EE 569: Digital Image Processing

Homework #4

Faiyadh Shahid
Student ID: 4054-4699-70
fshahid@usc.edu

November 29, 2015

Table of Contents

1 Problem 1 **3**

1.1 Abstract and Motivation *3*
1.2 Approach and Procedure *3*
1.2.1 Segmentation of symbols 4
1.2.2 Feature Extraction from symbols 5
1.2.3 Pre-processing the test images 11
1.2.4 Building the decision tree 13
1.3 Results *14*
1.4 Discussion *21*

2 Problem 2 **24**

2.1 Abstract and Motivation *24*
2.2 Approach and Procedure *24*
2.2.1 Snake Algorithm 24
2.2.2 Level-set Algorithm 26
2.3 Experimental Results *28*
2.4 Discussion *39*

3 Problem 3 **42**

3.1 Abstract and Motivation *42*
3.2 Approach and Procedure *42*
3.2.1 SIFT Feature Extraction Algorithm 42
3.2.2 SURF Feature Extraction 43
3.2.3 Bag of Words Algorithm 44
3.3 Experimental Results *45*
3.4 Discussion *49*

References **51**

1 Problem 1

1.1 Abstract and Motivation

Optical Character Recognition (OCR) is used to convert mechanical or electronic typed images into machine-encoded text. It is one of the widely studied topics in pattern recognition, artificial intelligence and computer vision. In a very basic OCR, the system is trained with previous set of characters and is used as a reference to detect new characters encountered. The features extracted from previously trained characters are used to create a decision tree, which is used to detect the testing characters. Although such developed system has its drawbacks, this is one of the simpler ways to develop an OCR system. In this problem, we are provided with a training image consisting of 8 alphabets (S, P, E, D, L, I, M, T) and 10 numbers (0-9). The training image will be used as a reference to compare with given four images, each of which requires different pre-processing steps.

1.2 Approach and Procedure

There are various approaches that can be employed to implement an OCR system. The use of decision tree as introduced in the abstract is preferred for this exercise. The steps that are followed in implementing such OCR system can be divided into two main parts, one is the training session and the other is the testing session:

Training session:

- Convert the given training image into a binary image.
- Segment the symbols (alphabets & numbers) in the training image.
- Extract various shape-based features from the segmented images.
- Develop a decision tree using the extracted features for each segmented symbol. While developing the decision tree, it should be kept in mind that we are looking for distinct features for each symbols. For example: in case of numbers ranging from 0 to 9, 8 is the only number which can attain Euler number of -1.

Testing session:

- Pre-process the test images before segmentation. Pre-processing includes binarization as well.
- Segment the symbols in the test image as done for training images.
- Extract the features from the segmented testing images as done for training images.
- Apply the decision tree developed in the training session to detect the symbols in segmented images.
- Since the decision tree is hard coded, we can tweak the parameters manually in our main decision tree to achieve a good output.

As it is observed, there are three principle tasks we need to achieve before developing the decision tree: i) Segmentation of symbols, ii) Feature Extraction from symbols, iii) Pre-processing the test images. We discuss each of these topics separately in broad in the following sub-sections.

1.2.1 Segmentation of symbols

In this problem, we use connected component labeling for achieving automated segmentation of symbols. The segmentations we achieve are labeled with distinct tags. The portions of the image with unique tags can be separated by writing a simple crop function, which are used for feature extraction later. A brief description of both tasks (connected component labeling & cropping) is provided below.

Connected Component Labeling Connected component labeling is used to uniquely label subset of connected components. Connected component labeling should not be confusion with image segmentation. Image segmentation provides us the result of different segment regardless of pixel values whereas connected component labeling is done on a binary image to find unique segments or partitions.

The algorithm used in this problem is inspired from a faster scanning algorithm developed by Lifeng *et al.* [1]. The implementation of the algorithm follows two-pass method:

First pass:

1) We raster-scan through each pixel of the image.
2) If the current pixel is not 0 i.e. 1 in our case
   i) We get the neighboring pixels of the current pixel.
   ii) If there are no neighboring pixels, the pixel is uniquely labeled.
   iii) Otherwise, we find neighboring pixels with the smallest label and denote the current pixel by that label.
   iv) We also store the equivalence among neighboring pixels.

Second pass:

1) We raster-scan through each pixel of the image again.
2) If the current pixel is not 0
   i) We re-label the pixel with the lowest equivalent label.
   ii) Otherwise, the label stays on the current pixel.

An example is provided from the trained image in **Figure 1.2.1**. Different colors are used to represent different segments. If we count the color labels, it is found to be eighteen (18) in number which is the number of symbols provided for the problem.

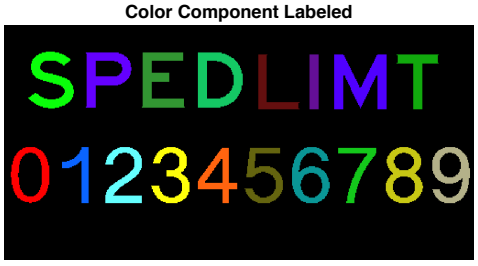


Figure 1.2.1: Color labelling of segmented symbols using connected component labelling

Cropping the labeled portions As observed in Figure 1.2.1 each symbol is segmented with a unique tag after using connected component labeling. However, we need to separate individual symbol so that we can use them for feature analysis. A quite straightforward algorithm of cropping can be used to separate the segments. Since they are labeled uniquely, we can write a function which finds the limit of boundaries in which the tag exists and thus cropping of the image is achieved. A pseudo code of the cropping function is as follows:

```
function crop(Labeled_image)
{
row = size(Labeled_image,1); col= size(Labeled_image,2);

rightBoundary = zeros(1, max(Labeled_image));
leftBoundary = col * ones (1, max(Labeled_image));
topBoundary = row * ones (1, max(Labeled_image));
bottomBoundary = zeros(1, max(Labeled_image));

for i = 1: max(Labeled_image)
        for r =1: row
                for c=1: col
                 if Labeled_image(r,c) == i
                        rightBoundary(1, i) = max (rightBoundary(1, i),c);
                        leftBoundary(1, i) = min(leftBoundary(1, i),c);
                        topBoundary(1, i) = min(topBoundary(1, i) ,r);
                        bottomBoundary(1, i) = max(bottomBoundary(1, i) ,r);
                end
                end
        end
crop{i}=Labeled_image(topBoundary(1,i):bottomBoundary(1,i),leftBoundary(1,i):rightBoundary(1,i) );
end
}
```

Here, *rightBoundary* and *leftBoundary* are considered as the rightmost and leftmost portions of the segmented images along the columns. Accordingly, *topBoundary* and *bottomBoundary* are considered as the topmost and bottommost portions of the segmented image along the rows. This is a quite straightforward implementation of cropped sections from the labeled images.

1.2.2 Feature Extraction from symbols

After we have implemented our automated segmentation of symbols, we can find various feature extraction parameters for each symbol. In this problem, we use features that are mainly based on shapes of the symbols. They are: i) Euler-number (4-connectivity & 8- connectivity), ii) Area, iii)

Perimeter, iv) Circularity, v) Aspect Ratio, vi) Various Spatial moments, vii) Hu's Invariant Moments, viii) Vertical and Horizontal Cuts. The procedure of calculating them are described below.

Using bit quads (Euler-Number, Area, Perimeter, Circularity)

Before we move onto calculating the feature parameters, we need to make ourselves familiar with the concept of bit quads. There are five types of quads that can be observed in a binary image. These are:

$$Q_0 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$Q_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$Q_2 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

$$Q_3 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$Q_4 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$Q_D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

We raster-scan to count the number of bit-quads of above pattern achieved in the entire image. The given bit quad number calculation can easily be done by adding the number of ones in a 2x2 window except for Q_2 and Q_D, where we need to be careful. Since both of the quad type adds to 2, we place a condition in search of the above pattern quads as a special case. The total number of quads that can be found in the entire image is equal to (row-1) * (col-1). For an image of size 61x41, the number of bit quads obtained in the image will be 60x40 = 2400. It is to be noted that the image should be binarized before counting the number of bit quads in the image. Using the above calculation of number of bit quads, we can formulate various shape-based features in a cropped image. Some of these will be found in the formulae of various feature parameters.

Euler-Number Euler Number is generally defined as difference between number of connected objects and number of holes. However, the number of connected objects and number of holes can not be calculated separately using local neighborhood calculation. We can easily calculate the Euler number for 4-connectivity and 8-connectivity using the bit quads as follows:

$$\text{Euler number (4-connectivity)} = \frac{1}{4}[n\{Q_1\} - n\{Q_3\} + 2n\{Q_D\}]$$

$$\text{Euler number (8-connectivity)} = \frac{1}{4}[n\{Q_1\} - n\{Q_3\} - 2n\{Q_D\}]$$

Perimeter The perimeter of each object is defined to be the number of pixel sides traversed around the boundary of the object starting at an arbitrary initial boundary pixel and returning to the initial pixel. The calculation of perimeter is only useful for binary images. An approximated calculation of perimeter can be done using the following formula as:

$$\text{Perimeter} = \boldsymbol{n\{Q_1\} + n\{Q_2\} + n\{Q_3\} + 2n\{Q_D\}}$$

Area The area of each object is the number of pixels in the object for which it's values is one (1). We can find the enclosed area of the object by calculating the total number of pixels for which its value is 1 within an outer perimeter boundary. An approximated calculation of area can be done using the following formula:

$$\text{Area} = \boldsymbol{\frac{1}{4}[n\{Q_1\} + 2n\{Q_2\} + 3n\{Q_3\} + 4n\{Q_4\} + 2n\{Q_D\}]}$$

Circularity With the establishment of area and perimeter, we can use various shape descriptors to specify an object's feature characteristics. One of the most commonly used feature is circularity. This attribute is also known as thinness ratio. There are various ways proposed to calculate circularity of an object. The most common one is:

$$\text{Circularity} = \frac{\boldsymbol{4\pi * (Area)}}{\boldsymbol{(Perimeter)^2}}$$

It is to be mentioned that the circularity of a circle-shaped object is unity whereas oblong-shaped objects comprise of a circularity less than 1.

Aspect Ratio The most commonly used feature of an image or a geometrical object is its aspect ratio. Although it is quite straightforward to formulate aspect ratio, the property of aspect ratio helps us in distinguishing longer and broader characters. The formula of aspect ratio is:

$$\text{Aspect Ratio} = \frac{\boldsymbol{Width(\#\ of\ columns)}}{\boldsymbol{Height\ (\#\ of\ rows)}}$$

Spatial Moments We can use moments for shape analysis in an object or image by applying the classical relationships of probability theory for central moment. Such calculation of moments can be extended for discrete image by forming spatial summations over the discrete image function.

Given the image coordinate begins at the lower left corner, we can define scaled coordinates as follows:

$$x_k = k - \frac{1}{2}$$

$$y_j = J + \frac{1}{2} - j$$

where x_k and y_j are scaled coordinates along the rows and columns respectively; k & j are the indices for rows and columns respectively; and (K, J) is the size of image. From the above scaled coordinates, we can generalize discrete spatial moment as:

$$M(m,n) = \frac{1}{J^n K^m} \sum_{j=1}^{J} \sum_{k=1}^{K} (x_k)^m (y_j)^n F(j,k)$$

Here, $F(j,k)$ is the image function or matrix. From the generalization, three importantly derived moments were used for the problem:

$$M(0,0) = \sum_{j=1}^{J} \sum_{k=1}^{K} F(j,k)$$

$$M(1,0) = \frac{1}{K} \sum_{j=1}^{J} \sum_{k=1}^{K} (x_k)^1 F(j,k)$$

$$M(0,1) = \frac{1}{J} \sum_{j=1}^{J} \sum_{k=1}^{K} (y_j)^1 F(j,k)$$

Here, $M(0,0), M(1,0), M(0,1)$ are called zero-order, first-order row and first-order column moments respectively. We can also calculate the image centroid from the afore-mentioned order moments as follows:

$$x_{kc} = \frac{M(1,0)}{M(0,0)}$$

$$y_{jc} = \frac{M(0,1)}{M(0,0)}$$

where x_{kc} and y_{jc} are x-centroid and y-centroids. The centroid, also known as center of gravity, is the balance point of the image function such that the mass of the image to the left and right of x_{kc} and above and below of y_{jc} is equal. In this problem, the x- and y-centroids were mainly used.

Hu's Invariant Moments Hu proposed a normalization of the unscaled spatial central moments, which can be explained by the relation as follows:

$$V(m,n) = \frac{U_U(m,n)}{[M(0,0)]^\alpha}$$

where

$$U_U(m,n) = \sum_{j=1}^{J} \sum_{k=1}^{K} (x_k - x_{kc})^m (y_j - y_{jc})^n F(j,k) \ [Unscaled\ Spatial\ Central\ Moment]$$

$$\alpha = \frac{m+n}{2} + 1; \quad for\ m+n = 2, 3, \ldots$$

Hu used these normalized central moments to develop a set of seven compound spatial moments that are invariant to translation, rotation and scale change in the continuous image domain. They are defined as follows [2]:

$$h_1 = V(2,0) + V(0,2)$$

$$h_2 = [V(2,0) - V(0,2)]^2 + 4[V(1,1)]^2$$

$$h_3 = [V(3,0) - 3V(1,2)]^2 + [V(0,3) - 3V(2,1)]^2$$

$$h_4 = [V(3,0) + V(1,2)]^2 + [V(0,3) - V(2,1)]^2$$

$$h_5 = [V(3,0) - 3V(1,2)][V(3,0) + V(1,2)][[V(3,0) + V(1,2)]^2 - 3[V(0,3) + V(2,1)]^2]$$
$$+ [3V(2,1) - V(0,3)][V(0,3) + V(2,1)][3[V(3,0) + V(1,2)]^2$$
$$-[V(0,3) + V(2,1)]^2]$$

$$h_6 = [V(2,0) - V(0,2)][[V(3,0) + V(1,2)]^2 - [V(0,3) + V(2,1)]^2]$$
$$+ 4V(1,1)[V(3,0) + V(1,2)][V(0,3) + V(2,1)]$$

$$h_7 = [3V(2,1) - V(0,3)][V(3,0) + V(1,2)][[V(3,0) + V(1,2)]^2 - 3[V(0,3) + V(2,1)]^2]$$
$$+ [3V(1,2) - V(3,0)][V(0,3) + V(2,1)][3[V(3,0) + V(1,2)]^2$$
$$- [V(0,3) + V(2,1)]^2]$$

In this problem, the sum of all the moments were calculated as one of the prominent feature parameters as follows:

$$h_{sum} = \sum_{i=1}^{7} h_i$$

Horizontal and Vertical Cuts A unique property called horizontal and vertical cuts in the middle of an object was used to calculate the number of 1s that are existent after thinning the object. In a step-wise procedure, we first thin the image. Then, we count the number of 1s available in the middle of the image vertically and horizontally. The number of 1s obtained in the horizontal direction are called

horizontal cuts and the numbers in the vertical direction are called **vertical cuts.** The addition of both the cuts provide us with total number of cuts obtained in the middle of the image. An example is provided below with **'S'**:

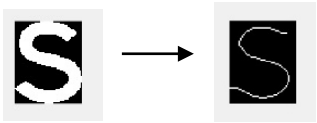


Figure 1.2.2: Effect of thinning on S

The effect of thinning can be seen on the thick bordered **S** in **Figure 1.2.2.** We can now easily calculate the number of 1's that will be obtained in the middle of the image vertically and horizontally. The number of cuts are 3 and 1, which adds up to total number of cuts as 4. An advantage of finding cuts in this manner provides a unique way of distinguishing among symbols either in terms of horizontality, verticality and sum of both. It will be observed that this property was utilized many times while creating the decision tree.

The procedure of implementing thinning can be divided into two steps. Let us suppose that F(u, v) is our given image, M(u, v) is our secondary image implemented from the first step, and G(u, v) is our final morphological image implemented from the final step. The top view of the algorithm is shown in **Figure 1.2.3:**

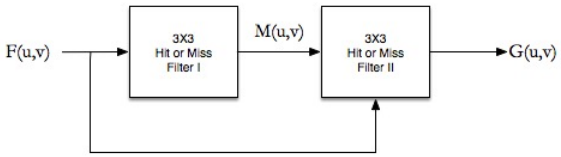


Figure 1.2.3: Top view of the algorithm

Step-1: 3 X 3 Hit-or-Miss Filter I in **Figure 1.2.4**

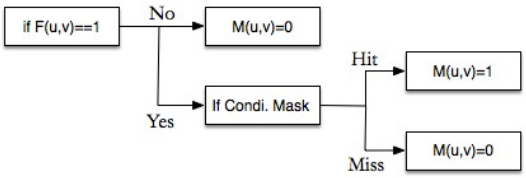


Figure 1.2.4: Step-1 implemented using Hit-or-Miss Filter I

Step-2: 3 X 3 Hit-or-Miss Filter II in **Figure 1.2.5**

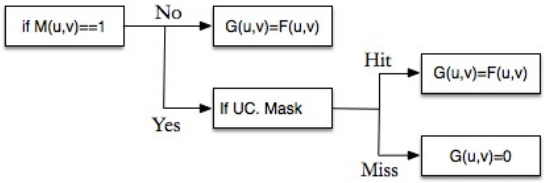


Figure 1.2.5: Step-2 implemented using Hit-or-Miss Filter II

In the first step, we compare the central pixel of our original image F(u, v) with the conditional mask provided in pattern table in a 3x3 window. In the second step, we compare the central pixel of our mask image M(u, v) with unconditional mask to obtain our final G(u,v). The masks are provided in the pattern table and for thinning we use the mask that are allocated with T. For the step of creating pattern tables, all the masks were hard coded to make sure that any mask is not missed while comparing. In case of D= 0 or 1(don't cares), A or B or C =1, all the combinations were considered to create the masks. For example: if a mask contained 3 D's, the pattern was created for all combinations of D, which is 8 (= 2^3) in number.

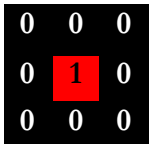
These were the features implemented for the OCR program. The features that were mostly used in the decision tree include Euler Number, Aspect Ratio, Horizontal and Vertical Cuts, Circularity and x- and y- centroids.

1.2.3 Pre-processing the test images

Pre-processing the test images is an important step since the visual appearance and content are different from the trained cases. In order to bring them closer to the training cases, we do pre-processing on each image. In our case, different methods of pre-processing were followed for different images. Different methods that were used for pre-processing are briefly described below.

Binarization with thresholding Binarizing the image with content of either 0 or 1 was used to pre-process the images. Before applying the binarization, we convert any color image to a grayscale image. Binarization is done with a simple thresholding routine where values below a certain value are set to 0 or above are set to 1. The thresholding value is an important factor in determining the image binarization, which assists in getting feature parameters for symbols closer to ideal cases.

Dilation In some cases, our test images required dilation since the characters provided were not completely filled on the edges or were affected as a result of binarization. As a measure of compensation, dilation was applied to fill the edges of the image. The idea behind dilation is that it causes objects to grow in size. The amount by which dilation occur depends on the choice of the structuring element. A usual choice of structuring element **B** is taken as follows:



From the structuring element, dilation is defined as [3]:

$$A \oplus B = \{z | (Z = a + b, a \in A, b \in B\}$$

Hence, dilation makes an object larger by adding pixels with values of '1' around its edges whereas erosion makes it smaller by removing the pixels from its edges. In terms of programming, it is quite

straightforward to create a function for erosion and dilation. We pass a window of size 3x3 and while passing it through the elements in the original image, we replace the center pixel by the maximum of the values in the window in case of dilation.

Histogram Equalization For some images with poor lighting and lesser contrast, histogram equalization was applied to get an image till the content becomes visible. For equalization, both contrast stretching based on transfer-function and histogram specification were tried.

The steps followed to implement histogram equalization based on transfer function are:

- We find the histogram for each channel separately
- We create a cumulative histogram, which basically adds up the total number of pixels in full range of intensities
- We create a normalized Cumulative Distribution Function (CDF) such that the values range from 0 to 1
- We create a temporary output which multiplies each values of CDF with the max intensity (in this case 255)
- Finally, we map the values of input channel using the temporary output in a final equalized matrix (channel). The process is repeated for all the channels and the final equalized output is thus obtained. The transfer function in this case is:

$$T(x) = \frac{255}{N} \sum_{i=0}^{x} H(i)$$

Here, *x* represents the gray level value, *H(i)* is the histogram value for grayscale value of *i* and N is the total number of pixels in the image.

Another method used to equalize the histogram was based on cumulative probability. The steps followed to implement the equalization are as follows:

- We find the histogram, cumulative histogram as done for method one
- We find the total number of pixels each gray value should consist of i.e. N = Total number of pixels / 256
- We create an array with 256 bins such that the maximum number of pixels that can be hold in each bin is equal to N
- We find the bin where each pixel should fall into from the cumulative histogram. In case the bin is full, we move to the next greater value
- Finally, we place all the pixels in its destined bins. Hence, we should get a histogram which has equal number of pixels for all gray values and the cumulative function should look like a ramp function.

In this problem, it was observed that histogram equalization based on transfer function provided better results compared to the other histogram equalization.

Hole-filling and Boundary smoothing filter Since thinning morphological operation is used for determining the number of cuts in a segmented image, it was necessary to pre-process the segmented image with a hole-filling and boundary smoothing filter for both training and testing images. A hole is defined as a background region (0) surrounded by a connected border of foreground pixel values (1). While performing thinning and skeletonizing, we need to perform the pre-processing step of filling the hole. Hence, we need to implement a hole-filling filter. The main purpose of hole-filling filter is to fill all the holes in the binary image. The basic formula used to implement a hole-filling filter is:

$$X_k = (X_{k-1} \oplus B) \cap A^c$$

Here, we are basically performing a dilation operation with the previous pixel values and then taking the common element between dilated operation and the complement of ***A*** whose elements are 8-connected boundaries with each boundary enclosing a hole [4]. In terms of we need to look for the patterns of such intersection and apply the above formula to obtain our desired result. In this case, **B** is considered as follows:

0	1	0
1	1	1
0	1	0

It is a common mistake to think of hole-filling filter equivalent of dilation. However, it is not a true statement. In hole-filling, we first apply the dilation and then take the intersection of dilated image with complemented A which limits the filling operation inside the region of interest. It is often known as conditional dilation [3].

Boundary smoothing filter is applied on the hole-filled image since it leaves extra pixels because of its dilation operation. Hence, we need to smoothen the boundary so that we do not loose the geometric structure of the object. As we are applying a smoothing filter, it is enough to use an averaging filter, so called Gaussian averaging filter. It should be mentioned that once we apply averaging filter, we have to convert our image back to binary image through thresholding.

1.2.4 Building the decision tree

We build the decision tree based on the feature parameters obtained in the training session and use it to compare with the parameters in the test case to detect symbols contained in it. Although decision tree can be built arbitrarily from the feature parameters, it is always better to make use of intuition while building the tree. For example: from our given list of symbols, the symbol with the least aspect ratio is expected to be **I** since the ratio of weight compared to height is really small. **'I'** is also expected to have a circularity of infinity since the perimeter of the symbol is zero (0) because there are pixels around which boundaries can be drawn. Hence, usage of intuition along with some hard tweaking based on testing can lead us to better performance. The decision tree developed for this problem is based on common intuition of feature parameters. However, there are cases where the conditions have to be tweaked manually in order to reach better performance.

1.3 Results

The feature parameters obtained for different symbols in the training case are shown in **Table 1.3.1.**

Table 1.3.1 Feature Extracted from Training.raw

Symbols	Area	Perimeter	Euler Number(4-connectivity	Euler Number(8-connectivity	Circularity	Aspect_Ratio	Hu's Moments Sum	X_Centroid	Y_Centroid	Vertical Cuts	Horizontal Cuts	Total Cuts
0	1039.5	309	0	0	0.13680986	0.68852459	0.497220082	0.496272812	0.49079066	2	2	4
1	624.5	110	1	1	0.64857012	0.475409836	0.673075724	0.436616945	0.26338407	1	2	3
2	1016	269	1	1	0.1764408	0.655737705	0.603888275	0.544197094	0.48596117	1	1	2
3	970.5	321	1	1	0.11835738	0.68852459	0.568715972	0.509192275	0.41832948	1	1	2
4	933.5	261	0	0	0.17220397	0.716666667	0.374574557	0.508737267	0.46343765	2	2	4
5	995.5	297	1	1	0.14182024	0.7	0.531643876	0.494512195	0.49843206	1	1	2
6	1153.5	347	0	0	0.12038393	0.672131148	0.406552132	0.51098458	0.53208589	3	1	4
7	684	185	1	1	0.25114383	0.7	0.964845663	0.368729604	0.47833833	1	1	2
8	1191	370	-1	-1	0.10932467	0.672131148	0.392753978	0.501316361	0.50728881	2	2	4
9	1163.5	348	0	0	0.12073071	0.672131148	0.401961499	0.491340616	0.48204122	3	2	5
S	1381.5	361	1	1	0.13321292	0.806451613	0.404806022	0.507230855	0.48801866	4	1	5
P	1373	258	0	0	0.25920358	0.833333333	0.359458743	0.387080057	0.5956183	2	2	4
E	1402	225	1	1	0.3480109	0.783333333	0.438946765	0.505018416	0.61671271	1	1	2
D	1502.5	232	0	0	0.35079095	0.816666667	0.392884589	0.502035379	0.56101571	2	2	4
L	765.25	95	1	1	1.06553076	0.75	0.929793474	0.647883759	0.73137573	1	1	2
I	531	0	1	1	Inf	0.166666667	0.749915123	0.5	0.5	1	0	1
M	1920	290	1	1	0.28688979	0.95	0.317252214	0.496702518	0.49984203	4	4	8
T	914.5	152	1	1	0.49740071	0.766666667	0.600372024	0.348163842	0.5	1	1	2

There were many decision trees implemented for the OCR program. The one which produced better performance and result is shown in **Figure 1.3.1.**

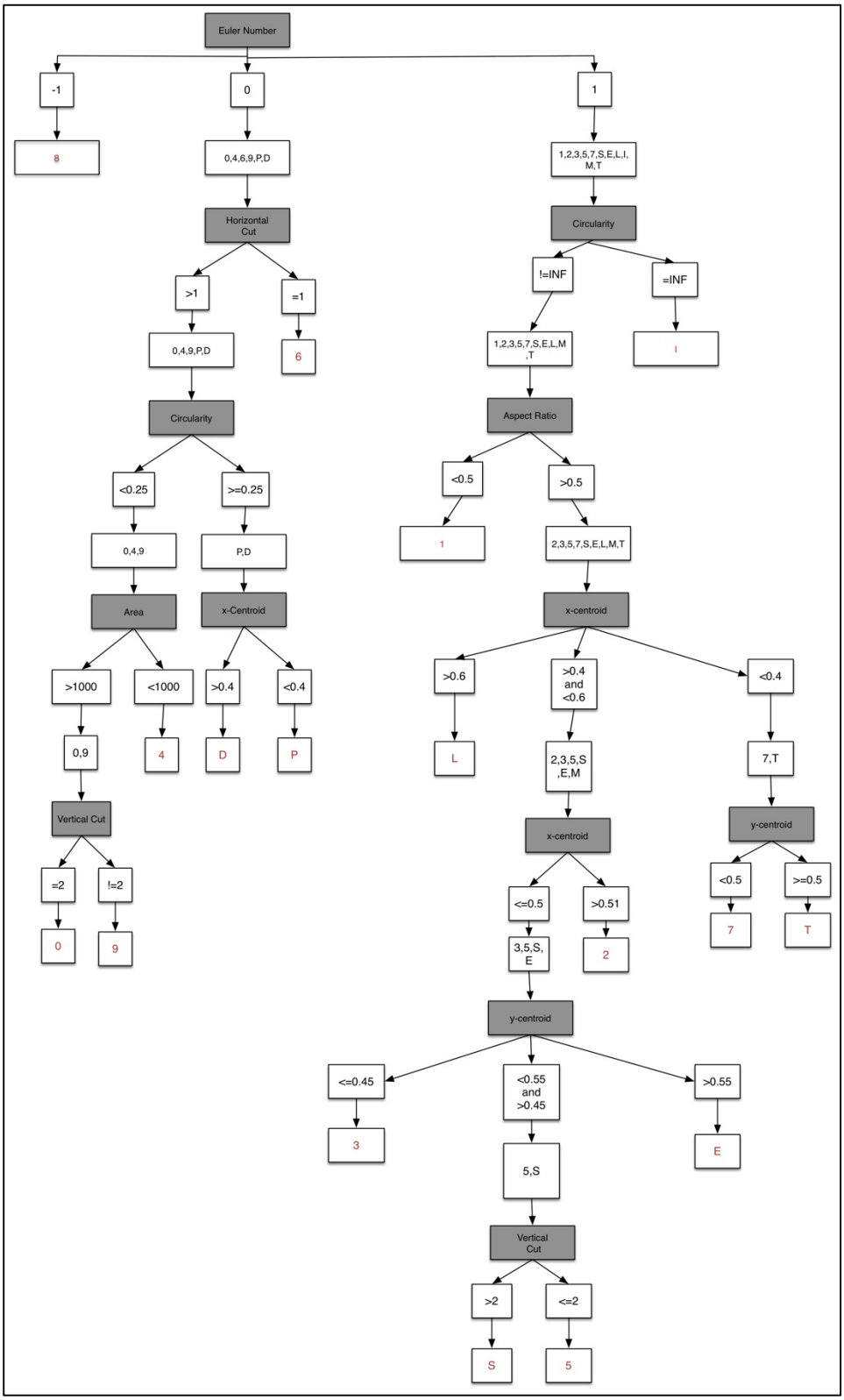


Figure 1.3.1: Decision tree for the implemented OCR program

In the decision tree, all the features used are placed inside a gray colored box. Along with it, when we reach a single symbol from a cluster of symbols going through various decision nodes, the symbol is written in red. After each feature box, we have a number of boxes appearing immediately which indicates the condition that needs to be satisfied for the feature and in the next line we have a number of boxes which indicates the symbols fulfilling the condition.

There were four test images provided for the problem. Before moving onto the results obtained by applying the OCR program on each of them, let us have a look at all four images **(Figure 1.3.2).**

Figure 1.3.2: (Left-Right): Testing images for the OCR program: Test_ideal1.raw; Test_ideal2.raw; Test_night.raw; Test_shade.raw.

It can clearly be observed from all the images that individual image requires their own pre-processing steps to be applied. The images that are obtained after applying pre-processing steps are shown in **Figure 1.3.3.**

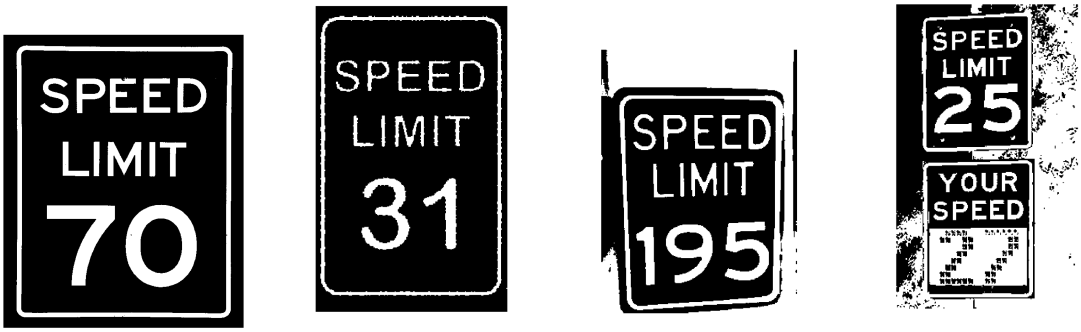


Figure 1.3.3: (Left-Right): Binarization of : Test_ideal1.raw; Test_ideal2.raw; Test_night.raw; Test_shade.raw.

For the third testing image with the text '**SPEED LIMIT 195'**, histogram equalization was applied to stretch the contrast of the image and make the content more visible. The image obtained as a result is shown in **Figure 1.3.4.**

Figure 1.3.4: Histogram Equalization applied on Test_night.raw

The result obtained for OCR program after applying the above mentioned decision tree is shown in **Figure 3.2.5-8.** Before the results are presented, it is to be mentioned that connected component labelling produces segmented images based on the order the pixel that appear in the image. Hence, we get segmented images in different orders. In order to verify the accuracy of OCR program, each obtained segments of pre-processed images were manually sorted with number tags for each symbols. The symbol included inside each tag are jotted down in **Table 1.3.2-5.**

Table 1.3.2: Segmented images tag for Test_ideal1.raw

Number Tags (for segmentation)	Content (Image-01)
1	0
2	7
3	S
4	P
5	E
6	E
7	D
8	L
9	I
10	I
11	M
12	T

Table 1.3.3: Segmented images tag for Test_ideal2.raw

Number Tags (for segmentation)	Content (Image-02)
1	1
2	3
3	S
4	P
5	E
6	E
7	D
8	L
9	I
10	I
11	M
12	T

Table 1.3.4: Segmented images tag for Test_night.raw

Number Tags (for segmentation)	Content (Image-03)
1	1
2	5
3	9
4	S
5	P
6	E
7	E
8	D
9	L
10	I
11	I
12	M
13	T

Table 1.3.5: Segmented images tag for Test_shade.raw

Number Tags (for segmentation)	Content (Image-04)
1	2
2	5
3	S
4	S
5	P
6	P
7	E
8	E
9	E
10	E
11	D
12	D
13	L
14	I
15	I
16	M
17	T
18	Y
19	O
20	U
21	R
22	2
23	7

The result obtained after applying the OCR program with the above decision tree are shown in **Figure 1.3.5-8**:

```
The output for segment 1 is 0
The output for segment 2 is 7
The output for segment 3 is S
The output for segment 4 is P
The output for segment 5 is E
The output for segment 6 is E
The output for segment 7 is D
The output for segment 8 is L
The output for segment 9 is I
The output for segment 10 is I
The output for segment 11 is M
The output for segment 12 is T
```

Figure 1.3.5: OCR result for Test_idea1.raw

```
The output for segment 1 is 1
The output for segment 2 is 3
The output for segment 3 is S
The output for segment 4 is P
The output for segment 5 is E
The output for segment 6 is E
The output for segment 7 is D
The output for segment 8 is L
The output for segment 9 is I
The output for segment 10 is I
The output for segment 11 is M
The output for segment 12 is T
```

Figure 1.3.6: OCR result for Test_idea2.raw

```
The output for segment 1 is 1
The output for segment 2 is 5
The output for segment 3 is 9
The output for segment 4 is S
The output for segment 5 is P
The output for segment 6 is E
The output for segment 7 is E
The output for segment 8 is D
The output for segment 9 is L
The output for segment 10 is I
The output for segment 11 is I
The output for segment 12 is M
The output for segment 13 is T
```

Figure 1.3.7: OCR result for Test_night.raw

```
The output for segment 1 is 2
The output for segment 2 is 5
The output for segment 3 is S
The output for segment 4 is S
The output for segment 5 is P
The output for segment 6 is P
The output for segment 7 is E
The output for segment 8 is E
The output for segment 9 is E
The output for segment 10 is E
The output for segment 11 is D
The output for segment 12 is D
The output for segment 13 is L
The output for segment 14 is I
The output for segment 15 is I
The output for segment 16 is M
The output for segment 17 is T
The output for segment 18 is T
The output for segment 19 is D
The output for segment 20 is M
The output for segment 21 is D
The output for segment 22 is others
The output for segment 23 is others
```

Figure 1.3.8: OCR result for Test_shade.raw

We can observe from the above results that the implemented OCR program is able to detect all the symbols present in Test_ideal1.raw, Test_ideal2.raw and Test_night.raw when compared with **Table 1.3.2-4**. In the final testing image (Test_shade.raw), the OCR program can detect almost every character accurately except for **YOUR** and **27,** which brings the accuracy rate of the OCR program down to 90%.

1.4 Discussion

Although the construction of decision tree for detecting symbols is quite straightforward, there are certain important points which needs particular attention in its choice of being used for the program. The important of them include the usage of connected component labelling for segmentation, binarization of grayscale image and dilation of the segmented images.

Why Connected Component Labelling An advantage of using connected component labelling is that it provides us with more automated control of symbol segmentation. We do not have to segment each symbol heuristically or use any primitive segmentation procedure to get our segmented symbols. However, there is a drawback that comes with this labelling method. Since the procedure can not distinguish between symbols and smaller segments, it comes with segmentations which are not part of the symbols. A naïve way to solve this problem is to pre-process the image such that only symbols are left for segmentations. However, we can also remove the unwanted segments by observing that the number of pixels required by symbols are more than it is for the other segments. We can write a subroutine which will remove the segmentations containing lesser pixel values. In this way, we can get rid of smaller segments.

Another advantage of using connected component labelling is that it does not require any sort of geometric modification and edge detection tasks to be performed since it segments based on the closed region it detects, which would still remain the same after geometric modification. Hence, the OCR program becomes more robust when segmentation is done using connected component labelling. However, such tasks can be performed to increase the performance accuracy of the OCR.
A drawback of connected component labelling is that it can not used if there are symbols with disconnected regions since it segments based on the connected regions. An example is **'i'**. In this case, connected component labelling would detect two segments: one is the dot on top of 'i' and the other one is the horizontal bar below the dot. Since our given testing images does not contain 'i', we can use connected component labelling without any problem.

Why proper thresholding matters Milyaev *et al.* [4] suggested that if proper binarization is done on the testing images, OCR program can perform better. In our case, the binarization was done using a thresholding sub routine as discussed. Although there are better binarization methods that can be applied to improve the quality of OCR, such methods are out of the scope of our given problem. As a whole, it was observed that the performance of OCR program was dependent hugely on the thresholding value chosen. It is common to choose 127 as the threshold value since it is the middle most intensity value. Since our given test images are not uniformly distributed, thresholding at 127 could not be the right choice. Four different thresholding values were chosen for each image during the binarization step (50 for Test_ideal1.raw, 27 for Test_ideal2.raw, 100 for Test_night.raw, 127 for Test_shade.raw). The choice of threshold value was obtained by looking at the histogram distribution for each image and writing a sub routine which would check for the segmentation of symbols based on the threshold value.

Why dilation is required Dilation was an important step in extracting features from the testing images. An example is presented in **Figure 1.4.1** where it shows how the edges of segmented image are not completely filled. Dilation is applied to fill the edges of the images so that the features can match to ideal training case. The result we obtain after applying dilation is shown in **Figure 1.4.2.** The image obtained gets closer to the ideal case. Hence, there is more probability of getting a closer result for same symbols in terms of shaped based properties.

Figure 1.4.1: Unfilled edges for testing symbols

Figure 1.4.2: Filled edges for testing symbols using dilation

Construction of decision tree As discussed before the construction of decision tree can be done arbitrarily. However, it would be really hard to reach a good performance rate with such type of construction. Hence, we need to apply our intuition by observing the properties for different characters and recognizing a factor which makes each symbol different from one another. As observed from the decision tree in **Figure 1.3.1,** the first parameter used was Euler Number, a parameter which almost provides similar value for similar set of characters. For example: 8 has two hole and one connected region which results in a Euler number of 1-2 =-1; 4,6,9,0, P, D have one hole and one connected region which results in an Euler number of 1-1 =0; the rest of the symbols have one connected region and no hole which results in an Euler number of 1-0=1. So, it is intuitive enough to start with Euler Number as way of comparing features. It was made sure during the pre-processing that such characteristic is hold by the characters so that they can be detected properly in the first stage.

Another two important characteristics that were found to be helpful while building the decision tree are centroids and cuts. It is intuitive to think that such characteristics will stay equivalent for similar symbols regardless of size, font. Because these depend on the geometric shape of the symbol, they provide a better solution for feature comparison.

There were certain unique features that were used to distinguish among characters. For example: **I** has no circularity compared to other symbols with Euler number of 1. So, circularity was used for separating **I**. **1** has a lesser aspect ratio compared to other symbols with Euler number of 1. So aspect ratio was used in this case. For the other symbols with Euler number of 1, centroids and cuts were mainly used. A similar mechanism was followed throughout the entire construction of decision tree.

However, some of them did not work. In those instances, hard tweaking was used to construct the decision tree.

A characteristic that was tried for feature comparison was symmetry. However, it was observed that when applied on the given test images, it requires a lot of manipulation in pre-processing step. For example: 0 has both horizontal and vertical symmetry; but in case of the first test image it was found that 0 was oblique to some angle.

It is to be mentioned that the constructed decision tree took almost a week to come up with. The decision tree which is found to provide an accuracy rate of almost 95% was also applied on other test images. Some of the test images are shown in **Figure 1.4.3** and the results were astonishingly great. An accuracy of 85% was obtained from the tested images. Hence, it shows to the least that the implemented decision tree holds some water in terms of its feature comparison for symbols.

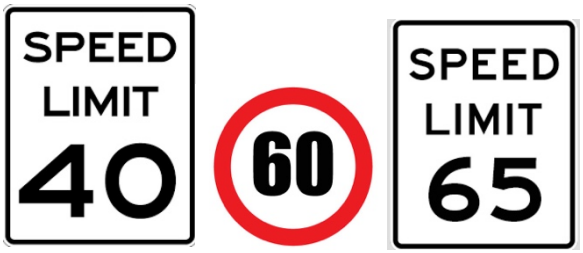


Figure 1.4.3: Testing images downloaded from online.

2 Problem 2

2.1 Abstract and Motivation

Contour Modelling is widely used in segmentation to find the region of interest (ROI). It is mainly used in different types of medical images such as, Magnetic Resonance imaging (MRI), Computed Tomographic (CT) Scan, X-Ray, Ultrasound to find the contours of different regions of interest including brain tumors, spine contours, separating red and white blood cells or outline of coronary. The basic idea of contour modelling is that a region of interest beforehand in an image is specified beforehand. The algorithms, based on the values provided, finds the contour of the region. In this problem, we will look onto two algorithms called the snake algorithm and the level-set algorithm to check the performance of each of this on contour modelling.

2.2 Approach and Procedure

2.2.1 Snake Algorithm

A snake is an energy-minimizing 2-D spline curve that evolves towards image features such as, intensity, edges [5]. Mathematically, it is defined by a set of n points p_i, where $i = 1,2,3,4, \dots n$, the internal energy $E_{internal}$ and the external edge-based energy $E_{external}$. The goal of the internal energy is to control the deformation i.e. elasticity that can be made to the contour whereas the goal of the external energy is to control the fitting of the contour onto the image. The external energy can be divided into two parts: one of which is based on the image itself and the other is based on the constraint introduced by the user denoted by E_{image} and $E_{constraint}$. The ultimate goal of the snake algorithm is to minimize the energy function as much as possible till it reaches contour possible. The energy function of the snake can be represented as follows:

$$E_{snake} = \int_0^1 E_{internal}\left(p(s)\right) + E_{image}\left(p(s)\right) + E_{constraint}\left(p(s)\right) ds$$

The internal energy is again composed of two energy functions denoted by E_{cont} and E_{curve}, which control the smoothness of the contour and continuity of the contour respectively. There are two parameters related to both the energy functions denoted by α and β, which are defined by the users. In practice, a large value of α affects changes in distances between points in the contour whereas a large value of β affects the oscillations. The overall formulation of internal energy can be written as:

$$E_{internal} = \frac{1}{2}\left(\alpha(s)\left\|\frac{dp}{ds}(s)\right\|^2 + \beta(s)\left\|\frac{d^2v}{ds^2}(s)\right\|^2\right.$$

The image energy is one of the most important points of this algorithm. Features can be added as much as wanted in this term. However, the ones that are given the most priority are edges, lines, and terminations. There are various derivative methods using which each of these can be calculated. The line functional energy calculates the intensity of the image i.e. it differentiates between dark and light portions of the image. The edge functional energy looks for the edge in the image, which are content with high frequencies. The calculation is based on the image gradient. The termination function is

calculated from the curvature of level lines in a smoothed image, which is used to detect corners and terminations in the image. Hence, the entire formulation of the image energy can be written as follows:

$$E_{image} = w_{line}E_{line} + w_{edge}E_{edge} + w_{termination}E_{termination}$$

There are certain algorithm which provides the user with the freedom to control the energy called $E_{constraint}$ – not only allowing to choose the region of interest but also control the energy terms such that user gets the freedom to control the features in and away from the region of interest in terms of energy. However, in our implementation we ignore this factor and take care of the above-mentioned energy functions.

The implementation of snake algorithm is based on greedy algorithm [6]. A greedy algorithm makes locally optimal choices such that final solution will be globally optimum. In the first step, each point of the snake is moved within a small neighborhood to the point which minimizes the energy functional. In the second step, it searches for the corners along the extreme.

In terms of programming implementation, various online sources codes [7,8,9] were attempted to get a proper result based on the algorithm discussed above. The one that was found to be more interactive and based on the actual algorithm is developed by Ritwik [8]. The code allows us to load a new image and filter the image using a Gaussian filter with a value for sigma. The steps that are followed to use the program is as follows:

- Load the new image and filter it with a Gaussian filter of desired sigma value.
- Choose an initial contour around which final contour is expected.
- Specify the value for the parameters according to the algorithm's necessity.
- Iterate the process till the desired region of interest is achieved.

The parameters that are asked in the given algorithm are:

Alpha: It controls the tension of the snake contour. This tension force acts to keep the curve from stretching a lot based on the value provided. A larger value makes it possible whereas a larger value makes it relatively flexible.

Beta: It controls the rigidity of the snake contour. It acts to keep the curve away from bending excessively. A smaller value makes the curve harder to bend whereas a larger value makes it more flexible to bend.

Gamma: It determines the step size of the contour. It can also be related to viscosity which control how quickly and how far the curve can be deformed between iterations. A larger values makes it harder and slower to deform.
Kappa: It controls the energy term of the contour for a larger value of whose a stronger force towards the edges is caused.

W_l: It determines the weight for line energy function.
W_{edge} : It determines the weight for edge energy function.
$W_{termination}$: It determines the weight for termination energy component.

An example of initial contour is provided in **Figure 2.2.1** to display how the contour is specified and parameters that are used in the algorithm

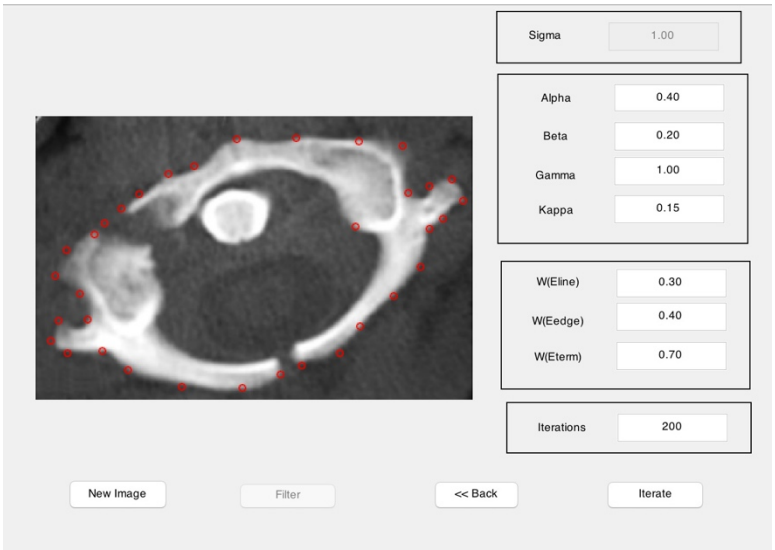


Figure 2.2.1: Initial Contour drawn on Spine.raw for snake algorithm

Despite its faster performance, snake algorithm does not solve the entire problem of finding contour in images. It purely depends on the user's choices in terms of initial contours, parameters determination, and number of iterations. At times, such procedure becomes tiresome specifically for a medical application. Hence, there are certain algorithms which does the contour modelling job with just fewer requirements from the user. The level-set algorithm does quite a good job in that.

2.2.2 Level-set Algorithm

Another algorithm that can be used to implement contour modelling is called level-set algorithm [10]. A level-set evolves the curve as the zero-set of a characteristic function. It can easily change topology and incorporate region-based statistics. The algorithm of level-set is mathematically intensive based on the condition that are placed on it. However, we will go through a high level understanding of the algorithm.

The level set algorithm for closed contours attempts to find the zero-crossings of a characteristic function defined by the user around the contour. In our case, it is a rectangle. Based on the characteristic function, it fits and tracks objects of interest based on the image features such as, edge, intensity, corner by modifying the underlying embedded function instead of the curve used for snake algorithm. As we can observe we need to find zero-crossings at each point and it needs to be updated

in each step. Here, we use a fast marching algorithm to achieve this in a robust fashion. The implementing of fast marching algorithm is based on the fact that the level set propagates inward. The steps followed in implementing the fast marching algorithm are as follows:

Step-1: We start with an initial time **T** at which contour crosses the grid point.
Step-2: We tag known initial value and update neighboring **T** values using difference approximation.
Step-3: We take another unknown value for T with the smallest **T** obtained previously.
Step-4: We keep on updating new neighbors until all nodes are known.
Step-5: We store all the unknowns obtained in a priority queue and we get the set of points reached at time **T** from the surface calculation.

In terms of programming implementation, online source code was used [11]. There is no requirement of details to be provided while initiating the contour. In this case, we take a level set within the limited area of the image. It was found from experimentation that it is better to keep the level set around the region of interest to get better performance. The steps followed to implement the level-set algorithm from the online source code is as follows:

- Load the image in .jpg format.
- Specify the level set region around the image.
- Specify the values of inner and outer iterations.
- Specify lamba, alpha, and eplison which are the coefficients of weighted length, weighted area and width of the step size function respectively. It was observed that only specifying alpha is enough for the algorithm to give a better performance.
- We can also specify the sigma for Gaussian filter kernel used to smooth the image.

An example of specifying initial level contour around the image is shown in **Figure 2.2.2.**

Figure 2.2.2: Initial Contour drawn on Spine.raw for level-set algorithm

2.3 Experimental Results

Before we start our discussion on results obtained, the images used for testing in this problem are shown in **Figure 2.3.1**.

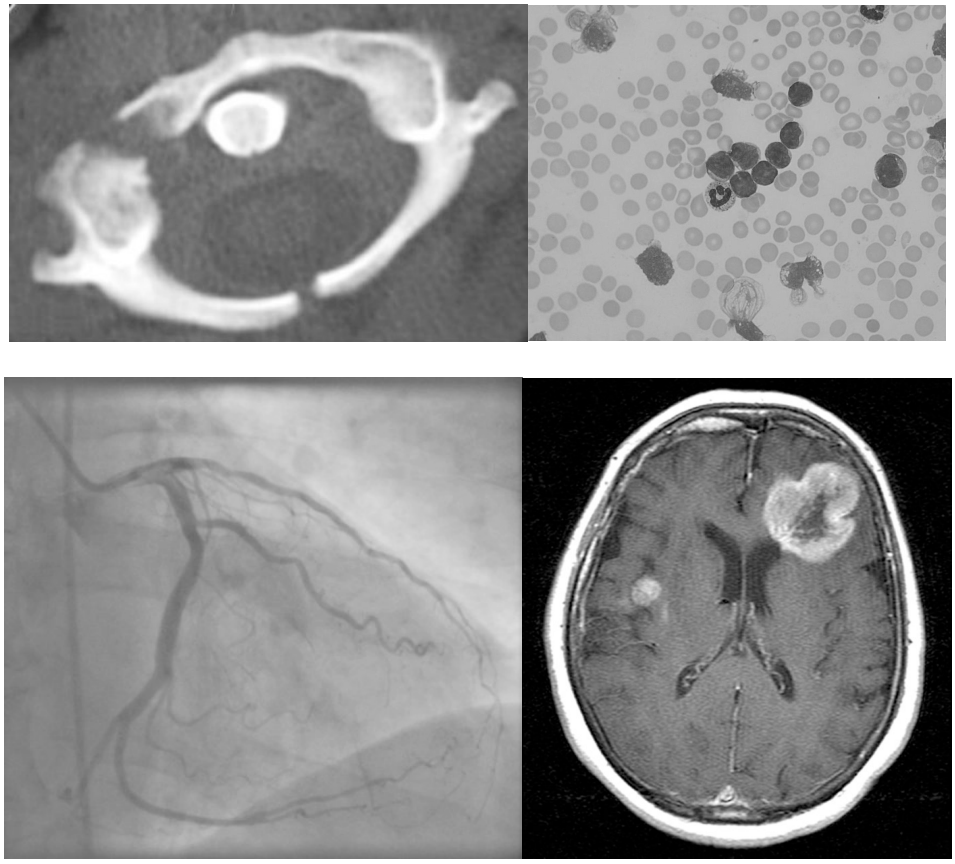


Figure 2.3.1: (From left to right): Images provided for the problem: spine.raw, blood_cells.raw, coronary.raw, brian.raw

The results obtained from snake level algorithm using the online source code is shown in **Figure 2.3.2(a-i).** For the spine image, the snake level algorithm is done with only the outer contour and with both outer and inner white contours. For the coronary angiogram image, the snake level algorithm is applied on both with details of angiogram and only the outline of the contour. Since snake algorithm allows to choose one contour region at a time, the contour is drawn for the tumor image separately for the bigger and smaller tumor. For the blood cell image, it is quite difficult to separate the red and white blood cells with snake algorithm. However, in **Figure 2.3.2(i),** it is proposed that we can take contours for white blood cells separately and attach them together even though it is a lot of work. We can also do the same for tumor image where both smaller and bigger tumors can be attached together in an image.

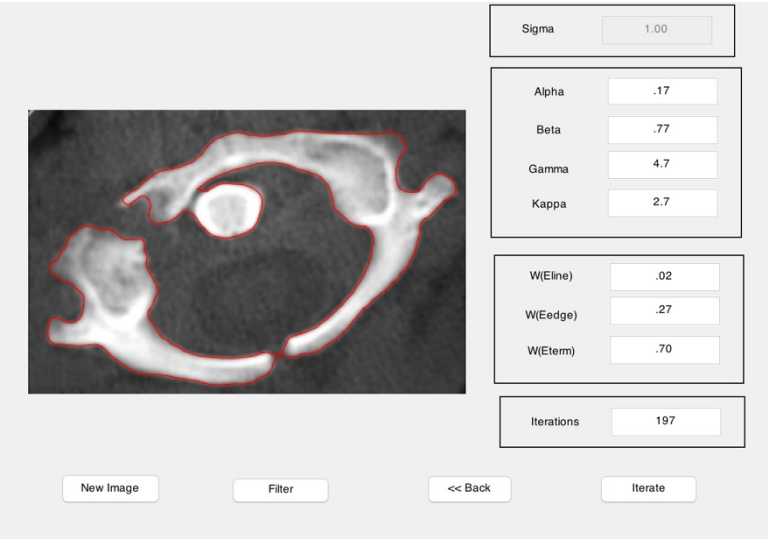


Figure 2.3.2(a): Snake algorithm on spine.raw (inner and outer)

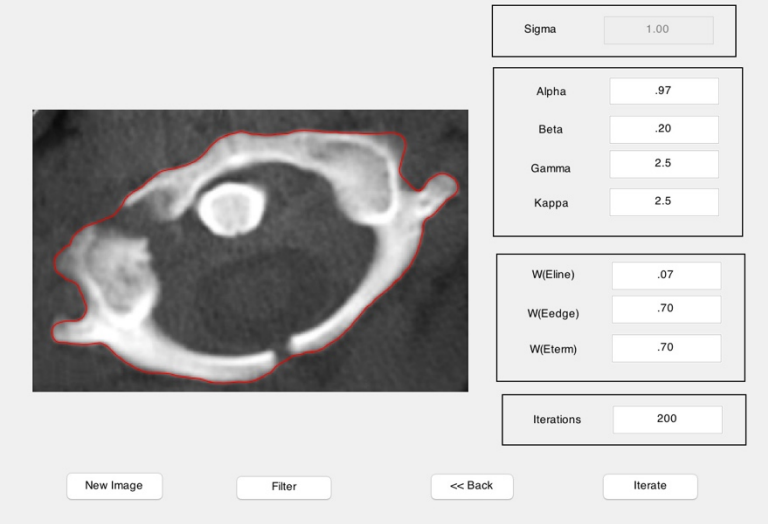


Figure 2.3.2(b): Snake algorithm on spine.raw (Outline)

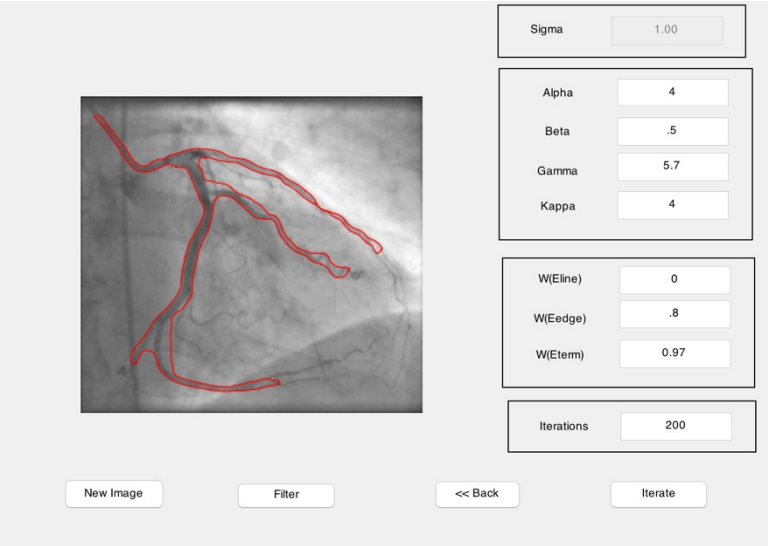


Figure 2.3.2(c): Snake algorithm on coronary.raw (inner and outer)

Figure 2.3.2(d): Snake algorithm on coronary.raw (outline)

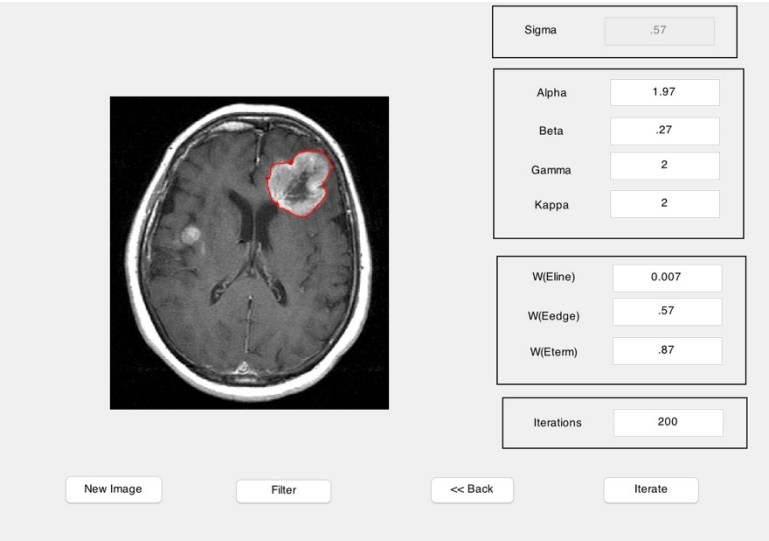


Figure 2.3.2(e): Snake algorithm on brian.raw (bigger tumor)

Figure 2.3.2(f): Snake algorithm on brian.raw (smaller tumor)

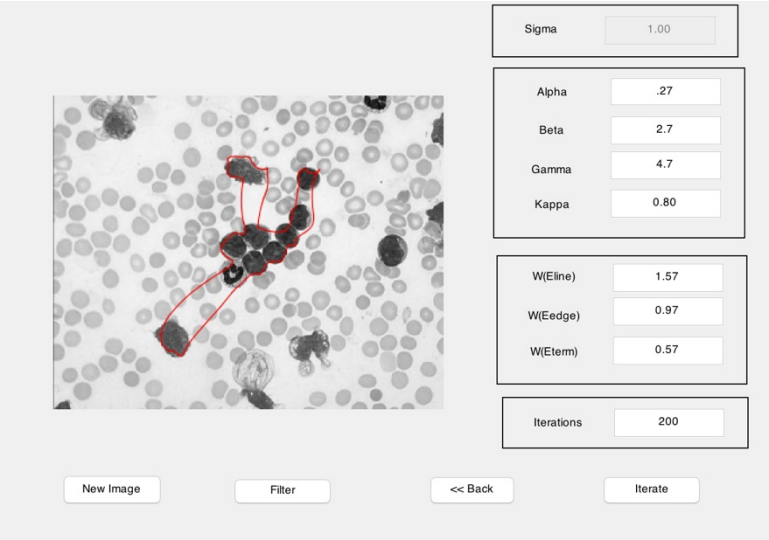


Figure 2.3.2(g): Snake algorithm on blood_cells.raw (trial-1)

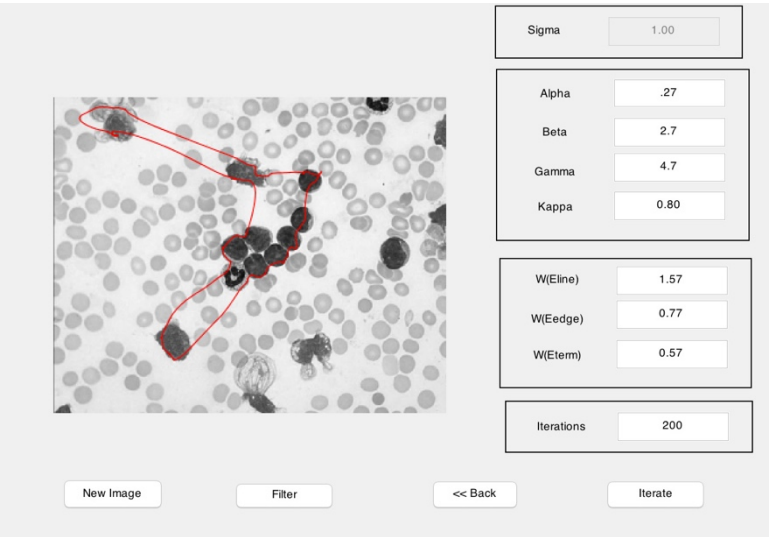


Figure 2.3.2(h): Snake algorithm on blood_cells.raw (trial-2)

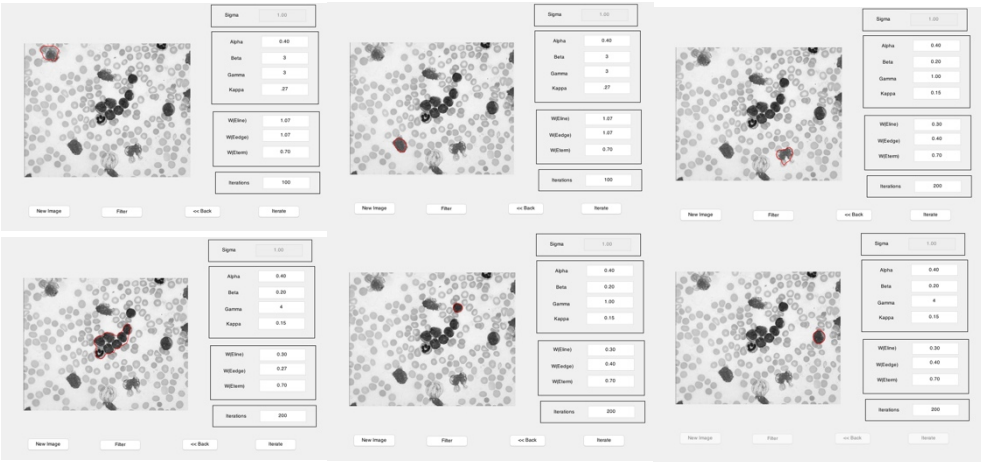


Figure 2.3.2(i): Snake algorithm on blood_cells.raw (smaller segments)

The results obtained by applying Level set algorithm are shown in **Figure 2.3.3 (a-e)** along with initial and final level set functions. There is one result for spine, coronary and blood cells images. There are two results for brain MRI image. There are two images for MRI image since trial on detecting both tumors were tested.

The values for which each figure performed well are shown in **Table 2.3.1.**

Table 2.3.1: Parameters for Level-Set algorithm for four medical images

Figure	Inner Iteration	Outer Iteration	Alfa	Sigma
2.3.3 a	10	45	1.67	1.87
2.3.3 b	19	68	1.68	1.67
2.3.3 c	37	15	1.60	4.80
2.3.3 d	50	41	1.47	2.70
2.3.3 e	20	70	2.00	5.00

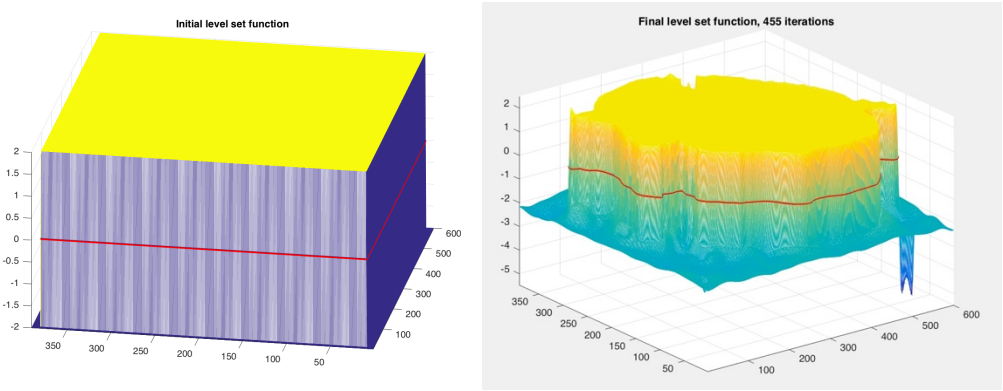
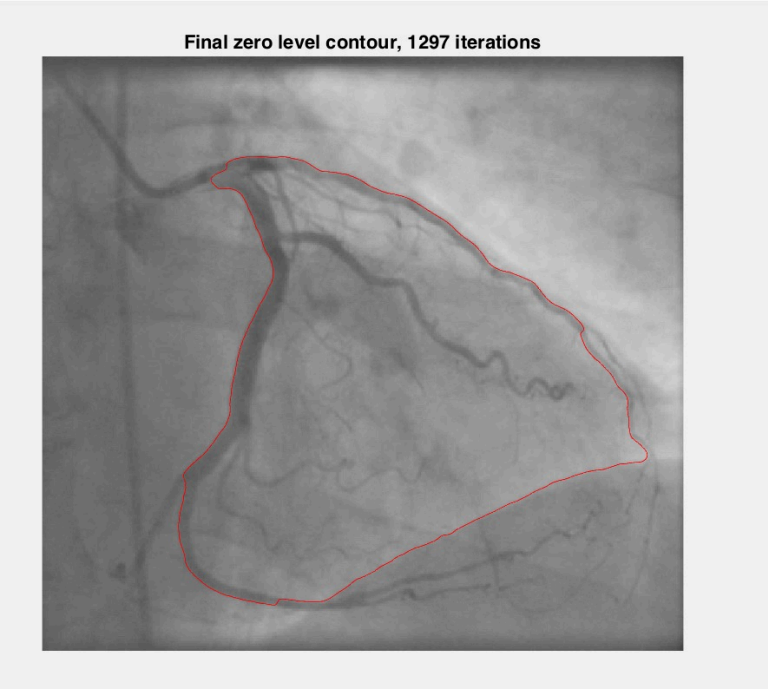


Figure 2.3.3(a): Level-set algorithm on spine.raw



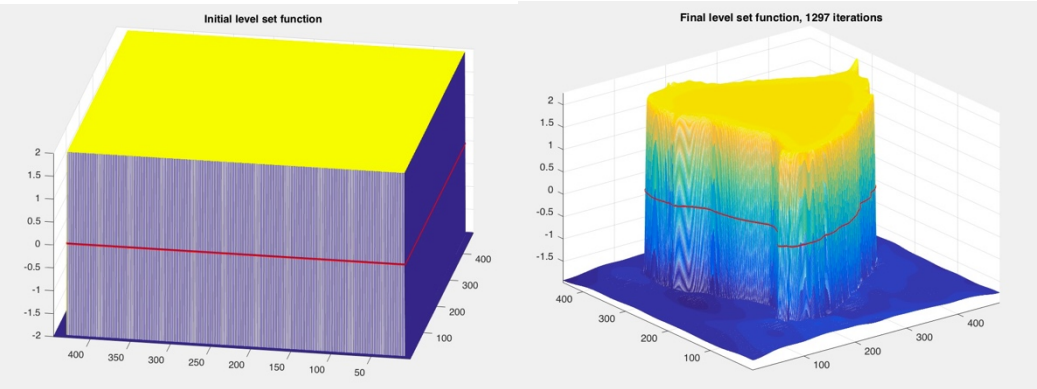
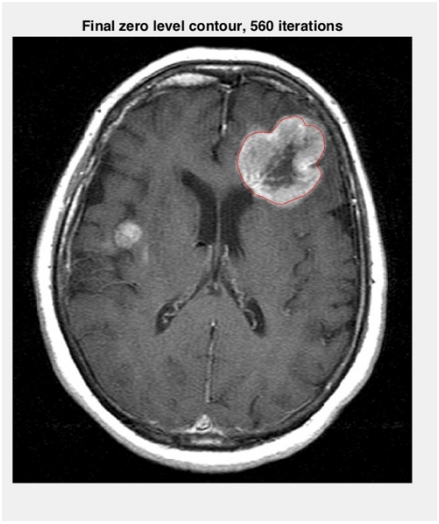


Figure 2.3.3(b): Level-set algorithm on coronary.raw



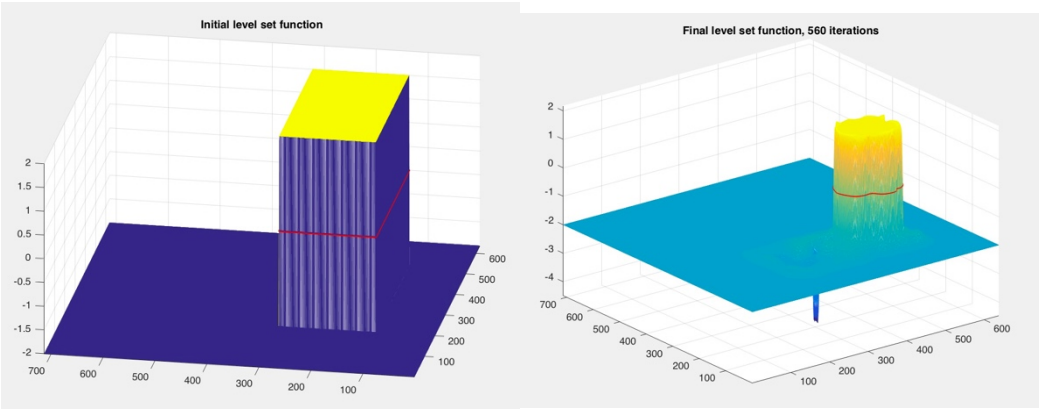
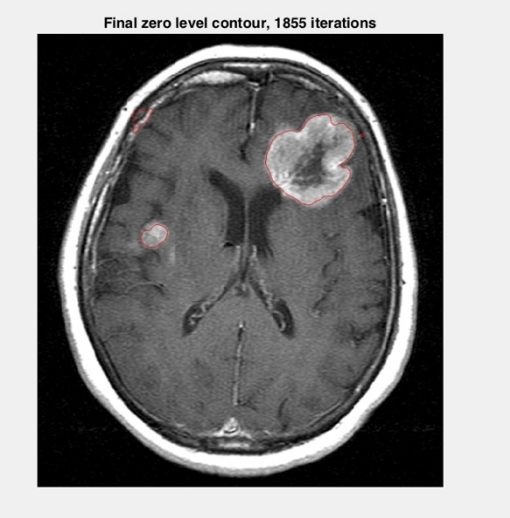


Figure 2.3.3(c): Level-set algorithm on brian.raw (bigger tumor)



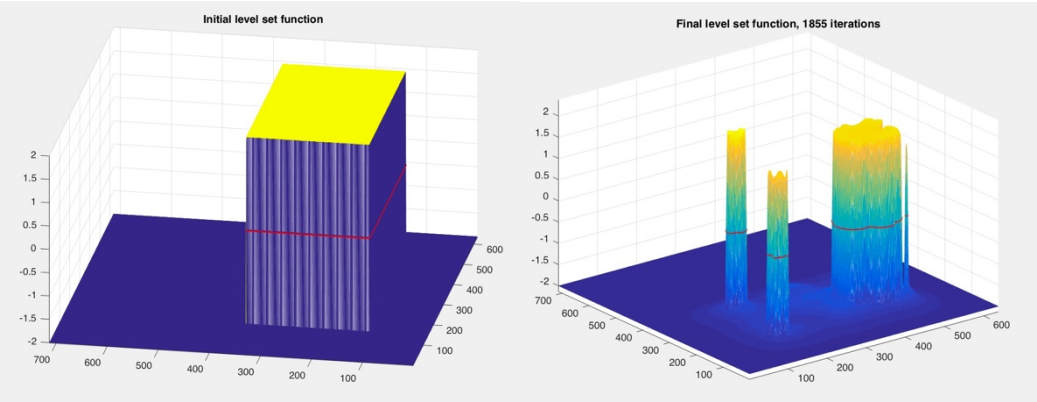


Figure 2.3.3(d): Level-set algorithm on brian.raw (both tumors)

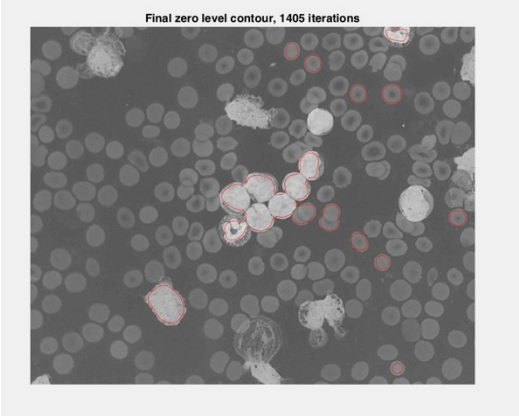


Figure 2.3.3(e): Level-set algorithm on blood_cells.raw

The result for blood cells was made better by applying a threshold and segmenting the image using level-set algorithm. The result is shown in **Figure 2.3.4.** Similar values were applied like the previous **Figure 2.3.3 (e)** and if looked closely white portions (white blood cells) are circled red in the image.

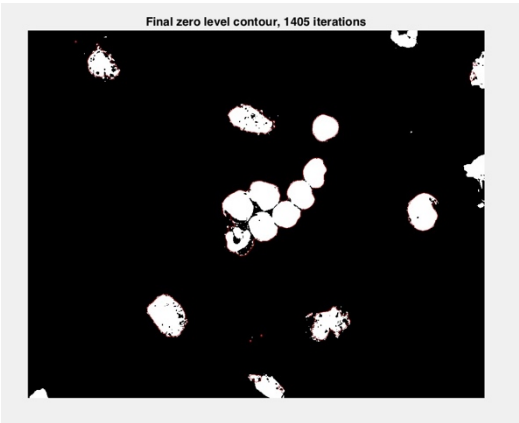


Figure 2.3.4: Level-set algorithm on blood_cells.raw using thresholding.

2.4 Discussion

Despite its faster performance, snake algorithm does not solve the entire problem of finding contour in images. It purely depends on the user's choices in terms of initial contours, parameters determination, and number of iterations. Snake algorithm provides the user with the freedom to achieve only one active contour. As a result, the active contours needs to be achieved separately as observed in **Figure 2.3.2(e-i).** At times, such procedure becomes tiresome specifically for a medical application. In this case, level-set algorithm does a better job by finding out the iterations through the algorithm as seen in **Figure 2.3.3(c-f).** Although there are certain pre-processing we need to make for level-set algorithm (as done for blood cells image), the performance of the level-set algorithm is certainly better than snake algorithm. Separating the red and white blood cells was a difficult task for both the algorithm; but level-set algorithm outperformed snake algorithm in terms of getting the contour of several white blood cells separated from the red blood cells.

A drawback of level-set algorithm is that it takes a lot of computational energy and time to perform contour modelling. To get the output of blood cells and tumor images, it almost took sixe and three minutes respectively for each image to solve the contour. In this case, snake algorithm is a definite plus. Since it calculates for minimization of energy functions, it needs lesser iterations to deal with which reduces the time required for contour modelling. However, the speed of level-set algorithm can be increased by using parallel processors since it uses fast marching algorithm whose values can be computed in different processors.

A huge difference between snake and level-set algorithm that can be observed from both algorithm and results obtained is the amount of details that snake algorithm takes into consideration than level-set. Snake algorithm takes more details into consideration than level-set which is evident from the energy functions. An example is shown in **Figure 2.4.1** for the spine image, in which snake algorithm taken intensity of smaller details into consideration whereas level-set algorithm skips it. Another example of details that can not be achieved with level-set algorithm is including both outer and inner contour as shown in **Figure 2.4.2.**

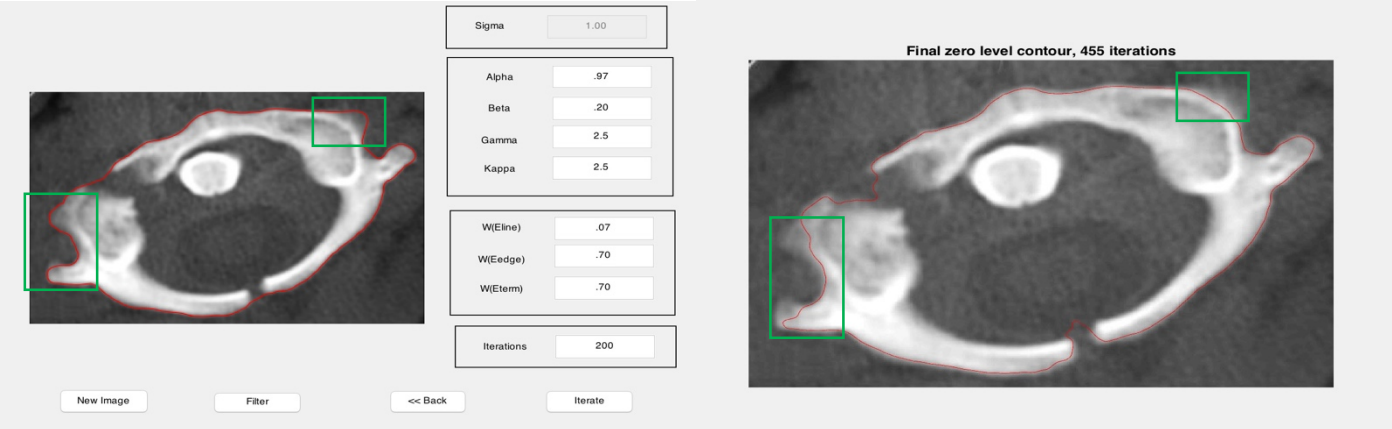


Figure 2.4.1: (Left-Right): Difference on intensity levels between: (Left): Snake algorithm; (Right): Level-set algorithm

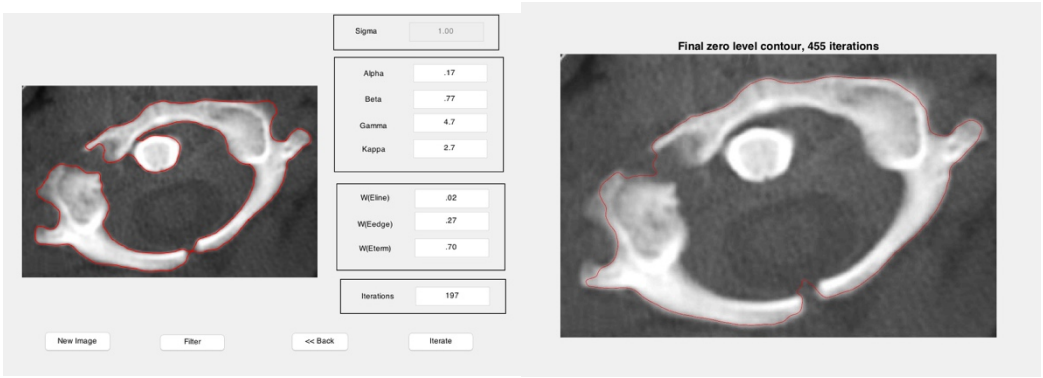


Figure 2.4.2: (Left-Right): Difference on details between: (Left): Snake algorithm; (Right): Level-set algorithm

Effect of initial contour in Snake Algorithm In case of snake algorithm, the initial contour defined by the user is a sensitive case. The initial contour needs to be closer to the region of interest. If the chosen contour does not follow the path of region of interest, a similar set of parameters will not work for the contour. In that case, we need to tweak our values more according to the contour chosen. An example is shown in **Figure 2.4.3** where two different initial contours were chosen and same set of parameters were applied. The final output show that it is possible to get different result if the initial contour is not chosen carefully for snake algorithm Although the initial contour chosen in both cases in almost in the same region of interest, choice of details makes a difference in getting the right output. It is evident from the figure that snake algorithm is not as robust and provides less freedom to the user in terms of initial contour definition.

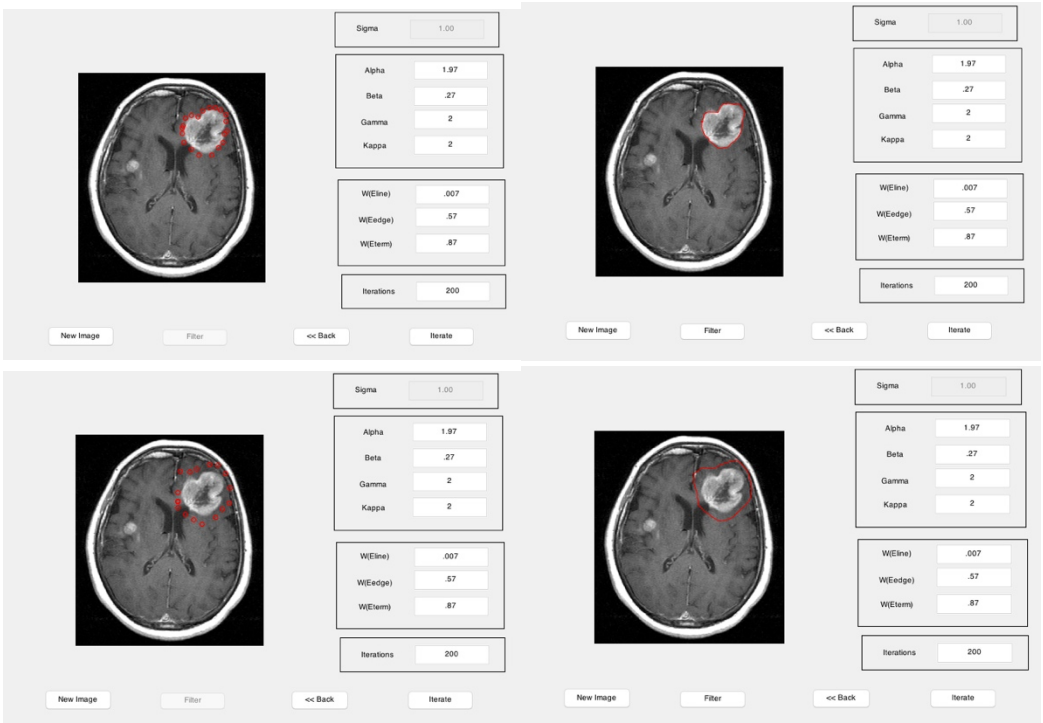


Figure 2.4.3: Effect of initial contour on snake algorithm

Effect of inner and outer iterations in Level-set algorithm The effect of inner and outer iterations become evident on level-set algorithm as we get used to the algorithm. An example is shown in **Figure 2.4.4** to illustrate how the difference between inner and outer iterations affect the result. It is observed from the results that as we decrease the difference between inner and outer iterations, we get two tumors separated almost accurately. The reason is actually behind the number of iterations increasing in the process. As we make inner and outer iterations go higher with their difference decreasing, we increase the number of iterations in the process, which makes the algorithm run for a longer time and gets refined from its pre-defined level sets. Hence, if we increased the inner iterations to 75 and outer to 73, we could get two tumors perfectly detected. However, it was beyond the scope of the system to achieve such result.

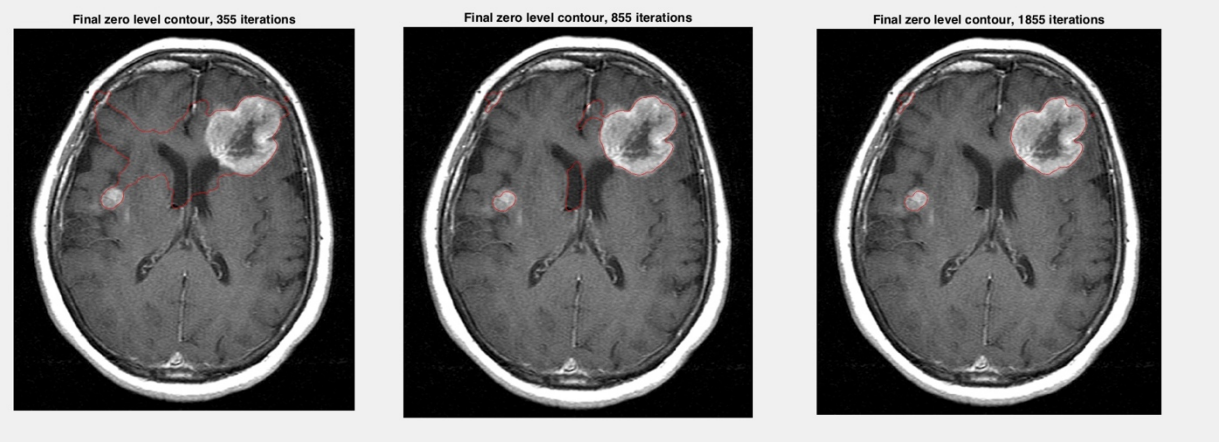


Figure 2.4.4: Effect of inner and outer iterations on Level-Set algorithm

From the above discussion, it is observed that both algorithms have their advantages and drawbacks. The choice of their usage depends on the type of problem we are trying to tackle. If we have a simpler image like spine or coronary whose contour we are trying to model, it is better to use snake algorithm in this case since the performance for both will be almost similar. In this case, time is an advantage that snake provides. If we are looking for more detailed contour modelling, snake algorithms would be more preferred. In case of complex images with many contours to be detected, level-set algorithm is a better option even though it comes with the downside of computational time. There are other state-of-the-art algorithms which can be used to model contours and image segmentation. However, it should be kept in mind that both algorithms require some sort of knowledge of the desired region of interest beforehand, specially snake algorithm.

3 Problem 3

3.1 Abstract and Motivation

Extracting salient feature points in an image is one of the important topics in computer vision. Such extraction of salient feature points provides us a guideline to find image matching, which has become a fundamental aspect of many problems in object or scene recognition, solving 3D structure from multiple images, stereo correspondence, and motion tracking. Recently, Facebook is developing their scene recognition system in a video by using the salient feature points.

In this problem, we will take a look into two important algorithms discussed in the literature of this newly developed topic. The first one is Scale Invariant Feature Transform (SIFT) [12] developed by David Lowe and the second one is Speeded-Up Robust Features (SURF) [13] developed by Herbert Bay *et al.* Although an in-depth discussion of the algorithms is beyond the scope of our problem, a general picture will be established such that we can build our discussion of the topic on it. Along with image matching, we will be using the feature extraction algorithms to build a bag of words and find closeness among images using K-means clustering.

3.2 Approach and Procedure

3.2.1 SIFT Feature Extraction Algorithm

SIFT was one of the first algorithms developed which provided a way to detect the salient points in an image and thus solve image matching problems. The idea of SIFT is to transform an image content into local feature coordinates that are invariant to translation, rotation, scale, illumination, viewpoint and other imaging features. The overview of algorithm implemented in SIFT is as follows [14]:

Step-1: We first construct a scale space using Gaussian Kernel.
Step-2: We take Difference of Gaussians (DoG), which is an approximation of Laplacian of Gaussians (LoG). We use DoG since it requires lesser computational cost than LoG.
Step-3: We locate the extrema of the DoG by scanning through each DoG images and identifying the minimum and maximum.
Step-4: We use 3D-Curve fitting to localize the sub-pixels. We can use Taylor Series Expansion in this case.
Step-5: We filter low contrast points by using scale space value at previously obtained locations.
Step-6: We filter edge responses using Hessian matrix where we use traces and determinants to find out the eigenvalues proportional to principle curvatures.
Step-7: We assign key point orientations by first creating histogram of local gradient directions at selected scale, then assigning canonical orientation at peak of smoothed histogram where each key specifies stable 2D coordinates.
Step-8: We build key point descriptors where each key point has location, scale, orientation. In this step, we find the blurred image of the nearest scale. Then, we sample the points around the key point. Next, we rotate the gradients and coordinates by previously computed orientation. After that, we

individualize the region into sub regions. Finally, we create the histogram for each sub region with a particular number of bins.

Hence, in this algorithm we have taken care of scale invariance, rotation invariance and viewpoint variance by usage of scale space, alignment of largest gradient and small viewpoint changes respectively. We need to be careful about illumination changes for which we need to normalize the vector.

In terms of programming, the implementation of SIFT is done using online source codes. We use the source code developed by **vlfeat** [15] for SIFT feature extraction. We also use the same source code for testing image matching between two image with SIFT descriptors. The implementation by the original author was tried for the given algorithm as well [16]. However, it turned to be really slow in computation and had redundant functions not used in MATLAB anymore.

3.2.2 SURF Feature Extraction

There are various claimed advantages of SIFT algorithm in the literature. An important one of them is that many features can be generated with the algorithm even for a smaller object. Another important one is robustness attained due to locality of features detected. However, there are other algorithms which performs similar tasks at a shorter time with more efficiency and robustness. One of them is SURF algorithm.

An important concept we need to understand before discussing the steps followed in SURF algorithm is creating integral images. The goal of creating integral images is the fast computation of mask convolutions regardless of the size of the filter. An integral image has the same size as the analyzed image and the value of integral image at any location is the sum of the intensity values for all locations lesser than or equal to the location at which integral image intensity is evaluated. This is how an integral image is created. Once the integral image is created, we can apply the steps as follows to implement SURF algorithm [17]:

Step-1: We find the interest points in the image by calculating the determinant of Hessian matrix, which is nothing but the product of eigenvalues since Hessian matrix is a square and symmetric matrix. The derivative filter that is used in the Hessian matrix is usually LoG, a second-order derivative filter. We use the integral image to get the sum of the intensities for the squares present in Hessian box filters which is multiplied by the weight factor and finally add the resultant sum together. We can use a threshold to evaluate the filter size where we can control the number of interest points we want to detect.

Step-2: We find major interest points in a scale space. Bay defines a term called **Octave** which is a series of filters having a range approximating double of a scale and it is used ensure full convergence of each scale. To find the main features, we use a 3x3 non-maximal suppression with the same scale space of thresholding used in the first step. We also do non-maximal suppression above and below

the scale space for each octave. We need to interpolate the interest points to reach at a correct scale since the scale space is coarsely scaled.

Step-3: We find the feature direction using Haar Transform. First, we look at pixels available in a circle with a radius of 6 times our scale space. Then, we compute both x & y Haar transform for each point. Next, we use the outcome as coordinates in the Cartesian map. Finally, we choose a direction of maximum total weight to get the feature direction.

Step-4: Finally, we generate the feature vectors, which is a 64 dimensional vector consisting of 4 values denoted by $\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|$ for 16 sub-regions created from a square description window, which is constructed with a size of 20 times the scaled space for each interest point. d_x and d_y are the Haar wavelet response in the horizontal and vertical directions respectively. They are calculated in the rotated direction. Hence, we have our feature vectors which are called SURF descriptors.

The algorithm described above has more description to each step. However, an in-depth discussion would be a topic of itself. Hence, we focus more on the implementation of it in our programming. An advantage of SURF compared to SIFT is that it is 3-5 times faster because it can be adapted to parallel processing. The authors claim it to be more resilient to noise than SIFT. But as observed from the procedure, SURF is less invariant to illumination and viewpoint change which are only broadly taken care in the algorithm.

In terms of programming implementation, SURF descriptors can be obtained easily using **detectSURFFeatures** included in the Computer Vision Toolbox of MATLAB. The function requires the image to be converted to a gray-scale image beforehand. It creates a *SURFPoints* class from which we can choose random number of features to be plotted on the image. However, for our purpose we choose 100 strongest features and try to check whether we get important features detected or not.

3.2.3 Bag of Words Algorithm

When we use SIFT feature extractor to images, we usually get the local features of individual image. However, when we want to classify images using local features, it is not quite straightforward to use the traditional image classification method since they use global features of images to serve the classification. The technique that can be used to perform image classification using local features of images is called **Bag of Words (BOW).** The simplified BOW algorithm can be laid as follows:
Step-1: We extract local features of image using SIFT feature extractor.
Step-2: We place all the local features vectors into a single set for evaluation.
Step-3: We apply a clustering algorithm (in our case K-means clustering algorithm) over the set of feature vectors to find the centroid coordinate and assign a codeword to each centroid. This is how we build our vocabulary which is of length K.
Step-4: We create a global feature vector where we create a histogram which counts how many times each centroid occurred in each image.

Step-5: As we have our feature vector based on global identities, we can use any traditional image classification algorithm. In our case, since we have a smaller set of training data, it would be better to use K-nearest neighbor algorithm. For a larger set of training data, it is optimum to use Support Vector Machine.

In terms of programming implementation, we use online source codes as well as MATLAB's built-in function. We apply bag of words on both SIFT and SURF descriptors to check the performance of both. Although both of them will provide same result for such a smaller set of images, we can analyze the codewords for each. In case of SIFT implementation on BoW, we use an online source code developed by **Mopuri** [18] to create the bag of words. In case of SURF implementation, we use MATLAB's built-in function **bagOfFeatures** and **encode** included in Computer Vision Toolbox. Both of the implementation uses K-means clustering for creating the codeword. In our case, K is equal to 8 bins. In case of MATLAB's built-in function usage, we have to create an **imageSet** class by calling the function with an input of directories in which images are stored.

After we get our codewords created for both training and test images, we implement the classification using MATLAB's built-in function **knnsearch**, which provides a label for the testing data to the training data to which it closely resembles to. The applied function uses k-nearest neighboring for the classification.

3.3 Experimental Results

Before we start our discussion on results obtained, the images used for testing in this problem are shown in **Figure 3.3.1**.

Figure 3.3.1: (From left to right): Images provided for the problem: Bus.raw; Sedan.raw; School_bus1.raw; School_bus2.raw

The results obtained by applying SIFT feature extraction algorithm on Bus.raw and Sedan.raw are shown in **Figure 3.3.2** for a random collection of 100 strongest features.

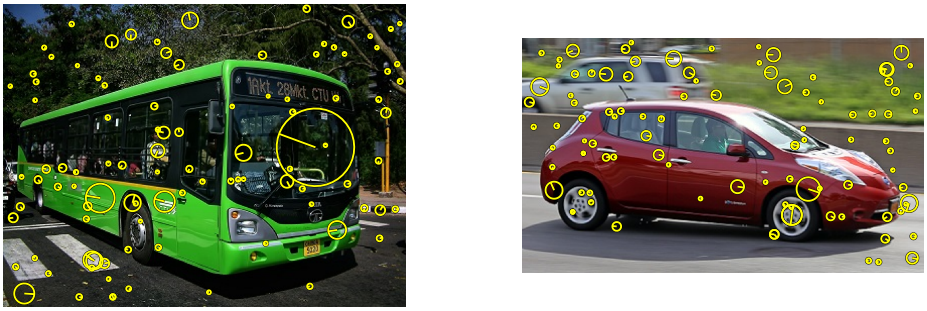


Figure 3.3.2: (From left to right): SIFT feature extraction applied on: Bus.raw; Sedan.raw

We can also overlay the descriptors of SIFT as shown in **Figure 3.3.3.**

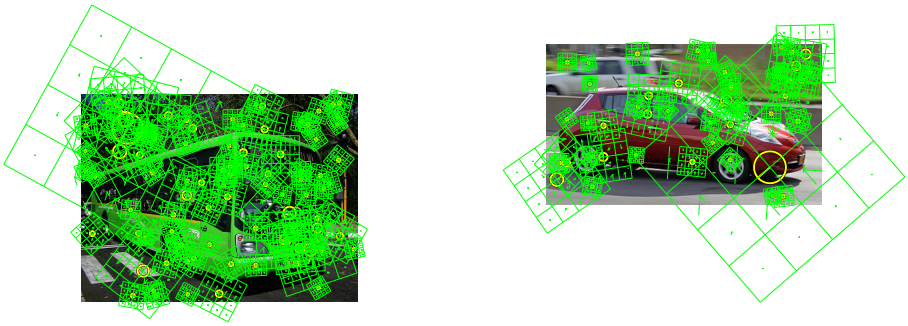


Figure 3.3.3: (From left to right): Overlaid descriptors of SIFT features on: Bus.raw; Sedan.raw

The results obtained by applying SURF feature extraction algorithm on Bus.raw and Sedan.raw are shown in **Figure 3.3.4** for a random collection of 100 strongest features.

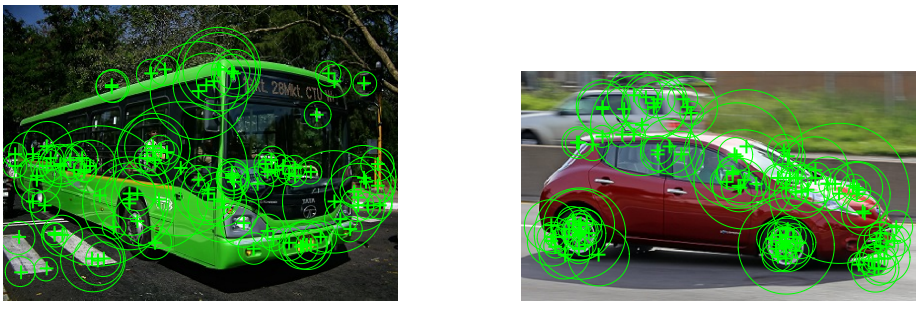


Figure 3.3.4: (From left to right): SURF feature extraction applied on: Bus.raw; Sedan.raw

The results obtained by applying the image matching algorithm with SIFT features of School_bus2.raw with School_bus1.raw, Bus.raw and Sedan.raw are shown below in **Figure 3.3.5.**

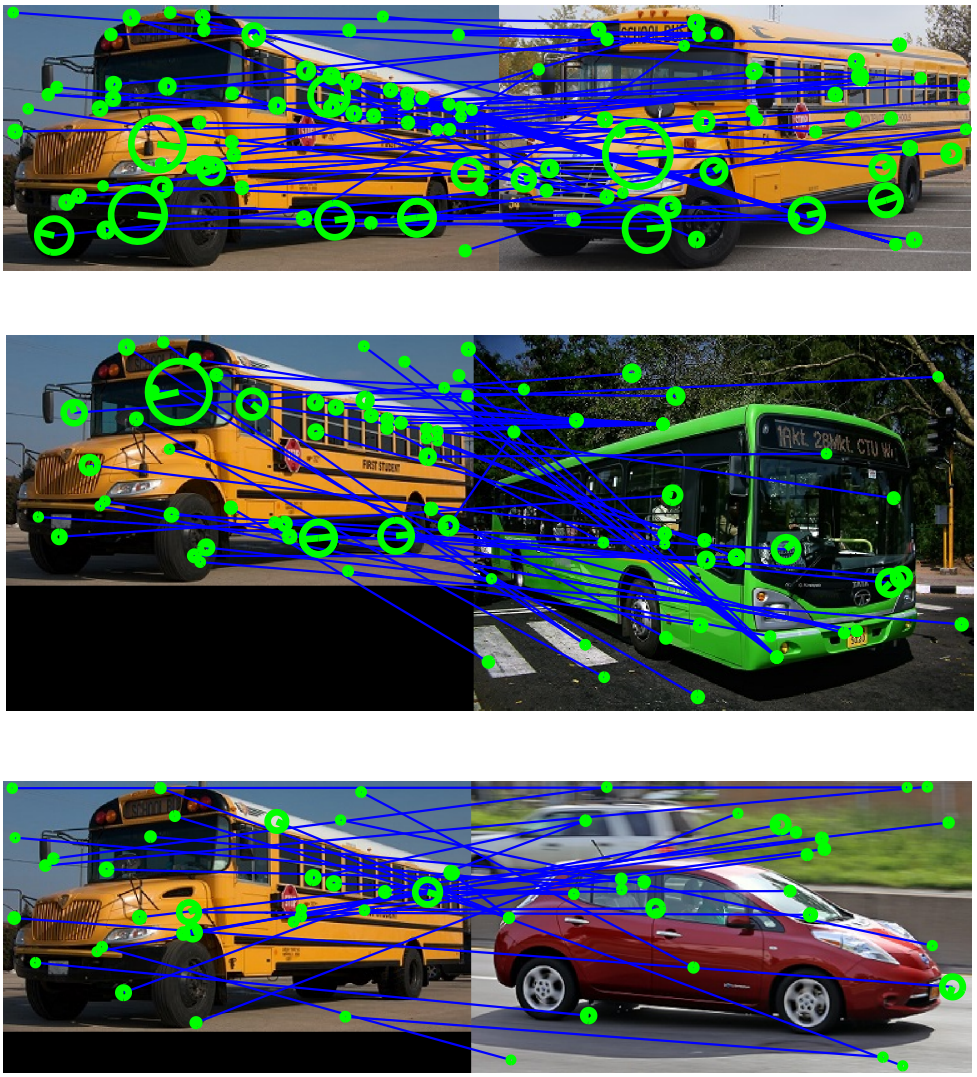
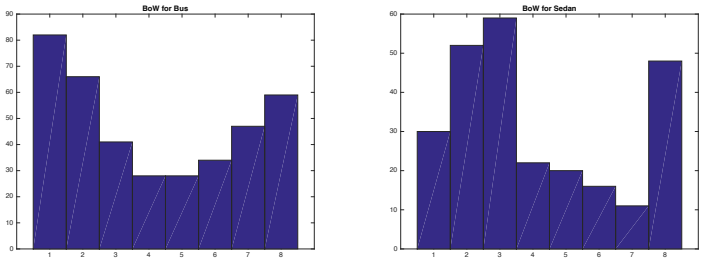


Figure 3.3.5: (From top to bottom): Image matching based on SIFT feature extraction between: i) School_bus2 & School_bus1; ii) School_bus2 & Bus; iii) School_bus2 & Sedan.

The histograms obtained by building codewords from SIFT feature extractor based on K-means clustering are shown in **Figure 3.3.6.**



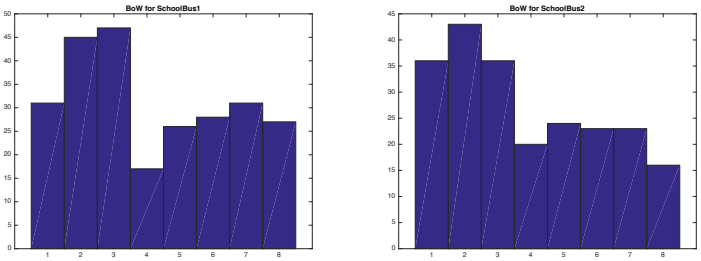


Figure 3.3.6: Histograms of Codewords built using K-means clustering using SIFT features

The histograms obtained by building codewords from SURF feature extractor based on K-means clustering are shown in **Figure 3.3.7.**

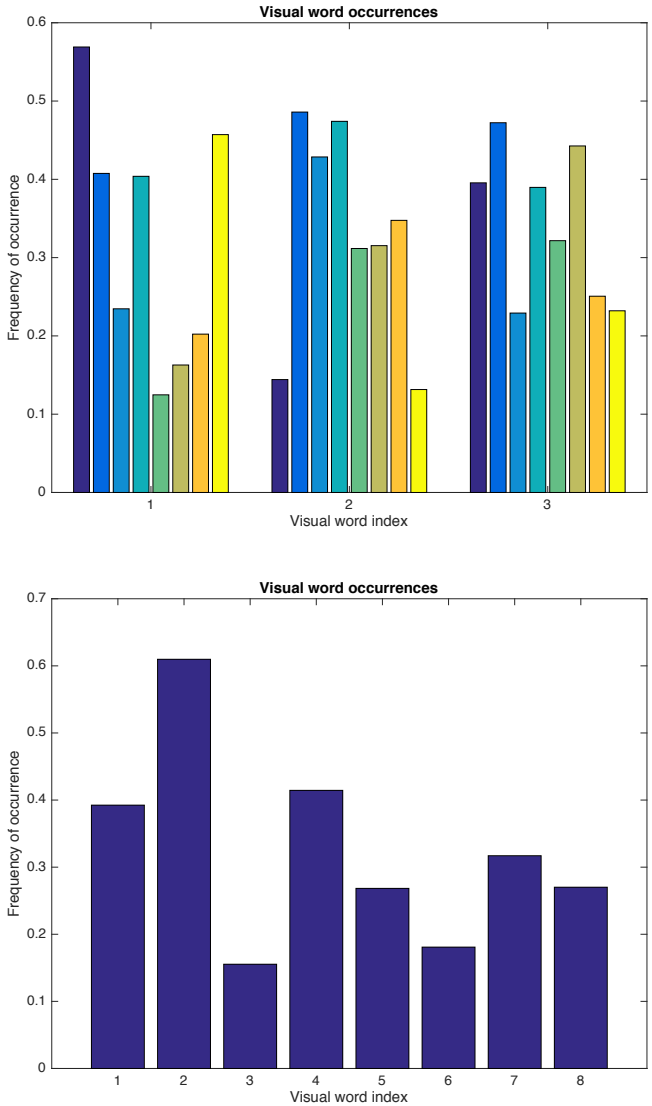


Figure 3.3.7: Histograms of Codewords built using K-means clustering using SURF features

As it is evident from the histogram itself School_bus2.raw has closer resemblance with School_bus1.raw. MATLAB's built-in function **knnsearch** provides the same result for both SIFT and SURF feature extractions with a distance of 19.3132 and .2597 respectively.

3.4 Discussion

An obvious advantage of SURF over SIFT is that it performs the feature extraction faster than SIFT. Hence, it will be more efficient in applications where we are dealing with a huge dataset and requires video processing. Although the feature extractions for these images are quite simpler for both extractors, the computation time required by SIFT was more than SURF. It took 2.22 seconds for SIFT feature extractor to get 100 features of images shown in **Figure 3.3.2** and 1.19 seconds for SURF feature extractor to get equivalent 100 features of images shown in **Figure 3.3.4.** Hence, it is evident that SURF is faster in execution than SIFT feature extractor.

Although SURF is faster than SIFT, the number of strongest features achieved with SIFT is more than SURF extractor. The features obtained from SIFT is more invariant to various features. Hence, it provides better features to compare when it comes down to image matching. An example is shown in **Figure 3.4.1** where 200 features are extracted for both SIFT and SURF cases. It is evident from the feature extraction that SIFT provides more features on different part of the car such as, windows and logo on the bottom of the door. It also provides more feature information of the environment which is not at all taken into consideration by SURF feature extractor. Although we need to comprise time for SIFT method, it is more preferred for image matching since it provides more image matching. As a result, image matching in part b with School_bus2 is done with SIFT extractor.

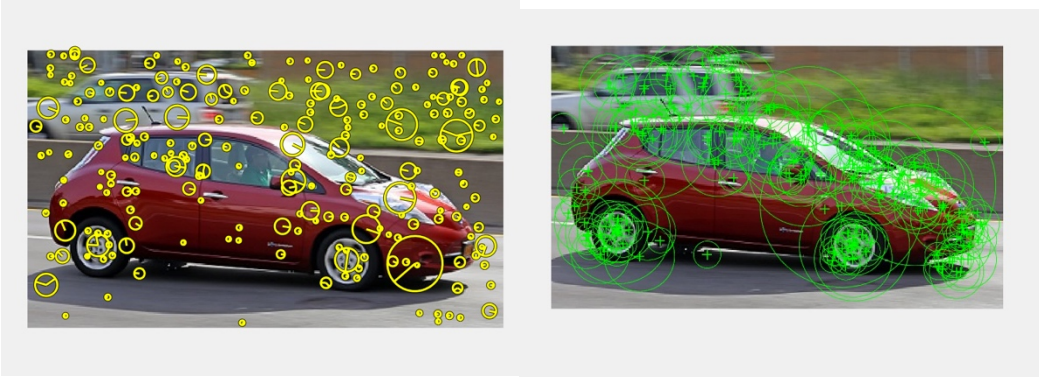


Figure 3.4.1: SIFT feature extraction for 200 salient points

Comparing the matching between School_bus1 and School_bus2, we observe that there are quite a lot of features that are found to be exactly matched by the algorithm correctly. Some of the salient features include detecting the SCHOOL BUS written on them of the buses; license plates detected accurately; STOP signs detected almost accurately. Although there are certain places where it fails, it has got most of the salient features of two buses into account. However, in case of Bus and Sedan, there are no salient features that are detected accurately. There are very few feature points that match. A reason could be the viewpoint of these images are not the same as School_bus2. Since SIFT uses invariant measures to find features that it believes are salient, we observe that the viewpoint was quite

variant in these images. Hence, it took other features such as edges, intensity as its point of detection in salient features. However, that resulted inaccuracy for both cases.

In **Figure 3.3.5,** it is observed that last two images are not of the same height as School_bus2.raw. So, the results are obtained are matched from different heights. We can resize the images into the height of our comparing image using bilinear interpolation implemented in one of our previous assignments. The results would still be the same. An example is provided in **Figure 3.4.2.**

Figure 3.4.2: Image matching between School_bus2 and Sedan by resizing the image using bilinear interpolation.

It is evident from the image matching itself that School_bus1 and School_bus2 has a resemblance to each other and implementation of Bag of Words algorithm on all the images proved to be right. The bag of words implemented with both SURF and SIFT extractor resulted in similar results. However, as discussed SIFT provides more salient features, which is helpful in providing a better classification for bag of words and k-means clustering. Although it did not make a difference for a problem of such smaller dataset, SIFT feature extraction would be preferable with more dataset. For the issue of time, there are various other state-of-the-art SIFT implementation which uses PCA as a mean of dimension reduction can be used to make the classification faster. It is to be mentioned that the viewpoint variance in the other two images (Bus.raw and Sedan.raw) also provided an edge in making the classification easier for both SIFT and SURF.

References

[1] Lifeng He, Yuyan Chao and K. Suzuki, 'A Run-Based Two-Scan Labeling Algorithm', IEEE Transactions on Image Processing, vol. 17, no. 5, pp. 749-756, 2008.

[2] M. K. Hu, "Visual Pattern Recognition by Moment Invariants," IRE Trans. Information Theory, IT-8, 2, February 1962, 179–187.

[3] Academia.edu, 'Morphological Image Processing', 2015. [Online]. Available: http://www.academia.edu/3482702/Morphological_Image_Processing

[4] S. Milyaev, O. Barinova, T. Novikova, P. Kohli and V. Lempitsky" Image binarization for end-to-end text understanding in natural images", Proc. 12th ICDAR, pp.128 -132

[5] Michael Kass, Andrew Witkin, Demetri Terzopoulos, "Snakes: Active contour models," International Journal of Computer Vision 1(4), 321-331, 1988.

[6] [Online]. Available: http://www.cse.unr.edu/~bebis/CS791E/Notes/DeformableContours.pdf.

[7] S. Contour, 'Snake : Active Contour - File Exchange - MATLAB Central', *Mathworks.com*, 2015. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/28149-snake---active-contour

[8] S. Models, 'Snakes: Active Contour Models - File Exchange - MATLAB Central', Mathworks.com, 2015. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/28109-snakes--active-contour-models

[9] Engineering.purdue.edu, 2015. [Online]. Available: https://engineering.purdue.edu/~malcolm/interval/1995-017/snake.m

[10] Tony F. Chan and Luminita A. Vese, "Active contours without edges," IEEE Trans. on Image Processing, Vol. 10, No. 2, February 2001.

[11] Imagecomputing.org, 'Code for image computing algorithms', 2015. [Online]. Available: http://www.imagecomputing.org/~cmli/code/

[12] David G. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, 60(2), 91-110, 2004.

[13] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, "SURF: Speeded Up Robust Features", Computer Vision and Image Understanding (CVIU), Vol. 110, No. 3, pp. 346-- 359, 2008

[14] [Online]. Available: http://web.eecs.umich.edu/~silvio/teaching/EECS598/lectures/lecture10_1.pdf

[15] Vlfeat.org, 'VLFeat - Tutorials > SIFT detector and descriptor', 2015. [Online]. Available: http://www.vlfeat.org/overview/sift.html#tut.sift.match

[16] cs.ubc.ca, 'Keypoint detector', 2015. [Online]. Available: http://www.cs.ubc.ca/~lowe/keypoints

[17] 2015. [Online]. Available: http://www.sci.utah.edu/~fletcher/CS7960/slides/Scott.pdf

[18] C. Image, 'File Exchange - MATLAB Central', Mathworks.com, 2015. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/43631-compute-bag-of-visual-word-representation-for-an-image/content/computeBoV/computeBoV.m

[19] Discussion Notes from EE 569, Fall 2015.

[20] W. Pratt, Digital image processing. New York: Wiley, 1978.

[21] http://sipi.usc.edu/database/