Report: Assignment #1

Course: CAP 5400 – Digital Image Processing

Name: John Doe

Report outline

1. Introduction
    1.1. Variable thresholding algorithm
    1.2. Color binarization algorithm
2. System description
    2.1. Implementing the main function
    2.2. Implementing the operation functions
3. Results
    3.1. Regular thresholding in ROI
    3.2. Variable thresholding
    3.3. Performance of the thresholding algorithm
    3.4. Color binarization
    3.5. Performance of the color binarization algorithm
4. Source code

1. Introduction

The goal of this assignment is to extend the program developed in the previous one so that it includes color binarization and variable thresholding. Also the user must have the capability to apply the operations in a specific region of the image (region of interest – ROI) instead of the whole image.

1.1. Variable thresholding algorithm

This algorithm calculates the average pixel value of a window. This mean, increased by a user defined fixed amount, is then used as a threshold. The following equation, describes this operation.

$$\hat{I}(i,j)=\begin{cases} 0 & I(i,j)<T+\text{window\_mean}(W) \\ 255 & \text{otherwise} \end{cases}$$

The calculation of the window mean is performed in 6 steps:

A. The ROI is set in such a way that all parts of the window (W) will be inside the image at all times as well as one pixel more that is needed for the incremental averaging.

1. if $x-\frac{(W+1)}{2}<0$ then $x\leftarrow\frac{(W+1)}{2}$
2. if $y-\frac{(W+1)}{2}<0$ then $y\leftarrow\frac{(W+1)}{2}$
3. if $x+sx+\frac{(W+1)}{2}>height(I)$ then $sx\leftarrow height(I)-x-\frac{(W+1)}{2}$
4. if $y+sy+\frac{(W+1)}{2}>width(I)$ then $sy\leftarrow height(I)-y-\frac{(W+1)}{2}$

B. The row just outside of the ROI is calculated by averaging all the pixels in the window and stored in a temporary matrix (TM). In this and the following step the algorithm calculates the pixels beyond the columns defined by the ROI so that the calculations on steps D and E will be simpler.

1. do $i\leftarrow y-\frac{(W+1)}{2}$ to $y+sy+\frac{(W+1)}{2}$
2. do $j\leftarrow(x-1)-\frac{(W-1)}{2}$ to $(x-1)+\frac{(W-1)}{2}$
3. $TM(i,x-1)\leftarrow TM(i,x-1)+I(i,j)$

C. The rows of the temporary matrix below the first one are calculated incrementally.

1. do $i\leftarrow y-\frac{(W+1)}{2}$ to $y+sy+\frac{(W+1)}{2}$
2. do $j\leftarrow x$ to $x+sx$
3. $TM(i,j)\leftarrow TM(i,j-1)-I\left(i,j-\frac{(W+1)}{2}\right)+I\left(i,j+\frac{(W-1)}{2}\right)$

D. The column just outside of the ROI is calculated by averaging all the pixels in the window using as an input the temporary matrix calculated on the former step and is stored in the averaged_image matrix.

1. do $j \leftarrow x$ to $x+sx$
2. do $i \leftarrow (y-1)-\frac{(W-1)}{2}$ to $(y-1)+\frac{(W-1)}{2}$
3. $AM(y-1,j) \leftarrow AM(y-1,j)+TM(i,j)$

E. The following columns of the averaged_image matrix are calculated incrementally

1. do $i \leftarrow y$ to $y+sy$
2. do $j \leftarrow x$ to $x+sx$
3. $AM(i,j) \leftarrow AM(i-1,j)-TM\left(i\frac{(W+1)}{2},j\right)+I\left(i+\frac{(W-1)}{2},j\right)$

F. The averaged_image matrix is normalized by dividing each pixel with the size of the window squared.

1. do $i \leftarrow y$ to $y+sy$
2. do $j \leftarrow x$ to $x+sx$
3. $AM(i,j) \leftarrow AM\frac{(i,j)}{W^2}$

After the averaging operation has concluded the regular thresholding operation is applied.

1. do $i \leftarrow x$ to sx
2. do $j \leftarrow y$ to sy
3. if $I(i,j) \geq T+AM(i,j)$
4. $\hat{I}(i,j)=(255,255,255)$
5. else
6. $\hat{I}(i,j)=(0,0,0)$

1.2. Color binarization algorithm

This algorithm sets the value of each pixel to white (255,255,255), if its distance to the user defined color C, in the RGB color space, is smaller than the threshold (T) and to black (0,0,0) otherwise.

7. do $i \leftarrow x$ to sx
8. do $j \leftarrow y$ to sy
9. if $\sqrt{(I(i,j)_{\text{red}}-C_{\text{red}})^2+(I(i,j)_{\text{green}}-C_{\text{green}})^2+(I(i,j)_{\text{blue}}-C_{\text{blue}})^2} \geq T$
10. $\hat{I}(i,j)=(255,255,255)$
11. else
12. $\hat{I}(i,j)=(0,0,0)$

2. System description

The program was developed based on the previous implementation of assignment 0. The basic difference in this implementation is that all the operations where moved to the files operations.cpp/operations.h. In these files, a timing class is also implemented to assess the performance of the algorithms.

An additional member variable was introduced into the image class, which carries the information if an image is grayscale or not. This allowed easier error checking and simplification of the read and write functions.

2.1. Implementing the main function

The program starts by checking if it has received as an argument the name of the parameter file. If no such file was provided the program prints the following informative message and aborts.

```
Usage: image parameterfile
```

Following that the program attempts to open the parameter file and reads it line by line. For each line the program expects a comma separated input of the following type:

inputfile,outputfile,operation,operationparameters

Then according to the operation, the validity of the operation parameters is checked. The valid operations and their parameters are described below. Parameters in [ ] can be omitted.

1. resize,*rf* Grayscale only
   (a) *rf* = resize factor (Valid for 2 or 4)
2. threshold,*T*,*W*[,*x*,*y*,*sx*,*sy*] Grayscale only
   (a) *T*=Threshold
   (b) *W*=Window size (Must be odd number. If *W*=1 regular thresholding is applied, if *W*>1 variable thresholding is applied.
   (c) *x*,*y*,*sx*,*sy*=Optional parameters that define the position and size of the ROI.
3. colorbin,*T*,*R*,*G*,*B*[,*x*,*y*,*sx*,*sy*] Color only
   (a) *T*=Threshold
   (b) *R*,*G*,*B*=RGB values of the color *C*
   (c) *x*,*y*,*sx*,*sy*=Optional parameters that define the position and size of the ROI.

The definition of ROI can be omitted and then the program sets the ROI to be all of the image. If it ROI is defined the *x*,*y*,*sx*,*sy* are checked to ensure that the region is inside the image. If the origin is outside, it is reset to the upper left corner. If the region extends beyond the boundaries of the image, it is truncated to the boundaries.

After all the checking is complete, the corresponding operation is called, which processes the image and saves the results.

If at any time an error occurs (e.g. file missing, invalid parameter, invalid operation for image type) the line is ignored and the next is processed. The program prints out a progress report for each line.

A sample parameter file with errors is provided below:

```
lenna.pgm,lenna_r2.pgm,resize,2
lenna.pgm,lenna_r4.pgm,rsize,4                  (Error: "rsize" is not a valid operation.)
lenna.pgm,lenna_t100.pgm,threshold,100,1
lenna.pgm,lenna_t130.pgm,threshold,130,2        (Error: "2" is not a valid window size.)
lenna.pgm,lenna_t160.pgm,threshold,160,3
lenna1.pgm,lenna_t160b.pgm,threshold,160,5      (Error: "lenna1.pgm" file does not exist.)
book.ppm,book_t50.ppm,colorbin,50,0,0,250
lib.pgm,lib_r2.pgm,resize,2
```

The former parameter file will produce the following output:

```
Processing line [1] [15msec] Operation completed succesfully. File saved.
Processing line [2] Unknown operation in parameter file. Line ignored
Processing line [3] [15msec] Operation completed succesfully. File saved.
Processing line [4] Wrong value for operation. Line ignored
Processing line [5] [62+31msec] Operation completed succesfully. File saved.
Processing line [6] Cannot open input file or wrong file format. Line ignored
Processing line [7] [15msec] Operation completed succesfully. File saved.
Processing line [8] [0msec] Operation completed succesfully. File saved.
Total time to process all files 198 msec.
```

The program also outputs the total execution time and the execution time of each algorithm inside brackets. Line 5 has two execution times. The first one refers to the averaging algorithm and the second one to the thresholding operation.

2.2. Implementing the operation functions

All functions are implemented as returning int. Each function accepts an Image class which carries the original image, a const char* which provides the output filename and several integers for each of the parameters.

The function starts by allocating space for the new image if needed. Then the operation algorithm is run and the resulting image is saved in a file. If the save operation was successful the function returns true and false otherwise.

The regular and variable thresholding operations are implemented in the same function. The function determines which to run by checking the window size parameter.

3. Results

The program was tested with various images both grayscale and color and of varying sizes. All the operations were used, several times with different parameter values. Also a performance analysis was carried out to assess the impact of image size on the execution time of regular thresholding and color binarization. This analysis was extended to include the effect of window size in addition to image size on variable thresholding.

Some of the resulting images converted to the JPEG format can also be found in the following web page: http://www.csee.usf.edu/~kdalamag/class.html

3.1. Regular thresholding in ROI

The regular thresholding operation was thoroughly tested in assignment 0. Here a couple of results will be presented that show the thresholding operation in ROI only. The images used are that of woman where the ROI is her head and that of some plants where the ROI is the center part of the image.

Image 1: Original photo of woman

Image 2: T=120, ROI=(45,15,175,190)

Image 3: Original photo of plants (scaled down)

Image 4: T=150, ROI=(60,60,640,415)

3.2. Variable thresholding

The following images are the result of applying the variable thresholding operations on the image of a church with varying window sizes.

Image 5: Original image of church

Image 6: T=15, W=3x3

Image 7: T=15, W=5x5

Image 8: T=15, W=7x7

Image 9: T=15, W=9x9

Image 10: T=15, W=11x11

It seems that the variable thresholding operation performs a kind of edge detection which is more pronounced for larger window sizes. This is because it in effect converts to white all the pixels that are above a limit larger than their neighboring, which in turn occurs usually in edges. The following images are from a picture of 5 cubes. The window size is kept constant and the threshold parameter is changed gradually from 3 to 9. The operation is carried out in a region that contains only the cubes and not the background.

Image 11: Original photo of cubes (scaled down)

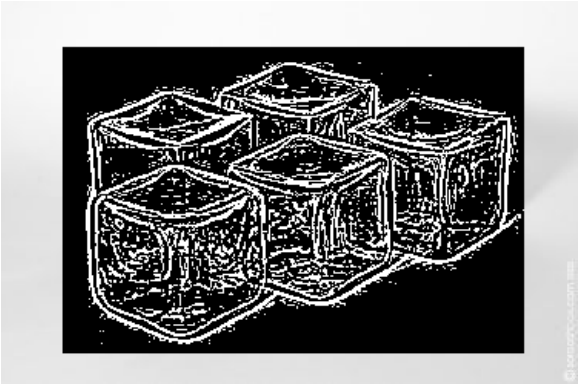

Image 12: T=3, W=5x5, ROI=(40,30,300,200)

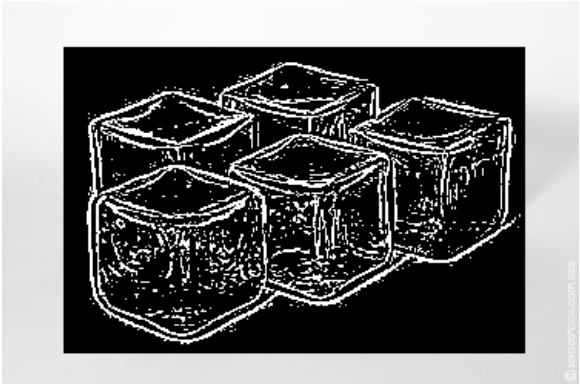

Image 14: T=5, W=5x5, ROI=(40,30,300,200)

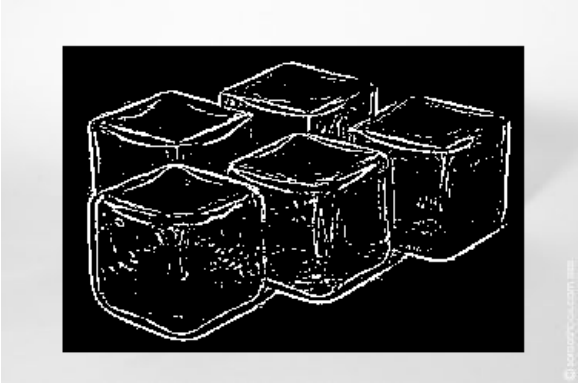

Image 13: T=9, W=5x5, ROI=(40,30,300,200)

3.3. Performance of the thresholding algorithm

In order to assess the performance of the thresholding algorithm, tests were carried out using window sizes from 1 to 21 and the following image sizes: 600x600, 500x500, 400x400, 300x300 and 200x200 pixels. The averaging part and the thresholding part were separated to evaluate the importance of each of them to the final execution time. For greater accuracy the thresholding algorithm was applied to each of these images 100 times and the resulting time was divided by 100. The following tables contain the results of these tests.

	W=3	W=5	W=7	W=9	W=11	W=21
200x200	3.12	3.13	3.28	2.97	3.13	2.97
300x300	8.13	8.12	8.13	7.97	8.12	7.66
400x400	13.59	13.44	13.44	13.28	13.44	12.81
500x500	23.44	23.28	23.28	23.43	23.13	22.18
600x600	31.25	31.09	30.94	30.94	30.78	29.53

Table 1: Average execution time in msec of averaging algorithm as a function of image size and window size

From Table 1, it is obvious that the averaging algorithm does not depend on the window size. The small decrease in larger window sizes is due to the fact that the algorithm is processing a smaller part of the image, since it needs to keep a larger distance from the edge. Chart 1 shows the dependence of averaging to the image size. We can deduce the conclusion that the execution time of the algorithm depends linearly on N^2. The same can be deducted by examining the algorithm.

	W=1	W=3	W=5	W=7	W=9	W=11	W=21
200x200	0.93	1.56	1.56	1.41	1.56	1.41	1.41
300x300	2.19	3.91	4.07	4.06	4.06	3.91	3.91
400x400	4.84	10.16	10.00	7.97	8.60	9.68	9.85
500x500	7.81	12.97	13.13	13.28	13.13	13.28	13.13
600x600	11.72	17.97	17.97	18.12	17.97	17.97	17.97

Table 2: Average execution time in msec of thresholding algorithm as a function of image size and window size

Table 2 contains the results of the thresholding operation. This operation is also independent of window size. The smaller values for W=1 are due to the fact that for each pixel of the image the variable thresholding algorithm needs to fetch an additional pixel from the averaged image and add it to the threshold parameter, before the comparison is made.

From the above results it also apparent that the averaging algorithm execution time amounts to about 62 to 67% of the total execution time.

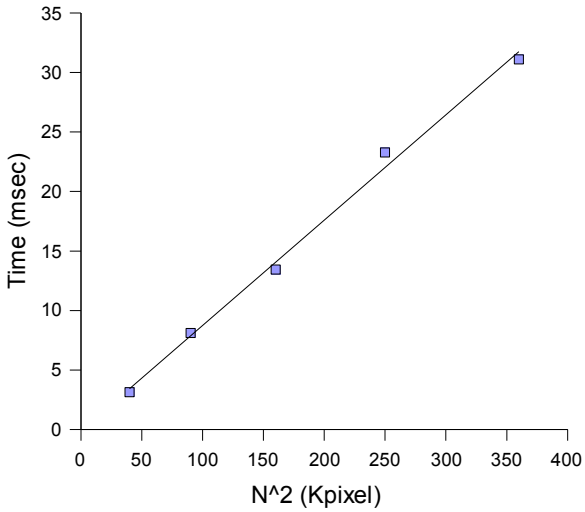

Chart 1: Pixel averaging performance as a function of image size

It is evident that we have at most two nested loops that cover at most the whole image, therefore the computational complexity of the algorithm is $O(N^2)$. Of course the algorithm is slower than regular thresholding since we have to execute the nested loops 4 times (incremental averaging on y axis, incremental averaging on x axis, normalization and thresholding). We also have to compute the first row and column to apply the incremental averaging.

In addition to that the memory requirements are larger because the regular thresholding can be applied in place since the algorithm is a point operation. On the other hand the variable thresholding is a local operation and therefore needs three times as much memory to store the intermediate temporary image and the final averaged image, that will be later used for the thresholding itself.

3.4. Color binarization

The set of pictures on the following page, illustrates the operation of color binarization. The original image is being thresholded against the red, green and blue colors. With a low threshold (T=50) the algorithm clearly separates the red, green and blue parts of the image and converts them to black while the rest are turned to white.

Unfortunately for larger values of threshold (T=250) the separation is not so good and includes some pixels with intermediate values, which some times are not the ones expected. For example by increasing the threshold on the red binarization operation, we got several pixels that were situated at the borders of the blue and green colors, while someone would expect that the additional pixels would be from the regions where the colors blend with each other. Also regions that appeared to have no color, like the black letters, where also included in the binarized image.

A border was placed after the processing on each image so that their boundaries are clearly defined.

Image 15: Original RGB picture

Image 16: T=50, C=(255,0,0)

Image 17: T=50, C=(0,255,0)

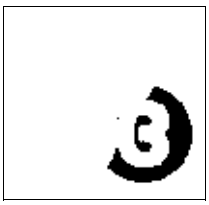

Image 18: T=50, C=(0,0,255)

Image 19: T=250, C=(255,0,0)

Image 20: T=250, C=(0,255,0)

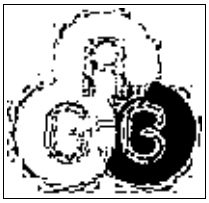

Image 21: T=250, C=(0,0,255)

In the photos below the color binarization was applied successfully to separate the oranges from the background. The algorithm was performed in a region of interest that contains only the oranges, so the plate is left untouched. The last set present the separation of the blue sea stars using color binarization.

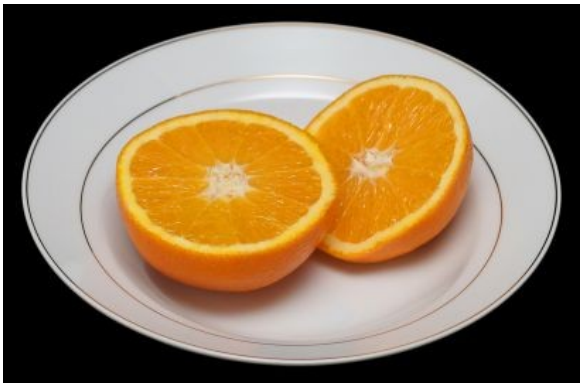

Image 22: Original photo of oranges (scaled down)

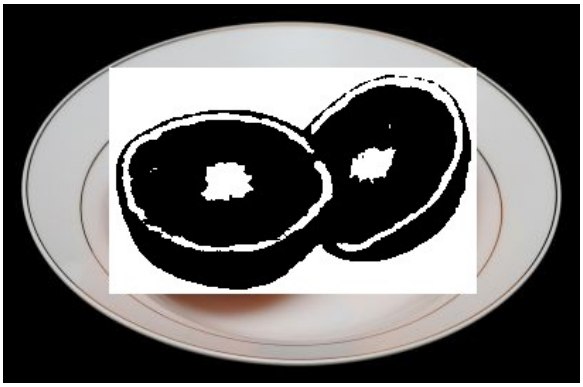

Image 23: T=100, C=(255,125,0), ROI=(75,45,260,160)

Image 24: Original photo of seastars

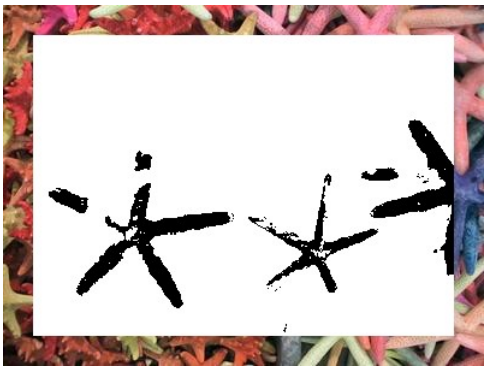

Image 25: T=200, C=0,0,255

3.5. Performance of the color binarization algorithm

In order to assess the performance of the color binarization algorithm, a test was carried out using the same image in the following sizes: 600x600, 500x500, 400x400, 300x300 and 200x200. The algorithm applied the color binarization algorithm to each of these images 100 times and the resulting time was divided by 100. The following graph shows the time for each image size (*N*). From Chart 2, it is evident that the execution time of the algorithm depends linearly on the N^2 .

The color binarization algorithm is about 20% faster than the variable thresholding operation and about 4 times slower than the regular thresholding. This is due to the fact that for each pixel the color binarization algorithm needs to compute the euclidean distance in 3D space before each comparison to the threshold.

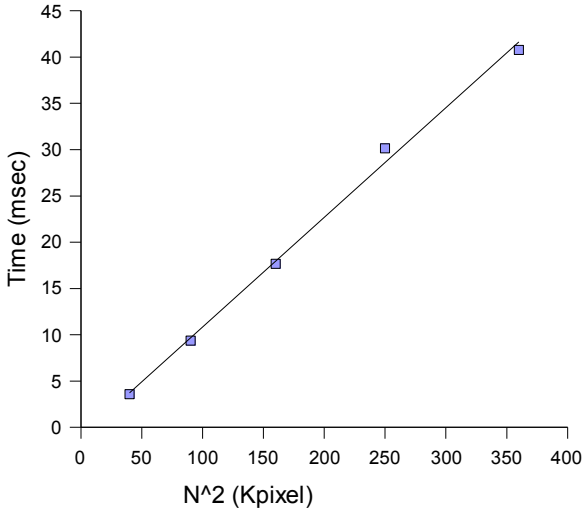

Chart 2: Color binarization performance as a function of image size