|  |  |
| --- | --- |
| Binary image rendering using halftoning  February 21, 2022 | Abstract The goal of this lab is to transform a grayscale image into a halftone image. This MATLAB script includes a halftone function that creates matrices containing black and white pixels and uses them to replace pixels in a grayscale image whose values vary. This method outputs an image that appears very similar to the input grayscale image, but only uses black and white pixels and no values in between. This report will discuss the techniques used to successfully create halftone transformations and what each output image tells us about the performance of the program. Author: Alejandro Sanchez CSE 107 – Digital Image Processing |

# Technical Discussion

## How Images are Loaded into the Program

The program starts by reading an image using the imread() function. The images that are used in this lab include “Fig0225(a)(face).tif”, “Fig0225(b)(cameraman).tif”, and “Fig0225(c)(crowd).tif”. The image files are converted to the uint8 type and assigned to a matrix, as shown below in Figure 1.

A picture containing text

Description automatically generated

Figure 1: Importing images and type casting to type uint8

After the matrices are created, they are sent to the halftone function as arguments where they will be processed by an algorithm that performs the halftone transformation. In Figure 2, matrices are set equal to the result of the halftone function. The result is a matrix of type logical.

Text

Description automatically generated

Figure 2: Assigning matrices as the halftone function’s resulting transformation

## The Structure of the Halftone Function

This section will cover the workflow and methodology of the halftone function. The function begins by assigning the input image matrix – called inputImage in the example shown in the figure below – to a matrix A. Next, the function proceeds to identify the dimensions of the image matrix using the size() function. More information on how the size() function operates on matrices can be found in the appendix. The number of pixels in the image’s x axis is identified as size(A,1), and is assigned to a variable called rows. Similarly, the number of pixels in the image’s y axis is found using size(A,2), and is assigned to another variable called cols. Since the image may be of any size, the number of pixels that will be left over after performing the halftone transformation must be considered. The remaining values in the x- and y-axes are found using the rem() function, and stored into variables r\_remain and c\_remain, respectively. These dimensions will be useful later when the function performs the halftoning calculations.

Text

Description automatically generated

Figure 3: The start of the halftone function and image data collection

Next, the halftone function proceeds to create 10 arrays called dot9, dot8, and subsequently dot0, in this order, as shown in Figure 4. These arrays are intended to represent raster images, or bitmaps, where a value of 0 is supposed to represent a black pixel, and a 255 representing a white pixel. These arrays represent a halftone cell that will be assigned to portions of a grayscale image.

Text

Description automatically generated

Figure 4: Arrays are created to represent halftone output cells

The goal of the grayscale image being assigned these bitmap array values is that it will only have pixel values of 0 or 255. When it is then converted to a binary image of logical type, the result will have 1s and 0s where 0s represent black pixels and 1s represent white pixels. These bitmap arrays have different quantities of black pixels (0-pixel value) because they are supposed to represent raster images, as shown below in Figure 5. The ratio of the black areas to the non-black areas of the raster image corresponds to the luminance of an input cell from the grayscale image. The goal of the halftone function is to produce a binary image (1s and 0s) that appears like the original grayscale image from afar. The black pixels in the bitmap arrays are static and will not move. Additionally, the quantity of black pixels will not change.

Chart, shape, arrow, bubble chart

Description automatically generated

Figure 5: Raster images of black (0 grayscale value) and white (255 grayscale value) pixels

After the bitmap arrays are created, the function proceeds to perform the halftone transformation in four different patterns. The first pattern is going from the top of the image, moving from left to right, and moving down towards the bottom. This pattern iterates through the number of pixels in the x- and y-axis in intervals of three. Since the image is assumed to have any type of dimensions, the pattern will move from left to right until the next interval of three pixels is detected to go out of the image’s boundaries (right edge). When this boundary detection is made, or until the intervals have met the end of the image on its right edge, the pattern resets back at the left side if another interval of three pixels may continue towards the bottom of the image. If the next interval of three pixels is detected to exceed the image’s boundary in the x-axis (bottom edge), no more transformations will occur, and this pattern ends. The first pattern, labeled “UPPER-LEFT LEFT-TO-RIGHT TRANSFORM” is shown below in Figure 6.

Table

Description automatically generated with medium confidence

Figure 6: First halftone transform algorithm

The first transform pattern starts with a for loop that goes by each row in intervals of three until the end of the image on its right side. The nested for loop afterwards carries on by going by each column in intervals of three until the end of the image on its bottom side. The intervals of three going in both right and down is what gives this pattern its name “UPPER-LEFT LEFT-TO-RIGHT”. The intervals in both directions form arrays of size 3x3. These arrays are selected in the image using element-wise selection, a method in MATLAB that allows the user to select regions inside of an array. This is apparent in the assignment of variable PXL\_AVG, where row\_idx is the current row index in the first for loop, and by using the character ‘:’, the element row\_idx+2 is the last row index of the current interval. Using this technique, the algorithm will select row ranges in the image that are three pixels in size. The same technique is applied when selecting the column ranges, using col\_idx as the current index, and col\_idx+2 as the last column in the current interval. PXL\_AVG takes the sum of all pixel values in these 3x3 pixel ranges (rows and columns) and computes the average with the mean() function.

After the average pixel value has been calculated, it is compared to different possible ranges. These ranges, shown below in Figure 7, have specific values. These values were determined by taking the range of pixels that could be assigned to the variable PXL\_AVG and split up into ten different ranges. There are ten different ranges because there are ten different bitmap arrays. When the value PXL\_AVG falls into one of these ranges, the algorithm will assign the same row and column range in the two for loops to the values of the bitmap. For example, dot9 is used in the first range because it has no black pixels. The image will be assigned to the values of this bitmap array if the average pixel value is between 0 and 25, which to the average person appears mostly white on a grayscale image. Finally, the image in this row and column range will contain only zeros. After this assignment, the transformation is over and the next column or row in the image is iterated to repeat this calculation and assignment process. Since there are different pixel averages throughout the image, the best-case scenario is that all the conditional statements will be used and all bitmap arrays dot9 through dot0 will be used.

Text

Description automatically generated

Figure 7: Assigning grayscale images to bitmap array values in first halftone transform

The next pattern in the halftone function is called the “RIGHT EDGE TRANSFORM”, which performs halftone transformations along the right edge of an image if the image’s dimension along the y axis is not divisible evenly by three. In other words, if the value of c\_remain is greater than zero, it means that there is a remainder of columns after dividing the number of column pixels by three. This suggests that the “UPPER-LEFT LEFT-TO-RIGHT TRANSFORM” pattern would not perform halftone transformations along the right edge.

The algorithm for this second pattern is shown below in Figure 8. It uses the same technique of going through rows and columns without stepping over bounds.

Graphical user interface, text

Description automatically generated with medium confidence

Figure 8: Second halftone transform algorithm

At the start of the second halftone transform algorithm, the value of col\_idx is equal to the start of the remaining range that needs to be iterated through. In other words, since the first algorithm covered a set of indices of multiples of three, this second algorithm will cover the remainder of columns on the y-axis. To do so, the value of col\_idx is set equal to the maximum number of columns (cols) minus the remainder of columns after performing the remainder function (rem()) in the beginning of the halftone function (c\_remain), plus one – to account for indices starting at ‘1’ in MATLAB, not ‘0’. In Figure 9 below, the methodology for assigning bitmap arrays to ranges in the grayscale image is shown for every average range of pixels.

Text

Description automatically generated

Figure 9: Assigning grayscale images to bitmap array values in second halftone transform

In the third halftone transform algorithm, named “BOTTOM EDGE TRANSFORM”, the bottom edge of the image in the x-axis is considered for any possible remaining pixels that were not covered in the first and second transformation. To accommodate these pixels, the algorithm shown below in Figure 10 utilizes a similar method as the “RIGHT EDGE TRANSFORM” algorithm such that the row and column indices are switched places. The grayscale image is permuted and assigned bitmap array values according to their average pixel values, as shown in Figure 11.

Graphical user interface, text, application, table

Description automatically generated

Figure 10: Third halftone transform algorithm

Table

Description automatically generated

Figure 11: Assigning grayscale images to bitmap array values in third halftone transform

The fourth and final algorithm that handles the grayscale to halftone process is called the “BOTTOM-RIGHT CORNER TRANSFORM” pattern. The algorithm functions to accommodate the pixels that were missed by all the previous transformation methods by taking the upper limits of remaining pixel ranges in the x- and y-axis. Like the second and third halftone transform algorithm, the row and column indices of the grayscale image are the upper limits in the “BOTTOM-RIGHT CORNER” algorithm, as shown in Figure 12. These ranges are used to identify the last pixels in the very bottom-right corner of the image, forming a very small rectangle or square – assuming there are remainders in the x- or y-axis. The bitmap arrays are set to ranges in the grayscale image respective to the average pixel values, as shown in Figure 13.

Text

Description automatically generated

Figure 12: Fourth halftone transform algorithm

Table

Description automatically generated with medium confidence

Figure 13: Assigning grayscale images to bitmap array values in fourth halftone transform

# Results and Discussion

## Grayscale Images and Halftone Transformations

The three images are originally grayscale images. As discussed before, they are used as arguments for the halftone function in the script. The input type is uint8 and the output of the halftone function is uint8 type as well. However, the values in the output image have only one of two values, ‘0’ and ‘255’, to represent a halftone transformation.

In the main function, the original image is presented to the user and the halftone image is presented after the function takes the original image and processes it. This process is done for all three images and is shown below in Figure 14.

Graphical user interface, text, application

Description automatically generated

Figure 14: Using original images to call halftone function and displaying processed images

## The Halftone Images

The first, second, and third images before their halftone transformation is shown below in Figure 15, 17, and 19, respectively, and their halftone versions are labeled 16, 18, and 20, respectively.

A picture containing text, person

Description automatically generated

Figure 15: Original face image

Text

Description automatically generated

Figure 16: Halftone face image

A person using a camera

Description automatically generated with low confidence

Figure 17: Original cameraman image

Diagram

Description automatically generated with medium confidence

Figure 18: Halftone cameraman image

A group of people

Description automatically generated with low confidence

Figure 19: Original crowd image

Qr code

Description automatically generated

Figure 20: Halftone crowd image

Upon closer review of each image, it is evident that the halftone transformation was successful.

## Dynamic Image Sizes and Edge Correction

This script works for images of varying dimensions, including heights and widths that are now evenly divisible by groups of three. The halftone function accommodates for this potential scenario and performs halftone transformation along edges of the input image.

In Figure 21, a zoomed-up region of the cameraman image in its lower-right corner shows that the halftone transformation, and that its right and bottom edges have been accounted for. Its lower-right edge was also considered, and the halftone image shows a successful transformation.

A picture containing graphical user interface

Description automatically generated

Figure 21: Original and halftone cameraman images are compared side-by-side to show successful transformation

With the side-by-side comparison, it is evident that the halftone transformation was successful. In Figure 22, another side-by-side comparison is made in the third image.

A picture containing text

Description automatically generated

Figure 22: Original and halftone crowd images are compared side-by-side to show successful transformation

## Creating a Wedge to Prove Halftone Function Effectiveness

Figure 23 contains a grayscale image of size 256x256 and contains a gradient going from top to bottom from dark to light pixels. The value of the pixels at the top of the gradient image is 0, and through each row, the value of each pixel increments by 1. At the end of the wedge (bottom edge), the pixel values are 255, the maximum value a pixel can be in a grayscale image. The script to create a grayscale image and assigning pixel values to each row in the image is shown in Figure 24.

Shape

Description automatically generated

Figure 23: Grayscale gradient image of 256x256 pixels

Text, letter

Description automatically generated

Figure 24: Script creates wedge image in grayscale

The grayscale image is then sent through the halftone process by using its variable inputWedge to call the halftone function (Figure 25) and the image is prepared. The resulting image can be seen in Figure 26.

Text

Description automatically generated

Figure 25: Calling the halftone function using the generated grayscale image

A picture containing chart

Description automatically generated

Figure 26: Grayscale gradient image of 256x256 pixels after halftone rendering

Using the halftone transformed image, it is possible to confirm that the transformation was successful. Upon closer inspection (Figure 27), it is evident that the transformation was successful at rendering the edges (right, bottom, and lower-right corner) – considering that 256x256 is not evenly divisible by three.

A picture containing background pattern

Description automatically generated

Figure 27: Close up of the bottom-right corner of the wedge image

In conclusion, the halftone transformation worked successfully, and the wedge image was generated and transformed appropriately.

# Conclusion

After reviewing the previous images produced by the halftone function, it is apparent that the transformations and rendering processes were successful. Looking at the results, the halftone images look very similar to the original grayscale images. The bitmap arrays that were used to assign grayscale pixel regions to black or white proved to be usable and viable options for this rendering technique. Additionally, the function is usable by different image sizes, which suggests that any image could be inserted into this function and a halftone render could be produced. Since the algorithms used in the halftone function accommodate for edges, the program should not fail, and it operates efficiently. By using a linear interpolation method and dividing the 255 into ten different ranges, it is possible to use all bitmap arrays to interpolate different halftone regions in the input image to produce a concise output image. All the images were successfully transformed, and all images kept their same dimensions, therefore the function works as expected without modifying files or doing any pixel count manipulation.

Appendix A

The following information includes sources used to generate scripts and build functions in halftone\_generator.m. For more information of a specific topic, refer to the link provided.

[1] Creating greyscale image

MathWorks, (2022). “How to create a gray scale image”. Retrieved Dec 2022 from <https://www.mathworks.com/matlabcentral/answers/108006-how-to-create-a-gray-scale-image>

[2] Converting image to type uint8

MathWorks, (2022). im2uint8 (R2021b). Retrieved Dec 2022 from <https://www.mathworks.com/help/images/ref/im2uint8.html>

[3] Remainder

MathWorks (2022). rem (R2021b). Retrieved Dec 2022 from <https://www.mathworks.com/help/matlab/ref/rem.html>

[4] “Ones” matrix generation

MathWorks, (2022). ones (R2021b). Retrieved Dec 2022 from <https://www.mathworks.com/help/matlab/ref/ones.html>

[5] Accessing specific column range in an array

MathWorks (2022). “Specific column range from an array”. Retrieved Dec 2022 from <https://www.mathworks.com/matlabcentral/answers/333727-specific-column-range-from-an-array>

[6] Determining sum of elements in a matrix

<https://www.mathworks.com/help/matlab/ref/sum.html>

[7] Using fprintf in MATLAB

<https://www.mathworks.com/help/matlab/ref/fprintf.html>

[8] Rounding values in MATLAB

<https://www.mathworks.com/help/matlab/ref/round.html>

[9] Accessing the size of an array in MATLAB

<https://www.mathworks.com/help/matlab/ref/size.html>

[10] Taking a mean of all elements in an array

<https://www.mathworks.com/help/matlab/ref/mean.html>

[11] Calculating remainder

<https://www.mathworks.com/help/matlab/ref/rem.html>

[12] Binarize an image from grayscale

<https://www.mathworks.com/help/images/ref/imbinarize.html>

[13] Using progress bar (waitbar)

<https://www.mathworks.com/help/matlab/ref/waitbar.html>

[14] Special characters in waitbar

<https://www.mathworks.com/matlabcentral/answers/376325-how-to-write-percentage-sign-into-a-txt-file>

[15] Working with subplots

<https://www.mathworks.com/help/images/display-multiple-images.html>