Homework 4

ECE 253
Digital Image Processing

December 1, 2017

Make sure you follow these instructions carefully during submission :

- Homework 4 is due by 11:59 PM, December 14, 2017.
- All problems are to be solved using MATLAB unless mentioned otherwise.
- You should avoid using loops in your MATLAB code unless you are explicitly permitted to do so.
- Submit your homework electronically by following the two steps listed below -

  1. Upload a pdf file with your write-up on Gradescope. This should include your answers to each question and relevant code snippet. Make sure the report mentions your full name and PID. Finally, carefully read and include the following sentences at the top of your report:

     Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.

     By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

  2. Send an email to ndeo@ucsd.edu with the subject line ECE 253 HW4. The email should have one file attached. Name this file: **ECE_253_hw4_lastname_studentid.zip**. This file should contain all of your MATLAB scripts and functions (as .m files) in a folder called **code** (Note that you have to include your code in the write-up in addition to this). This should include all files necessary to run your code out of the box. *Please use the exact format shown for the subject line and attachment since these are automatically archived and downloaded.*

Problem 1. Canny Edge Detection (10 points)

In this problem, you are required to write a function that performs *Canny Edge Detection*. The function has the following specifications:

- It takes in two inputs: a grayscale image, and a threshold t_e.
- It returns the edge image.
- You are allowed the use of loops.

A brief description of the algorithm is given below. Make sure your function reproduces the each step as given.

1. **Smoothing**: It is inevitable that all images taken from a camera will contain some amount of noise. To prevent noise from being mistaken for edges, noise must be reduced. Therefore the image is first smoothed by applying a Gaussian filter. A Gaussian kernel with standard deviation $\sigma = 1.4$ (shown below) is to be used.

$$k = \frac{1}{159} \cdot \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

2. **Finding Gradients** The next step is to find the horizontal and vertical gradients of the smoothed image using the *Sobel* operators. The gradient images in the x and y-direction, G_x and G_y are found by applying the kernels k_x and k_y given below:

$$k_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, k_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

   The corresponding gradient magnitude image is computed using:

$$|G| = \sqrt{G_x^2 + G_y^2},$$

   and the edge direction image is calculated as follows:

$$G_\theta = arctan(\frac{G_y}{G_x}).$$

3. **Non-maximum Suppression (NMS)**: The purpose of this step is to convert the thick edges in the gradient magnitude image to "sharp" edges. This is done by preserving all local maxima in the gradient image, and deleting everything else. This is carried out by recursively performing the following steps for each pixel in the gradient image:

   - Round the gradient direction θ to nearest $45°$, corresponding to the use of an 8-connected neighbourhood.

- Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient direction i.e. if the gradient direction is north ($\theta = 90°$), then compare with the pixels to the north and south.
- If the edge strength of the current pixel is largest; preserve the value of the edge strength. If not, suppress (remove) the value.

4. **Thresholding**: The edge-pixels remaining after the NMS step are (still) marked with their strength. Many of these will probably be true edges in the image, but some may be caused by noise or color variations. The simplest way to remove these would be to use a threshold, so that only edges stronger that a certain value would be preserved. Use the input t_e to perform thresholding on the non-maximum suppressed magnitude image.

Evaluate your canny edge detection function on *geisel.jpg* for a suitable value of t_e that retains the structural edges, and removes the noisy ones.

Things to turn in:

- The original gradient magnitude image, the image after NMS, and the final edge image after thresholding.
- The value for t_e that you used to produce the final edge image.
- MATLAB code for the function.

Problem 2. Hough Transform (10 points)

(The first two parts of this problem is borrowed from Professor Belongie's past CSE 166 class.)

(i) Implement the Hough Transform (HT) using the (ρ, θ) parameterization as described in GW Third Edition p. 733-738 (see 'HoughTransform.pdf' provided in the data folder). Use accumulator cells with a resolution of 1 degree in θ and 1 pixel in ρ.

(ii) Produce a simple 11×11 test image made up of zeros with 5 ones in it, arranged like the 5 points in GW Third Edition Figure 10.33(a). Compute and display its HT; the result should look like GW Third Edition Figure 10.33(b). Threshold the HT by looking for any (ρ, θ) cells that contains more than 2 votes then plot the corresponding lines in (x,y)-space on top of the original image.

(iii) Load in the image 'lane.png'. Compute and display its edges using the Sobel operator with default threshold settings, i.e.,

```
E = edge(I,'sobel')
```

Now compute and display the HT of the binary edge image E. As before, threshold the HT and plot the corresponding lines atop the original image; this time, use a threshold of 75% maximum accumulator count over the entire HT, i.e. `0.75*max(HT(:))`.

(iv) We would like to only show line detections in the driver's lane and ignore any other line detections such as the lines resulting from the neighboring lane closest to the bus, light pole, and sidewalks. Using the thresholded HT from the 'lanes.png' image in the previous part, show only the lines corresponding to the line detections from the driver's lane by thresholding the HT again using a specified range of θ this time. What are the approximate θ values for the two lines in the driver's lane?

Things to turn in:

- HT images should have colorbars next to them
- Line overlays should be clearly visible (adjust line width if needed)
- HT image axes should be properly labeled with name and values (see Figure 10.33(b) for example)
- 3 images from 2(ii): original image, HT, original image with lines
- 4 images from 2(iii): original image, binary edge image, HT, original image with lines
- 1 image from 2(iv): original image with lines
- θ values from 2(iv)
- Code for 2(i), 2(ii), 2(iii), 2(iv)

Problem 3. K-Means Segmentation (15 points)

In this problem, we shall implement a K-Means based segmentation algorithm from scratch. To do this, you are required to implement the following three functions -

- *features = createDataset(im)* : This function takes in an RGB image as input, and returns a dataset of features which are to be clustered. The output *features* is an $N \times M$ matrix where N is the number of pixels in the image *im*, and $M = 3$ (to store the RGB value of each pixel). You may not use a loop for this part.

- *[idx, centers] = kMeansCluster(features, centers)* : This function is intended to perform K-Means based clustering on the dataset *features* (of size $N \times M$). Each row in *features* represents a data point, and each column represents a feature. *centers* is a $k \times M$ matrix, where each row is the initial value of a cluster center. The output *idx* is an $N \times 1$ vector that stores the final cluster membership ($\in 1, 2, \cdots, k$) of each data point. The output *centers* are the final cluster centers after K-Means. Note that you may need to set a maximum iteration count to exit K-Means in case the algorithm fails to converge. You may use loops in this function.

  Functions you may find useful : *pdist2(), isequal()*.

- *im_seg = mapValues(im, idx)* : This function takes in the cluster membership vector *idx* ($N \times 1$), and returns the segmented image *im_seg* as the output. Each pixel in the segmented image must have the RGB value of the cluster center to which it belongs. You may use loops for this part.

  Functions that you may find useful : *mean(), cat()*.

With the above functions set up, perform image segmentation on the image *white-tower.png*, with the number of clusters, *nclusters = 7*. To maintain uniformity in the output image, please initialize clusters centers for K-Means as follows -

```
rng(5);
id = randi(size(features, 1), 1, nclusters);
centers = features(id, :);
```

Things to turn in:

- The input image, and the image after segmentation.
- The final cluster centers that you obtain after K-Means.
- All your MATLAB code for this problem (in the Appendix).