ECE661: Homework 7

Ahmed Mohamed (akaseb@purdue.edu)

November 6, 2014

1 Character Recognition

In order to recognize the characters in a set of test images using a training image, my implementation follows the following steps for each of the images:

1. Segment the foreground using Otsu's algorithm as shown in Section 2. The foreground represents the letters on the image. The output of this process is a mask with non-zero pixels corresponding to the foreground pixels.
2. Perform component labeling as shown in Section 3 in order to label the different disconnected components in the foreground mask. The output of this process is an image whose pixel values are labels (integers). Each unique label correspond to a component (which is actually a letter) in the image.
3. Clean the components as shown in Section 4 by removing the very big components which represent the background, not the actual characters, as well as the very small components (likely to be noise).
4. Perform Harris corner detection to find the sharpest N (I use N = 9) corners in each component as shown in Section 5.
5. Construct a shape vector for each single component (character) in the image as shown in Section 6.
6. If this is the training image, keep the shape vectors for the matching process. If this is a test image, find the best component match of the shape vector of each component in the test image as shown in Section 7.

2 RGB Image Segmentation Using the Otsu algorithm

Given a color image, my implementation follows the following steps to extract the foreground of an image.

1. Separate the RGB color channels of the input image into three grayscale images.
2. Get the foreground mask for each channel using the Otsu algorithm as described in the next subsection.
3. To merge the three masks together into a single foreground mask, we observe that the foreground is always colorful (high pixel values), and the background is either black or white. We also note that letters have different colors. So we need the foreground to be the union of all the foregrounds from the three RGB channels. Hence, the overall foreground mask is:

$$mask = mask_b \ OR \ mask_g \ OR \ mask_r$$

where $mask$, $mask_b$, $mask_g$, $mask_r$ are the overall, blue, green, and red masks respectively. The training image is a little different since the foreground is black and the background is white, so the masks have to be inverted.

2.1 Grayscale Otsu Segmentation

Given a grayscale image, my implementation of the Otsu algorithm follows these steps:

1. Construct a 256-level histogram h of the image, such that $h[i] = n_i$ is the number of pixels whose grayscale value equal to i.
2. Calculate the average grayscale value of the image.

   $$\mu_T = \sum_1^L i p_i$$

   where

   $$p_i = n_i/N$$

   and L is the total number of levels, and N is the total number of pixels in the image.
3. For each level in the histogram, calculate:
   (a) The zeroth-order cumulative moment

   $$\omega(k) = \sum_1^k p_i$$

   (b) The first-order cumulative moment

   $$\mu(k) = \sum_1^k i p_i$$

   (c) The between-class variance

   $$\sigma_B^2(k) = [\mu_T \omega(k) - \mu(k)]^2 / [\omega(k)(1 - \omega(k))]$$
4. Choose $threshold = k^*$ such that $\sigma_B^2(k^*)$ is maximum.
5. Construct a mask whose pixels is 1 if the corresponding pixels in the original image is greater than the threshold, and 0 otherwise. This mask represents the foreground of the image.

3 Component Labeling

Given the foreground mask, the output of this process is a labels image whose pixel values are labels (integers). Each unique label correspond to a component (which is actually a letter) in the image. My component labeling implementation follows the following steps:

1. First pass: assign temporary labels for connected pixels, and record labels equivalences. To achieve that, for each pixel in the mask:
   (a) If the pixel value is 0, assign its label to 0.
   (b) Construct the equivalence set of labels for this pixel as the labels of the neighboring pixels (west, north west, north, north east) whose pixel value is equal to the value of this pixel.

(c) Note that, in case of 4-connectivity, we only check for the west and north pixels. We needed the 4-connectivity with the first test image because two characters were 8-connected.

(d) Check the size of the equivalence set:

i. If the size of the equivalence set is equal to 0, this means that none of the neighboring pixels has labels, so we assign a new label to this pixel.

ii. If the size of the equivalence set is equal to 1, this means that only one neighboring pixel has a label, so we assign this label to this pixel.

iii. If the size of the equivalence set is more than 1, this means that the neighboring pixels have different labels. In this case we need to record this equivalence. So, we add this equivalence set to a global list of equivalence lists. And we choose the least label from the equivalence set to this pixel.

iv. Note that when adding the equivalence set the the list of equivalence sets, the equivalence set may share some labels with any of the previous equivalence sets. In this case, we merge all the sets who intersect into a single equivalence set.

2. Second pass: resolve equivalences. To achieve that, for each pixel with non-zero label in the labels image:

(a) Check the list of equivalence sets. If the label exists in one of the sets, assign the least label from the set.

(b) If not, leave the label as is.

(c) Note that after this pass, all the labels who exist in the same equivalence set will have the least label in teh set.

4 Cleaning Components

This process has two main gaols.

1. Remove the components whose size is more than 30,000 pixels. These components are likely to be the background, because some images have white backgrounds. This is done by creating a histogram that represents the frequency of each label. After that, we iterate over the image to set the the labels whose frequency is more than 30,000 to the label 0 (background).

2. Remove the components whose size is less than 100 pixels. These components are likely to be noise.

3. For convenience and implementation ease, the method replace the non-continuous labels with continuous ones. The output of the component labeling could contain labels 2, 20, 40, etc. After this step, the new labels will be 1, 2, 3, etc.

5 Harris Corner Detector

My Harris corner detection program follows the following steps to find a maximum of N corners in each component in an image given a specific scale σ.

1. Initialize a list of corners for each label, and a list of the ratios of the corresponding corners. These lists categorizes the corners by their labels, and will be used to choose the sharpest corners.

2. Smooth the input image by applying a Gaussian filter with the given σ.

3. Find the x-derivative and y-derivative of each pixel in the smoothed image by applying a Haar wavelet filter. The Haar window size is the least even number that is greater than 4σ. If $\sigma = 1.2$, $windowsize = 6$. As a result, the following operator will be used to find dx:

$$\begin{pmatrix} -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \end{pmatrix}$$

And the following operator will be used to find dy:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

In order to perform this convolution efficiently, I use the integral image of the smoothed image.

4. For each labeled pixel in the image:

   (a) Construct the following matrix using the $5\sigma \times 5\sigma$ window around the pixel:

   $$C = \begin{pmatrix} \sum d_x^2 & \sum d_x d_y \\ \sum d_x d_y & \sum d_y^2 \end{pmatrix}$$

   (b) If the pixel is not a corner, one of the eigen values will be very small. At each pixel, the corner strength (how likely the pixel is a corner) is given by the following relation:

   $$CornerStrength = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

   Where k is a constant equal to 0.04. To avoid finding the eigen values of the matrix, it is known that:

   $$Det(C) = \lambda_1 \lambda_2$$

   $$Tr(C) = \lambda_1 + \lambda_2$$

   So the corner strength can now be computed as:

   $$CornerStrength = Det(C) - kTr(C)^2$$

   (c) The pixel is considered a corner if its Corner Strength is greater than a certain threshold HARRIS_THRESHOLD which I set to 10^{12}.

   (d) If the pixel is considered a corner, add the pixel to the list of corners of its label, and the ratio to the list of rations of its label.

5. A Non-maximum Suppression process is performed. In other words, a corner is eliminated if one of its neighbors has a larger corner strength.

6. Use the lists of corners and ratios to choose the sharpest N corners for each label, i.e. the corners with the largest ratios.

6 Constructing Shape Vectors

To construct a shape vector that represent a component (character), my method follows the following steps for each component.

1. Calculate the center of the component. The center is computed as follows:

$$x_{center} = (x_{max} + x_{min})/2$$

$$y_{center} = (y_{max} + y_{min})/2$$

   where x_{center} and y_{center} are the x and y coordinates of the center point. x_{max} and y_{max} x_{min} and y_{min} are the coordinates of the extreme points in the two directions.

2. Calculate the angle of each corner with the center of the component:

$$tan^{-1}((y_{corner} - y_{center})/(x_{corner} - x_{center}))$$

   The angle is then adjusted if it is negative.

3. Calculate the angle between each two consecutive corners by sorting the angles and then subtracting each two consecutive angles. The angle between the corners is the same as the arc length between the projections of the corners on the unit circle. The angle is easier to implement than the arc length between the projected corners, so I didn't project the corners.

$$ArcLength = r\theta$$

   where r is the radius of the circle (1 because it is unit circle), and θ is the angle between the two corners.

4. The shape vector is the vector of the angles between the lines joining the component center and the corners.

5. As an enhancement, I calculate the distance between each corner and the center pixel. That's because some letters have corners with the same angles, but the distances to the corners are different. Then I append the list of normalized distances to the list of normalized angles in the shape vector. This enhancement will be evaluated later.

6. Optimally, we would have N angles between the N corners. But for some components, there is no N corners, so we have to pad the shape vector with zeros.

7 Matching Letters

Given a shape vector of a component, to find the best training component match, my method follows the following procedures:

1. For each element in the shape vector:
   (a) In order to make the matching process rotation invariant, Circularly rotate the vector such that this element is the first.
   (b) For each training shape vector:
      i. Find the Euclidean distance between the rotated shape vector and the training shape vector.
      ii. If the Euclidean distance is the least one so far, the best component match is the component corresponding to the current training shape vector.

8 More Thoughts

The results section show that the overall average recognition accuracy for all the letters for all the test images is 31%. Here are my observations:

1. In my opinion, Harris corner detection is the main reason of the low performance due to the following reasons:

   (a) It is hard to tune the variable N (the maximum number of corners for each letter. If I use large N, Harris will include many not-sharp corners in the letters with few corners (e.g. I, O), which makes the matching process hard. If I use small N, Many important corners won't be included from the letters with many corners (e.g. M, W).

   (b) It is hard to set a Harris threshold that is suitable for all the images. If the threshold is low, this will include many bad corners. If it is high, this will miss many important corners.

   (c) In case of a letter with many corners (e.g. M, W, A), not all the corners will be included. For example, the letter A has 11 corner, but the maximum I use is 9, so two of the corners won't be included each time. However, the 9 sharpest corners differ from image to image, which result in a shape vector that is greatly different from the training shape vector.

2. The fonts used for different images are not the same, which makes it very hard to recognize the letters. For example, the letter O has no corners in the training image, but it has many edges in the last test image (in the word "Hollywood". This shows that the approach of using the corners to recognize letters will always be limited. This is very clear because the statistics indicate that the images with the best recognition accuracy are images 3 and 4 because their fonts are similar to the training set, while no letters recognised in Image 2 because the font is very different.

As an enhancement, I try using a modified shape vector that includes the distance to the center point. I expected that the results would be better if the shape vector encodes the normalized distance to the corner as well. That's because some letters have corners with the same angles, but the distances to the corners are different. Using this enhancement, the accuracy for some images has greatly increased (image 2 from 0% to 15%). However, the overall accuracy for all the images was almost the same 30% (vs. 31% without the enhancement).

Many other things can be done to achieve better results such as using larger training set. The training set should contain many fonts and many ways to write each letter, so that it is easier to recognize letters from the test set.

9 Results

9.1 Training Image

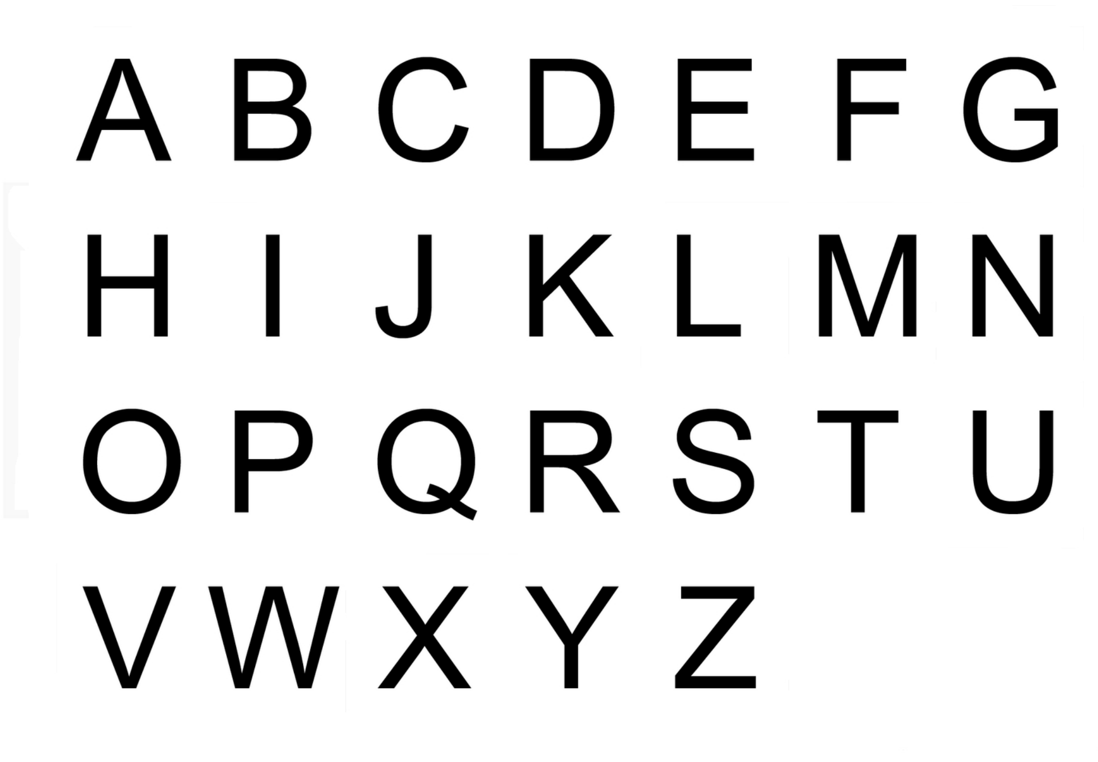

Figure 1: The input training image

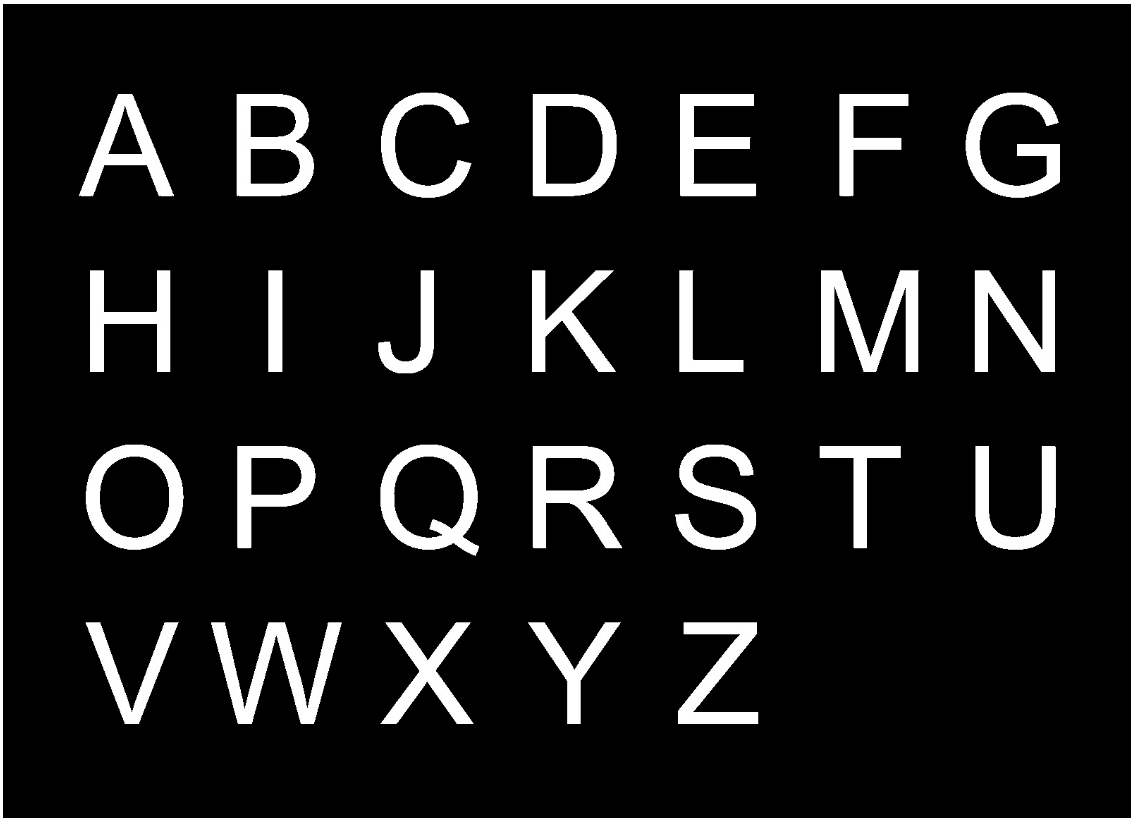

Figure 2: Otsu: the foreground mask of the image. White pixels are the foreground, while black pixels are the background.

Figure 3: Component labeling: an image representing the output of the component labeling process. Each unique label (component) is given a random color for visualization. Note that this image is after the component cleaning process as well.

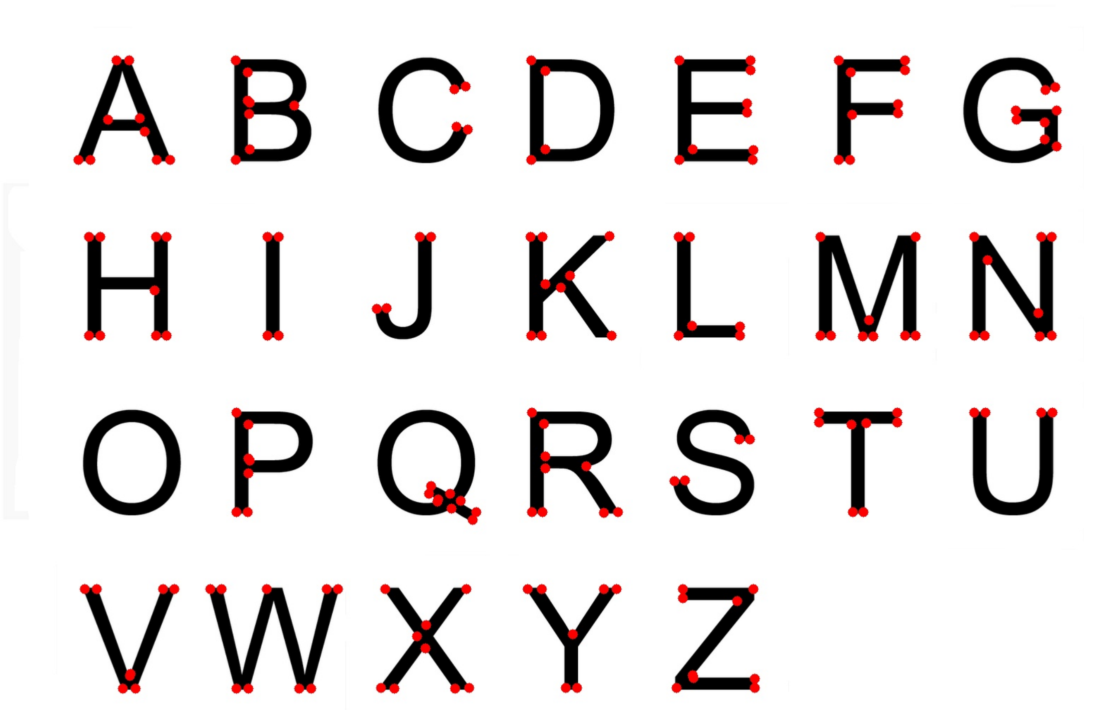

Figure 4: Harris corners (red circles) for each component. Note that some characters don't have enough corners. The number of corners per character will not exceed N = 9

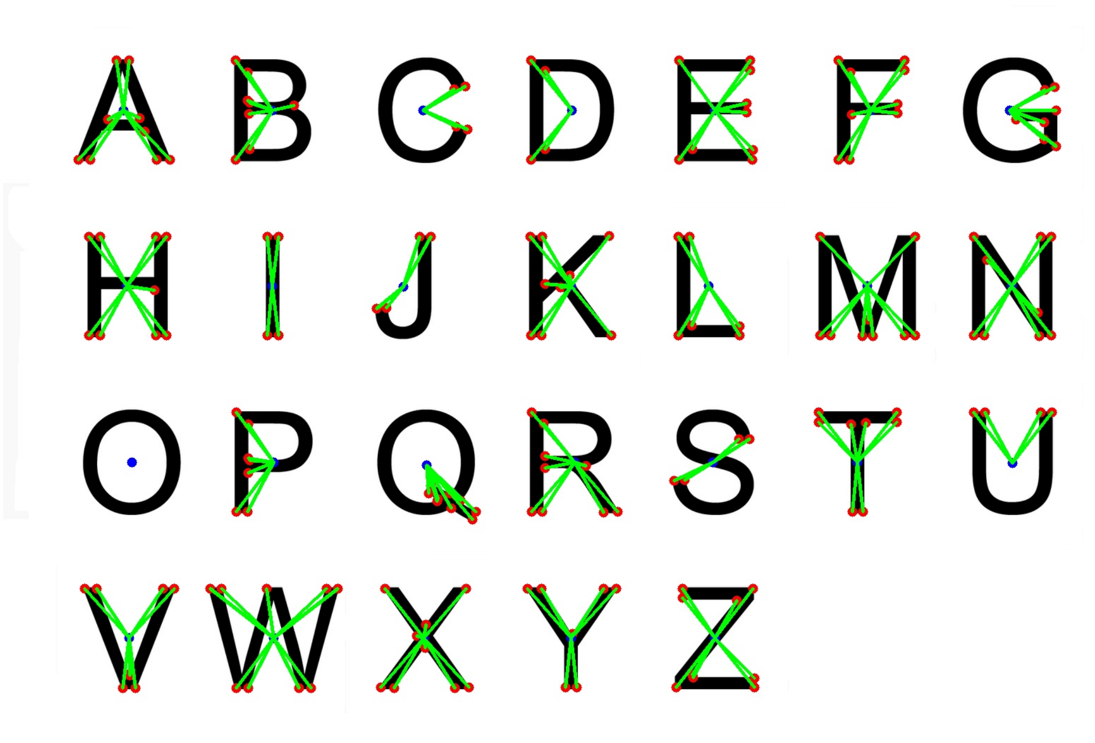

Figure 5: The shape vector of each component is the N-vector of angles between each successive pair of the N (green) lines that join the component cetner (blue) with the corners (red)

9.2 Test Image 1

Figure 6: The input test image

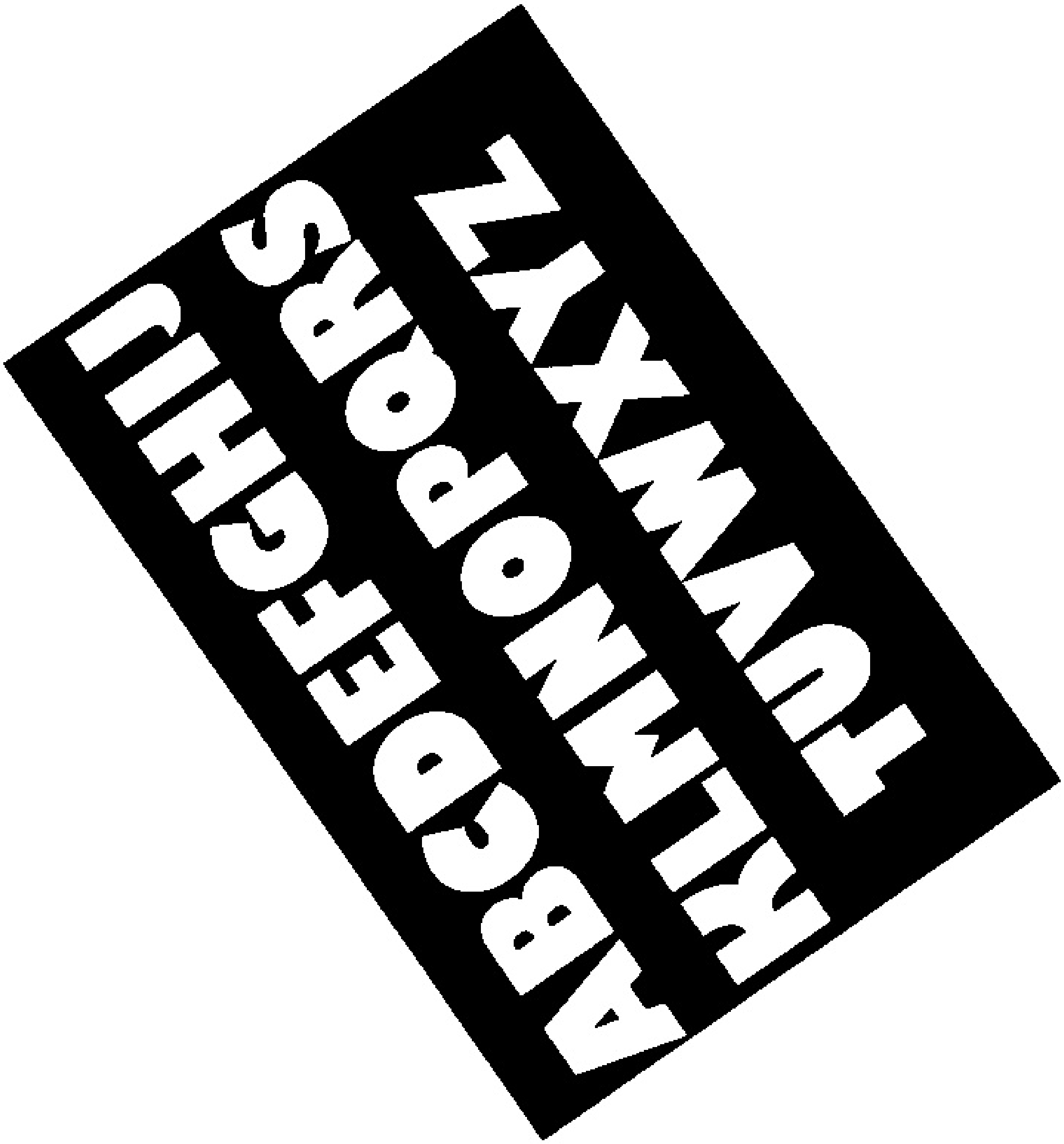

Figure 7: Otsu: the foreground mask of the image. White pixels are the foreground, while black pixels are the background.

Figure 8: Component labeling: an image representing the output of the component labeling process. Each unique label (component) is given a random color for visualization. Note that this image is after the component cleaning process as well.

Figure 9: Harris corners (red circles) for each component. Note that some characters don't have enough corners. The number of corners per character will not exceed N = 9

Figure 10: The shape vector of each component is the N-vector of angles between each successive pair of the N (green) lines that join the component cetner (blue) with the corners (red)

Figure 11: The output of the character recognition process. The overlaid blue characters are the output of the recognition process for the corresponding underlaid characters.

9.3 Test Image 2

Figure 12: The input test image

Figure 13: Otsu: the foreground mask of the image. White pixels are the foreground, while black pixels are the background.

Figure 14: Component labeling: an image representing the output of the component labeling process. Each unique label (component) is given a random color for visualization. Note that this image is after the component cleaning process as well.

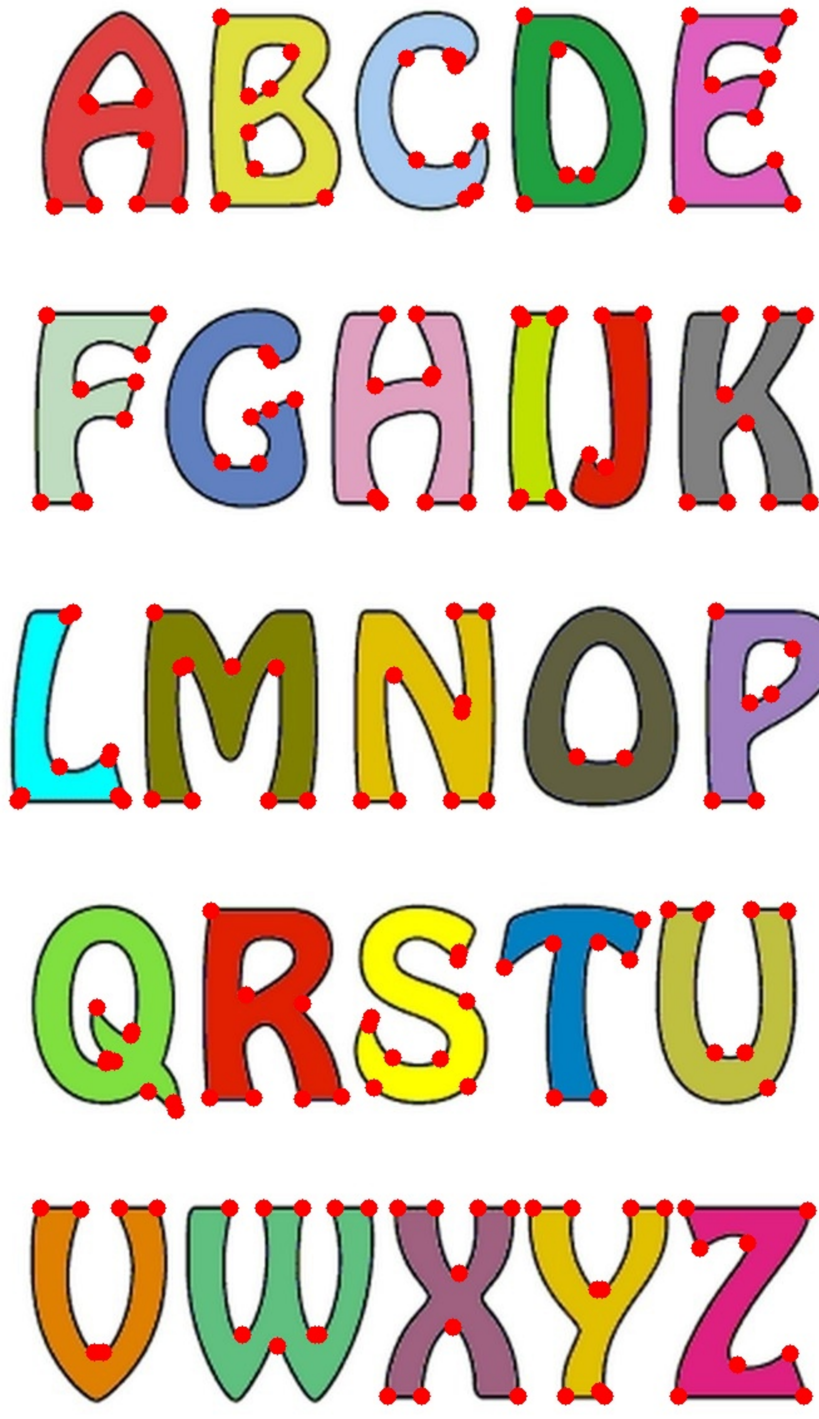

Figure 15: Harris corners (red circles) for each component. Note that some characters don't have enough corners. The number of corners per character will not exceed N = 9

Figure 16: The shape vector of each component is the N-vector of angles between each successive pair of the N (green) lines that join the component cetner (blue) with the corners (red)

Figure 17: The output of the character recognition process. The overlaid blue characters are the output of the recognition process for the corresponding underlaid characters.

9.4 Test Image 3

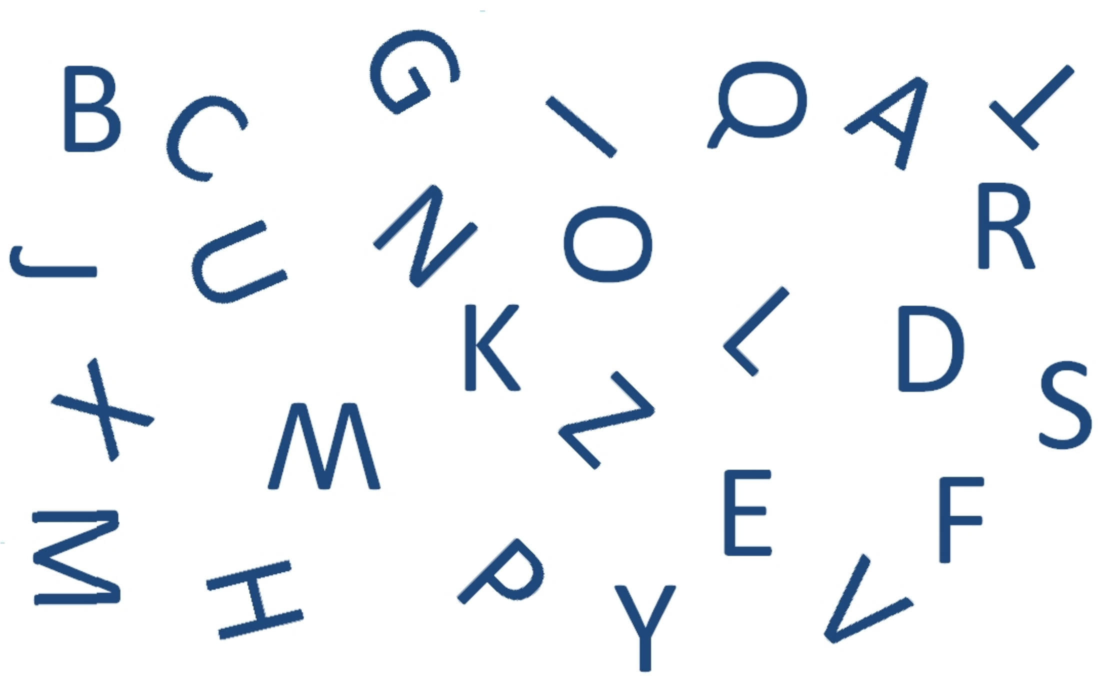

Figure 18: The input test image

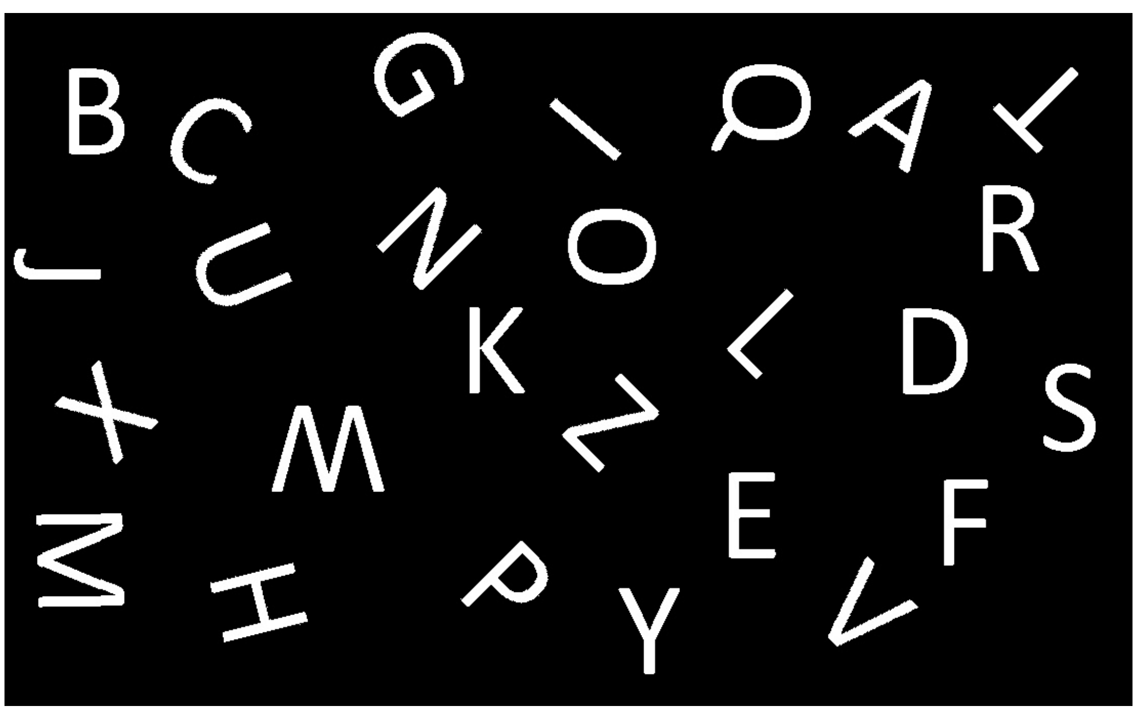

Figure 19: Otsu: the foreground mask of the image. White pixels are the foreground, while black pixels are the background.

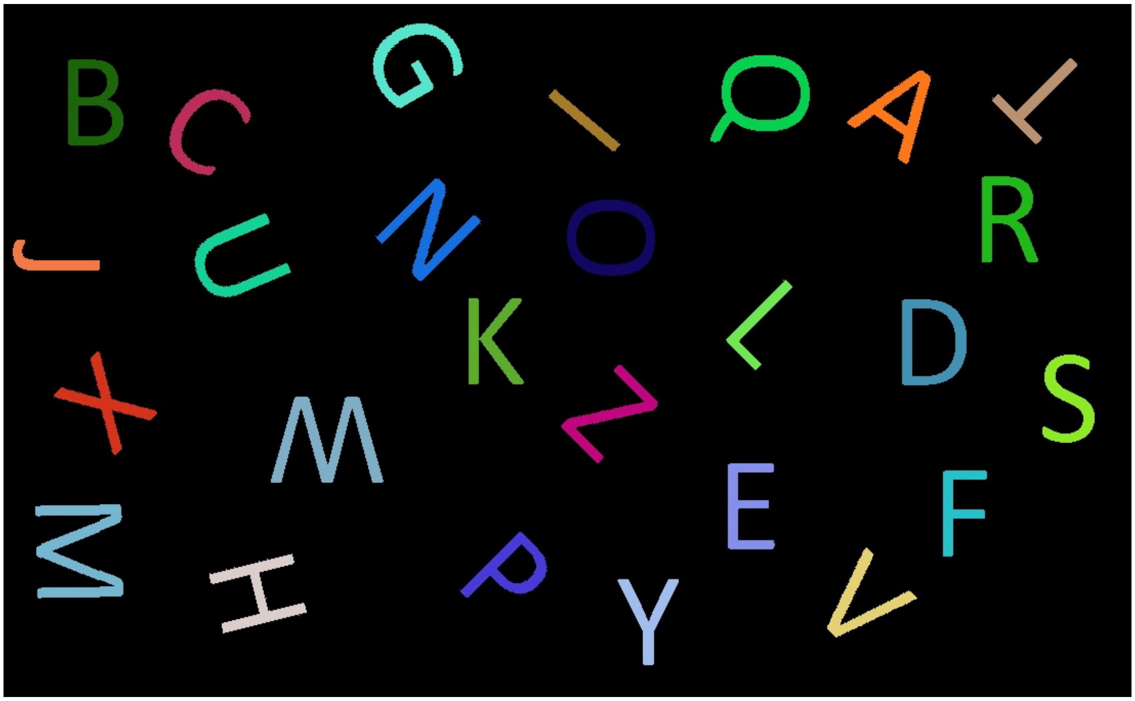

Figure 20: Component labeling: an image representing the output of the component labeling process. Each unique label (component) is given a random color for visualization. Note that this image is after the component cleaning process as well.

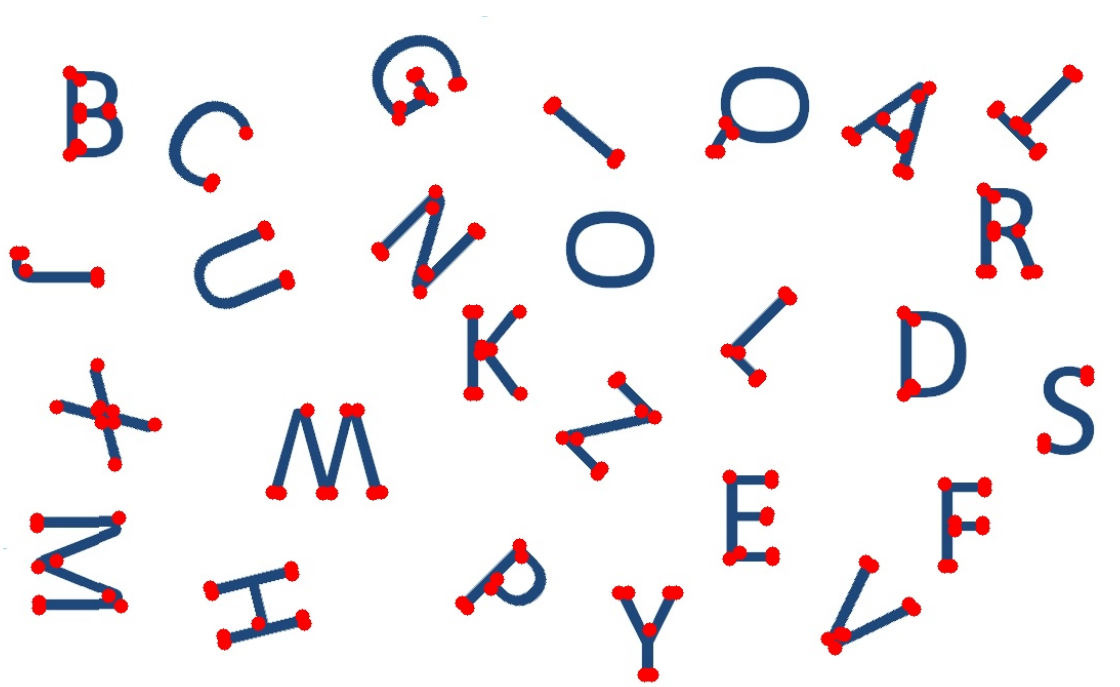

Figure 21: Harris corners (red circles) for each component. Note that some characters don't have enough corners. The number of corners per character will not exceed N = 9

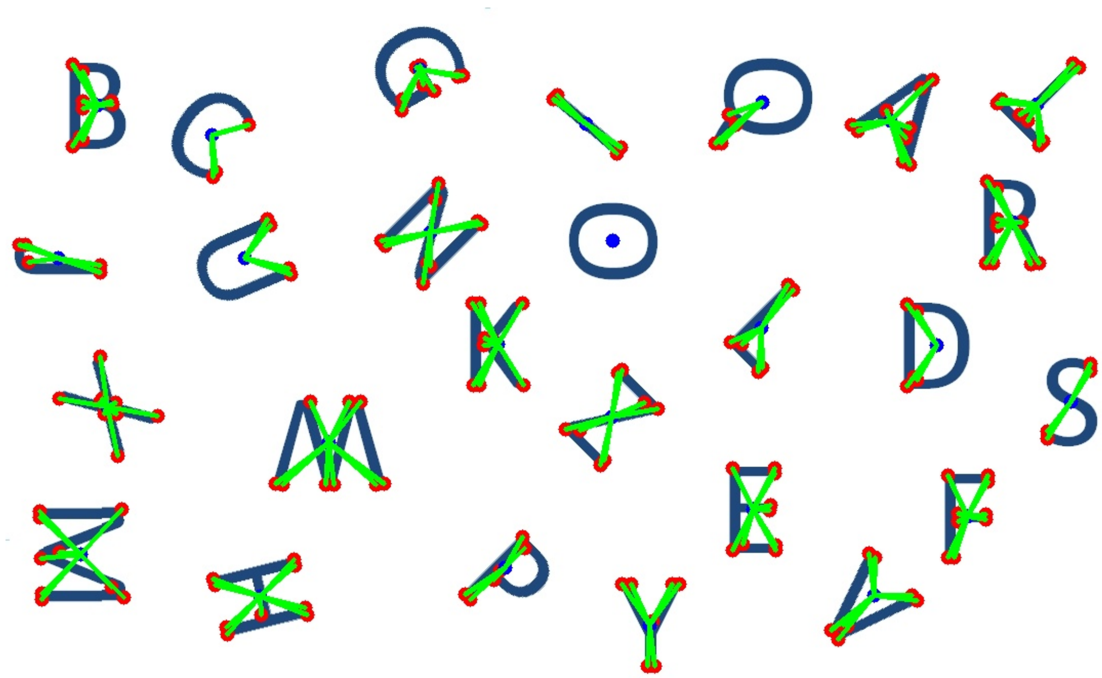

Figure 22: The shape vector of each component is the N-vector of angles between each successive pair of the N (green) lines that join the component cetner (blue) with the corners (red)

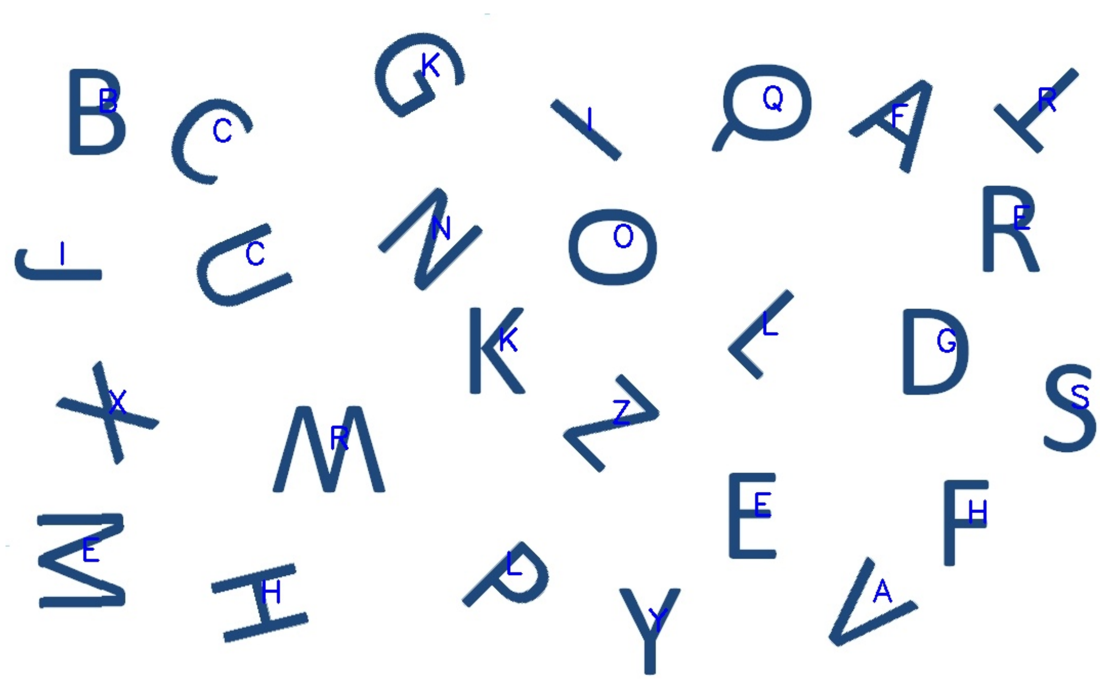

Figure 23: The output of the character recognition process. The overlaid blue characters are the output of the recognition process for the corresponding underlaid characters.

9.5 Test Image 4

Figure 24: The input test image

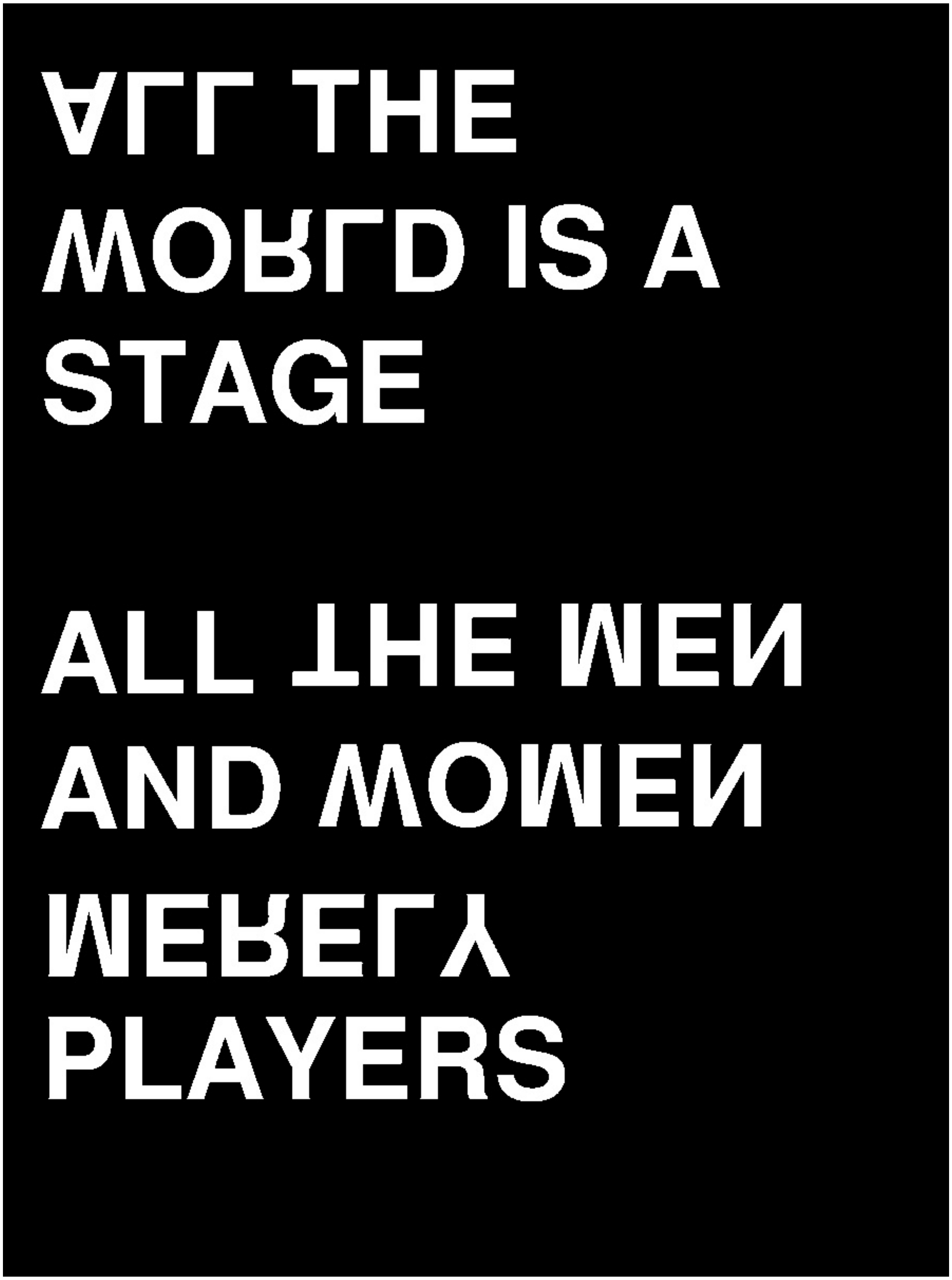

Figure 25: Otsu: the foreground mask of the image. White pixels are the foreground, while black pixels are the background.

Figure 26: Component labeling: an image representing the output of the component labeling process. Each unique label (component) is given a random color for visualization. Note that this image is after the component cleaning process as well.

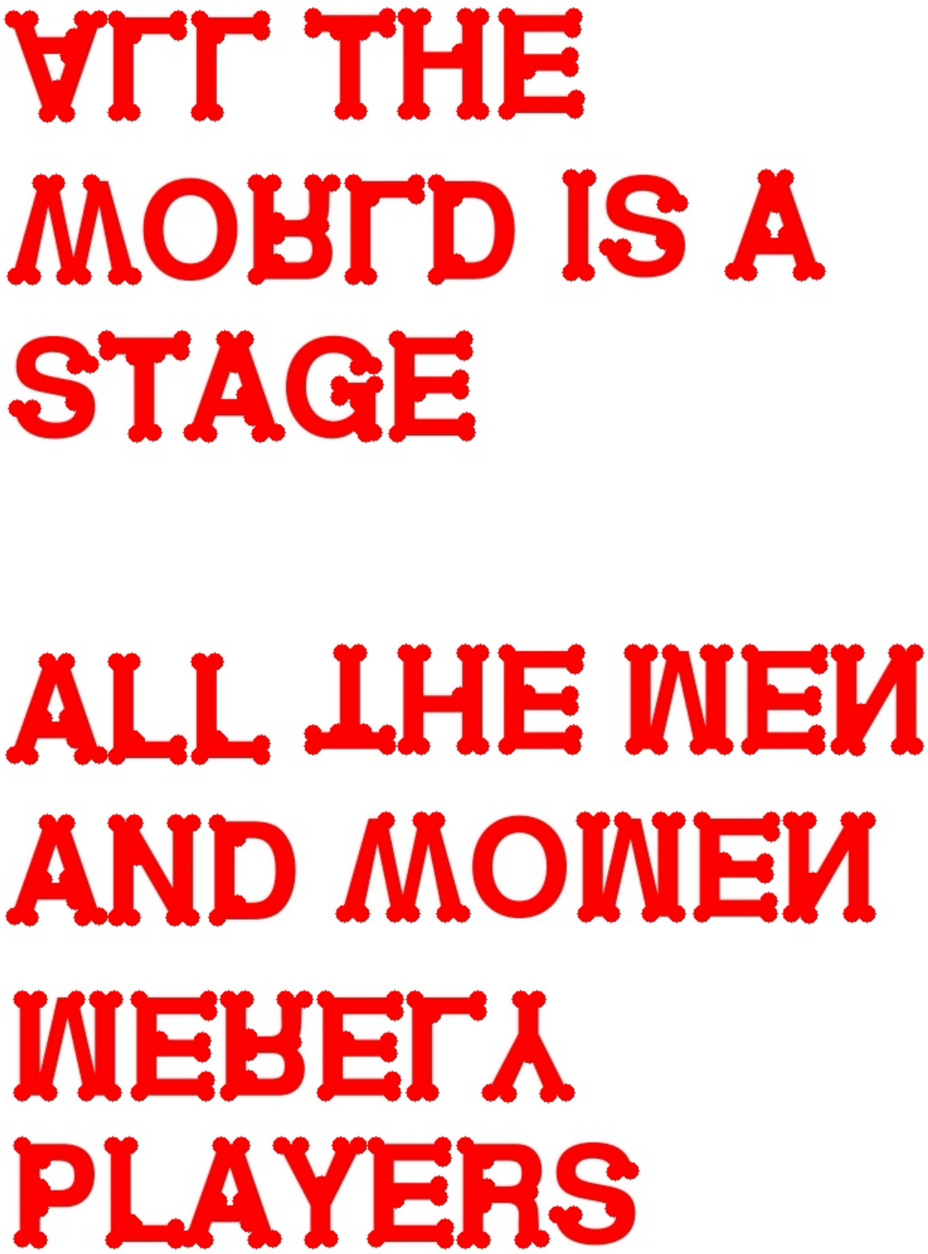

Figure 27: Harris corners (red circles) for each component. Note that some characters don't have enough corners. The number of corners per character will not exceed N = 9

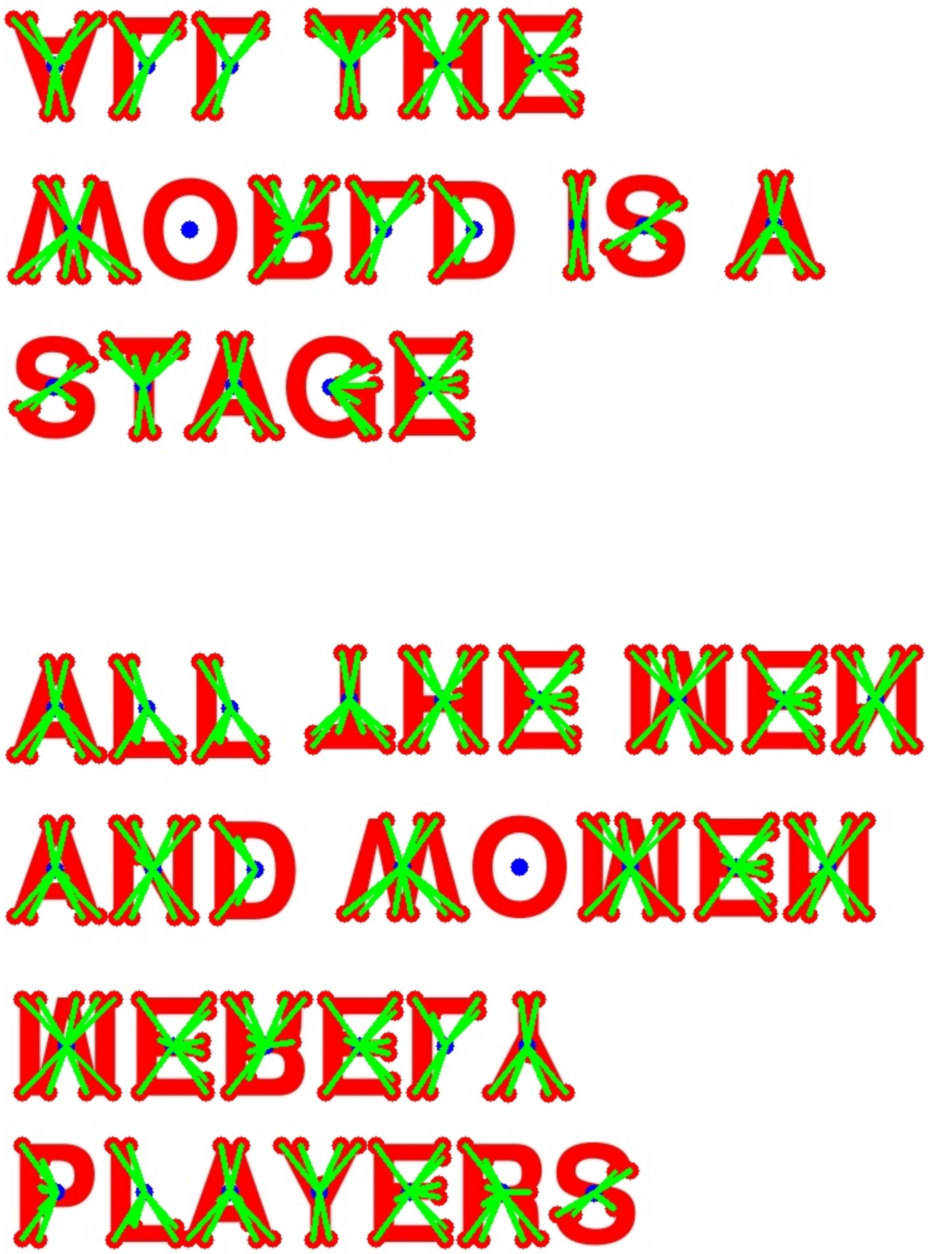

Figure 28: The shape vector of each component is the N-vector of angles between each successive pair of the N (green) lines that join the component cetner (blue) with the corners (red)

Figure 29: The output of the character recognition process. The overlaid blue characters are the output of the recognition process for the corresponding underlaid characters.

9.6 Test Image 5

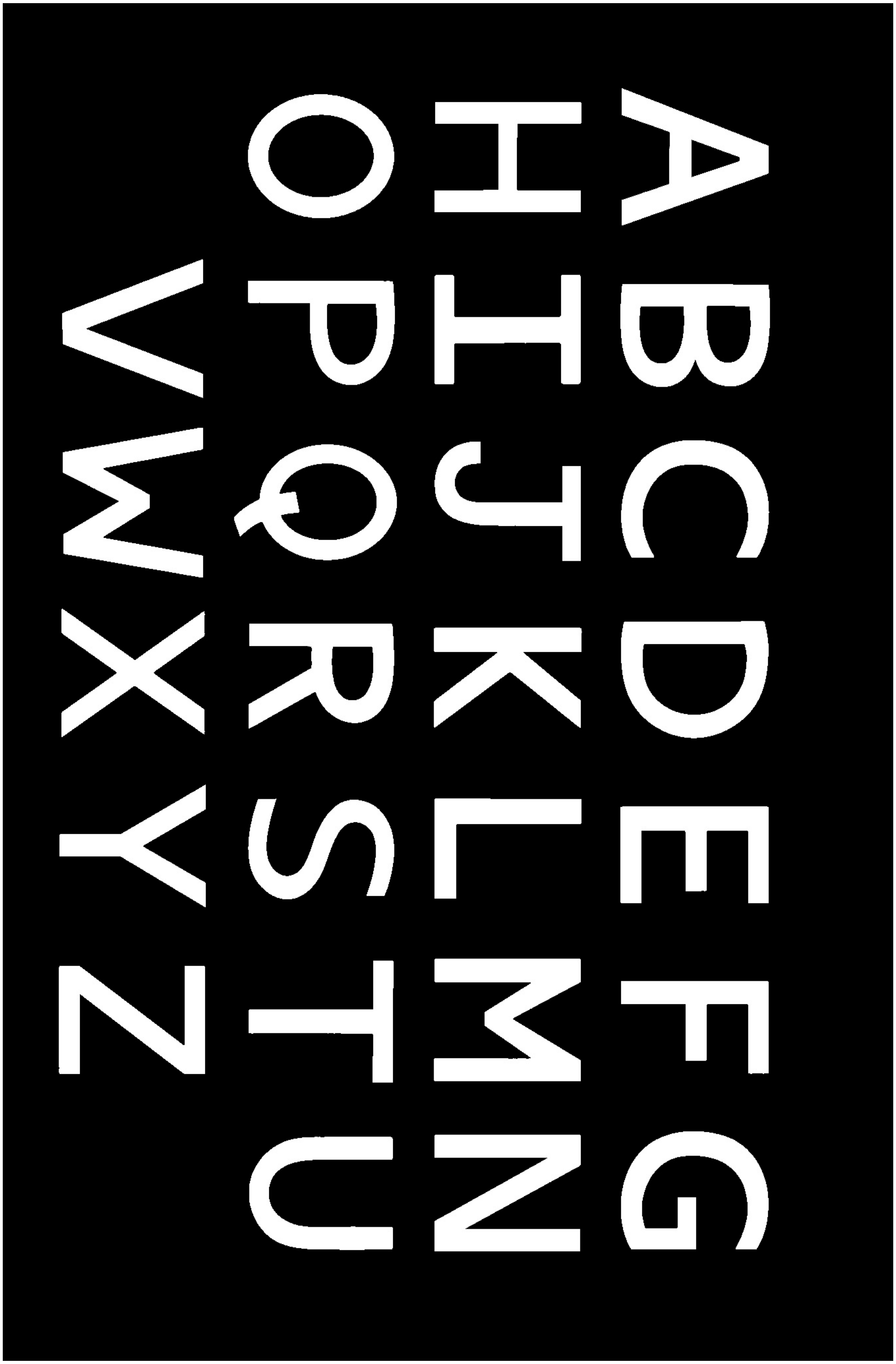

Figure 31: Otsu: the foreground mask of the image. White pixels are the foreground, while black pixels are the background.

Figure 32: Component labeling: an image representing the output of the component labeling process. Each unique label (component) is given a random color for visualization. Note that this image is after the component cleaning process as well.

Figure 33: Harris corners (red circles) for each component. Note that some characters don't have enough corners. The number of corners per character will not exceed N = 9

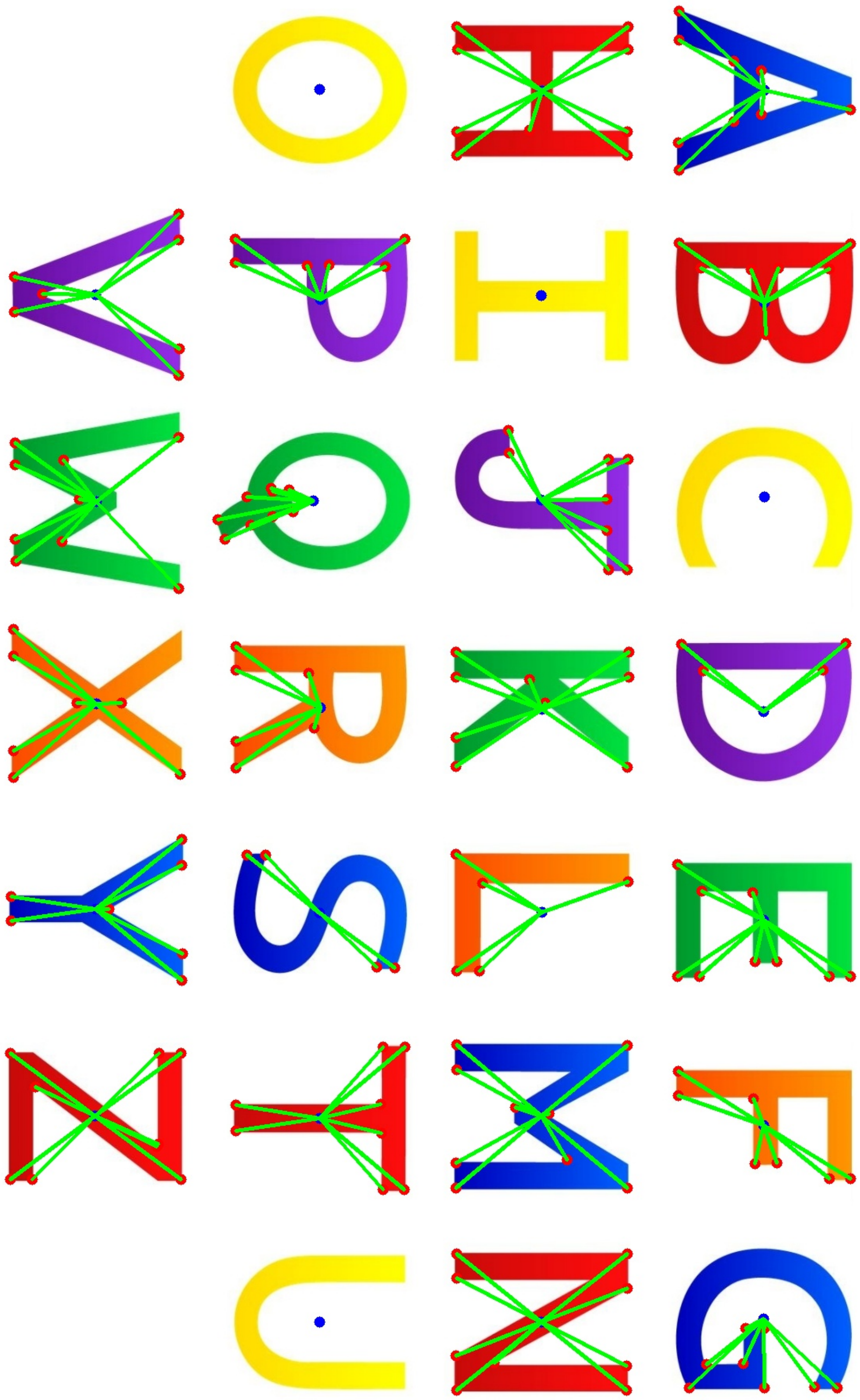

Figure 34: The shape vector of each component is the N-vector of angles between each successive pair of the N (green) lines that join the component cetner (blue) with the corners (red)

Figure 35: The output of the character recognition process. The overlaid blue characters are the output of the recognition process for the corresponding underlaid characters.

9.7 Test Image 6

Figure 36: The input test image

Figure 37: Otsu: the foreground mask of the image. White pixels are the foreground, while black pixels are the background.

Figure 38: Component labeling: an image representing the output of the component labeling process. Each unique label (component) is given a random color for visualization. Note that this image is after the component cleaning process as well.

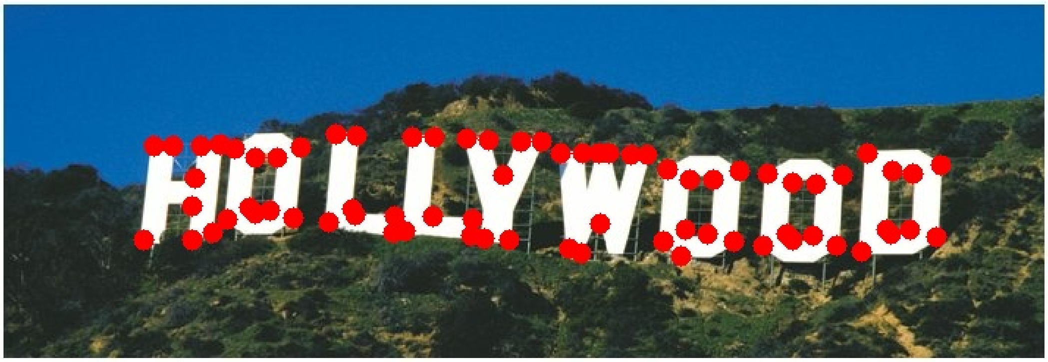

Figure 39: Harris corners (red circles) for each component. Note that some characters don't have enough corners. The number of corners per character will not exceed N = 9

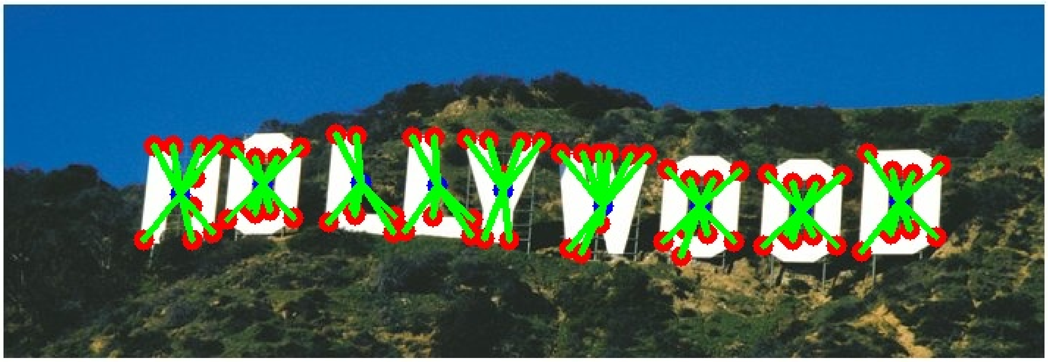

Figure 40: The shape vector of each component is the N-vector of angles between each successive pair of the N (green) lines that join the component cetner (blue) with the corners (red)

Figure 41: The output of the character recognition process. The overlaid blue characters are the output of the recognition process for the corresponding underlaid characters.

9.8 Statistics

	Image 1		Image 2		Image 3		Image 4		Image 5		Image 6		Overall		
	#	C	#	C	#	C	#	C	#	C	#	C	#	C	%
A	1	0	1	0	1	0	6	4	1	0	0	0	10	4	40.00%
B	1	0	1	0	1	1	0	0	1	0	0	0	4	1	25.00%
C	1	0	1	0	1	1	0	0	1	0	0	0	4	1	25.00%
D	1	0	1	0	1	0	2	2	1	1	1	0	7	3	42.86%
E	1	0	1	0	1	1	8	3	1	0	0	0	12	4	33.33%
F	1	0	1	0	1	0	0	0	1	0	0	0	4	0	0.00%
G	1	0	1	0	1	0	1	1	1	0	0	0	5	1	20.00%
H	1	0	1	0	1	1	2	1	1	1	1	0	7	3	42.86%
I	1	1	1	0	1	1	1	1	1	0	0	0	5	3	60.00%
J	1	0	1	0	1	0	0	0	1	0	0	0	4	0	0.00%
K	1	0	1	0	1	1	0	0	1	0	0	0	4	1	25.00%
L	1	0	1	0	1	1	7	0	1	1	2	1	13	3	23.08%
M	1	1	1	0	1	0	3	0	1	0	0	0	7	1	14.29%
N	1	0	1	0	1	1	3	1	1	1	0	0	7	3	42.86%
O	1	1	1	0	1	1	2	2	1	1	3	0	9	5	55.56%
P	1	0	1	0	1	0	1	1	1	1	0	0	5	2	40.00%
Q	1	0	1	0	1	1	0	0	1	1	0	0	4	2	50.00%
R	1	1	1	0	1	0	3	1	1	0	0	0	7	2	28.57%
S	1	0	1	0	1	1	3	0	1	0	0	0	7	1	14.29%
T	1	0	1	0	1	0	3	2	1	1	0	0	7	3	42.86%
U	1	0	1	0	1	0	0	0	1	0	0	0	4	0	0.00%
V	1	1	1	0	1	0	0	0	1	0	0	0	4	1	25.00%
W	1	0	1	0	1	0	2	0	1	0	1	0	7	0	0.00%
X	1	1	1	0	1	1	0	0	1	0	0	0	4	2	50.00%
Y	1	0	1	0	1	1	2	1	1	0	1	0	7	2	28.57%
Z	1	0	1	0	1	1	0	0	1	1	0	0	4	2	50.00%
Tot	26	6	26	0	26	14	49	20	26	9	9	1	162	50	
%	23.08%		0.00%		53.85%		40.82%		34.62%		11.11%		**30.86%**		

Figure 42: Statistics of the recognition Process WITHOUT the enhancement of the corners distances. The first column contains the letters. The second and third columns contain the actual number of characters in Image 1 and the correctly recognized letters respectively. Columns 4-13 are the same for all the other test images. Column 14 and 15 contain the actual number of characters in all the images and the correctly recognized letters respectively. Column 16 contains the overall recognition accuracy for each letter. The last two rows show the overall statistics for all letters in each image. The bold-ed number shows that the overall average recognition accuracy is 31correctly recognized out of 162 letters)

	Image 1		Image 2		Image 3		Image 4		Image 5		Image 6		Overall		
	#	C	#	C	#	C	#	C	#	C	#	C	#	C	%
A	1		1	1	1	1	6		1	1	0		10	3	30.00%
B	1		1		1	1	0		1		0		4	1	25.00%
C	1		1		1		0		1		0		4	0	0.00%
D	1		1		1		2	2	1	1	1		7	3	42.86%
E	1		1		1	1	8		1		0		12	1	8.33%
F	1		1		1		0		1		0		4	0	0.00%
G	1		1		1		1		1		0		5	0	0.00%
H	1		1		1	1	2	2	1	1	1		7	4	57.14%
I	1	1	1		1	1	1	1	1		0		5	3	60.00%
J	1		1	1	1		0		1		0		4	1	25.00%
K	1		1		1		0		1		0		4	0	0.00%
L	1	1	1		1	1	7	4	1	1	2	1	13	8	61.54%
M	1		1		1	1	3		1		0		7	1	14.29%
N	1		1		1		3		1		0		7	0	0.00%
O	1	1	1		1	1	2	2	1	1	3		9	5	55.56%
P	1		1		1		1	1	1		0		5	1	20.00%
Q	1		1	1	1		0		1	1	0		4	2	50.00%
R	1		1		1	1	3	1	1		0		7	2	28.57%
S	1		1		1		3	3	1		0		7	3	42.86%
T	1		1		1		3	2	1		0		7	2	28.57%
U	1	1	1		1		0		1		0		4	1	25.00%
V	1		1		1		0		1		0		4	0	0.00%
W	1		1	1	1	1	2	1	1		1		7	3	42.86%
X	1		1		1	1	0		1		0		4	1	25.00%
Y	1		1		1		2	1	1		1	1	7	2	28.57%
Z	1		1		1		0		1	1	0		4	1	25.00%
Tot	26	4	26	4	26	11	49	20	26	7	9	2	162	48	
%	15.38%		15.38%		42.31%		40.82%		26.92%		22.22%		**29.63%**		

Figure 43: Statistics of the recognition Process WITH the enhancement of the corners disntance. The first column contains the letters. The second and third columns contain the actual number of characters in Image 1 and the correctly recognized letters respectively. Columns 4-13 are the same for all the other test images. Column 14 and 15 contain the actual number of characters in all the images and the correctly recognized letters respectively. Column 16 contains the overall recognition accuracy for each letter. The last two rows show the overall statistics for all letters in each image. The bold-ed number shows that the overall average recognition accuracy is 31correctly recognized out of 162 letters)

10 Source Code

The following is the entire Python source code.

```
import sys
import cv2
import numpy as np
from math import atan2, pi

# Harris Corner Detection Threshold
HARRIS_THRESHOLD = 1e12

# The number of corners in one letter.
CORNERS_NUM = 9


def main():

    # The shape vectors of the training image.
    training_vectors = None

    # The order of the letters in the training image according to the labels
    # numbers.
    training_letters = 'CGABDEFHIJKLMNOQSPRTUVWXYZ'

    for i in xrange(7):

        print 'Image', i

        # The file name of the input image.
        file_name = 'images/{}.jpg'.format(i)

        # Read the input image.
        image = cv2.imread(file_name)

        image_clone = image.copy()

        # Segment using Otsu's algorithm on the image.
        if i != 6:
            mask = otsu_rgb(image, file_name, inverted_masks=[1, 1, 1],
                            is_and=0)
        else:
            mask = otsu_rgb(image, file_name, inverted_masks=[0, 0, 0])

        # Perform component labeling. Use four connectivity only for the
        # testing image number 1.
        labels_image = label_components(mask, four_connectivity=(i == 1))

        # Clean the components by removing the very large components.
        clean_components(labels_image)

        # Visualize the labels by assigning a random color to each label.
        show_labels(labels_image, file_name)

        # Convert the color image to grayscale.
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # Find the corners in the grayscale image using Harris corner detection.
        components_corners = harris(gray_image, labels_image)

        # Mark the corners in the first image.
        for component in components_corners:
            for corner in component:
                cv2.circle(image, (corner[0], corner[1]), 7, (0, 0, 255), -1)

        # Save the results.
        cv2.imshow(inject(file_name, 'corners'), image)
        cv2.imwrite(inject(file_name, 'corners'), image)
```

```python

        # Get the shape vectors.
        components_centers, components_vectors = get_shape_vectors(
            labels_image, components_corners)

        # The shape vectors of the first image, are the training vectors.
        if i == 0:
            training_vectors = components_vectors

        for j, center in enumerate(components_centers):
            cv2.circle(image, (center[0], center[1]), 7, (255, 0, 0), -1)

            for corner in components_corners[j]:
                cv2.line(image, (center[0], center[1]), (corner[0], corner[1]),
                         (0, 255, 0), 3)

        # Save the results.
        cv2.imshow(inject(file_name, 'features'), image)
        cv2.imwrite(inject(file_name, 'features'), image)

        if i == 0:
            # Print the letters on the image.
            for j, center in enumerate(components_centers):
                cv2.putText(image_clone, training_letters[j],
                            center, cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
        else:
            # Match the letters in the image with the training image.
            matchings = recognize_components(training_vectors,
                                             components_vectors)
            # Print the letters on the image.
            for j, center in enumerate(components_centers):
                cv2.putText(image_clone, training_letters[matchings[j]],
                            center, cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)

        # Save the results.
        cv2.imshow(inject(file_name, 'recognition'), image_clone)
        cv2.imwrite(inject(file_name, 'recognition'), image_clone)

    while not cv2.waitKey(50) & 0xFF == 27:
        pass
    cv2.destroyAllWindows()


def inject(image_name, suffix):
    return '{}_{}.{}'.format(image_name.split('.')[0], suffix,
                             image_name.split('.')[1])


def otsu_rgb(image, file_name, inverted_masks=[0, 0, 0], is_and=1,
             iterations=[1, 1, 1], org_image=None, s=''):

    # The original image is just used for the results.
    if org_image is None:
        org_image = image

    # Initialize the overall mask.
    overall_mask = np.zeros((image.shape[0], image.shape[1]), np.uint8)
    if is_and:
        overall_mask.fill(255)

    # For each channel in the three color channels:
    for c in xrange(3):

        # The image representing channel c.
        channel_image = np.zeros_like(image)
        channel_image[:, :, c] = image[:, :, c]

```

```
        # The mask of channel c.
        mask = None

        # For an arbitrary number of iterations: perform the segmentation
        # using Otsu's algorithm.
        for i in xrange(iterations[c]):
            # Perform the segmentation using Otsu's algorithm.
            mask = otsu(image[:, :, c], mask)

            """
            # Save the results.
            cv2.imwrite(inject(file_name, 'mask_{}_{}{}'.format(c, i, s)), mask)
            """

        # Invert the masks that are indicated in inverted_masks.
        if inverted_masks[c] == 1:
            mask = cv2.bitwise_not(mask)

        # Calculate the overall mask as the logical and/or of masks.
        if is_and:
            overall_mask = cv2.bitwise_and(overall_mask, mask)
        else:
            overall_mask = cv2.bitwise_or(overall_mask, mask)

    # Save the results.
    cv2.imwrite(inject(file_name, 'mask{}'.format(s)), overall_mask)

    return overall_mask


def otsu(image, mask=None):

    # The histogram of grayscale levels.
    histogram = [0] * 256

    # The total number of pixels in the mask.
    pixels_num = 0

    # The average grayscale value for the entire image (masked by the mask).
    mu_t = 0

    # Initialize the histogram based on the pixels of the image.
    for r in xrange(image.shape[0]):
        for c in xrange(image.shape[1]):
            if mask is None or mask[r][c] != 0:
                pixels_num += 1
                mu_t += image[r][c]
                histogram[image[r][c]] += 1

    # The average grayscale value for the entire image (masked by the mask).
    mu_t = float(mu_t) / pixels_num

    # The cumulative probability of pixels less than or equal level i.
    omega_i = 0

    # The cumulative average grayscale value less than or equal level i.
    mu_i = 0

    # The final chosen threshold.
    threshold = -1

    # The maximum sigma_b ^ 2 corresponding to the final chosen threshold.
    max_sigma_b = -1

    # For every grayscale level in the histogram:
    for i in xrange(256):

```

```
        # The number of pixels in the grayscale level i.
        n_i = histogram[i]

        # The probability of pixels in level i.
        p_i = n_i / float(pixels_num)

        # Update the cumulative probability of pixels less than or equal
        # level i, and the cumulative average grayscale value less than or
        # equal level i.
        omega_i += p_i
        mu_i += i * p_i

        # Ignore the very first levels and the very last levels that don't
        # contain any pixels. For these levels, sigma_b_i will cause division
        # by zero exception.
        if omega_i == 0 or omega_i == 1:
            continue

        # Update the between-class variance sigma_b ^ 2.
        sigma_b_i = (mu_t * omega_i - mu_i) ** 2 / (omega_i * (1 - omega_i))

        # Compare the between-class variance sigma_b ^ 2 to the maximum,
        # and update the best threshold.
        if sigma_b_i > max_sigma_b:
            threshold = i
            max_sigma_b = sigma_b_i

    # The image of the output mask.
    output_mask = np.zeros_like(image)

    if threshold == -1:
        return output_mask

    # For each pixel in the input image:
    for r in xrange(image.shape[0]):
        for c in xrange(image.shape[1]):
            # Set the corresponding output mask pixel to 1 if the pixel
            # values is greater than the threshold.
            if image[r][c] > threshold:
                output_mask[r][c] = 255

    return output_mask


def label_components(image, four_connectivity=False):

    # The output image of labels.
    labels_image = np.zeros(image.shape, np.uint16)

    # The next label to use.
    next_label = 1

    # The list of labels equivalence sets.
    equivalence_sets = []

    # The first pass: assign temporary labels, and record equivalences.
    # For each pixel in the image:
    for r in xrange(image.shape[0]):
        for c in xrange(image.shape[1]):

            # Process only non-zero pixels.
            if image[r][c] == 0:
                continue

            # The equivalence set of labels for this pixel.
            equivalence_set = set([])

```

```
            # If this pixel value is equal to the west pixel:
            if c - 1 >= 0 and image[r][c] == image[r][c - 1]:
                # Add the label to the equivalence set.
                equivalence_set.add(labels_image[r][c - 1])

            # If this pixel value is equal to the north pixel:
            if r - 1 >=0 and image[r][c] == image[r - 1][c]:
                # Add the label to the equivalence set.
                equivalence_set.add(labels_image[r - 1][c])

            # If it is 8-connectivity, check the north west and north east.
            if not four_connectivity:
                # If this pixel value is equal to the north west pixel:
                if (r - 1 >= 0 and c - 1 >=0 and
                        image[r][c] == image[r - 1][c - 1]):
                    # Add the label to the equivalence set.
                    equivalence_set.add(labels_image[r - 1][c - 1])

                # If this pixel value is equal to the north east pixel:
                if (r - 1 >=0 and c + 1 < image.shape[1] and
                        image[r][c] == image[r - 1][c + 1]):
                    # Add the label to the equivalence set.
                    equivalence_set.add(labels_image[r - 1][c + 1])

            # Check the number of labels in the equivalence set.
            if len(equivalence_set) == 0:
                # If no labels in the equivalence set, assign new label.
                label = next_label
                next_label += 1
            elif len(equivalence_set) == 1:
                # If only one label in the equivalence set, choose it.
                label = min(equivalence_set)
            else:
                # Choose the least label for the current pixel.
                label = min(equivalence_set)
                # If more than one label in the equivalence set:
                # For every set in the global list of equivalence sets.
                for i in xrange(len(equivalence_sets) - 1, -1, -1):

                    # If the current pixel equivalence set share any labels
                    # with the equivalence set in the list:
                    if bool(equivalence_set & equivalence_sets[i]):
                        # Merge the two into the pixel equivalence set.
                        equivalence_set |= equivalence_sets[i]
                        # Delete the merged set from the list of sets.
                        del equivalence_sets[i]

                # Add the pixel equivalence set to the list.
                equivalence_sets.append(equivalence_set)

            labels_image[r][c] = label

    # The second pass: resolve equivalences.
    # For each pixel in the image.
    for r in xrange(labels_image.shape[0]):
        for c in xrange(labels_image.shape[1]):

            # Process only non-zero labels.
            if labels_image[r][c] != 0:

                # The new label to be assigned after resolving equivalences.
                new_label = -1

                # If the pixel label belongs to one of the equivalence sets,
                # assign the least label from the equivalence set.
                for equivalence_set in equivalence_sets:
                    if labels_image[r][c] in equivalence_set:
```

```
                        new_label = min(equivalence_set)
                        break

                # Assign the final label in case of equivalences, otherwise,
                # leave the label as is.
                if new_label != -1:
                    labels_image[r][c] = new_label

    return labels_image


def clean_components(labels_image):

    # This method removes the components whose size is larger than 30000
    # pixel. These components are likely to be the background.

    # The frequencies of the labels in the image.
    labels_frequencies = np.bincount(labels_image.ravel())

    # The set of labels to remove.
    removed_labels = set([])

    # For each label, if its frequency exceeds 30000, add it to the set.
    for i in xrange(1, len(labels_frequencies)):
        if labels_frequencies[i] > 30000 or labels_frequencies[i] < 100:
            removed_labels.add(i)

    # For each pixel in the image.
    for r in xrange(labels_image.shape[0]):
        for c in xrange(labels_image.shape[1]):

            # If the label is one of the labels to remove, remove it.
            if labels_image[r][c] != 0 and labels_image[r][c] in removed_labels:
                labels_image[r][c] = 0

    # Then, the method assigned new labels so that they are continuous 1, 2, ...

    # The current unique labels (not continuous).
    labels = np.unique(labels_image)

    # For each pixel in the image.
    for r in xrange(labels_image.shape[0]):
        for c in xrange(labels_image.shape[1]):

            # Assign the index of the label. The indices are continuous.
            if labels_image[r][c] != 0:
                labels_image[r][c] = np.where(labels == labels_image[r][c])[0]


def show_labels(labels_image, file_name):

    # The image with the colored components
    labels_colors = np.zeros((labels_image.shape[0], labels_image.shape[1], 3),
                             np.uint8)

    # Get the unique labels.
    labels = np.unique(labels_image)
    print 'Components #', len(labels) - 1

    # Generate a list of random colors for the list of unique labels.
    random_colors = np.random.random_integers(255, size=(len(labels), 3))

    # For each pixel in the image.
    for r in xrange(1, labels_image.shape[0]):
        for c in xrange(1, labels_image.shape[1]):
            # Process only non-zero labels.
            if labels_image[r][c] != 0:
```

```
                # Color the label with the corresponding color.
                label_index = np.where(labels == labels_image[r][c])[0]
                labels_colors[r][c][:] = random_colors[label_index]

    # Save the results.
    cv2.imwrite(inject(file_name, 'labels'), labels_colors)
    cv2.imshow(inject(file_name, 'labels'), labels_colors)


def harris(image, labels_image, sigma=1.2):

    # The unique labels.
    labels = np.unique(labels_image)

    # The array for the corners of each component.
    components_corners = [[] for i in xrange(len(labels) - 1)]
    # The array of the corresponding ratios.
    corners_ratios = [[] for i in xrange(len(labels) - 1)]

    smoothed_image = cv2.GaussianBlur(image, (0, 0), sigma)

    integral_image = np.zeros(image.shape)
    for y in range(image.shape[0]):
        integral_image[y][0] = smoothed_image[y][0]
        for x in range(1, image.shape[1]):
            integral_image[y][x] = (smoothed_image[y][x] +
                                    integral_image[y][x - 1])

    for x in range(image.shape[1]):
        for y in range(1, image.shape[0]):
            integral_image[y][x] += integral_image[y - 1][x]

    haar_window = int(np.ceil(sigma * 4))
    haar_window += haar_window % 2

    dx = np.zeros(image.shape)
    dy = np.zeros(image.shape)
    for y in range(haar_window / 2, image.shape[0] - haar_window / 2):
        for x in range(haar_window / 2, image.shape[1] - haar_window / 2):
            dx[y][x] = get_dx(integral_image, x, y, haar_window)
            dy[y][x] = get_dy(integral_image, x, y, haar_window)

    harris_window = int(np.ceil(sigma * 5))
    harris_window -= (1 - harris_window % 2)

    corners = []
    ratio_image = np.zeros_like(dx)

    # For each pixel in the input image:
    for y in range(harris_window / 2, image.shape[0] - harris_window / 2):
        for x in range(harris_window / 2, image.shape[1] - harris_window / 2):

            # Don't process un-labeled pixels.
            if labels_image[y][x] == 0:
                continue

            # The summation of the sqr(x-derivative), sqr(y-derivative),
            # and x-derivative * y-derivative over the Harris window.
            dx2 = 0
            dxdy = 0
            dy2 = 0

            # For each pixel in the Harris window around the pixel (x,y).
            for j in range(y - harris_window / 2,
                           y + harris_window / 2 + 1):
                for i in range(x - harris_window / 2,
                               x + harris_window / 2 + 1):
```

```

                    # Get the x-derivative and y-derivative of the pixel (i,j).
                    dx_ij = dx[j][i]
                    dy_ij = dy[j][i]

                    # Add to the summation of the sqr(x-derivative),
                    # sqr(y-derivative), and x-derivative * y-derivative over
                    # the Harris window.
                    dx2 += np.square(dx_ij)
                    dxdy += dx_ij * dy_ij
                    dy2 += np.square(dy_ij)

            if dx2 == 0 and dy2 == 0 and dxdy == 0:
                continue

            dx2 /= harris_window
            dy2 /= harris_window
            dxdy /= harris_window

            # Compute the determinant and the trace of the C matrix at the
            # pixel (x,y)
            det_c = dx2 * dy2 - np.square(dxdy)
            tr_c = dx2 + dy2

            # Compute the ratio between the determinant and the trace squared.
            #ratio = det_c / np.square(tr_c)
            ratio = det_c - 0.04 * np.square(tr_c)
            ratio_image[y][x] = ratio

            if ratio >= HARRIS_THRESHOLD:
                components_corners[labels_image[y][x] - 1].append((x, y))
                corners_ratios[labels_image[y][x] - 1].append(ratio)

            """
            # If the ratio is above the HARRIS_THRESHOLD, the pixel is
            # considered a corner.
            if ratio >= HARRIS_THRESHOLD:
                corners.append((x, y))
            """


    for i in xrange(len(components_corners)):
        for j in xrange(len(components_corners[i]) - 1, -1, -1):

            if not is_corner(components_corners[i][j][0],
                             components_corners[i][j][1],
                             ratio_image, harris_window):
                del components_corners[i][j]
                del corners_ratios[i][j]

    for i in xrange(len(components_corners)):
        if len(components_corners[i]) < CORNERS_NUM:
            print len(components_corners[i])

        max_indices = sorted(range(len(corners_ratios[i])),
                             key=lambda x: corners_ratios[i][x])[-CORNERS_NUM:]
        components_corners[i] = [components_corners[i][j] for j in max_indices]

    return components_corners


def fast_convolute(integral_image, x1, y1, x2, y2):
    return integral_image[y2][x2] - integral_image[y2][x1 - 1] - \
           integral_image[y1 - 1][x2] + integral_image[y1 - 1][x1 - 1]


def get_dx(integral_image, x, y, window_size):
```

```
    half1 = fast_convolute(integral_image,
                           x, y - window_size / 2,
                           x + window_size / 2 - 1, y + window_size / 2 - 1)
    half2 = fast_convolute(integral_image,
                           x - window_size / 2, y - window_size / 2,
                           x - 1, y + window_size / 2 - 1)
    return half1 - half2


def get_dy(integral_image, x, y, window_size):

    half1 = fast_convolute(integral_image,
                           x - window_size / 2, y - window_size / 2 + 1,
                           x + window_size / 2 - 1, y)
    half2 = fast_convolute(integral_image,
                           x - window_size / 2, y + 1,
                           x + window_size / 2 - 1, y + window_size / 2)
    return half1 - half2


def is_corner(x, y, ratio_image, harris_window):

    # Get the ratio of the pixel at (x, y).
    ratio = ratio_image[y][x]

    # For each pixel in the Harris window around the pixel (x,y).
    for j in range(y - harris_window / 2,
                   y + harris_window / 2 + 1):
        for i in range(x - harris_window / 2,
                       x + harris_window / 2 + 1):

            # If the ratio of the pixel at (x, y) is less than one of its
            # neighbors, eliminate it from the corners.
            if ratio_image[j][i] > ratio:
                return False

    return True


def get_shape_vectors(labels_image, components_corners):

    # The unique labels.
    labels = np.unique(labels_image)[1:]

    # The array for the corners of each component.
    components_centers = []

    # The list of shape vectors of the components.
    components_vectors = [[] for i in xrange(len(labels))]

    # For each label:
    for i, label in enumerate(labels):
        # Find where the label is.
        py, px = np.where(labels_image == label)

        # Find the center of the component with that label.
        center_x = (np.max(px) + np.min(px)) / 2
        center_y = (np.max(py) + np.min(py)) / 2

        # Add the center to the array of centers.
        center = (center_x, center_y)
        components_centers.append(center)

        # The list of angles of the corners.
        angles = []

```

```python
        # Compute the angles of the corners,
        for corner in components_corners[i]:

            angle = atan2(corner[1] - center[1], corner[0] - center[0])
            angle = angle * 180 / pi
            if angle < 0:
                angle += 360

            angles.append(angle)

        # Sort the angles ascending.
        angles.sort()

        # Find the differences between the angles, which represent the arc
        # length between the corners on the unit circle.
        for j, angle in enumerate(angles):
            diff = angle - angles[j - 1]
            if diff < 0:
                diff += 360
            components_vectors[i].append(diff)

        if len(components_vectors[i]) < CORNERS_NUM:
            for j in xrange(CORNERS_NUM - len(components_vectors[i])):
                components_vectors[i].append(0.0)

    return components_centers, components_vectors


def recognize_components(training_vectors, components_vectors):

    # The list of the best matches.
    matches = []

    # For each component:
    for i, component_vector in enumerate(components_vectors):

        # The index of the best match.
        best_match = -1
        # The minimum distance to the best match.
        min_dist = sys.float_info.max

        # For each element in the shape vector:
        for j in xrange(len(component_vector)):

            # Circularly rotate the vector around index j.
            rotated_vector = component_vector[j:] + component_vector[:j]
            #print rotated_vector

            # For each training vector:
            for k, training_vector in enumerate(training_vectors):

                #print training_vector
                # Calculate the euclidean distance.
                dist = np.linalg.norm((np.array(rotated_vector) -
                                       np.array(training_vector)))

                # Keep the best match with its distance.
                if dist < min_dist:
                    min_dist = dist
                    best_match = k

        matches.append(best_match)

    return matches


if __name__ == "__main__":
```

```
    main()
```

Listing 1: The entire Python source code