Problem 1

Boundary extraction

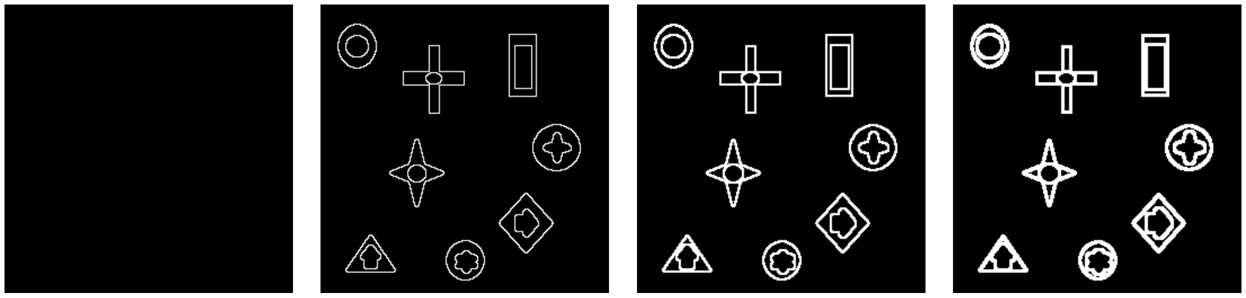

(a) 1×1 (b) 3×3 (c) 5×5 (d) 7×7

Figure 1: B

Boundary extraction is composed of erosion and its complement

$$B = I_1 \wedge \sim (I_1 \ominus SE) \tag{1}$$

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.

Counting objects

The algorithm is described below

1. Pick a non-zero pixel by scanning sequentially from the origin. Name this one pixel image as A.
2. Start from that location, dilate A using structure element SE, a 3-by-3 square in this case. Dilate until A stops expansion, in comparison to the input image I_1.
3. Set the selected pixels in A as value of N, a book-keeping variable, and increment N.
4. Remove selected pixels in A from I_1.
5. Repeat this process until I_1 is empty.

Though there is the possibility to modify size and shape of SE, since we are interesting in selecting the object, with SE being too wide will certainly cause the algorithm to miss it or accidentally select its neighbor. Therefore, 3 is kept here.

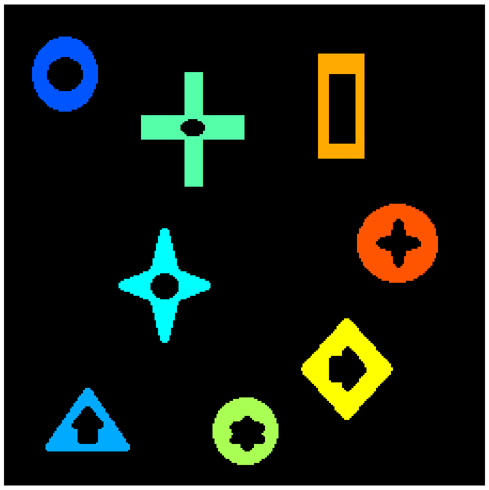


Figure 2: Labeled connected components

One may consider using running length encoding to efficiently sort the result, since RLE can better utilize the cache mechanism, but two-pass is still required similar to the naive implementation provided here.

During writing of the erosion algorithm, I also found out the slide misplaced result of Sternberg and Serra definition, despite the equations themselves are correct.

Skeletonizing

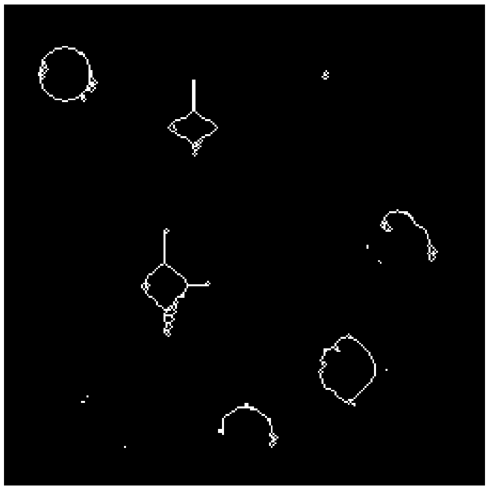


Figure 3: S, skeletonized I_1

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

$$G = X \cap [\sim M \cup P] \tag{2}$$

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

$$V = \sum_{i \in \text{neighbors}} M_i 2^i \tag{3}$$

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

$$\begin{cases} P_1 = (X \vee K_M) \wedge K_C \\ P_2 = (T \wedge \sim K_M) \wedge (K_C \wedge \sim K_M) \end{cases} \tag{4}$$

and the final result is determined by

$$P = P_1 \wedge P_2 \tag{5}$$

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,

$$A \cup B \cup C = 1 \tag{6}$$

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

Under this consensus, Equation 7 can be modified as

$$\begin{cases} P_1 = \sim ((X \vee K_M) \vee K_C) \\ P_2 = (T \wedge \sim K_M) \wedge (K_C \wedge \sim K_M) \end{cases} \tag{7}$$

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

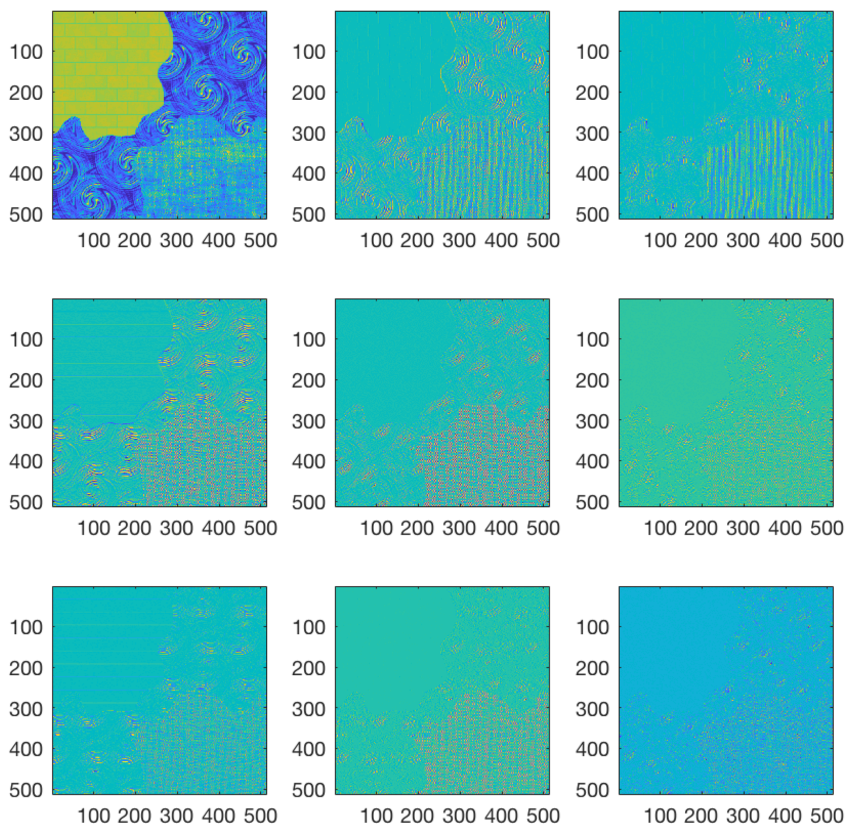


Figure 4: Filters

Figure 4 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,

$$T = \cap \cap_{(i,j) \in w} M(i,j)^2 \tag{8}$$

yields fabulous result.

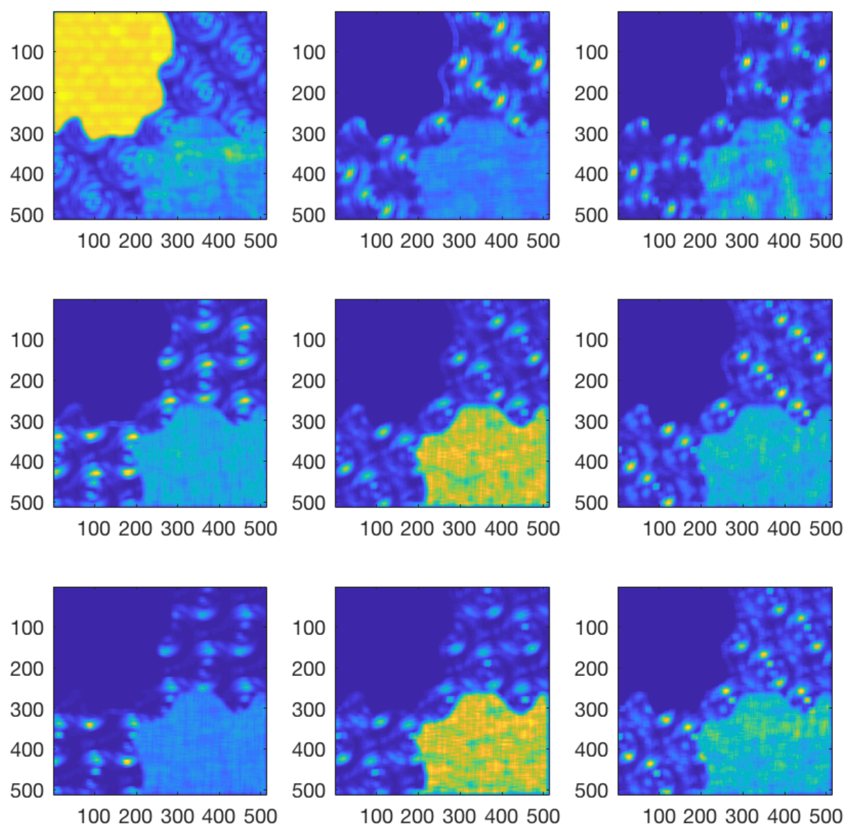


Figure 5: Energy

Figure 5 oriented the same way as Figure 4, 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.

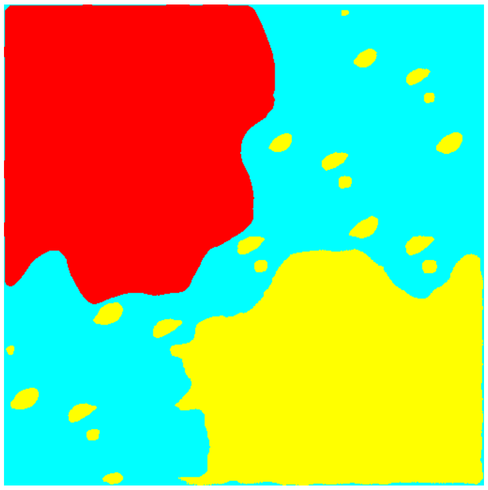


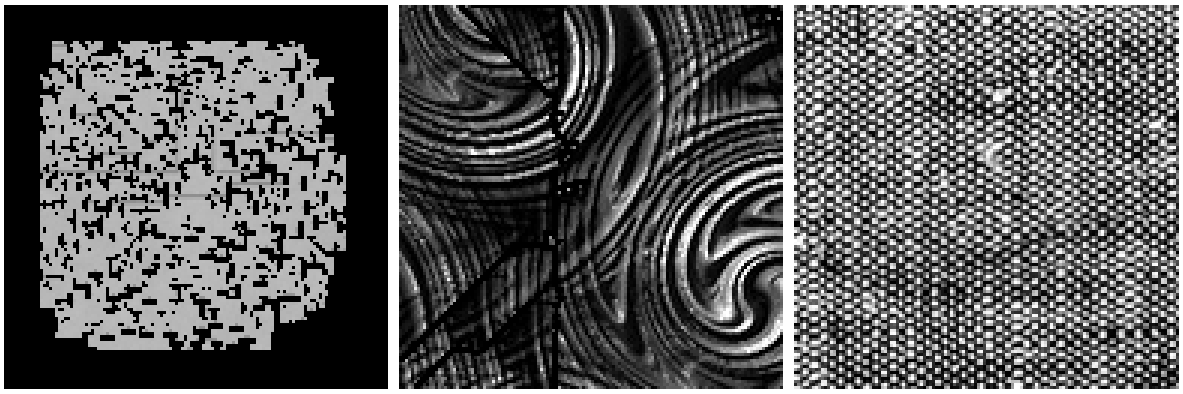
Figure 6: Segmented reuslt

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 7: 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 7 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 7b. 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 7a.

For pattern that has high frequency components, visualized in Figure 5, relatively small window and less iteration is required as hypothesized, the result in Figure 7c is pleasing as well.

Bonus