Problem 1

Homework required one to design an algorithm to recognize the provided grayscale CAPTCHA based on provided training set. Three separate stages can be deduced

1. Training set feature extraction.
2. CAPTCHA character isolation.
3. Character recognition.

In the following sections, I will walk through the trial-and-error process I had gone through.

Trial 1

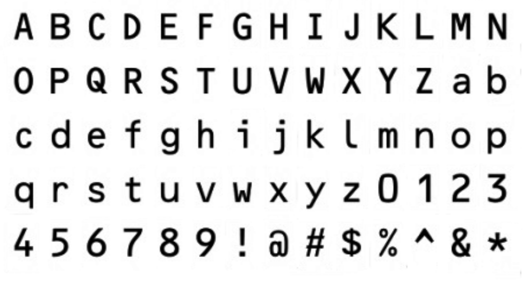
Training set feature extraction

We are interested in the characters themselves, a mask with ones on the characters is closer to expectation. Trivial observation indicates no significant noise exists, and gray level is marginal to non-existence along character edges, we can try to threshold it directly by averaging global minimum and maximum.

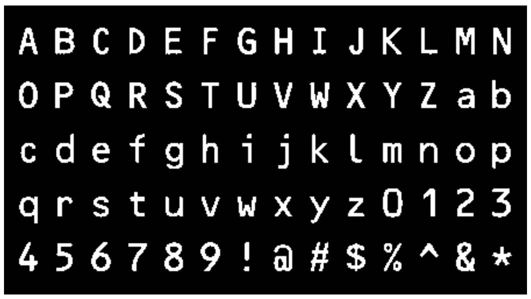
$$T = \sum_i \sum_j \frac{max(I(i,j)) - min(I(i,j))}{2} \tag{1}$$

Due to the fact that `TrainingSet.raw` is provided as black-on-white style, thresholding criteria is reversed.

$$I(i,j) = \begin{cases} 1, I(i,j) < T \\ 0, I(i,j) \geq T \end{cases} \tag{2}$$


A B C D E F G H I J K L M N
O P Q R S T U V W X Y Z a b
c d e f g h i j k l m n o p
q r s t u v w x y z 0 1 2 3
4 5 6 7 8 9 ! @ # $ % ^ & *

(a) Grayscale


A B C D E F G H I J K L M N
O P Q R S T U V W X Y Z a b
c d e f g h i j k l m n o p
q r s t u v w x y z 0 1 2 3
4 5 6 7 8 9 ! @ # $ % ^ & *

(b) Binary

Figure 1: Pre-processed training set

The training set characters are displayed in montage, we need to isolate them to individual characters before proceeding to next step. The montage contains 14-by-5 characters, with consistence spacing between horizontal and vertical neighbors. We can easily calculate the appropriate spacing to separate them. Margins of the image can causes misalignment after cropping, margin size of 5 and 7 pixels are empirically chose for horizontal and vertical edges respectively.

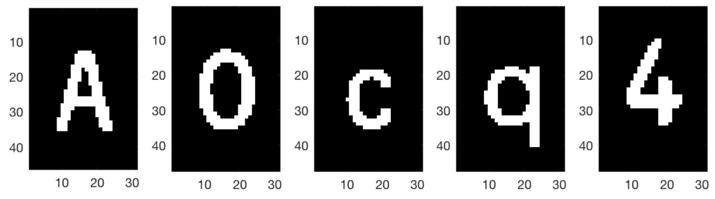


Figure 2: Cropped characters, first column is shown.

To perform further processing, I choose to resize them to image of size 32-by-32 pixels to ease the process.

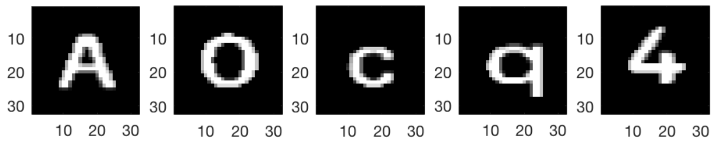
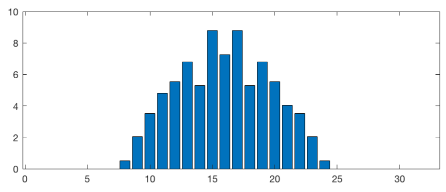
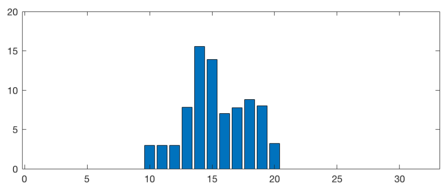


Figure 3: Resized characters to square, first column is shown.

Characters seem to have enough differences from scanline perspective, therefore, I project sum each character along horizontal and vertical direction to derive their 2-by-32 feature vector in Figure 4.



(a) X direction

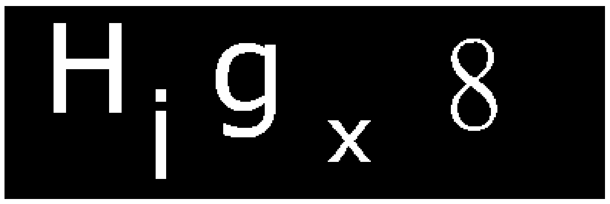


(b) Y direction

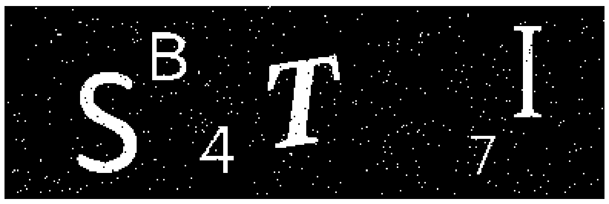
Figure 4: Cumulative sum projection of A

CAPTCHA Character isolation

Both sample images are easy to process as the training set.



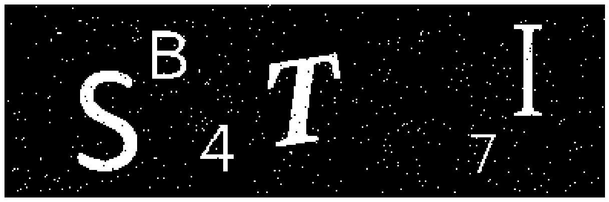
(a) Sample 1



(b) Sample 2

Figure 5: Binary sample images

Sadly, `sample2.raw` contains unwanted speckle noises, visualized in Figure 5b. Therefore, median filter is applied specially for it. Kernel size of 3 is arbitrarily chosen by trials.


(a) Before

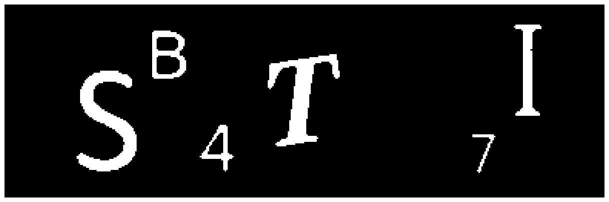
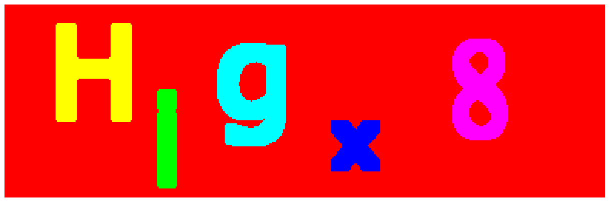

(b) After

Figure 6: Median filter for `sample2.raw`

For efficient labeling of connected components, disjoint-set algorithm based on handouts of Duke University *CS100e: Program Design and Analysis II* (`https://www2.cs.duke.edu/courses/cps100e/fall09/notes/UnionFind.pdf`) is used. However, characters such as `i` will be labeled as separate components, hence, dilation is first performed prior to the labeling, and masked out dilated regions later on. Dilation kernel size is empirically chosen to be 7. 8-connected component is used.


(a) Regions dilated

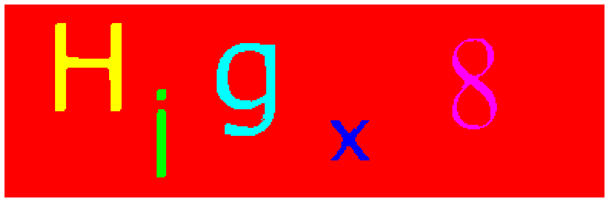

(b) Masked result

Figure 7: Character labeling (`sample1.raw`)

Using labels, we can crop out each characters and sort them according to their horizontal position, implicit requirement for English reading convention.

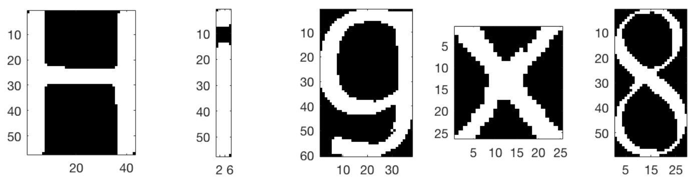
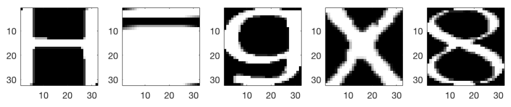


Figure 8: Cropped characters

At last, characters are resized to 32-by-32 squares as well, same goes for `sample2.raw`.


(a) `sample1.raw`

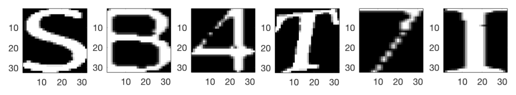

(b) `sample2.raw`

Figure 9: Resized characters

Character recognition

Directly subtract the differences between projection sum, I can retrieve the difference between curves, and use it as an index to find out which character is the most similar.

$$\begin{cases} d_x = \sum |\vec{c_x} - \vec{t_x}|^2 \\ d_y = \sum |\vec{c_y} - \vec{t_y}|^2 \end{cases} \tag{3}$$

where $\vec{c}$ is the feature character of CAPTCHA characters, while $\vec{t}$ is the feature character of training template. d_x and d_y are scalars. All the vectors are normalized prior to operations. A unified distance metric is used

$$d = \sqrt{d_x^2 + d_y^2} \tag{4}$$

Evaluation

100% error on both image.

Trial 2

Training set feature extraction

After reviewing the source code, I realize training set did not stretch the characters. Therefore

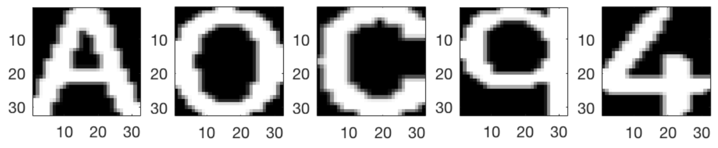


Figure 10: Stretched

Evaluation

90% average error rate. Only `H` is correctly recognized in `sample1.raw`. Apparently, direct curve comparison is not feasible.

Trial 3

Character recognition

If we accumulate the projection result again, then the normalization to acquire the distance metric is essentially calculating the cumulative distribution function for pixel intensity along different direction (horizontal and vertical). By treating the problem as a binary classifier system, we can utilize the receiver operating characteristic (ROC) curve, but instead of favoring true positive direction, which requires maximizing the ROC curve area over $y = x$ line, we try to *minimize* that area, since same character should not deviate its distribution along any axis.

Feature vectors along both axis are now treated as sample points along X and Y direction, which are linearly interpolated to uniform X grid of size 32. The rationale of 32 is due to the resampled image size, applied earlier, is 32-by-32.

We can formulate the calculation of ROC area as

$$A_{ROC} = \sum | \tag{5}$$

Boundary extraction

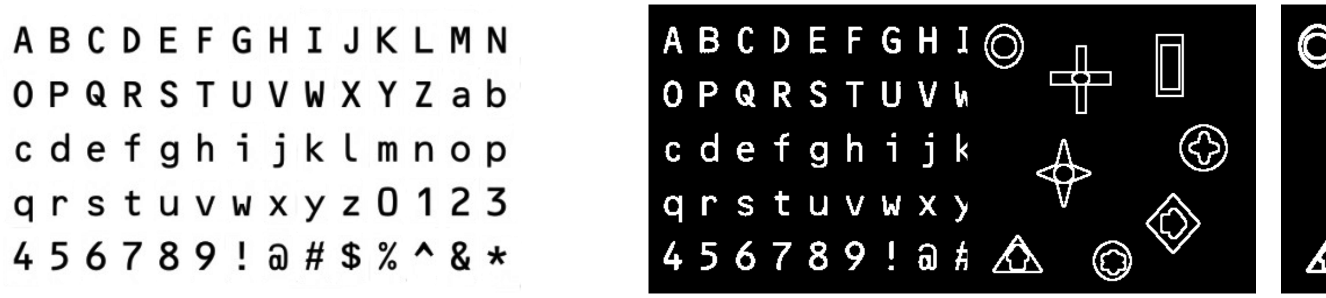

(a) Grayscale (b) Binary (c) 5×5

Figure 11: B

Boundary extraction is composed of erosion and its complement

$$B = I_1 \wedge \sim (I_1 \ominus SE) \tag{6}$$

where SE is the structural element, square in this case. When the sides of SE increases, the more the erosion is applied on I_1, causing more blanked region appeared after & $(I_1 \ominus SE)$, leading to thicker edges.

Skeletonizing

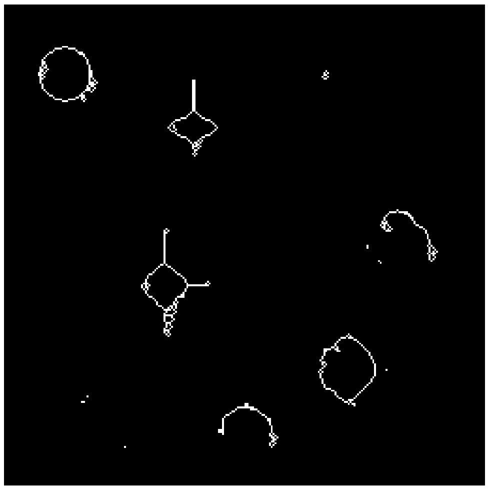


Figure 12: S, skeletonized I_1

The implementation try to follow the implementation from textbook, which requires two set of lookup tables. General operation is

$$G = X \cap [\sim M \cup P] \tag{7}$$

where X is the input image, G is the output image, M is the conditional marker determined in the process, and P is the inhibiting variable.

During the first pass, M is determined by hit-and-miss transformation using the kernel from Table 1. I put tons of time to figure out the decimal equivalent of them, I might as well put my result here, and there seems to be no one posting this out there on the Net.

Table 1: Conditional

Type	Decimal number
S1	64, 16, 4, 1
S2	128, 32, 8, 2
S3	192, 96, 48, 24, 12, 6, 3, 129
TK4	160, 40, 10, 130
STK4	193, 112, 28, 7
ST5	176, 161, 104, 194, 224, 56, 14 ,131
ST6	177, 108
STK6	240, 225, 120, 60, 15, 135, 195
STK7	241, 124, 31, 199
STK8	227, 248, 62, 143
STK9	243, 231, 252, 249, 124, 63, 159, 207
STK10	247, 253, 127, 223
K11	251, 254, 191, 239

During the second pass, which is coined as the unconditional stage, output result is determined by combination of X and M. Since there are spaces for combinations in the textbook kernel list, I separated them to three categories to ease the computation process.

First type is simply equal-or-not comparison with values from Table 2. One simply extract the surrounding pixels

$$V = \sum_{i \in \text{neighbors}} M_i 2^i \tag{8}$$

where V is the corresponding decimal value constructed by the fully connected neighbors. Most significant bit is the first pixel, designated M_0 in the textbook.

Table 2: Unconditional, EQ

Type	Decimal number
Spur	1, 4, 64, 16
4-connected	2, 128, 8, 32
L	160, 40, 130, 10

Second type requires one to provide a mask before comparison. The comparison is composed of two parts

$$\begin{cases} P_1 = (X \vee K_M) \wedge K_C \\ P_2 = (T \wedge \sim K_M) \wedge (K_C \wedge \sim K_M) \end{cases} \tag{9}$$

and the final result is determined by

$$P = P_1 \wedge P_2 \tag{10}$$

P_1 used the mask to ignore flexible D terms in the table provided in the textbook, while P_2 verifies the constant terms, M, 0 and 1 are satisfied as well. K_M is the mask while K_C is the conditional equivalent decimal number, both of them are enlisted as a pair (K_M, K_C) in Table 3.

Table 3: Unconditional, OR-EQ

Type	(Mask, Condition)
Corner	(31, 255), (241, 255), (199, 255), (124, 255)
Tee	(84, 252), (213, 255), (117, 255), (93, 255)
Diagonal	(17, 181), (68, 109), (17, 91), (68, 214)

Third type is similar to the second type, but instead of equivalent, it requires unequal, since in the textbook,

$$A \cup B \cup C = 1 \tag{11}$$

in other words, one of them has to be 1, complementary is relatively easy to test for.

Under this consensus, Equation 12 can be modified as

$$\begin{cases} P_1 = \sim ((X \vee K_M) \vee K_C) \\ P_2 = (T \wedge \sim K_M) \wedge (K_C \wedge \sim K_M) \end{cases} \tag{12}$$

Table 4: Unconditional, OR-NEQ

Type	(Mask, Condition)
Vee	(184, 248), (42, 62), (138, 143), (162, 227)

If short segments are unwanted, pruning algorithm can be employed. In my implementation, some objects seem to being thinned too aggressively, however, I haven't found the pitfalls in the pipeline for the time being.

Problem 2

Law's method

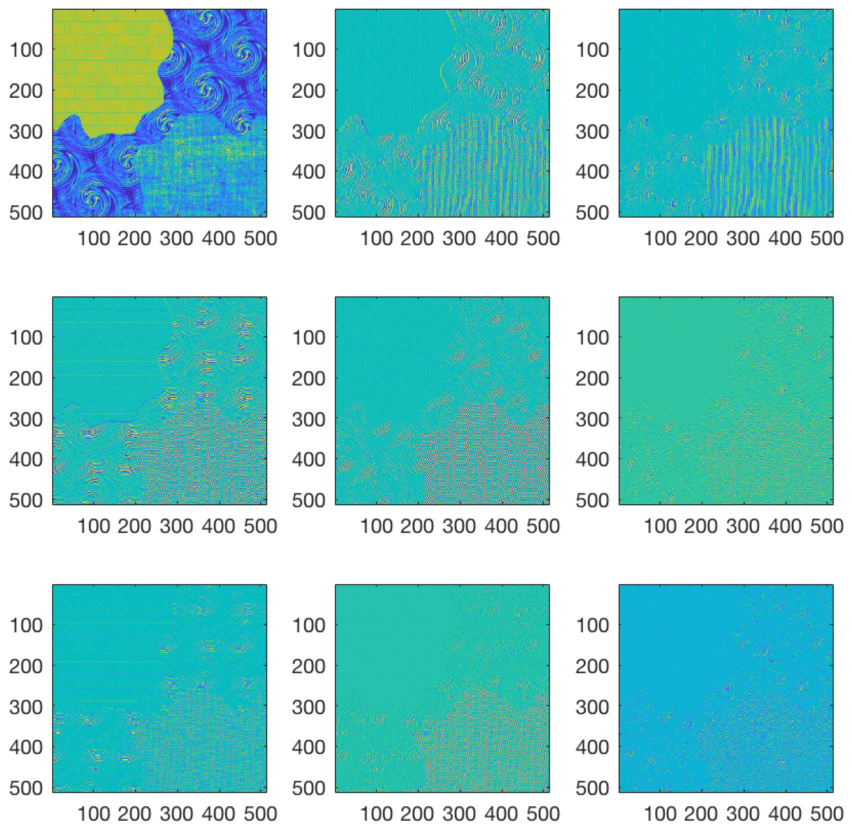


Figure 13: Filters

Figure 13 shows the results after applying 3-by-3 filters described in the slides, oriented from left-to-right, top-to-bottom are low-to-high pass.

The result is not quite clear in this case, however, the simple energy function of windows size 15,

$$T = \cap \cap_{(i,j)\in w} M(i,j)^2 \tag{13}$$

yields fabulous result.

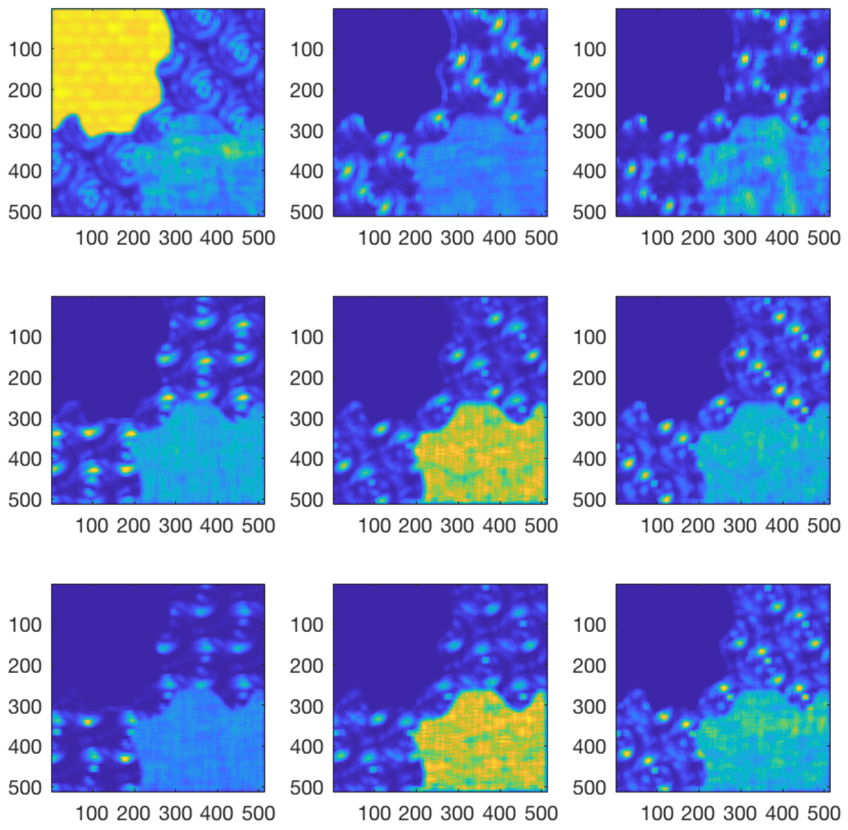


Figure 14: Energy

Figure 14 oriented the same way as Figure 13, in this case, we can easily see the whirls have their frequency components evenly distributed, while the cloth on the lower right is relatively high frequency, compares to the brick wall on the upper left side.

Using classic K-mean clustering algorithm with three classes and randomly initialized seeds, results in acceptable result.

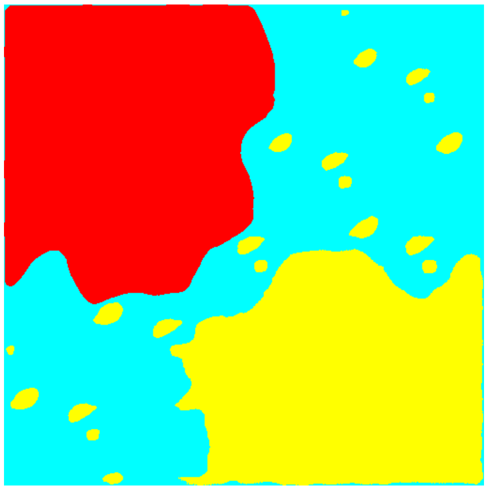


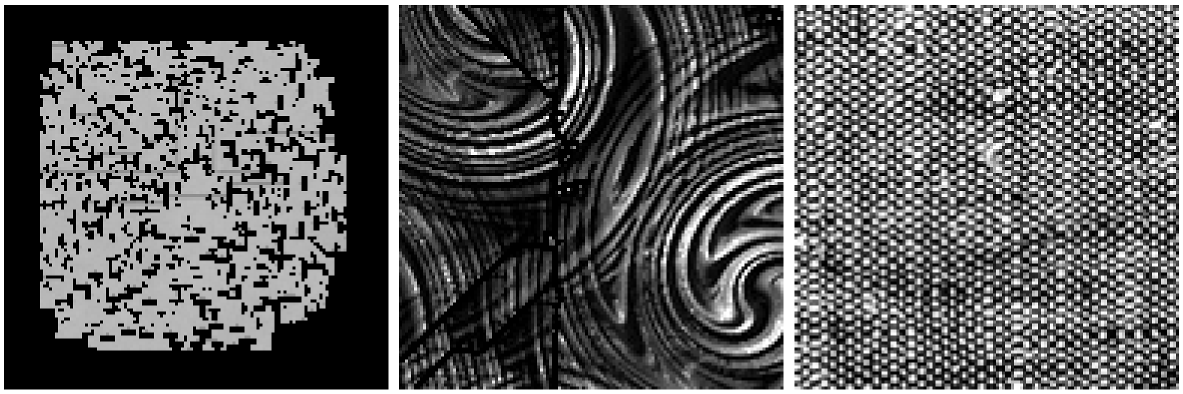
Figure 15: Segmented reuslt

However, blobs can be observed in the middle section of the whirls, one may remove them by removing objects smaller then specified size.

Texture swapping

Efos-Leung algorithm is adapted for this task, courtesy to the MATLAB implementation presented by *asteroidhouse* on GitHub, with small modification, one can adapt the algorithm for partial texture synthesis (patch the regions, relatively smaller), instead of full synthesis (generate entire image from a small patch).

The algorithm generates the texture pixel-wise, outwards from the initial seed, using pixels inside a defined windows as the context. The basic idea is modeling the patch as a generalized Markov chain, similar to the n-gram idea in natural language processing. Therefore, one build a probability for tables for each n-gram pixel combinations through repeatedly sample in the window, and synthesize new pixels by observing the most probable intensity in the sampled n-gram "corpus".



(a) Class 1 (b) Class 2 (c) Class 3

Figure 16: Synthesized

Using random patch size pick in the regions, no larger than 128-by-128, due to deadline, window size is configured to 4, and synthesis result size is restricted to 128-by-128 instead of full size 512-by-512. Figure 16 shows the synthesized result of three identified categories retrieved from previous section. Synthesis iteration is limited to 1000 due to limited time as well, instead of ensuring all the pixels are determined.

As one may expected, insufficient window size will result in information lost, causing the texture unable to be continuous, visualized as the crack in Figure 16b. The brick pattern requires much larger window to capture the grid lines, this will significantly increase the synthesis time, very limited window and limited iteration cycle not only lead to incomplete output, but also incorrect pattern in Figure 16a.

For pattern that has high frequency components, visualized in Figure 14, relatively small window and less iteration is required as hypothesized, the result in Figure 16c is pleasing as well.

Unable to generate complete synthesized swapped image on time.