Edge Detection

Alejandro Sanchez – CSE 107, April 28, 2022

Abstract

The goal of this lab is to perform a simple edge detection scheme using linear spatial filtering. Linear spatial filtering modifies an image by replacing the value at each pixel with some linear function. This filtering operation can be referred to as a mask and is applied to many small areas of an image at a time. Using the results of a mask can provide an images gradient, which helps in calculating "edges" in an image. An image gradient is a directional change of intensity values in an image. Using an image's gradient, it is possible to determine if a pixel belongs to an edge or not using linear spatial filtering. In this report, the methodology used to compute an image's gradient is further discussed and the algorithm used to apply a linear spatial filter.

Technical Discussion

Loading the test image and user input

The program begins by initializing a variable called 'inputMatrix' as type uint8. This variable is the test image that will be evaluated for edges in the edge detection scheme. The program prints the image as a base reference for the user. Next, the program will ask the user to enter a scalar value between 0 and 255. Entering a value of '-1' will quit the program. The important step is assigning the variable 'finalImage' as the return value of the find_edges() function. This function returns an image that has highlighted edges. The script in Figure 1 shows how the user input and test image are used in the program.

```
% Define the input image
inputMatrix = imread("watertower.tif");
figure()
imshow(inputMatrix);
scalar = 1;
while (scalar ~= -1)
    scalar = input("Enter a scalar value (-1 to quit): ");
    finalImage = find_edges(inputMatrix,scalar);
    figure()
    imshow(finalImage);
end
fprintf("Exiting program...\n\n")
```

Figure 1: Importing a test image and using user input.

The structure of the find_edges() function

The find_edges() function takes an input matrix of type ‘uint8’ and a scalar value between 0 and 255. The function returns a variable called ‘resultingImage’ of type ‘uint8’ and will include highlighted edges that satisfy the threshold requirement determined by the user’s desired scalar value. As shown in Figure 2, the find_edges() function begins by assigning the variable ‘magnitude’ as the resulting matrix from the gradient_magnitude() function and storing its dimensions in the x and y direction as ‘M’ and ‘N’, respectively. Next, the function iterates through each pixel in the magnitude matrix and reassigns each value depending on if they are greater than or equal to the scalar value determined by the user in the main function. If the pixel value satisfies this condition, it is assigned to a value of ‘255’. Otherwise, it is assigned as ‘0’. This will eliminate gray areas in the image and leave it as black and white to help identify edges.

```
function resultingImage = find_edges(inputImage, scalar)
    % Call gradient magnitude function
    magnitude = gradient_magnitude(inputImage);
    M = size(magnitude,1);
    N = size(magnitude,2);
    % Create result matrix
    resultingImage = zeros(M,N);
    for x=1:M
        for y=1:N
            if magnitude(x,y) >= scalar
                resultingImage(x,y) = 255;
            else
                resultingImage(x,y) = 0;
            end
        end
    end
    resultingImage = uint8(resultingImage);
end
```

Figure 2: The find_edges() function.

The structure of the gradient_magnitude() function

The gradient_magnitude() function takes a matrix of type uint8 as an argument and returns a matrix of type double. It returns the magnitude of the input image’s gradient. In Figure 3, it can be shown that two masks are created to apply a linear spatial filter on the input image. These masks are named ‘sobelX’ and ‘sobelY’, which refer to the sobel filter – also known as the sobel operator. The sobel operator helps emphasize edges in an image by applying a discrete differentiation operator. ‘sobelX’ and ‘sobelY’ are two matrices that are small, separable, and contain only integers. This helps keep a relatively lower computational load on images that are large and small. Deeper into the gradient_magnitude() function, matrices ‘gX’ and ‘gy’ are created, which represent a transformed version of the input image ‘inputImage’ using the variables ‘sobelX’ and ‘sobelY’, respectively. These two

matrices are the gradients in the x- and y-direction, which will help with determining edges in the input image. Finally, the program iterates through a new matrix called 'magnitude', which has the same size as the input image, and assigns each of its values with the magnitude of 'gX' and 'gY' combined.

```
function magnitude = gradient_magnitude(inputImage)
    % Define the sobel masks
    sobelX = [-1 -2 -1;
               0 0 0;
               1 2 1];
    sobelY = [-1 0 1;
              -2 0 2;
              -1 0 1];
    % Call spatial_filter function
    gX = spatial_filter(inputImage, sobelX);
    gY = spatial_filter(inputImage, sobelY);
    M = size(gX,1);
    N = size(gX,2);
    magnitude = zeros(M,N);
    for x = 1:M
        for y = 1:N
            magnitude(x,y) = sqrt(gX(x,y).^2 + gY(x,y).^2);
        end
    end
end
```

Figure 3: The gradient_magnitude() function.

The structure of the spatial_filter() function

The spatial_filter() function takes a matrix of type uint8 and a filter mask as arguments and returns a matrix of type double. It produces an image copy of the input image after a specified spatial filter is applied to the input image's values. In Figure 4 below, it can be shown how the spatial filter is applied to the input image. First, the function assigns the variables 'M' and 'N' as the size of the input image in the x- and y-direction, respectively. The function proceeds to assign variables 'filterM' and 'filterN' as the size of the input filter size in the x- and y-direction as well, respectively. A new matrix named 'temp' is created to perform linear spatial operations onto. This new matrix requires a larger size to accommodate for zero padding, which is a process to help MATLAB navigate outside the boundaries of the input image. This is an important step for the program because it will need to apply the spatial filter to the input image in indices in and out of the input image. Variables 'offsetX' and 'offsetY' help reassign values in the 'temp' matrix.

The program continues by iterating through each value in the input image and the 'temp' matrix where indices overlap. Finally, the program produces a variable with values using the offset variables. This variable is called 'outputImage', which contains a copy of the input image after the spatial filter is applied.

```
function outputImage = spatial_filter(inputImage, filterMatrix)
    M = size(inputImage,1);
    N = size(inputImage,2);
    filterM = size(filterMatrix,1);
    filterN = size(filterMatrix,2);
    % Create temp matrix to apply filter matrix
    temp = zeros(M+filterM-1,N+filterN-1);
    offsetX = ((filterM - 1) / 2) + 1;
    offsetY = ((filterN - 1) / 2) + 1;

    % Initialize the output image
    outputImage = zeros(M,N);

    % Reassign the values in the temp matrix
    for x = offsetX:M+1
        for y = offsetY:N+1
            temp(x,y) = inputImage(x-offsetX+1,y-offsetY+1);
        end
    end

    % Go through each value in the image (the values where the image and the
    % temp matrix overlap)
    for x = offsetX:M+1
        for y = offsetY:N+1
            % Go through each pixel in the image and apply the filter
            val = 0;
            for m=1:filterM
                for n=1:filterN
                    val = val + (temp(x-offsetX+m,y-offsetY+n) * filterMatrix(m,n));

                end
            end
            outputImage(x-offsetX+1, y-offsetY+1) = val;
        end
    end

end
```

Figure 4: The spatial_filter() function.

Results

The results of the functions and processes in the program are shown below in Figure 5, 6, and 7.

Figure 5: The original input image.

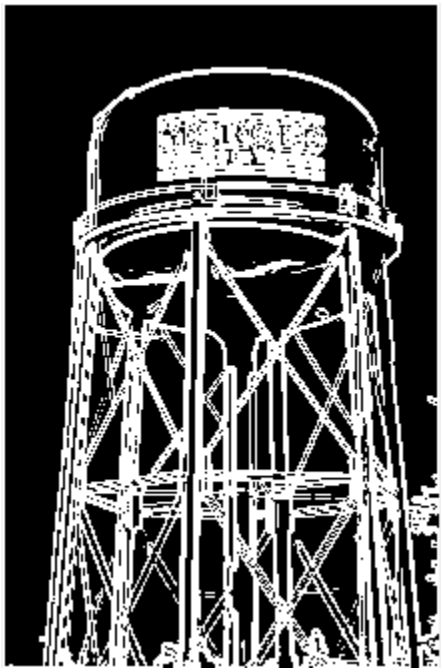

Figure 6: The output image using a threshold of 100.

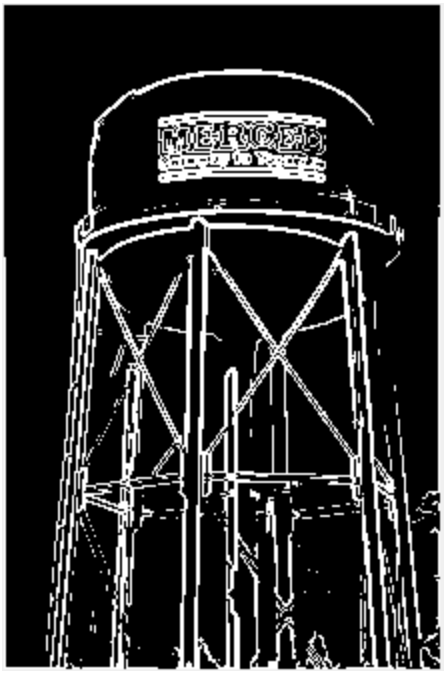

Figure 7: The output image using a threshold of 255.

Conclusion

In conclusion, the magnitude of the gradient of a filtered image shows that edge detection is possible through means of a linear spatial operator. Additionally, it is proven through the MATLAB script that the linear spatial operator is applicable for different threshold values.

It can seen that with higher thresholds, the images can have higher contrasts of edges. This is because the program can identify higher intensity values with a higher scalar value. If the user enters a scalar value of 0, it is expected that the resulting image will be white.

```
clc;
close all;
clear all;

%{
Linear Spatial Filtering
https://www.math.uci.edu/icamp/courses/math77c/demos/linear_spatial_filtering.pdf

Image Gradient Wikipedia
https://en.wikipedia.org/wiki/Image_gradient

Sobel Operator Wikipedia
https://en.wikipedia.org/wiki/Sobel_operator

What is zero padding
https://www.icsid.org/uncategorized/what-is-zero-padding-in-image-processing/
%}

% Define the input image
inputMatrix = imread("watertower.tif");
figure()
imshow(inputMatrix);
scalar = 1;
while (scalar ~= -1)
    scalar = input("Enter a scalar value (-1 to quit): ");
    finalImage = find_edges(inputMatrix,scalar);
    figure()
    imshow(finalImage);
end
fprintf("Exiting program...\n\n")

% Debugging test matrix
% testMatrix = randi(10,10,10);
% resultMatrix = spatial_filter(testMatrix, filterToApply);

%{
This is the find_edges function. It takes an input matrix of
type uint8 and a scalar value between 0 and 255, then returns a
gradient image. The return value is type uint8.
%}
function resultingImage = find_edges(inputImage, scalar)
    % Call gradient magnitude function
    magnitude = gradient_magnitude(inputImage);
    M = size(magnitude,1);
    N = size(magnitude,2);
    % Create result matrix
    resultingImage = zeros(M,N);
    for x=1:M
        for y=1:N
            if magnitude(x,y) >= scalar
                resultingImage(x,y) = 255;
            else
                resultingImage(x,y) = 0;
            end
        end
```

```
    end
    resultingImage = uint8(resultingImage);
end

%{
This is the gradient_magnitude function. It takes a matrix of
type uint8 as an argument and returns a matrix of type double.
It returns the magnitude of the gradient of an image.
%}
function magnitude = gradient_magnitude(inputImage)
    % Define the sobel masks
    sobelX = [-1 -2 -1;
               0 0 0;
               1 2 1];
    sobelY = [-1 0 1;
              -2 0 2;
              -1 0 1];
    % Call spatial_filter function
    gX = spatial_filter(inputImage, sobelX);
    gY = spatial_filter(inputImage, sobelY);
    M = size(gX,1);
    N = size(gX,2);
    magnitude = zeros(M,N);
    for x = 1:M
        for y = 1:N
            magnitude(x,y) = sqrt(gX(x,y).^2 + gY(x,y).^2);
        end
    end
end

%{
This is the spatial_filter function. It takes a matrix of type
uint8 and a filter mask as arguments and returns a matrix of type
double. It returns an image copy of the input image after a
filter is applied to its values.
%}
function outputImage = spatial_filter(inputImage, filterMatrix)
    M = size(inputImage,1);
    N = size(inputImage,2);
    filterM = size(filterMatrix,1);
    filterN = size(filterMatrix,2);
    % Create temp matrix to apply filter matrix
    temp = zeros(M+filterM-1,N+filterN-1);
    offsetX = ((filterM - 1) / 2) + 1;
    offsetY = ((filterN - 1) / 2) + 1;

    % Initialize the output image
    outputImage = zeros(M,N);

    % Reassign the values in the temp matrix
    for x = offsetX:M+1
        for y = offsetY:N+1
            temp(x,y) = inputImage(x-offsetX+1,y-offsetY+1);
        end
    end
```

```
    % Go through each value in the image (the values where the image and the
    % temp matrix overlap)
    for x = offsetX:M+1
        for y = offsetY:N+1
            % Go through each pixel in the image and apply the filter
            val = 0;
            for m=1:filterM
                for n=1:filterN
                    val = val + (temp(x-offsetX+m,y-offsetY+n) * filterMatrix(m,n));

                end
            end
            outputImage(x-offsetX+1, y-offsetY+1) = val;
        end
    end

end
```

For commit history, refer to the GitHub repository at the link below:

https://github.com/alejandrohsanchez/Digital-Image-Processing-Projects/tree/main/Labs/04