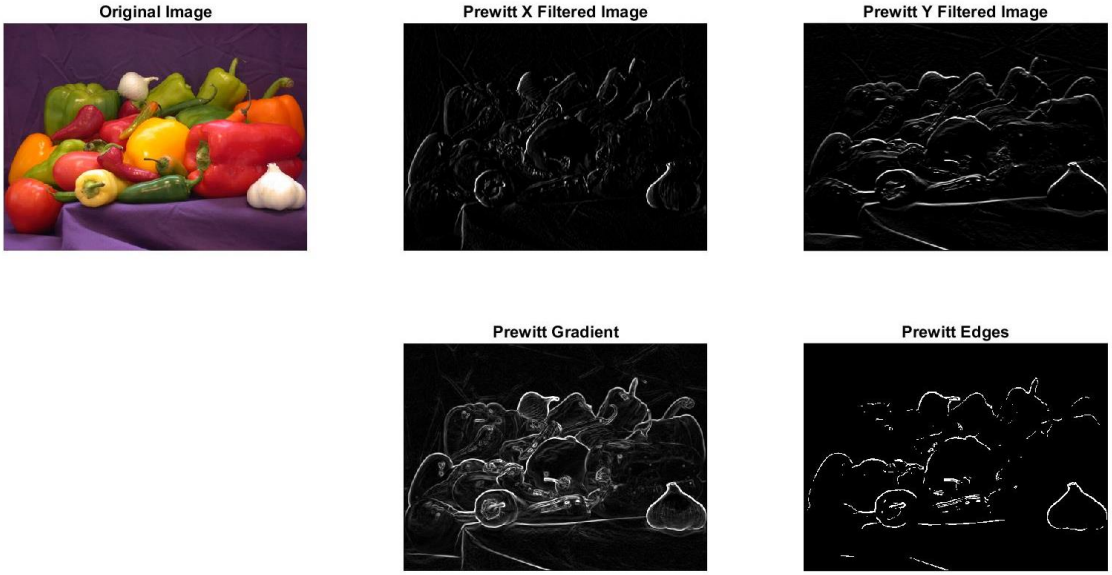
```
for c1 = 1:1:c
    Inew(ii,c1) = ILoG(r1, c1);
end

% Convert the resulting images to unsigned 8-bit images
and return the
% results.
output1 = uint8(ILoG);

% Since uint8 doesn't include negative values, we return
the array for the
% gradient profile as a double.
output2 = Inew;

end
```

The following code calls the LoG filter function and applies it on the input image, before displaying the results.

```
%% Laplacian of Gaussian
% Read the image to be preprocessed
c = imread('Object_contours.jpg');

% We call the LoG function. v represents the LoG filtered
% image, while g represents the gradient profile.
[v, g] = lab3log(c);

%The original image, the LoG filtered image, and a
%gradient profile are displayed.
figure
subplot(1, 3, 1)
imshow(c)
title('Original Image')

subplot(1, 3, 2)
imshow(v)
title('LOG Filtered Image')

subplot(1, 3, 3)
plot(30:1:60, g(:, 30:1:60), 'LineWidth', 2)
title('Gradient Profile'); xlabel('Pixel Location');
ylabel('Gradient Magnitude');
```

RESULTS

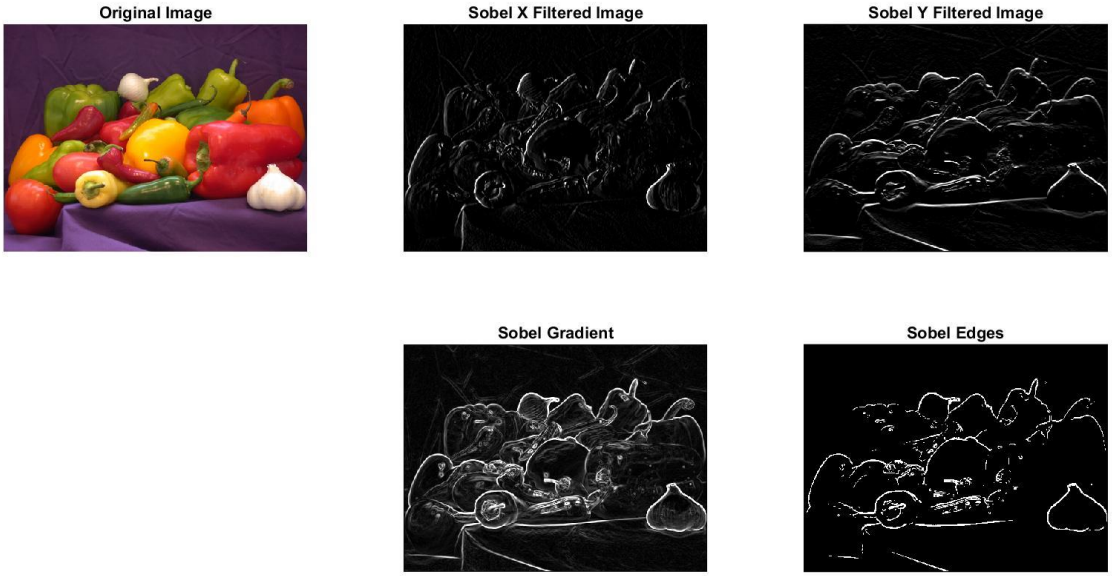
Prewitt Operator



The Prewitt filter does a good job of obtaining the edges in the image. Since it is based on convolution, it is suitable for real-time applications.

-

Sobel Filter

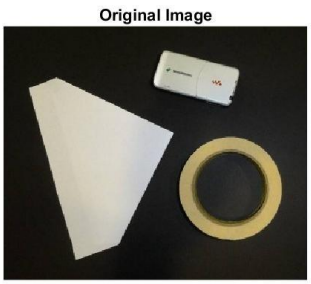


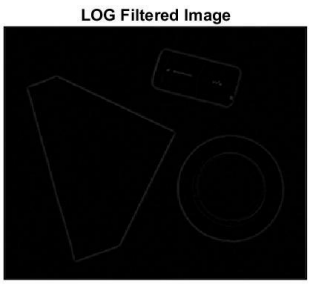
- The x-filter was used to obtain the Sobel vertical filtered image.
- The y-filter was used to obtain the Sobel horizontal filtered image.

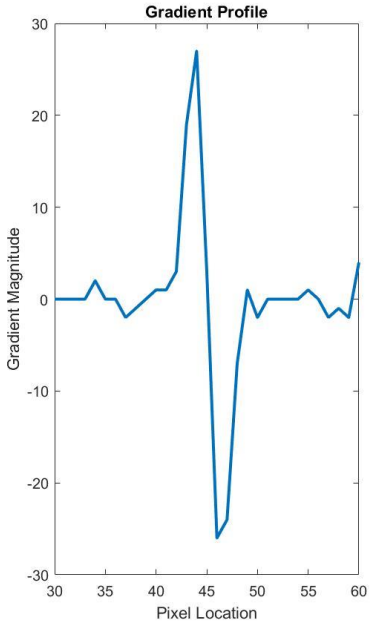
Sobel filter is good for edge detection. It wouldn't be an ideal method for smoothing since the scale applied in the filter are not uniform.

- Since it is based on convolution, it is suitable for real-time applications.

Laplacian of Gaussian Filter





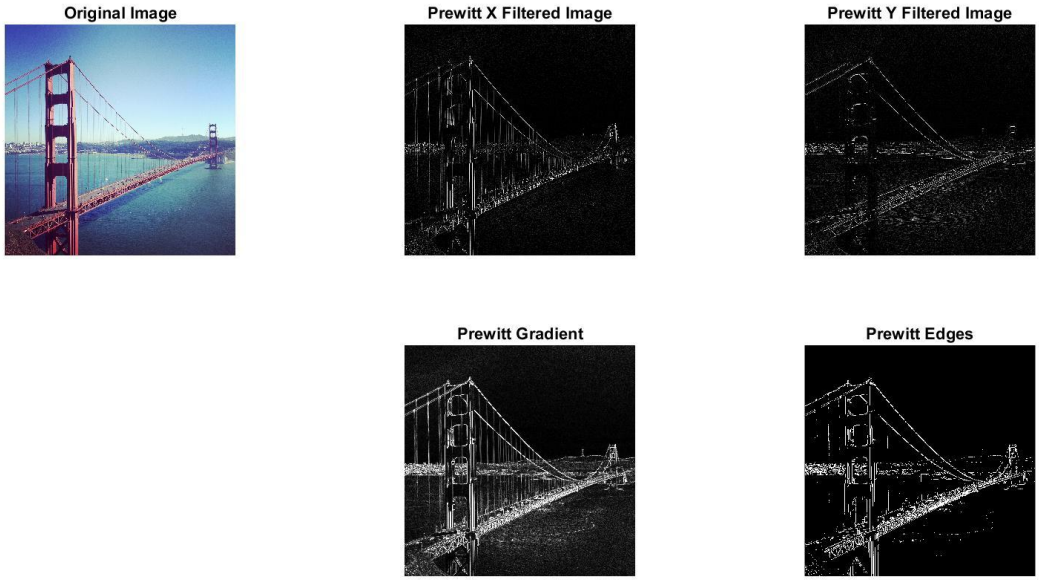


Notice: The edges above really faint, so we could futher process the LoG filtered image to make the edges more visible. This could be done by normalizing the image to fit a range, e.g., from 0 to 250.

MULTIPLE RESULTS

Prewitt Operator

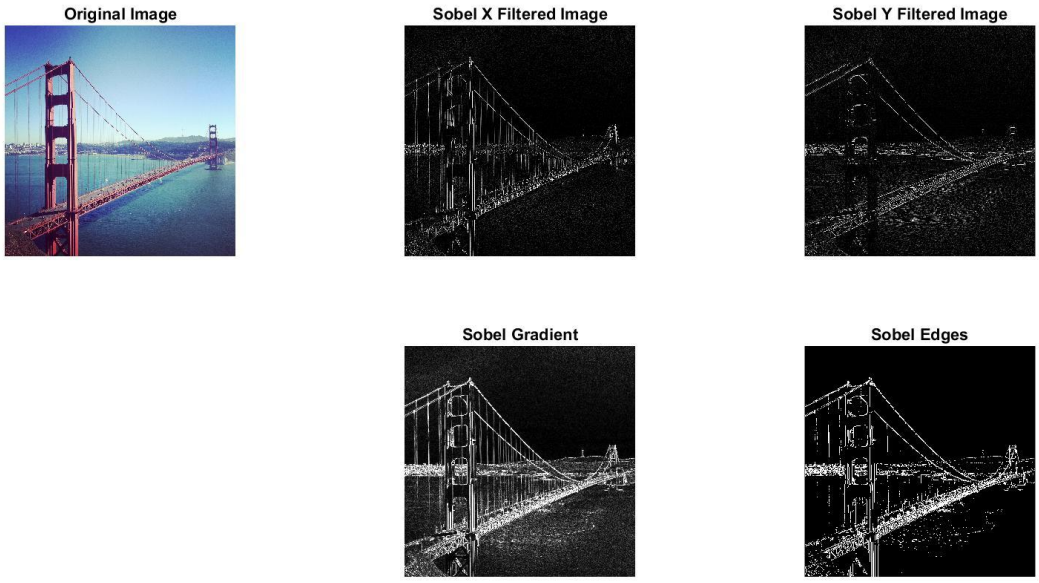
Let's apply the Prewitt operator on an RGB image of the Golden State Bridge:



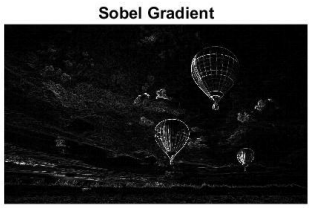
Notice: The Prewitt operator was able to obtain a nice gradient of the image. For the Prewitt Edges, if we lower our threshold, we should be able to obtain more edges.

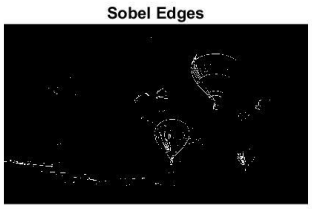
Sobel Filter

Applying the Sobel filter on the same Golden State Bridge:



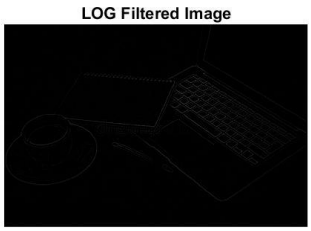
P.S. A comparison of the results from Prewitt and Sobel is discussed in the next section.

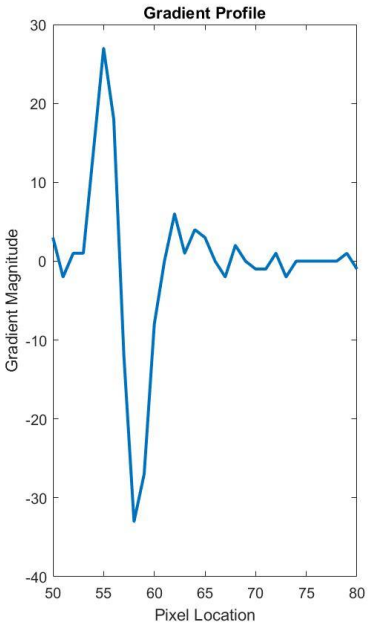




Laplacian of Gaussian Filter

Apply the LoG filter on an RGB image:





QUESTIONS

Zero Crossing on a LoG Filtered Image

The Sigma Filter computes an average based a predefined intensity range enforced by a user input σ. The equation for the Sigma Filter is as follows:
Zero Crossing is used to pinpoint edges. It simply looks for areas in a LoG filtered image where the Laplacian goes from positive to negative and vice versa.

In MATLAB, the following function was written for the Zero Crossing. The function accepts an image and return a LoG filtered image and the Zero Crossing image in both uint8 and double formats.

```
function [output1, output2, output3, output4] =
lab3ZeroCrossing(img)

% First of all, smoothen the image using Gaussian
filtering.
img = lab3gaussfilt(img);

% The row, column, and channels of the image are obtained
along with the cardinality of the image.
[r, c, ch] = size(img);
Card = r*c;

% This is added in case the image introduced is an RGB
image.
% It functions to convert it to a gray-scale image.
if (ch == 3)
    img = rgb2gray(img);
end

%Convert the image to double before performing any
%mathematical operation
I = double(img);

% Threshold
t = 0.5 * mean(abs(img(:)));
```

```
% Create the sobel filters; the x-filter will be used to
create the
% vertically filtered image, while the y-filter will be
used to create the
% horizontally filtered image.
LoG_filt = [0 1 0; 1 -4 1; 0 1 0];

% The prewitt filter has a size (2k+1)x(2k+1), where k = 1;
k = 1;

% To ensure that we don't end up with images of different
%sizes relative to the original images, we initialize our
%final images with the same number of rows and columns as
%the original ones.
ILoG = zeros(r, c);
IZC = zeros(r, c);

% Use nested for-loops to create a window for scanning the
%image.
for i=(k+1):1:r-k
    for j=(k+1):1:c-k
        %the window will go from -k to +k
        wp = I(i-k:i+k, j-k:j+k);
        % Perform convolution using the previously created
masks.
        ILoG(i,j) = sum(wp(:).*LoG_filt(:));
    end
end

% Now, we find the zero crossings.
for i= 2:r-1
    for j = 2:c-1
        if ILoG(i,j)>0
            if ((ILoG(i,j+1) >= 0 && ILoG(i,j-1) < 0) ||
(ILoG(i,j+1) < 0 &&...
                    ILoG(i,j-1) >= 0)) && abs(ILoG(i,j+1) -
ILoG(i,j-1)) > t
                IZC(i,j) = ILoG(i,j+1);
            elseif ((ILoG(i+1,j) >= 0 && ILoG(i-1,j) < 0)
|| (ILoG(i+1,j) < 0 &&...
                    ILoG(i-1,j) >= 0)) && abs(ILoG(i+1,j) -
ILoG(i-1,j)) > t
```

```
                IZC(i,j) = I(i,j+1);
            elseif ((ILoG(i+1,j+1) >= 0 && ILoG(i-1,j-1) <
0) || (ILoG(i+1,j+1) < 0 &&...
                    ILoG(i-1,j-1) >= 0)) &&
abs(ILoG(i+1,j+1) - ILoG(i-1,j-1)) > t
                IZC(i,j) = ILoG(i,j+1);
            elseif ((ILoG(i-1,j+1) >= 0 && ILoG(i+1,j-1) <
0) || (ILoG(i-1,j+1) < 0 &&...
                    ILoG(i+1,j-1) >= 0)) && abs(ILoG(i+1,j-
1) - ILoG(i-1,j-1)) > t
                IZC(i,j) = ILoG(i,j+1);
            end
        end
    end
end

% Convert the resulting images to unsigned 8-bit images,
and return them.
output1 = uint8(ILoG);
output2 = uint8(IZC);
output3 = ILoG;
output4 = IZC;
end
```

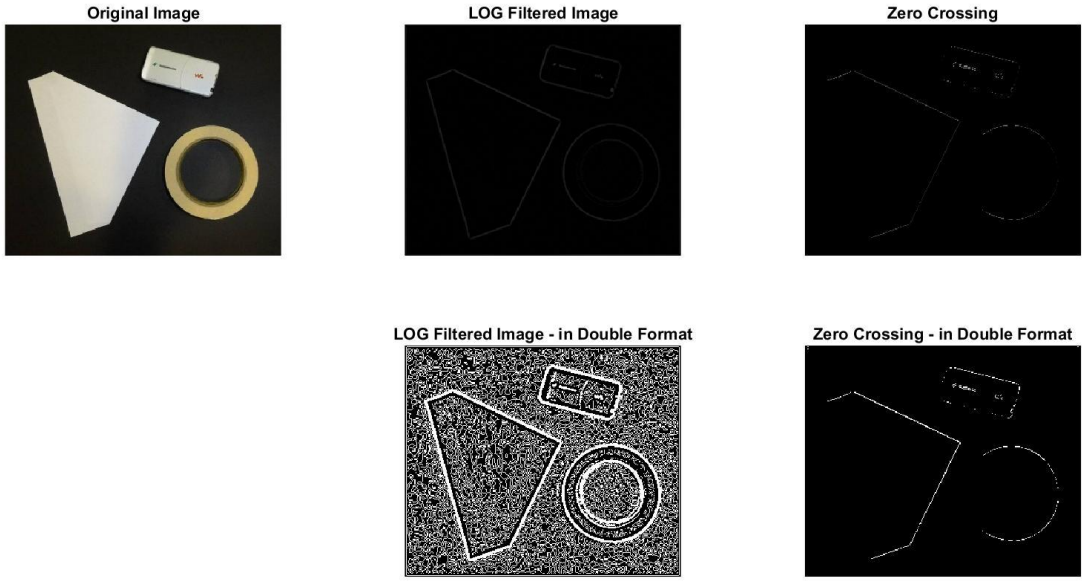
The following code calls the Zero Crossing function and applies it on the input image, before displaying the results.

```
%% Zero Crossing
% Read the image to be preprocessed
pp = imread('Object_contours.jpg');

% We call the LoG function. E represents the LoG filtered
image, while F represents the gradient profile. G and H are
% the same as E and F but in double format.
[E, F, G, H] = lab3ZeroCrossing(pp);

%The original image, the LoG filtered image, and the zero
crossing are displayed.
figure
subplot(2, 3, 1)
```

```
imshow(pp)
title('Original Image')

subplot(2, 3, 2)
imshow(E)
title('LOG Filtered Image')

subplot(2, 3, 3)
imshow(F)
title('Zero Crossing');

subplot(2, 3, 5)
imshow(G)
title('LOG Filtered Image - in Double Format')

subplot(2, 3, 6)
imshow(H)
title('Zero Crossing - in Double Format');
```

Results

The result is as follows:



DISCUSSION

-

Prewitt Operator

- A 1D Prewitt filter is very **noise sensitive.** Therefore, 3 1D central difference filters are used to simultaneously perform some smoothing.

-1	0	1

- From close observation, it won't be out-of-place to opine that the smoothing performed by the Prewitt filter is akin to box filter. This is because the weights of the window are ones on either side of the central column. The negative signs on the first column simply show that the left column is subtracted from the right column (this is how the derivative is performed).

-1	0	1
-1	0	1
-1	0	1

X Filter

Sobel Filter

- A 1D Prewitt filter is very **noise sensitive**. Therefore, 3 1D central difference filters are used to simultaneously perform some smoothing.

-1	0	1

- The Sobel filter gives more weights to the 4 immediate neighbors of the central pixel. This applies the principle **of 4-adjacency**. Therefore, we may say that Sobel implemented a **primitive Gaussian** in his filter such that if we normal the weights, we could come up with a gaussian curve. So, *the Sobel filter is similar in principle to the Gaussian filter*.

- Like in the Prewitt filter, the negative signs on the first column simply show that the left column is subtracted from the right column (this is how the derivative is performed).

-1	0	1
-2	0	2
-1	0	1

X Filter

-1	-2	-1
0	0	0
1	2	1

Y Filter

- Instead of a 1D filter, a 2D filter is used because 1D filters are very **noise sensitive**.
- Notice that the x-filter simply emphasizes the vertical lines in the image due to the change in that direction.
- Likewise, the y-filter emphasizes the horizontal lines due to the change in that in direction.

Key Observation:

- If we apply derivative directly on noisy images without first smoothing, we'll obtain very bizarre results.
- This is specifically why if we take a closer look at the Sobel filter, we notice that it performs smoothing and derivative simultaneously.

-1	0	1
-2	0	2
-1	0	1

X Filter

-1	-2	-1
0	0	0
1	2	1

Y Filter

As we can see above, for the:

a. x-filter – derivative is performed horizontally while smoothing is performed vertically.
b. y-filter – derivative is performed vertically while smoothing is performed horizontally.

- The central difference method is preferred for taking derivative because it produces errors of the order ε^2 as opposed to $\varepsilon -$

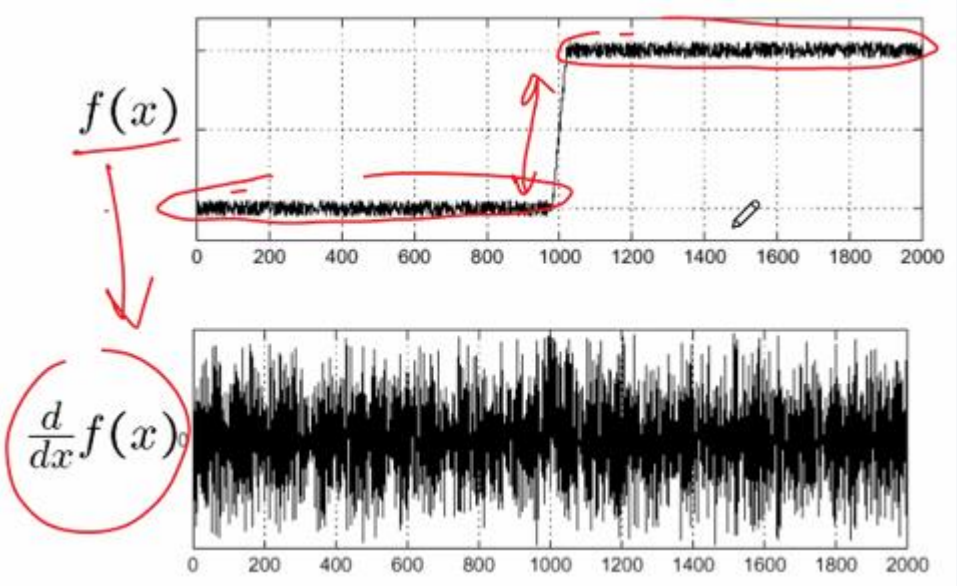
order errors produced by forward and backward difference methods.

- To properly apply convolution in our code, we implement the element-by-element matrix multiplication as seen below:

```
Ihor(i,j) = sum(wp(:).*y_filt(:));
```

Laplacian of Gaussian

- Just as we've emphasized for some time now, it is imperative that an image be smoothened before finding its derivative to avoid bizarre results like the following:



-

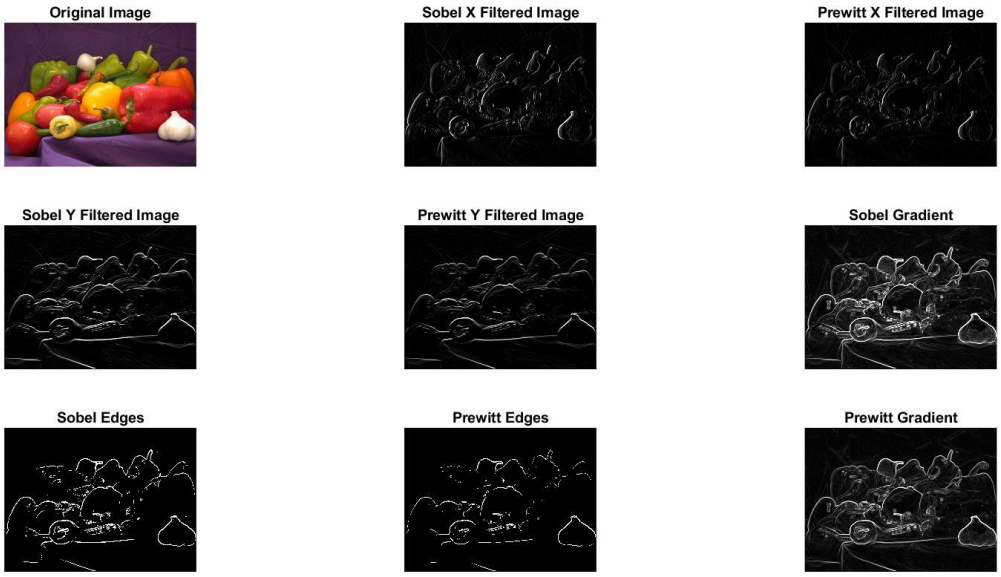
- Notice, the above result was obtained only after the first derivative. Imagine taking a derivative of the above noisy result… This underpins the requirement to smoothen an image first before such an operation. Hence, the LoG filter is applied on a Gaussian filtered image.
- For edge detection in this case, we are not just looking at for peaks. There must be zero-crossing (-ve to +ve or vice versa) to conclude on the presence of an edge.
- A peculiar advantage of the Laplacian is that **it can be performed using only one mask**. Prewitt and Sobel filters on the other hand require two masks – one for X and one for Y.

- *A major drawback with the Laplacian is that it produces a scalar result. At first glance, this might seem nice, but this results in the loss of orientation information from the final image*.

An important observation is that sharpening is used to emphasize the

Comparison – Prewitt and Sobel

Let's juxtapose the results from the Prewitt and Sobel filters for the sake of identifying the differences.



We immediately notice that: at the same threshold, the Sobel filter highlights more edges than the Prewitt filter.

- Sobel places emphasis on pixels close to the central pixel of each window, while Prewitt doesn't.
- Some false edges are detected as a result of noise.

REFERENCES

Reinhard Klette. 2014. Concise Computer Vision: An Introduction into Theory and Algorithms. Springer Publishing Company, Incorporated.

https://www.mathworks.com/discovery/edge-detection.html

https://en.wikipedia.org/wiki/Prewitt_operator

https://en.wikipedia.org/wiki/Sobel_operator

https://www.researchgate.net/post/What_are_the_differences_in_first_order_derivative_edge_detection_algorithms_and_second_order_edge_detection_algorithms

https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm

https://lokeshdhakar.com/projects/lightbox2/

https://www.dreamstime.com/pen-organizer-black-coffee-laptop-black-background-close-up-pen-organizer-black-coffee-laptop-black-background-image101116944

https://scholarworks.calstate.edu/downloads/1g05ff65f