# 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.

NOTES FROM TA:

include how I handled edges and how each 3by3 interval is accommodated

how edges are completed

# 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 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 Transform Pattern

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

Graphical user interface, text

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

Text, table

Description automatically generated