DIP Homework Assignment #2

姓名：鍾毓安

學號：B01902040

系級：資訊四

Email: [iamyuanchung@gmail.com](mailto:iamyuanchung@gmail.com)

繳交日期：2016.4.12

**Execution**

To reproduce the result, simply execute README.m under Matlab environment.

**Problem 1: Edge Detection**

Three images are given in this problem. For each given image, you are required to generate several edge maps using the following methods. [Please mark the edge points with intensity value 1 and background points with intensity value 0.]

1. Sobel edge detection
2. 2nd order edge detection
3. Canny edge detection

For each method, please apply different parameters and provide some discussions on how they would affect the resultant edge maps. From the observations of your results, list pros and cons of each method, respectively.

**Source code**

1. sobelEdgeDetection.m: Sobel edge detection is one of the most commonly used 1st order methods for detecting the edges of objects in an image. By convolving the given image with two Sobel filters and , the method generates the row and column gradients and . The resultant gradient approximations can be obtained by combining and , using:

The given threshold is used to decide how large the gradient magnitude will be taken as edge, that is, the final edge map, denoted as , is generated by:

1. laplacianOfGaussian.m: Given an image and filter size , this function performs the Laplacian of Gaussian (LoG) technique for detecting edges. Hinted by its name, LoG uses a Gaussian smoothing filter for reducing the noise in before Laplacian is applied. However, this results in two rounds of convolution, including the Gaussian smoothing process and the Laplacian operation, which causes heavy computation. Since the convolution operations are associative, one can convolve the Gaussian smoothing filter with the Laplacian filter first, and then convolve this “hybrid filter” with to achieve the required result. We will refer to this hybrid filter as LoG filter. By using the LoG filter, only one round of convolution operation with the image is needed. Argument is used to determine the size of the LoG filter. This function returns the resulting edge map.
2. cannyEdgeDetection.m: Given an image and two thresholds and , Canny edge detection includes 5 steps:
   * 1. Smoothing: we use a Gaussian smoothing filter to reduce the noise, here we use a 5 x 5 Gaussian filter and set σ = 1.4.
     2. Finding large magnitude of gradients: we use the Sobel edge detection method for finding gradients.
     3. Non-maximal suppression: after Sobel edge detection method is applied, we preserve only the local maxima of the gradient map . This is done by comparing each gradient pixel with two of its neighbor pixels, and only when is greater than both of its neighbor pixels will it be preserved. The neighbor pixels to be compared with are determined by the gradient orientation of .
     4. Double thresholding: the two thresholds and are used to discriminate the so-called "strong" and "weak" edges:

.

will then be used to generate the resultant Canny edge map in the next step.

* + 1. Edge tracking: for those gradient pixels labeled as 2, they are directly taken as edges; for those labeled as 0, they are discarded immediately; and for those labeled as 1, they will be preserved only when they are connected to at least one strong edge (by 8-connected neighbor).

More details of implementation can be found in the corresponding source files.

**Solution**

1. For each of sample1.raw, sample2.raw, and sample3.raw, we list the resultant images with different parameters applied. Figure 1, 2, and 3 are for sample1.raw, sample2.raw, and sample3.raw, respectively.

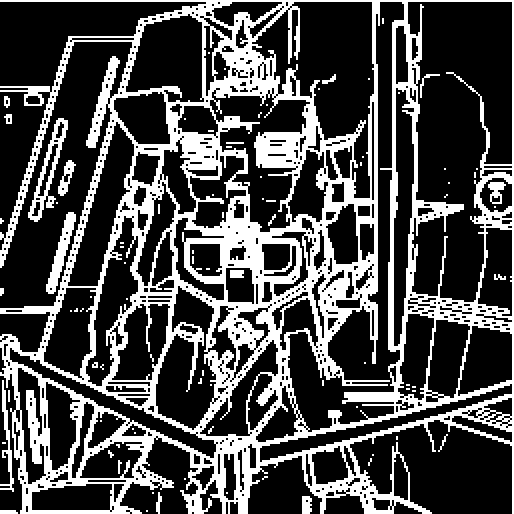
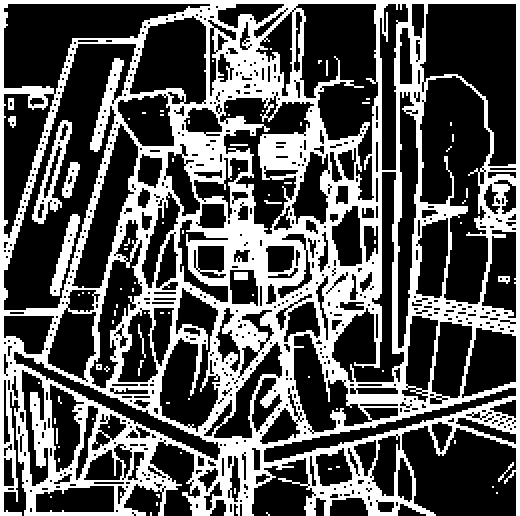
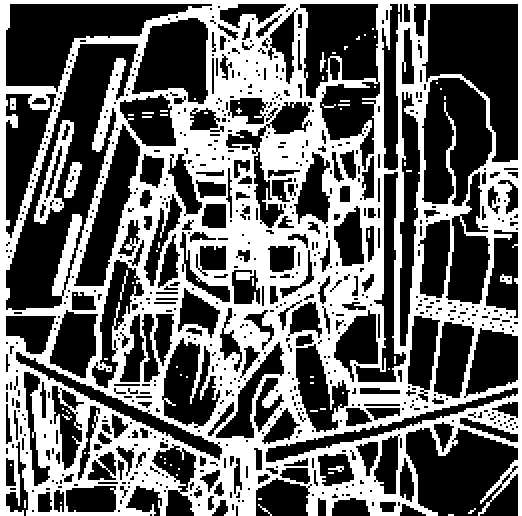


Figure 1: edge maps of sample1.raw generated by Sobel filters with threshold equal to 40, 60, and 80 (from left to right)

With increasing thresholds, less pixels will be taken as edges. From Figure 1, we can observe that when threshold equals to 40, too many details, for example, the gundam’s facial texture instead of pure edges were preserved. On the other hand, when threshold equals to 60, some edges in legs and shield were loss. Setting 60 as the threshold is perhaps the best choice based on the observation.

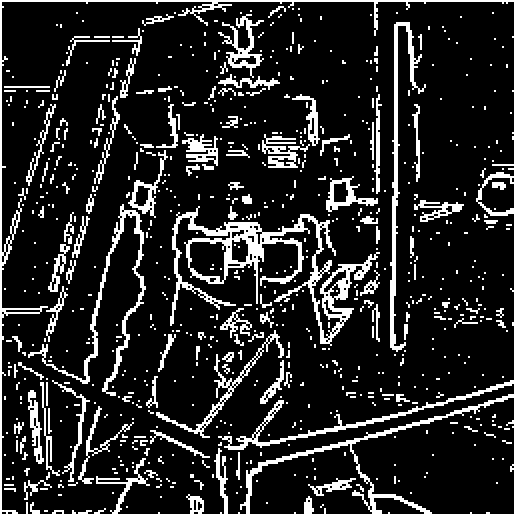
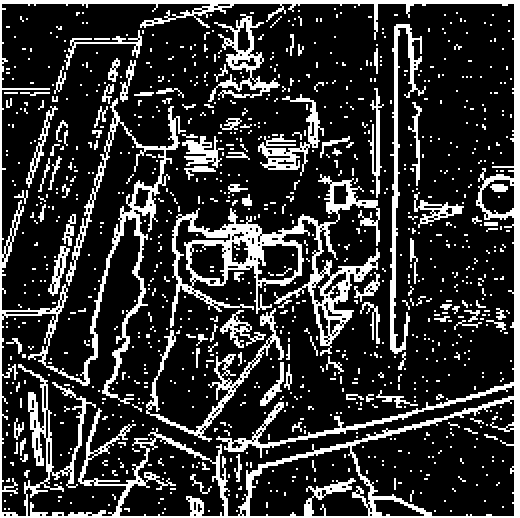
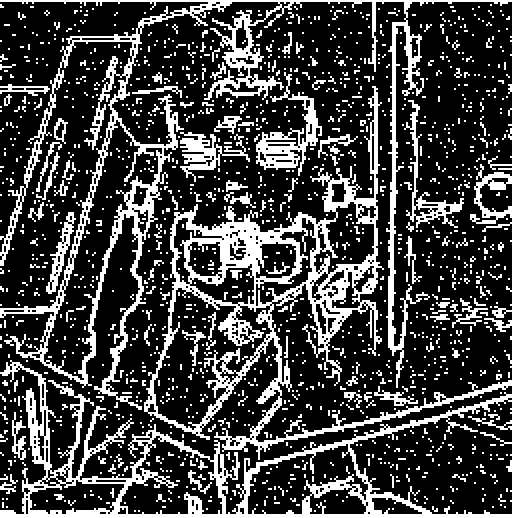


Figure 2: edge maps of sample2.raw generated by Sobel filters with threshold equal to 160, 190, and 220

From Figure 2, we can observe that the number of the noisy pixels decreased as the threshold increased, while sacrificing some important edges, for example, those edges in leg and faces of the gundam simultaneously.

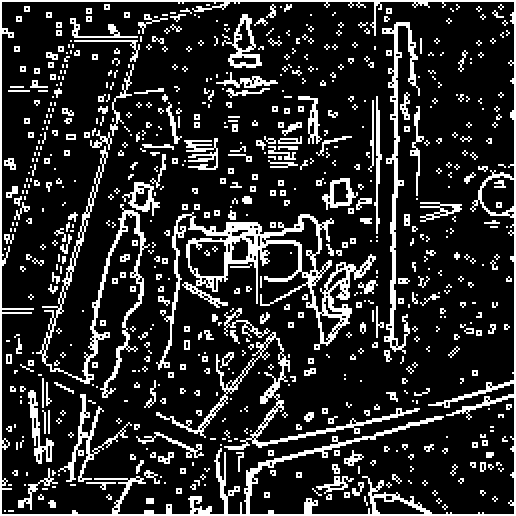
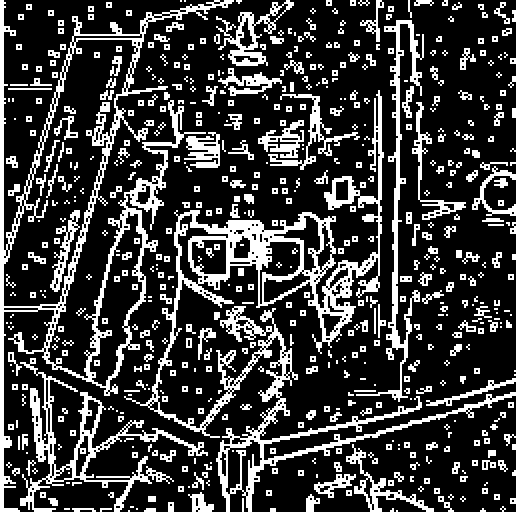
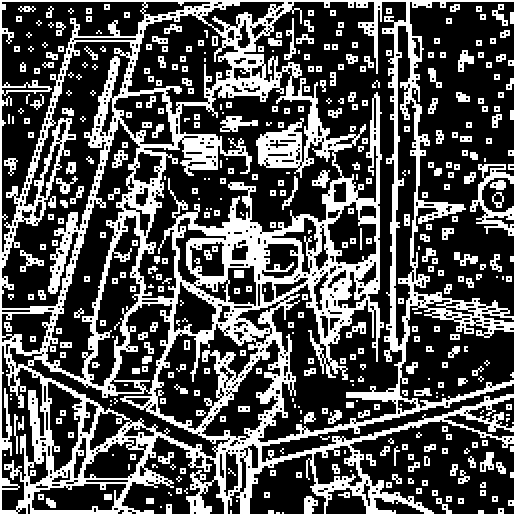


Figure 3: edge maps of sample3.raw generated by Sobel filters with threshold equal to 120, 180, and 240

As can be observed jointly from Figure 2 and 3, Sobel filters seemed to perform dreadfully on both impulse noise and Gaussian noise. More edges were sacrificed than the reduction of noisy pixels as the threshold increased.

1. Similar with (a), we provide Figure 4, 5, and 6 for sample1.raw, sample2.raw, and sample3.raw, respectively.

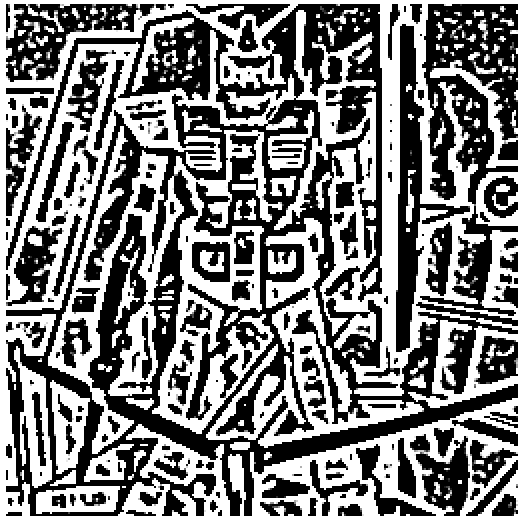
 

Figure 4: edge maps of sample1.raw generated by LoG filter with filter size = 9x9 and 11x11 (from left to right)

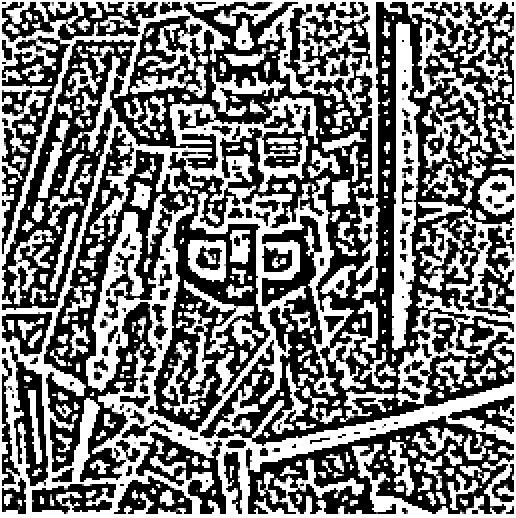
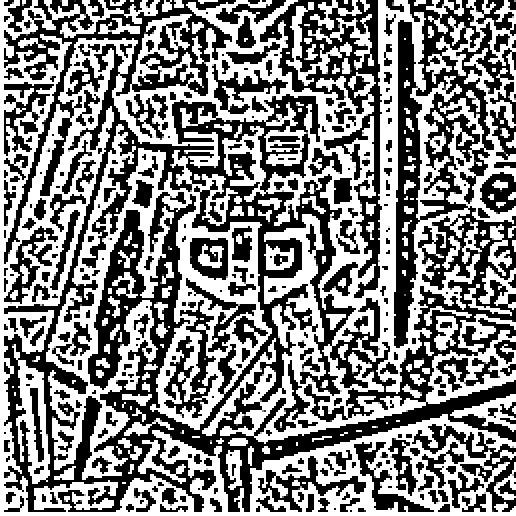
 

Figure 5: edge maps of sample2.raw generated by LoG filter with filter size = 9x9 and 11x11

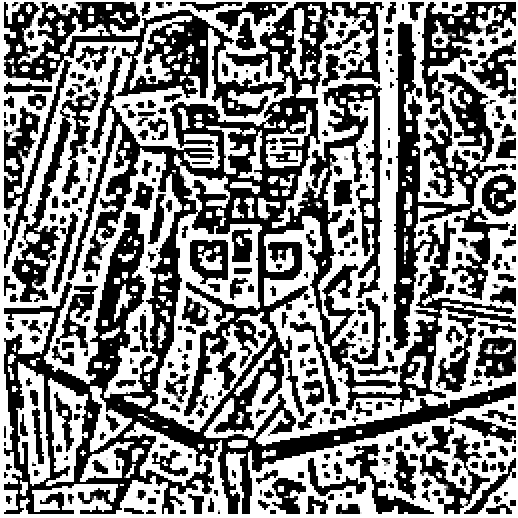
 

Figure 6: edge maps of sample3.raw generated by LoG filter with filter size = 9x9 and 11x11

1. Similar with (a) and (b), we provide Figure 7, 8, and 9 for sample1.raw, sample2.raw, and sample3.raw, respectively.

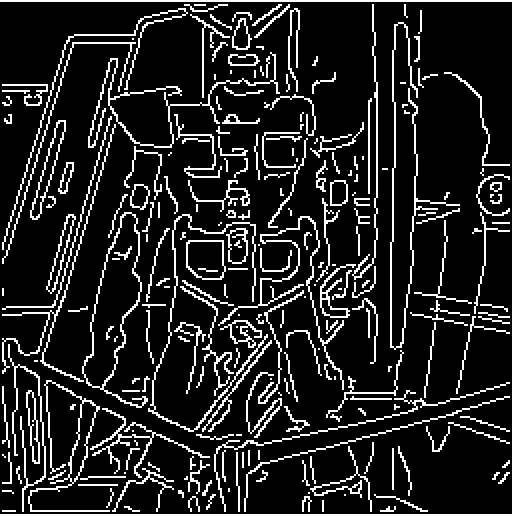
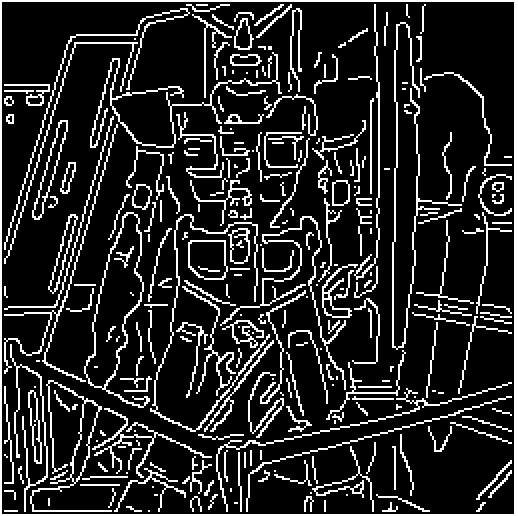
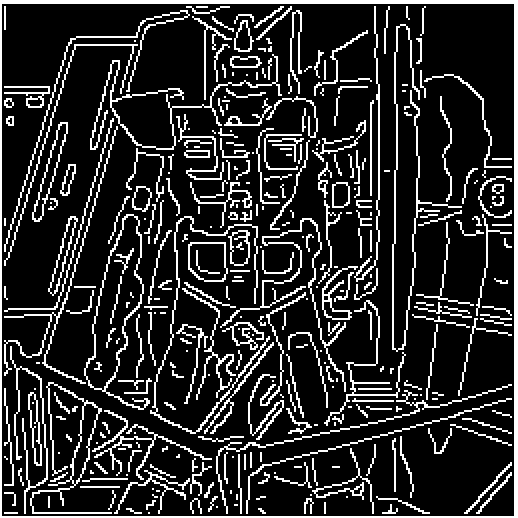


Figure 7: edge maps of sample1.raw generated by Canny with fixed and , and (from left to right)

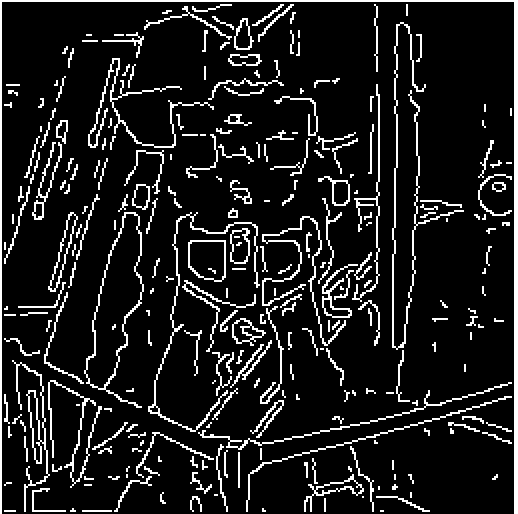
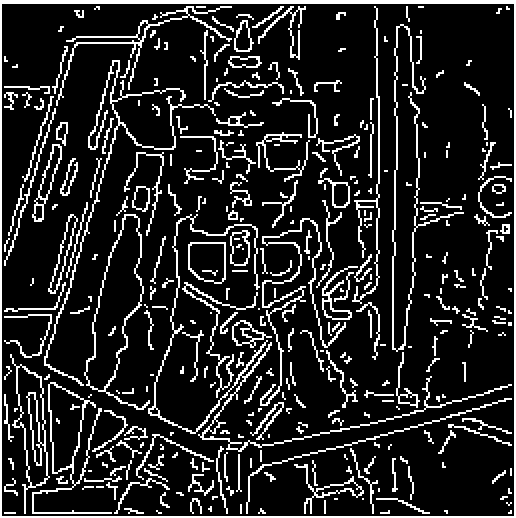
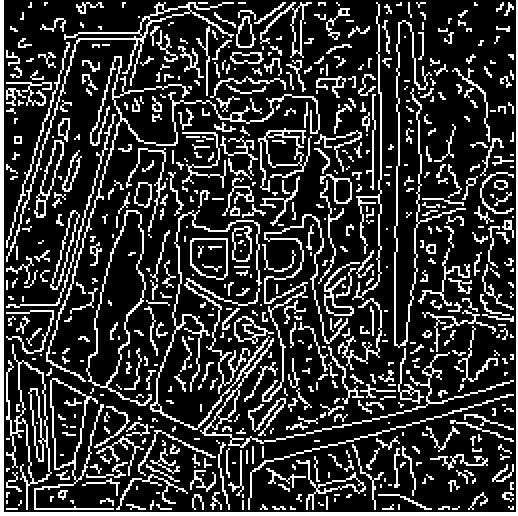


Figure 8: edge maps of sample2.raw generated by Canny with fixed and , and

From Figure 8, we can feel the powerfulness of Canny algorithm. As increased, the noisy pixels were removed efficiently, and not many edges were sacrificed. This may attribute to the first step of the Canny algorithm: the low-pass filter helped remove the Gaussian noise in sample2.raw.

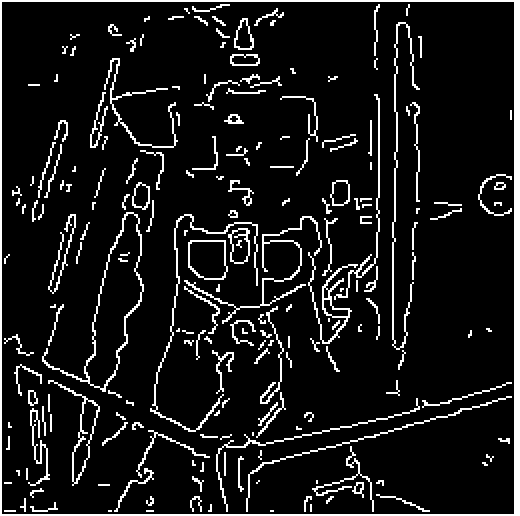
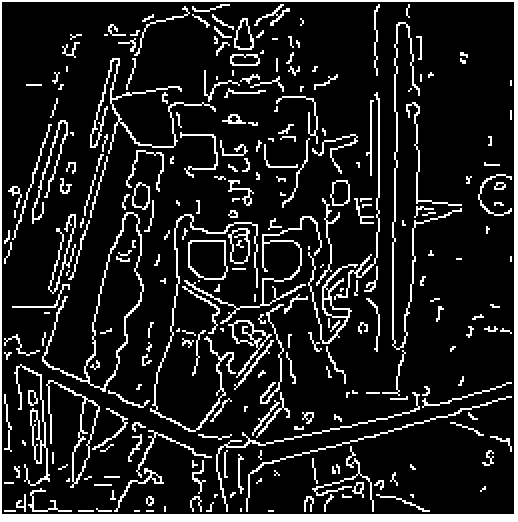
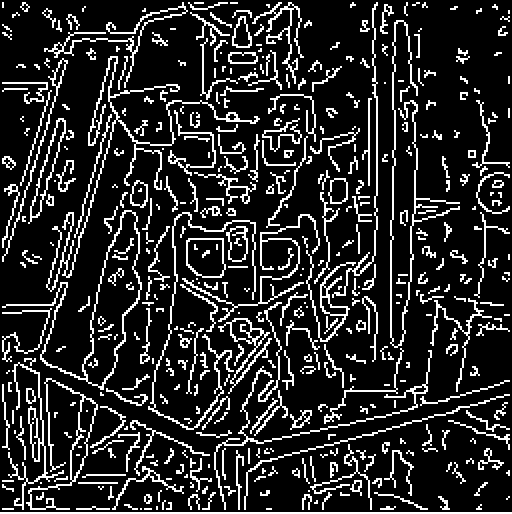


Figure 9: edge maps of sample3.raw generated by Canny with fixed and , and

The results of sample3.raw were quite surprising to me, since the Gaussian smoothing process in the first step of the Canny algorithm seemed not to affect much by the impulse noise in sample3.raw: as increased, the noisy pixels were removed efficiently, while the extent of loss of edges was also greater than that happened in Figure 8.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sobel edge detection | 2nd order edge detection (Laplacian of Gaussian) | Canny edge detection |
| Pros | * Simple to implement * Relatively fast | * After the Gaussian smoothing process, the Laplacian operation can capture the edges much better * More insensitive to noise than Sobel edge detection | * Capture the edges very efficiently despite the existence of noise (both impulse or Gaussian noise) |
| Cons | * Very sensitive to noise and fails to capture the edges when the image is noisy | * The edges captured looked unclear … * The algorithm runs very slow if the Gaussian and Laplacian operations are seperated. | * Contains many steps which cause the algorithm runs relative slower than the other two * The selection and is somewhat difficult |

**Problem 2: Geometrical Modification**

The goal of this problem is to register the given four images and perform proper geometrical modification on the overlapped square image to obtain a desired shape.

1. Please stich these four images into one complete image and paint the residual regions in black. Denote the result as R.
2. Crop the largest square image of image R and denote it as S. (Hint: the size of S is 512 x 512.)
3. Segment the image S into three parts with predefined size and design three warping functions to convert the image to a gourd-shaped image. Output the result as G.

**Source code**

1. stitchFourFigures.m
2. warpToGourdShape.m

**Solution**

1. stitchFourFigures.m takes sample4.raw ~ sample7.raw, denoted as as inputs and outputs the registered image and the cropped 512 x 512 image . Before coming up a way for stitching these four figures, we should guess what will the resultant image look like to give us a better starting point. From our guess as displayed in Figure 10, we get the information of the relative positions of : , , , and are placed clock-wisely and the relationships of overlapping are also obtained. For each pair , used two for loops that slid through to find the maximal number of pixels matching for locating the overlapping region between and . The resultant image, denoted as , is shown in Figure 11.



Figure 10: my guess of what will look like



Figure 11: the resultant image after registration

1. stichFourFigures.m also returns the cropped image , and is shown in Figure 12. Same as the hint, is a image matrix.



Figure 12: the cropped image

1. warpToGourdShape.m was implemented to perform this task. This task can be divided into three parts: warping to a small rectangle with height equals to 64 and a specified width, to a circle with radius equals to 112, and to an oval with major axis equals to 512 and minor axis equals to 224. The small rectangle was generated by simply sliding a window through and averaging the columns vectors within the window together. To warp B to a circle, the following steps were implemented (please refer to Figure 13):

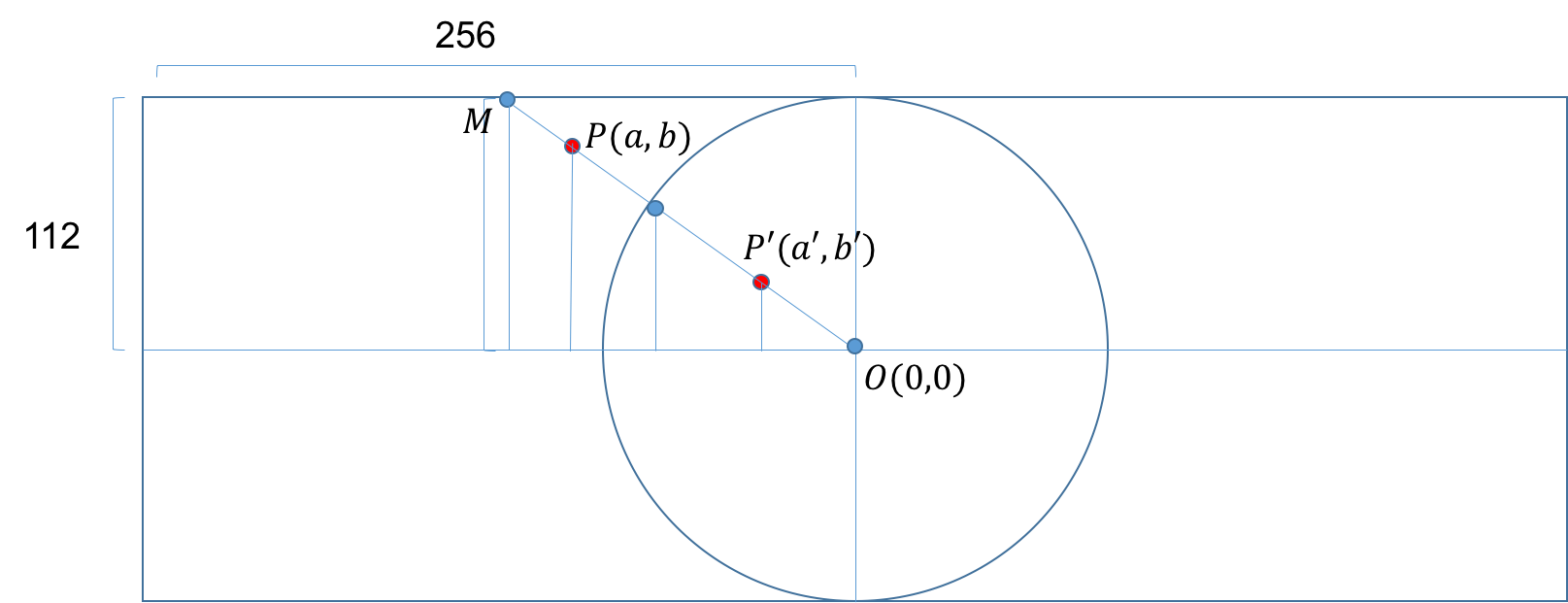


Figure 13: the design of the warping function for

* + - 1. For each pixel in , we can connect it with and form a line .
      2. Find the crossing point of line and the boundary (four sides of ).
      3. Compute the distance of and and scale to with the same proportion.

The warping function for is the same as , and the resultant gourd-shaped image is shown in Figure 14.



Figure 14: the resultant gourd-shaped image

**Problem 3: Texture Analysis**

The attached figure demonstrates a gray-level image which is compose of several animals with different texture patterns.

1. Perform Law’s method on the given image to obtain the feature vector of each pixel.
2. Use k-means to classify each pixel and label same kind of texture with same intensity. Please specify the intensity levels you adopt and output the result as L.
3. Based on image L, try to attach the correct texture to each animal as best as you can. Output the result as C.

Please provide the details of each step and discussions for each part in the report.

**Source Code**

1. lawsFeatureExtraction.m
2. classifyPixels.m
3. kMeansCluster.m
4. attachTexture.m

**Solution**

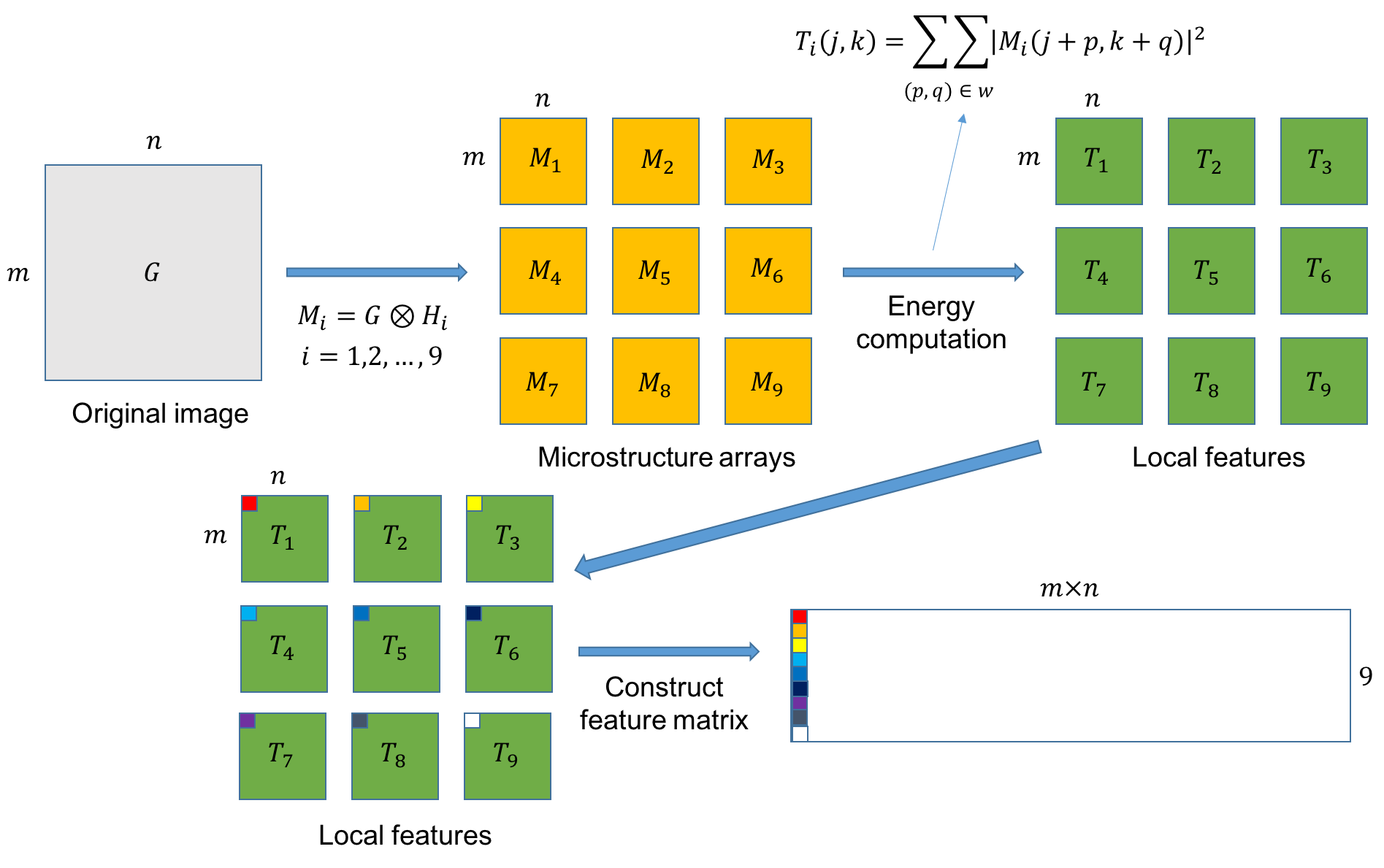
1. lawsFeatureExtraction.m implements Law’s method by taking an image and a specified window size as inputs and outputting a by feature matrix , where is the size of and each column is the feature vector of the -th pixel, indexing from top to bottom and from left to right of the original image . Basically, my implementation follows the steps stated in the course slides.

Figure 15: flow of Law’s method

Law’s method introduces 9 masks of size 3 by 3. Convolving with each of these 9 masks emphasizes the microstructures of the texture in and yields 9 microstructures arrays , where is the result of convolving with and has the the same size with . Next, Law’s method extracts the local texture features by computing the region energy of , yielding 9 feature sets , respectively, and also has the same size as . The input argument determines the window size for computing the region energy and should contains a few cycles of the repetitive texture, I chose in my implementation. The formula for computing energy is the square sum of intensity values within the window size. From the 9 feature sets , we can construct a feature matrix where each column is the feature vector of the pixel whose position is in . For each pixel whose position is in , its corresponding feature vector can be constructed by taking , , …, and and concatenating them to form a vector of length 9. Figure 15 describes the flow of Law’s method.

1. classifyPixels.m takes the feature matrix obtained from lawsFeatureExtraction.m as input, calls kMeansCluster.m to perform k-means clustering for labeling each pixel to one of the four categories of textures, including zebra, leopard, giraffe, and the background. Finally, based on the label obtained from kMeansCluster.m, classifyPixels.m generates the required image by assigning intensity value 0 to those pixels that belong to category 1, 80 to those that belong to category 2, 160 to those that belong to category 3, and 240 to those that belong to category 4. My implementation of k-means clustering algorithm is stated as follows:
   * + 1. Takes feature matrix as input where each column is the feature vector of the -th pixel.
       2. Randomly set 4 data points in as initial centers.
       3. While the assignments of all data points remain unchanged:

For each data point, compute the Euclidean distances to the 4 centers and assign the data point to the category whose center is closest to the data point.

Update the centers by averaging the data points under the same category.

* + - 1. Return a vector of length where records the category the -th pixel belongs to.

Note that due to the random initialization of centers in step 2, the labels may not always refer to as the same categories! That is, if we execute kMeansCluster.m twice, label 1 may indicate the texture of zebra for the first time, but indicate the texture of giraffe for the second time. Figure 16 displays the resultant image after classifyPixels.m. We can observe that the shapes of the leopard and the giraffe are nicely depicted, while the shape of the zebra looks just passable: the main structure of the zebra is captured but the contour is somewhat uneven and unsmooth. I think the reason is that the area of the dark gray structure of the texture within the zebra is too large, which makes it hard to distinguish it apart from the background during the feature extraction of the microstructure, causing some of those dark gray parts are mislabeled as the background.

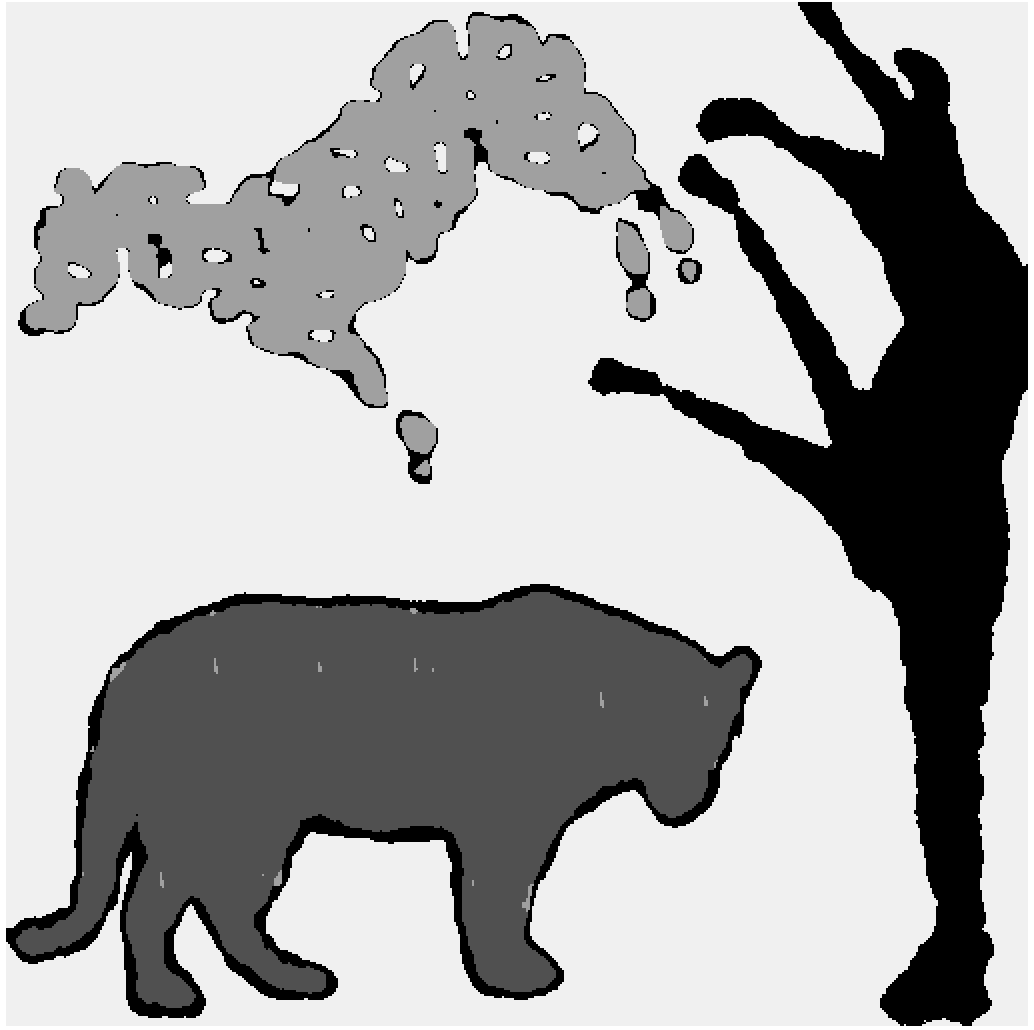


Figure 16: the resultant after classifyPixels.m

1. attachTexture.m was implemented to perform this task. In sample8.raw, the zebra has the texture of the giraffe, the leopard has the texture of the zebra, and the giraffe has the texture of the leopard. attachTexture.m first tries to grab the structure of texture from the three animals from sample8.raw and attach them to the correct animals in by repeating the corresponding structures. Figure 17 display the structures of texture grabbed from sample8.raw where the grabbed areas are bounded by squares, and Figure 18 is the resultant after attaching the correct textures to the corresponding animals. From Figure 18, we can observe that the boundary of the leopard is filled with the texture of the giraffe, such phenomenon can be explained when we look back on the boundary of the leopard in Figure 17: the leopard’s boundary was labeled as the same category as the giraffe. Fortunately, the texture on the body of the leopard and the giraffe look nice and comfortable.

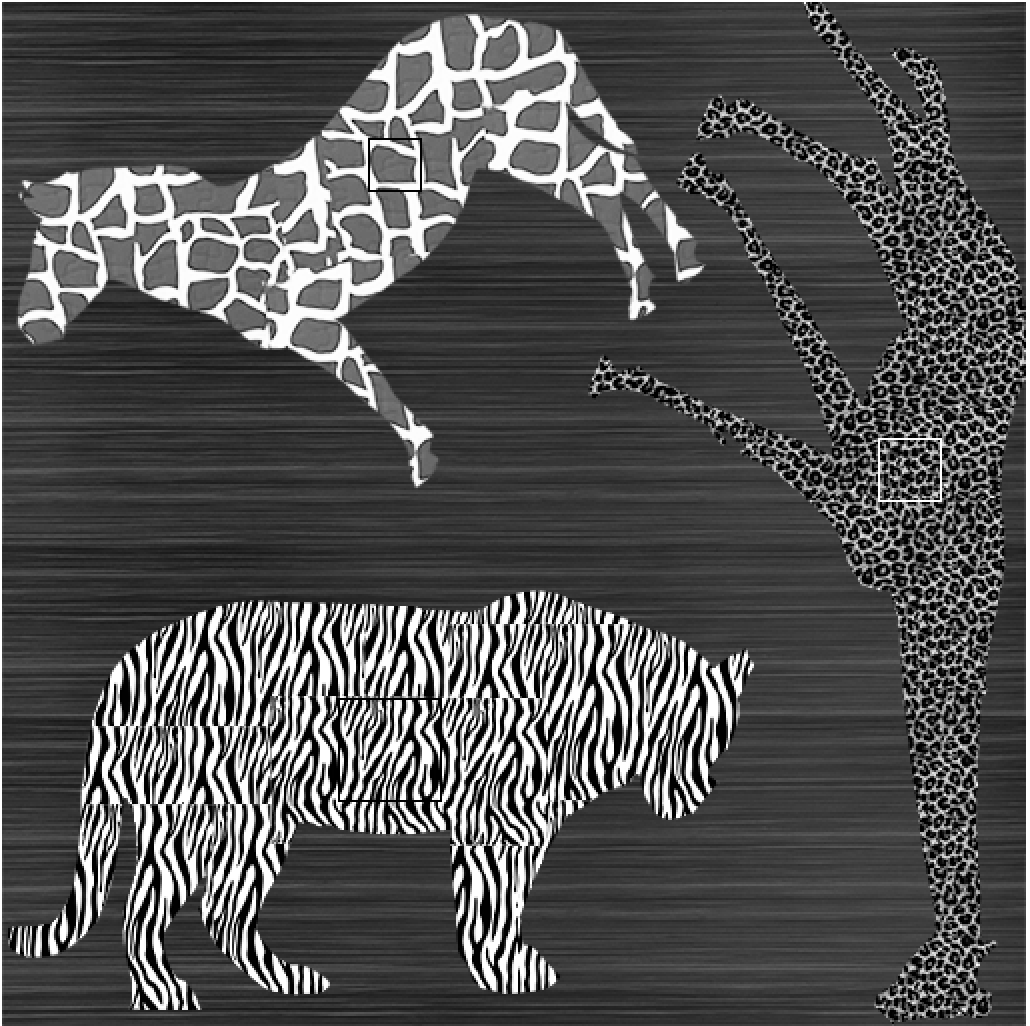


Figure 17: the structures of textures that will be repeating during attaching

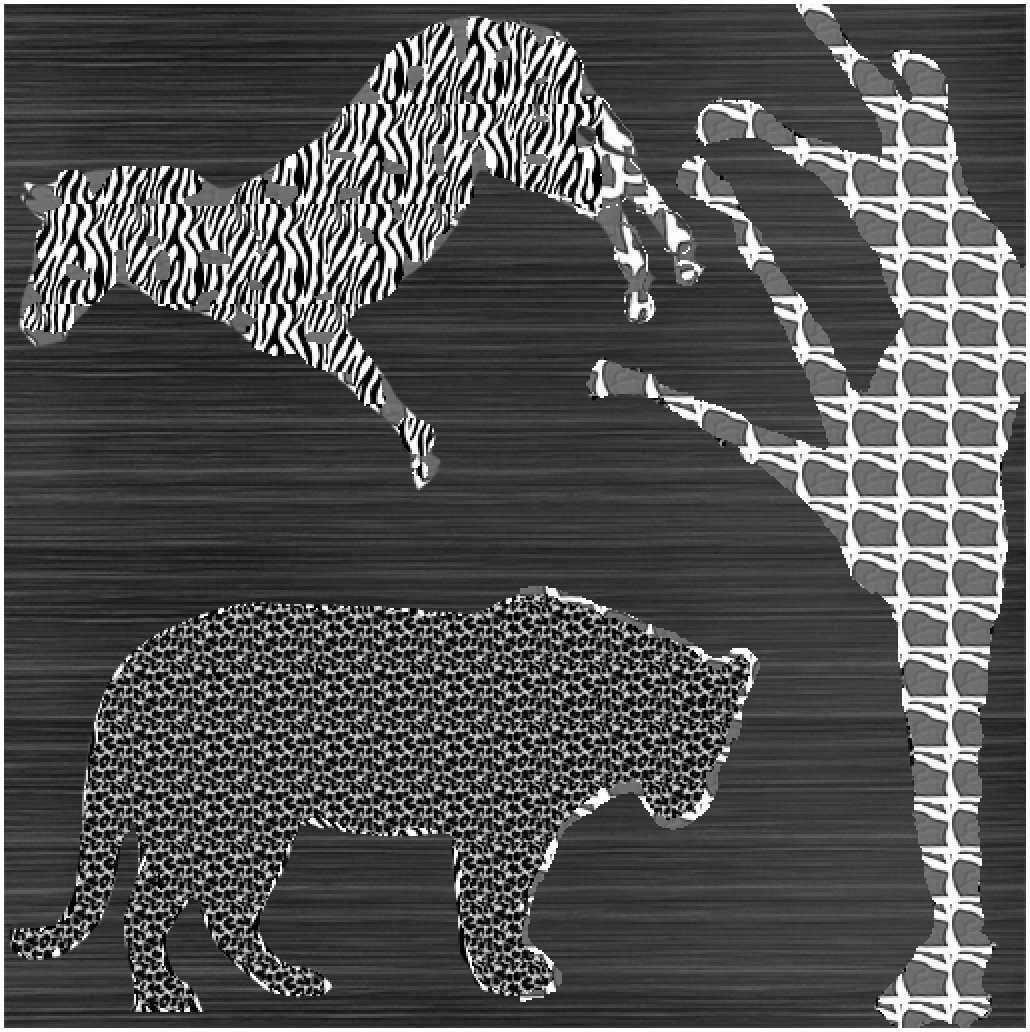


Figure 18: the resultant after attaching the correct textures to the corresponding animals