ME 5405 Machine Vision Project Report

Submitted by -

Khakimjon Saidov (A?????)

Utkarsh Anand (A0147589J)

Introduction

In the last 4 decades, machine vision has had an important role in industry, security, medicine, and academia. Humans vision is very diverse, for example we can distinguish between different objects and can isolate the background from the object of interest very efficiently. But what humans vision is not very good at is performing the same task repetitively as we tend to make more mistakes and lose neutrality as we do the same thing over and over again. Inspection of manufactured goods for quality control, surveillance of a traffic junction for road safety, and detection of abnormalities in a MRI scan are few of the examples where machine vision is extremely useful. While researchers have extensively been working to develop better and faster image processing algorithms to do more complex and demanding jobs, the rapid growth by the semi-conductor industry to make cheaper and faster computers has completely revolutionized machine vision. Nowadays, a lot of processes can be done fast because of the quick feedback provided by the state-of-the-art image processing algorithms running on state-of-the-art computers. Through this module we learn the basic processes involved in image processing and we apply some of this knowledge to process two different types of images.

Overview

This report comprises of two parts that describe our strategy to perform the required operations on the two different images. The image processing codes were first implemented in Octave (because of its lower memory requirement), and after all the algorithms worked as expected they were tested for compatibility with MATLAB. We tried to implement most of the functions on our own, and as a result they are not optimized and take longer to run. The codes are also available on www.github.com (link).

For each image we are required to do the following operations -

1. Display the original image.
2. Create binary image using thresholding.
3. Separate and identify different characters.
4. Rotate the characters clockwise by 90 degrees.
5. Rotate the characters anti-clockwise 35 degrees.
6. Find the boundary of all the characters.
7. Find the one-pixel thin image of all the characters.
8. Rearrange the characters in a particular sequence.

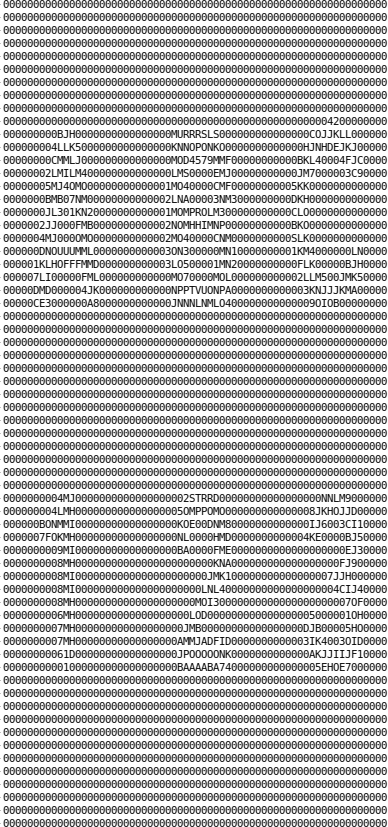
For each of the operations, we discuss a few different methods, to the best of our knowledge, that can be used to obtain the desired result with their respective advantages and disadvantages.

Processing of Image 1 - charact1.txt

Part 1 - Display the original image

The original image is in the form of a text file ‘charact1.txt’ (Figure 1). We use the inbuilt function ‘fileread()’ to read the contents of the input file as a vector of characters. This vector is then passed as an argument to a custom defined function ‘textToascii()’. The textToascii function creates a new empty vector, and then goes through each of the characters of the input vector one-by-one. Then the character is converted to its corresponding ASCII value using the inbuilt function ‘toascii()’ which converts the character to an integer in the range 0-255. If the ASCII character does not have a value of 10 (corresponds to new line) or 13 (corresponds to carriage return), it is valid and is appended to the empty vector. At the end of this operation we get a new vector of numbers of length 4096 elements.

It is important to note that this vector of characters can be converted into numbers by building a custom look-up table which links each character to a unique intensity value. For example, we could assign all the ‘0’ characters to have intensity value 0, all the ‘A’ characters to have intensity value 40, all the ‘R’ characters to have intensity value 100, and so on. But we chose to use the ASCII value as a look-up table because the inbuilt function ‘toascii()’ is easy and convenient to use and it maps all the characters in the image to a unique intensity value.



Next, the problem statement says that the image is 64 pixels x 64 pixels, and the vector of numbers is reshaped accordingly. The first 64 elements of the vector form the first row of the image, elements 65 to 128 form the second row of the image, elements 129 to 192 form the third row of the image and so on. This generate a grayscale image, with 32 unique intensity values with minimum and maximum intensity values of 48 (corresponds to character 0) and 86 (corresponds to character V) respectively (Figure 2). We observe that the background is darker than the foreground which comprises of the letters A, B, C and numbers 1, 2, 3.



Part 2 - Create binary image using thresholding

If we observe the greyscale intensity values of the image we see that all the background pixels correspond to intensity value of 48, and the foreground pixels value are higher. Thus we can use this information to our advantage and threshold the image using different strategies which are discussed below.

1. Constant threshold - We can set up a global/constant threshold as 48. Then we scan through all the pixels of the image and if the pixel value is less than or equal to 48 it is set as 0 (background) but if it greater than 48 it is set as 1 (foreground/object). This process will be very quick as we have to scan through all the pixels just once and we will get a binary image after this operation. However, the problem with this step could be its generality. For example, if we decide to enhance the contrast (appendix section 1) of the image by histogram stretching or histogram equalization the same global threshold value might not work and it will have to be changed. This is implemented in our ‘threshold()’ which takes in the image, threshold method, and threshold value as parameters. The function returns a thresholded binary image and the set threshold value. To use the function in constant threshold mode, type threshold( img, method = ”constant”, th = 48 ).
2. Mean threshold - We can calculate the average intensity value from all the pixels and set it as the threshold. For the image in Figure ?? generated using the ASCII character mapping, the average intensity value is 50.660. All the pixels less than or equal to 50.660 are set as 0 and those greater than 50.660 are set as 1. This method is slower because we need to scan through all the pixels twice. In the first run, we calculate the mean intensity value and in the second run we apply it as a threshold. Even though the mean threshold method is slower it is more robust to some point processing operations like histogram stretching and equalization. As long as the background pixels remain darker then the foreground pixels, the mean threshold will work reasonably well for similar images. To use the function in mean threshold mode, type threshold( img, method = ”mean” ).
3. Median threshold - Just like the mean, median is also a number which describes the central tendency of the intensity values of the image. The median intensity value for the image in Figure 1 is 48 and the binary image is generated accordingly. To use the function in mean threshold mode, type threshold( img, method = ”median” ).
4. Otsu and maximum entropy (Kapur) threshold - These are more modern thresholding techniques which work best on a bi-modal distribution (Figure ??). The complete details of these methods are out of the scope of this report but they work on the principle of minimizing the inter-class-variance (Otsu) or maximizing the entropy (Kapur) between the two distributions which make up the foreground and the background. For the image in Figure ??, the calculated Otsu and Kapur threshold values are 57 and 48 respectively. To use the function in Otsu threshold mode, type threshold( img, method = ”otsu” ). To use the function in Kapur threshold mode, type threshold( img, method = ”maxentropy” ).

It should be noted that higher the threshold value, smaller are the foreground objects. For example, a threshold value of 48 will give the larger objects compared to if we use threshold value of 50.660. In Figure ?? we present the binary image generated using the mean threshold method.



Part 3 - Separate and identify different characters.

At the end of step 2, we get a binary image where the pixels containing the letters and alphabets are marked as 1 and the background is marked as 0. Then we do a connected component analysis where we scan through the image and group the pixels into components based on the pixel connectivity. For example, in Figure ?? the all the pixels representing alphabet ‘B’ have intensity 1. We start from one such pixel and start labeling all the 1 value pixels which are directly 8-connected to that pixel and give it the same label. This analysis is repeated for the whole image until all the pixels are assigned a label. After this operation, the generated labeled image has 7 unique labels. 0 corresponds to the background, and labels 1,2,3,4,5,6 correspond to ‘A’, ‘1’, ‘B’, ‘2’, ‘C’, ‘3’ respectively. We use the MATLAB inbuilt function ‘bwlabel()’ to perform this labeling using 8-connected neighbors.

We do a histogram stretching on the labeled image to change the range of intensity values from 0-7 to 0-255 to make it visually appealing and present it in Figure 4A. We also use another image processing software Fiji (cite) to represent the labels using different colors (Figure 4B).

From the labeled image, we can extract more features like the centre, bounding box, area etc. for each of the labels. A bounding box is the smallest rectangle, oriented along the rows and columns of the image, which can fully contain the desired region of interest (ROI). In our case, the ROI is each label of the foreground object that we get from the labeled image. Once we get the bounding box, we treat its centre as the centre of the object, the width of the box corresponds to the width of the object and the height of the box corresponds to the height of the object. The total number of pixels that have the same label is defined as the area of that object. For example, in the labeled image (Figure 4??) alphabet ‘A’ has a label 1 and there are 84 pixels with the same label. So the area of alphabet ‘A’ is 84. Similarly number ‘2’ has label 4 and number of pixels with label 4 are 77. Hence, the area of number ‘2’ is 77.

Using the information from bounding box, we can crop smaller sections from the original grayscale image and plot the different characters separately (Figure ??). Here we arrange all the characters in the sequence of their labels mentioned earlier.

To find the properties of labeled region, we can use the following two functions.

[ labelImg , numLabel ] = bwlabel ( bImg, 8 ) ;

props = regionProps ( labelImg ) ;

The first function ‘bwlabel()’ takes in the binary image, and connectivity as input. Here we use 8-connected neighbors to label different objects in binary image bImg. The function return the labeled matrix labelImg and also the number of objects ‘numLabel’ that were found in the binary image. Then we use the custom written regionProps() which takes in the labeled matrix as input and finds out the bounding box, area, centre, width, and height of each of the connected components.

Part 4 - Rotate the characters clockwise by 90 degrees

Rotation of each of the characters clockwise can be done in two ways.

1. We can think of rotation a matrix by 90 degrees clockwise as doing a transformation which takes the first row of a matrix to the last column, second row to the second last column, third row to the third last column and so on. If the input matrix has dimensions MxN then the final transformed matrix has dimensions NxM. Even though this can be very fast and easy to implement, we did not use it because it can be used only for this specific purpose.
2. In general, a rotation matrix is defined as

Here, is the angle of rotation in the counter clockwise direction. So, if we have to rotate an object by 90 degrees clockwise, it would be equivalent to rotating the object by -90 degrees counter clockwise. Hence, we can use the above transformation by setting = -90 degrees. We choose to use this approach because it can be used very easily for the next part of the problem where we are required to rotate the image by 35 degrees counter clockwise.

We write our custom function ‘imageRotate()’ which takes in three parameters. First one is the input image, second is the angle in degrees by which we want to rotate the image, and the third one is the interpolation method to be used after rotation. We implemented the nearest neighbor and the bi-linear interpolation methods. Nevertheless, for rotation by 90 degrees using any interpolation method produces same result as all the pixel position, after transforming through the rotation matrix get mapped to integer pixel values and hence bi-linear interpolation becomes redundant. For this specific problem, we use the imageRotate ( cropImg, -90, ‘nearestNeighbor’ ).



Part 5 - Rotate the characters anti-clockwise 35 degrees

We use the same ‘imageRotate()’ with input rotation angle as 35 degrees and generate two images. Figure ?? compares the nearest neighbor interpolation with the bi-linear interpolation. It should be noted that the bi-linear interpolation is faster than the bi-linear interpolation method. We also tried implementing bi-cubic interpolation method discussed in the lectures but our implementation was not correct.

