American University of Armenia

Zaven and Sonia Akian College of Science and Engineering

Segmentation using Front Propagation and Partitioned Images

Elen Tumasyan, Larisa Malkhasyan, Nelli Hovhannisyan

Supervisor: Varduhi Yeghiazaryan

Spring 2019

Abstract

This work aims to discuss image processing algorithms that particularly focus on image segmentation. Throughout the thesis, we will cover gradient magnitude on several convolution filters, boundary detection, image thresholding, region growing, watershed, fast marching and fast dashing methods. We have written the code for each of them in Python language using exclusively our knowledge and analysis (no built-in functions are used). For our testing purposes we use the Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500 [5]). The purpose of this work is to compare and discuss various image segmentation techniques and their results.

Acknowledgements

We would like to express our gratitude for all the guidance and advice we have received during this project to our supervisor Varduhi Yeghiazaryan. We would like to thank her for providing an interesting topic for our capstone project and for her help throughout the whole process.

Contents

Chapter 1

Introduction

The first thing we do when we cross the street is to look left and right, detect that there are no vehicles and cross the street. Our brain is capable of analyzing what kind of a vehicle is approaching us in milliseconds. Can computers do that?

Taking into account the rapid development of Computer Science, the answer is going to be "Yes". Nowadays, computers are able to detect objects, decide their size and even predict the direction that the object will take. There are many challenges in computer vision, but even more ways to deal with them.

In this work we will discuss and analyze multiple algorithms on edge detection and image segmentation. We will cover gradient magnitude on several convolution filters, boundary detection, image thresholding, region growing, watershed, fast marching and fast dashing methods. We have written the code for each of them in Python language using exclusively our knowledge and analysis (no built-in functions are used). At the end of the report we will demonstrate all of the results from our code implementations and do comparisons. The images that we used were taken from Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500).

Chapter 2

Image Grayscale

2.1 Image Representation in Computers

Before starting the main topic, let us shortly discuss the very basics of image representation in computers.

Every image which we see in real life, can be represented as sequence of numbers in computers. Images are multidimensional arrays of pixels. Pixel is the basic unit of image representation. Each pixel has color and transparency. Colors can be described differently in various systems, for example, RGB (red, green, blue) or CMYK (cyan, magenta, yellow, black). RGB means every pixel is a combination of 4 numbers, the amount of red, green, blue and alpha (transparency). In addition, every image can easily be converted to grayscale by leaving out information about color. This is discussed more thoroughly in the next section.

2.2 GrayScale

In computer science and image processing, in particular, grayscale image (also grayscale) is the representation of image data that has only one value for each of its pixels. That value represents the intensity data of the pixel. Such images are also known as gray monochrome, meaning having just one color that is gray. The range of the gray color goes from, black, having the lowest intensity, to white, having the highest intensity.

Figure 2.1: A Color image on the left, with its Grayscale version on the right

In Figure 2.1 we can see the color image with code 20069 from BSDS500 data set that we have converted into its grayscale version. There are several algorithms for grayscale conversion and we will discuss them in the upcoming sections.

2.3 Isolated Color Channels

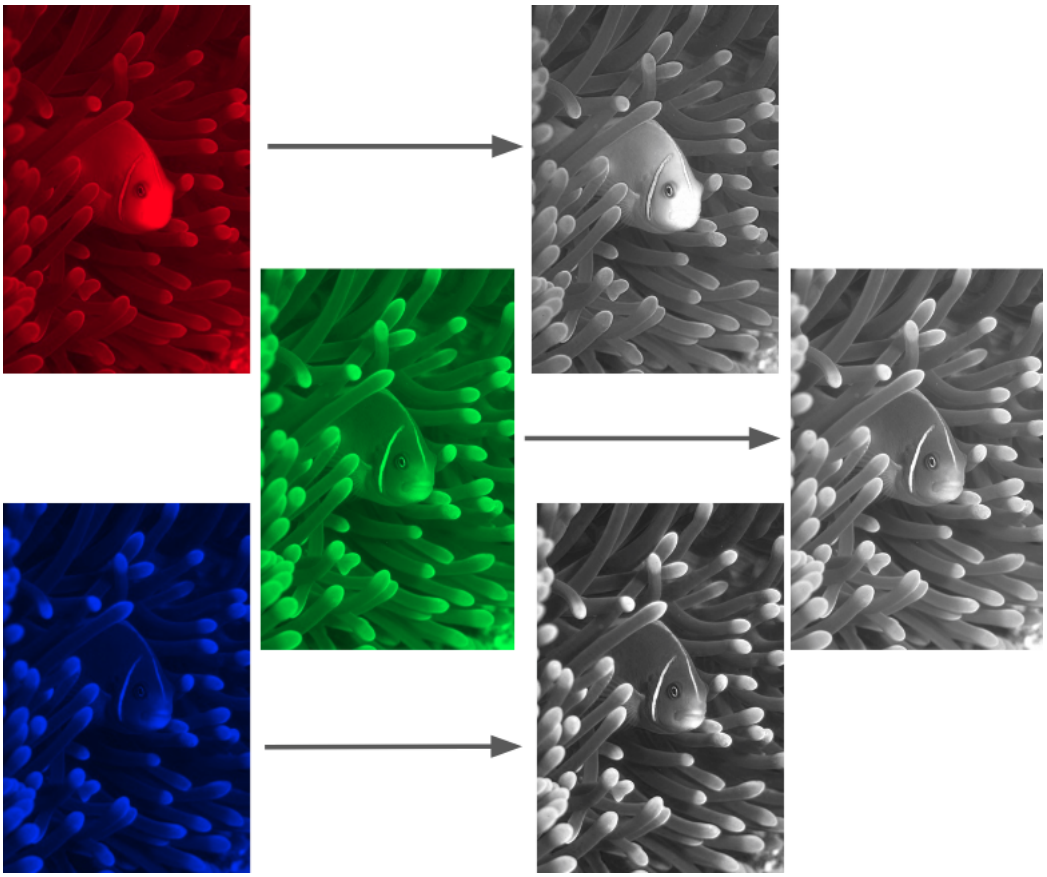
Color images are represented in multiple independent channels of colors. Every channel gives the color intensity in that particular channel. For RGB images, for example, there are three independent channels: Red, Green and Blue. One way to convert an image into a grayscale version is isolating the color channels. For instance, take only the value of the green channel of a pixel and give that value to the gray color of a grayscale.

Figure 2.2 isolated channels of Red, Green and Blue and their respective grayscale versions of the original image, which is taken from the BSDS500 dataset, the corresponding code is 210088. Comparing these three results, we can see that the "worst" effect for this particular image was after the conversion of the Red. Green and Blue gave similar results, though after Blue conversion the grayscale image is

darker which was expected, as the blue channel is darker and closer to black with values in the color spectrum.

Figure 2.2: Image 210088 from BSDS500 dataset: Channels Isolation and Grayscale Conversion

Original image



Isolated channels of RGB and their respective Grayscale versions

2.4 Choosing Min, Max and Average Value

Another approach for grayscale conversion could be to map the values of Red, Green and Blue into one value by some criteria. These criteria vary from implementation to another. We are going to consider mainly 3 of them.

Firstly, consider taking the min value of these three. Here the expectations will be to have a darker tone, as the smaller the values of the pixels are, the darker the image will be. Another approach may be to take the max of the three, resulting in a lighter tone version of grayscale due to grayscale values being closer to white. And finally we have tried to take the average of Red, Green and Blue values as it would intuitively give more precise approximation of image's intensity information. In other words our new pixel is $x = (RED + GREEN + BLUE)/3$.

Figure 2.3: Image 220075 from BSDS500 dataset: Choosing Min, Max and Average Values for the conversion into a Grayscale

Original image | Taking the maximum

Taking the minimum | Taking the average

In order to compare these three approaches, we have illustarted them on Image 220075, shown in Figure 2.3. We do think that the best result, as expected, was obtained by taking the average. The average combines and summarizes the three values into one number, thus with intermediate average values corresponding to intermediate brightnesses, we are getting an "average" brightened image, compared to the other two mentioned cases of getting too bright or too dark versions. Thus, in the following discussions, we rely on grayscale images obtained by taking the average of the color channels.

Chapter 3

Image Blurring

3.1 Introduction

Many techniques and algorithms have been developed to get a blurred image from an original. When we blur an image, we make color transition around edges of objects in the image look smooth rather than sudden. The effect is to average out rapid changes in pixel intensity. Generally, blurring can be considered as a convolution of the original image matrix and a blur kernel. A simple formula for getting a blurred image would be

$$g(x,y) = f(x,y) \oplus h(x,y) + \eta(x,y)$$

where

- $g(x,y)$ is a blurred image
- $f(x,y)$ is a image to be blurred
- $h(x,y)$ is a blur kernel
- $\eta(x,y)$ is the noise

There are several different blurring algorithms, but we will focus on just two: the Average Blur and the Gaussian Blur. We have chosen these two, as these are the most widely used methods. Moreover, both of them are linear, resulting in better performance. Also, they are rather simple algorithms, and they do give the needed result. Thus, our choice has fallen on these two in this scope of the thesis.

3.2 Convolution

Before proceeding to the main algorithms, we have decided to give some theoretical background on what convolution is.

In mathematics, convolution is an operation on two functions, that results in a third one, generally it is a modified version of one of the functions. In next sections we will denote the convolution operator as *. So convolution between f and g functions is denoted by $f * g$. By definition,

$$(f * g)(t) = \int_{-\infty}^{\infty} (f(\tau)g(t-\tau)d\tau$$

In image processing convolution can be interpreted in a different way and implemented by a certain algorithm. As all images are matrices, therefore in convolution we will also deal with matrices. If we take convolution $f * g$, g will be called a mask or a filter. It is a two dimensional matrix, that always has odd size, for example, 5×5 or 7×7. We always take odd size for mask, as in convolution algorithms we need to find mid of matrix. Now let us look at the steps of image convolution. At first we should flip the mask vertically and horizontally. Then go over the whole image, so that the center of the mask is on the corresponding element. Finally, we take all the elements of the image that fall under the mask and dot product the corresponding image element with the one in the mask. We should repeat this procedure for all terms of the image.

Let us suppose this is our initial mask:

3	6	9
12	15	18
21	24	27

Let us flip the mask vertically and horizontally

27	24	21
18	15	12
9	6	3

Now, suppose the following is our image matrix

1	2	3
4	5	6
7	8	9

By the algorithm, we should do the following action

27	24		21	
18	15	1	12	2	3
9	6	4	3	5	6
		7	8	9

For the first pixel of the image, the result will be

$$15 * 1 + 12 * 2 + 6 * 4 + 3 * 5 = 78$$

78	*	*
*	*	*
*	*	*

By performing the same action for all other elements, the resulting matrix can be filled.

3.3 Average Blurring

Average blurring is a very simple way of getting an image blurred. We retrieve all pixels within a kernel of specific width and height, simply add them together, divide by the size of the kernel and assign the resulting value to the center pixel. For better understanding, here is an example. Let's denote our kernel, which is a 5x5 matrix, with K and assign 1 to all the elements:

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

For each pixel of the original image, this 5x5 matrix is placed around its position. All the pixels falling within this matrix range are summed up and the result is then divided by 25. This algorithm simply computes the average of the pixel values inside that matrix. This process is implemented on all the pixels in the original image to produce the desired output image.

3.4 Gaussian Blurring

Gaussian blurring is the result of blurring an image by a Gaussian function. This is the most widely used blurring filter. The general form of the Gaussian function is given as follows

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution. Gaussian function is a function which creates a distribution of values around the center point. Thus the center pixel of the kernel has the biggest value and others are uniformly decreasing. The result is that the center pixel contributes more towards the new pixel value than those further away. For example, we may have a 3x3 pixel kernel with σ=1:

$$\begin{bmatrix} 0.07 & 0.12 & 0.07 \\ 0.12 & 0.19 & 0.12 \\ 0.07 & 0.12 & 0.07 \end{bmatrix}$$

As we can see, the center has the biggest value and by increasing σ we can get a more blurred image.

The difference between the Average blur and the Gaussian blur is that in the Average blur every pixel in the kernel is given the same weight. In the Gaussian blur, however, the pixels in the kernel that are closer to the center pixel have more weight than the pixels near the edge of the kernel.

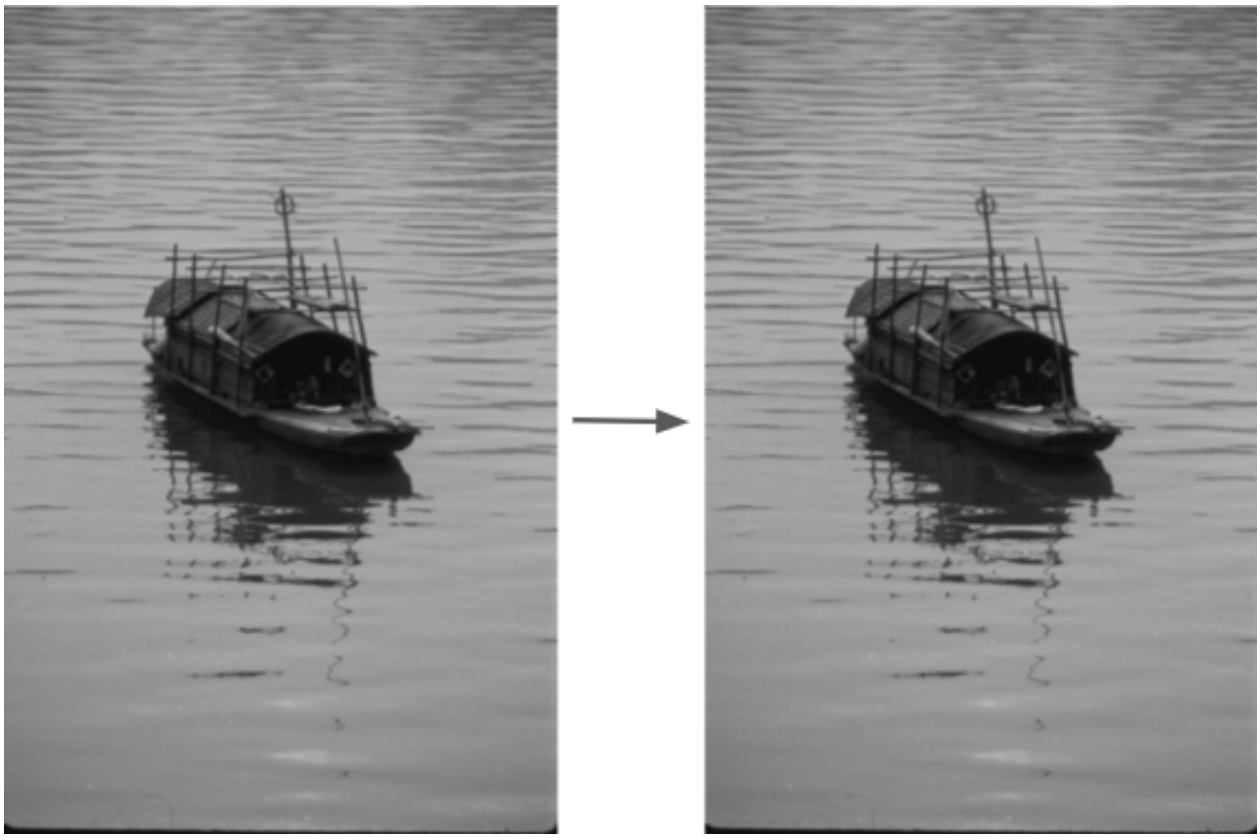
3.5 Convolution Examples

We decided to write a couple of Convolution examples with Python to demonstrate Image Convolution (blurring, etc.). Note that all images are first converted into grayscale.

Figure 3.1 shows the image with code 15088 and its convolution with the identity kernel:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Figure 3.1: Image 15088 from BSDS500 dataset: Convolution with the identity kernel



It is easy to guess that the image must remain the same after the convolution as we do not sum with any other pixel besides itself (all remaining values are zeros).

In Figure 3.2 we can see the image with code 25098 and its convolution with the average (box) blur kernel.

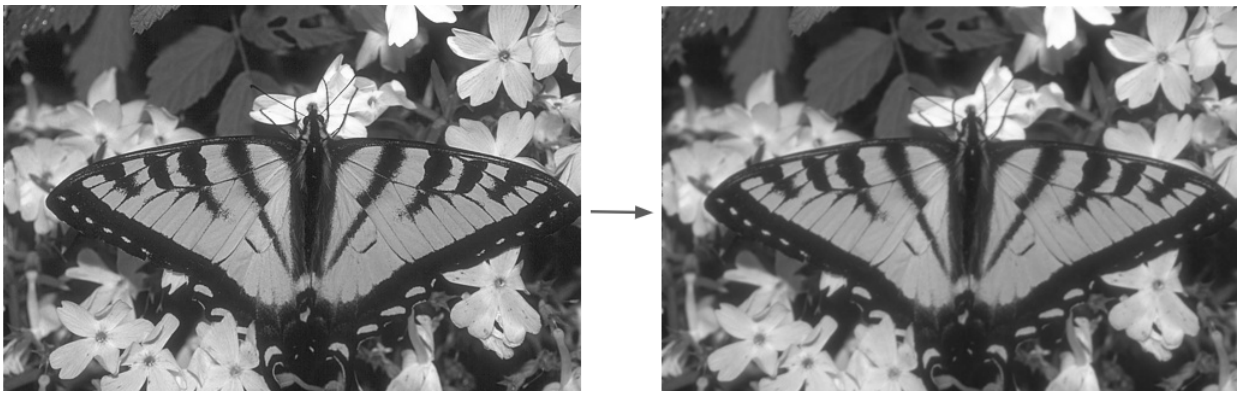
$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

And finally, in Figure 3.3 we can see the image with code 35010 and its convolution with the following gaussian blur kernel:

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 3.2: Image 25098 from BSDS500 dataset: Convolution with the average blur kernel

Figure 3.3: Image 35010 from BSDS500 dataset: Convolution with the gaussian blur kernel

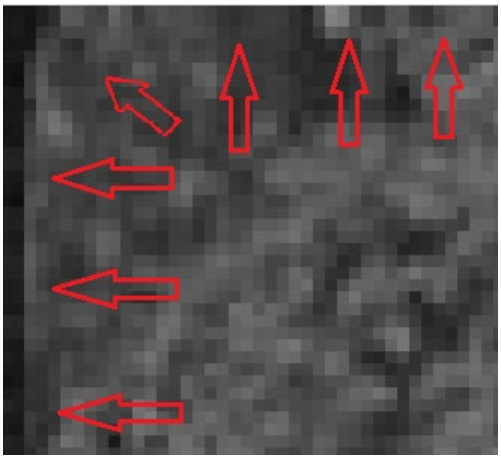


3.5.1 Boundary Pixels

When performing a convolution with a kernel, there is the issue of boundary pixels. One solution would be to wrap the image with a layer of pixels containing the same set of values as the boundary pixels, so that we have relatively correct blurring at the boundary points. We can see the results of performing the wrapping operation in Figure 3.4 (where you can see the region of interest within the red square). The mentioned region is zoomed next to it. The duplicate boundary pixels are easily visible in our zoomed image.

Figure 3.4: Image 107172 from BSDS500 dataset: Boundary Wrapping using Pixel Duplicates

Boundary wrapping



Zoomed illustration of image wrapping

For more information on image blurring see [3].

Chapter 4

Gradient Magnitude

4.1 Introduction

The previous chapters discussed kernels and image convolution. In this chapter we will examine different methods of image convolution with kernels which perform edge detection. In order to understand it easily, we can think of these methods as measuring the change in pixels' intensity. If their change is big, then there is a good chance that the detected pixel is located on an edge within our image, so we should separate it from other pixels.

To be mathematically precise, the way of measuring the edges (changes in pixel intensities) is done by calculating the gradient at each point on an image. Let us recall the formula of gradient vector operator:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{4.1}$$

And after computing the gradient, we can compute its magnitude as well.

$$||\nabla f|| = \sqrt{\frac{\partial f}{\partial x}^2 + \frac{\partial f}{\partial y}^2} \tag{4.2}$$

We know that the gradient of the function is continuous, however as we are dealing with discrete values in image processing, then we can simply approximate the gradient at a given point (pixel). Now we proceed to examine a few edge detecting gradient operators.

4.2 Prewitt Operator

The Prewitt operator does convolution on the image with the Prewitt kernel to obtain the central difference of the pixel. As a one-dimensional formula it has the

following form:

$$\frac{\partial f}{\partial x} \approx (f(x+1) - f(x-1))/2$$

Or, for the two-dimensional case, taking our image to be I, for x (horizontal) we have the formula

$$\frac{\partial I}{\partial x} \approx (I(x+1,y) - I(x-1,y))/2$$

which corresponds to the normalized kernel

$$\frac{1}{2}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

and, for y (vertical) we have

$$\frac{\partial I}{\partial y} \approx (I(x,y+1) - I(x,y-1))/2$$

which corresponds to the normalized kernel

$$\frac{1}{2}\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

However, this approach would be vulnerable to noise, so we reduce the risk by averaging y when calculating $\frac{\partial I}{\partial x}$ and averaging x when calculating $\frac{\partial I}{\partial y}$. So the convolution kernels for each of them will be

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} for \ \frac{\partial I}{\partial x}$$

and

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} for \ \frac{\partial I}{\partial y}$$

We should not forget about normalizing the kernels: in our case it would be done by multiplying the matrices by the scalar $\frac{1}{6}$.

4.3 Sobel Operator

Sobel operator is another filter that has nearly the same explanation as Prewitt. Besides that, it takes more focus on the central pixels than in outer ones in the convolution kernel, having the form

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} for \ \frac{\partial I}{\partial x}$$

and

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} for \ \frac{\partial I}{\partial y}$$

Keeping in mind that we should normalize the matrix, it must be multiplied it by the scalar $\frac{1}{8}$.

4.4 Gradient Magnitude with Sobel and Prewitt

The implementation of the operators is done by convolving the grayscale image with the corresponding normalized kernel of each operator. We first get the convolution result on horizontal axis, let's denote it G_x, then on vertical axis, let's denote it G_y. After that we compute the Gradient Magnitude: GM with the formula (noted in the introduction section as well)

$$GM = \sqrt{{G_x}^2 + {G_y}^2} \tag{4.3}$$

As for our grayscale images we have the range [0, 255], we need to map the results from calculation of gradient magnitude into that range, in order to represent the gradient magnitude as an image. We decided to implement it in two alternative ways. We have two techniques to calculate G_x and G_y, and respectively, two ways to calculate the Gradient Magnitude. Let us examine them separately in the next subsections.

4.4.1 Gradient Magnitude by Shifting

In this method, in order to represent G_x and G_y results, we convolve our image with the respective normalized operator. After getting the value from convolving with kernel, we add 127 to that value in order to shift the zero point to 127. The reason for that operation is the following. When we convolve with our kernel, we get a value that represents the rate of change for that pixel area. If the value is zero, then there is no change, meaning we are not near an edge. Otherwise, if the value is bigger than zero or less than zero (sign is not crucial), this means that we have a change near that pixel area. When we shift our range of values to right by 127 units, we make the average gray color represent the area where there is no change,

and the pixels with colors near black and white represent areas with a high rate of change.

After getting G_x and G_y, we proceed to calculating the Gradient Magnitude. However, the GM formula for this case for will be

$$GM = \sqrt{(G_x - 127)^2 + (G_y - 127)^2} \tag{4.4}$$

because we need to shift back to left by 127 units, in order to get mathematically correct result of GM. Now let us analyze the range of values we get from this formula.

The pixels are from 0 to 255, and the formula for GM is increasing as G_x and G_y increase, so the lowest value of GM is when we have 0 for both G_x and G_y, that is

$$\sqrt{0^2 + 0^2} = 0$$

And the highest value of GM is when we have 255 for both G_x and G_y, that is

$$\sqrt{(255 - 127)^2 + (255 - 127)^2} \approx 181$$

So the range of values for GM function is [0, 181]. We can map the values from [0, 181] into [0, 255] by dividing the value by 181 and multiplying it by 255. For instance, if we get $GM = 110$, then its value for image representation will be

$$\frac{110}{181} * 255 \approx 155$$

4.4.2 Gradient Magnitude by Scaling

In this second method we again find an accurate way of representing G_x and G_y, but instead of shifting the value range to right by 127 units, we take the absolute value and scale it up by multiplying by 2. Let us examine this in detail. When we take the absolute value, we map the range [-127, 127] (this range occurs because, when performing convolution, the kernel is normalized) into range [0, 127], so in order to further map it to [0, 255] image representable format, we multiply it by 255 and divide by 127, which is almost the same as multiplying the value by 2. Thus we calculate G_x and G_y.

Now we proceed to calculating the Gradient Magnitude. The formula, as we know it, is

$$GM = \sqrt{{G_x}^2 + {G_y}^2} \tag{4.5}$$

Again the pixels are from 0 to 255, and the formula for GM is increasing as G_x and G_y increase, so the lowest value of GM is when we have 0 for both G_x and G_y, that is

$$\sqrt{0^2 + 0^2} = 0$$

And the highest value of GM is when we have 255 for both G_x and G_y, that is

$$\sqrt{255^2 + 255^2} \approx 360$$

So the range of values for GM function is [0, 360]. We can map the values from [0, 360] into [0, 255] by dividing the value by 360 and multiplying it by 255. For instance if we get $GM = 130$, then its value for image representation will be

$$\frac{130}{360} * 255 \approx 92$$

4.4.3 Results of Sobel and Prewitt operators combined with both methods for GM calculation

In order to see the results of the operators, first let us differentiate between the two methods by displaying the result on a simple chess board in Figure 4.1. The shifting and scaling methods are in Figure 4.2.

Figure 4.1: Chess board

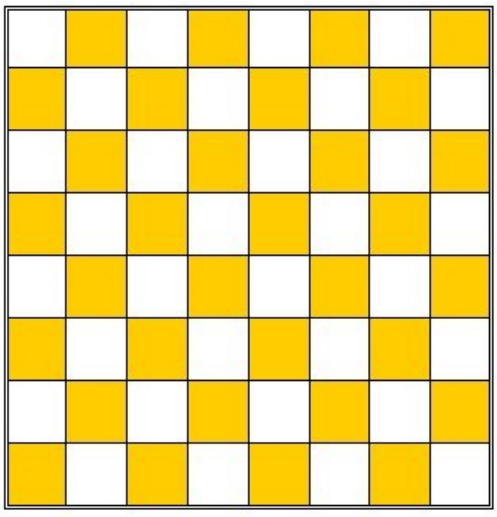
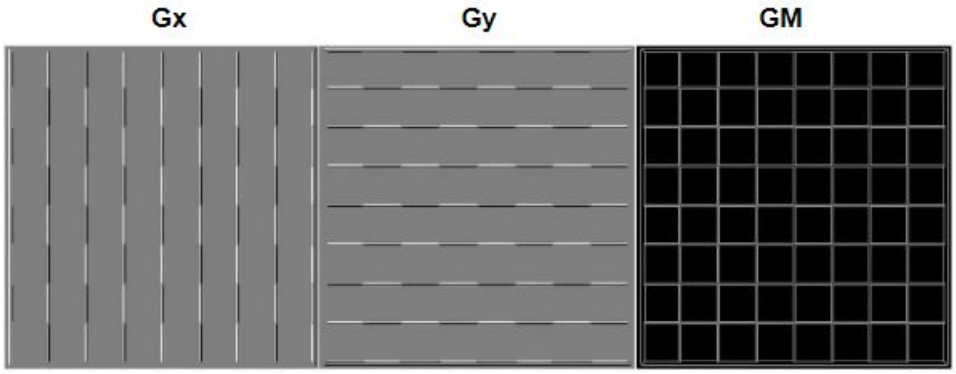
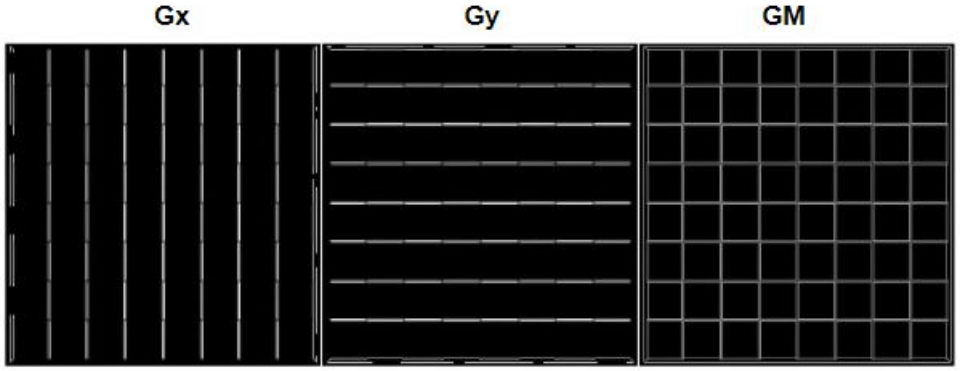


Figure 4.2: Chess board image: The Gradient Magnitude by shifting and scaling methods



The Gradient Magnitude by shifting method



The Gradient Magnitude by scaling method

Now let us check the results of the Sobel Filter (the taken image's code is 217090). The results of the Sobel shifting and scaling methods are shown in Figure 4.3.

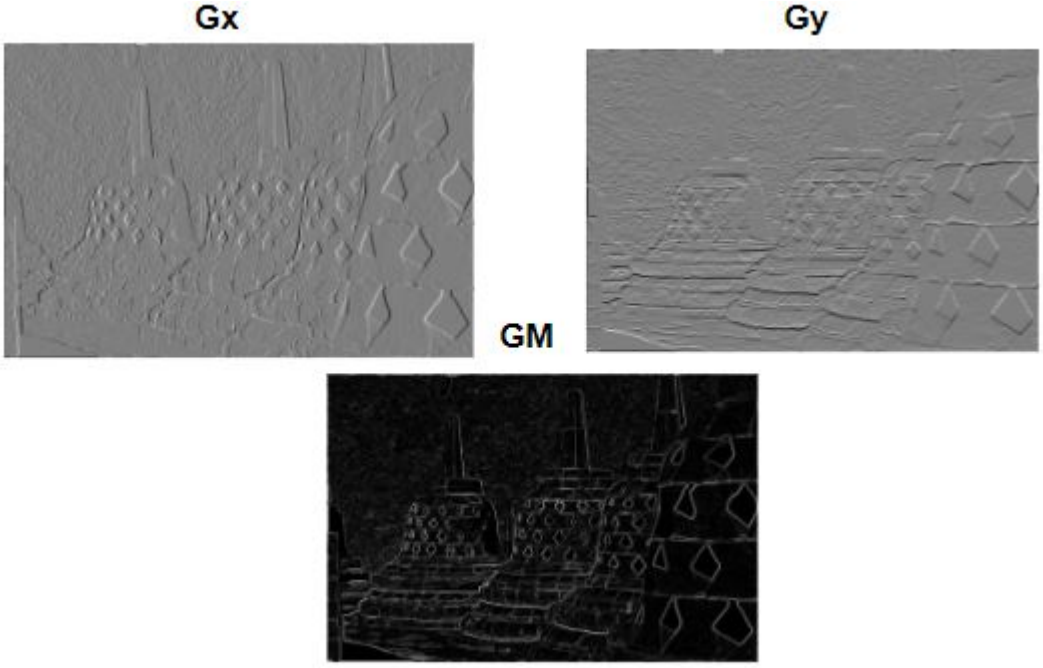
And, finally, let us see the results of Prewitt Filter for the image with code 16068 in 4.4. The comparison of both scaling and shifting methods is shown.

From our tests, we have noticed that Prewitt is very sensitive to noise, thus for our discussions in the upcoming chapters, we decided to primarily use Gradient Magnitude with Sobel, though Prewitt will be used in some as well.

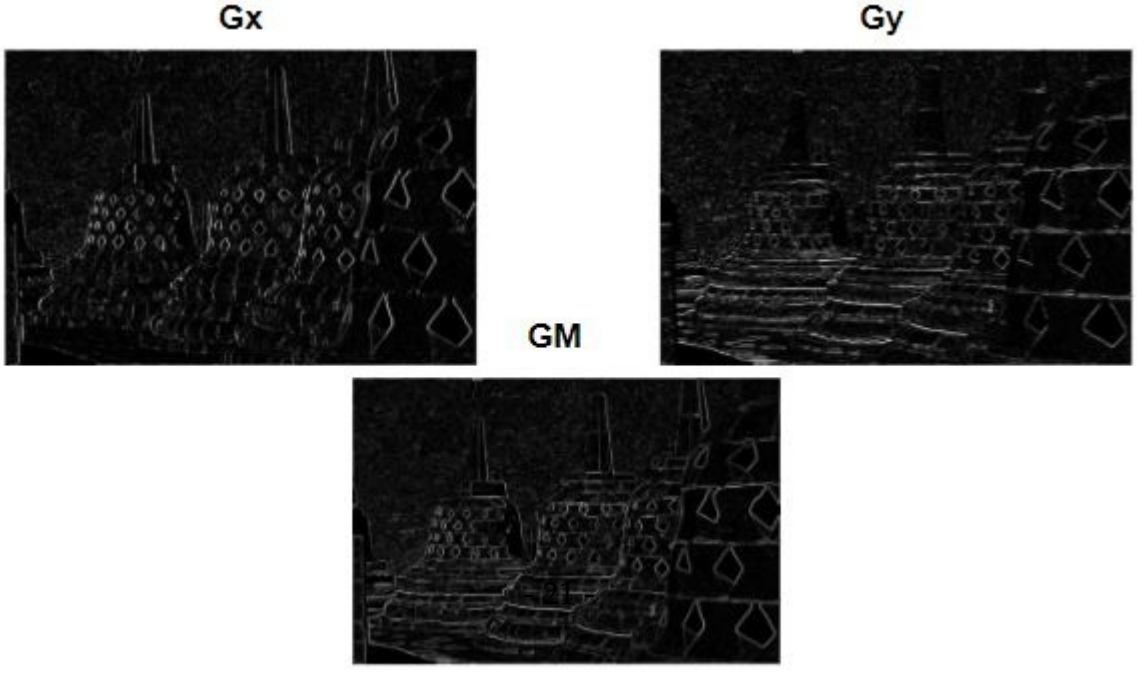
For more information on Gradient Magnitude see [8, 2].

Figure 4.3: Image 217090 from BSDS500 dataset: Sobel Filter by shifting and scaling methods

Original image



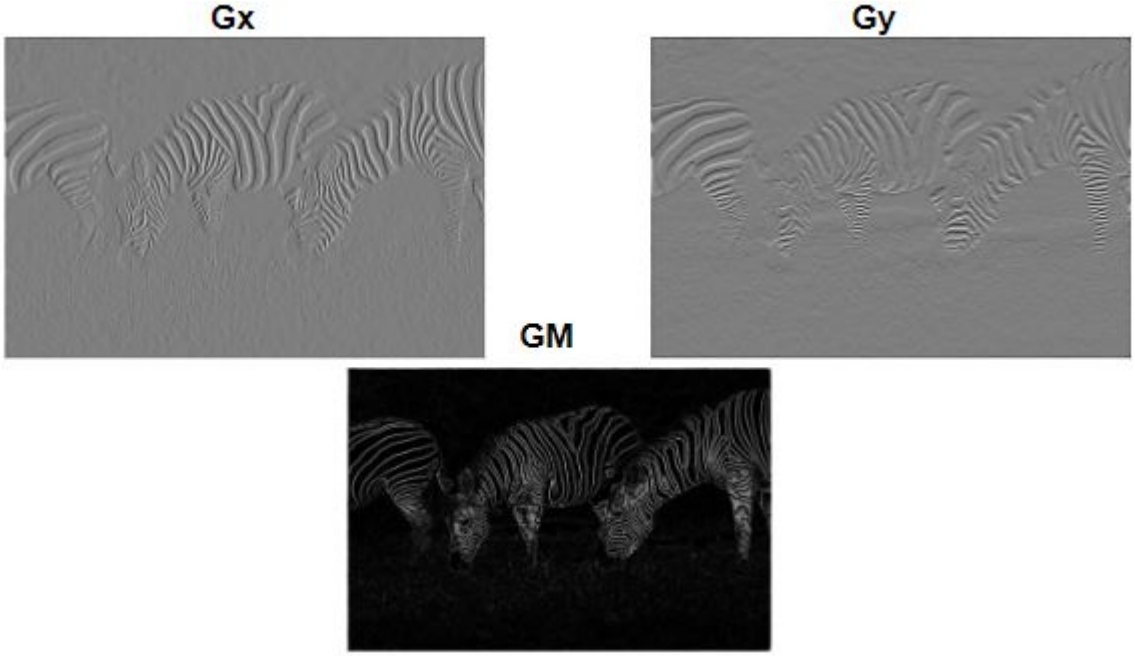
Sobel Filter by shifting method



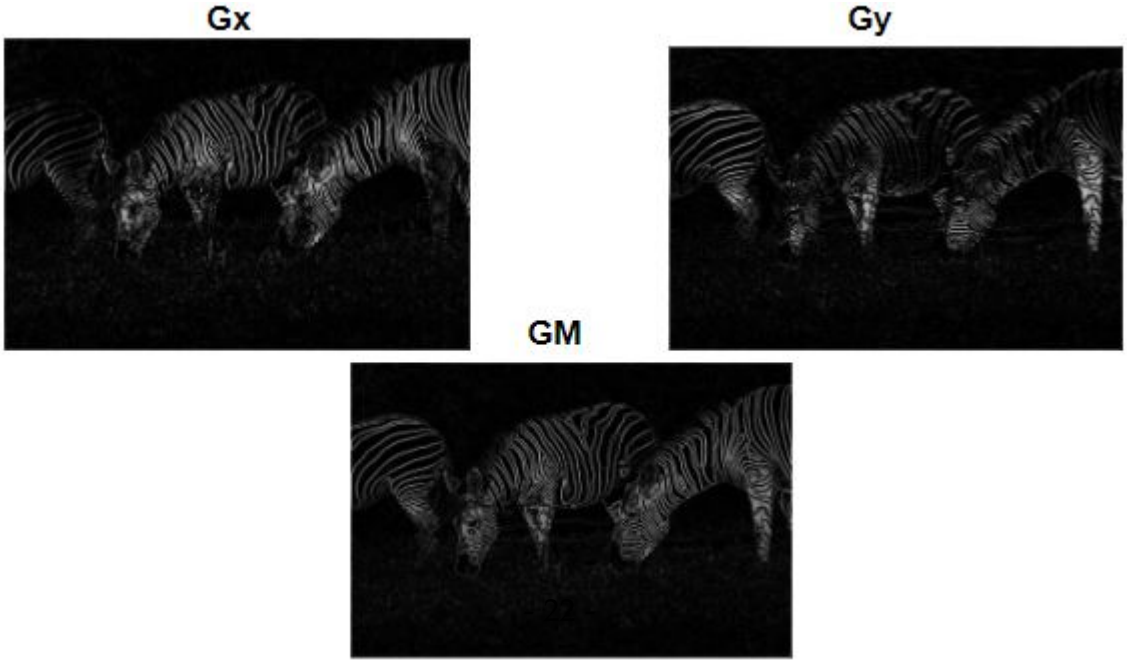
Sobel Filter by scaling method

Figure 4.4: Image 16068 from BSDS500 dataset: Prewitt Filter by shifting and scaling methods

Original image



Prewitt Filter by shifting method



Prewitt Filter by scaling method

Chapter 5

Thresholding

5.1 Introduction

Thresholding is the most basic method for image segmentation. It is mainly used on grayscale images to create binary images. The main challenge of the thresholding process is the choice of the threshold value. There are several methods for choosing a threshold: either manually or automatically. The most basic method of thresholding is to manually choose some number T and to compare with every pixel of the original image. If the intensity of the image is less than the number T then we replace that pixel with a black pixel. If it is greater than T we replace with white pixel. Thus we get a black and white image. The other way of getting a threshold is using some algorithms to compute it automatically.

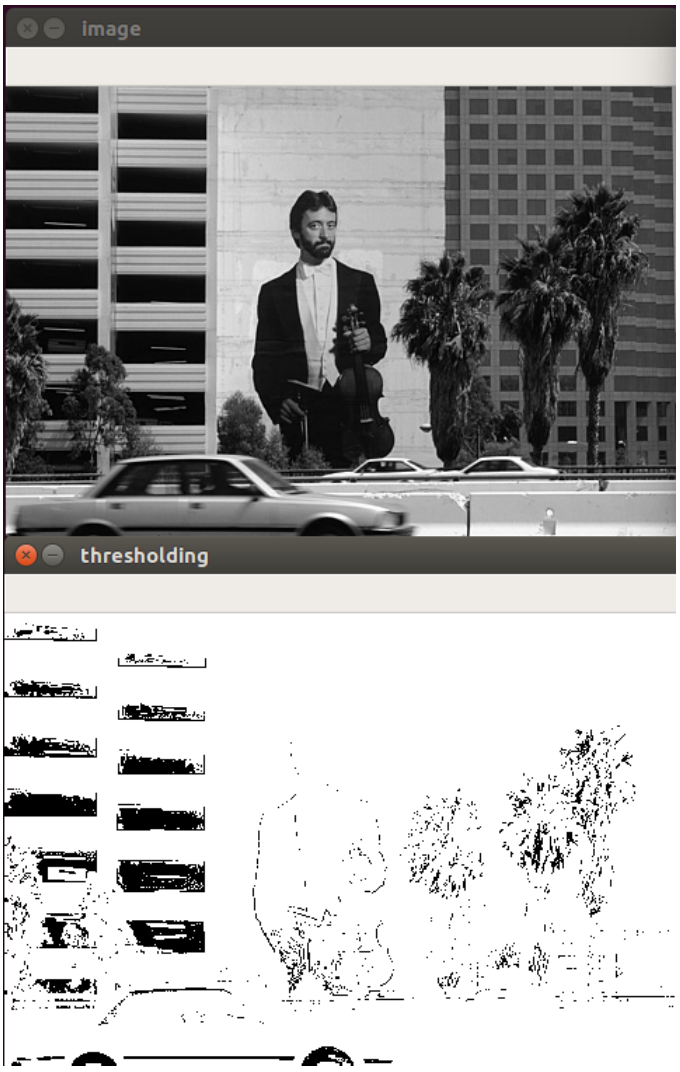
5.2 Histogram-Based Thresholding

Histogram-based thresholding is one of the most commonly used methods. This method uses a histogram to group pixels according to their frequencies. So the method assumes that on an image there are two main entities: the background and the object. The highest peak on the histogram is the indicator of the background as it usually occupies most of the image. The second highest peak of the histogram is the level of the object in the image. By choosing a threshold point in the valley between the two peaks, we assign black to all pixels that are smaller than the threshold and assign white to all remaining pixels.

We have implemented the histogram-based method by creating a list from 0 to 255 and assigning to each value the number of times we encounter a pixel with that corresponding intensity. After that we take the first biggest and second biggest values to indicate the first most common and the second most common occurring

pixel values. After that we take the average of those two and set that value as our threshold. This approach is quite intuitive, however when experimenting with it on real images there are some problems that we encounter. Let us start examining this approach by looking at Figure 5.1 on which we performed this thresholding algorithm.

Figure 5.1: The result of Histogram-based thresholding done on grayscale image



We can clearly see that the attempt is not successful. In fact, the algorithm gave the first most common pixel intensity as 11 and the second most common pixel intensity as 12. Why is that? We can see on the picture that 11 (close to black color) is not the most common shade we notice. The most common shades vary from 150 to 200, the problem is that they are spread in the range so that the black color exceeds them all. For example, we might have 1010 occurrences of pixels

with intensity 161, 1022 occurrences of pixels with intensity 162, 1150 occurrences of pixels with intensity 163, and then if we have just 1500 occurrences of pixels with intensity 11. It appears to be more frequent, however it is not giving accurate results. Now let us try to modify our algorithm in the next section so that it explores the color range more accurately.

5.3 Modified Histogram-Based Thresholding

In the previous section we discussed the histogram-based approach and the problem we encountered performing that algorithm on an image. In order for us not to give errors when the intensities are changing with minor offsets, we can combine them into groups, in other words combine them into color ranges, so that we represent more correct intensity frequency chart. We chose to divide the color range from 0 to 255 into the following groups.

1. 0 - 50 maps to 25

2. 50 - 100 maps to 75

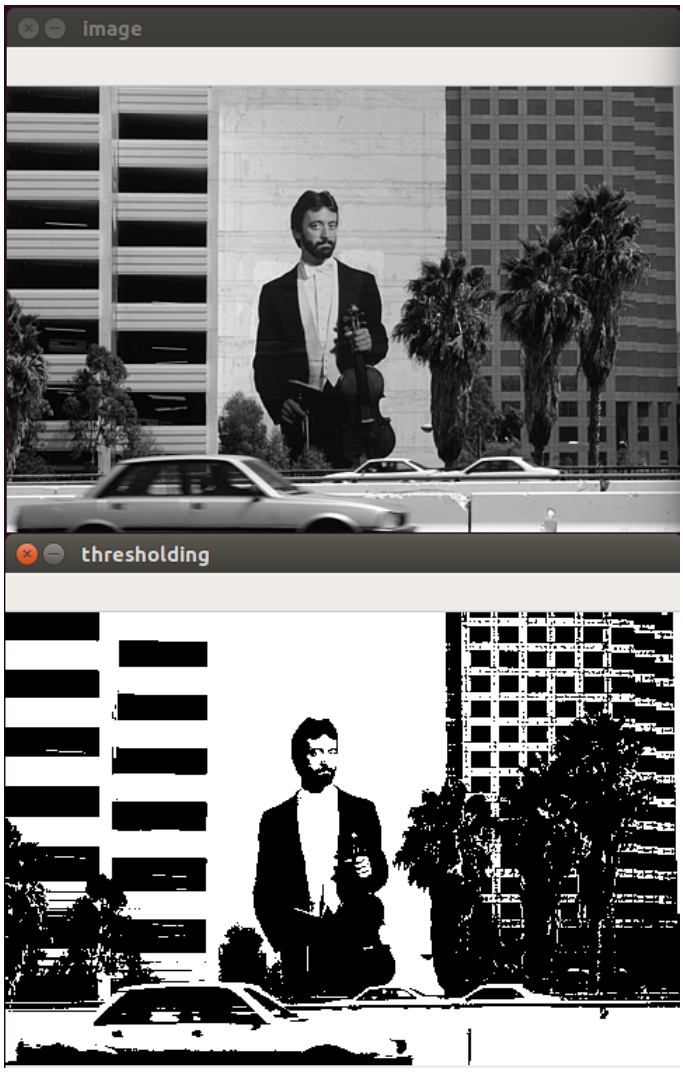
3. 100 - 150 maps to 125

4. 150 - 200 maps to 175

5. 200 - 255 maps to 225

Based on this new mapping, we can choose the first most frequent intensity and the intensity which is the second most frequent one. Then we take the average of those two and perform thresholding on that average value. See the algorithm described performed in Figure 5.2.

Figure 5.2: The result of Modified Histogram-Based Thresholding done on the grayscale image



5.4 Boundary Detection

We have already discussed the calculation of Gradient Magnitude and the techniques of thresholding. Now if we take the already calculated Gradient Magnitude of the image and perform thresholding on it with good choice of threshold, we will detect the boundaries on our image. Let us experiment with the already tested gradient magnitude (Figure 5.3) and find its boundaries by running our modified histogram-based thresholding algorithm on it. See the result in Figure 5.4.

Figure 5.3: Image 16068 from BSDS500 dataset: Gradient Magnitude done by Sobel Filter

Figure 5.4: Image 16068 from BSDS500 dataset: Boundary detection done by Modified Histogram-Based Thresholding algorithm

Chapter 6

Region Growing

6.1 Introduction

Region Growing is an algorithm in Computer Vision that takes a point (pixel) on an image and return an area from image that is considered to be the segment containing the given point. This can be done by using thresholding to see if the neighboring pixels are close to current pixel's intensity or not. If they are, we include them into our region.

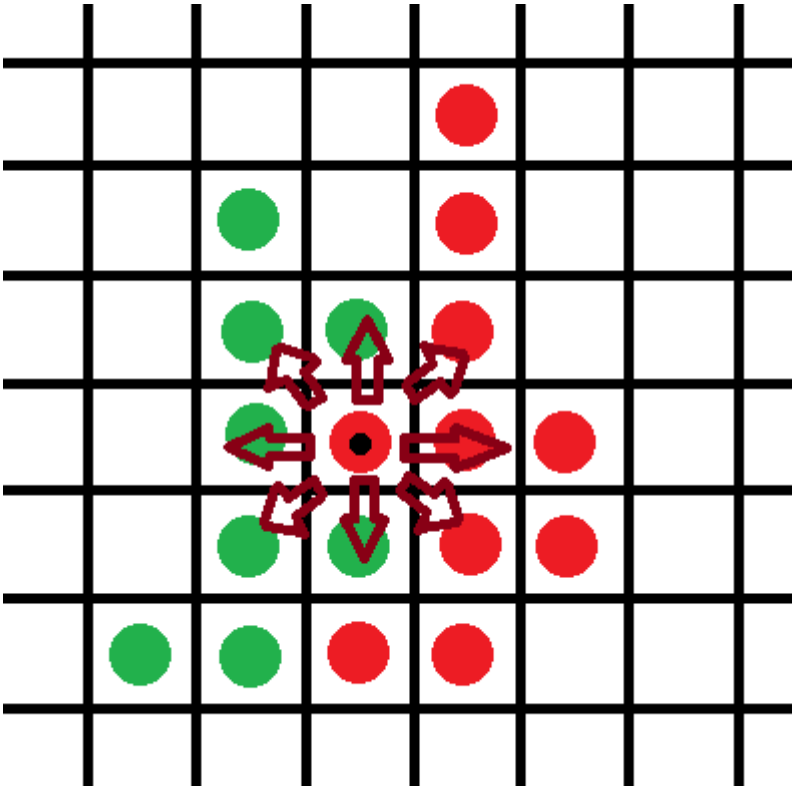
To understand it more thoroughly, let us visualize it with a simple example on a pixel map from Figure 6.1. Suppose we click on the red pixel (the one with a black dot in it), then we expand on all eight neighboring pixels. Suppose we go right, where the pixel's value is the same as ours (or near to ours, based on the value of the threshold), so we will include it into our region. But once we go left, where pixel's value is not the same or near to ours, we omit it from our region. Thus we construct our region gradually.

6.2 Algorithm Steps

We have written our own region growing algorithm with the following steps.

1. Construct a discovery matrix (initially all zeros) that will represent about the pixels we have already discovered (assign 1 if discovered)
2. Construct a pixel-queue of pixels and add to the queue the starting pixel
3. While the pixel-queue is not empty, do the following steps
4. Get the current-pixel from the queue and do the following steps if it has not been discovered yet.

Figure 6.1: Pixel Map: Red pixel with the black dot on it is the starting point. Red points have the same value as the starting point. Green points have different values from starting point



5. Make the current-pixel discovered

6. Check if the current-pixel is similar to the initial starting pixel. If it is, then color it.

7. Check color similarity for all eight neighbors of the current-pixel. If they are not discovered yet. If they were not discovered and are similar to the starting pixel, then add them to the queue

8. Return to step 4.

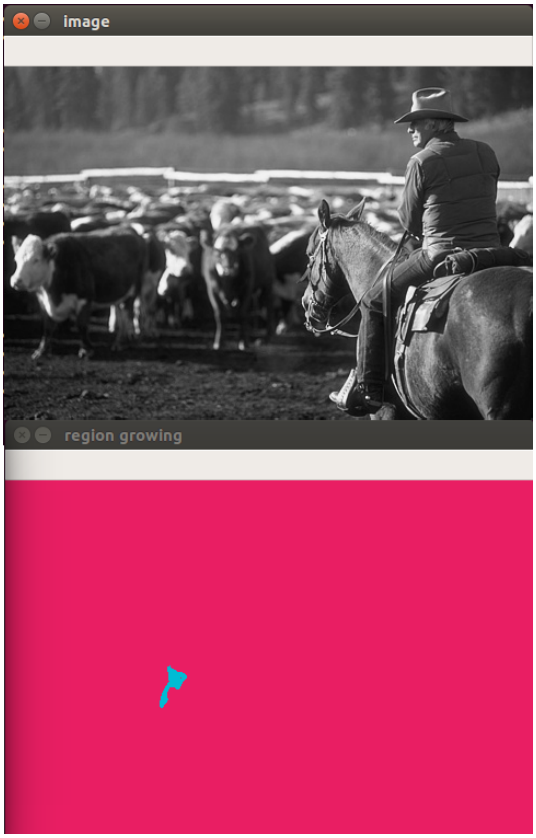
At the end we get a colored region that has grown step by step. You can see some of the results in Figure 6.2.

In order to detect multiple regions on one image, we can click on the image several times and every time add the region of the selected object to the image. Thus, we are able to choose multiple objects on one image. Please see the experiment on three different examples: Figure 6.3, Figure 6.4 and Figure 6.5.

For more information on Region Growing see [4].

Figure 6.2: Image 220075 from BSDS500 dataset: Region Growing done on grayscale image

(a) Example region 1



(b) Example region 2

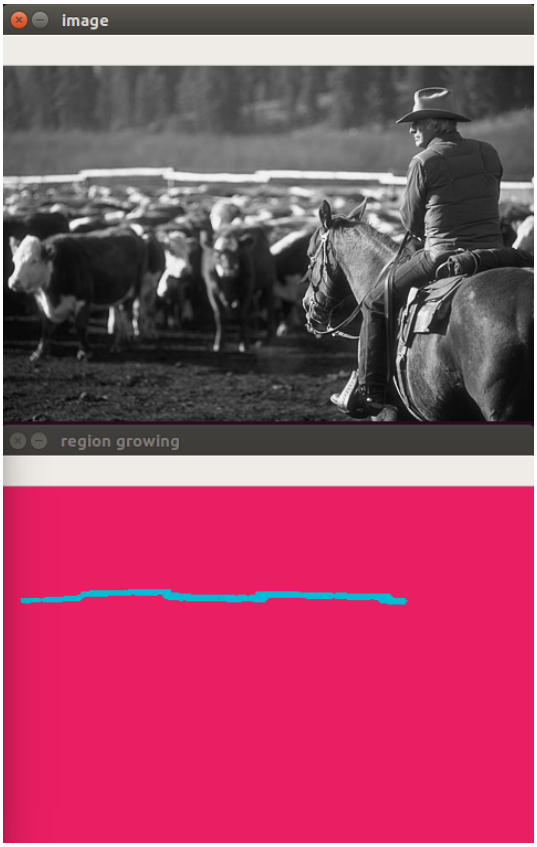


Figure 6.3: Image 169012 from BSDS500 dataset: Example 1) Region Growing Segmentation done on grayscale image by multiple clicking

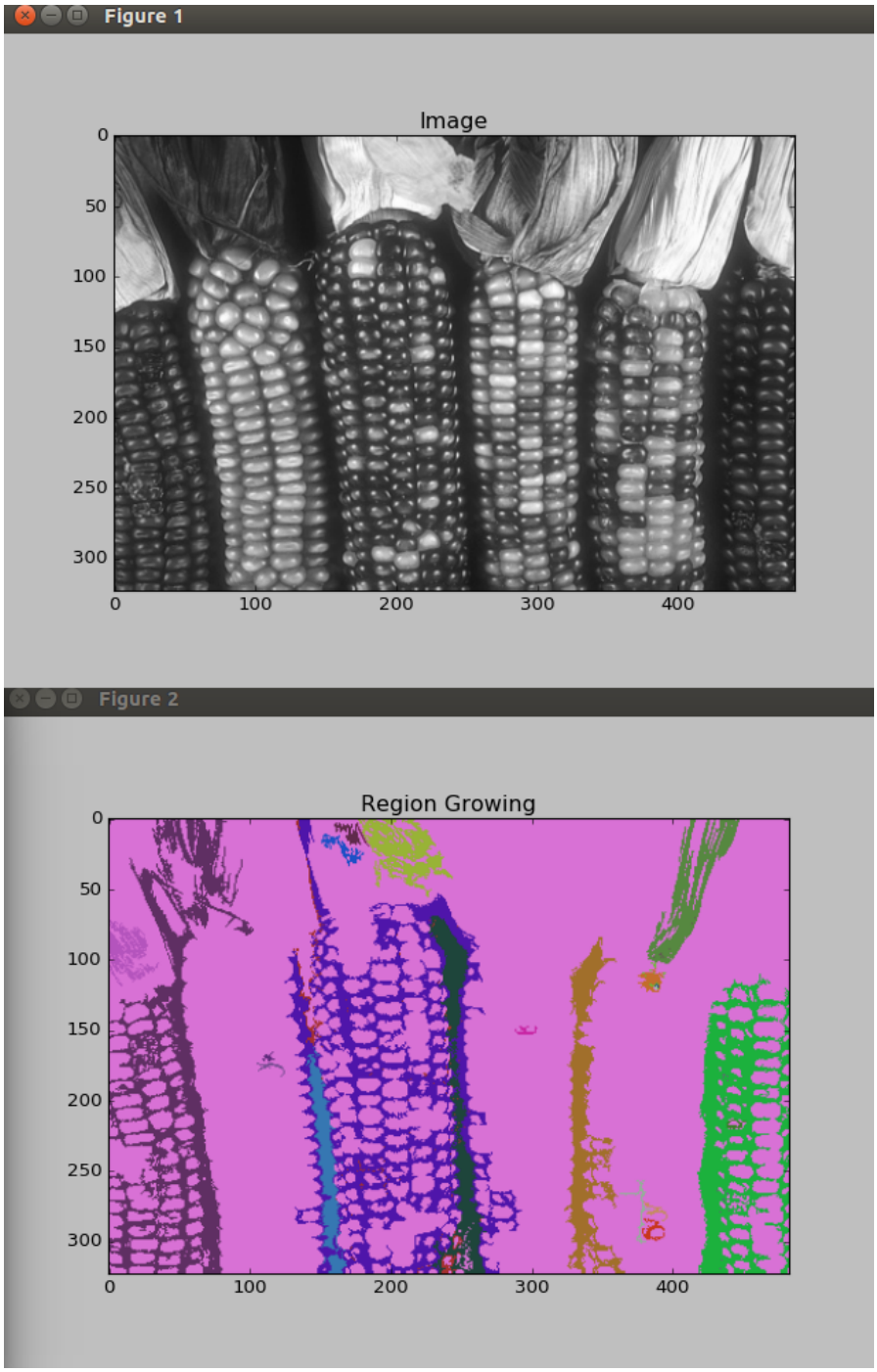
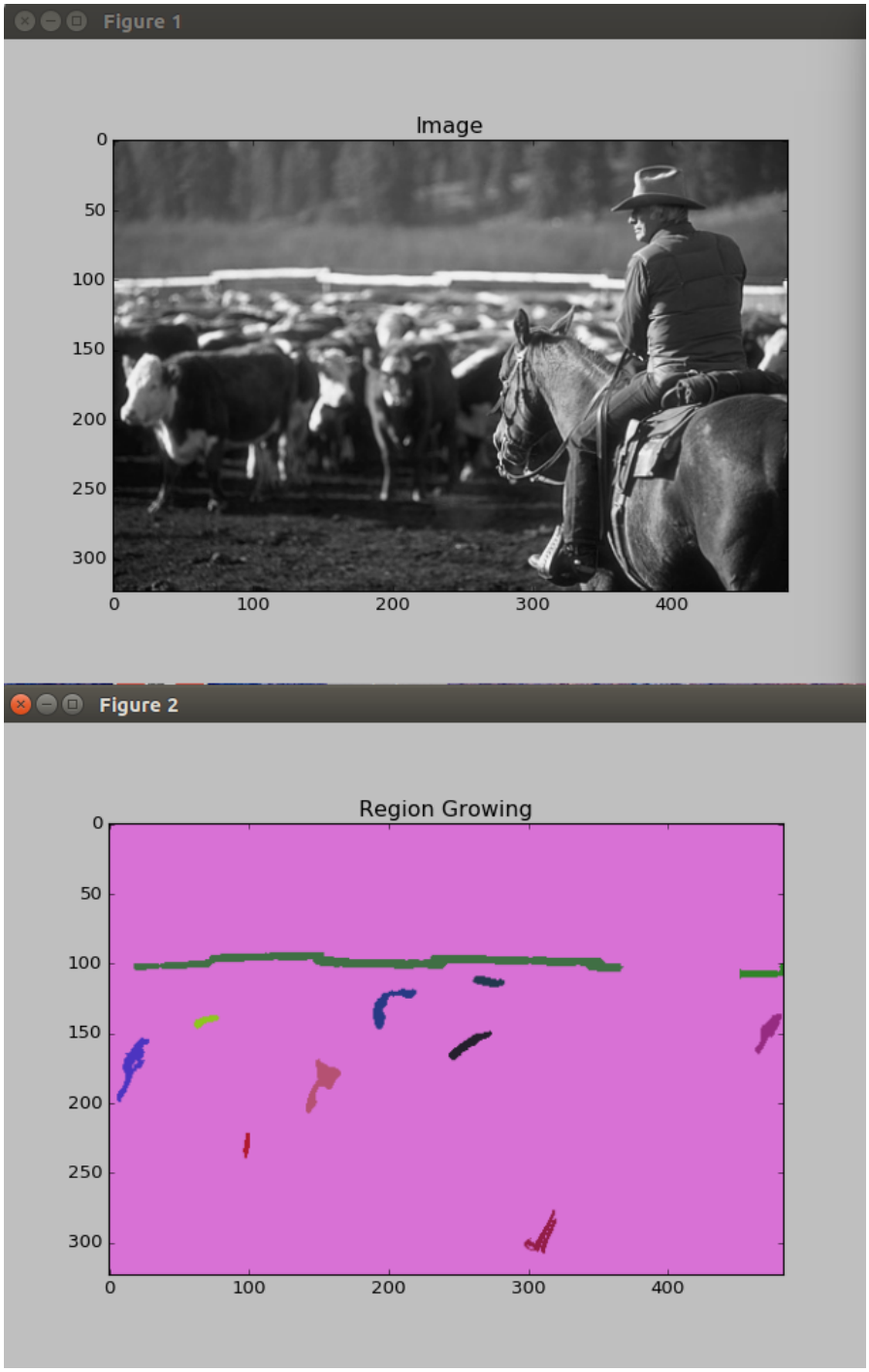


Figure 6.4: Example 2) Region Growing Segmentation done on grayscale image by multiple clicking

Figure 6.5: Image 220075 from BSDS500 dataset: Example 3) Region Growing Segmentation done on grayscale image by multiple clicking



Chapter 7

Watershed

7.1 Introduction

Another approach for digital image segmentation is to use a watershed transformation. Watershed transformation comes from the field of mathematical morphology. It is commonly used in image processing and is typically applied to a gradient magnitude of the input image.

The idea behind watershed is to look upon the grayscale image as a landscape. The gray level is considered to be the height of that pixel. Thus, correspondingly the higher grayscale pixels will have a higher altitude, and lower grayscale pixel - lower one. If we drop water on the surface, it will follow the steepest descent path until it reaches the local minimum. Around these minima, disjoint regions are formed which are called *catchment areas* or *basins*. For some pixels it is hard to determine to which basins they belong, these are called *watershed lines* and they form the boundaries between the regions.

When the watershed transform is applied to the gradient magnitude image, the pixels with the high gradient values form the boundaries of different catchment areas. These, in turn, correspond to the edges of the regions that we are interested in. Therefore, the watershed transform segments the image into areas of interest.

7.2 Plateaus

When using watershed algorithms and taking the direction of the flow of waterdrop may not bring us to the expected results because of *plateaus*. (see Figure 7.1). Plateaus are the regions of a constant gray values. In these cases the direction of steepest descent cannot be determined. There are different approaches to solve this issue. In our work we use the method of making lower-complete images.

7.3 The Approach of Lower-Complete Images

Making an image lower-complete helps us to turn any non-minimal plateaus into slopes. Any pixel in a lower-complete image has to have at least one neighbour that has a smaller value or it must be inside a *regional minimum* (see Figure 7.1). As you can see in Figure 7.2, every non-minimum plateau in (a) is replaced with a slope in (b).

Figure 7.1: Topographical relief. Image taken from Joanna Gancarczyk.

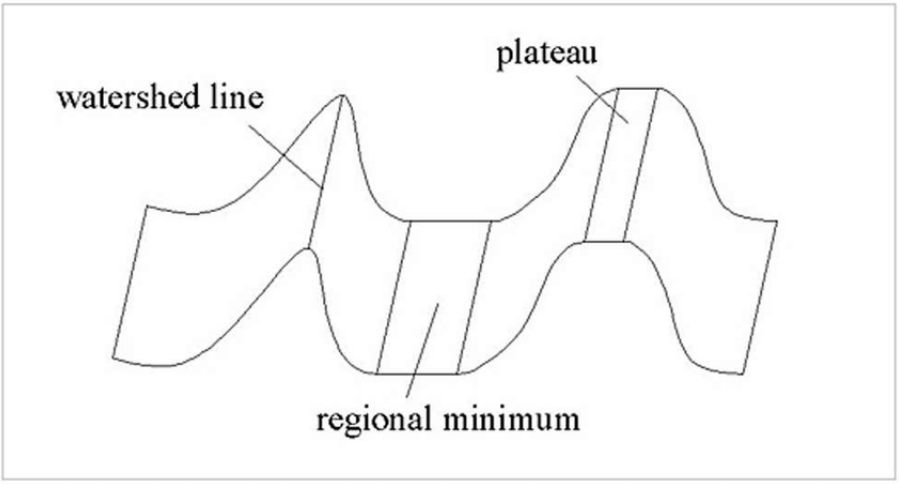
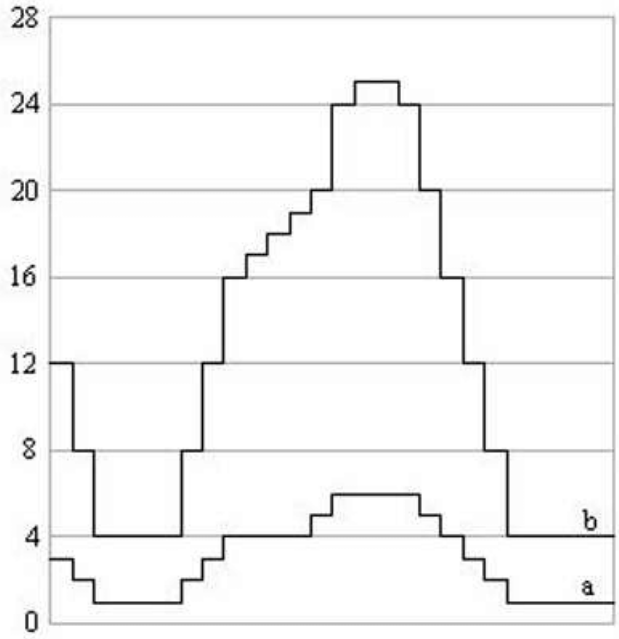


Figure 7.2: Lower-completion: a) original image, b) lower complete. Image taken from Joanna Gancarczyk.



7.4 Problem of Over-Segmentation

Though the watershed algorithm is very popular for image processing and is used in many applications, including image segmentation, edge detection, etc., over-segmentation is one of the main drawbacks of this technique. It is very sensitive to the variations of the gray level in the image, and that is why we end up with too many minor catchment areas. There are several known techniques for improving the results. In this work, we will discuss two of the most known ones: *heavy smoothing technique* and *markers*.

7.4.1 Smoothing Technique

By applying Gaussian blur after constructing a gradient magnitude image, we eliminate minor local variations. The result is a smooth image with fewer basins in the watershed result. From the results, we can see that the noise is significantly reduced.

7.4.2 Idea behind the Markers

The marker-controlled watershed segmentation expects markers for the catchment basins before running the watershed. Each initial marker has a one-to-one relationship to a specific watershed region; thus the number of markers will be equal to the final number of watershed regions. After segmentation, the boundaries of the watershed regions are arranged on the desired ridges, thus separating each object from its neighbors. The markers can be manually or automatically selected, but high throughput experiments often employ automatically generated markers to save human time and resources.

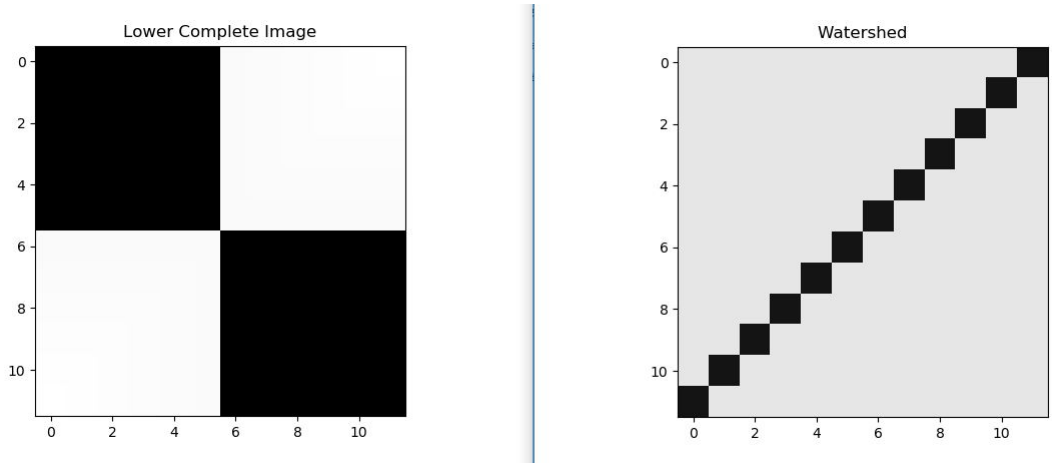
7.5 Watershed algorithm

The algorithm that we used in our code is applied on the Gradient Magnitude of the image. We start off by transforming the GM image into a lower-complete image and then we run the Watershed algorithm on it. For reducing the problem of over-segmentation, we smooth our image before running the GM algorithm. The smoothing can be done either on the gray scale image and then the Gradient Magnitude is computed or first the Gradient Magnitude is computed followed by a smoothing algorithm on it. Note that smoothing represents a convolution with a

kernel (see Chapter 3). In our code we have used a 5x5 smoothing kernel and we have smoothed the image before computing the Gradient Magnitude.

The main algorithm of performing watershed is implemented in a recursive manner. From each point we go towards the direction of the steepest lower neighbor and go down until we reach the minimum. The points from which multiple steepest paths originate, i.e. ones that belong to various regions, are the watershed lines that we will mark and show. See the result in Figure 7.3 of watershed algorithm on a simple image with two minimum regions. The watershed line separates those two regions by one straight line on the image diagonal.

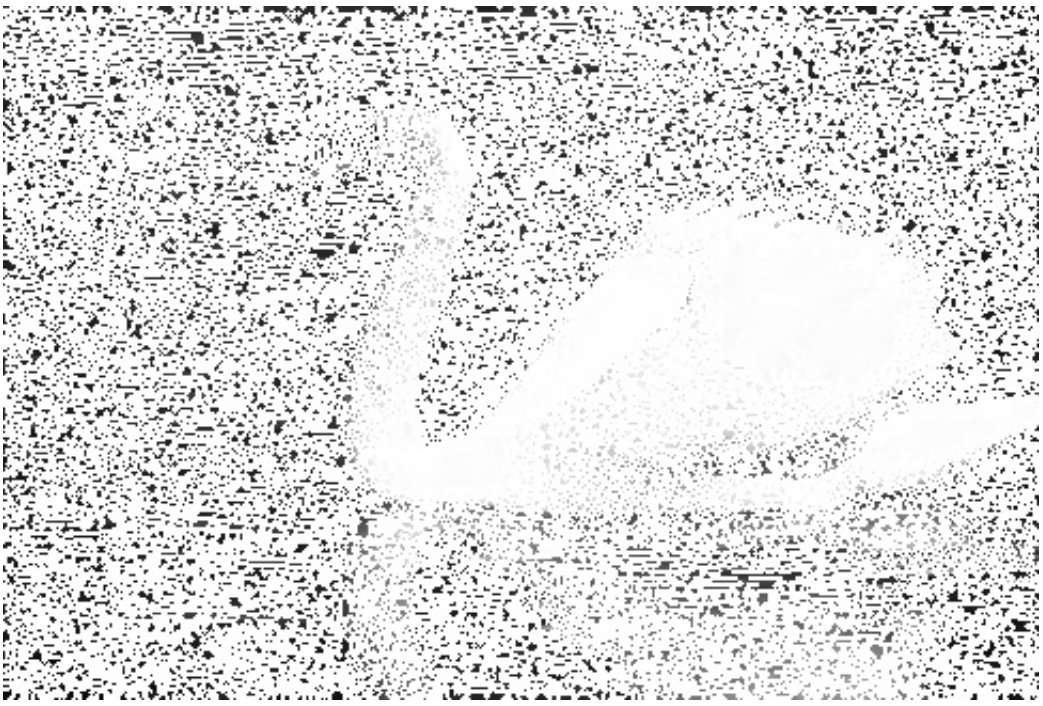
Figure 7.3: Simple image to demonstrate Lower-Complete and Watershed algorithms



Moreover, the result of the watershed algorithm implementation is shown in Figure 7.5 where we also marked the boundary points of a region, besides watershed lines. For more information about the watershed algorithm see [6].

Figure 7.4: Image 8068 from BSDS500 dataset: The GM of the image done by Sobel Filter

Figure 7.5: Image 8068 from BSDS500 dataset: Watershed of the image



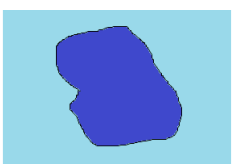
Chapter 8

Fast Marching

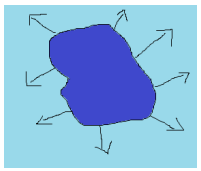
8.1 Introduction

The Fast Marching Method (FMM) is a very popular algorithm to model the evolution of a closed surface as a function of time. This method was introduced in 1995 by James Sethian. Since then, it has been used for many different applications such as robotics, medical computer vision, fluid simulation, etc. To fully understand this method, let's firstly examine the following example. Let's assume that we have an interface that is separating one region from another. We also have a speed F that tells how each point in the region is going to move. In Figure 8.1 (a) the black line separates the dark blue region from the light blue one. The speed F is given at each point of black line. Moreover, F is always positive and it always moves outwards. For example, imagine that the dark blue region is a drop of water on some surface and it is gradually expanding, Figure 8.1 (b).

Figure 8.1



(a)Region



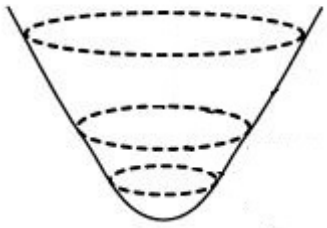
(b)Gradually expanding region

8.2 Fast Marching Approach

Let's imagine that there is a grid on top of our water drop. Now, let's suppose that when the front reaches each grid point, that time is recorded as T. Thus, a function T(x,y) gives the time of the water drop crossing the point (x,y). For

example, suppose a circle is propagating outwards. By crossing over each of the timing spots, the function T(x,y) gives a cone-shaped surface (Figure 8.2).

Figure 8.2: The cone-shaped surface that was generated by function T running on a circle



The Fast Marching Method approximates exactly this process. Starting from the initial element, we are choosing the closest one to that element and move forward by the given speed F (in the case of the constant speed we get a curve, as the "growth" is symmetric). Repeat this process over and over and by the time the process ends the entire surface would be completed. For more detailed description see [7].

8.3 Fast Marching Method

In the Fast Marching Method we assume that the front moves in the direction of the normal and the speed is always non-negative. At a given point, the function of the movement is described by the Eikonal equation

$$||\nabla u(x)|| = \frac{1}{f(x)} \tag{8.1}$$

where u is the function of time, f is the speed and x is the given point.

The FMM implementation is very similar to Dijkstra's algorithm. This is a popular method for computing shortest paths since 1950s. To understand the algorithm, suppose there is a cost for entering each node. The algorithm is as follows:

1. The starting point is in a set called "Visited".

2. Name all the grids that are one point away "Neighbors".

3. Find all the costs of reaching the Neighbors.

4. Take the smallest of the Neighbors and call it Visited. Then take all the Neighbors of that point which are not Visited and go to step 3. This is repeated until all the points are Visited.

In case of FMM at every step of the algorithm a point is marked as "Visited" or "NarrowBand". Visited points are the already computed points. NarrowBand points are still pending evaluation. All the other points are given infinite values and their time is still not calculated. First, the starting point is marked as Visited and the time cost is set to zero. Then, the algorithm is as follows:

1. From the NarrowBand points the one with the smallest time cost is selected.
2. This point is marked as Visited.
3. The neighbours of this element are calculated and they are updated if necessary.
4. Steps 1-3 are repeated.

During the update process, the Eikonal equation (8.1) is approximated. The four neighbours of an updating point are considered and those are selected that are smaller than the updating point. To solve the PDE in Equation 8.1, we approximate $||u(x)||$ in the following way:

$$||\nabla u(x)|| = \sqrt{\left(\frac{u(x_0) - u(x)}{\triangle x}\right)^2} \tag{8.2}$$

where $u(x_0)$ is the time cost of the neighbour and $\triangle x$ is the difference of the pixels. After that we get the following equation which has to be solved for $u(x)$:

$$\left(\frac{u(x_0) - u(x)}{\triangle x}\right)^2 = 1/f(x)^2 \tag{8.3}$$

In our case we consider $\triangle x$ as 1, because the difference of the neighbouring pixels is 1. For the speed the value of the gradient magnitude of an image ($gm(x)$) at that point is considered.

In our examples we took $f(x) = 1/(gm(x)^2 + 1)$ and $f(x) = e^{-gm(x)}$. After simplifying (Equation 8.3) we get a quadratic equation. For solving that equation we need to compute the discriminant. If the discriminant is non-negative we take the maximum of the computed values. If the discriminant is negative we take $1/f + min(u(x_0))$. You can see the difference that the choice of $f(x)$ makes in the examples in Figures 8.3 and 8.4. For more information on Fast Marching see [1].

Figure 8.3: Image 8068 from BSDS500 dataset: FMM when $f(x) = 1/(gm(x)^2 + 1)$, threshold $T_1 = 100, T_2 = 200, T_3 = 300, T_4 = 400, T_5 = 500, T_6 = 600, T_7 = 800$

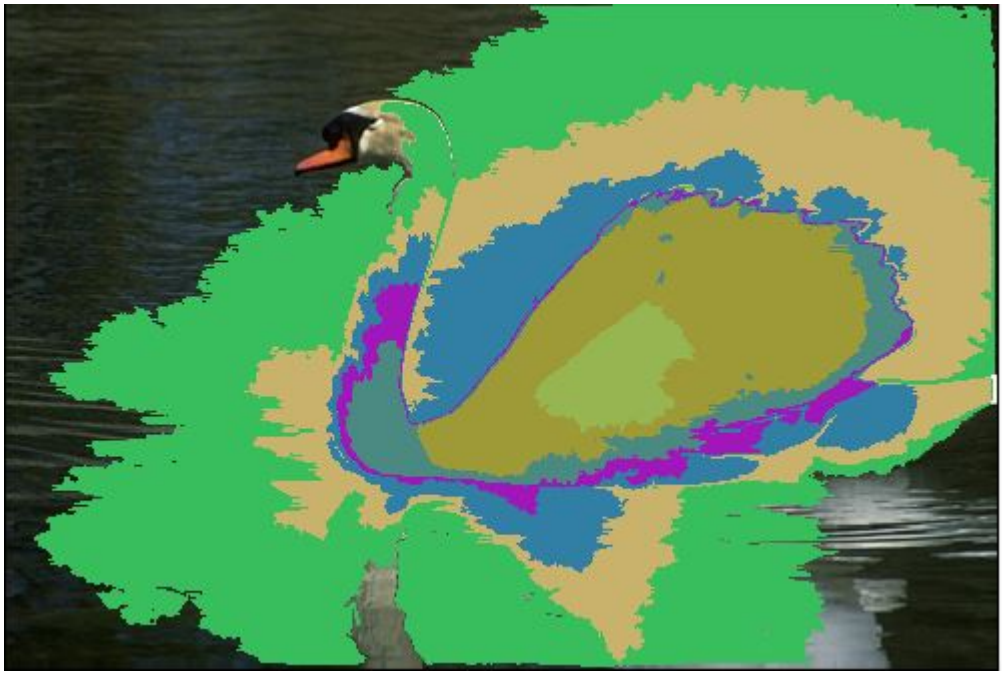
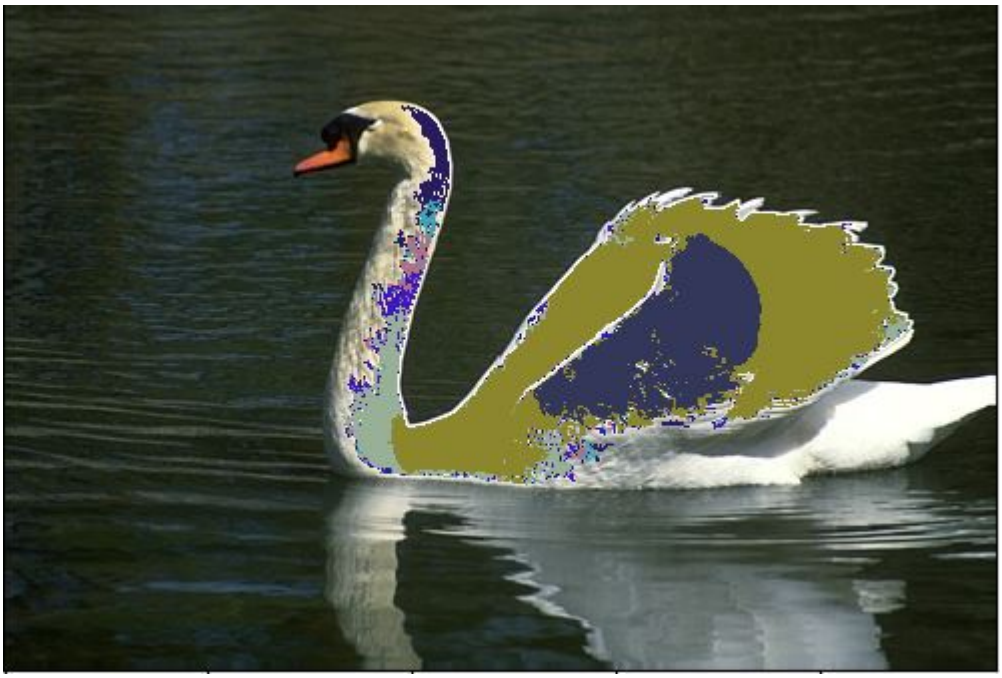


Figure 8.4: Image 8068 from BSDS500 dataset: FMM when $f(x) = e^{-gm(x)}$, threshold $T_1 = 100, T_2 = 200, T_3 = 300, T_4 = 400, T_5 = 500, T_6 = 600, T_7 = 800$



Chapter 9

Fast Dashing

9.1 Introduction

As discussed in the last two chapters, watershed and fast marching are two different algorithms that have varying techniques. Both of them are efficient in cases of certain conditions. Fast Dashing is a method where at first we perform Watershed and then using the regional information from watershed result we perform Fast Marching on the Gradient Magnitude of the image.

9.2 Merging two algorithms

The result from our previously discussed watershed algorithm of just detecting the watershed lines will not be enough for Fast Dashing, because we also need the data about which pixel belongs to which region. The approach that we have chosen is editing the previous watershed code to also extract an identifier for each region. Thus, any pixel will have a single identifier and all the pixels with the same identifier will be considered to be in the same region. To visualize this we mapped each identifier into a random color. See the example in Figure 9.1.

After attaining the identifiers for each region on every pixel, we then perform Fast Marching in a modified manner to make use of these identifiers. The only thing that we change from our Fast Marching implementation is that we consider the time required to reach a pixel within the same region as zero. Thus the NarrowBand will expand, letting watershed regions be included inside the layer easily. See the result of Fast Dashing in Figure 9.2.

For more information on Fast Dashing see [9].

Figure 9.1: Watershed region coloring with identifiers

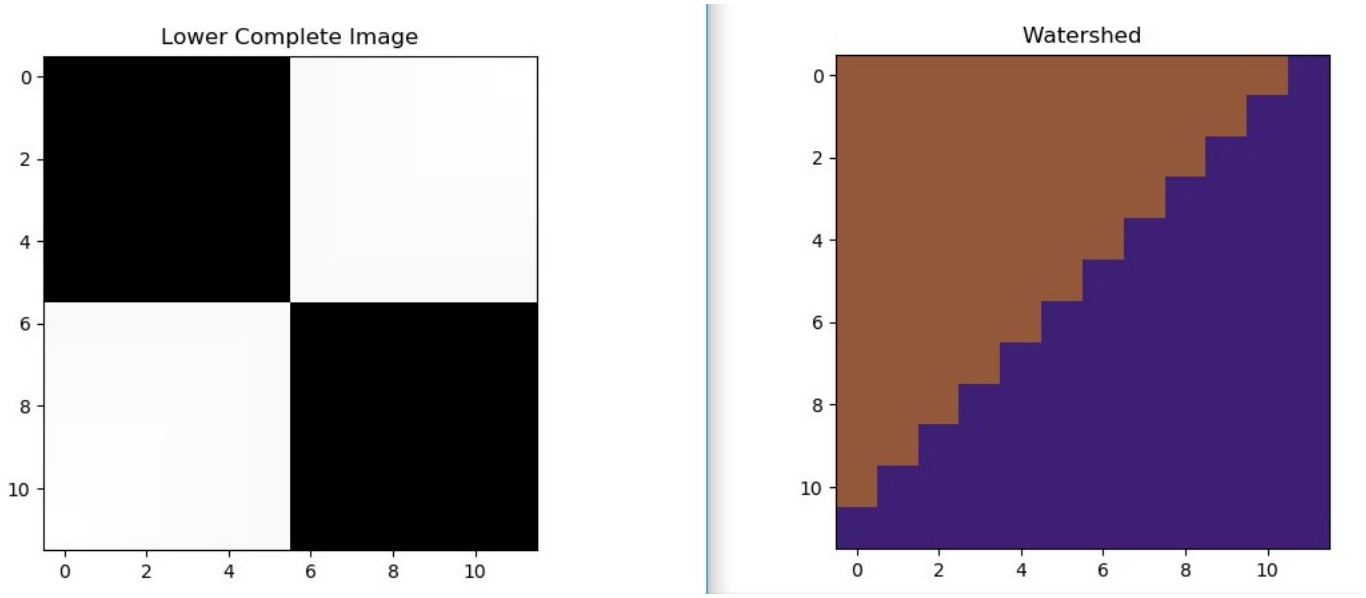
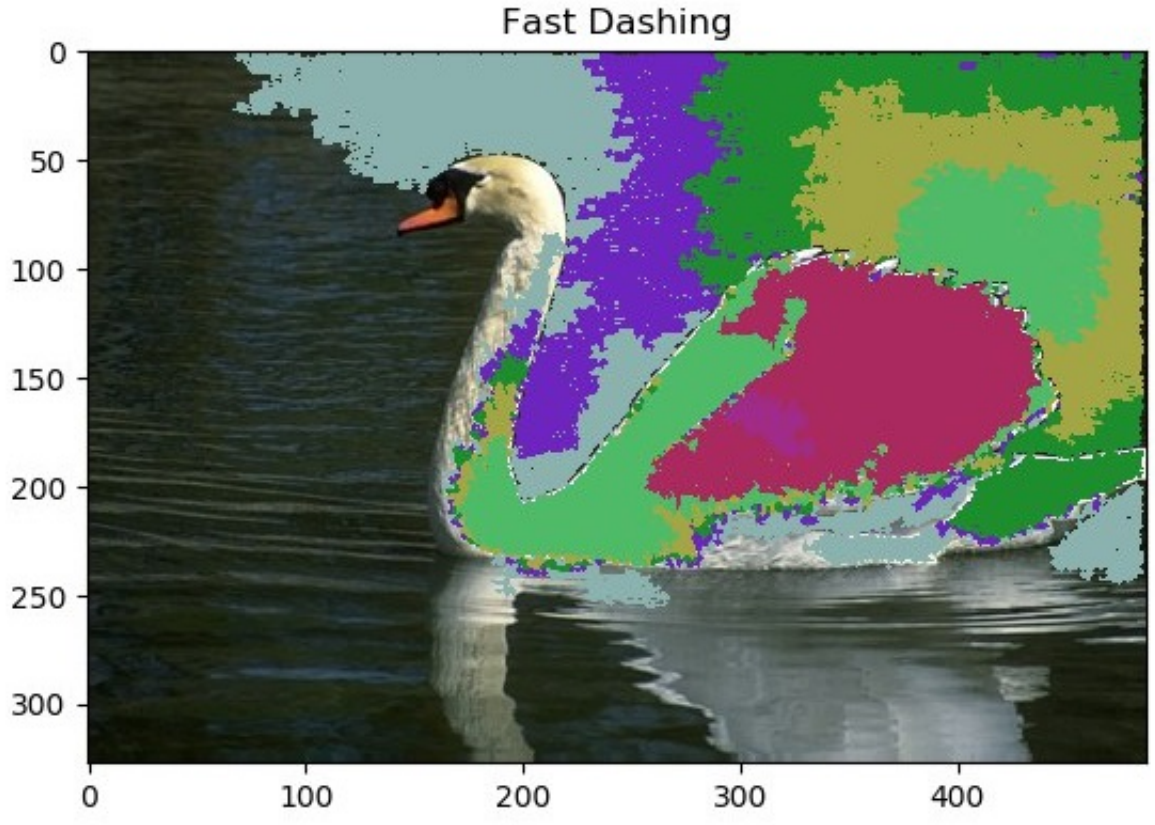


Figure 9.2: Image 8068 from BSDS500 dataset: Fast Dashing when $f(x) = 1/(gm(x)^2 + 1)$, threshold $T_1 = 100$, $T_2 = 200$, $T_3 = 300$, $T_4 = 400$, $T_5 = 500$, $T_6 = 600$, $T_7 = 800$



Chapter 10

Results, Comparison and Conclusions

10.1 Results

Let us take several images from BSDS500 dataset and experiment on them with the codes that we have written throughout our project. Below you can see some of the pictures that we have used for running our algorithms along with the results.

- a - image
- b - grayscale
- c - Prewitt Gx
- d - Prewitt Gy
- e - Prewitt GM
- f - Sobel Gx
- g - Sobel Gy
- h - Sobel GM
- i - Boundary Detection (threshold=30 on GM)
- j - Fast Dashing $f(x) = 1/(gm(x)^2 + 1)$
- k - Fast Marching $f(x) = 1/(gm(x)^2 + 1)$
- l - Modified Thresholding
- m - Region Growing
- n - Watershed - only watershed lines
- o - Watershed - with boundaries marked

Figure 10.1: Car

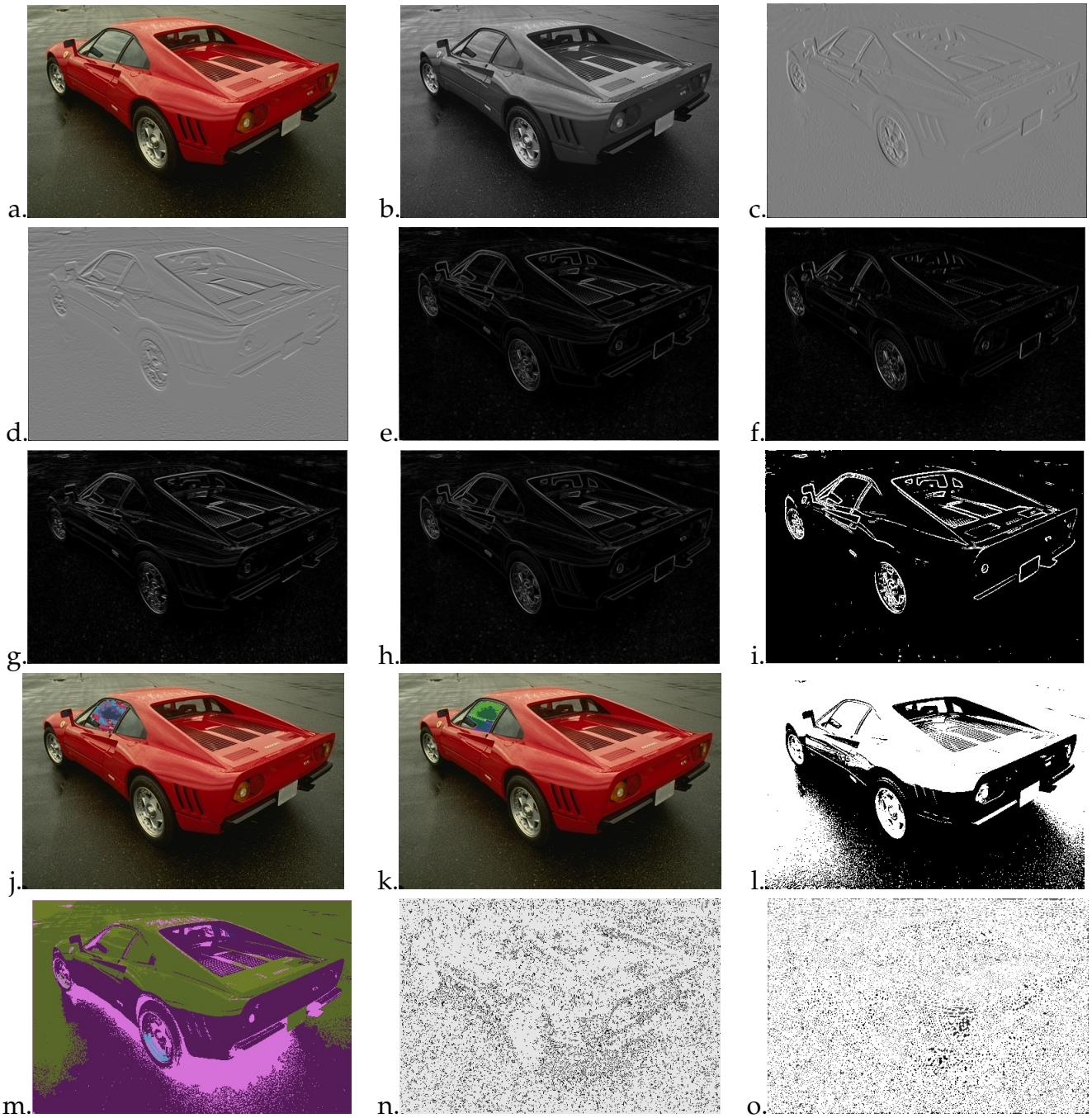


Figure 10.2: Plane

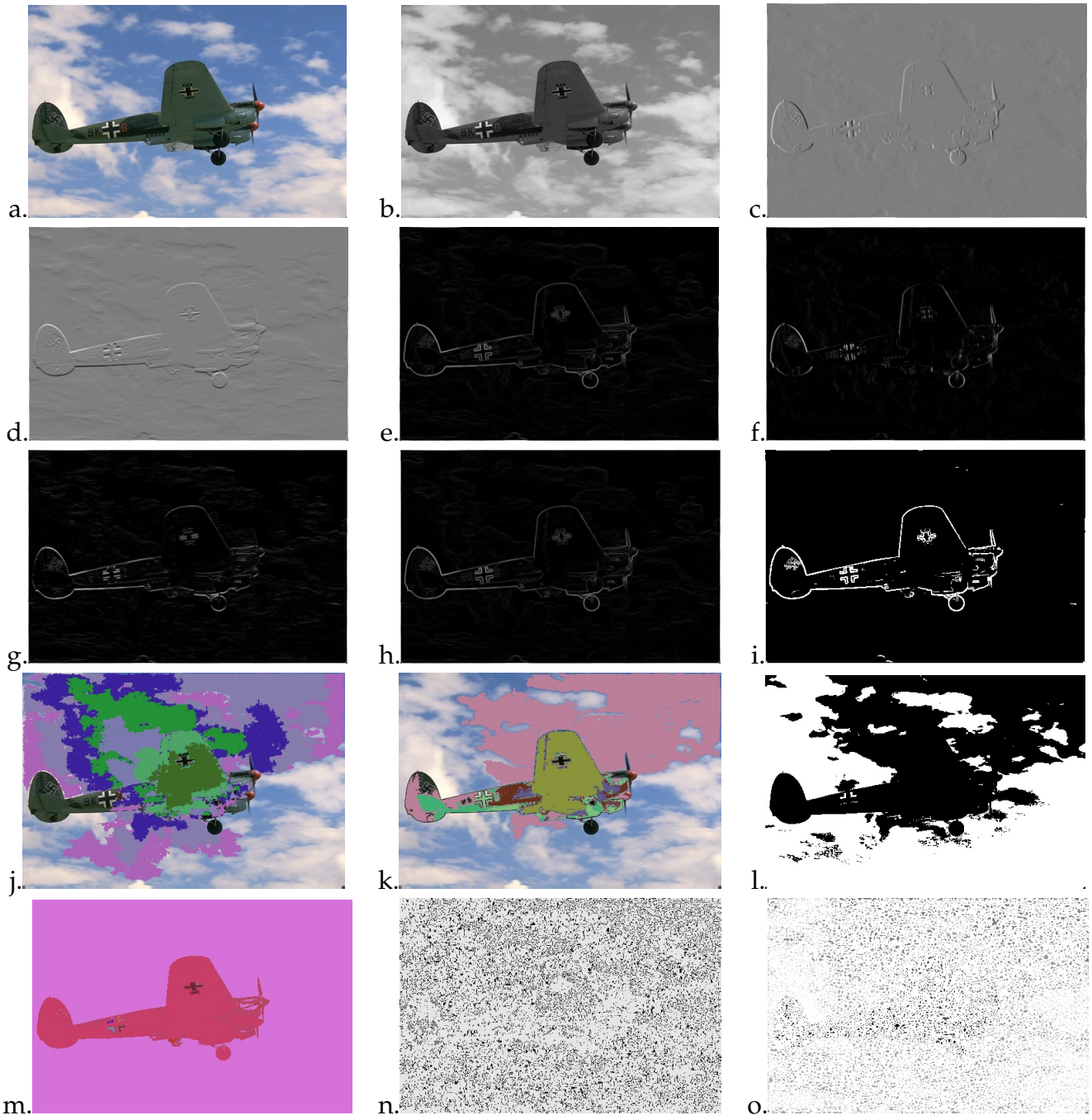


Figure 10.3: Bear

Figure 10.4: Giraffe

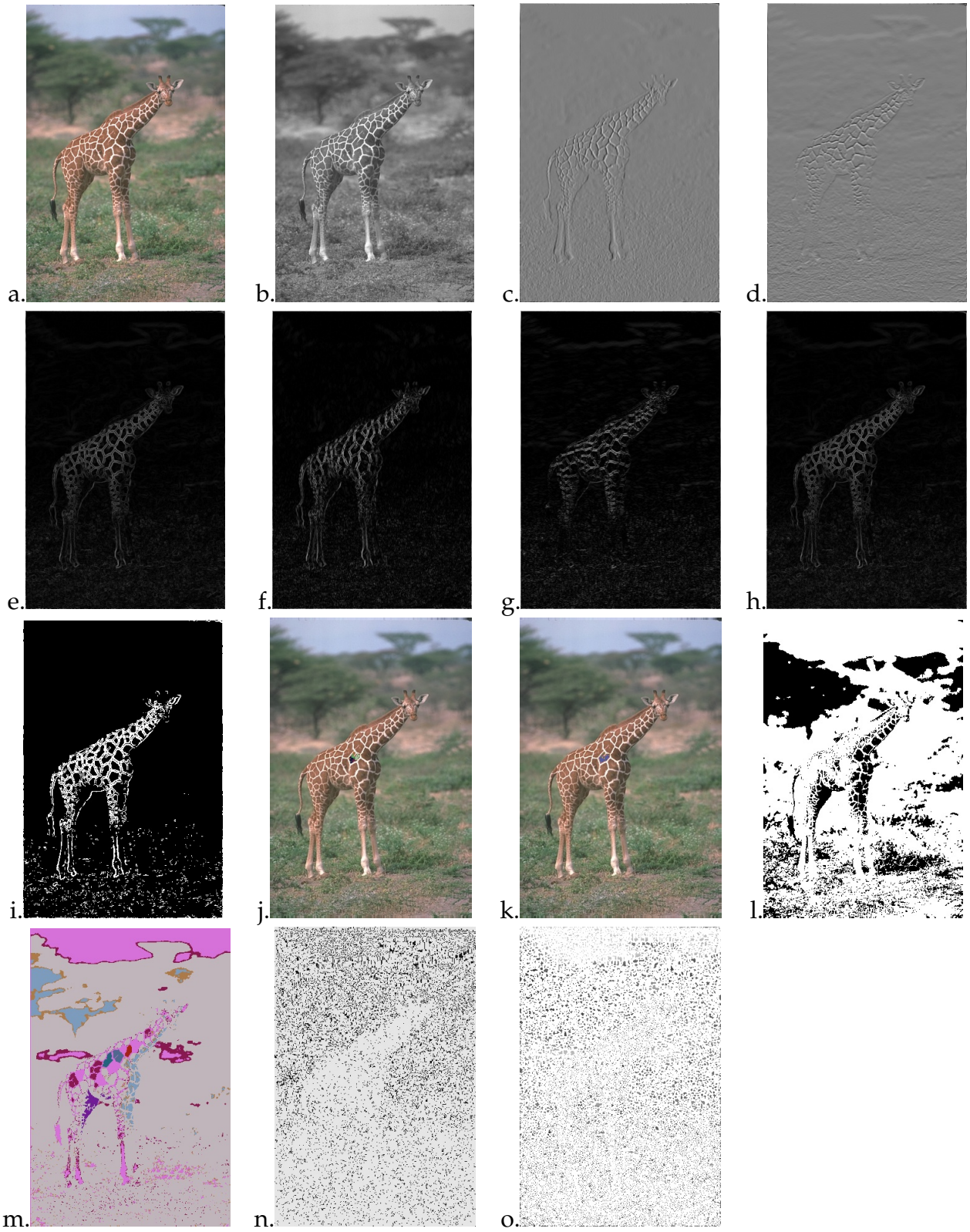


Figure 10.5: Parrot

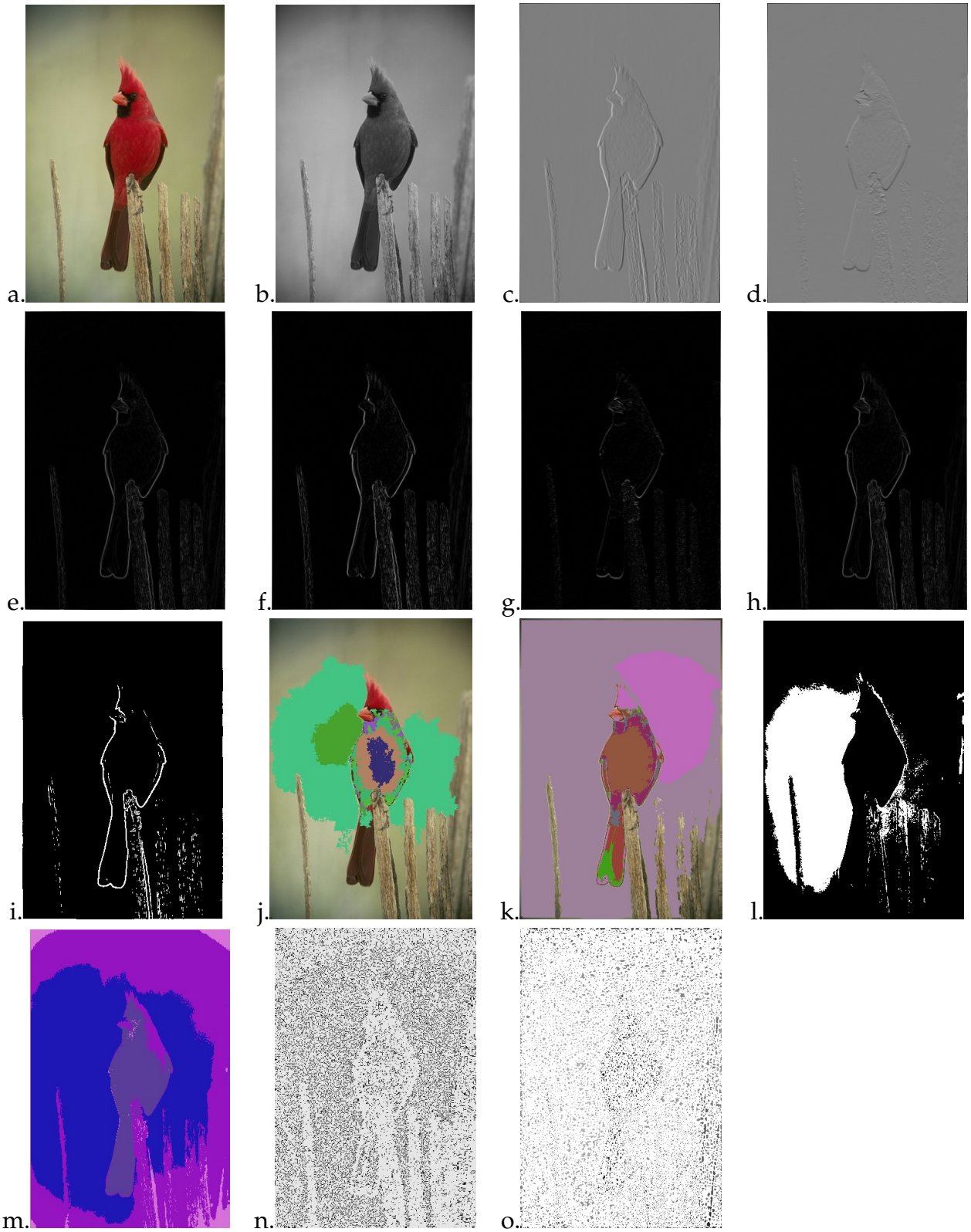
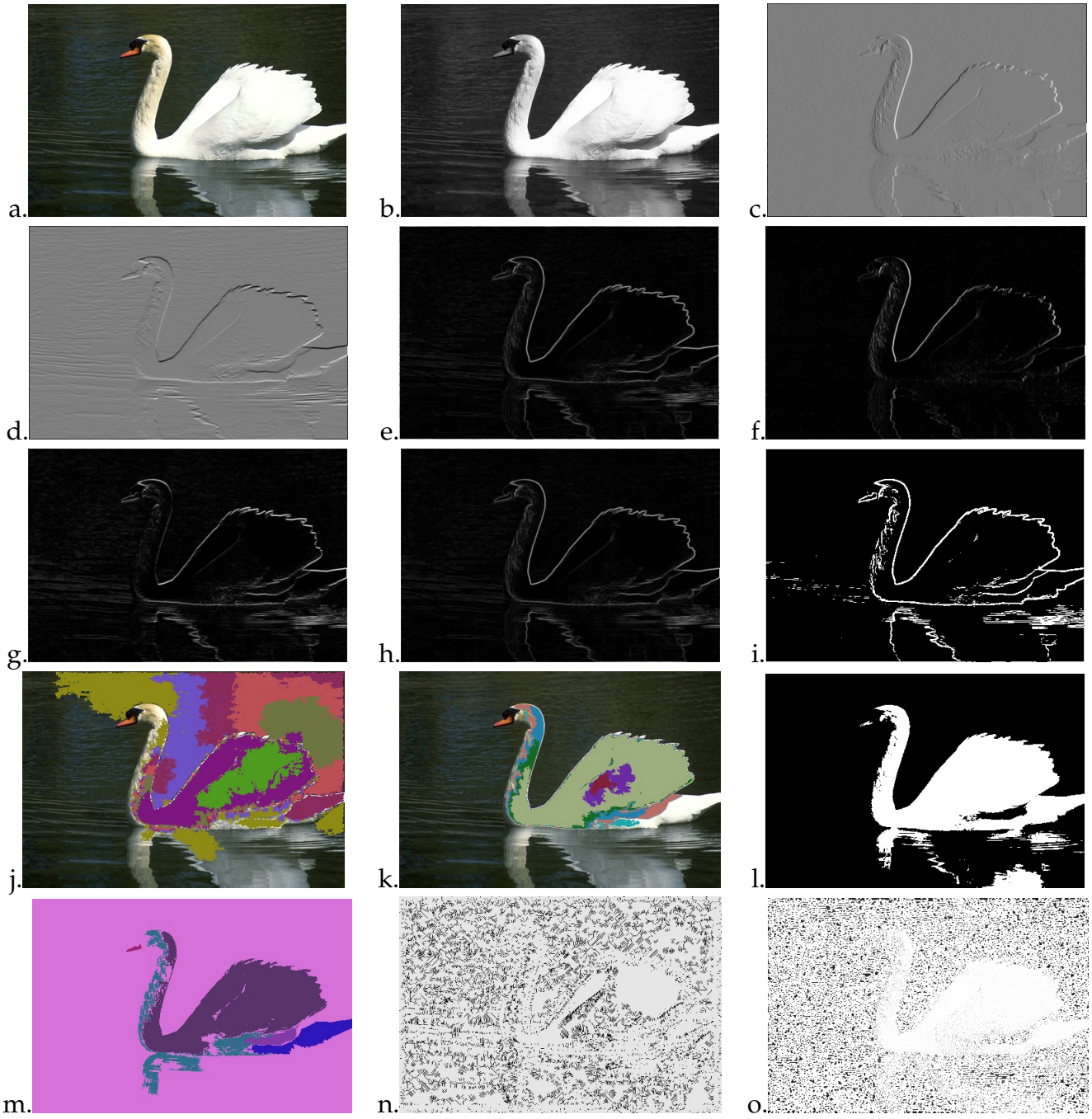


Figure 10.6: Swan



10.2 Conclusions

To sum up our results, we have experimented on many techniques and wrote the Python code for those algorithms by hand. We acquired great knowledge on Python language, image segmentation and image processing in general. We have also learned how to use scholarly articles and make the best use of information in there.

From the algorithms discussed, we believe Fast Marching and Fast Dashing are quite efficient and effective. Watershed algorithm requires more work in terms of reducing over-segmentation, which can be a future analysis material for us. Gradient magnitude is fundamental for most of the techniques that we covered and it is also a good edge detection result in itself. Region Growing and modified thresholding required as much coding workload as Fast Dashing and Fast Marching, however being very simple in their descriptions. Overall, the project covers various segmentation algorithms that give useful results on grayscale images.

Bibliography

[1] Bærentzen, J. A. On the implementation of fast marching methods for 3d lattices.

[2] Chaple, G. N., Daruwala, R., and Gofane, M. S. Comparisions of robert, prewitt, sobel operator based edge detection methods for real time uses on fpga. In *2015 International Conference on Technologies for Sustainable Development (ICTSD)* (2015), IEEE, pp. 1–4.

[3] Drimbarean, A., Corcoran, P., and Steinberg, E. Image blurring, Dec. 23 2008. US Patent 7,469,071.

[4] Hojjatoleslami, S., and Kittler, J. Region growing: a new approach. *IEEE Transactions on Image processing 7*, 7 (1998), 1079–1084.

[5] Martin, D., Fowlkes, C., Tal, D., and Malik, J. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision* (July 2001), vol. 2, pp. 416–423.

[6] Meijster, A., and Roerdink, J. B. A disjoint set algorithm for the watershed transform. In *9th European Signal Processing Conference (EUSIPCO 1998)* (1998), IEEE, pp. 1–4.

[7] Sethian, J. A. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences 93*, 4 (1996), 1591–1595.

[8] Xue, W., Zhang, L., Mou, X., and Bovik, A. C. Gradient magnitude similarity deviation: A highly efficient perceptual image quality index. *IEEE Transactions on Image Processing 23*, 2 (2014), 684–695.

[9] Yeghiazaryan, V., and Voiculescu, I. The use of fast marching methods in medical image segmentation. Tech. rep., Tech. Rep. CS-RR-15-07, Department of Computer Science, University of Oxford, 2015.