Final Project:
Optical Braille Recognition

Team: ICU **Instructor:** Dr. Zhang

Derek Nguyen	Garth Leung	Jayze Crawford
dnguy121@calpoly.edu	gfleung@calpoly.edu	jcrawf@calpoly.edu

California Polytechnic, San Luis Obispo
Department of Electrical Engineering, College of Engineering
CPE 428 - Computer Vision

Abstract: Braille is a writing system used for the visually impaired, created by Louis Braille. Currently, there are various Optical Braille Recognition (OBR) systems and computer vision algorithms that allow people who work with Braille to digitize out of print Braille books. This final project explores various methods and algorithm for detecting one sided Grade 1 Braille recto dot. We segment the Braille image from the internet using Otsu Thresholding method. We then use connected component to rebuild the complete binary image of the Braille text. Braille cells are recognized using the traditional grid method and the standard Braille cell dimensions. Our new step of recreating the binary image of Braille gives an alternative way for pre-processing Braille images. The project's result shows the possible 100% Braille cell recognition on single sided Braille text images but have not been tested on different real-life sample due to limited Braille resources at Cal Poly.

Keywords: Recto dot, single sided Braille text, connected component. intersection, Braille cell, digitize.

I. Introduction

A. About Braille

Braille is a writing system used for the visually impaired, created by Louis Braille [1]. It is typically written on embossed writing paper. Braille characters are employed through rectangular blocks named cells that have tactile bumps called recto dots. The arrangement and quantity of these raised dots will specify differences from each other. For example, there are three levels of encoding for English Braille: Grade 1 - a direct letter translation to the English alphabet, Grade 2 - associated with abbreviations and contractions and Grade 3 - a non-standardized personal stenography [2].

Each Braille cell consists of six dots: two columns and three rows. The dots can be raised in various combination to make different characters. However, their dimensions are set to specific measurement for uniformity by the Library of Congress. The horizontal distance (HD) and vertical distance (VD) are the same at 2.5 mm. The distance between horizontal cells (DBH) is approximately 1.5 times HD and VD. The vertical distance between vertical cells is approximately 2 times HD and VD. The dimensions are shown in figure 1 [3].

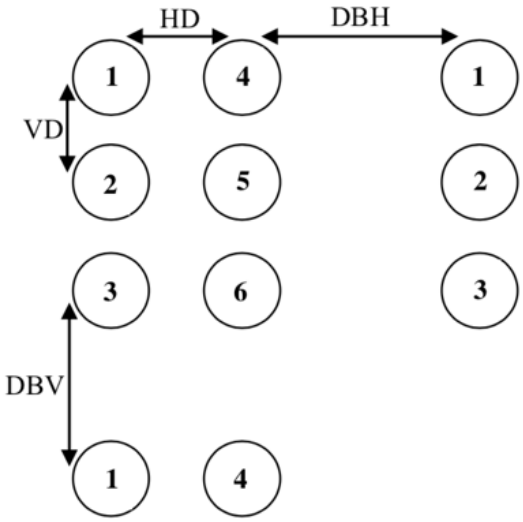

Fig. 1 Braille cell with measurement dimensions.

B. Motivation

As of 2012, the World Health Organization (W.H.O.) estimates that there were 39 million visually impaired people in the world. As this number has increased throughout the years, Braille has become disseminated throughout today's society, which can be seen on public service signs (e.g. restrooms and navigation signs). To assimilate the visually impaired into today's society, there are Braille textbooks, notebooks, and tablets as well. In order to live together with the visually impaired, both populations should be able to translate between each other. Therefore, an optical braille recognition and translator would be a great first step.

C. Problem Statement

Optical Braille Recognition (OCR) will deal with a three-part process in order to translate: (1.) Preprocessing images to obtain a binary image of braille, (2.) Detecting of the braille characters in cells of recto dots and (3.) Matching between templates of the Grade 1 Braille characters to translate the image into English Braille. This project will assume that all the photos given to the algorithm will only be in Grade 1 Braille as there will be a direct English alphabetic translation. Eventually, the next steps would be to include Grade 2 and Grade 3 Braille to become an all-encompassing Braille translator.

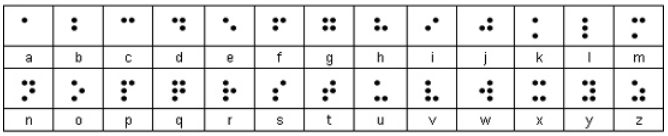

Fig. 2 Grade 1 English Braille alphabet.

An assumption in the Optical Braille Detection algorithm is that the images being fed into this process will only be digital images, which will eliminate lighting issues and rotation issues. The main focus of this project will deal with the detection and translation of the Braille characters. Furthermore, this project can be improved in the future by including images of textbooks or signs. However, Cal Poly's Kennedy Library did not have any physical Braille texts on hand for this experiment. Therefore, the scope of this project will deal with scanned images of Braille characters as indicated in Figure 3.

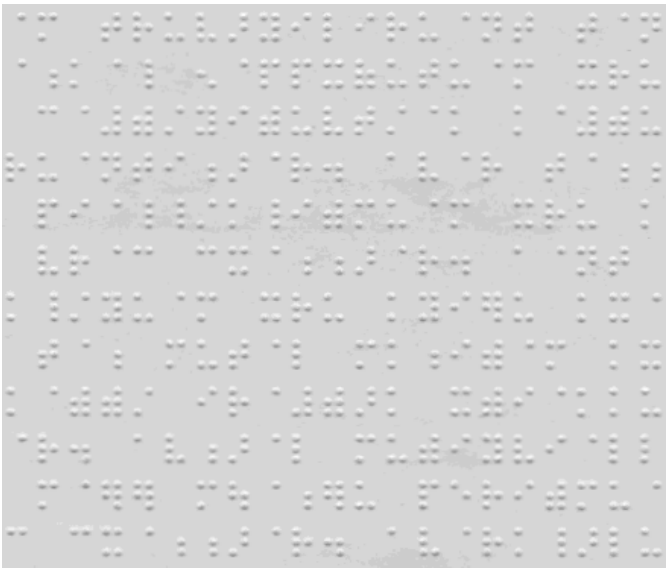

Fig. 3 A scanned of Braille text obtained from Google Image search.

II. Literature Review & Previous Work

A. Previous Work in Detection

"Smart Braille System Recognizer" by Aisha Mousa, Hazem Hiary, Raja Alomari, and Loai Alnemer published in the International Journal of Computer Science Issues, Vol. 10, Issue 6, No 1 discusses their system of Optical Braille Recognition

[3]. In their system, they are able to scan written Braille Grade 1 or Grade 2 document, and successfully recognize these Braille characters. Their system's dots' detection accuracy ranges between "94.39% up to 99.76%".

In order for their system to be fully functional, they break down the Optical Braille Detection process into six parts: Image acquisition, Image pre-processing, Image enhancement, Image segmentation, Feature Extraction, and Braille cells recognition.

During the image acquisition stage, they only assume the use of a single sided embossed Braille documents and use a flat-bed scanner. After obtaining a scanned image of the Braille Document, they enter the image pre-processing stage where they convert the image from the RGB color space to gray level images. Because the resulting image can contain frames around borders that disrupt the next image processing techniques, they perform standard image cropping. These two steps are indicated in the Figure 4 and 5 below shown in their report.

Fig. 4 Sample grayscale Braille Document

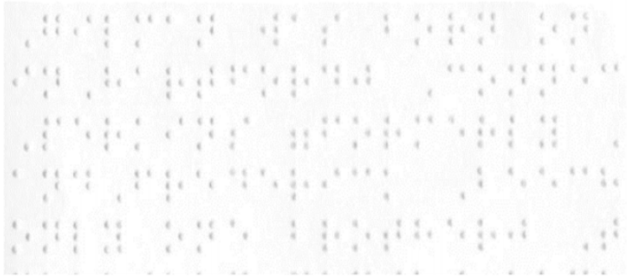

Fig. 5 Cropped Image.

The image enhancement stage will deal with highlighting specific image features such as the dots and their relative location. One image enhancement technique they employ deals with noise reduction. Because a scanned document results in random noise that is reflected throughout the image, they eliminate noise through an averaging filter. Because the image acquisition step can happen in various lighting conditions, they perform contrast enhancement to increase the intensity of the recto dots. The noise reduction and contrast enhancement techniques are shown in Figure 6.

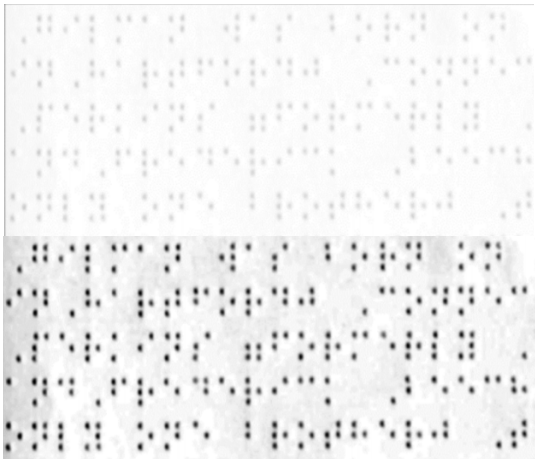

Fig. 6 Noise-Reduced Image (top) and Contrast Enhanced Image (bottom).

Entering the image segmentation stage, they separate the desired dots from the background by using an image compliment. After obtaining the image compliment, they perform a binarization step to obtain a binary image of white dots and black background. This step will pave the way for easier recognition of the Braille characters due to the elements of being valued 0 (white) or 1 (black), as shown in Figure 7 and 8.

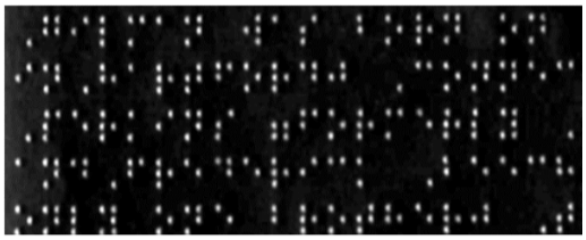

Fig. 7 Braille Image Compliment.

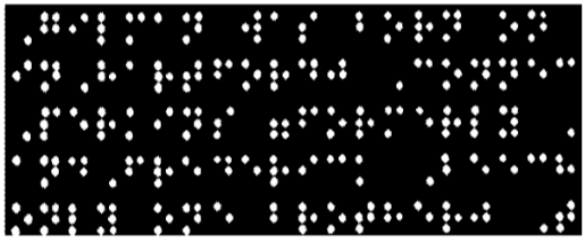

Fig. 8 Binarized Braille Image

In order to obtain the relevant knowledge from the Binary Braille image, they then enter the feature extraction step. This is a major step where they compute the centroids of the dots through the geometrical shape of the circles. This is done for each of the pixels within the dot. Then, they align the coordinates of the digitized page with their Braille coordinate system to protect against slight angle changes or rotation during the image acquisition step.

Finally, they enter the Braille cells recognition step where they aim to group the dots based on the location information from the feature extraction stage. This stage relies on the standard Braille measurements such as the distance between dots on neighboring cells is 3.75 mm or the distance between the centers of the dots in a single Braille cell is 2.5 mm. These measurements are only a few of the considerations when determining a Braille cell, which will lead to the recognition of these Braille characters as shown in Figure 9.

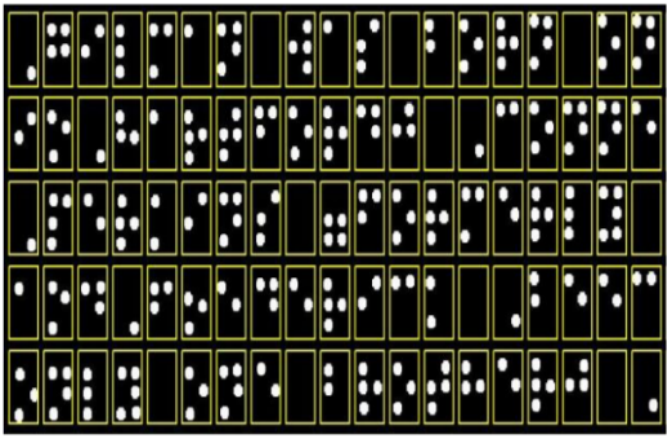

Fig. 9 Detected Braille cells.

B. Previous Work in Image Thresholding

"Dot Detection of Braille Images Using A Mixture of Beta Distributions" by Amany Al-Saleh, Ali El-Zaart and Abdul Malik Al-Salman was published in the Journal of Computer Science Volume 7, 2011. This study presents a new thresholding algorithm for scanned Braille images using Beta distribution.

In this study, the authors build a histogram of a grayscale Braille image (Fig. 10) and recognized the potential modes for the recto dots, verso dots, and background. [4]

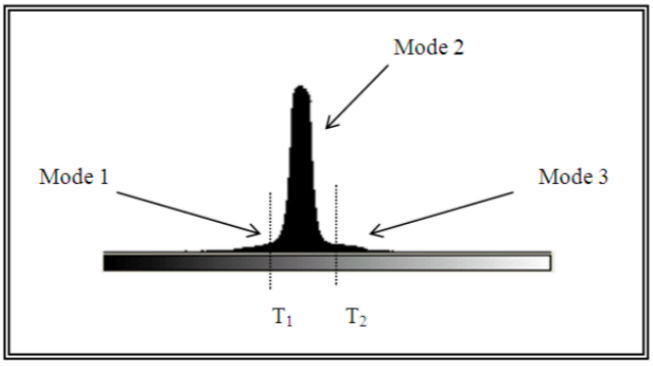

Fig 10. Histogram of a grayscale Braille Image [4].

From the histogram, three modes represent 3 classes of pixels:

- Mode 1: dark region of a recto and verso dot.
- Mode 2: background pixels.
- Mode 3: light region of a recto and verso dot.

From the proposed modes, 2 thresholds value can be deduced for thresholding. Thresholding to the threshold values will segment the image into 3 classes according to equation 1:

$$g(x,y) = \begin{cases} 0 & if \quad f(x,y) < T_1 \\ 100 & if \quad T_1 \le f(x,y) \le T_2 \\ 255 & if \quad f(x,y) > T_2 \end{cases} \qquad \text{(eq. 1)}$$

In this situation, having an assigned threshold might be problematic due to the different exposures images may have. The authors developed a new technique called Stability Thresholding with Beta distribution.

The process of Stability Thresholding will iterate through all modes of the histogram in order to estimate two thresholds [4]. The Beta parameters α and β, and the prior probability P of each mode are estimated using initials value of threshold calculated from equation 2 and 3:

$$T_1^0 = \frac{\sum_{i=0}^{Max\ Index}(i*h(i))}{\sum_{i=0}^{Max\ Index}(h(i))} \qquad \text{(eq. 2)}$$

$$T_2^0 = \frac{\sum_{i=Max\ Index}^{255}(i*h(i))}{\sum_{i=Max\ Index}^{255}(h(i))} \quad \text{(eq. 3)}$$

The prior probability P for each mode is estimated using the follow equation:

$$P_i = \frac{\sum_{j\in Mode\ i}\left(h(x_j)\right)}{\sum_{j=0}^{255}\left(h(x_j)\right)} \qquad for\ i = 1,2,3 \quad \text{(eq. 4)}$$

Using the estimated α, β and prior probability P, new threshold T^{New} can be calculated:

$$T_i^{New} = 1 - \exp\left(\frac{-A - B\log\left(T_1^0\right)}{C}\right) \quad \text{(eq. 5)}$$

Where:

- $A = \log\left(\frac{P_i K_i}{P_{i+1}K_{i+1}}\right)$ (eq. 6)
- $B = \alpha_i - a_{i+1}$ (eq. 7)
- $C = \beta_i - \beta_{i+1}$ (eq. 8)
- $K_i = \frac{\Gamma(\alpha_r - \beta_r)}{\Gamma(\alpha_{r+1} - \beta_{r+1})}, r = i, i+1$ (eq. 9)

The procedure is repeated until the error between the threshold values are zero using eq. 10 below. The technique of recalculating threshold for image segmentation is similar to that of Otsu Thresholding; a technique that we've learned in class.

$$\left|(error = (T_1^0 - T_1^{New}) - (T_2^0 - T_2^{New}))\right| \quad \text{(eq. 10)}$$

The results of this algorithm are two new thresholds value. The scanned Braille image will be segmented using the new threshold values.

The authors start with a scanned one-sided image as shown in Fig. 11. The result of stability thresholding using their technique is shown in Fig. 12. The initial values and final value for the calculated threshold values can be seen in Table 1.

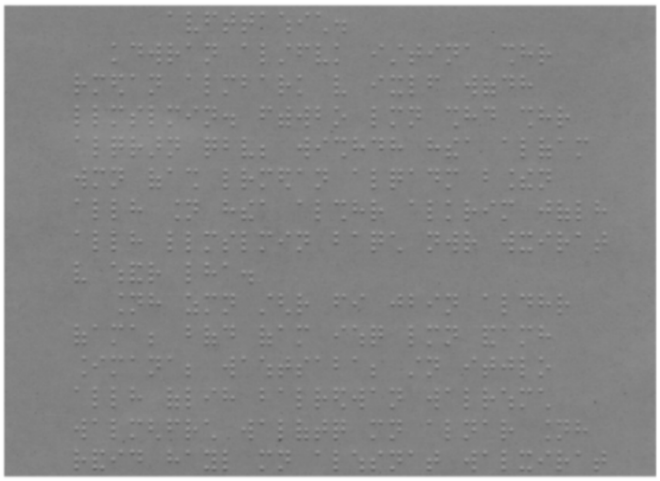

Fig. 11 A single-sided scanned Braille image. [4]

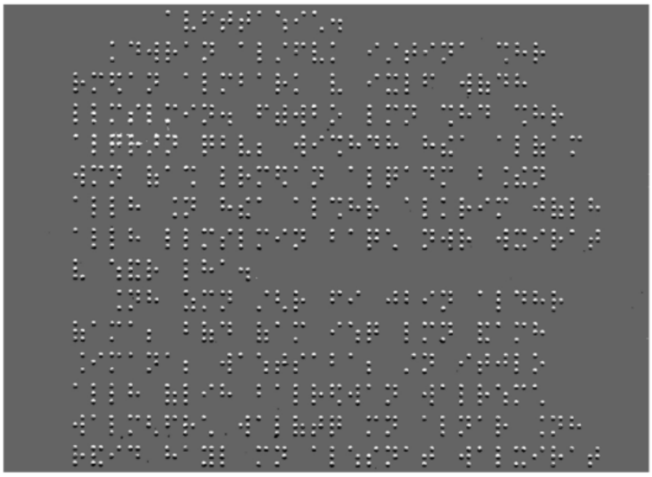

Fig. 12 The segmented image of Fig. 11 [4]

Table 1. The initial and estimated threshold values for figure 12. [4]

	T1	T2
Initial Value	126	135
Estimated Value	119	142

The result of this study shows a new algorithm for image thresholding and dot detection. Their method shows the ability to eliminate the shadows cast by the recto dot at the time of scanning.

C. Previous Work in Grid Formation

In "*An Efficient Braille Cells Recognition"* by AbdulMalik S. Al-Salman, Ali El-Zaart, Yousef Al-Suhaibani, Khaled Al-Hokail, AbdulAziz O. Al-Qabbany [5] and the previously mentioned study above, a grid system was implemented to extract individual Braille cell. A grid system is formed by generating pairs of horizontal lines and vertical lines that hug the segmented recto dots. Figure 13 shows a grid section from a cropped image. Figure 14, in turn, shows the full grid formation for a scanned Braille text.

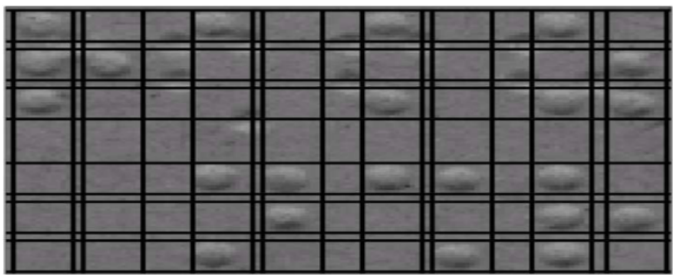

Fig. 13 A sample grid section formed by pairs of horizontal and vertical lines hugging recto dots.

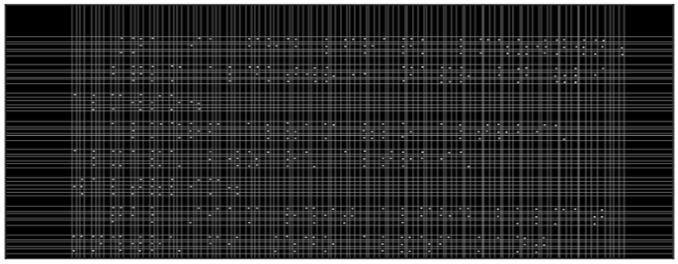

Fig 14. A grid system visualization for a full scanned Braille text.

Using the created grid system, the Braille cell can be recognized with the standard Braille cell spacing measurement as shown in Figure 1. The result is an efficient detection of Braille cell extraction.

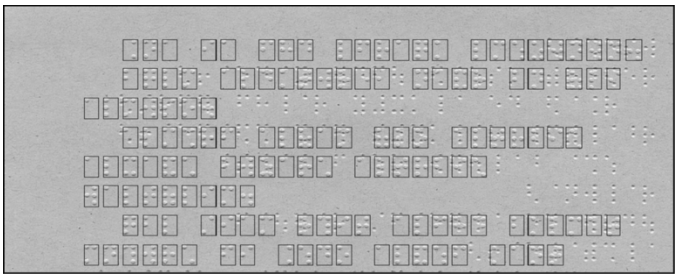

Fig. 15 The recognized Braille cell from the grid system.

III. Approach

A. System Overview

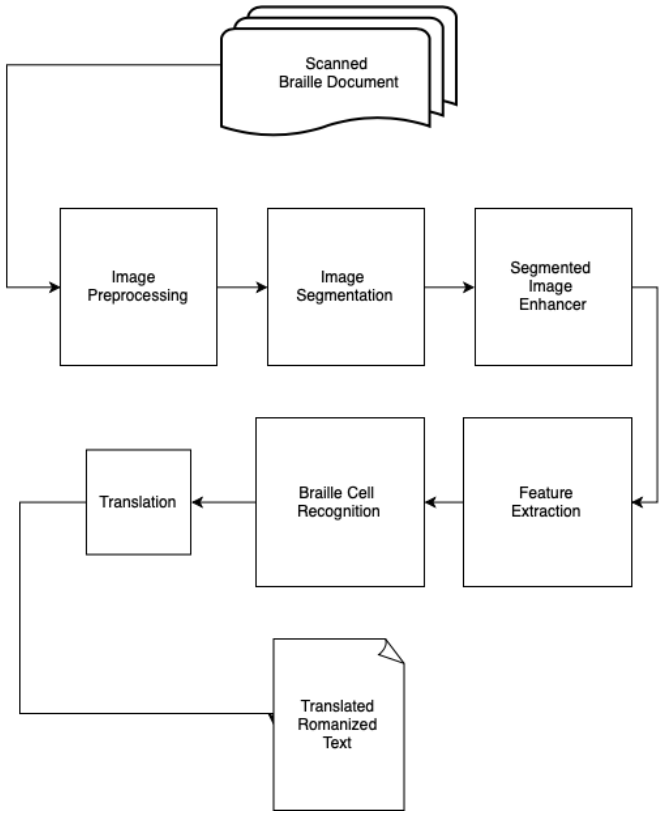

Fig 16. System overview

Our proposed system is slightly different. (1.) First, we start with an image of scanned Braille document; for the purpose of our algorithm, the Braille document must be single sided. (2.) Then through preprocessing, we will remove noise by applying filter. (3.) Then for segmentation, we will be using Otsu's method. (4.) After obtaining the segmented image, we will enhance the image by rebuilding the recto dots to a fuller circle. (5.) For feature extraction, we want to use the distance between each recto dot as representation for Braille cell recognition. (6.) For Braille cell recognition, we will build a grid system and use the calculated distance to represent Braille cell and perform crop to extract individual cell. (7.) Translation will include digitizing the Braille cell and use matching to translate individual text character.

B. Image Segmentation

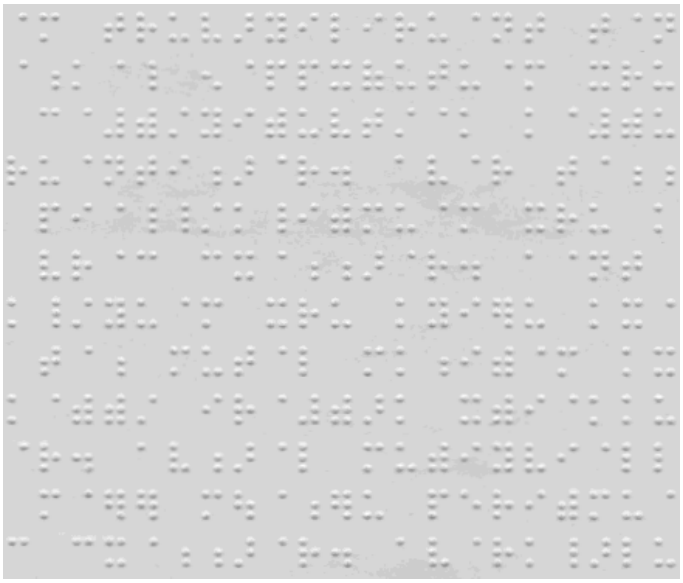

Fig. 17 A scanned of Braille text obtained from Google Image search. This will be our demonstration image.

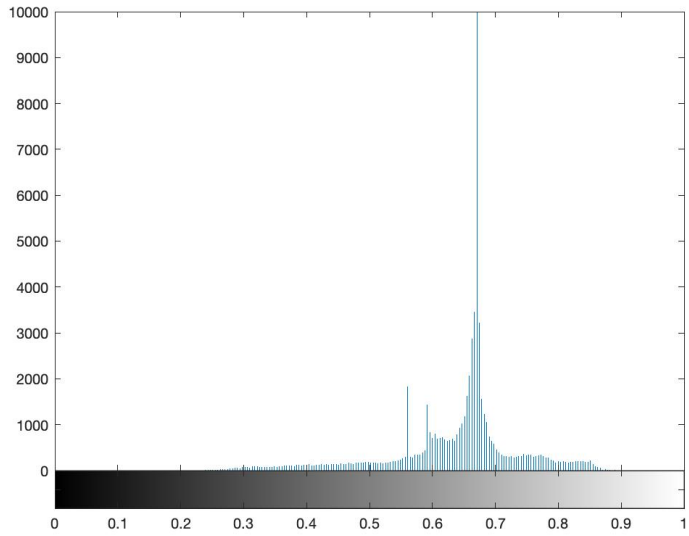

Fig 18. The histogram representation of Fig. 17 after it has been filtered from Gaussian filter and normalized.

In this section, we will use Figure 17 as our demonstration image. We first build a histogram of Figure 17. From observing the histogram, we can correlate the lower values (< 0.6) to be the black pixel of the recto dots. Our original approach was to try to implement a stability threshold with beta distribution parameters; however, the calculation did not yield a useable threshold value. Thus, we will implement segmentation using MATLAB's built-in thresholding method.

This thresholding takes the filtered image and then binarizes it in such a way that only the bumps are black, and the rest of the image is white, this enables us to work directly with what we know are bumps and ignore the rest. The method of thresholding we are employing is Otsu's Method which we found to work better than a basic thresholding of pixels greater than a certain greyscale value. Otsu's method aims to find the optimal thresholding for the images based on a histogram made from the number of pixels of each greyscale value. Although Otsu's method was originally designed for a bi-modal set of histogram data and our images tend to only have a single mode, this method might not be one that most would initially consider. However, after trying other methods of thresholding, Otsu's method yielded the best results for determining with a high rate of success which areas were individual bumps and which areas were not.

C. Image Filtering

Each image needs to be filtered in order to remove the background, identify the bumps and remove imperfections such as marks on the paper or other areas which are not bumps. In order to do this the images our program takes are filtered using a gaussian filter to try any remove some of these areas we do not wish to work with. Before the gaussian filter is applied, the original image goes through greyscaling so that our program does not need to account for the different colors since the focus is on the dots. The gaussian filter then removes noise from the image and smooths the bumps with the shadows they create in the images so that the shadow and the bump itself are not misclassified as being two separate objects. Due to most image inputs of braille being a white or semi-white page the gaussian filter was chosen to reduce gaussian noise which might occur, and we need not account for other image noise types like salt and pepper noise.

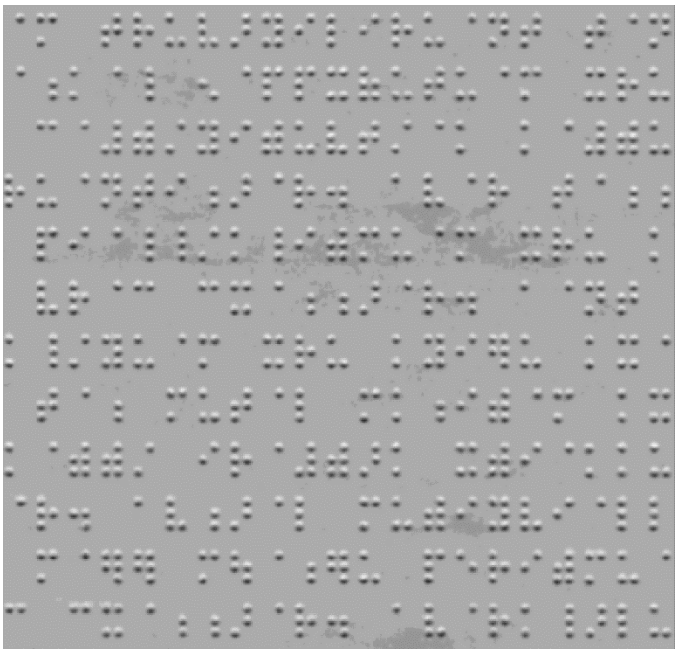

Fig. 19 The gaussian filtered image with kernel size of [5, 5] and SD of 0.6.

D. Segmented Image Enhancing

Once a black and white preprocessed image is obtained using the methods of filtering and thresholding, most likely the bump locations are not going to be perfect circles or even circular in nature which could potentially mess up the character detection algorithm. In order to prevent these potential errors, the bumps are located in the image and transposed onto a blank image with a uniform area and perfectly circular shape. In order to perform this operation first, the location of each individual bump is detected. We classify the bumps as individual objects and can detect the size and centroid location of each bump. To detect the individual bumps as separate objects we use the *bwconncomp(...)* function built into MATLAB. This function runs connected component analysis to classify each bump as a separate entity. We then use *regionprops(...)* to obtain the centroid location and total area of each bump. This location and area data are stored and can then be used to place ideal circles in place of the bumps on a blank image for our character recognition algorithm to work with. A new dot of predetermined radius is constructed at each identical centroid location on the blank image. Once this process is finished, we have a completely blank image with no background imperfections and, perfectly placed and identically sized dots to run through our character recognition algorithm.

Fig. 20 Sample region of recto dots before enhancing.

Fig. 21 Sample region of recto dots after enhancing.

E. Feature Extraction

Feature extraction revolves around extracting the distance between recto dots centroid. To do this, we will build a general grid of for the image. First, we will go through every single recto dot and place a horizontal line on top of the recto dot and a vertical line to left of the it. This is a modification to the proposed method by earlier works. Instead of flanking the recto dots with 4 lines, we are only using 2 lines per dot. Since we are using less lines, we will not have to calculate as many intersections.

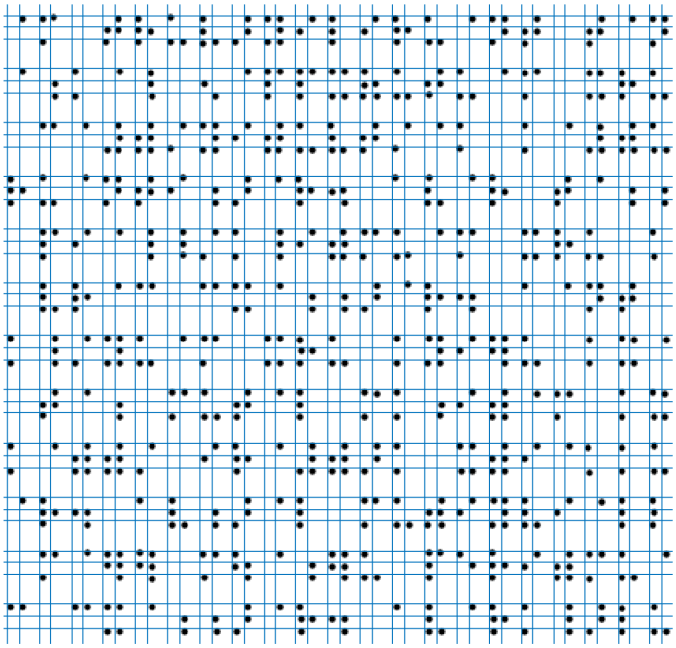

Fig. 22 The generated grid.

After generating the grid, we find the all of the intersections by the lines. Since previous work does not mention how they extract cells from the generated grid, we will use intersections to determine starting point for each cell. However, before we can start detecting intersection, we have to detect duplicate lines. To do this, we individually check if each line is within a certain percentage of the other line vicinity. If it is, we remove the line and considered it to be repeated. The distance percentage is one of the parameters that must be set by user.

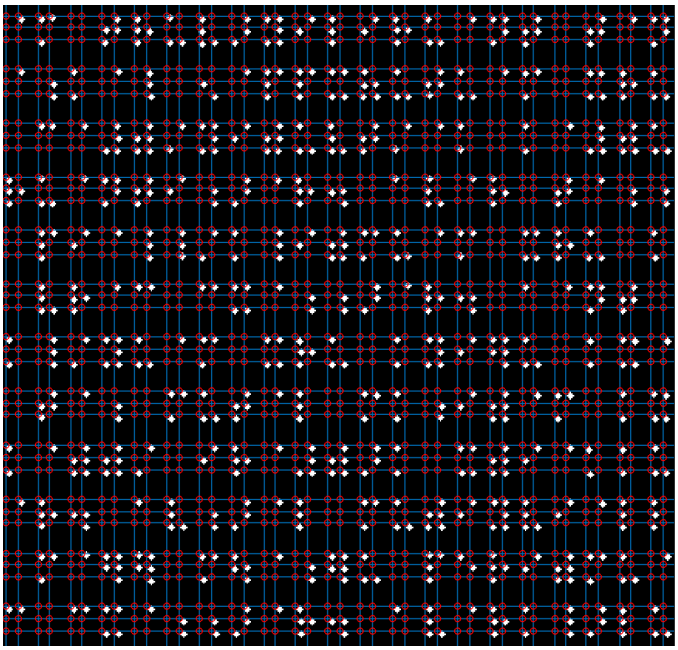

Fig. 23 The detected intersection

After determining the intersection of each line, we will run the algorithm of extracting the shortest distance and longest distance horizontally as well as vertically between each intersection. The distances that was calculated will be our distance between each recto dots and distance between each cell.

F. Braille Cell Recognition

Before we can iterate through each point and place our cell, we have to eliminate unneeded detected intersection. For this purpose, we will use the upper left corner as potential cell starting point. That means, we will have to eliminate every other detected intersection horizontally and retain every 2 intersection points vertically. The result of extracting subset of intersection points will yield us the potential upper left corner of a Braille cell. By using the previously determined standard distance for Braille cell, we can highlight the region of interest on the Braille image. We will extract those regions of interest for translation process.

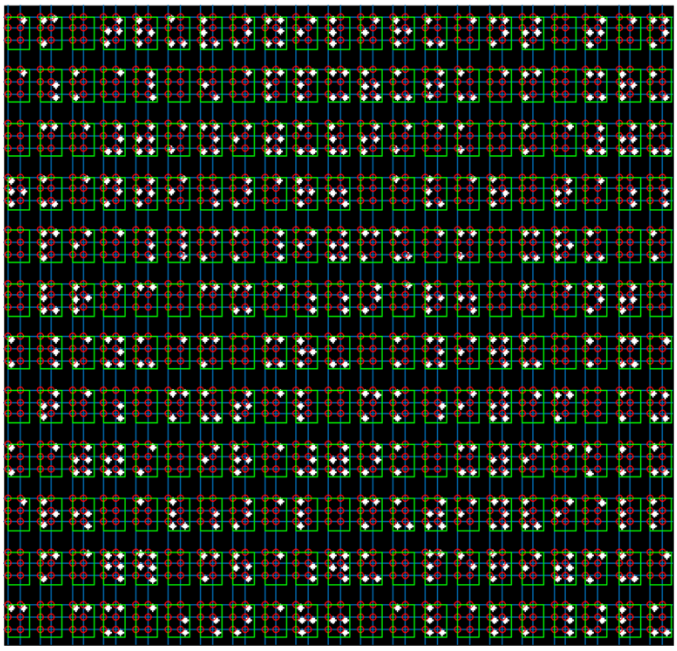

Fig. 24 The outlined of region of interest on a scanned Braille image.

G. Translation

After splitting each Braille character into its respective cell, we now have the ability to read each separate cell on their own. As mentioned previously, we will be utilizing Grade 1 Braille due to the direct translation to the English alphabet.

Although we are able to detect each Braille cell, we still need to translate the quantity and orientation of the “raised dots” in each of these cells to determine the translation. The method implemented in our Optical Braille Recognition system divide the singular Braille cell into six quadrants as shown in Figure X below. The Braille cell is separated by dividing the cell into two columns and three rows, then cropping each of these sub-cells into a set.

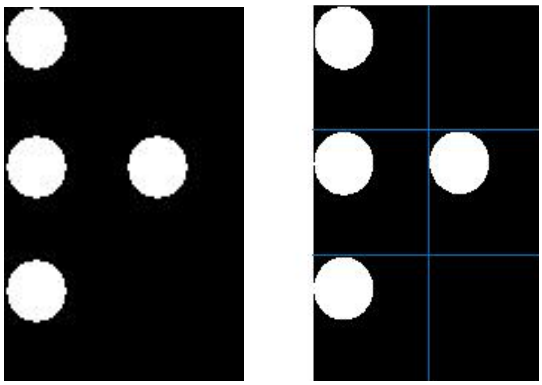

Fig. 25 A singular Braille cell separated into six different quadrants

Because each Braille cell is split into six respective quadrants, the orientation of the "raised dots" and the quantity can now be utilized. The methodology to determine whether a dot exists inside the separate quadrant uses a simple mean calculation process. By using a simple black background that has a binary value of 0 and finding the mean of each quadrant, any cell with a white circle or "raised dot" will have a mean greater than zero. Furthermore, these quadrants can be paired into a set of six with their respective position number. By employing both these techniques, we can determine the orientation and quantity of these "raised dots" inside each Braille cell.

After determining whether each quadrant has a "raised dot" or not, we can create a 3x2 matrix that will correspond with that respective Braille cell. If the mean is greater than 0 in any of the separate quadrants, the corresponding cell in the matrix will have a score of 1. If the mean is 0, no white dot exists, then the corresponding cell will be zero as shown in Figure X.

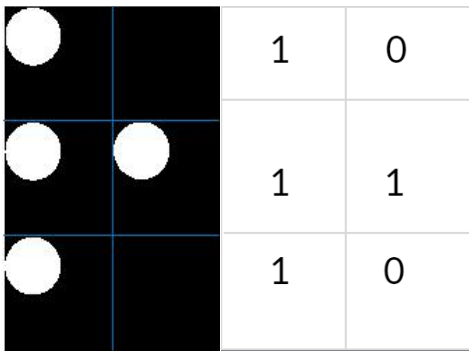

Fig. 26 Assigning binary values for the separate Braille cell submatrix

Once the Braille cells can be reduced to the submatrices like Figure X, the translation step would involve creating a database of these submatrices for each English alphabet character by using the Grade 1 Braille system. After creating the Braille-to-English binary alphabet value database as shown in Figure X, every character can be translated by matching their submatrix values to the database.

IV. Analysis of Experiment Results

For this section we will experiment with Braille text in video games. The main reason for the usage is that it would be easier to find Grade 1 Braille text as oppose to more complicated Braille books. Being able to recognize only Grade 1 Braille is a limitation of this algorithm. The following figure is the example image we will be using to demonstrate the algorithm and its capability.

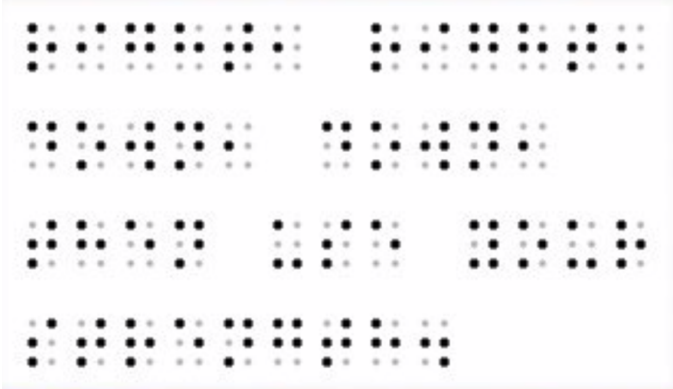

Fig. 27 The sample Braille text from Pokémon Omega Ruby.

The following is the histogram of the filtered and normalized Braille text. This histogram shows the main problem of thresholding with Otsu's Method: The histogram is unimodal. Currently, we are using Otsu method as a mechanism for automatic thresholding. In the future, an algorithm that utilize stability thresholding will be more desirable. Figure 29 shows the result of using Otsu's Method to perform segmentation.

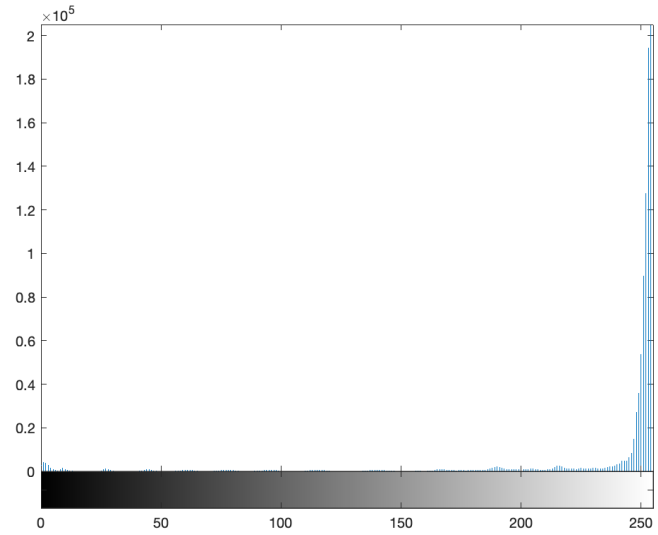

Fig 28. The grayscale histogram of Fig 27.

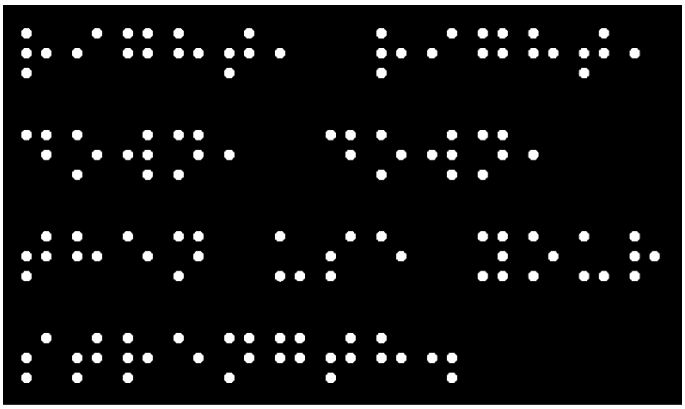
Fig. 29 The threshold image of Fig. 27 using Otsu's method to determine the threshold value

From the result of Figure 29, the threshold eliminates the filler gray dots of Figure 27. Sometimes, digital Braille cell includes smaller gray dots for the non-recto dots. This is very helpful in the sense that we do not need to filter those dots out before running the translation algorithm.

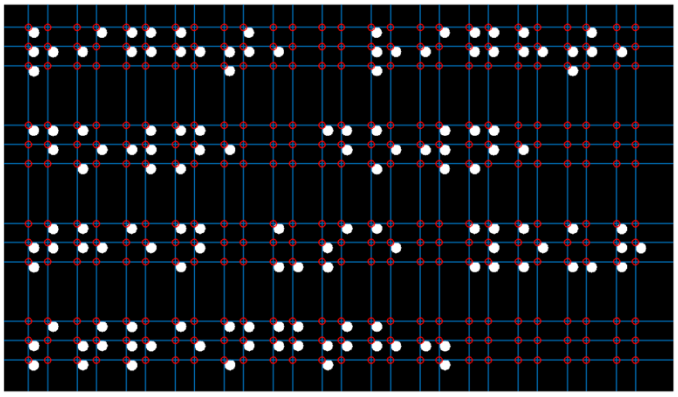
Figure 30. The grid formation of the Braille Cell.

When running the grid formation algorithm, one parameter we have to set is the percentage difference between each grid line. Because of algorithm relies on each dot to generate their own vertical and horizontal lines, we need to make sure that the dot does not draw another line at the location that already has a line. Whether it will be 5% difference or 10% difference between an existing line and line about to be placed, the user has to specify this parameter. This is another limitation that take away the autonomy of the program.

After generating the grid, Figure 31 shows the extract cell. For this algorithm, we rely on the process to extract out the unique x location and y location of the grid. This means that we have to loop through the detected intersection to filter out the unique x and y location. Once again depends on the image being worked on, the user has to set the percentage deviation value.

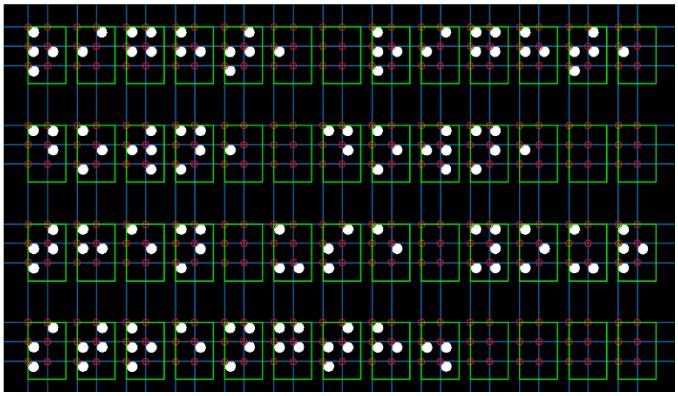
Figure 31. The detected Braille cells.

In comparison with our test for Figure 24, it is 100% detection for each Braille cell, even for the spacing empty cell. However, the limitation is one condition: collectively, each row and column must sum up to be a full Braille Cell. This means that it relies on a Braille cell with 6 full dots to determine the grid. Just as long as the entire row of Braille cells can form a cell with 6 dots, the algorithm will pick up 100% of the cells.

Up to this point, the following are the parameters the user of the program has to take in consideration.

Table 2. The user entered parameters required for Braille Cell Detection.

Parameters	Description
rebuild_radius_scale	The scaling factor rebuilding rector dots. We take the detected connected component's radius and multiply by this factor.
rebuild_percent_dif_area	This is the percentage difference between the smallest connected component and the current iteration. Set this to -1 if your segmented image yield evenly distributed recto dots.
pixel_dif_x	This is the percentage difference in X-axis value of each vertical lines.
pixel_dif_y	This is the percentage difference in Y-axis value of each vertical lines.
intx_per_dif_x	This is the intersection X-axis values percentage difference we use to filter out unique X-axis value.
intx_per_dif_y	This is the intersection Y-axis values percentage difference we use to filter out unique Y-axis value.
cell_width_scale	This is the scale factor we use to multiply the horizontal distance with to get the cell width. Default is 2.
cell_height_scale	This is the scale factor we use to multiply the vertical distance with to get the cell height. Default is 3.

For matching, we've made assumption that the program will come with a folder of Braille alphabet cropped Braille Cell. We will use those images to build an alphabet to match with. Currently, we only have the elementary Braille alphabet of A-Z. We do not have any special characters for Grade 1 Braille yet so the translation algorithm will skip a few untranslatable characters. Figure 32 show the result of our translation algorithm.

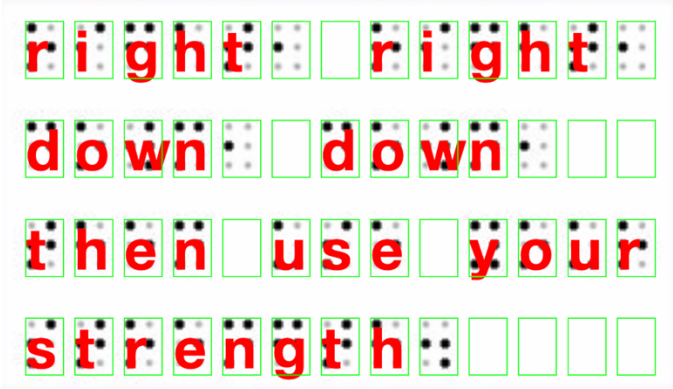

Figure 32. The translated Braille texts.

The translation algorithm was limited by the alphabet that we used. The untranslated Braille cells are Braille characters for comma and period. However, due to this, we can be sure that translation is merely the limitation of what alphabet was fed into the algorithm.

V. Future Works and Issues

Based on our current solution and algorithms, we can't achieve 100% automation. In the future, our main goal is to implement an automatic thresholding algorithm. A more robust algorithm would allow us to test with different environment for Braille text scanning. To be more specific, we would like to be able to scan double sided Braille text. With the current algorithm of thresholding, we will extract both the recto dots and verso dots instead of just the recto dots. We have thought about performing thresholding with K-means or EM. However, this also required the user to set the number of clusters.

As for the extraction algorithm, the current solution that most published studies used is to generate a grid. Our grid algorithm relies on every recto dot to participate. This algorithm increased in run time the more dots we have in the sample text. One proposed solution that we can come up with is to detect the parameters of the whole braille text and just automatically generate the grid base on the 2 by

3 standard ratio of each braille cell. We know that the horizontal distance between each Braille cell is 1.5 times the horizontal distance between each within cell dots, and 2 times for vertical distance. This way, we do not have to relies on the parameters in Table 2 to calculate repeating lines.

There are many issues that we've encountered with our approach:

1. The first and foremost is the run time and scalability. Since the code relies on looping over individual dots, the run time of the code is deeply impacted. For Figure 24, the run time was over 5 minutes. As for Figure 31, the run time was instantaneous. In the future, we would also want to keep track of run time to determine the relationship between algorithm run time and large Braille text.

2. The second issue that we've ran into is the constant changing and re-adjusting the parameters. Even though the naïve algorithm is robust enough to determine Braille cell, we have to blindly adjust the parameters to get any result. Our future algorithm would need to automate the parameterizing step. If we want to implement an automatic translation through the use of smartphone camera like Google translate, we would need to automatically generate those parameters of Table 2.

VI. Summary and Conclusion

The process for this experiment is as follow:

1. *Image acquisition:* We obtain the Braille image through either scanning the physical text, taking a picture of the text or obtain the text from the internet.

2. *Image preprocessing:* We put the image through a Gaussian filter to filter out any random noise of the image. We also normalized the image and convert the image into grayscale.

3. *Image Segmentation:* We perform Otsu's algorithm to determine a threshold. However, this is not ideal. In the future we would prefer to implement other form of unimodal thresholding like Rosin's Method.

4. *Enhancing:* Sometime, image segmentation will not capture the complete Braille dots and erosion would not give us a perfect circle. We place circles on the location of detected Braille dots.

5. *Braille Cell Detection:* We generate a grid system based on the Braille text and extract the grid intersection. We then calculate the distance between the intersection to obtain the standard distance between and within Braille cells.

6. *Translation:* We digitize the cropped Braille to a 6 elements array; one for each cell. We also digitize the known Braille alphabet. We then perform array comparison with the digitized alphabet.

Overall, the result of this experiment can be considered to be incomplete. On one hand, we can extract 100% of the Braille cells given that we calculate our own parameters. On the other, we can't automate the system of calculating the parameters. Due to our limited knowledge of Beta distribution and how it works, we can't implement the Stability Thresholding algorithm suggested in one of the studies. However, this project provides a solid foundation to build upon. We've explored the limitations and challenges of previous studies and identified key area to improve upon.

As technologies become more accessible to people with disabilities, a system of Braille recognition would be beneficial for translate to and from Braille for people with impaired vision. Even though our project could be considered to be in a very early stage, it was based on some of the incredible studies and algorithm that make computer vision an exciting field for accessibility.

VII. References

[1] Ng, C.m., et al. “Regular Feature Extraction for Recognition of Braille.” *Proceedings Third International Conference on Computational Intelligence and Multimedia Applications. ICCIMA'99 (Cat. No.PR00300)*, 1999, doi:10.1109/iccima.1999.798547.

[2] Åhlén, Johan. “Braille Alphabet - Decoder, Translator.” *Boxentriq*, www.boxentriq.com/code-breaking/braille-alphabet.

[3] Mousa, Aisha, et al. “Smart Braille System Recognizer.” *International Journal of Computer Science Issues*, vol. 10, no. 6, 2013, ISSN (Print): 1694-0814.

[4] Al-Saleh, Amany, et al. “Dot Detection of Optical Braille Images for Braille Cells Recognition.” *Computers Helping People with Special Needs: 11th International Conference; Proceedings*, by Klaus Miesenberger, Springer, 2008, pp. 821–826.

[5] S, N. Curve intersections. Computer software. Vers. 3.0. 21 Sept. 2010. 21 Mar. 2019 <https://www.mathworks.com/matlabcentral/fileexchange/22441-curve-intersections>.

VIII. Appendix

A. External Code

For this experiment, we used an external function called *InteX(…) [5]* by a user name NS from MATLAB's online forum Mathworks. The function is only used to determine if two line intersect each other or not.

```
function P = InterX(L1,varargin)
%INTERX Intersection of curves
%   P = INTERX(L1,L2) returns the intersection points of two curves L1
%   and L2. The curves L1,L2 can be either closed or open and are described
%   by two-row-matrices, where each row contains its x- and y- coordinates.
%   The intersection of groups of curves (e.g. contour lines, multiply
%   connected regions etc) can also be computed by separating them with a
%   column of NaNs as for example
%
%         L  = [x11 x12 x13 ... NaN x21 x22 x23 ...;
%               y11 y12 y13 ... NaN y21 y22 y23 ...]
%
%   P has the same structure as L1 and L2, and its rows correspond to the
%   x- and y- coordinates of the intersection points of L1 and L2. If no
%   intersections are found, the returned P is empty.
%
%   P = INTERX(L1) returns the self-intersection points of L1. To keep
%   the code simple, the points at which the curve is tangent to itself are
%   not included. P = INTERX(L1,L1) returns all the points of the curve
%   together with any self-intersection points.
%
%   Example:
%       t = linspace(0,2*pi);
%       r1 = sin(4*t)+2;  x1 = r1.*cos(t); y1 = r1.*sin(t);
%       r2 = sin(8*t)+2;  x2 = r2.*cos(t); y2 = r2.*sin(t);
%       P = InterX([x1;y1],[x2;y2]);
%       plot(x1,y1,x2,y2,P(1,:),P(2,:),'ro')

%   Author : NS
%   Version: 3.0, 21 Sept. 2010

%   Two words about the algorithm: Most of the code is self-explanatory.
%   The only trick lies in the calculation of C1 and C2. To be brief, this
%   is essentially the two-dimensional analog of the condition that needs
%   to be satisfied by a function F(x) that has a zero in the interval
%   [a,b], namely
%           F(a)*F(b) <= 0
%   C1 and C2 exactly do this for each segment of curves 1 and 2
%   respectively. If this condition is satisfied simultaneously for two
%   segments then we know that they will cross at some point.
%   Each factor of the 'C' arrays is essentially a matrix containing
%   the numerators of the signed distances between points of one curve
%   and line segments of the other.

    %...Argument checks and assignment of L2
    error(nargchk(1,2,nargin));
    if nargin == 1,
        L2 = L1;    hF = @lt;   %...Avoid the inclusion of common points
    else
        L2 = varargin{1}; hF = @le;
    end

    %...Preliminary stuff
    x1  = L1(1,:)';  x2 = L2(1,:);
    y1  = L1(2,:)';  y2 = L2(2,:);
    dx1 = diff(x1); dy1 = diff(y1);
    dx2 = diff(x2); dy2 = diff(y2);
```

```
    %...Determine 'signed distances'
    S1 = dx1.*y1(1:end-1) - dy1.*x1(1:end-1);
    S2 = dx2.*y2(1:end-1) - dy2.*x2(1:end-1);

    C1 = feval(hF,D(bsxfun(@times,dx1,y2)-bsxfun(@times,dy1,x2),S1),0);
    C2 = feval(hF,D((bsxfun(@times,y1,dx2)-bsxfun(@times,x1,dy2))',S2'),0)';

    %...Obtain the segments where an intersection is expected
    [i,j] = find(C1 & C2);
    if isempty(i),P = zeros(2,0);return; end;

    %...Transpose and prepare for output
    i=i'; dx2=dx2'; dy2=dy2'; S2 = S2';
    L = dy2(j).*dx1(i) - dy1(i).*dx2(j);
    i = i(L~=0); j=j(L~=0); L=L(L~=0);  %...Avoid divisions by 0

    %...Solve system of eqs to get the common points
    P = unique([dx2(j).*S1(i) - dx1(i).*S2(j), ...
                dy2(j).*S1(i) - dy1(i).*S2(j)]./[L L],'rows')';

    function u = D(x,y)
        u = bsxfun(@minus,x(:,1:end-1),y).*bsxfun(@minus,x(:,2:end),y);
    end
end
```

B. Braille_Detection_Test.m

```
close all;
clear, clc;

% figure;

is_demo = 1;
mkdir('Cropped');
addpath('Cropped');
addpath('Alphabet');
cr_dir = dir('Cropped');
test_image = 'test.png';

% USER SET PARAMTERS
%==========================================================================
rebuild_radius_scale = 1.0;
rebuild_percent_dif_area = -1;
pixel_dif_x = 5;
pixel_dif_y = 5;
intx_per_dif_x = 0.3;
intx_per_dif_y = 0.3;
cell_width_scale = 2;
cell_height_scale = 3;
%==========================================================================

% Example Run

% IMAGE PRE-PROCESSING
%==========================================================================
% [1] Read in braille image
i_og = imread('pokemon_ruby.png');
i_gray = rgb2gray(i_og);
i_norm = mat2gray(i_gray);
s = size(i_norm);

% [2] Filter Image
k = fspecial('gaussian', [5 5], 0.6);
i_filt = imfilter(i_norm, k);

%==========================================================================


% IMAGE SEGMENTATION
%==========================================================================
% [3] Threshold image
otsu = graythresh(i_filt);
BW = imbinarize(i_filt, otsu);
rect = [2 3 (s(2) - 3) (s(1) - 4)];
im = ~(imcrop(BW, rect));
%==========================================================================

% IMAGE ENHANCING
%==========================================================================
% [4] Rebuild image
im_new = bwrebuild(im, rebuild_radius_scale, rebuild_percent_dif_area);
subplot(2, 2, 1); imshow(im_new);
%==========================================================================

ax = subplot(2, 2, 2); hold on;

% FEATURE EXTRACTION
%==========================================================================
% [5] Generate Grid
% figure; imshow(im_new);
% pause;
[h_line, v_line, stats] = gridgen(im_new, pixel_dif_x, pixel_dif_y, ax);
```

```
% [6] Get intersection
intersections = get_intersection(stats, h_line, v_line);


% [7] Remove repeating intersection
intersections = remove_repeat(stats, intersections, ...
                              intx_per_dif_x, intx_per_dif_y);

% [8] Obtain the unique X and Y location of intersections
[x, y] = get_xy(intersections);
%=========================================================================
hold off;

ax_2 = subplot(2, 2, [3 4]);
hold on;
% BRAILLE CELL DETECTION
%=========================================================================
% [9] Extract region of interest for translation
extract_cell(im_new, x, y, cr_dir, is_demo, ...
            cell_width_scale, cell_height_scale, ax_2, i_og);
%=========================================================================
hold off;

if ~is_demo
% GENERATE ALPHABET TEMPLATE
%=========================================================================
alph_dir = dir('Alphabet');
template = getAlphabetTemplate(alph_dir, '.jpg');
%=========================================================================


% DECODE CROPPED AREA OF INTEREST
%=========================================================================
img_dir = dir('Cropped');
r = decode(template, img_dir, '.jpg');
disp(r);
%=========================================================================
end
```

C. bwrebuild.m

```
function [I] = bwrebuild(I_bw, r_scale, pdiff_area)
% BWREBUILD  Rebuild segmented image by filling out uneven Braille dots.

    im_size = size(I_bw);
    I = 255 * ones(im_size(1), im_size(2), 'uint8');

    % Perform connected component
    cc = bwconncomp(I_bw);
    stats = regionprops(cc, 'Centroid', 'EquivDiameter', 'Area');

    % Get the smallest area
    areas = vertcat(stats.Area);
    smallest_area = min(areas);

    % Iterate through the detected components
    %   to remove noise that was picked up as a
    %   braille dot.
    for i = 1:numel(stats)

        c = stats(i).Centroid;
        radius = (stats(1).EquivDiameter / 2) * r_scale;
        area = stats(i).Area;

        % Check if the area is similar to the smallest area by
```

```
        %   user specified percent.
        perdif = per_diff(area, smallest_area);

        % If it is more than user specified percent,
        %   it is not a noise.
        if perdif > pdiff_area
            x = c(1);
            y = c(2);

            % Insert circle to image.
            I = insertShape(I,'FilledCircle',[x y radius], ...
                                'LineWidth', 5, ...
                                'Opacity', 1, ...
                                'Color', 'Black');
        end
    end
end
```

D. gridgen.m

```
function [h_line, v_line, stats] = gridgen(im, x_diff, y_diff, s_ax)

    im_gray = rgb2gray(im);
    im_bw = ~imbinarize(im_gray);
    im_size = size(im_bw);
    cc = bwconncomp(im_bw);
    bw = bwareafilt(im_bw, cc.NumObjects);
    stats = regionprops(cc, 'Centroid', 'EquivDiameter');

    subplot(s_ax); imshow(bw);

    centroids_array = vertcat(stats.Centroid);
    [~, sortindx] = sort(centroids_array(:,1));
    stats = stats(sortindx);

    h_line = cell(numel(stats), 1);
    v_line = cell(numel(stats), 1);

    x_coord = []; count_x = 1;
    y_coord = []; count_y = 1;


    for x = 1:numel(stats)

        similar_x = 0;
        similar_y = 0;

        c = stats(x).Centroid;
        radius = stats(x).EquivDiameter / 2;

        y_1 = round(c(2) - radius);
        x_1 = round(c(1) - radius);

        for i = 1:numel(x_coord)
            if abs(x_coord(i) - x_1) < x_diff && abs(x_coord(i) - x_1) ~= 0
                similar_x = 1;
                break;
            end
        end

        for i = 1:numel(y_coord)
            if abs(y_coord(i) - y_1) < y_diff && abs(y_coord(i) - y_1) ~= 0
                similar_y = 1;
                break;
            end
```

```
        end

        if similar_x == 0
            x_coord(count_x) = x_1;
            count_x = count_x + 1;
            line([x_1, x_1], [0, im_size(1)], 'LineWidth', 1.0);
            l_1 = [x_1 x_1; 0 im_size(1)];
            h_line{x} = l_1;
        end

        if similar_y == 0
            y_coord(count_y) = y_1;
            count_y = count_y + 1;
            line([0, im_size(2)], [y_1, y_1], 'LineWidth', 1.0);
            l_2 = [0 im_size(2); y_1 y_1];
            v_line{x} = l_2;
        end
    end

    emptyCells = cellfun('isempty', h_line);
    h_line(all(emptyCells,2),:) = [];

    emptyCells = cellfun('isempty', v_line);
    v_line(all(emptyCells,2),:) = [];

end
```

E. get_intersection.m

```
function x_points = get_intersection(stats, h_line, v_line)

    count = numel(stats) * numel(stats);
    x_points = cell(count, 1);
    n = 1;

    for i = 1:numel(h_line)
        for j = 1:numel(v_line)
            p = InterX(h_line{i}, v_line{j});
            x_points{n} = p;
            n = n + 1;
        end
    end

end
```

F. remove_repeat.m

```
function x_points = remove_repeat(stats, points, x_diff, y_diff)

    count = numel(stats) * numel(stats);

    for j = 1 : count
        poi = points{j};

        if ~isempty(poi)

            for k = j + 1 : count

                if ~isempty(points{k})
```

```
                    p_x = points{k}(1);
                    p_y = points{k}(2);
                    poi_x = poi(1);
                    poi_y = poi(2);
                    pd_x = per_diff(p_x, poi_x);
                    pd_y = per_diff(p_y, poi_y);
                    if pd_x < x_diff && pd_y < y_diff
                        points{k} = [];
                    end
                end
            end
        end
    end

    emptyCells = cellfun('isempty', points);
    points(all(emptyCells,2),:) = [];

    x_points = points;

end
```

G. get_xy.m

```
function [unique_x, unique_y] = get_xy(intersections)

    unique_x = [];
    unique_y = [];

    c = numel(intersections);

    c_x = 1;
    c_y = 1;

    for i = 1:c

        p = intersections{i};

        if ~ismember(p(1), unique_x)
            unique_x(c_x) = p(1);
            c_x = c_x + 1;
        end

        if ~ismember(p(2), unique_y)
            unique_y(c_y) = p(2);
            c_y = c_y + 1;
        end

        plot(p(1,:), p(2,:), 'ro');
    end

    unique_x = sort(unique_x);
    unique_y = sort(unique_y);
end
```

H. extract_cell.m

```
function extract_cell(im, x, y, directory, is_demo, ...
  cell_width_scale, cell_height_scale, ax, im_og)

   [~, x_s] = difPoints(x);
```

```
    [~, y_s] = difPoints(y);

    p_x = removeX(x, x_s);
    p_y = y(1:3:end);

    dir_str = directory.folder;
    count = 1;

    subplot(ax), imshow(im_og);

    for i = 1:numel(p_y)
        for j = 1:numel(p_x)
            x = p_x(j);
            y = p_y(i);
            width = x_s * cell_width_scale;
            height = y_s * cell_height_scale;

            if ~is_demo
                crop = ~(imcrop(im, [x, y, width, height]));
                imname = strcat(dir_str, '/', num2str(count), '.jpg');
                imwrite(crop, imname);
                count = count + 1;
            else
                crop = ~(imcrop(im, [x, y, width, height]));
                T = getText(crop);
                text(x, y + (height / 2), T, 'FontSize', 50, 'FontWeight', 'bold', 'Color', 'red');
                rectangle('Position', [x y width height], ...
                          'LineWidth', 1, ...
                          'EdgeColor', 'g');
            end
        end
    end
end
```

I. getAlphabetTemplate.m

```
function [r] = getAlphabetTemplate(alphabet_dir, kind)

    % Letter is just to build image path
    letter = 'abcdefghijklmnopqrstuvwxyz';

    r = cell(numel(letter), 1);

    % start at 3 to skip over the . and .. directory
    for x = 3 : 1 : numel(alphabet_dir)

        % Reseting back to 1
        index = x - 2;

        % Don't worry about this, this is just build path to image.
        dir = strcat(alphabet_dir(x).folder);
        imname = strcat(letter(index), kind);
        im_path = strcat(dir, '/', imname);

        % Calling parse function to parse crop image into logical array.
        % Then store it into a cells of array.
        r{index} = parse(im_path);

    end
end
```

I. decode.m

```
function [r] = decode(alpha_template, img_dir, kind)

    letter = 'abcdefghijklmnopqrstuvwxyz';

    for x = 3 : 1 : numel(img_dir)

        index = x - 2;

        dir = strcat(img_dir(x).folder);
        imname = strcat(num2str(index), kind);
        imdir = strcat(dir, '/', imname);

        result = parse(imdir);

        for i = 1:numel(letter)

            % PRETTY MUCH IF ARRAY MATCH ONE OF TEMPLATE, THAT IS THE
            % LETTER
            if isequal(alpha_template{i}, result)
                r(index) = letter(i);
            end
        end
    end
end
```

J. parse.m

```
function [r] = parse(img)

    I = imread(img);
    I_bw = imbinarize(I);
    I_size = size(I_bw);

    y = round(I_size(1) / 3);
    x = round(I_size(2) / 2);

    x_1 = imcrop(I_bw, [0, 0, x, y]);
    x_2 = imcrop(I_bw, [x, 0, x, y]);
    x_3 = imcrop(I_bw, [0, y, x, y]);
    x_4 = imcrop(I_bw, [x, y, x, y]);
    x_5 = imcrop(I_bw, [0, y * 2, x, y]);
    x_6 = imcrop(I_bw, [x, y * 2, x, y]);

    set = {x_1, x_2, x_3, x_4, x_5, x_6};

    result = zeros(1, 6);

    for x = 1:numel(set)

        % GET MEAN VALUE OF ALL PIXEL
        m = mean(set{x}, 'all');

        % IF MEAN IS NOT BLACK, IT MEANS THERE IS A WHITE DOT.
        if m > 0.01
            result(x) = 1;
        end
    end

    r = result;
end
```

K. per_diff.m

```
function p = per_diff(x, y)
    p = 100 * abs((x - y) / x);
end
```

L. difPoints.m

```
function [l, s] = difPoints(x)
    s = intmax('uint64');
    l = 0;

    x_c = 2;

    while x_c <= numel(x)
        dif_x = x(x_c) - x(x_c - 1);

        if s > dif_x && dif_x > 2
            s = dif_x;
        end

        if l < dif_x
            l = dif_x;
        end

        x_c = x_c + 1;
    end
end
```

N. getText.m

```
function T = getText(image)

   alph_dir = dir('Alphabet');
   template = getAlphabetTemplate(alph_dir, '.jpg');

   I_bw = image;
   I_size = size(image);

   y = round(I_size(1) / 3);
   x = round(I_size(2) / 2);

   x_1 = imcrop(I_bw, [0, 0, x, y]);
   x_2 = imcrop(I_bw, [x, 0, x, y]);
   x_3 = imcrop(I_bw, [0, y, x, y]);
   x_4 = imcrop(I_bw, [x, y, x, y]);
   x_5 = imcrop(I_bw, [0, y * 2, x, y]);
   x_6 = imcrop(I_bw, [x, y * 2, x, y]);
   set = {x_1, x_2, x_3, x_4, x_5, x_6};

   result = zeros(1, 6);

   for x = 1:numel(set)

       % GET MEAN VALUE OF ALL PIXEL
       m = mean(set{x}, 'all');

       % IF MEAN IS NOT BLACK, IT MEANS THERE IS A WHITE DOT.
       if m > 0.01
           result(x) = 1;
```

```
        end
    end

    letter = 'abcdefghijklmnopqrstuvwxyz';

    for i = 1:numel(letter)

        % PRETTY MUCH IF ARRAY MATCH ONE OF TEMPLATE, THAT IS THE
        % LETTER
        if isequal(template{i}, result)
            T = letter(i);
            return;
        end
    end

    T = ' ';

end
```