ME5405 Machine Vision

AY22/23 Semester 1

Computing Project

Group 31:

Brina Shong Wey Tynn (A0170862A)

Chung Jun Yu Chester (A0201459J)

Zhu Rundong (A0263429Y)

Table of Contents

1. Image 1: Chromosomes

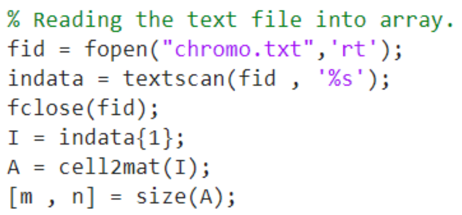
1.1 Image Display

Introduction to the problem

As indicated by the problem statement, image 1 is an image represented by 32 different gray levels and its size is 64 pixels by 64 pixels. This image is a coded array that consists of alphanumeric characters in place of each pixel gray-level, and these characters ranging from 0-9 and A-V correspond to the 32 levels of gray. Therefore, in order to derive and display the original image, it is necessary to decode the image by mapping the alphanumeric characters to their corresponding gray levels.

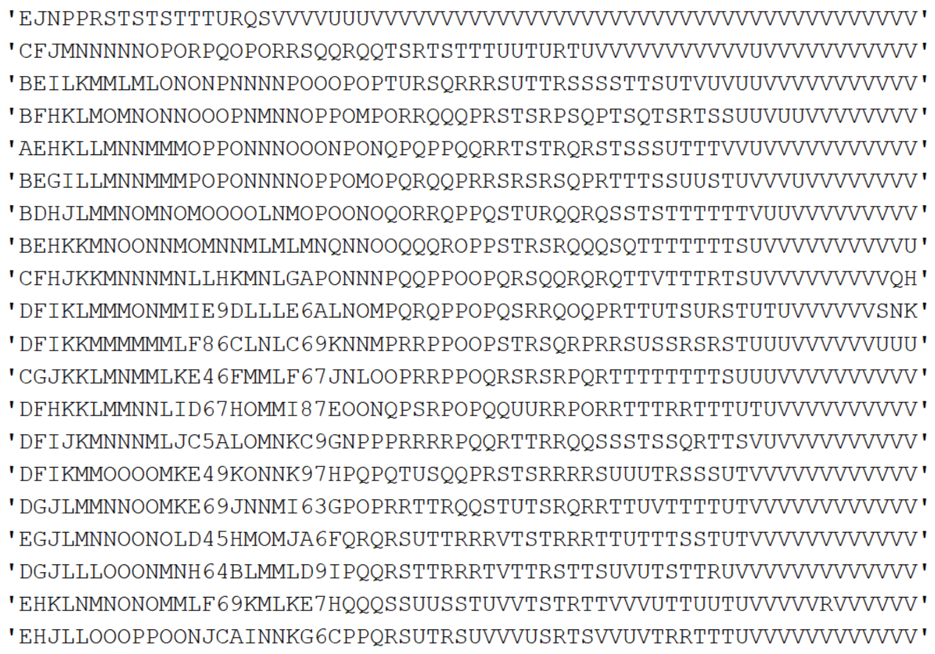
Description of the algorithm and flowchart

Firstly, to display the original image, we need to read the provided text file (in .txt format) into a cell array, and subsequently convert it into an ordinary array/matrix. The corresponding MATLAB code is shown in Figure 1 below. The ordinary array (matrix A), which has size of 64-by-64 characters, is shown in Figure 2.



```
% Reading the text file into array.
fid = fopen("chromo.txt",'rt');
indata = textscan(fid , '%s');
fclose(fid);
I = indata{1};
A = cell2mat(I);
[m , n] = size(A);
```

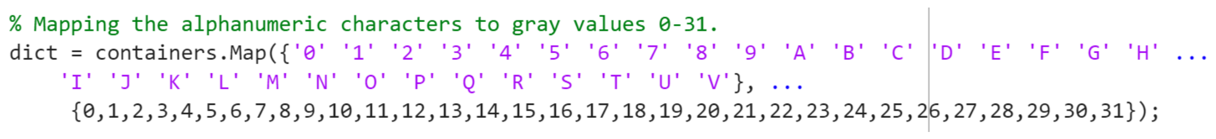
Figure 1. MATLAB code for reading the text file into an ordinary array



```
'EJNPPRSTSTSTTTURQSVVVVUUUVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV'
'CFJMNNNNNOPORPQOPORRSQQRQQTSRTSTTTUUTURTUVVVVVVVVVVVVUVVVVVVVVVV'
'BEILKMMLMLONONPNNNNPOOOPOPTURSQRRRSUTTRSSSSTTSUTVUVUUVVVVVVVVVVV'
'BFHKLMOMNONNOOOPNMNNOPPOMPORRQQQPRSTSRPSQPTSQTSRTSSUUVUUVVVVVVVV'
'AEHKLLMNNMMMOPPONNNOOONPONQPQPPQQRRTSTRQRSTSSSUTTTVVUVVVVVVVVVVV'
'BEGILLMNNMMMPOPONNNNOPPOMOPQRQQPRRSRSRSQPRTTTSSUUSTUVVVUVVVVVVVV'
'BDHJLMMNOMNOMOOOOLNMOPOONOQORRQPPQSTURQQRQSSTSTTTTTTVUUVVVVVVVVV'
'BEHKKMNOONNMOMNNMLMLMNQNNOOQQQROPPSTRSRQQQSQTTTTTTTSUVVVVVVVVVVU'
'CFHJKKMNNNMNLLHKMNLGAPONNNPQQPPOOPQRSQQRQRQTTVTTTRTSUVVVVVVVVVQH'
'DFIKLMMMONMMIE9DLLLE6ALNOMPQRQPPOPQSRRQOQPRTTUTSURSTUTUVVVVVVSNK'
'DFIKKMMMMMLF86CLNLC69KNNMPRRPPOOPSTRSQRPRRSUSSRSRSTUUUVVVVVVUUU'
'CGJKKLMNMMLKE46FMMLF67JNLOOPRRPPOQRSRSRPQRTTTTTTTTSUUUVVVVVVVVVV'
'DFHKKLMMNNLID67HOMMI87EOONQPSRPOPQQUURRPORRTTTRRTTTUTUVVVVVVVVVV'
'DFIJKMNNNMLJC5ALOMNKC9GNPPPRRRRPQQRTTRRQQSSSTSSQRTTSVUVVVVVVVVVV'
'DFIKMMOOOOMKE49KONNK97HPQPQTUSQQPRSTSRRRRSUUUTRSSSUTVVVVVVVVVVVV'
'DGJLMMNNOOMKE69JNNMI63GPOPRRTTRQQSTUTSRQRRTTUVTTTTUTVVVVVVVVVVVV'
'EGJLMNNOONOLD45HMOMJA6FQRQRSUTTRRRVTSTRRRTTUTTTSSTUTVVVVVVVVVVVV'
'DGJLLLOOONMNH64BLMMLD9IPQQRSTTRRRTVTTRSTTSUVUTSTTRUVVVVVVVVVVVVV'
'EHKLNMNONOMMLF69KMLKE7HQQQSSUUSSTUVVTSTRTTVVVUTTUUTUVVVVVRVVVVVV'
'EHJLLOOOPPOONJCAINNKG6CPPQRSUTRSUVVVUSRTSVVUVTRRTTTUVVVVVVVVVVVV'
```

Figure 2. 64-by-64 matrix A

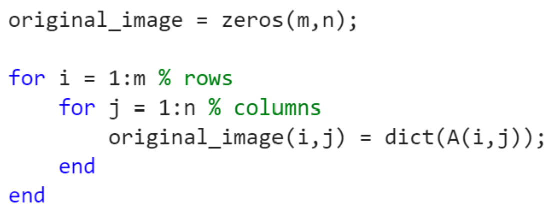
Secondly, to conduct the mapping of alphanumeric characters in the matrix A into their corresponding gray levels, we need to create a map container which enables us to retrieve values (gray levels) using a corresponding key (alphanumeric characters). Since there are 32 gray levels, we need to map the characters ranging from 0-9 and A-V to gray levels ranging from 0-31. The creation of the map container is shown in Figure 3 below.



```
% Mapping the alphanumeric characters to gray values 0-31.
dict = containers.Map({'0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' ...
    'I' 'J' 'K' 'L' 'M' 'N' 'O' 'P' 'Q' 'R' 'S' 'T' 'U' 'V'}, ...
     {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31});
```

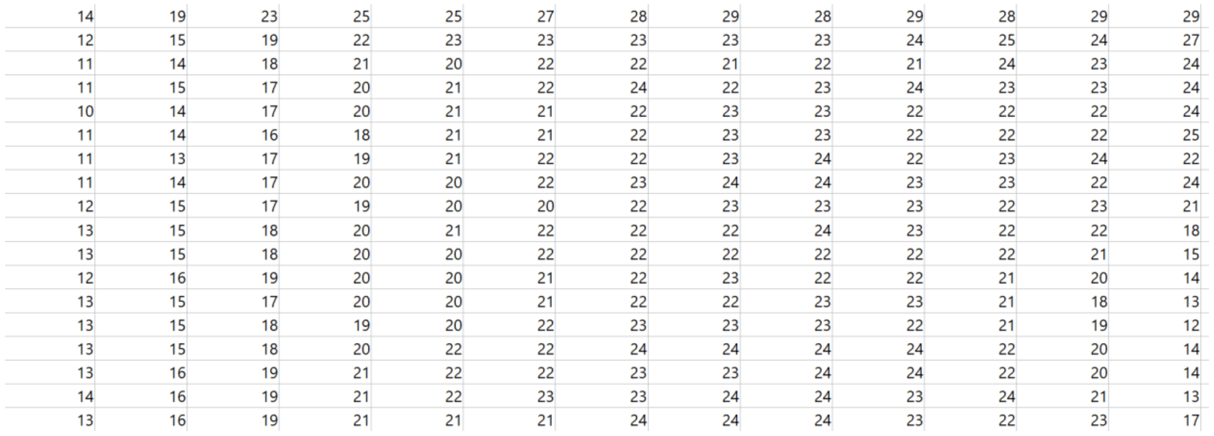
Figure 3. Map container for mapping alphanumeric characters to gray levels

Thirdly, we need to complete the mapping of the alphanumeric characters to their corresponding gray levels. Therefore, we initialized a new array of zeros with size 64-by-64 and the program ran through each alphanumeric element in the matrix A, mapped the element into its corresponding gray value using the map container, and subsequently replaced the zeros in the new array with the gray value obtained. This mapping process is shown in Figure 4 below. The resultant array, which is a 64-by-64 array of gray level integers, is shown in Figure 5.



```
original_image = zeros(m,n);

for i = 1:m % rows
    for j = 1:n % columns
        original_image(i,j) = dict(A(i,j));
    end
end
```

Figure 4. Mapping of alphanumeric elements into gray levels into a newly initialised array



14	19	23	25	25	27	28	29	28	29	28	29	29
12	15	19	22	23	23	23	23	23	24	25	24	27
11	14	18	21	20	22	22	21	22	21	24	23	24
11	15	17	20	21	22	24	22	23	24	23	23	24
10	14	17	20	21	21	22	23	23	22	22	22	24
11	14	16	18	21	21	22	23	23	22	22	22	25
11	13	17	19	21	22	22	23	24	22	23	24	22
11	14	17	20	20	22	23	24	24	23	23	22	24
12	15	17	19	20	20	22	23	23	23	22	23	21
13	15	18	20	21	22	22	22	24	23	22	22	18
13	15	18	20	20	22	22	22	22	22	22	21	15
12	16	19	20	20	21	22	23	22	22	21	20	14
13	15	17	20	20	21	22	22	23	23	21	18	13
13	15	18	19	20	22	23	23	23	22	21	19	12
13	15	18	20	22	22	24	24	24	24	22	20	14
13	16	19	21	22	22	23	23	24	24	22	20	14
14	16	19	21	22	23	23	24	24	23	24	21	13
13	16	19	21	21	21	24	24	24	23	22	23	17

Figure 5. 64-by-64 array of gray level integers

Lastly, to display the original image, we use the MATLAB function *‘imshow’*, while indicating the gray level range of the integers, which is from 0-31. This step is crucial as the *‘imshow’* function will display the gray level range from 0-255 by default. Hence, indicating the gray level range (0-31) ensures that our image is displayed such that pixels with gray level of 0 are the darkest/blackest pixels while pixels with gray level of 31 are the brightest/whitest pixels. The image display process is shown in Figure 6 below. The original image is shown in Figure 7.

```
% Display original image. (Task 1)
figure(1);
subplot(1,2,1);
imshow(original_image, [0 31]);
title("Original Image");
```

Figure 6. Image display

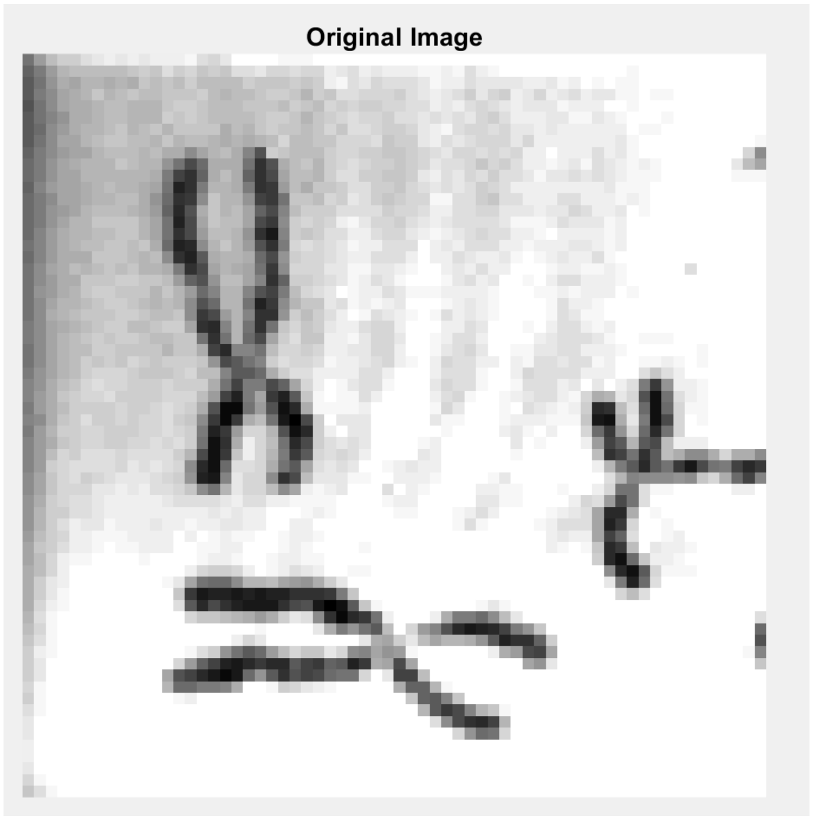


Figure 7. Original image

To summarise the main steps in the algorithm used, a flow chart is presented in Figure 8 below.

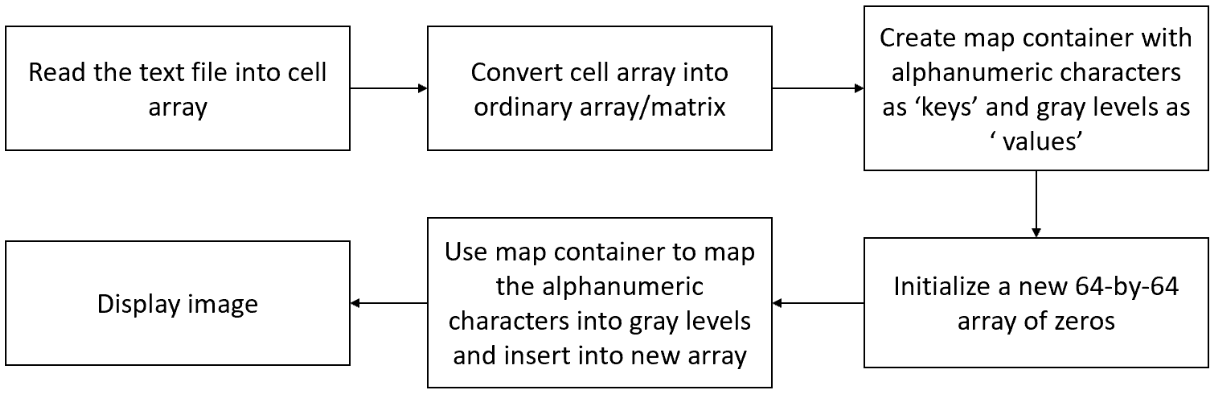

Read the text file into cell array
Convert cell array into ordinary array/matrix
Create map container with alphanumeric characters as 'keys' and gray levels as ' values'
Initialize a new 64-by-64 array of zeros
Use map container to map the alphanumeric characters into gray levels and insert into new array
Display image

Figure 8. Flowchart to summarise the main steps in the algorithm (Task 1)

Discussion & Conclusion

a. Explanation for chosen method

For this task of displaying the original image, there may be subtly different methods, but the main idea remains, which is that it is necessary to map the alphanumeric characters from the given text file into their corresponding gray values. Our employed method of accomplishing this task is systematic and logical, and it offers flexibility for future changes. For instance, if there are any future tasks which require mapping of the alphanumeric characters ranging from 0-9 and A-V into a different set of gray values (e.g., 223-255), our code can be easily adjusted to accomplish those tasks.

b. Investigation performed

In the process of completing this task, we performed two main sets of investigation. The first investigation was focused on the process of reading the formatted data from the provided text file. Initially, we assumed that merely using the function *'textscan'* was sufficient to read the formatted data successfully. However, there was still some additional input required – conversion specifier. As seen in Figure 1, we input the conversion specifier *'%s'* into the *'textscan'* function, to enable the data in the text file to be read as a cell array of character vectors. Without the conversion specifier, the program would encounter an error due to lack of input arguments.

The second investigation was centred on displaying the original image correctly. As mentioned earlier, it is necessary to indicate the gray level range (0-31) to make sure that MATLAB normalises the default gray level range (0-255) to our desired range (0-31), otherwise, the image would be displayed incorrectly, as shown in Figure 9 below.

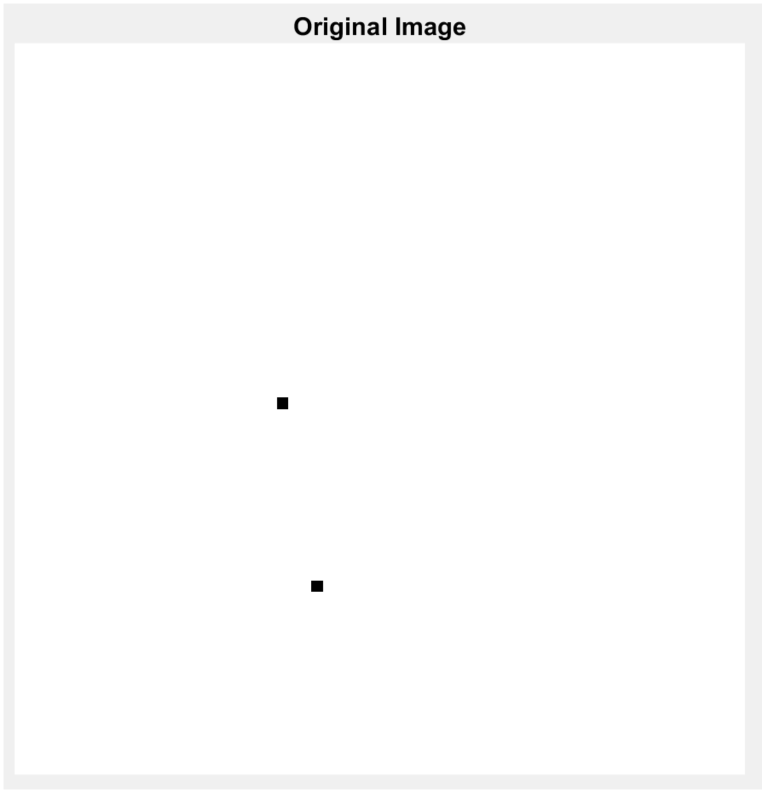


Figure 9. Incorrect image displayed without proper indication of gray level range

c. Lessons learnt

For this task, the main takeaway is to be careful and have great attention to detail. Special care must be taken especially during the process of creating the map container for mapping the alphanumeric characters to the corresponding gray values. If either the key or the values in the map container are incorrectly input, the resultant image will subsequently turn out to be incorrect. Therefore, multiple rounds of checks must be conducted to ensure that there are no errors within the code. In addition, from the project information, we are provided the information that the image should display some chromosomes. Hence, we can use this information to double-check that our original image displayed is correct to a large extent. If our image displays something completely unrelated to chromosomes, it is a sign that we need to adjust our code.

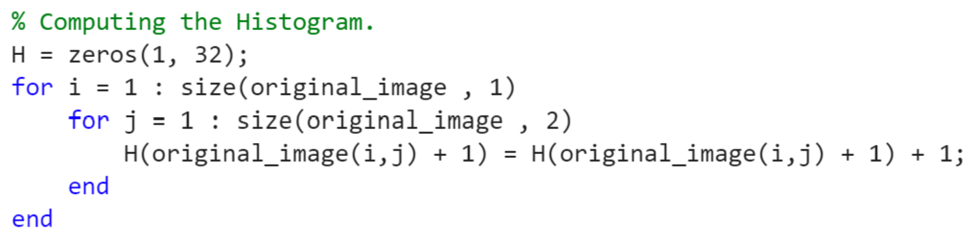
1.2 Image Thresholding and Binarization

Introduction to the problem

After the previous task, the original image derived is a grayscale image. The objective of this task is to convert the grayscale image into a binary image. The main consideration to take note is to choose an appropriate threshold level to apply to the image, to obtain a suitable binary image. To choose a good threshold level, we need to rely on two techniques – observation of the image histogram and trial and error. Pixels with gray values below or equal to the threshold level will be converted to black pixels with gray value of 0 (object pixel), while pixels with gray values above the threshold level will be converted to white pixels with gray value of 1 (background pixel).

Description of the algorithm and flowchart

Firstly, to decide an appropriate threshold level, we need to compute the histogram for the grayscale image. An image histogram displays the frequencies of pixels with the same gray values. To compute the histogram, the program initialises an array of zeros of size 1-by-32, which represents the individual counters for the 32 gray levels. The program runs through each pixel in the original image and adds to the counter corresponding to the gray value of the pixel. Figure 10 below shows the MATLAB code for computing the histogram.



```
% Computing the Histogram.
H = zeros(1, 32);
for i = 1 : size(original_image , 1)
    for j = 1 : size(original_image , 2)
        H(original_image(i,j) + 1) = H(original_image(i,j) + 1) + 1;
    end
end
```

Figure 10. Computing the image histogram

Secondly, we need to plot the histogram for visualisation. By default, the *'bar'* function displays the histogram with x-axis ranging from 1-32, which is incorrect. Therefore, minor adjustments to the x-axis are made to change the x-axis range to 0-31, which is correct as it corresponds to the 32 gray values in the original image. Figure 11 below shows the MATLAB code for displaying the histogram, with adjustments to its x-axis. The histogram is shown in Figure 12.

```
% Plotting the Histogram.
figure(2);
histogram_x_axis = 0:31;
histogram_y_axis = H;
bar(histogram_x_axis , histogram_y_axis);
title('Histogram (Original image)');
```

Figure 11. Displaying the image histogram

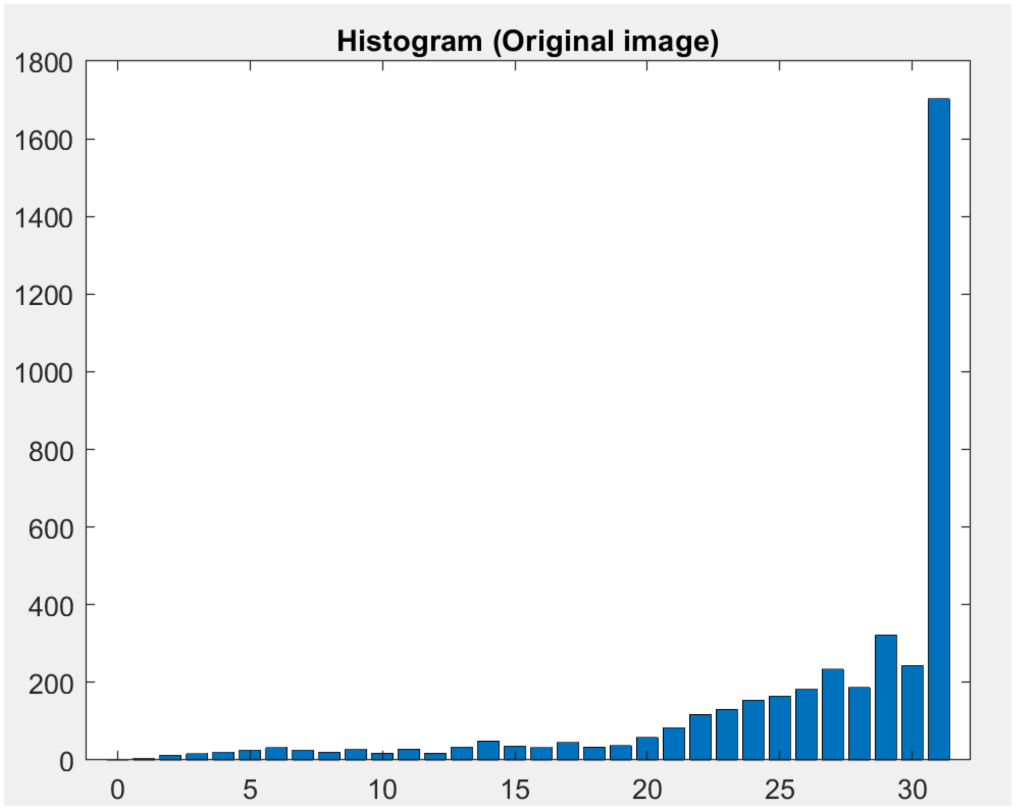


Figure 12. Image histogram

From the image histogram, we can see that the valley (lowest point) of the image histogram is in the region ranging from 16-20. Although histogram visualisation is an often-reliable technique to choose an appropriate threshold level, it is not sufficient. We need to conduct trial and error as well, to determine a good threshold level. To conduct trial-and-error, we compared the binary images for threshold values of 16, 18, 20 and 21. This trial-and-error process is further explained in the *'Investigation performed'* section under *'Discussion & Conclusion'*. Eventually, we deduced that a threshold value of 20 is the most appropriate for a good balance of detail for the objects and background.

Thirdly, we need to convert the grayscale image into a binary image. The process for this is simple, whereby pixels with gray values below or equal to the threshold level will be converted to black pixels with gray value of 0 (object pixel), while pixels with gray values above the threshold level will be converted to white pixels with gray value of 1 (background pixel). Figure 13 below shows the MATLAB code for thresholding and converting the grayscale image to binary image.

```
% Thresholding and converting to binary image. (Task 2)
min_level = min(min(original_image));
max_level = max(max(original_image));
threshold_image = zeros(m,n);
threshold = 20; % By trial & error and histogram visualization, this threshold value is the most appropriate.
for i = 1:m
    for j = 1:n
        if (original_image(i,j) > threshold)
            threshold_image(i,j) = 1;
        else
            threshold_image(i,j) = 0;
        end
    end
end
```

Figure 13. Thresholding and converting to binary image

Lastly, like the image display in task 1, we need to indicate gray level range (0-1), to ensure that our image is displayed such that pixels with gray level of 0 are the darkest/blackest pixels while pixels with gray level of 1 are the brightest/whitest pixels. Otherwise, the default *'imshow'* function will display the gray level range from 0-255 by default, which is incorrect. Figure 14 below shows the MATLAB code for the binary image display process. The resultant binary image is shown in Figure 15.

```
% Display binary image. (Task 2)
figure(1);
subplot(1,2,2);
imshow(threshold_image , [0 1]);
title("Binary image (Threshold = 20)");
```

Figure 14. Binary image display

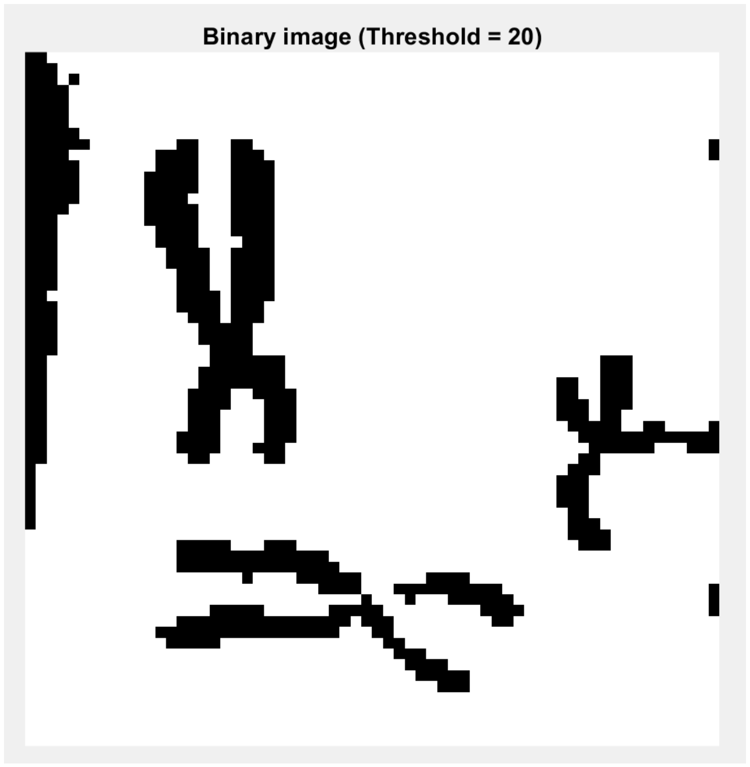


Figure 15. Resultant binary image

To summarize the main steps in the algorithm used, a flow chart is presented in Figure 16 below.

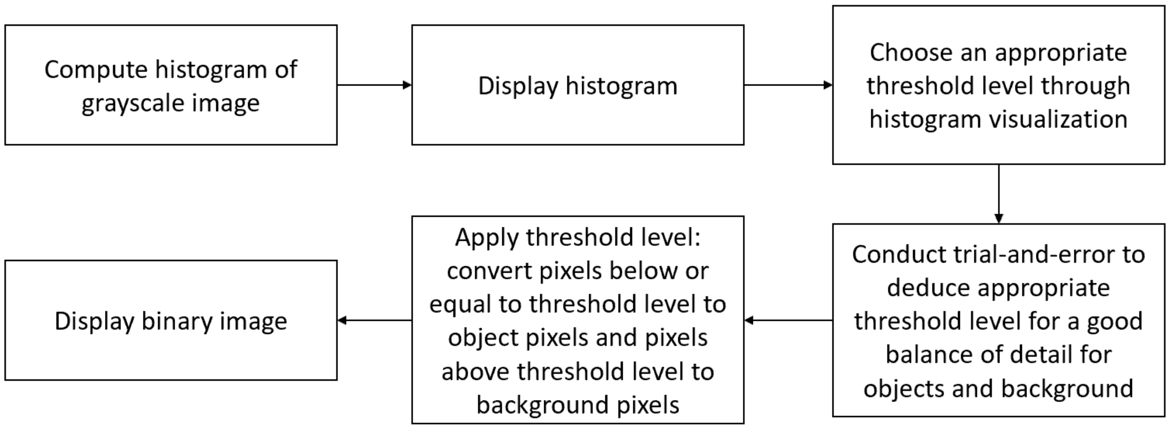

Compute histogram of grayscale image
Display histogram
Choose an appropriate threshold level through histogram visualization
Conduct trial-and-error to deduce appropriate threshold level for a good balance of detail for objects and background
Apply threshold level: convert pixels below or equal to threshold level to object pixels and pixels above threshold level to background pixels
Display binary image

Figure 16. Flowchart to summarise the main steps in the algorithm (Task 2)

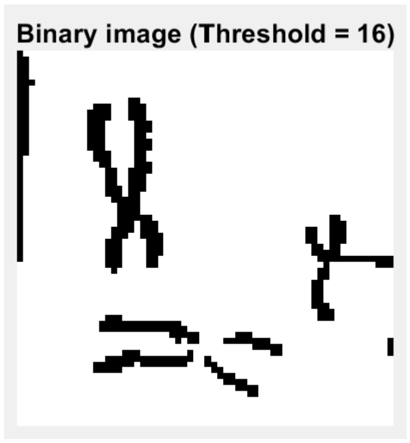
<u>Discussion & Conclusion</u>

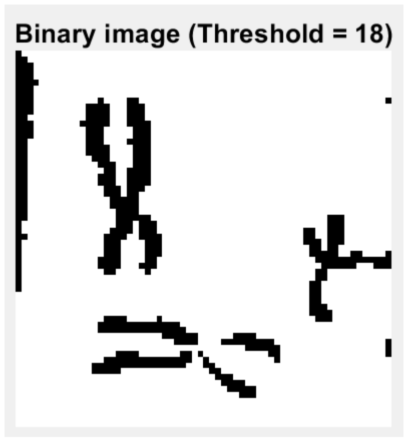
a. Explanation for chosen method

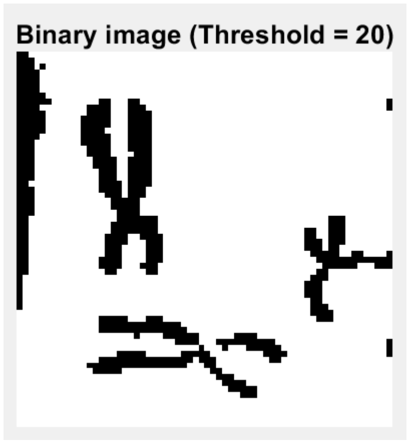
For the task of thresholding and converting the original image into a binary image, the main process should be to decide on the most appropriate threshold value for the application, and then to conduct the conversion process. Our method of completing this task is logical and flexible, as our program displays the image histogram for visualisation, which aids in deciding the most appropriate threshold level. Besides, our program can be easily modified to change the threshold level to any desired level, by simply changing the value of the variable named *'threshold'*.

b. Investigation performed

Although histogram visualisation is an often-reliable technique to choose an appropriate threshold level, it is not sufficient. We need to conduct trial and error as well, to determine a good threshold level. To conduct trial-and-error, we compared the binary images for threshold values of 16, 18, 20 and 21. These binary images are shown in Figure 17 below.







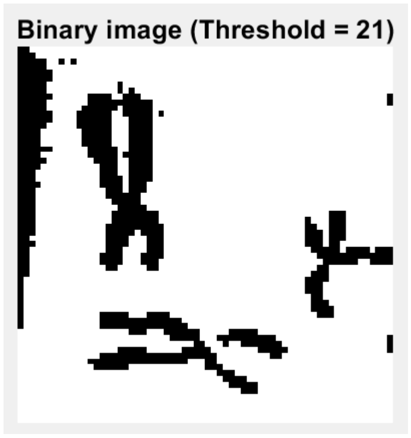


Figure 17. Comparison of binary images for threshold values of 16, 18, 20, and 21

From Figure 17, we can see that for the binary images with threshold values of 16 and 18, there is a significant lack of detail for the chromosome at the bottom of the image, such that the chromosome seems disconnected. Hence, we deduce that the threshold level of 18 is too low and therefore inappropriate. For the binary image whereby threshold value is 20, there is an improvement in the detail of the bottom chromosome, as it seems more connected. However, more of the supposed background pixels (top left corner of the image) are converted to object pixels. For the binary image whereby threshold value is 21, there is slight improvement in the detail for the bottom chromosome. However, we can start to see that there are individual pixels around the top chromosome showing up as object pixels and appearing as disconnected dots on the binary image. This is a case of the threshold value being too high. As a result, we deduce that a threshold value of 20 is the most appropriate for a good balance of detail for the objects and background.

c. Lessons learnt

The main takeaway from this task is that there is no correct or incorrect threshold level for conversion to binary image. The most appropriate threshold level is one that is the most useful depending on the application of the image. In the case of this project, we are studying the features and details of the chromosomes. Therefore, it is essential that the resultant binary image captures the details of the chromosome, as much as possible, while ensuring that the outlines of the chromosomes are not affected by dots around them (as represented in the binary image whereby threshold level was 21 in Figure 17).

1.3 Image Skeletonization (One-pixel Thin Image)

<u>Introduction to the problem</u>

Following the task of deriving the binary image from the grayscale image, this task requires the derivation of a one-pixel thin image of the objects (chromosomes). A one-pixel thin image of the objects is basically the skeleton of the objects. The thinning or skeletonization process reduces the image complexity, as it removes details from the objects but maintains its basic shape and structure. For this task, existing thinning algorithms such as the Stentiford thinning algorithm and Zhang-Suen thinning algorithm will be explored.

To better understand the objective of this task, Figure 18 below shows the desired resultant image after the application of the thinning algorithm.

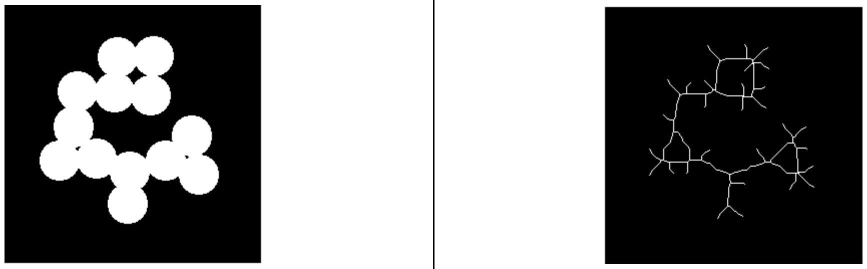


Figure 18. Original binary image and resultant one-pixel thin image

<u>Description of the algorithm and flowchart</u>

For this task, we decided to employ the Stentiford thinning algorithm, which is an iterative algorithm which checks each pixel in the binary image to decide whether they should be deleted (converted from object pixel to background pixel), based on a set of conditions.

The Stentiford thinning algorithm can be summarised into the following steps:

1. Find a pixel location (i, j) where the pixels in the image match those in template T1. Using this template T1, all the pixels along the top edge of the image are removed from left to right, and from top to bottom.
2. If the central pixel is not an endpoint, and has connectivity number of 1, we delete this pixel. A pixel is an endpoint pixel if it is connected to just one other pixel. This means that if a black pixel has only one black pixel adjacent to it, out of the eight possible neighbours in the case of 8-connectivity, then the pixel is an endpoint pixel. Connectivity number is defined as the number of transitions from black to white, in the sequence of the eight neighbours around pixel (i, j), where the sequence starts and ends at the same neighbour, making a complete circle. The mathematical definitions of connectivity number and endpoint are described by the MATLAB code shown in Figure 20.

```
i = 2;
j = 2;
N0 = inwindow(i,j);
N1 = inwindow(i,j+1);
N2 = inwindow(i-1,j+1);
N3 = inwindow(i-1,j);
N4 = inwindow(i-1,j-1);
N5 = inwindow(i,j-1);
N6 = inwindow(i+1,j-1);
N7 = inwindow(i+1,j);
N8 = inwindow(i+1,j+1);
```

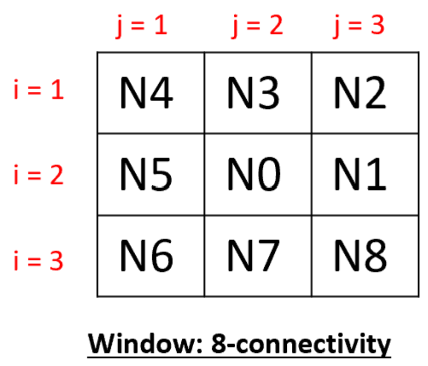


Figure 19. Neighbours in 8-connectivity

```
val = [N1 N2 N3 N4 N5 N6 N7 N8];
arr = [N1<N2 N2<N3 N3<N4 N4<N5 N5<N6 N6<N7 N7<N8 N8<N1];

Cn = sum(arr);
EndPoint = abs(N0 - sum(val));
```

Figure 20. Mathematical definitions of connectivity number (Cn) and Endpoint

3. Repeat steps 1 and 2 for all pixel locations matching the template T1.
4. Repeat steps 1 to 3 for the other templates T2, T3 and T4. Template T2 will match pixels on the left edge of the object, from bottom to top and left to right. Template T3 will select pixels along the bottom edge of the image, from right to left and bottom to top. Template T4 will select pixels on the right edge of the object, from top to bottom and right to left.
5. Set to white the pixels marked for deletion.

The four templates T1, T2, T3, and T4 used in the Stentiford thinning algorithm are shown in Figure 21 below.

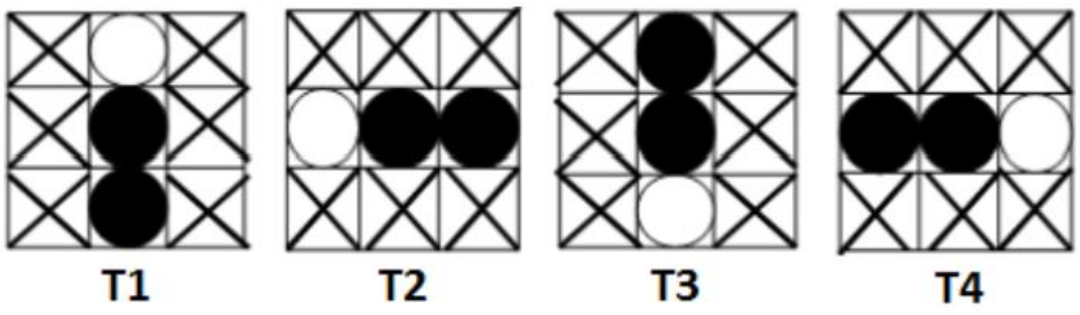


Figure 21. Templates T1, T2, T3, and T4

Because the Stentiford thinning algorithm is applied to the binary image such that the object pixels are white (gray value of 1) and the background pixels are black (gray value of 0), this configuration is opposite to our desired output image whereby object pixels are black (gray value of 0), and background pixels are white (gray value of 1). Therefore, the final step is to perform a negative transformation of the resultant image after the Stentiford thinning algorithm. This negative transformation process is shown in Figure 22 below.

```
for i = 1:row
    for j = 1:col
        if imgBin(i,j) == 0
            imgBin(i,j) = 1;
        elseif imgBin(i,j) == 1
            imgBin(i,j) = 0;
        end
    end
end
```

Figure 22. Negative transformation to derive desired output thinned image

The binary image and resultant one-pixel thin image are shown in Figure 23 below.

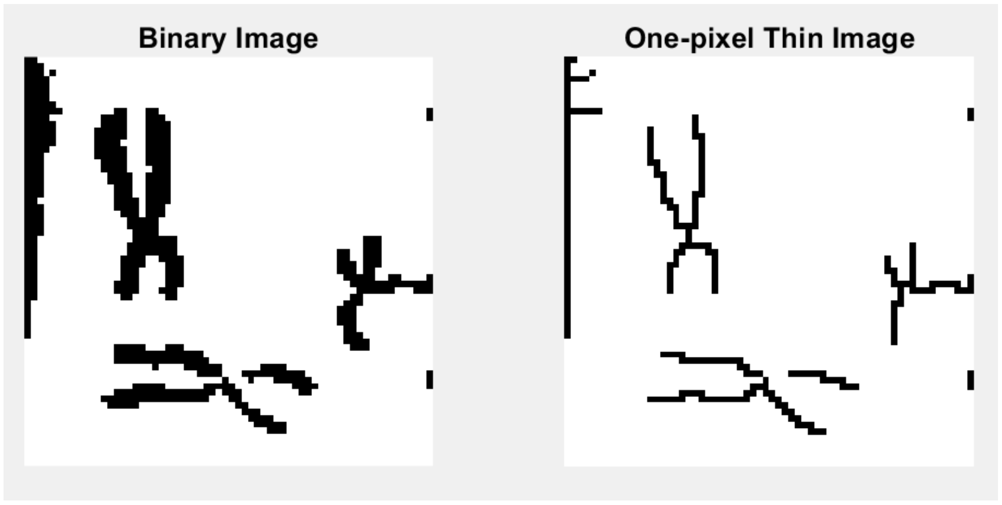


Figure 23. Binary image and one-pixel thin image

To summarise the main steps in the algorithm used, a flow chart is presented in Figure 24 below.

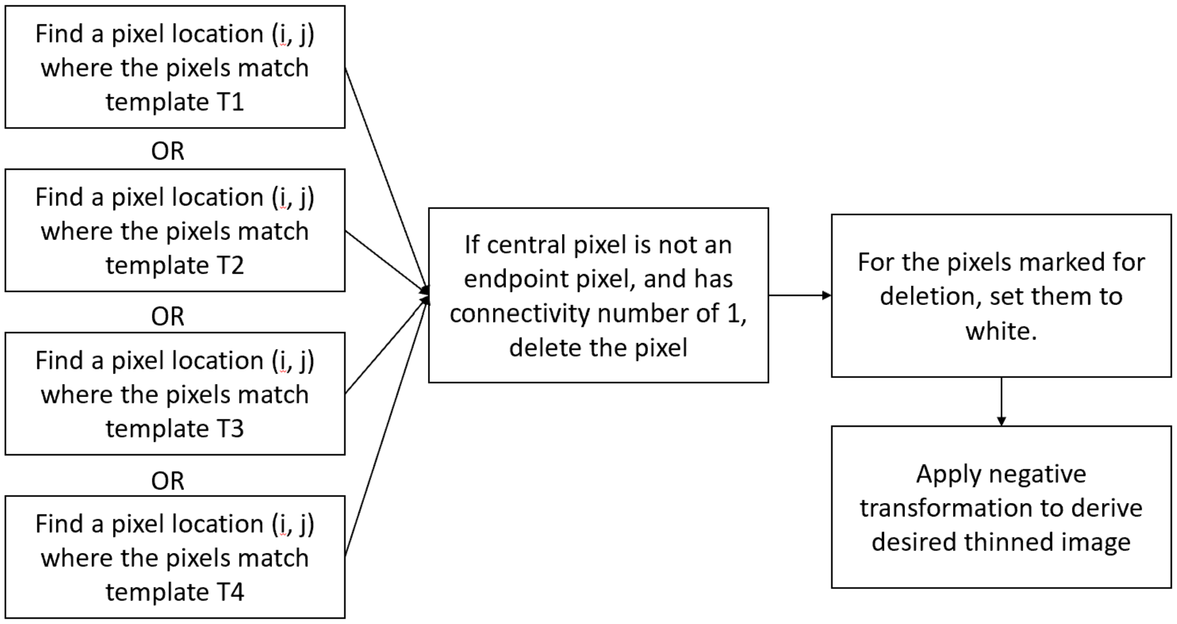

Find a pixel location (i, j) where the pixels match template T1
OR
Find a pixel location (i, j) where the pixels match template T2
OR
Find a pixel location (i, j) where the pixels match template T3
OR
Find a pixel location (i, j) where the pixels match template T4
If central pixel is not an endpoint pixel, and has connectivity number of 1, delete the pixel
For the pixels marked for deletion, set them to white.
Apply negative transformation to derive desired thinned image

Figure 24. Flowchart to summarise the main steps in the algorithm (Task 3)

Discussion & Conclusion

a. Explanation for chosen method

Before we decided to employ the Stentiford thinning algorithm for this task, we compared it with another commonly used thinning algorithm known as the Zhang-Suen thinning algorithm. The objective of the Zhang-Suen thinning algorithm is also to take a binary image and derive a one-pixel wide skeleton structure of that image while retaining the structure and shape of the original binary image. The Zhang-Suen thinning algorithm is a 2-pass algorithm, which means that it performs two sets of checks to remove pixels from the image, for each iteration. These 2 passes are defined below accordingly.

The Zhang-Suen thinning algorithm removes a pixel if it satisfies all the following conditions:

Pass 1:

1. The pixel is black and has eight neighbours.
2. The number of black pixels among the eight neighbours around pixel (i, j) is between and inclusive of 2 and 6.
3. The number of transitions from white to black, in the sequence of the eight neighbours around pixel (i, j), where the sequence starts and ends at the same neighbour, is exactly equal to 1.
4. At least one of the north (N3), east (N1), and south (N7) neighbours is white.
5. At least one of the east (N1), south (N7), and west (N5) neighbours is white.

Pass 2:

1. The pixel is black and has eight neighbours.
2. The number of black pixels among the eight neighbours around pixel (i, j) is between and inclusive of 2 and 6.
3. The number of transitions from white to black, in the sequence of the eight neighbours around pixel (i, j), where the sequence starts and ends at the same neighbour, is exactly equal to 1.
4. At least one of the north (N3), east (N1), and west (N5) neighbours is white.
5. At least one of the north (N3), south (N7), and west (N5) neighbours is white.

If a pixel is marked for deletion by either Pass 1 or Pass 2, then it is deleted. These passes are repeated iteratively until there is not a pixel marked for deletion by either pass.

Figure 25 below displays the labels of the eight neighbours around the central pixel (i, j) in the 8-connectivity neighbourhood.

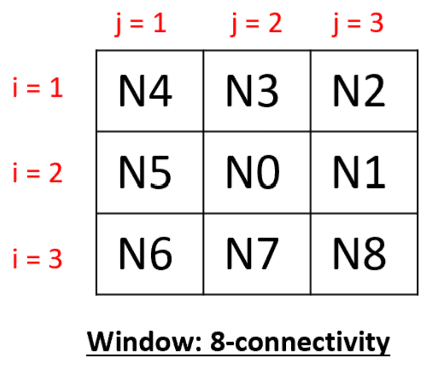


Figure 25. Neighbours in 8-connectivity

Figure 26 below shows the comparison of the one-pixel thin image derived by using the Stentiford thinning algorithm and the Zhang-Suen thinning algorithm.

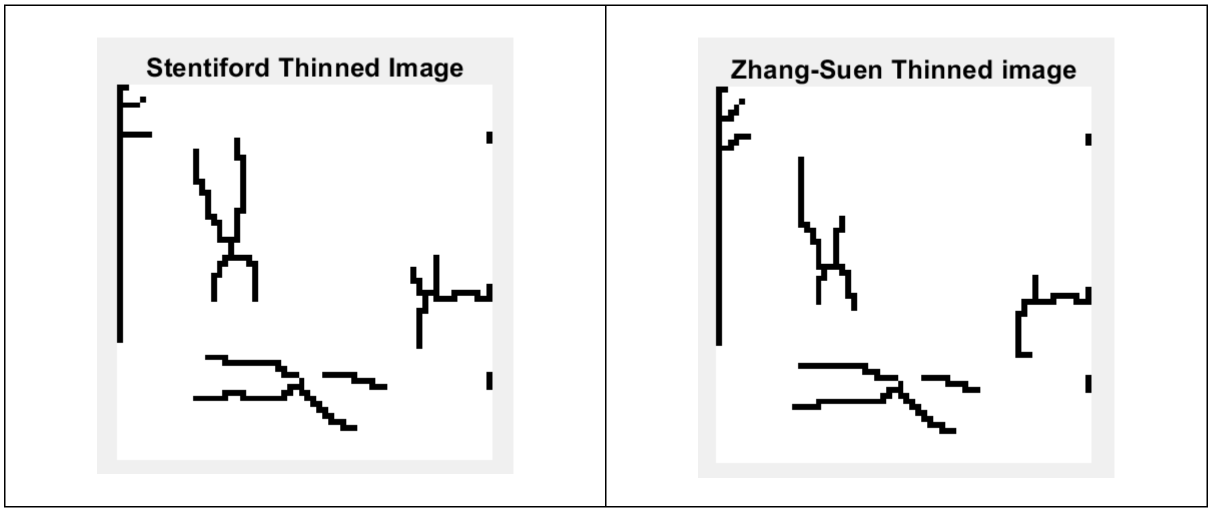


Figure 26. Comparison between one-pixel thin images derived from Stentiford thinning algorithm and Zhang-Suen thinning algorithm

As seen from Figure 26, the one-pixel thin image derived from the Zhang-Suen thinning algorithm lacks many details in the structure of the chromosomes, especially for the top chromosome and the right chromosome. In contrast, the one-pixel thin image derived from the Stentiford thinning algorithm successfully retains the shape and structure of all three chromosomes to a large extent. Therefore, the Stentiford thinning algorithm proved to be more effective for our application, and hence we chose to employ this method over the Zhang-Suen thinning algorithm.

b. Investigation performed

Besides the detail and quality of the output one-pixel thin image produced by both the Stentiford thinning algorithm and the Zhang-Suen thinning algorithm, we investigated and compared the time difference of running each algorithm. Running the Stentiford thinning algorithm took a total time of 3.270 seconds (Figure 27) while running the Zhang-Suen thinning algorithm took a total time of 2.198 seconds (Figure 28). Therefore, it is evident that the Zhang-Suen thinning algorithm runs much faster and is more efficient than the Stentiford thinning algorithm. As our image is simple, this time difference is not significant. However, for images which are more complex and contain a lot more objects, the time difference could be magnified and therefore needs to be taken into consideration in deciding which algorithm to use for thinning the objects.

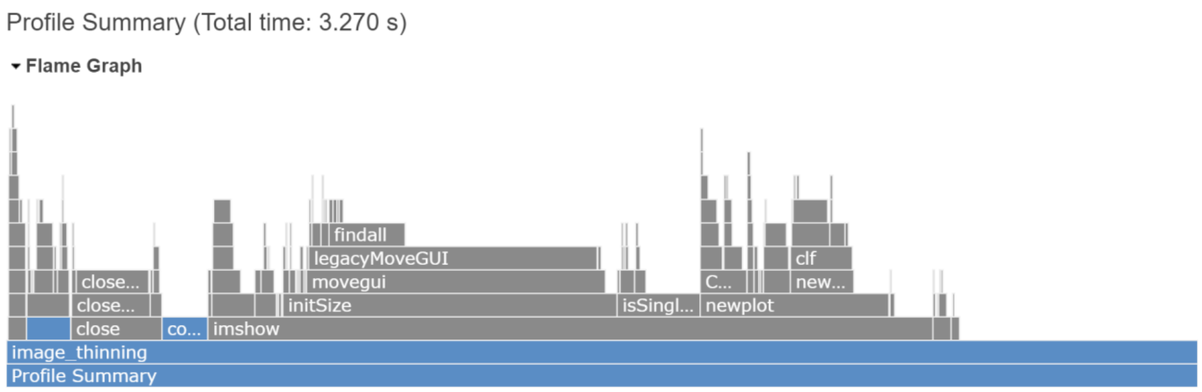

Profile Summary (Total time: 3.270 s)

Figure 27. Time profile for Stentiford thinning algorithm

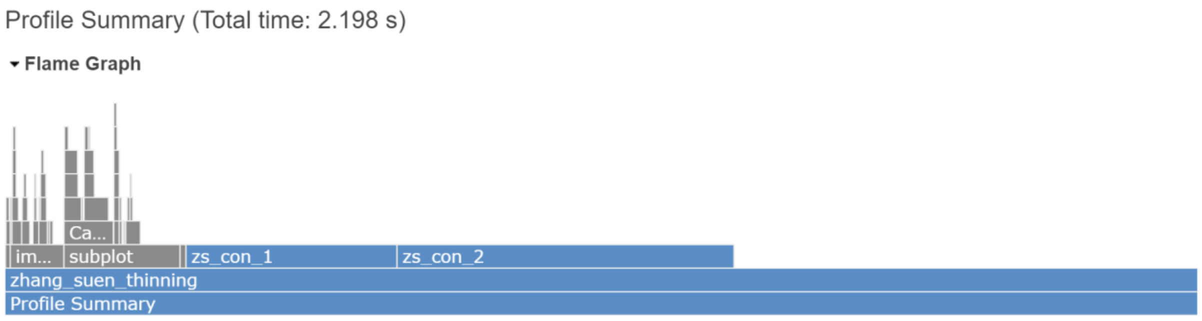


Figure 28. Time profile for Zhang-Suen thinning algorithm

<u>Lessons learnt</u>

Existing algorithms are useful in helping users complete a specific task. However, one lesson learnt is that the final decision of which framework/algorithm to employ for the user's application is dependent on two main factors – accuracy and efficiency. For our project, accuracy is prioritised over efficiency and therefore, we chose to use the Stentiford thinning algorithm, given that the time difference between both algorithms was not significant. However, in future, when working on larger-scale projects, tests need to be conducted to compare the difference in accuracy and efficiency of different algorithms. User's decision on which method to employ should always aim to strike a good balance between accuracy and efficiency, unless there are strict guidelines on either aspect.

1.4 Image Outlining

<u>Introduction to the problem</u>

Aside from obtaining the skeleton of the objects as explained above, binary image processing can also be done to obtain the outlines of objects. This process essentially marks the interior of objects as background pixels, leaving only the boundaries or edges of the objects. For this task, we implement an algorithm to determine the outlines, and compare the result with that of the existing MATLAB image processing toolbox method.

<u>Description of the algorithm and flowchart</u>

In the algorithm used to outline the objects, we first initialise a new image matrix to be of the same size as the original binary image matrix, and set all the cells to 1 (white background). The given binary image is subsequently scanned in two passes. Each pass is to separately detect changes in the x and y directions, whereby a change in binary level from 0 to 1 or vice versa would indicate the presence of an edge. For changes in the y direction, each pixel is checked row by row; if the current pixel value is different from that of the pixel above it, the corresponding pixel in the new image matrix is set to 0 (black edge). Similarly, for changes in the x direction, the pixels are iterated through column by column, comparing every pixel with the pixel to its right, modifying the corresponding pixels in the new image matrix accordingly. At the end of the two passes, the new image matrix will contain the object outlines as shown in Figure 29 below.

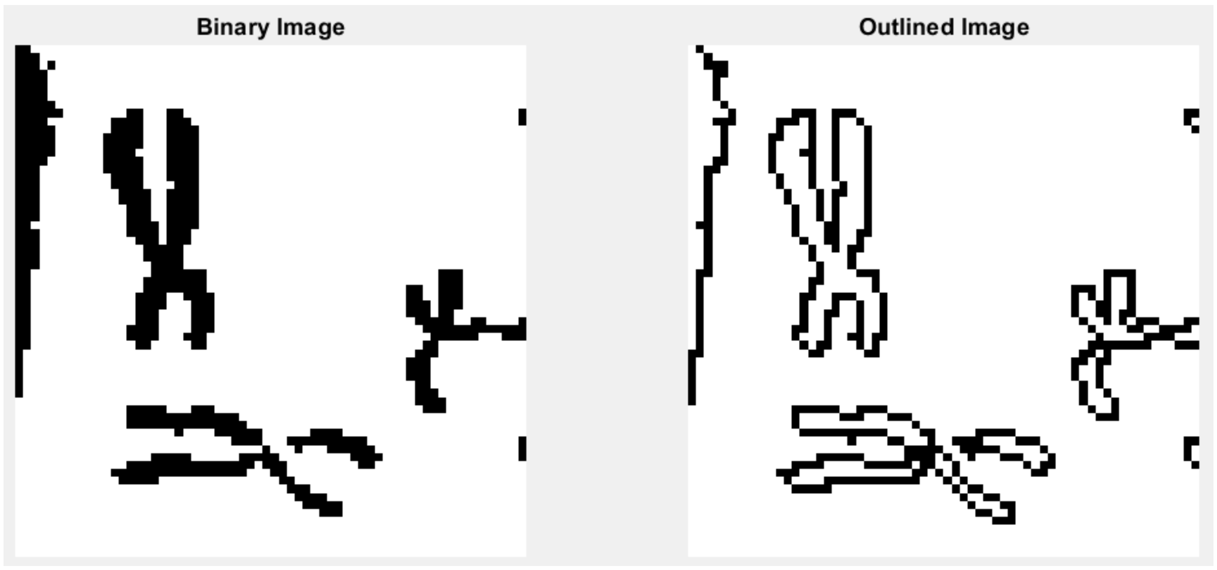


Figure 29. Binary image and outlined image

A flowchart to summarise the main steps of the algorithm is illustrated in Figure 30 below.

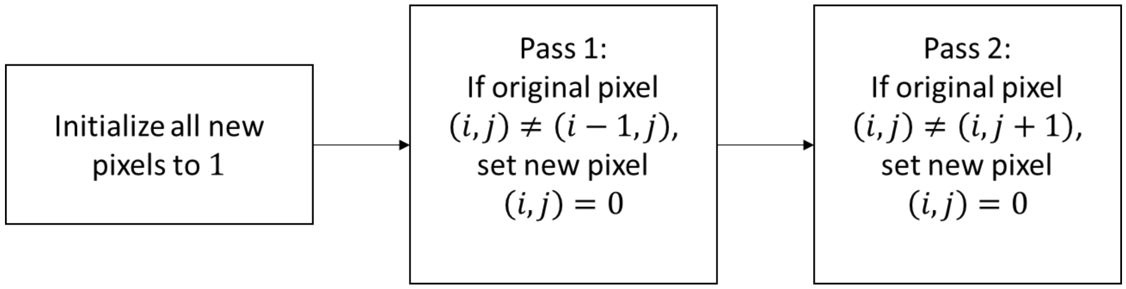

Initialize all new pixels to 1
Pass 1: If original pixel $(i,j) \neq (i-1,j)$, set new pixel $(i,j) = 0$
Pass 2: If original pixel $(i,j) \neq (i,j+1)$, set new pixel $(i,j) = 0$

Figure 30. Outlining algorithm flowchart

Discussion & Conclusion

a. Explanation for chosen method

This method is fairly straightforward and can be easily implemented without using the MATLAB Image Processing Toolbox. We decided to use our own function implementation in order to demonstrate our understanding of this particular binary image processing method.

b. Investigation performed

Nevertheless, we explored the method implemented in the MATLAB Image Processing Toolbox that can also outline the objects of a binary image. Passing in the original binary image and the *'remove'* option as arguments to the *bwmorph* method, the shape outline is returned. Figure 31 below shows a side-by-side comparison of the results obtained from the two methods.

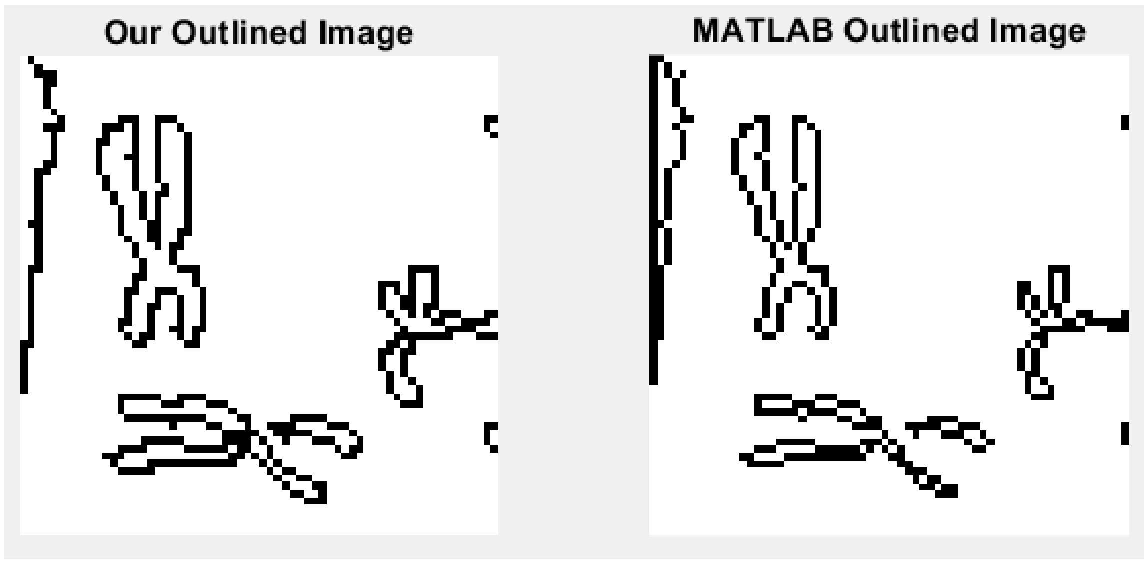


Figure 31. Comparison between the outlined image obtained from our algorithm and MATLAB's bwmorph method

With reference to the figure above, the objects appear thinner using MATLAB's implementation. This is because internally, the *bwmorph* method marks a pixel as "background" if all its 4-connected neighbours are "object" pixels. In other words, it outlines the inner boundaries of objects. On the other hand, our implementation mainly outlines the outer boundaries of the top and left sides of an object, and the inner boundaries for the bottom and right sides, resulting in thicker objects.

c. Lessons learnt

Different implementations can achieve a similar outcome, but if there are specific requirements on the image outlines, the user can employ the desired implementation accordingly. In writing our own implementation, we also learned to consider space complexity, as this would be critical when the image of interest grows in size. Initially, we made use of three matrices of the same size as the original image; two as temporary result matrices for the two passes, and one for combining the temporary result matrices. As we analysed the code further, we realised that a single result matrix is sufficient if we initialised it correctly, so updating the cells in the two passes can be done directly in the result matrix.

1.5 Image Connected-Components Labelling

Introduction to the problem

Connected components labelling is a binary image processing method which groups together object pixels that belong to the same region, and assigns unique labels to each of the groups. For this task, it is hence important to first define the pixel connectivity adopted. Typically used are 4-connectivity and 8-connectivity, where a pixel can have four or eight neighbours, as illustrated in Figure 32 below.

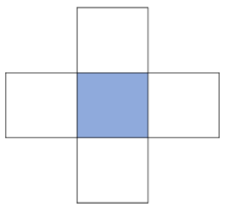
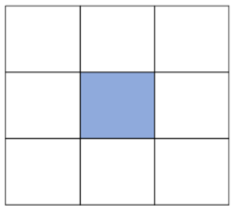



Figure 32. 4-connectivity and 8-connectivity

Description of the algorithm and flowchart

This algorithm requires two passes through the image, performing a left-right top-down scan on the pixels. The first pass performs label propagation, checking every pixel against its neighbours-of-interest. In the case of 4-connectivity, two neighbouring pixels (top and left) of the current pixel are checked, whereas for 8-connectivity, four neighbouring pixels (top, left, top-left and top-right) are checked. We begin by assigning the label "1" to the first occurrence of an object pixel. Subsequently, as we iterate through each pixel, if all its neighbours-of-interest are not labelled, we increment the current label by 1 and assign this new label to the current pixel. However, if there is only one neighbour-of-interest that has been labelled, the current pixel takes on the same label. Furthermore, if more than one of its neighbours-of-interest are labelled, the smallest label is selected to be the label of the current pixel, and each of the equivalence pairs is recorded in an equivalence table.

Once we obtain the complete equivalence table, the MATLAB *dfsearch* (depth-first search) function is used to resolve the table entries and derive the equivalent classes. Thereafter, the minimum label in each equivalence class is used in the second pass through the image, updating the pixel labels.

Finally, for better visualisation of the different connected components, a unique colour is randomly generated for each connected component. Figure 33 below shows a flowchart of the algorithm.

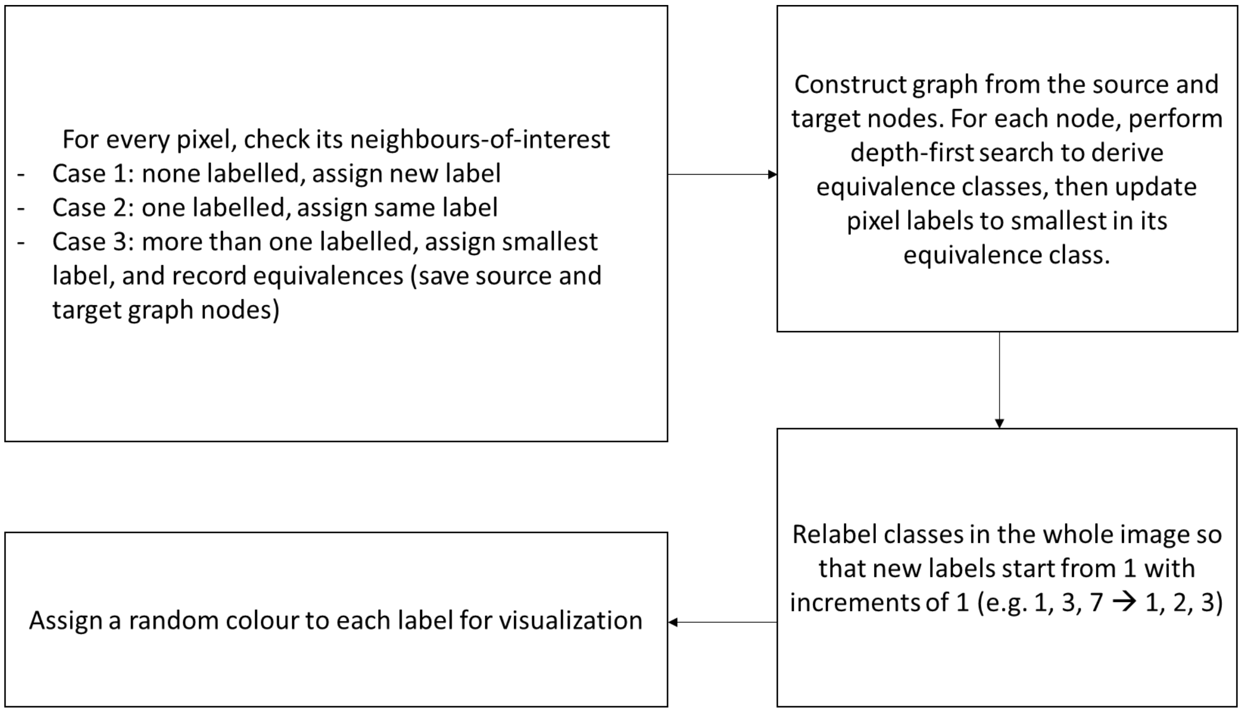

For every pixel, check its neighbours-of-interest
- Case 1: none labelled, assign new label
- Case 2: one labelled, assign same label
- Case 3: more than one labelled, assign smallest label, and record equivalences (save source and target graph nodes)
Construct graph from the source and target nodes. For each node, perform depth-first search to derive equivalence classes, then update pixel labels to smallest in its equivalence class.
Relabel classes in the whole image so that new labels start from 1 with increments of 1 (e.g. 1, 3, 7 → 1, 2, 3)
Assign a random colour to each label for visualization

Figure 33. Connected component labelling algorithm flowchart

<u>Discussion & Conclusion</u>

a. Investigation performed

Both the 4- and 8-connectivity methods are implemented and their results are shown in Figure 34 below.

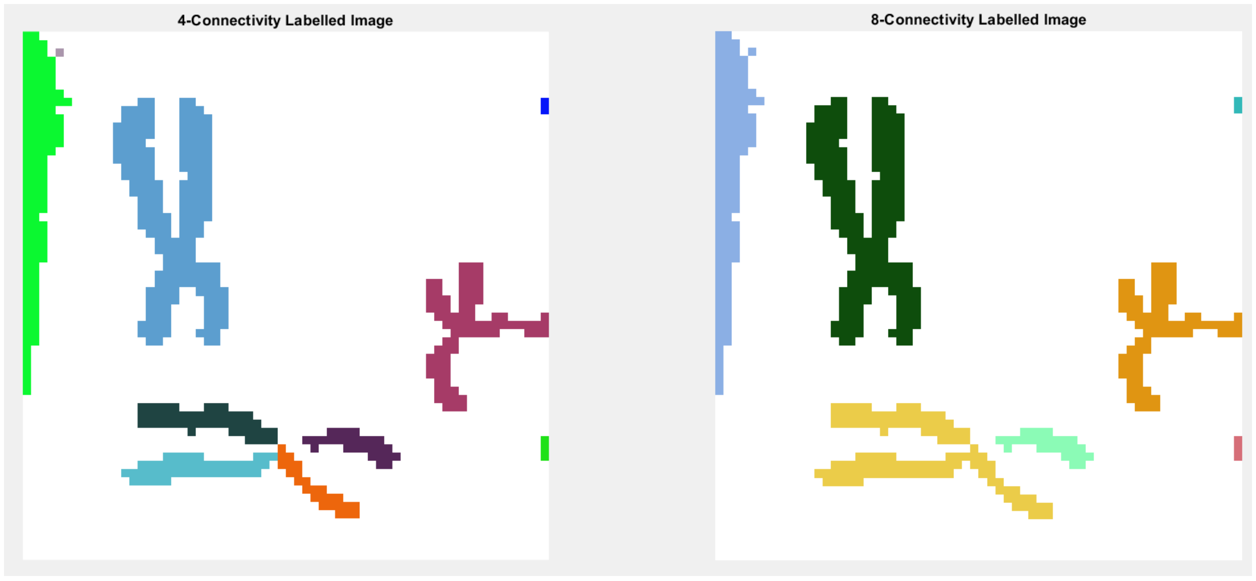


Figure 34. 4-connectivity and 8-connectivity labelled image

b. Explanation for chosen method

After analysis, the chosen method is the 8-connectivity connected component labelling. This is because 7 components are detected using the 8-connectivity method, which is closer to the actual desired number of 3 chromosomes, compared to 10 using the 4-connectivity method. Moreover, the bottommost chromosome is labelled as 4 different components in the 4-connectivity method, whereas the 8-connectivity is better able to group together tightly-coupled pixels.

c. Lessons learnt

There are definitely different use cases for the two methods described above. Although the 8-connectivity method is more accurate in our application, it is worth noting that the 4-connectivity algorithm is more time-efficient, as it performs only half the checks of that done in the 8-connectivity algorithm. In applications where time is a more important factor compared to accuracy, the 4-connectivity method may be used, but the accuracy of the labelled image may be compromised.

1.6 Image Rotation

Introduction to the problem

Image rotation is a geometric transformation operation used in computer graphics. Furthermore, it is a commonly-used image processing method with essential applications in alignment, image matching, image reconstruction, and other image-based algorithms. For this task, we are required to rotate the original image counter-clockwise by 30 degrees, 60 degrees and 90 degrees respectively.

Description of the algorithm and flowchart

In image rotation, the centre of the image to a certain pixel can be regarded as a vector, and the rotation of the vector about the midpoint can rotate the pixel, and the rotation of all pixels completes the rotation of the image.

Firstly, we initialise a geometric transformation function for the image to be rotated. The purpose of this function is to transform the spatial coordinates of the image and perform intensity interpolation, which is important in the process of image rotation. The function, consisting of its inputs and output, are displayed below in Figure 35.

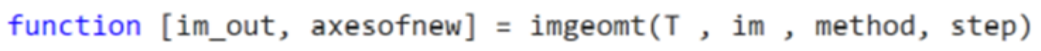
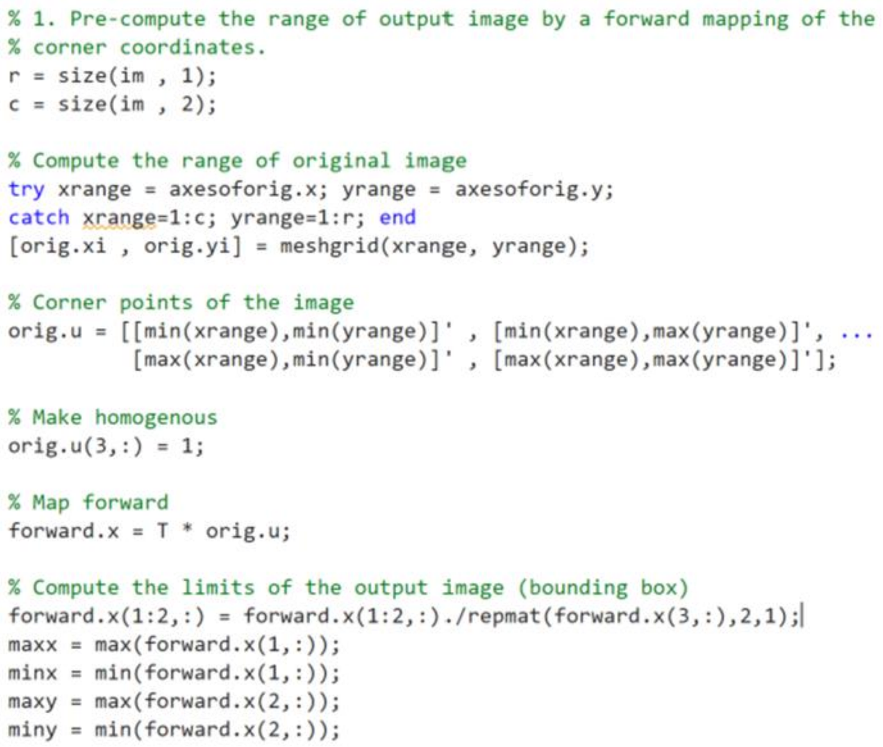

```
function [im_out, axesofnew] = imgeomt(T , im , method, step)
```

Figure 35. Geometric transformation function for image rotation

From Figure 35, it is evident that the inputs to the function ‘imgeomt’ are *T*, *im*, *method*, and *step*. *T* refers to the transformation matrix, *im* refers to the image to be transformed, *method* refers to the method of interpolation and *step* refers to the inverse of the sampling factor, which determines how fine the grid of spatial coordinates of the new image is prepared. The outputs of the function are *im_out*, which refers to the transformed/rotated image, and *axesofnew* contains the x and y spatial coordinates of the newly rotated image.

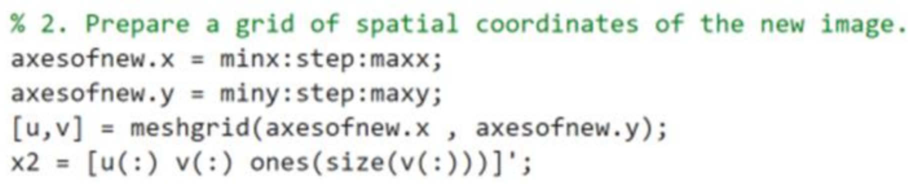
In the geometric transformation function, there are a total of four main steps to transform the image and perform intensity interpolation. The first step is to pre-compute the range of output image spatial coordinates by applying a forward mapping of the corner coordinates. To do so, we need to compute the range of the original image spatial coordinates, obtain the homogenous corner points of the image, and map forward to compute the limits of the output image, which also represents the bounding box. A snippet of the MATLAB code is shown below in Figure 36.



```
% 1. Pre-compute the range of output image by a forward mapping of the
% corner coordinates.
r = size(im , 1);
c = size(im , 2);

% Compute the range of original image
try xrange = axesoforig.x; yrange = axesoforig.y;
catch xrange=1:c; yrange=1:r; end
[orig.xi , orig.yi] = meshgrid(xrange, yrange);

% Corner points of the image
orig.u = [[min(xrange),min(yrange)]' , [min(xrange),max(yrange)]', ...
          [max(xrange),min(yrange)]' , [max(xrange),max(yrange)]'];

% Make homogenous
orig.u(3,:) = 1;

% Map forward
forward.x = T * orig.u;

% Compute the limits of the output image (bounding box)
forward.x(1:2,:) = forward.x(1:2,:)./repmat(forward.x(3,:),2,1);
maxx = max(forward.x(1,:));
minx = min(forward.x(1,:));
maxy = max(forward.x(2,:));
miny = min(forward.x(2,:));
```

Figure 36. Pre-computing range of output image

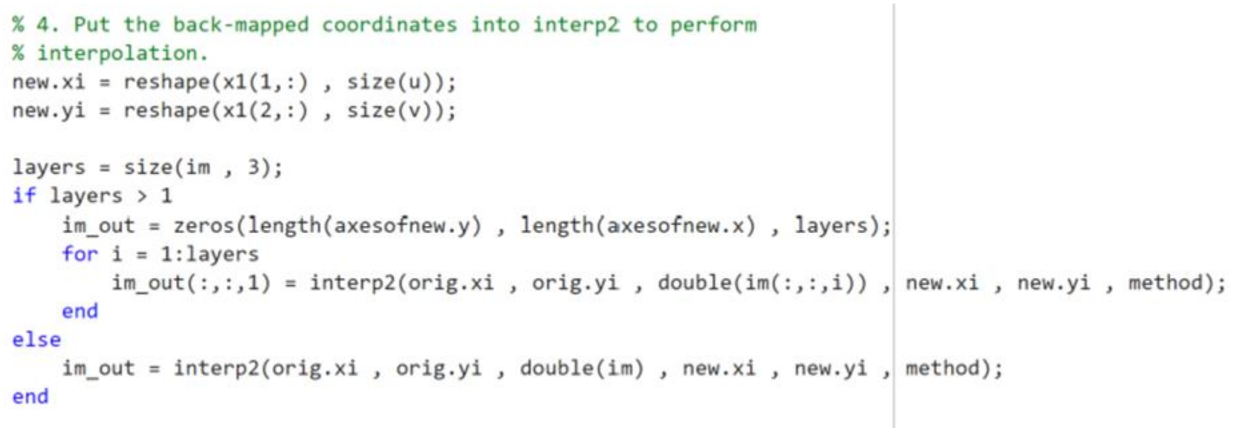
The second step is to prepare a grid of spatial coordinates of the new output image. As mentioned earlier, one of the function input ‘step’ determines how fine the grid of spatial coordinates of the new image is produced. A higher value will result in a coarser grid of spatial coordinates, which can ultimately lessen the quality and intensity interpolation of the rotated image. However, it reduces computational time. Whereas, a smaller value will result in a finer grid of spatial coordinates, which improves the quality of the output image, but increases computational time. We have chosen a ‘step’ value of 0.05, which we deem to be sufficient to produce a good quality of output image as desired. A snippet of the MATLAB code is illustrated below in Figure 37.



```
% 2. Prepare a grid of spatial coordinates of the new image.
axesofnew.x = minx:step:maxx;
axesofnew.y = miny:step:maxy;
[u,v] = meshgrid(axesofnew.x , axesofnew.y);
x2 = [u(:) v(:) ones(size(v(:)))]';
```

Figure 37. Preparing grid of spatial coordinates of output image

The third step is to perform backward mapping of the newly obtained spatial coordinates, which is essential in the process of image rotation. Lastly, we put the back-mapped coordinates into the MATLAB in-built function ‘interp2’ to perform intensity interpolation, to complete the image rotation process. A snippet of the MATLAB code is displayed below in Figure 38.



```
% 4. Put the back-mapped coordinates into interp2 to perform
% interpolation.
new.xi = reshape(x1(1,:) , size(u));
new.yi = reshape(x1(2,:) , size(v));

layers = size(im , 3);
if layers > 1
    im_out = zeros(length(axesofnew.y) , length(axesofnew.x) , layers);
    for i = 1:layers
        im_out(:,:,1) = interp2(orig.xi , orig.yi , double(im(:,:,i)) , new.xi , new.yi , method);
    end
else
    im_out = interp2(orig.xi , orig.yi , double(im) , new.xi , new.yi , method);
end
```

Figure 38. Performing intensity interpolation

The original image, and the 30-degree, 60-degree and 90-degree rotated images are displayed in Figure 39 below.

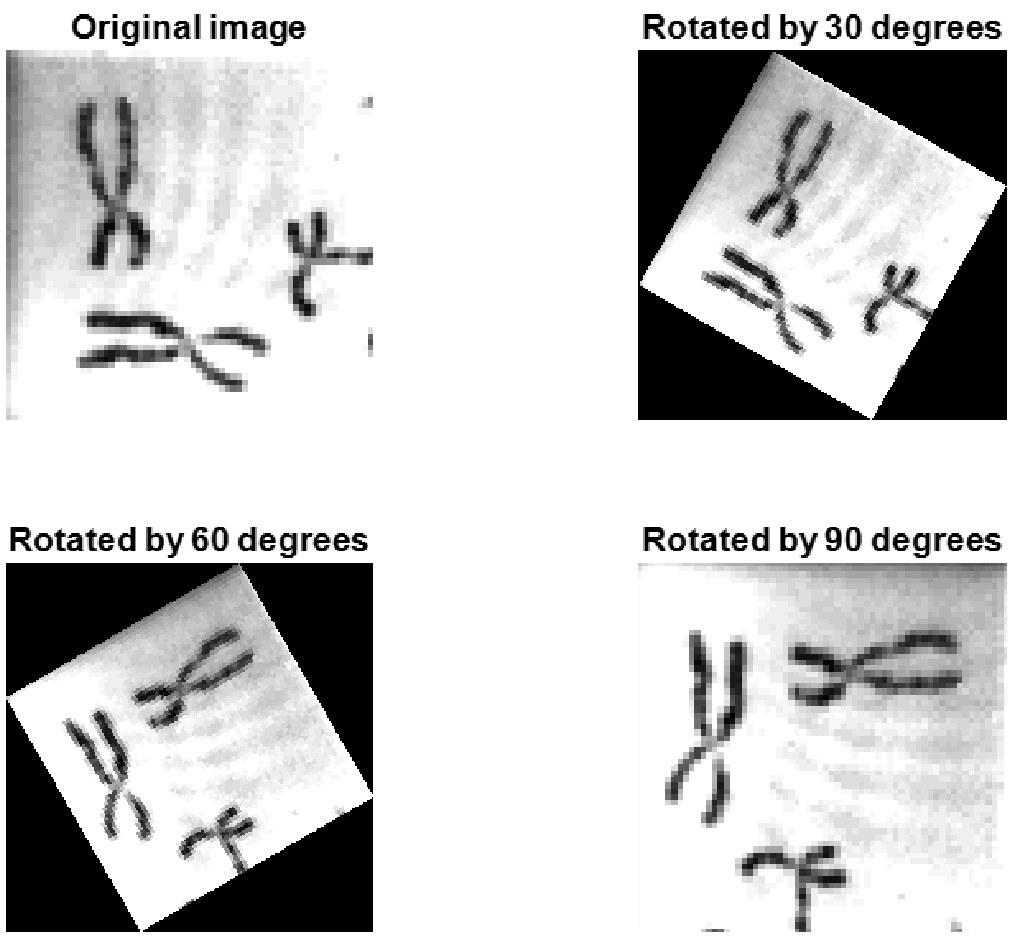


Figure 39. Original image and rotated images (30, 60, 90 degrees)

To summarise the main steps in the algorithm used, a flow chart for the image rotation algorithm is presented in Figure 40 below.

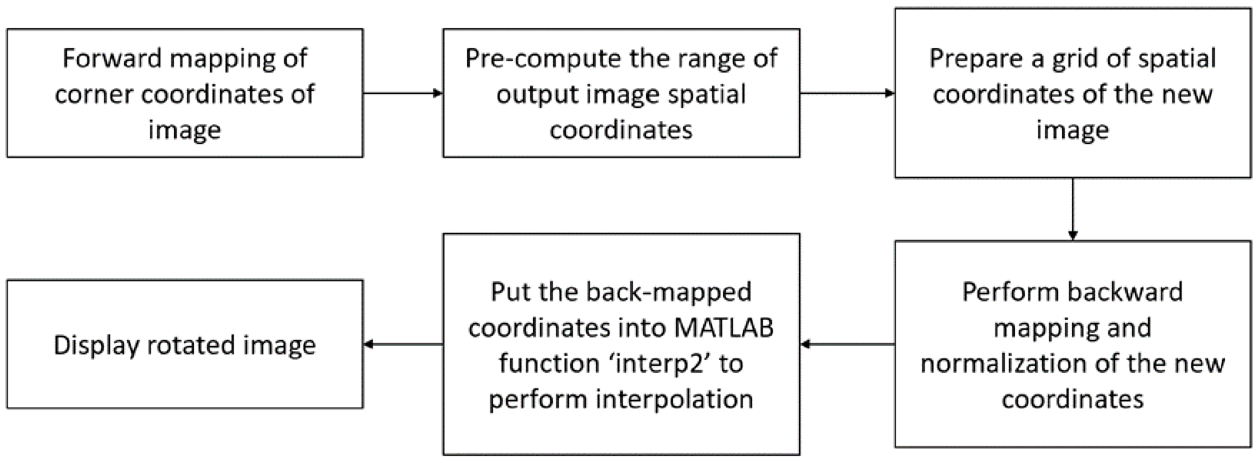

Forward mapping of corner coordinates of image
Pre-compute the range of output image spatial coordinates
Prepare a grid of spatial coordinates of the new image
Perform backward mapping and normalization of the new coordinates
Put the back-mapped coordinates into MATLAB function 'interp2' to perform interpolation
Display rotated image

Figure 40. Flowchart to summarise the main steps in the algorithm (Task 6)

Discussion & Conclusion

a. Investigation performed

For this image rotation task, we explored a total of three different methods of interpolation, to find out which one best suits our application. These three methods refer to nearest neighbours interpolation, linear/bilinear interpolation, and cubic/bicubic interpolation. The output images of these three methods are displayed below in Figure 41.

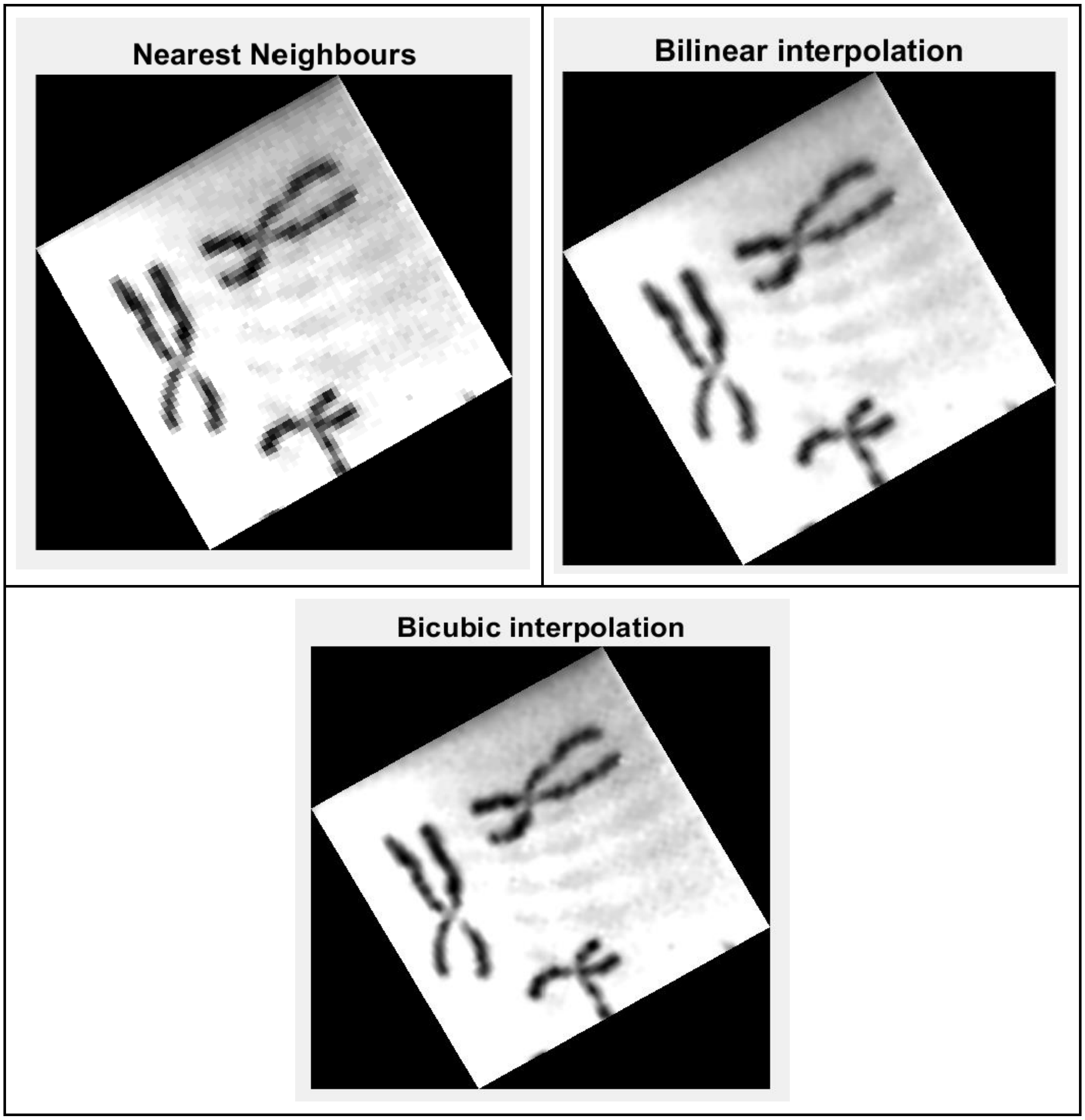


Figure 41. Comparison of output images of nearest neighbours interpolation, bilinear interpolation, and bicubic interpolation

For the comparison of these three methods, we utilised the MATLAB function ‘interp2’ implementation of interpolation.

According to MATLAB, the linear/bilinear interpolation is a method of interpolation whereby the interpolated value at a query point is dependent on linear interpolation of the values at neighbouring grid points in each respective dimension. However, it needs to have at least two grid points in each dimension and consumes more memory than the other interpolation method ‘nearest’.

The nearest neighbours interpolation is a method of interpolation whereby the interpolated value at a query point is the value at the nearest sample grid point. It is the fastest method computationally and requires only a small amount of memory to run. Like the linear interpolation method, it also requires at least two grid points in each dimension.

The cubic/bicubic interpolation is a method of interpolation whereby the interpolated value at a query point is based on a cubic interpolation of the values at neighbouring grid points in each respective dimension, and the interpolation is derived through using a cubic convolution. Compared to the linear and nearest neighbours interpolation, the cubic/bicubic interpolation requires at least four points in each dimension, and takes up more memory and computation time.

b. Explanation for chosen method

For our application, we have chosen to implement the nearest neighbours interpolation method. As seen in Figure 40, although the output image of the nearest neighbours interpolation method appears to be the least smooth out of the three explored methods, it effectively preserves the appearance of the original unrotated image. For our application, we merely want to rotate the original image, without any additional smoothing of the image. From Figure 41, it is evident that the output image of the nearest neighbours interpolation method appears pixelated (similar to the original image), while that of the bilinear and bicubic interpolation methods appear smoother. For applications which require smoothing of the rotated image, bicubic interpolation method should be implemented for the best improvement in appearance of the image.

c. Lessons learnt

The results obtained by this method are quite good, and the resolution is similar to the original image. In addition, there is another method of image rotation that can be implemented without using the MATLAB in-built function ‘imrotate’ . The essence of that method is vector rotation, which is easy to understand and simple to implement. Using that method of image rotation, the centre of the image to a certain pixel can be regarded as a vector, and the rotation of the vector about the midpoint can rotate the pixel, and the rotation of all pixels completes the rotation of the image. We can then figure out the position of each pixel of the new image in the original image before rotation, and then assign the corresponding pixel in the original image to the new image, and finally get the output rotated image.

2. Image 2: Characters

2.1 Image Display

Introduction to the problem

As opposed to Section 1.1 Image Display which requires decoding the text and mapping the characters to their corresponding gray levels, the image in “jpg” format is directly provided for this task, reducing the problem to a simple image display.

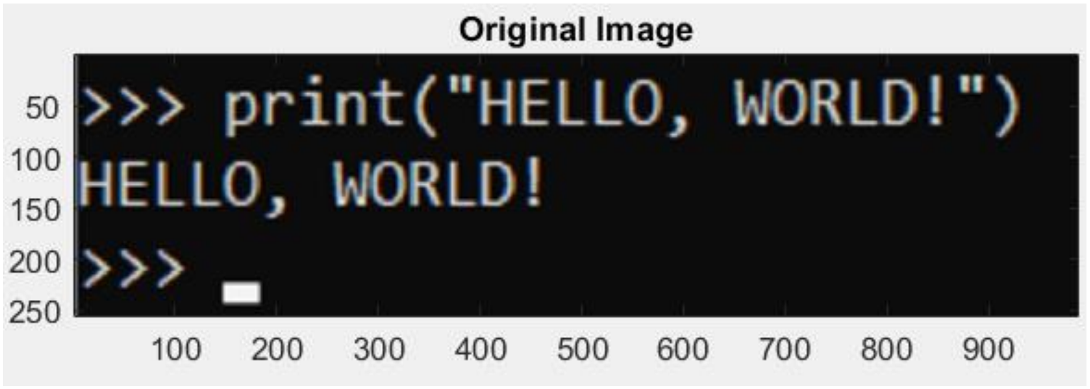
Description of the algorithm and flowchart

For this task, we make use of the MATLAB *imread* function. A snippet of the code is shown in Figure 42 below.

```
ori_image = imread('hello_world.jpg');
figure(1);
image(ori_image);
axis image;
title('Original Image');
```

Figure 42. MATLAB code to display image

The output image is displayed in Figure 43 below.



```
>>> print("HELLO, WORLD!")
HELLO, WORLD!
>>> 
```

Figure 43. Original image

2.2 Sub-Image Creation

Introduction to the problem

This task requires the creation of a smaller sub-image that is part of the original image, specifically comprising only the middle line of three lines – HELLO, WORLD.

Description of the algorithm and flowchart

We use the MATLAB function *imcrop* to obtain the desired sub-image, passing in the original image, as well as the size and position of the crop rectangle, specified as a vector of the form `[xmin ymin width height]` as shown in Figure 44 below. Since the height of each line in the image can be assumed to be the same, we determine the minimum y pixel coordinate (`ymin`) of the middle line by dividing the height of the whole image by 3. The height of the sub-image can also be approximated as one-third of the original image height.

```
[h, w] = size(ori_image);
sub_image = imcrop(ori_image, [0 h/3 w h/3]);
figure(2);
image(sub_image);
axis image;
title('Sub-image');
```

Figure 44. MATLAB code for sub-image creation

The sub-image is displayed in Figure 45 below.

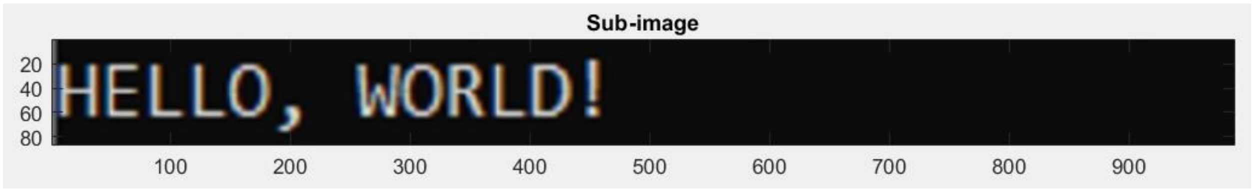


Figure 45. Sub-image

2.3 Image Thresholding and Binarization

Introduction to the problem

The task requires us to create a binary image from 2.2 using thresholding. As is done in 1.1 and 1.2, here we should convert the image from 2.2 to grayscale image first and then convert the grayscale image to a binary image. The main consideration to take note is to choose an appropriate threshold level to apply to the image, to obtain a suitable binary image. To choose a good threshold level, we need to rely on two techniques – observation of the image histogram and trial and error. Pixels with gray values below or equal to the threshold level will be converted to black pixels with gray value of 0 (object pixel), while pixels with gray values above the threshold level will be converted to white pixels with gray value of 1 (background pixel).

Description of the algorithm and flowchart

For details about the description of the algorithm and flowchart, please refer to 1.2 Image Thresholding and Binarization. The output image is displayed below in Figure 46.

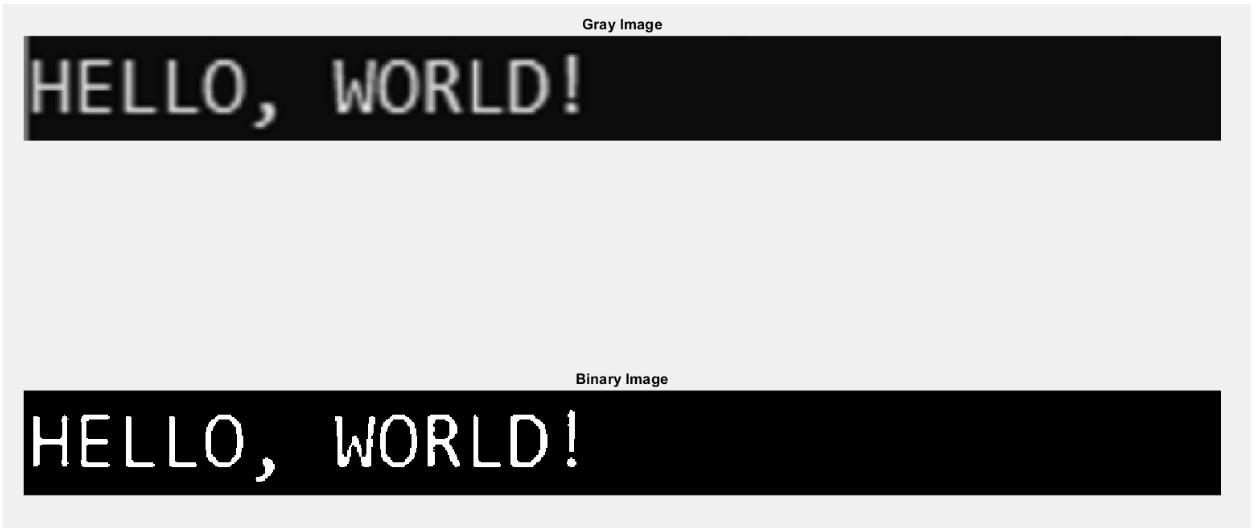


Figure 46. Gray image and binary image

2.4 Image Skeletonization (One-Pixel Thin Image)

Introduction to the problem

This task requires us to determine a one-pixel thin image of the characters. As is introduced in 1.3, a one-pixel thin image of the characters is basically its skeleton. The thinning or skeletonization process reduces the image complexity, as it removes details from the objects but maintains its basic shape and structure.

Description of the algorithm and flowchart

For details about the description of the Zhang-Suen thinning algorithm and flowchart, please refer to 1.3 Image Skeletonization (One-pixel Thin Image). The output image is displayed below in Figure 47.

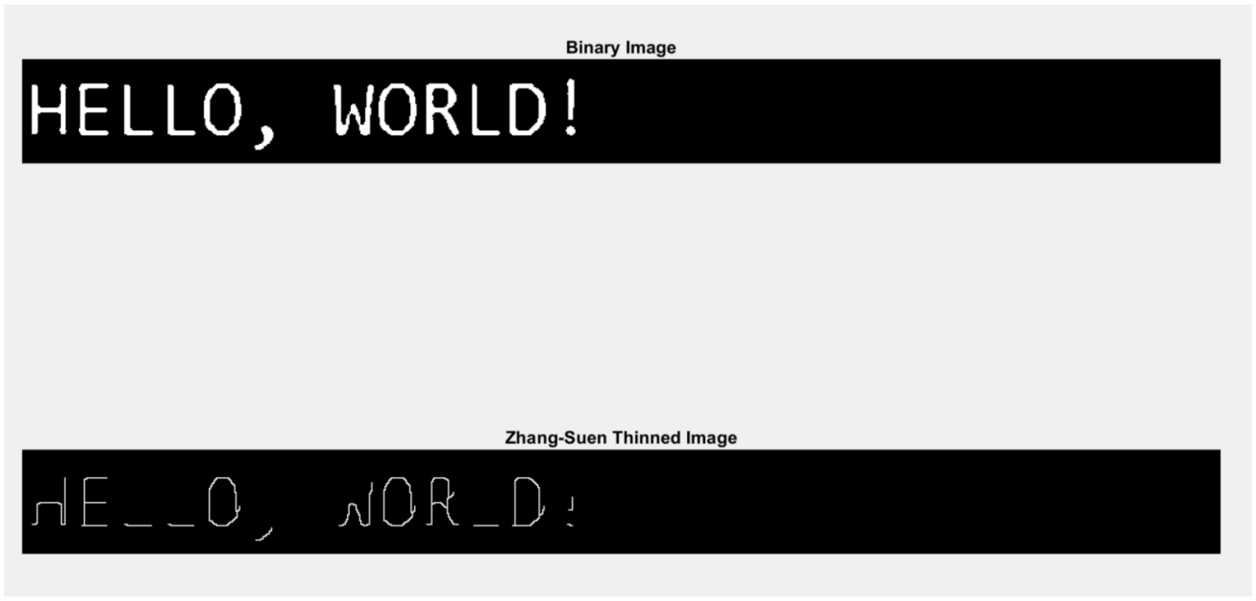


Figure 47. Binary image and Thinned image using Zhang-Suen thinning algorithm

Figure 48 illustrates the difference between the thinned image implementing the Zhang-Suen thinning algorithm, coded from scratch, and the thinned image utilising MATLAB's bwmorph method.

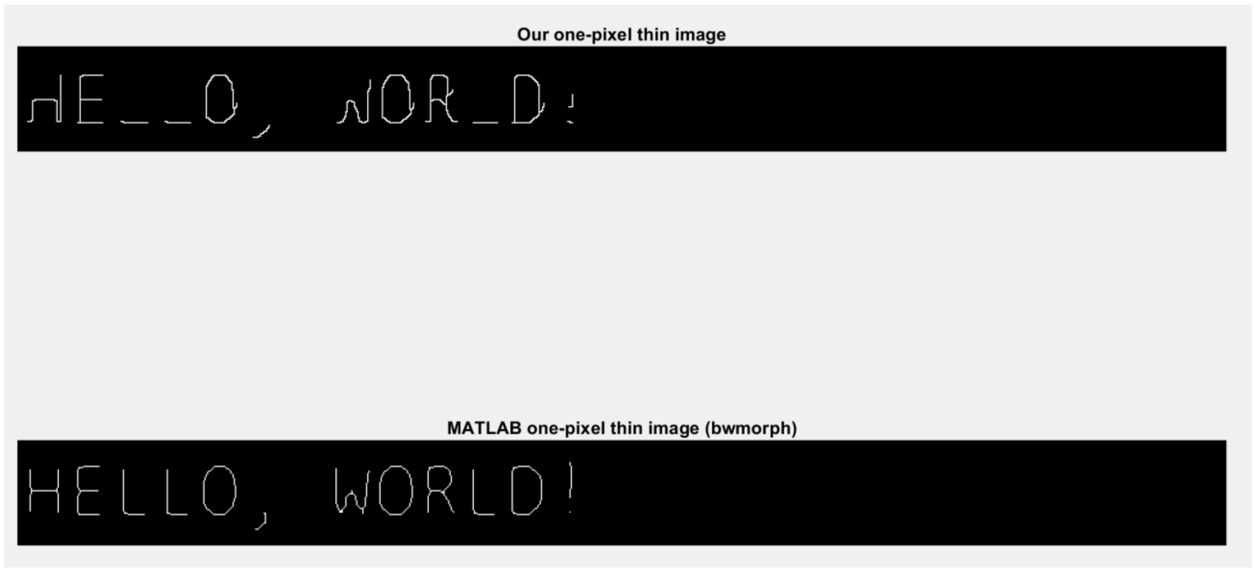


Figure 48. Comparison between the thinned image of our algorithm and MATLAB's bwmorph method

From Figure 48, it is evident that the one-pixel thin image coded from scratch, using the Zhang-Suen thinning algorithm lacks much details compared to the one-pixel thin image derived using MATLAB's bwmorph method. One of the possible reasons is because the Zhang-Suen thinning algorithm is relatively sensitive to noise in the image, and therefore is unable to perform as well as MATLAB's inbuilt function bwmorph.

2.5 Image Outlining

Introduction to the problem

As introduced earlier, the image outlining process marks the interior of objects as background pixels, leaving only the boundaries or edges of the objects. For this task, we use the same algorithm used for Image 1 to determine the outlines, and compare the result with that of the existing MATLAB image processing toolbox method.

Description of the algorithm and flowchart

The outlining algorithm illustrated in Figure 30 is reused for this task, but an additional step is performed to flip the pixels so that background pixels are black and the outlines are white as shown in Figure 49, as opposed to the outlining result of Image 1.

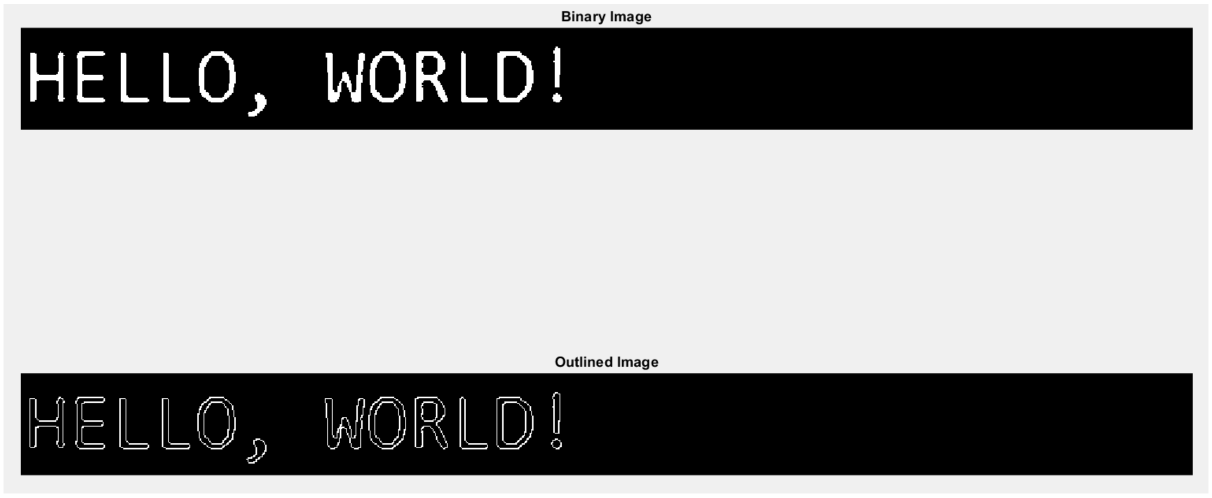


Figure 49. Binary image and outlined image

We again tested the MATLAB Image Processing Toolbox method - *bwmorph*. The result is found to be similar to that of our own implementation as illustrated in Figure 50 below.

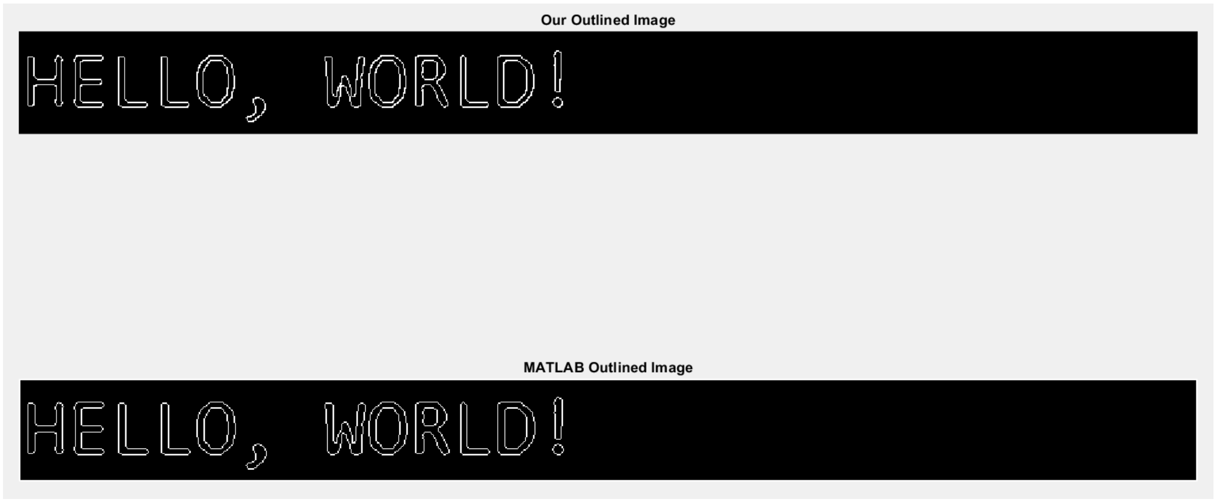


Figure 50. Comparison between the outlined image of our algorithm and MATLAB's bwmorph method

2.6 Image Labelling and Segmentation

<u>Introduction to the problem</u>

As described before, connected-components-labelling groups together object pixels that belong to the same region, and assigns unique labels to each of the groups. Subsequently, these groups can be segmented within their own regions to create new images each containing a single group. These images can then be used for other purposes such as image classification, For this task, both the 4-connectivity and 8-connectivity labelling algorithms implemented for Image 1 are reused, and an additional function to segment the different groups into individual images is implemented.

<u>Description of the algorithm and flowchart</u>

This task involves two main steps – firstly, label the connected components in an image; secondly, segment the connected components into individual images.

The 4- and 8-connectivity labelling algorithms used are the same as that implemented for Image 1, where the flowchart is illustrated in Figure 33. The results of labelling are as shown in Figure 51 below, where background pixels are coloured white to aid visualization.

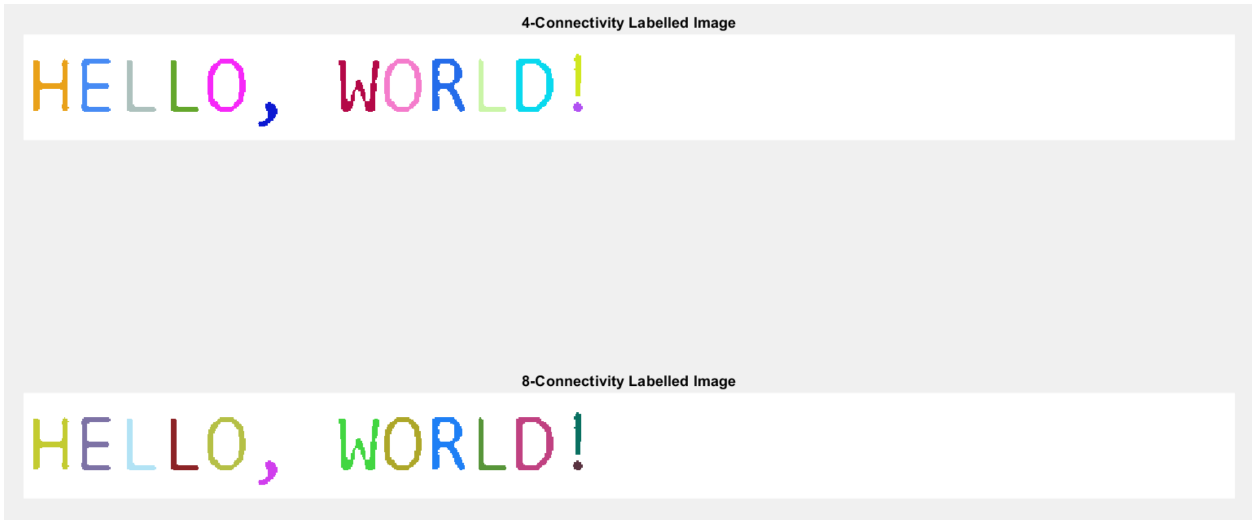


Figure 51. 4-connectivity and 8-connectivity labelled image

After the labelling step, the segmentation algorithm takes in the labelled image and the desired segment size as inputs. It first finds the number of unique labels (equivalent to the number of connected components) in the image. For each unique label, the minimum and maximum (bounding box) coordinates of the object pixels are determined. A new binary image of the specified segment size is created and initialized to be all 1 (white background pixels). Then, the offset of the top-left corner of the bounding box enclosing the object is calculated to ensure that the object is centered in the new image. Thereafter, the pixels within the bounding box of the object are traversed, checking for object pixels and setting the corresponding pixels in the new image as 0 (black object pixels). Figure 52below displays all the segments identified.

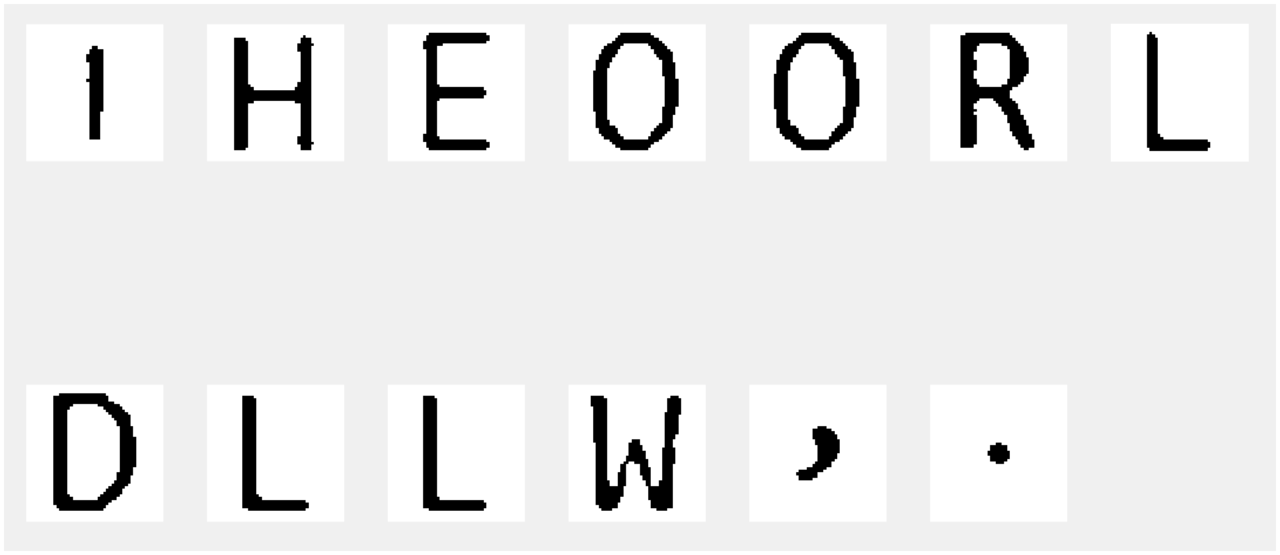


Figure 52. Segmented images of the original binary image

Finally, we select the 10 segmented images of interest and manually label each of them accordingly. The resulting labelled segments are shown in Figure 53.

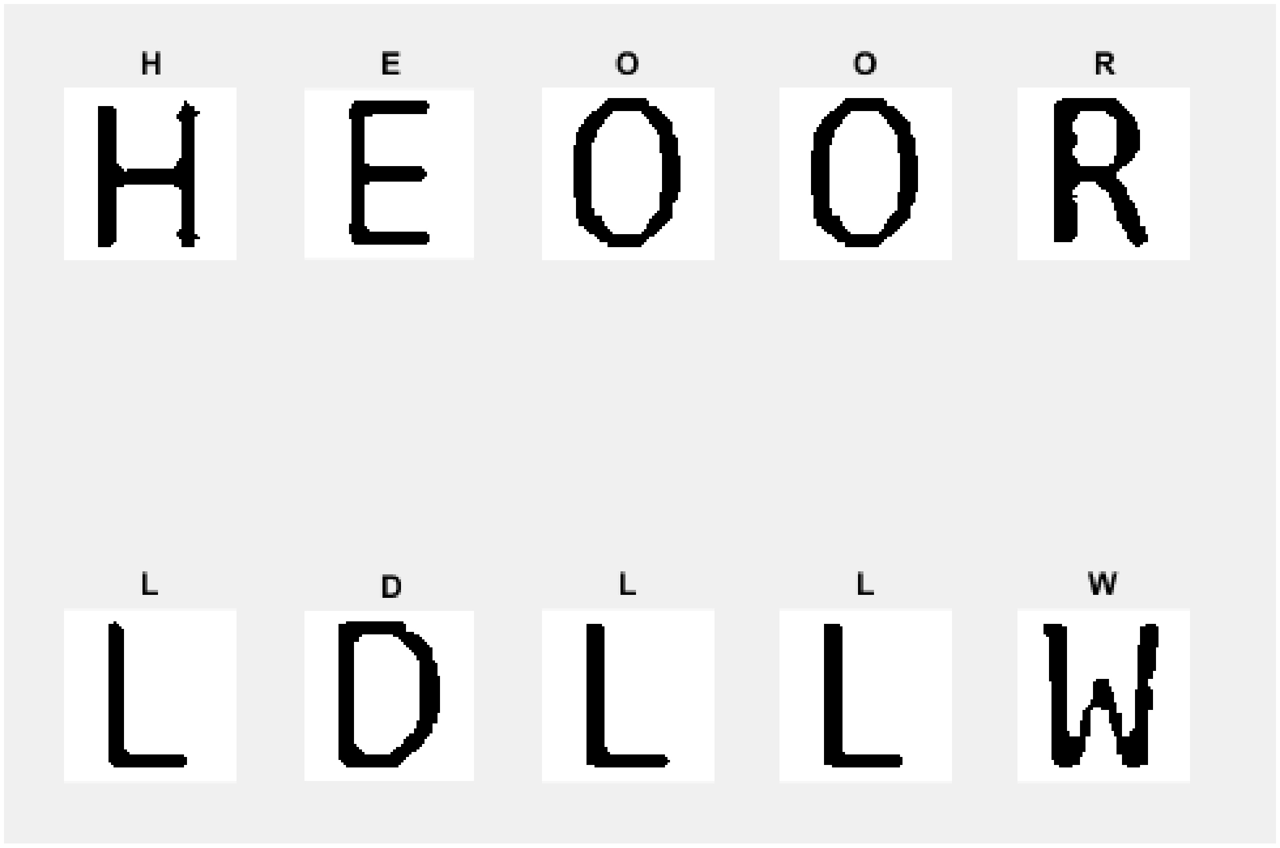


Figure 53. Selected segments with their labels

The algorithm flowchart is shown in Figure 54 below.

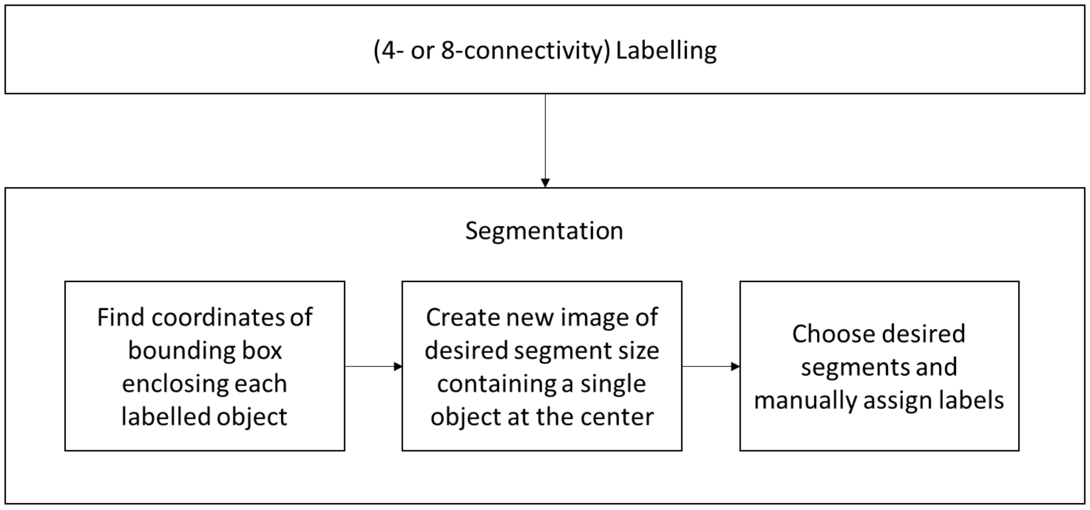

(4- or 8-connectivity) Labelling
Segmentation
Find coordinates of bounding box enclosing each labelled object
Create new image of desired segment size containing a single object at the center
Choose desired segments and manually assign labels

Figure 54. Labelling and segmentation algorithm flowchart

<u>Discussion & Conclusion</u>

a. Investigation performed

With reference to Figure 54 above, it can be observed that the 4- and 8-connectivity methods yield the same labelling results. The results from both methods used on Image 2 are ideal due to the considerably good quality of the original image which has low noise. The image quality is also further enhanced through thresholding and binarization.

During the segmentation process, we conducted a few trials of different segment sizes for the new images, and found that an acceptable size for the segmented images is 50-by-50 pixels.

b. Explanation for chosen method

After analysis, the 4-connectivity is chosen over the 8-connectivity connected component labelling as it is more time-efficient and is also sufficient to identify the connected components accurately. Besides, we decided to implement a function to automate the segmentation process instead of manually deciding the bounding boxes of each connected component, as the manual process can become tedious as the number of characters in the image increases.

c. Lessons learnt

As seen in Figure 52 above, the exclamation mark in the original image has been identified as two separate components by the labelling algorithm. While this is not an issue in our task of segmenting only uppercase alphabets, it could pose a problem if the user wanted to create segments for characters that inherently contain multiple components such as “i”. In such cases, either the labelling algorithm can be modified to consider corner cases, or manual labelling can be done for special characters.

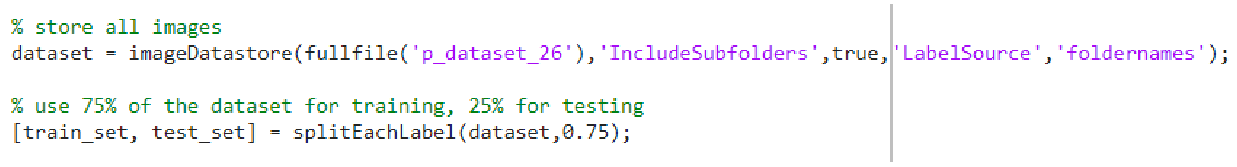
2.7 Image Classification

Introduction to the problem

In recent years, the technology of character segmentation, recognition and classification has developed rapidly, among which the most widely used is the recognition of vehicle licence plates at the entrance and exit of parking lots. Combining with the computing system to automatically determine the time and payment amount of vehicles in the parking lot, it has greatly improved the operation efficiency and the accuracy of information in the parking lot. For this task, we are required to use (conventional) unsupervised classification methods to train classifiers that can recognize different characters (“H”, “E”, “L”, “O”, “W”, “R”, “D”). In the following sections, we will explore two different classification methods, namely k-Nearest Neighbours (kNN) and Support Vector Machine (SVM).

Description of the algorithm and flowchart

The labelled dataset contains images of seven different characters of interest (“H”, “E”, “L”, “O”, “W”, “R”, “D”). We first store all the image data using the MATLAB function *imageDatastore* with the necessary arguments to match the folder structure of the given dataset. Then, the dataset is partitioned into two using the MATLAB function *splitEachLabel*, whereby 75% of the images for each character form the training set and 25% the test set. A snippet of the MATLAB code is shown in Figure 55 below.



```
% store all images
dataset = imageDatastore(fullfile('p_dataset_26'),'IncludeSubfolders',true,'LabelSource','foldernames');

% use 75% of the dataset for training, 25% for testing
[train_set, test_set] = splitEachLabel(dataset,0.75);
```

Figure 55. MATLAB code to store images and split them for training and testing

Feature extraction is then performed on the images in both the training and test sets. There are several techniques commonly used to extract features, such as Histogram of Oriented Gradients (HOG), Speeded-Up Robust Features (SURF) and Local Binary Pattern (LBP). Specifically for this task, we found that the more suitable technique is HOG, and we use the MATLAB function *extractHOGFeatures* from the Computer Vision Toolbox.

The above steps prepare the images for the training and validation of the two classification methods under study, which will be presented separately below.

1. k-Nearest Neighbours (kNN)

In this study, the kNN algorithm will be briefly explained. The k-Nearest Neighbours (KNN) algorithm is an easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. Simply, the kNN algorithm holds the assumption that similar things exist in close proximity. The main steps in the kNN algorithm are depicted in Figure 56 below.

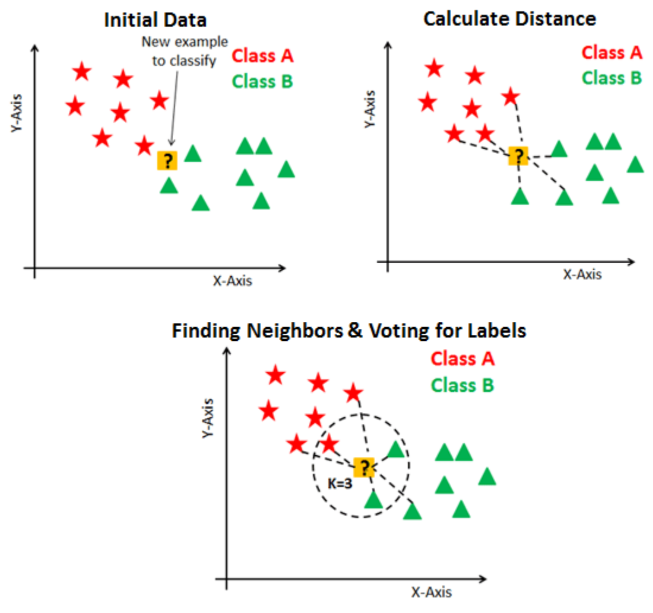


Figure 56. Main procedure of kNN algorithm

The calculation process of kNN algorithm is as follows: when a sample with unknown classification enters the data set, then the most similar (closest in feature space) k samples are which kind, so it is which kind. The main flow of kNN algorithm is as follows:

a) Establish a training sample set, each sample in the training set knows its classification, and specify the nearest neighbour number k;

b) After inputting a new sample without classification, each attribute of the new sample is compared with the corresponding attribute of the sample in the training set, and then the classification of the closest K samples is extracted;

c) The most frequent classification of these k samples is the classification of new data.

The algorithm flowchart is shown in Figure 57 below.

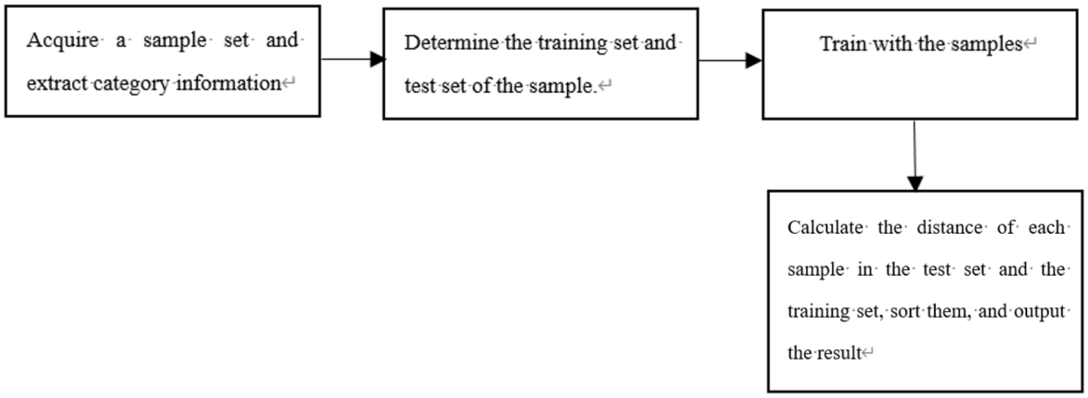

Acquire a sample set and extract category information
Determine the training set and test set of the sample.
Train with the samples
Calculate the distance of each sample in the test set and the training set, sort them, and output the result

Figure 57. kNN algorithm flowchart

2. Support Vector Machine (SVM)

The SVM method separates data of one class from those of the other class(es) by finding the best separating hyperplane with the largest margin. This hyperplane can be visualized in Figure 58 below, where the support vectors of each class closest to the hyperplane are also shown.

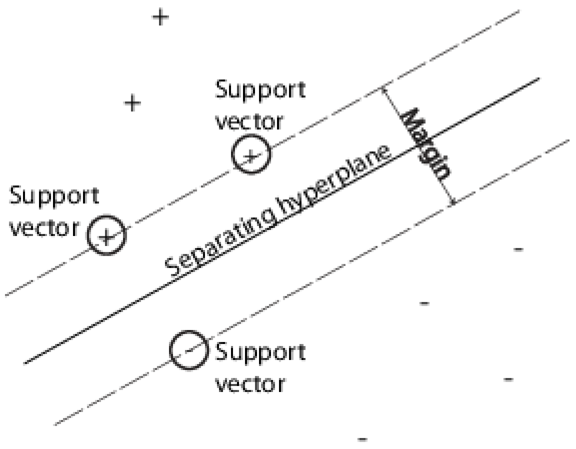


Figure 58. SVM for binary classification

Since character classification is a multiclass classification problem, it needs to be reduced to binary classification subproblems in order to utilize the SVM classification method. To do this, we apply the *fitcecoc* MATLAB function from the Statistics and Machine Learning Toolbox which internally uses one SVM for each binary subproblem. After training the multiclass SVM model on the training set data, the resulting SVM classifier is used to predict the labels of the images in the test set. A confusion matrix can then be plotted to measure the performance of the SVM classification model.

The training step can be repeated using different model hyperparameters to observe their effects on the prediction accuracy of the model. On top of that, different image pre-processing techniques such as padding and resizing can be applied on the input images to test their effects on the prediction results. The results of these experiments will be presented in the next section 2.8 Pre-processing & Hyperparameter Optimization.

In summary, the high-level algorithm flowchart for image classification is as shown in Figure 59 below.

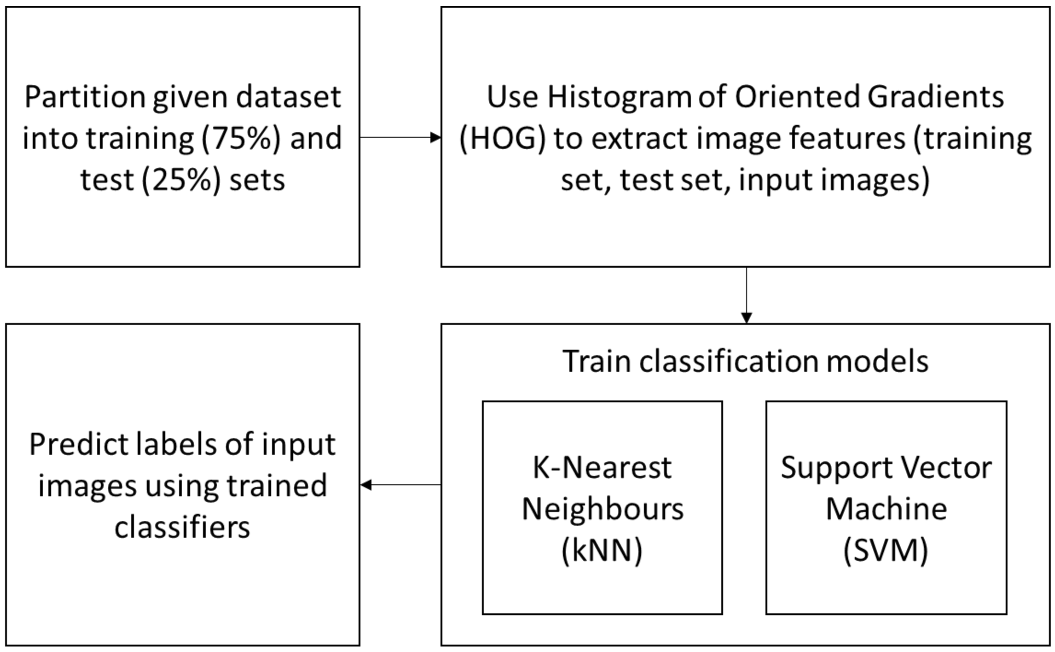

Partition given dataset into training (75%) and test (25%) sets
Use Histogram of Oriented Gradients (HOG) to extract image features (training set, test set, input images)
Train classification models
K-Nearest Neighbours (kNN)
Support Vector Machine (SVM)
Predict labels of input images using trained classifiers

Figure 59. Algorithm flowchart for image classification

2.8 Pre-processing & Hyperparameter Optimization

KNN Model Hyperparameter Optimization

In this study of pre-processing and hyperparameter optimization for the KNN classifier, we will explore a total of three different parameters that can be adjusted and optimized to maximize the accuracy of the classifier, which is the most important factor in our study. The three parameters are namely the Histogram of Oriented Gradients (HOG) feature cell size, the number of nearest neighbours, and the distance metric.

HOG feature cell size

As mentioned in section 2.7 Image Classification, we used the HOG feature extraction technique to extract features for the training of our KNN and SVM classifiers. The HOG feature cell size refers to the size of the HOG cell, which is specified in pixels in the form of a 2-element vector. The rule of thumb in choosing the optimal HOG feature cell size is to increase the cell size if the main objective of the study is to capture large-scale spatial information. However, the trade-off is that small-scale details may be lost when the cell size is increased.

In this optimization study of the HOG feature cell size, we explore the effect of implementing 8x8, 16x16 and 32x32 cell sizes on the accuracy of the KNN classifier. The HOG features of the aforementioned cell sizes are displayed below in Figure 60.

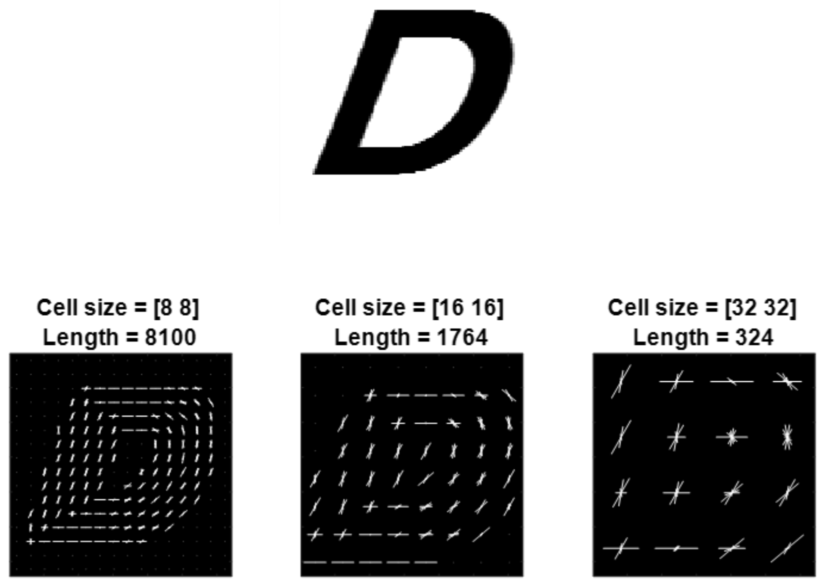


Figure 60. HOG Features of different cell sizes

As evident in Figure 60, as the cell size increases, the dimensionality of the feature vector, as indicated by its length, decreases. We will explore whether this reduction in dimensionality of the feature vector affects the eventual accuracy of the KNN classifier.

The classifier accuracy against the number of neighbours plots for different HOG feature cell sizes are displayed below in Figure 61.

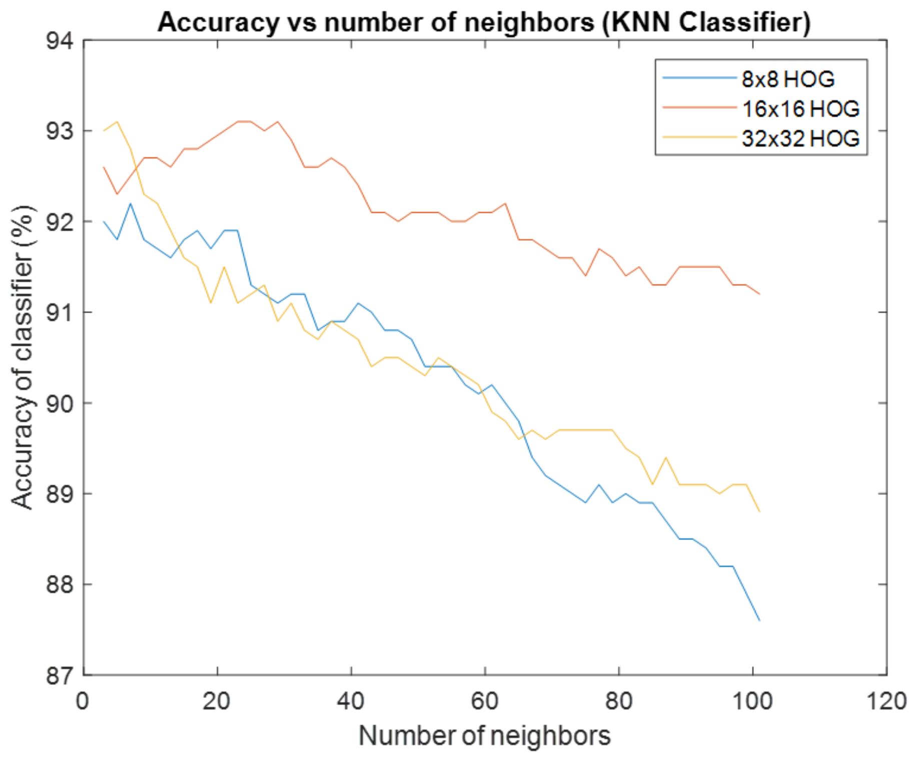


Figure 61. Effect of HOG feature cell size on accuracy of KNN classifier

As seen from Figure 61, it is apparent that the 16x16 HOG feature cell size gives the KNN classifier the best accuracy out of the three different cell sizes. However, it was initially expected that the 8x8 HOG feature cell size would perform the best in terms of accuracy, given that the 8x8 HOG cell size would be able to better capture smaller and finer details compared to the 16x16 HOG cell size. The main reason why 16x16 HOG cell size has performed the best is because it strikes a good balance between keeping the small-scale details and capturing the large-scale spatial information. In classification of alphabets or digits, both small and large-scale information are necessary to ensure that similar looking alphabets/digits are not misrepresented during prediction.

The run time against the number of neighbours plots for different HOG feature cell sizes are illustrated below in Figure 62.

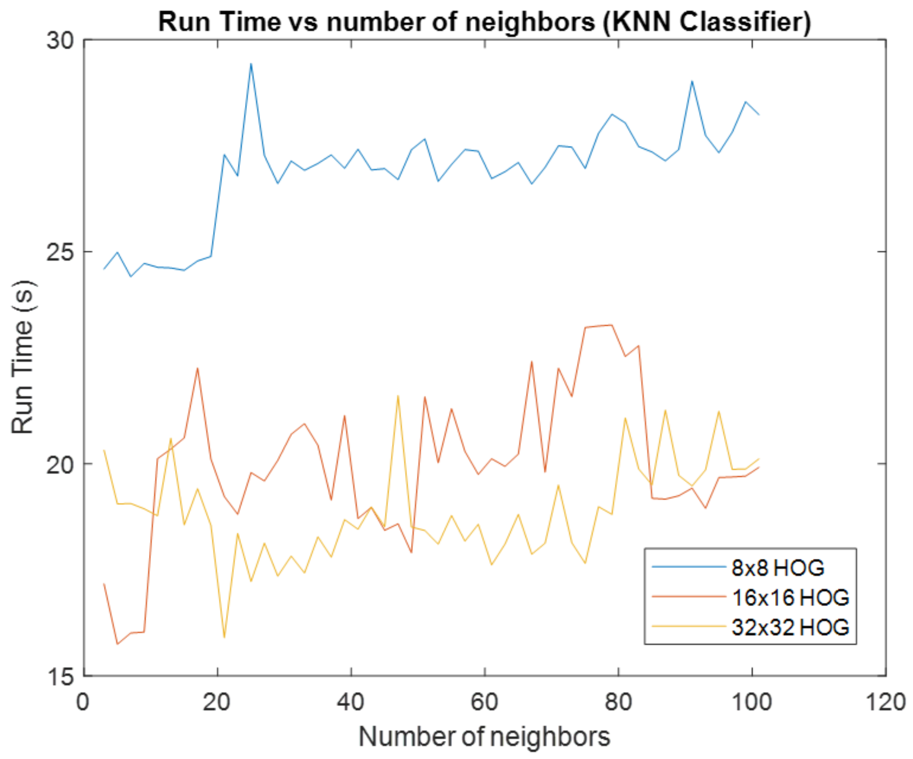


Figure 62. Effect of HOG feature cell size on run time

Although runtime is not the most important factor in our study, it is still necessary to find out how the different HOG feature cell sizes can affect the total run time of the program, which is essential in further studies which prioritise program speed as a factor of paramount importance. From Figure 62, it is evident that increasing the HOG feature cell size generally leads to a reduction in the total run time. This agrees with the theory that as the cell size decreases, the dimensionality of the feature vector increases, which results in generally longer training time of the classifier.

Therefore, the 16x16 HOG feature cell size is considered the optimal cell size for our study, as it attains the highest classifier accuracy overall, and achieves a reasonable run time.

Number of nearest Neighbours

In KNN classification, K indicates the number of nearest neighbours, which is considered as the most crucial determinant in the algorithm. K is typically chosen to be an odd number, to avoid situations whereby the classifier is unable to predict which class an object belongs to. If K is an even number, there could be a difficult situation whereby the classifier predicts that the object belongs to class A with a likelihood of 50%, but also belongs to class B with the same likelihood of 50%. Research has shown that there is no optimal value of K that suits all kinds of datasets well. This hyperparameter should be chosen to optimize the model during the time of model building through trial and error.

The classifier accuracy against the number of neighbours plot for the 16x16 HOG feature cell size is displayed below in Figure 63.

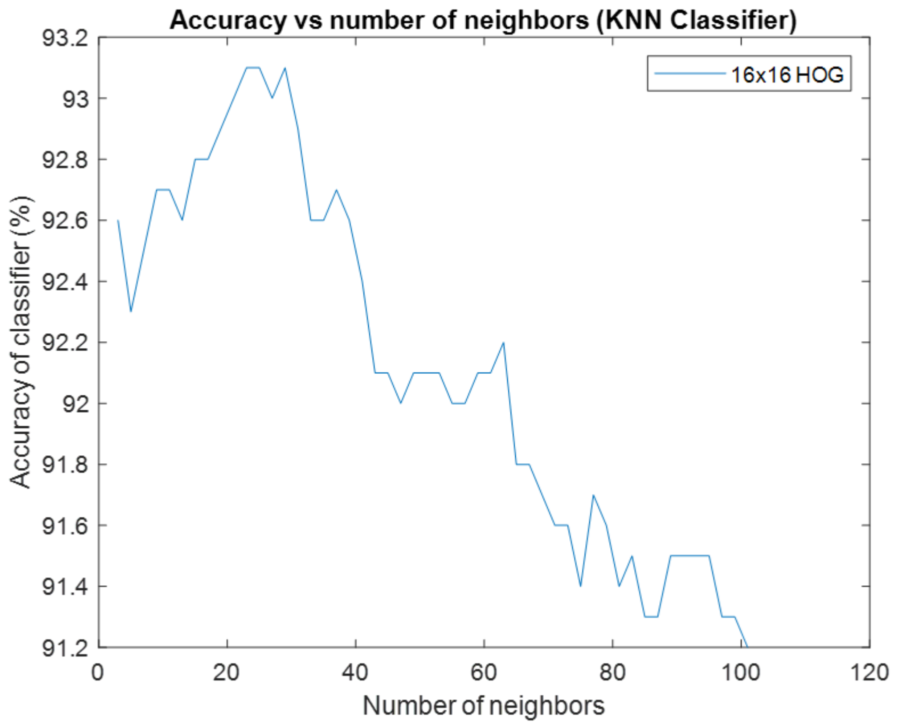


Figure 63. Effect of number of neighbours on classifier accuracy

From Figure 63, it is evident that generally, as the number of nearest neighbours increases, the accuracy of the classifier reduces. However, from the plot itself, we can see that there is a peak in the classifier accuracy at approximately 93.1% when the number of nearest neighbours ranges from 23 to 29. Therefore, the optimal value of K is concluded to be within the range of 23 to 29, for our study.

In addition, referring to Figure 61, it is also evident that regardless of the HOG feature cell size, the KNN classifier accuracy tends to decrease when the number of nearest neighbours increases. However, this trend is only specific to our data set and hence does not signify that one should always select a low value of K to optimize their model. The K hyperparameter should be selected based on trial and error. One useful tip to note is that generally, a smaller number of neighbours are the most flexible fit, which will result in high variance and low bias. On the other hand, a large number of neighbours will tend to give a smoother decision boundary, which results in lower variance but higher bias.

Distance metric

In KNN classification, one of the most important steps in the classification procedure is to find the K nearest neighbours to the data point to be classified. The distance metric is used to define the distance between the data points to the nearest neighbours. This illustration of the definition of distance metric is displayed below in Figure 64.

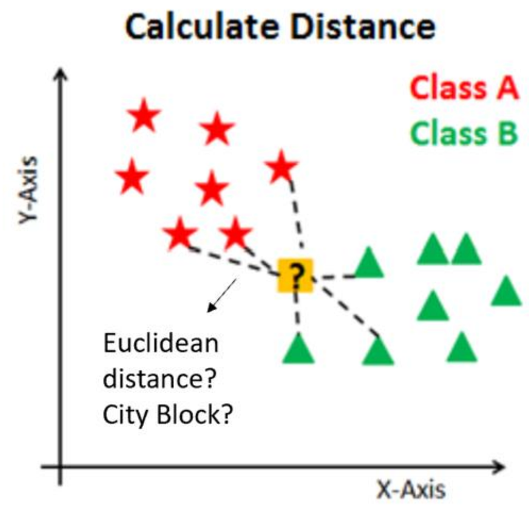


Figure 64. Distance metric in KNN classification

In this study, we explore a total of three different distance metrics to investigate how it affects the accuracy of the KNN classifier. These three distance metrics are namely, Euclidean distance, City Block distance, and Chebyshev distance.

Euclidean distance is commonly used in coordinate geometry and is defined as the distance between two points. To derive the two points on a plane, the length of a segment joining the two points is measured using the Pythagoras theorem. Fundamentally, Euclidean distance is the straight-line distance between two points on a plane. Assuming that (x_{-1}, y_1) and (x_2, y_2) are two points in a two-dimensional plane, the Euclidean distance between them is as follows:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

City Block distance, also known as Manhattan distance, is calculated as the sum of the distance in x and the distance in y. The City Block distance is always greater than or equal to zero, and it is zero for identical points and holds a high value for points that are dissimilar. Although City Block distance is largely similar to Euclidean distance, one difference is that with City Block distance, the effect of a large difference in a single dimension (x or y) is dampened, as the distances are not squared. Assuming that (x_A, y_A) and (x_B, y_B) are two points on a plane, the City Block distance between them is as follows:

$$dist(A, B) = |x_A - x_B| + |y_A - y_B|$$

Chebyshev distance is simply known as the maximum coordinate difference between two points on a plane. To further explain, it is a metric defined on a vector space where the distance between those two vectors is the greatest of their differences along any coordinate dimension x, y, or z. On a two-dimensional plane, assuming that (x_{-1}, y_1) and (x_2, y_2) are two points on that plane, the Chebyshev between them is as follows:

$$D_{\text{Chebyshev}} = \max\left(|x_2 - x_1|, |y_2 - y_1|\right)$$

The classifier accuracy against the number of neighbours plots for different distance metrics are displayed below in Figure 65.

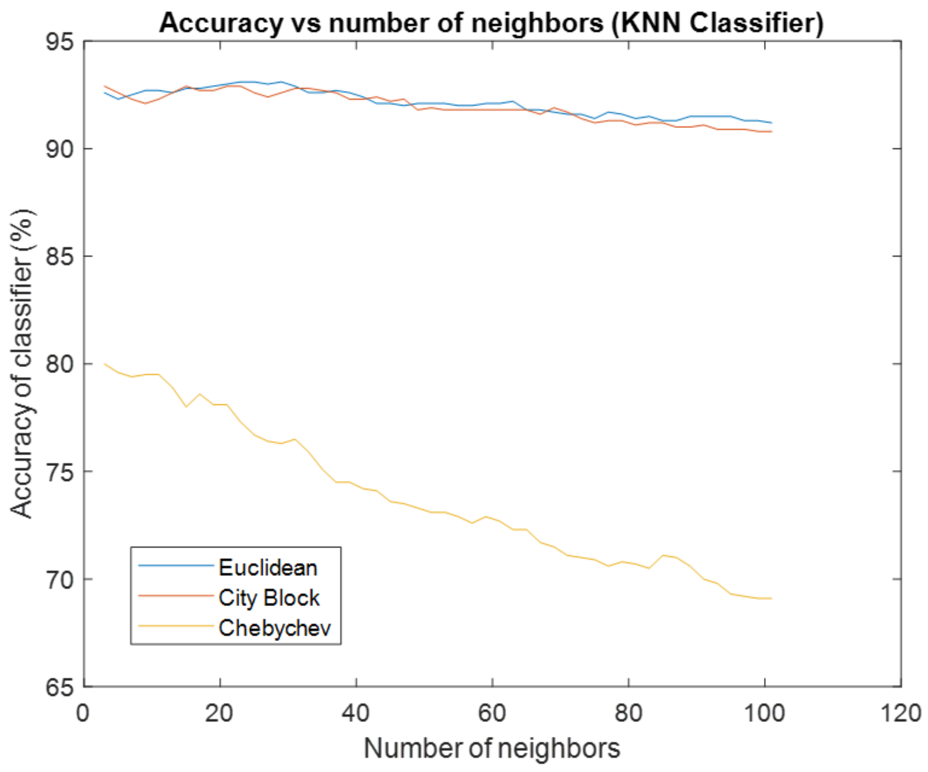


Figure 65. Effect of distance metric on classifier accuracy

From Figure 65, it is evident that using the Euclidean distance metric for KNN classification results in the highest accuracy compared to using the City Block and Chebyshev distance metrics. Like the K hyperparameter as previously discussed, there is no optimal distance metric that fits all types of data sets perfectly. Even though Euclidean distance is the most widely used distance metric in KNN classification, it does not necessarily mean that it is the most suitable or optimal distance metric. The distance metric hyperparameter should be selected based on trial and error during the process of model building. In our study, we find that the Euclidean distance metric is the optimal distance metric.

SVM Model Hyperparameter Optimization and Image Pre-processing

For the SVM classifier, we will attempt to tune two different parameters to maximize the accuracy of the classifier. The two parameters are namely the multiclass method and the kernel function. Besides, we will also experiment the effects of padding the input images on the prediction results.

Multiclass Method

There are mainly two methods to reduce the multiclass classification problem to a set of binary subproblems – "one-vs-one" whereby one learner is trained for each pair of classes, and "one-vs-all" where only one learner is trained for each class. A comparison of the accuracies using the different methods can be visualized in Figure 66 below.

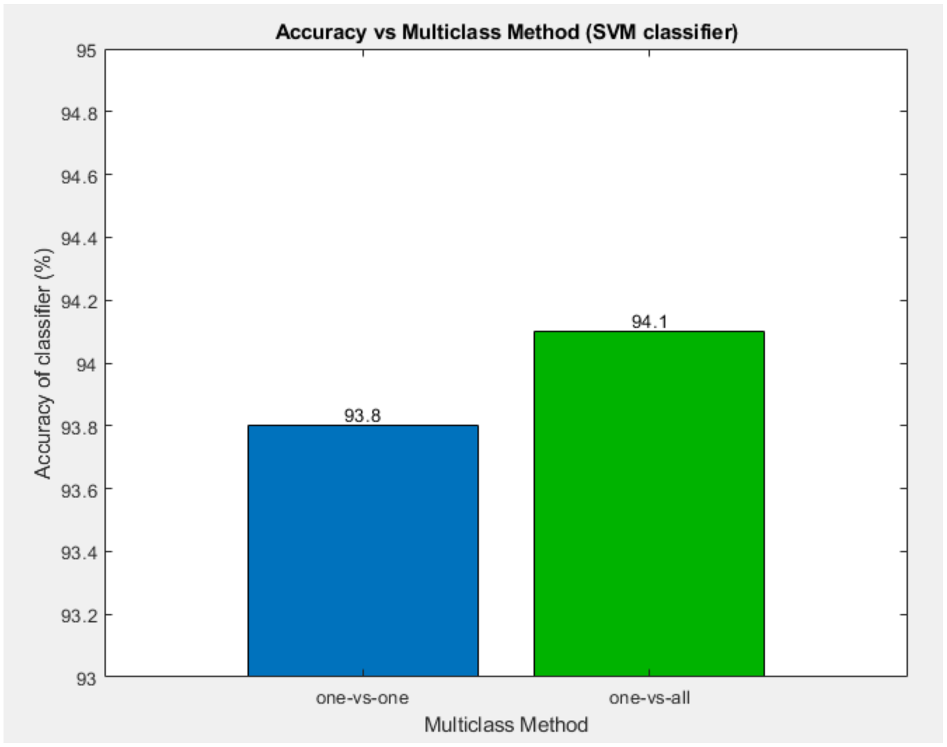


Figure 66. Comparison of classifier accuracies using different multiclass methods

Using the classifiers to predict the labels of the segmented images obtained from Section 2.6 Image Labelling and Segmentation, all 10 characters were correctly classified as shown in Figure 67 below.

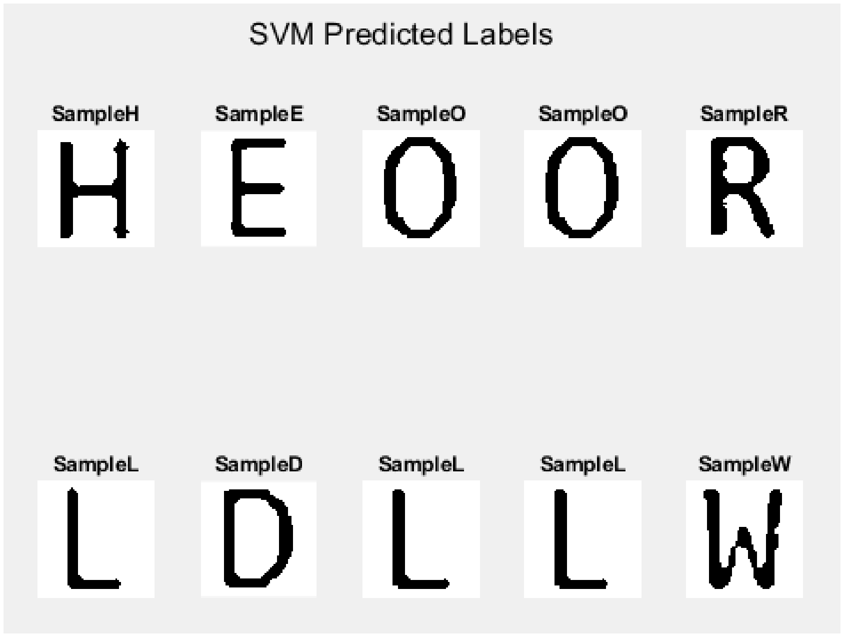


Figure 67. Predicted labels of the input images

As such, the "one-vs-all" method is preferred since it has higher accuracy and also requires fewer learners, consuming less memory and speeding up training time.

<u>Kernel Function</u>

Using the "one-vs-all" encoding scheme, we tested five different kernel functions – linear, gaussian, as well as 2nd, 3rd and 4th-order polynomials. Additionally, we also recorded the time taken to train the SVM model for each kernel function. Figure 68 below shows the classifier accuracies and run times for the different kernel functions.

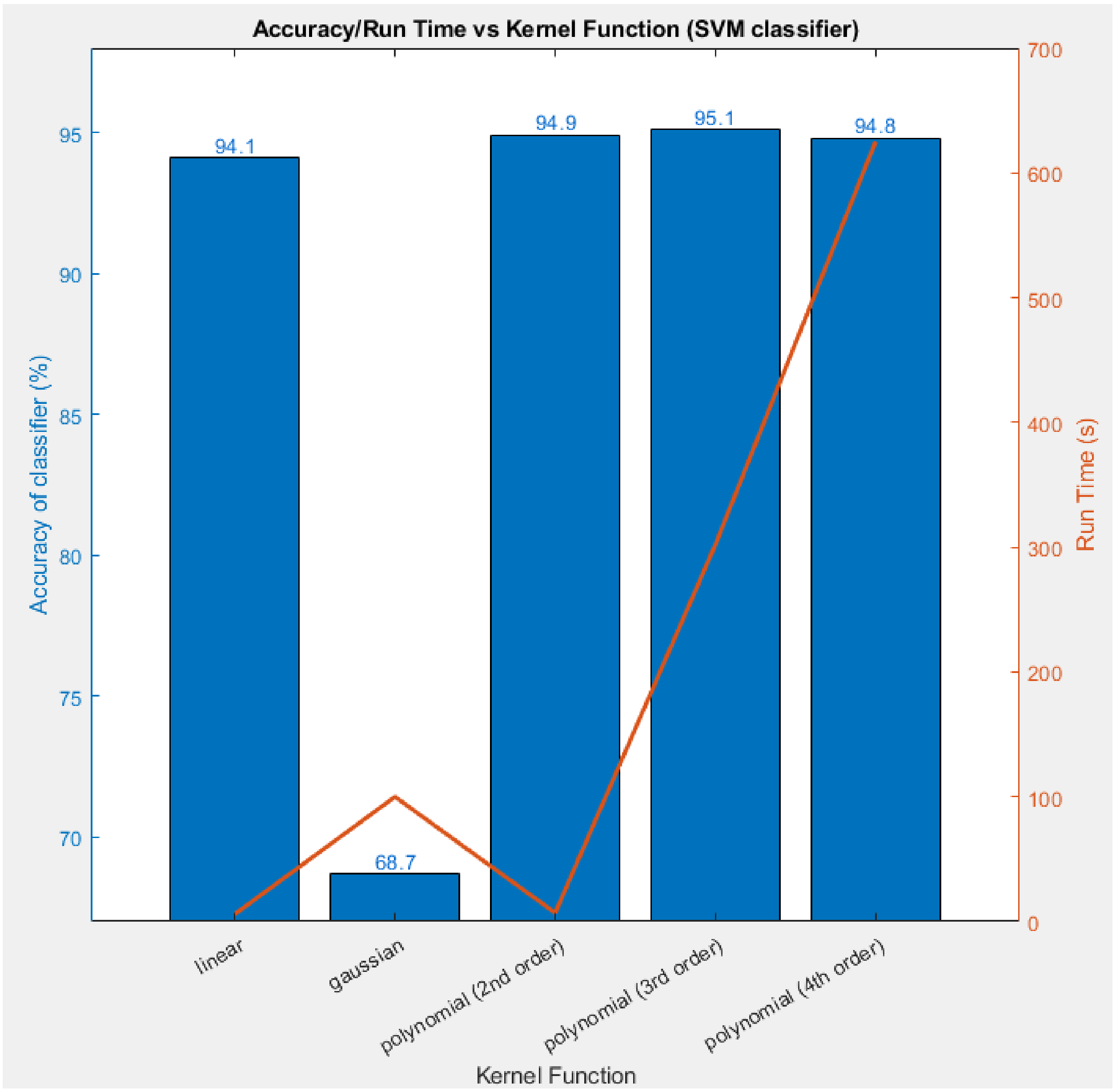


Figure 68. Comparison of classifier accuracies and run times using different kernel functions

From Figure 68 above, it can be observed that the “linear” and “2nd-order polynomial” are the optimal functions to use to train the SVM model for our character classification task. The run time using the “linear” function is approximately 6 seconds, achieving an accuracy of 94.1%. On the other hand, the “2nd-order polynomial” function uses about 7 seconds for training, hitting an accuracy of 94.9%. Increasing the order of the polynomial does not significantly improve the model accuracy, but the training time increases almost linearly with the polynomial order.

Padding input images

Using the “one-vs-all” multiclass method and the “linear” kernel function SVM classifier, we experimented with different padding sizes for the input images to determine the approximate range that the classifier can correctly predict the labels. In running these experiments, the segmentation algorithm described in Section 2.6 Image Labelling and Segmentation is used. The desired segment size can be inputted to the segmentation algorithm, then resized to 128-by-128 pixels, matching the image size of the training samples. The following figures show the prediction results for different segment sizes.

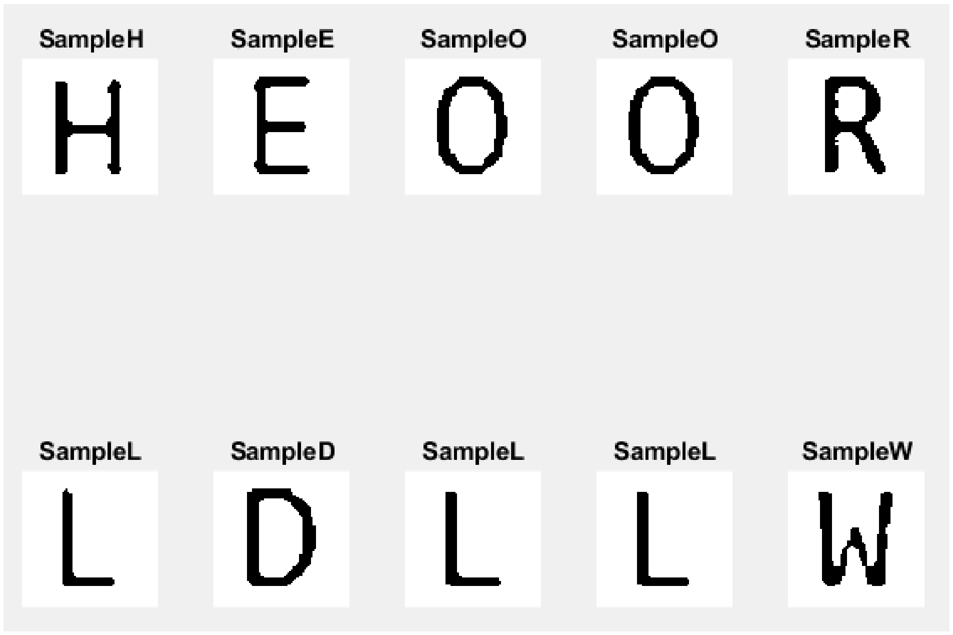


Figure 69. Prediction result after resizing a 60-by-60 pixels segment

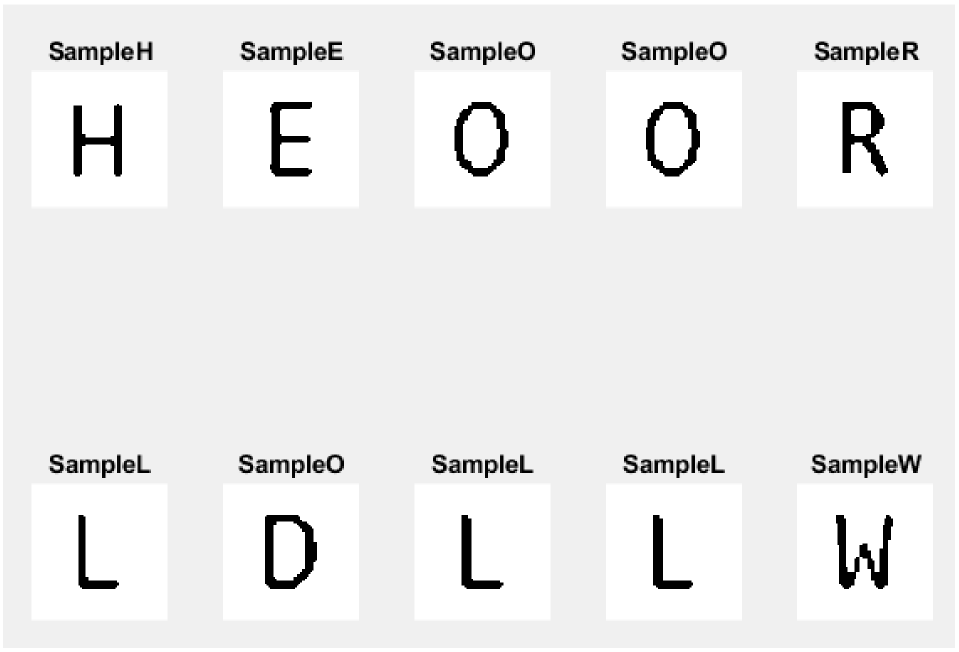


Figure 70. Prediction result after resizing a 80-by-80 pixels segment

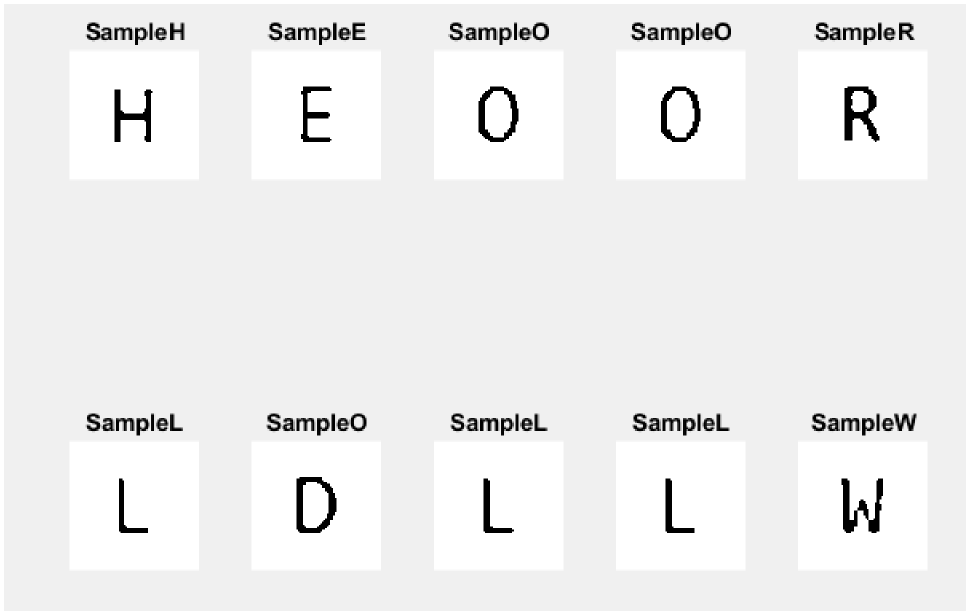


Figure 71. Prediction result after resizing a 100-by-100 pixels segment

From Figure 70 and Figure 71, the character "D" has been wrongly labelled as "O" since they are quite similar.

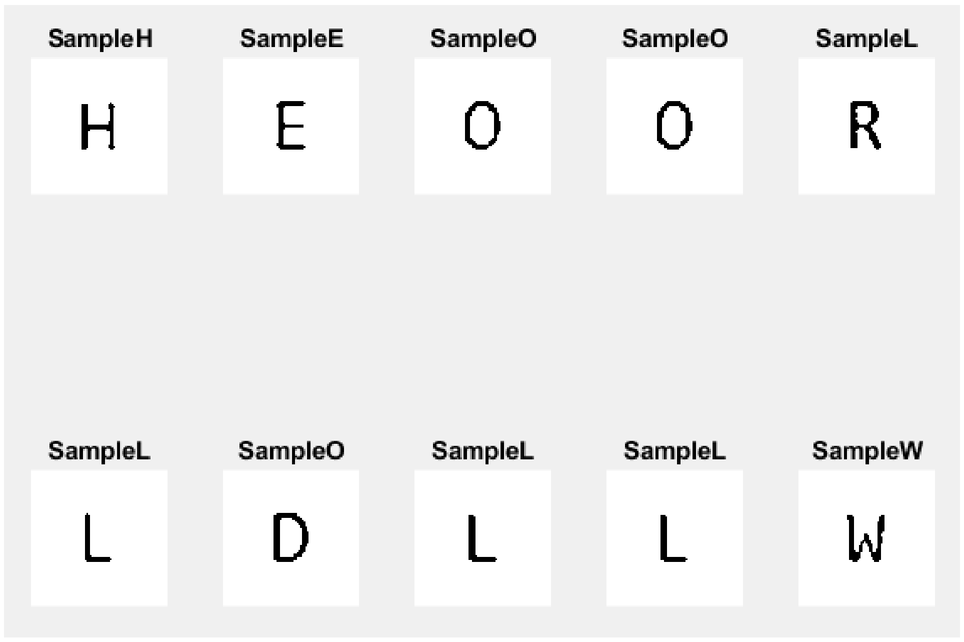


Figure 72. Prediction result after resizing a 120-by-120 pixels segment

From Figure 72, on top of “D” being wrongly labelled, the character “R” has also been wrongly predicted as “L”. As such, we conclude that the suitable range for the segment size to be inputted to the segmentation algorithm is between 50-by-50 pixels to 60-by-60 pixels in order to achieve optimal accuracy for this particular SVM model.