**Optical Character Recognizer**

**Introduction**

Optical character recognition (OCR) is the process of transforming image based text character from a scanned document or a photo of a document into active editable text character. Early OCR methods may be traced to as early as 1914, Emanuel Goldberg developed a machine that read characters and convert them into standard telegraph code.

In this program, I applied the Edge histogram descriptor introduced by Benny Thornberg of Mid Sweden University as a feature descriptor for describing each alphabetical character. In addition, I also implemented a secondary and tertiary feature for the sake of improving accuracy. Ultimately, the accuracy of the program has been examined to be within the range of 85~99% depending on the font of the input character.

The algorithm of OCR can be simplified as following:

1. Image preprocessing (Noise removal, RGB to Binary Image conversion, etc.)
2. Image labelling (Identify each character in an image as blob and extract these blobs)
3. Feature extraction

* Edge histogram descriptor
* Average Pixel Bin calculation
* Horizontal and Vertical scan line method

1. Classification

* Minimum distance classifier
* Weight consideration (Consider how much each feature should weight in the classification process)

1. Post-processing

* Sort each blob and locate their position in output document.

In contrast, the efficiency of the program is relatively slow comparing to other OCR techniques, this is caused by many factors of the algorithm and will be explained later on.

**Image Labelling**

The image labelling algorithm I applied here is introduced by youtuber “The coding Train” in his video <https://www.youtube.com/watch?v=ce-2l2wRqO8>.

The algorithm is defined by the following pseudo code:

for each pixel **p** in **input\_image**:

if(**p.rgb** != 0): //if pixel is not white

for each **blob** **b** in **blob\_vector**:

if (**p** is within **b** or **p** is near **b**):

add **p** to **b**;

readjust **b.size**;

end

else:

create new **blob**;

add new **blob** to **blob\_vector**;

end

end

end

end

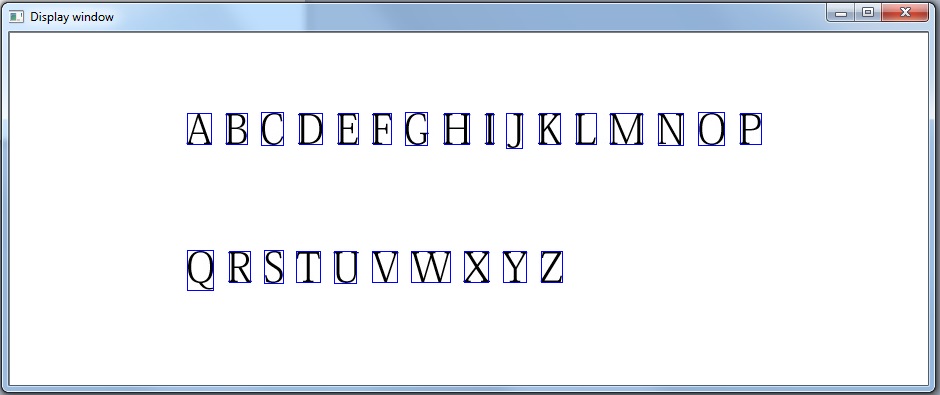
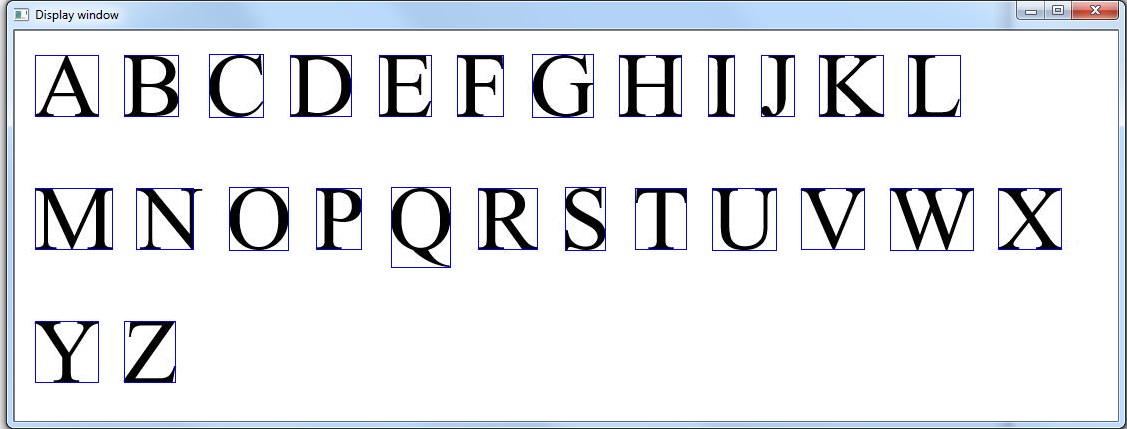


Figure . Results of applying the blob detection algorithm.

****

Figure

The pros and cons for this algorithm is straight forward, it is fairly fast and cheap in terms of calculation complexity, but it also requires a suitable distance threshold between each blob to be adjusted hence it may require user to input different distance threshold multiple times to figure out the correct threshold.

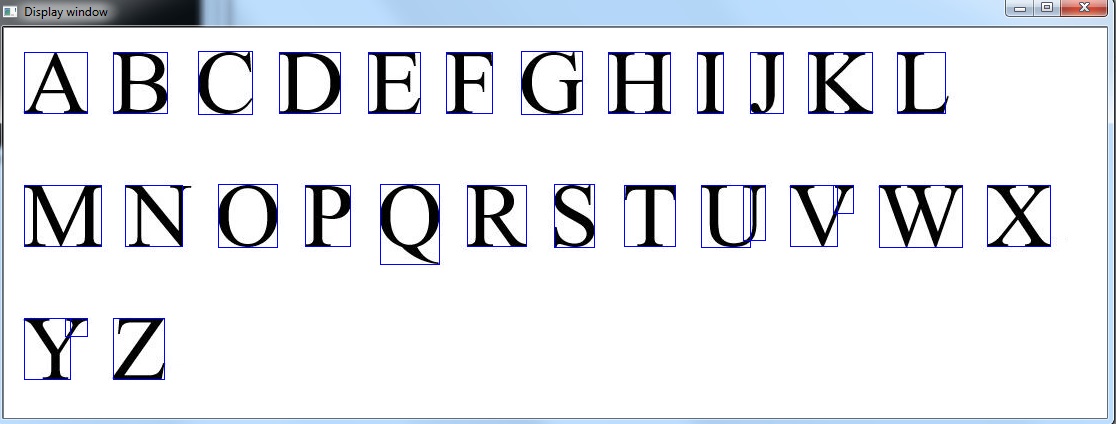


Figure Result of image labelling when distance threshold is not properly defined

To avoid this situation, I introduced a method to eliminate the small blobs, a method acquiring a second distance threshold, normally this threshold would be much smaller than the first threshold, so whenever the distance between two blobs is smaller than this threshold, it would merge the two blobs so the small blobs are eliminated.

This method is essential, because there are situations where the font of input characters is so large that the distance between parts of a character can exceed the spacing distance (the first threshold), so often small blobs are created.

**Edge Histogram Descriptor**

The EHD method is based on the fact that each character has unique gradient information and uses it as each character’s feature to distinguish the 26 alphabetical characters.

To build the edge histogram descriptor would first require calculating the gradient information of an image. In this case, I discovered that applying the Sobel operator of sized 13 produces optimal result, using any other operators or a smaller or larger kernel size would affect the accuracy of the program.

13x13 sized Sobel operator is a large kernel, since it denotes 337 floating point operations (169 multiplications and 168 additions) for each pixel in an image. This has contributed to the high calculation complexity of the program.

After calculating the pixel gradient information, pigeonhole each pixel gradient into 15 bins (360 degrees sectored into 15 bins, so each bin share 24 degrees). For instance, if the gradient of a pixel is between 0~24 degrees, it would be pigeonholed into bin 1, if its gradient is between 24~48 degrees then it would be placed into bin 2.

Finally, normalize the number of pixels in each bin into probability by dividing the total number of pixels.

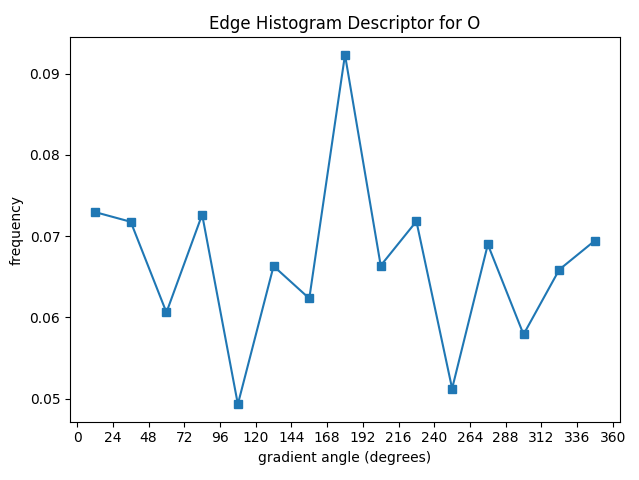


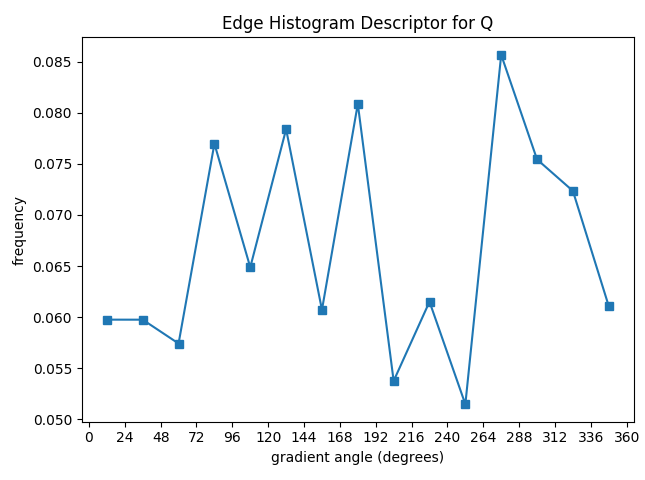
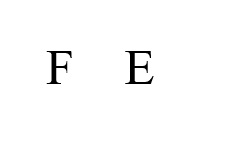
Figure Edge histogram of character ‘O

Figure Edge histogram of character ‘Q’

**Issue:**

The major issue with the EHD method is the algorithm is position invariant. When a feature is position invariant, two different characters with similar structure may be misclassified as the same character by this feature.

In the case of shown in figure 6, ‘F’ was misclassified as ‘E’ because they have a very similar character structure.

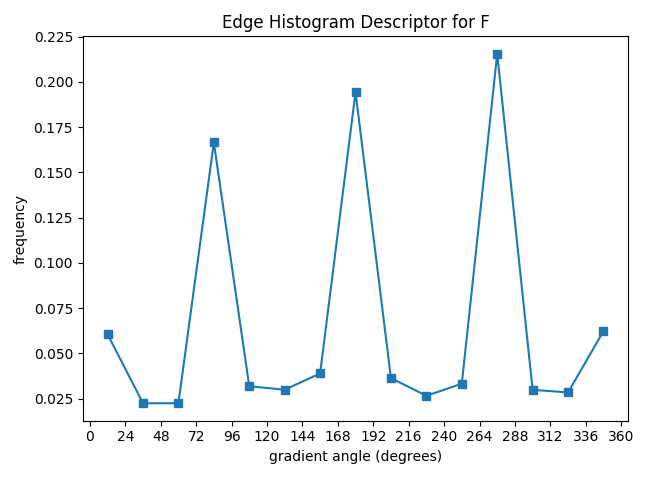


Figure 6 Character ‘F’ and ‘E’ in Times new roman font

Figure 6 EHD of character ‘F’

As a result, ‘A’ and ‘X’ formed a similar EHD and caused ‘A’ to be misclassified.

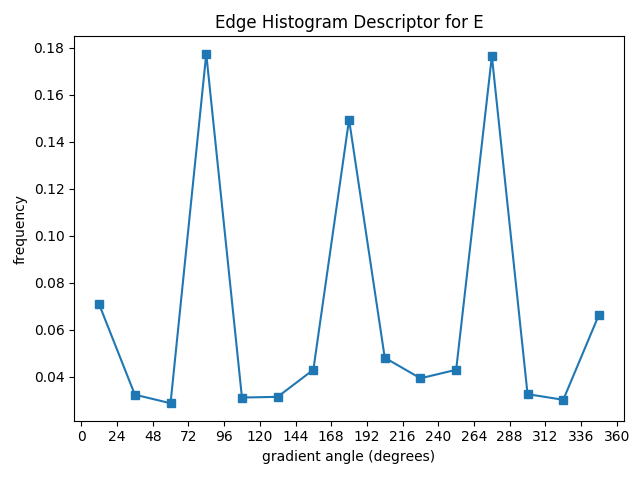


Figure EHD of character ‘E’

As a result, ‘F’ and ‘E’ formed a similar EHD and caused ‘F’ to be misclassified.

Similar scenarios also apply on the pairs (‘X’, ‘A’), (‘Y’, ‘V’), etc.

**The bounding box method**

To avoid the issue presented in EHD, a secondary feature is mandatory; Hence Benny Thornberg introduced the bounding box method.



Figure Bounding box applied to character ‘R’ with four bins

In addition to Benny’s method, I added on 4 extra bins placed in each of the four corners in the hopes of improving accuracy, thereby 8 bins are applied.

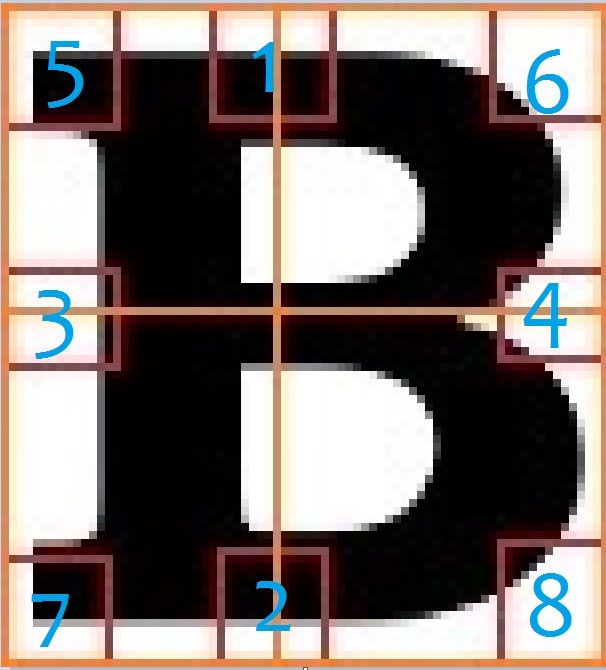


Figure Bounding box applied to ‘B’ with eight bins

First define the size of the boxes, in my program they are defined as width = 1/5 \* image.cols and height = 1/5 \* image.rows. Then determine the number of pixels that fall into each bounding box. So, eight new additional features are introduced to further distinguish each character.

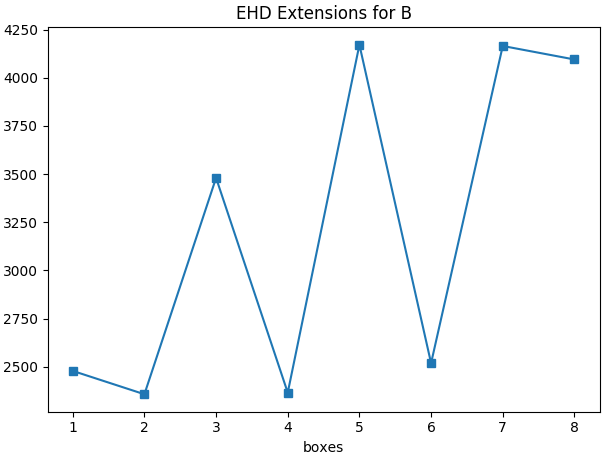
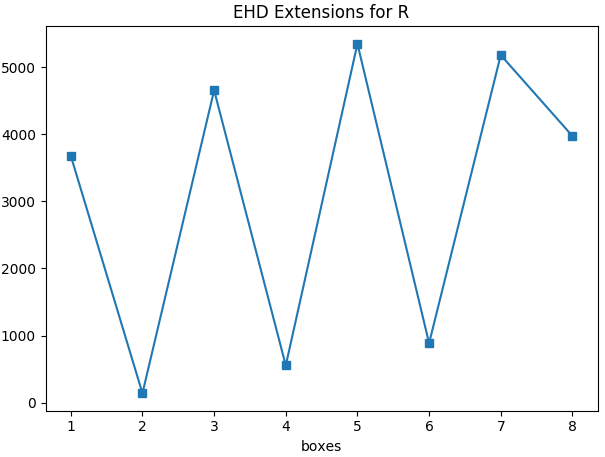


Figure 10 Average number of pixels fallen in each bounding box of ‘R’

Figure 11 Average number of pixels fallen in each bounding box of ‘B’

**Classification**

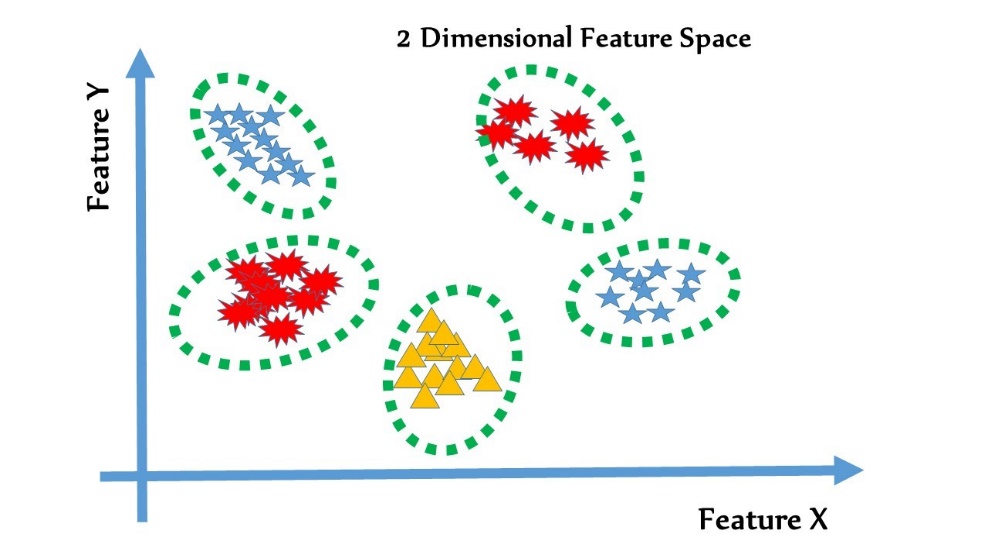


Figure five feature clusters in a 2-dimensional feature space

In the classification process, each feature is treated as a dimension in the feature space, therefore 15 bins in the EHD method creates a 15-dimensional feature space, in addition, the bounding box method introduces another 8-dimensional feature space, so a 23-dimensional feature space is formed.

Suppose there are **K** classes and **nk** number of samples in each class. To calculate the centroid of cluster **K** in feature space would require finding the mean of each feature **F** of class **K**.

j {1, 2, 3, …. , 23} // 23 features

k {1, 2, 3, ….. , 26} //26 alphabet characters

nk {1, 2, 3, …., n} // n number of samples

**Fj =**  \* ij(k)

After calculating the mean of each feature, we could compare each class’s centroid with the input text.

Let Xi be the value of input text’s ith feature.

Distance = 2  + 2

The first 15 features correspond to the 15 bins in EHD and the second 8 features correspond to the 8 bins in the bounding box method, where is an arbitrary value to normalize and determine the weight of the feature values in bounding box method.

Repeat this process for 26 times to calculate the Euclidean distance between each class’s centroid to input text in feature space, and then locate the cluster with the minimum distance to the input text. The class with the minimum distance to input text is likely to be the character of input text.

**Scan line method**

C:\Users\John\AppData\Local\Microsoft\Windows\INetCache\Content.Word\TemplateF_2.jpgA third feature was introduced by me to further improve the accuracy of text recognition. After examining the first two features of input text, some text might have been recognized as the incorrect character, but still have a high probability to be within the top three minimum distance. Using this scheme, we could select three candidates with minimum distance to perform a further analysis and hopefully be able to determine the correct candidate.

Figure 13 Visualization of applying the scan line method to ‘F’

This method will draw a horizontal and vertical line across the middle of input character with an arbitrary width for the line, in my program, I chose the width to be 10 pixels. The two lines will determine the number of pixels they encountered vertically and horizontally.

Each class contains a template which I refer to as the standard template. The standard template is used to compare with the input text character using this method, hence the standard template should be a more commonly used font, so I used Calibri in this case.

After determining the top three candidates, compare each of their standard template to the input text character. Since there are two lines, this provides two features, so the minimum distance classification method can be again applied in this case to perform classification.

Since this method is font dependent, so it is reasonable to put a weight variable to limit its weight on the final determination to avoid situations of misclassification caused by different fonts. In this program, a 0.3 weighted variable is applied.

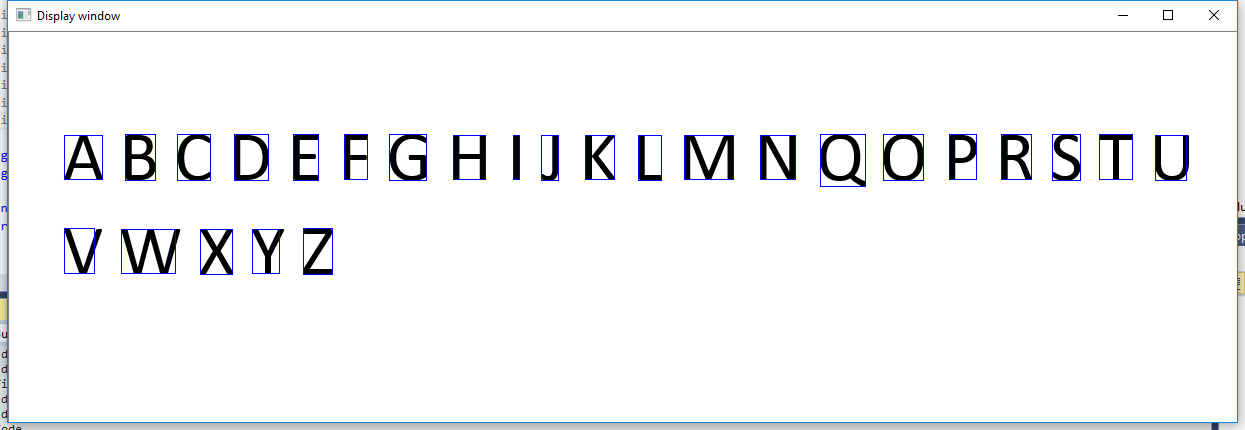
**Results**

Figure 14 Input Image of Calibri font text



Figure 15 Output of figure 15

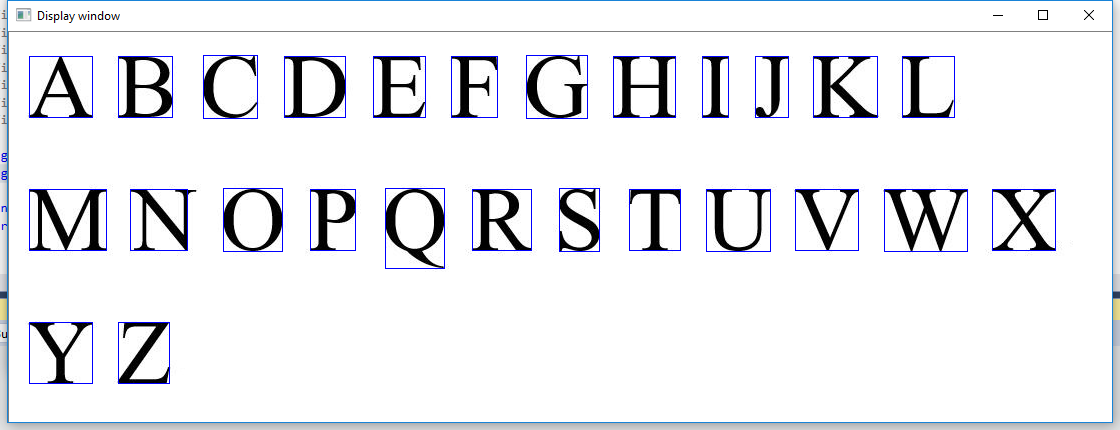
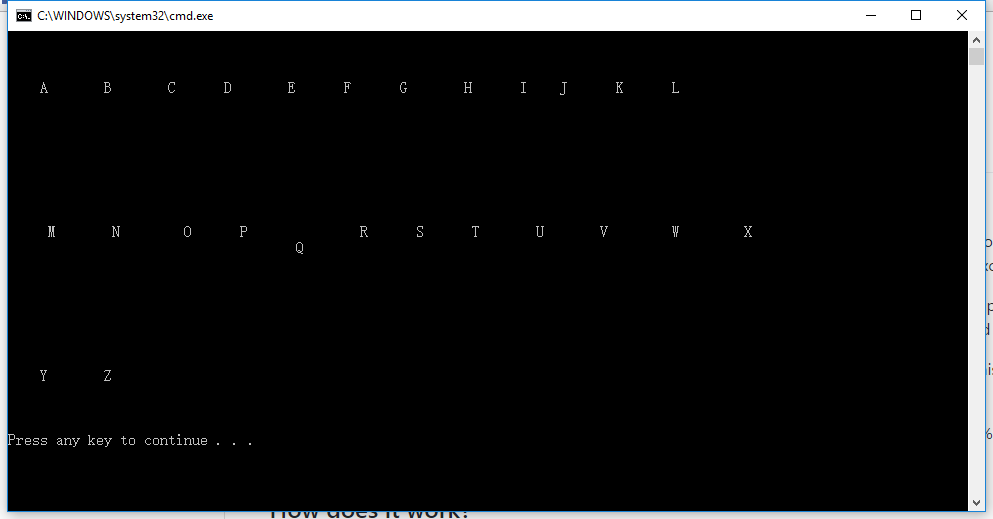


Figure Output of figure 17

Figure 17 Input image of Times new roman font text

**Reference**

Thornberg Benny, F.M. [[Benny Thörnberg](https://www.youtube.com/channel/UCJ51977zHWvVPpS-DODgMvA)] (2014, Jan, 20), *Lecture in Machine Vision: OCR - Original Character Recognition.* Retrieved from <https://www.youtube.com/watch?v=yPlwMUy2y2U>

F.M. [The coding train] (2016, Jul, 7), *11.7: Computer Vision: Blob Detection - Processing Tutorial.* Retrieved from <https://www.youtube.com/watch?v=ce-2l2wRqO8&t=728s>