Using OpenCV to Create Kanji Calligraphy Database

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A Thesis Presented To Eastern Washington University Cheney, Washington

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In Partial Fulfillment of the Requirements for the Master of Science in Computer Science

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

By Marco H. Karier

Spring 2018

THESIS OF MARCO KARIER APPROVED BY

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_DATE\_\_\_\_\_\_\_\_\_\_\_

Kosuke Imamura, CHAIR, GRADUATE STUDY COMMITTEE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_DATE\_\_\_\_\_\_\_\_\_\_\_

Shamima Yasmin, Member, GRADUATE STUDY COMMITTEE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_DATE\_\_\_\_\_\_\_\_\_\_\_

tbd, Member, GRADUATE STUDY COMMITTEE

Table of Contents

[Introduction 4](#_Toc515906841)

[Background 6](#_Toc515906842)

[Convolution 6](#_Toc515906843)

[Gaussian Blur 8](#_Toc515906844)

[Sobel Edge Detection 8](#_Toc515906845)

[Contour Detection 9](#_Toc515906846)

[Methods 11](#_Toc515906847)

[Results 12](#_Toc515906848)

[Conclusion 19](#_Toc515906849)

[References 20](#_Toc515906850)

Using OpenCV to Create Kanji Calligraphy Database

by Marco H. Karier

June 15, 2018

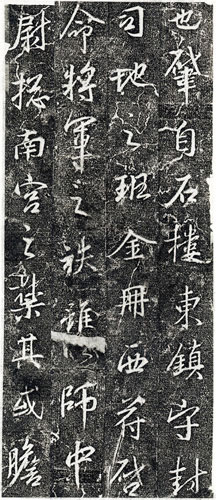
# Introduction

The goal of this project was to evaluate the effectiveness of a state of the art image recognition software, OpenCV, to identify and match particularly difficult images. For this purpose I created a database of the contours of Chu Suiliang’s 7th century calligraphy to test the accuracy of the “image compare” function in OpenCV. OpenCV is designed for image processing and utilizes more than 2,500 optimized algorithms[[1]](#footnote-1) to analyze images.

In order to evaluate this function, it was necessary to create a database of images that were capable of being read by the program. Each character had to be digitally smoothed and restored. The images were then processed by OpenCV, creating a database of this particular calligraphy. Finally, each image was input into the program and then compared to all 207 characters in the database. An exact match was identified by a moment of 0.0 and the degree of similarity with other characters was indicated by a positive number.

An additional outcome of this project is that it should help historians identify the valuable work of Wang Xizhi. The original work of calligraphy saint, Wang Xizhi, would have been lost forever if it wasn’t for the work of Tang dynasty calligrapher Chu Suiliang. The emperor of the Tang Dynasty requested Chu to carve a stone tablet or *stele* which became known as the “Ji Zi Sheng Jiao Xu” (672 A.D) in the style of the Wang Xizhi. To complete his task, Chu spent decades collecting works of Wang Xizhi to make sure that each calligraphy character was done in the same style. Work by Wang can still be found, but the only way to authenticate it is to compare it to Chu’s Chinese stele. The work described here allows a historian to use modern image recognition software to make this comparison efficiently and effectively.

Figure 1: Sample of Chu Suiliang’s Calligraphy



Other methods, such as neural networks, may in the future be able to improve on the matching capabilities documented in this research. However, they will need to demonstrate a better performance than what was found here using a popular image recognition software, OpenCV. At the time this paper was written OpenCV claims to have 47 thousand users and an estimated 14 million downloads that take advantage of its diverse image processing software.

In the next section I describe details about particular functions in OpenCV that were used in this research. The description focuses on the underlying algorithms that are used by each particular function. Next I describe the steps taken using OpenCV functions to clean the images, find the contours, save the information in a database, and then evaluate the matching function which is named *matchShapes()*. In the confirmation process, I present the results of using each of the polished images as an input to look for matches in the complete database. This is essentially what a historian could do with a new character to compare the degree of similarity between the character and Chu Suiliang’s set of calligraphy. However, because I use actual characters from the database there is one perfect match for each character with itself. At the end I include some comments and suggestions for further work.

# Background

OpenCV uses 2,500 algorithms but for my purposes I will focus on a few particularly high-level concepts in order to provide an understanding of how the program processes and matches images. These high-level concepts include convolution, Sobel edge detection, Gaussian blur, and contour detection. These concepts are explained in the following sections.

# Convolution

The basis of many of the OpenCV algorithms is convolution which is best defined as a mathematical operation that uses one function to modify an original function into a third function. OpenCV uses a function defined as a “kernel” to make these modifications. There are different types of convolutions and they are distinguished by different kernels which are represented as fixed size arrays or matrices of numerical coefficients.

OpenCV documentation is a particularly good source for the forms of convolution recommended for use in the program. For example, I used a common 3x3 kernel for two functions, one to blur the image and the other to detect edges (Sobel edge detection). Because these were relatively small images, it was appropriate to use a small (3x3) kernel.

This task is accomplished by sweeping through the image and reducing values to the left of the center and increasing values to the right. For example it is possible to apply the convolution of a 1dimensional array containing [10,10,10,12,17,19,21,21,21] with a 1-dimensional kernel [1,2,1] and for alpha values equal to 0.2 and 0.4. The alpha is used to change the value of the pixels after convolution. Lower values of alpha produce more subtle changes from the original array.

Using 0 array based indexing, the pseudo code for the middle part of the convolution will look like Equation 1.

Equation 1: representation of pseudo-code to perform a convolution.

OpenCV will loop through each possible index as represented in Equation 1. Each convolution index calculation includes division by the sum of the kernel elements. This sum is modified for the first and last entry of the array because they lack a match for all three elements in the kernel which causes the program to crash. For this reason the edge values are divided by 3 instead of 4.

To get the base convolution value you subtract the original from the convolution and then multiply by the alpha. This is the basis of many of the OpenCV’s functions that I have implemented.

Table 1 Steps to Perform Convolution

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Original | 10 | 10 | 10 | 12 | 17 | 19 | 21 | 21 | 21 |
| Convolution | 30/3=10 | 40/4=10 | 42/4=10.5 | 51/4=12.75 | 65/4=16.25 | 76/4=19 | 82/4=20.5 | 84/4=21 | 63/3=21 |
| Original – Convolution | 0 | 0 | -0.5 | -0.75 | 0.75 | 0 | 0.5 | 0 | 0 |
| Alpha 0.2 | | | | | | | | | |
| (Original -Convolution) \* Alpha | 0 | 0 | -0.1 | -1.5 | 1.5 | 0 | 0.1 | 0 | 0 |
| Added to Original | 10 | 10 | 9.9 | 8.5 | 18.5 | 19 | 21.1 | 21 | 21 |
| Alpha 0.4 | | | | | | | | | |
| (Original - Convolution) \* Alpha | 0 | 0 | -0.2 | -0.3 | 0.3 | 0 | 0.2 | 0 | 0 |
| Added to Original | 10 | 10 | 9.8 | 11.7 | 17.3 | 19 | 21.2 | 21 | 21 |

The first row in Table 1 contains the original array values. The second row contains the values after convolution, and the other rows apply different values of alpha to derive a final array.

# Gaussian Blur

A Gaussian blur kernel is designed to give more weight to the pixels in the center, or anchor point, of the image. The pixels furthest from the anchor point have less weight compared to pixels that are closer. Applying a Gaussian blur removes some of the noise around the Chinese Characters and allows for more accurate Contour detection.

I use this operation to blur the calligraphy to try and smooth out the edges of each character. Figure 2 shows a character before it is run through the Gaussian Blur. The spaces around the main shape make it difficult for OpenCV to detect an edge. Figure 3 shows the same image but after it has run through a Gaussian Blur. Notice that the edges are smoother after the Gaussian. A blurred image removes the extra noise and improves OpenCV’s edge detection and Contour Sobel Edge Detection.





Figure 2 Before the Gaussian Blur

Figure 3 After multiple Gaussian Blur

# Sobel Edge Detection

Sobel is a powerful convolution filter that is used to detect edges. Like all convolution operations it has a unique kernel. A 3x3 Scharr kernal for a Sobel edge detection is illustrated in Equation 2 below. In Equation 2, ‘A’ is every x and y coordinate of the 2-dimensional image source.

Equation 2: Sobel 3x3 Kernel[[2]](#footnote-2)

and

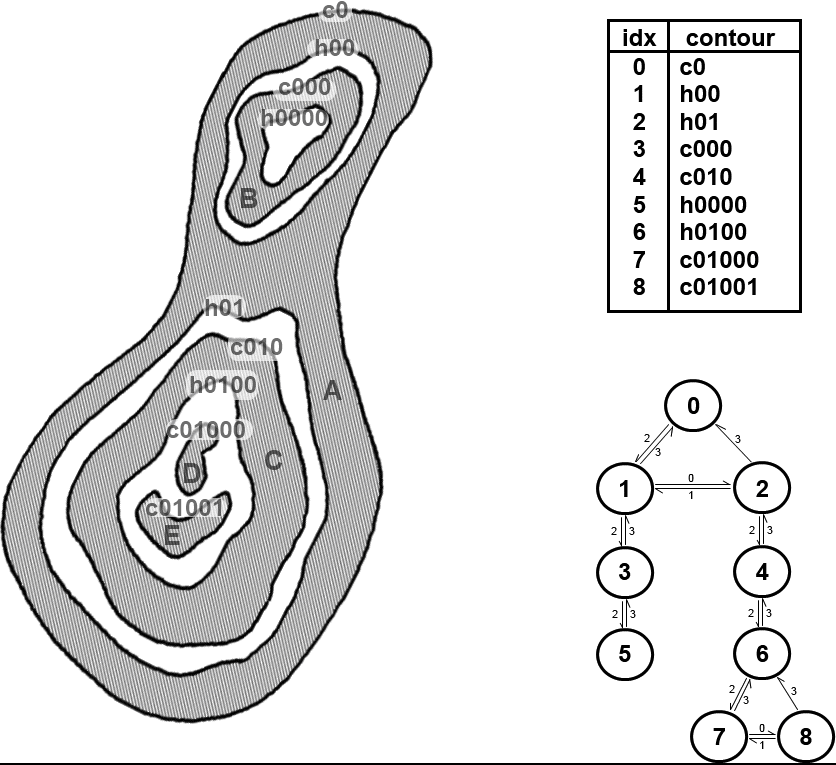
The computer detects an edge by finding the gradient of pixels across the edge, for example a change from dark to light. By applying the Sobel convolution the gradient of an edge becomes more apparent. The pixels that constitute an edge are identified and recorded.

# Contour Detection

A contour is just the outer shape of an image. Contour detection uses Sobel edge detection but also records characteristics of edges such as curves or angles. OpenCV stores a vector of points for each pixel location on an image essentially compiling a list of every edge. OpenCV looks for specific characteristics of edges such as whether the edges form an enclosed shape without any overlap.

OpenCV labels these contours with either cX or hX. “cX” means a contour which is usually the outside shape, and “hX” means a hole referring to an interior part of the image that might have white space[[3]](#footnote-3). X in this case is used to represent a number which describes the hierarchy of the different contours. For example Figure 4 shows how each image can have different layers of contours.

Figure 4 Image of Layers of contours[[4]](#footnote-4)



These contours are what OpenCV uses to perform the matchShape function. Even though OpenCV can use many contour layers, the characters used in this study are defined only by a single outer contour.

To be able to compare two different images, OpenCV uses the significant information about each edge to create a list[[5]](#footnote-5). Significant information about the contour are called contour moments. Contour moments represent high level characteristics of a contour. The formula that OpenCV uses to calculate contour moments is in Equation 3.

Equation 3: Formula for Image Moments[[6]](#footnote-6)

Where is defined as the contour moment and I(x,y) is the value of the pixel. In this case, is “the sum over all the pixels in the object in which the value at x,y is multiplied by

According to the OpenCV tutorial, “if the image is a binary image (i.e., one in which every pixel is either 0 or 1), then is just the area of the nonzero pixels in the image. In the case of a contour, the result is the length of the contour, and in the case of a point set it is just the number of points.[[7]](#footnote-8)”

There are 3 different ways for OpenCV to compare the different contour moments which are summarized in Table 2.

Table 2 Matching Methods used by cv::matchShapes()[[8]](#footnote-9)

|  |  |
| --- | --- |
| Value of method | cv::matchShapes() return value |
| cv:: CONTOURS\_MATCH\_I1 |  |
| cv::CONTOURS\_MATCH\_I2 |  |
| cv::CONTOURS\_MATCH\_I3 |  |

OpenCV uses these contour moments to compare the shapes of each contour and determine if it is a similar image. It uses this compare method and returns a number that indicates a degree of similarity. A value of 0 indicates an exact match and is what I expected when using the program to search for the original image. Users are able to set a maximum level for these numbers when looking for close matches.

# Methods

This research was designed to determine if OpenCV could create a useful database of 7th Century calligraphy and determine if new characters were similar to those in the database. In order to evaluate additional images, one of the requirements was that the program should be able to find the original input image in the database as an exact match. It was also expected to produce a number identifying the degree of similarity to other characters.

The first step was to isolate each calligraphy character into its own image file. I then cleaned them up using different convolution methods described above, including Gaussian and Sobel convolutions to remove the noise and restore edges. I grouped significant contours together and removed contours that were clearly noise and not part of the original image. After this was completed, the final contours were saved in the database.

In the example in Figure 5 the character included additional contours that appeared to be noise rather than part of the original image. In this case I removed all but two of the contours, assuming the others represented noise.

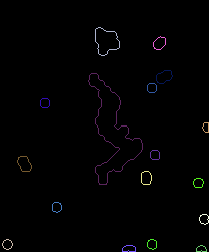


Figure 5: Noisy Image Contour

Finally, I used the matchShapes() function to compare each finished image with the entire database. My expectation was that each file would have an exact match with itself indicated by a value of 0.0. Also, low numbers should be returned for similar images and high numbers for dissimilar images.

# Results

Initially I had difficulty getting the matchShapes() function to work properly with the calligraphy characters. In order to isolate the problem, I used a smaller data set of characters from the English alphabet. This helped because English letters generally have a single contour with the exception of “i” and “j”. As it turned out, these were the only two letters which OpenCV had difficulty matching. Instead of identifying that “i” and “i” were identical it would return a high number indicating a poor match. This was essentially due to the fact that it was comparing the two initial contours which make up the “i”, a circle and a line, to two other contours, also a circle and a line. The circles matched but they did not match with the lines resulting in a high number indicating a poor match. This required a small adjustment in the program to count only one comparison per contour, i.e. the one with the lowest value or highest match. This successfully resolved the problem.

The important output from matchShapes() is the number indicating a degree of similarity. I used this number to create a matrix of results. Table 3 represents an illustrative subset of the matrix. As mentioned earlier, an exact match is indicated by a value of 0.0 which is the value of the diagonal in the matrix. This confirmed that the program was able to accurately find each character in the database.

Table 3: A Section of the Result Matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Names | scan1.png | scan2.png | scan3.png | scan4.png |
| scan1.png | 0 | 0.697062 | 0.152959 | 0.372741 |
| scan2.png | 0.697062 | 0 | 0.404698 | 0.243374 |
| scan3.png | 0.152959 | 0.404698 | 0 | 0.431593 |
| scan4.png | 0.372741 | 0.243374 | 0.431593 | 0 |

Each row represents an image tested against each character in the columns of the database. For example, scan1 is compared to all other images in the first row. This shows a value of 0.0 for itself and a value of 0.697062 for similarity to scan2. Alternatively, when scan2 is compared to scan1 in the second row it returns the same value, 0.697062. This is expected because comparison of scan1 to scan2 should produce the same result as comparing scan2 to scan1. Even though this is obvious, this was not true for some of the original runs. Revising the operations and improving the implementation of the program did ultimately resolve this issue.

Of more importance is the ability of OpenCV to find similar and dissimilar characters. Towards this end, I analyzed the matrix to identify the smallest values other than zero to verify that these images were in fact similar. I also looked at the largest values to verify that these images were in fact dissimilar.

Initially, some of the low-value matches did not appear to be similar. I found that this was a problem with incomplete contours which produced unusually low numbers. An example of an incomplete contour is presented in Figure 6. Additional work to improve the contours resolved this problem.

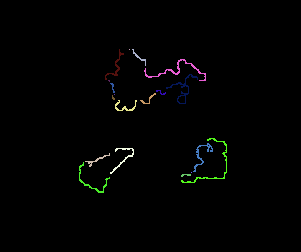


Figure 6: Incomplete Contours

Because there are 207 characters there are 42,642 matches in the complete matrix, not counting the matches to themselves. Because half of these are duplicates, (i.e. scan1 to scan2 is the same as scan2 to scan1) there are 21,321 unique matches. I then found the seven lowest values (other than 0.0) and those are presented below in Table 4.

Table 4: the 3 closest matches (other than a perfect zero match)

|  |  |  |
| --- | --- | --- |
| Scan192 & scan187: 0.0154983 |  |  |
| scan155 & scan189:  0.01878 |  |  |
| Scan189 & scan194:  0.023818 |  |  |
| Scan37 &  Scan 107:  0.030625 |  |  |
| Scan197 &  Scan26  0.032658 |  |  |
| Scan37 &  Scan99:  0.032942 |  |  |
| Scan40 &  Scan20:  0.034481 |  |  |
| Scan8 &  Scan18:  0.036186 |  |  |

The first three matches in Figure 4 (scan192/187, 155/189, and 189/194) are obviously strong because they involve the same characters that were entered as different images from the original stele. They are only slightly different in size and shape because it is impossible to carve stone characters with exact precision. Each of these matches were particularly good because they had the lowest values, less than 0.025.

The next four matches in Figure 4 are ranked as the next best matches but can be seen to be more dissimilar. They have values of 0.03 and higher.

I also sorted the matrix to find the largest numbers indicating the highest degree of dissimilarity and the three matches with the highest numbers are presented in Table 5. Visual inspection confirms that the images in Table 5 are not particularly good matches and they have values greater than 12.

Table 5: The most dissimilar comparison

|  |  |  |
| --- | --- | --- |
| scan103 & scan166:  12.7103 |  |  |
| scan103 & scan156:  12.4833 |  |  |
| scan103 & scan170:  12.3566 |  |  |

Finally, I conducted some additional tests to explore the possibility of trying to match new characters with those in the database. I started with two images based on scan 155 which is depicted in Table 4. The results of this test are included in Table 6.

Table 6: Tests with two new characters, Traced 155 and Sketched 155

|  |  |  |
| --- | --- | --- |
| Original Image of 155 |  | |
| Traced scan155 Figure &  Scan155:  0.0300612 |  |  |
| Sketched scan155 Figure  &  Scan155:  0.220922 |  |  |

The first new image was developed by simply tracing the contour of scan 155 with a piece of paper placed over the computer screen. This assured the production of an almost exact replica of the image in the database. When this was matched against all of the images in the database it found a good match with scan 155, as expected, with a relatively low value of 0.0300. However, this was only the second-best match for the tracing. The best match was found to be with scan 194 which is also depicted in Table 4 because it is very similar to scan 155. The value of the match with the tracing and scan 194 was 0.0245.

The second new image was simply a smooth sketch of the original image from 155, without the bumpy contour lines. Unfortunately, this image was not a particularly strong match with scan 155. The value in this case was relatively high, 0.2209 and there were in fact 15 other images which were determined to be better matches. Even the variations of scan 155 (scan 194 and scan 189) identified in Table 4 were not strong matches. When matched with the sketch, they produced even higher values indicating a worse match.

This, probably more than any other test, shows the limitations of using OpenCV to identify new characters. Because the contours created by OpenCV still contain a lot of noise they will be difficult to match with new characters, especially if those images are based on characters with smoother curves.

# Conclusion

I was successful in my efforts to create a searchable database of a particular 7th Century calligraphy. This required scanning and processing the original characters to create a set of useable individual images. I confirmed this by separately comparing each of the images to the entire database. Exact matches were confirmed with a value of 0.0, good matches were less than 0.03, and poor matches ranged as high as 12.7.

This research also confirmed the value of using version 3.4.1 of OpenCV for both cleaning up the images through convolution and edge detection and ultimately matching images using the matchShapes() function.

In the process I learned several things about the program that were essential for successful application. For example, I had to make adjustments when comparing characters with more than one contour. Without this adjustment, even perfect matches will fail to produce a 0.0 value. I also learned that images in the database were not useful for searches if the contour lines were not continuous. If contours are discontinuous, or broken, the results are not reliable.

I also found it was essential to remove the noise from each scan. While the program was able to automatically remove some of the noise, this was limited to particularly small shapes. I had to manually delete some of the larger shapes that appeared to be noise. This represents a limit on the performance of the program. As is true for using any complex computer program, it is always helpful to understand the basic algorithms in order to use the program properly and interpret the results correctly.

This research suggests that OpenCV and the database created can be used to verify if additional characters are similar to or derived from Chu Suiliang’s 7th Century Caligraphy. I worked with 207 images but there are thousands more. Although OpenCV provides a simple and efficient way to search the characters in Chu Suilangs original stele, the research shows that there are limitations because of the difficulty of removing all of the noise in the contours. More specifically, contours with bumpy curves based on noise do not match well with similar shapes constructed of smooth curves.

There may be better matching routines but OpenCV sets a certain standard that can be compared to other methods. OpenCV simply sets a baseline that other routines would need to surpass.

# References

Bradski, Gary and Adrian Kaehler. *Learning OpenCV*. O'Reilly, 2008.

Hu, Ming-Kuei. "Visual Pattern Recognition by Moment Invariants." *IRE Transactions on Information Theory* 8.2 (1962): 179-187.

Kaehler, Adrian and Gary Bradski. *Learning OpenCV3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, Inc., 2017.

Puzicha, Jan, Jitendra Malik and Serge Belongie. "Shape Matching and Object Recognition Using Shape Contexts." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.24 (2002): 509-588.

Suzuki, Satoshi and Keiichi Abe. "New Fusion Operations for Digitized Binary Images." *IEEE* (1985): 638-651.

team, OpenCV. *https://opencv.org/about.html*. 28 05 2018. <https://opencv.org/about.html>.

Teh, Cho-Huak and Roland T. Chin. "On the Detection of Dominant Point on Digital Curves." *IEEE Transactions on Pattern Anaylsis and Machine Intelligence* 11.8 (1989): 859-872.

1. (team) [↑](#footnote-ref-1)
2. (Kaehler and Bradski) [↑](#footnote-ref-2)
3. (Suzuki and Abe) [↑](#footnote-ref-3)
4. (Kaehler and Bradski) [↑](#footnote-ref-4)
5. (Teh and Chin) [↑](#footnote-ref-5)
6. (Kaehler and Bradski) [↑](#footnote-ref-6)
7. (Kaehler and Bradski) [↑](#footnote-ref-8)
8. (Kaehler and Bradski) [↑](#footnote-ref-9)