COMS 4731 – Computer Vision
Spring 2013
Homework #2
Due: Thu, February 21, 2013

All solutions (e.g., code, README write-ups, output images) should be zipped up as **HW2_yourUNI.zip** and posted on CourseWorks, where 'yourUNI' is your Columbia UNI. Only basic MATLAB functions are allowed, unless otherwise specified. If you are unsure of an allowable function, please ask before assuming! You will get no credit for using MATLAB to answer any questions where you should be answering them. When in doubt, ASK! This is a list of allowed "basic" functions from the Image Processing toolbox: `imread, imwrite, imshow, imagesc, imresize`. You may use the `line` function to draw lines in the image plots. You may **NOT** use: `edge, bwlabel, bwlabeln, im2bw, graythresh, im2double, bwboundaries, bwtraceboundary, hough, houghlines, houghpeaks, imgradient`. Any other functions not listed here as allowable or not allowable please ask.

Programming Assignment

1) Our goal is to develop a vision system that recognizes two-dimensional objects in images. The two objects we are interested in are shown in the image **two_objects.pgm**. (All of the images given to you are gray-level PGM images and can be downloaded from CourseWorks). Given an image such as **many_objects_1.pgm**, we would like our vision system to determine if the two objects are in the image and if so compute their positions and orientations. This information is valuable as it can be used by a robot to sort and assemble manufactured parts.

   The task is divided into four parts, each corresponding to a MATLAB function you need to write and submit. Each of the following parts a)-d) requires a separate MATLAB function, one for each (so 4 in total). Then write a "driver" program which loads the images that are needed, calls each of the functions, and reports/displays/writes the results. Therefore there should be 5 total files handed in with this question: **p1.m**, **p2.m**, **p3.m**, **p4.m**, and **test_objects.m**. Please stick to these naming conventions to make the grading easier.

   a. Write a MATLAB function named **p1** that converts a gray-level image to a binary one using a threshold value:

      ```
      function binary_out = p1(gray_in, thresh_val)
      ```

      Select any threshold that results in "clean" binary images for the gray-level ones given to you (A binary image can be saved as a PGM file). You should be able to use the same threshold value for all the images. (When submitting your assignment, please indicate the value you used in a separate README file.) Apply function **p1** to the image **two_objects.pgm**.

b. Implement the sequential labeling algorithm (and name the function **p2**) that segments a binary image into several connected regions:

```
function labels_out = p2(binary_in)
```

Note that you may have to make two passes of the image to resolve possible equivalences in the labels. In the "labeled" output image each object region should be painted with a different gray-level: the gray-level assigned to an object is its label. The labeled images can be displayed to check the results produced by your program. (To make gray-levels look significantly different, you may want to use consecutive natural numbers as labels in your output file, and view with `imagesc`). Note that your program must be able to produce correct results given any binary image. You can test it on the images given to you. Apply function **p2** to the binary version of the image **two_objects.pgm**.

c. Write a MATLAB function named **p3** that takes a labeled image and computes object attributes, and generates the objects database:

```
function [database_out, overlays_out] = p3(labels_in)
```

The generated object database should be an array of structs which includes a field for each of the objects with the following values:

i. `object_label`, the index label of the object
ii. `x_position` of the center, `y_position` of the center,
iii. `min_moment`, the minimum moment of inertia,
iv. `orientation`, the angle (in degrees) between the axis of minimum inertia and the vertical axis,
v. `roundness`, the roundness of the object.

You can compute any additional properties if you want. Just add them as additional fields, and be sure to mention these additional properties in your README file. These attributes will serve as your object model database. The output image should display positions and orientations of objects in the input image using a circle for the position and a short line segment originating from the center circle for orientation. Apply function **p3** to the labeled image of **two_objects.pgm**.

d. Now you have all the tools needed to develop the object recognition system. Write a function named **p4** that recognizes objects from the database:

```
function overlays_out = p4(labels_in, database_in)
```

Your program should compare (using your own comparison criteria) the attributes of each object in a labeled image file with those from the object model database. It should produce an output image which would display the positions and orientations (using circles and line segments, as before) of only those objects that have been recognized. Using the object database generated from **two_objects.pgm**, test your function on the images **many_objects 1.pgm** and **many_objects_2.pgm.** In your README file, state the comparison criteria and thresholds that you used.

2) Your task here is to develop a vision system that recognizes lines in an image using the Hough Transform. Such a system can be used to automatically interpret engineering drawings, etc. We will call it the "line finder". Three images are provided to you (available on CourseWorks): **hough_simple_1.pgm**, **hough_simple_2.pgm**, and **hough_complex_1.pgm**.

   a. First you need to find the locations of edge points in the image. For this you may either use the squared-gradient operator or the Laplacian. Since the Laplacian requires you to find zero-crossings in the image, you may choose to use the square gradient operator. The convolution masks proposed by Sobel should work reasonably well. Else, try your favorite masks. Generate an edge image where the intensity at each point is proportional to the edge magnitude. In the README, write down which mask(s) you used and why.

   ```
   function edge_image_out = p5(image_in)
   ```

   You may **NOT** use the `edge` function from the Image Processing Toolbox. However, you can compare your output to the output of this function for verification (in fact this is a good idea to test out your code!).

   b. Threshold the edge image so that you are left with only strong edges. We will not use edge orientation information as it is generally inaccurate in the case of discrete images. Next, you need to implement the Hough Transform for line detection. As discussed in class, the equation $y = mx + c$ is not suitable as it requires the use of a huge accumulator array. So, use the line equation $x\sin(\vartheta) - y\cos(\vartheta) + \rho = 0$. What are the ranges of possible θ values and possible ρ values (be careful here)? You can use these constraints to limit the size of the accumulator array. Also note them down in the README file.

   The resolution of the accumulator array must be selected carefully. Low resolution will not give you sufficient accuracy in the estimated parameters, but very high resolution will increase computations and reduce the number of votes in each bin. If you get bad results, you may want to vote for small patches rather than points in the accumulator array. Note that because the number of votes might exceed 255, you should create a new 2d array and store the votes in this array. Once you're done voting, copy over the values into an output image, scaling values so that they lie between 0 and 255. This image should be of the same resolution as your hough array, which will be different from the input image resolution, in general. In the README, write down what resolution you chose for your accumulator array (and why), what voting scheme you used (and why), and what edge threshold you chose.

```
function [edge_image_thresh_out, hough_image_out] =
p6(edge_image_in, edge_thresh)
```

c. To find "strong" lines in the image, scan through the accumulator array looking for parameter values that have high votes. Here again, a threshold must be used to distinguish strong lines from short segments (write down the value of this threshold in the README; it can be different for each image). After having detected the line parameters that have high confidence, paint the detected lines on a copy of the original scene image (using the `line` MATLAB function). Make sure that you draw the line using a color that is clearly visible in the output image.

```
function line_image_out = p7(image_in, hough_image_in,
hough_thresh)
```

d. Note that the above implementation does not detect end-points of line segments in the image. Implement an algorithm that prunes the detected lines so that they correspond to the line segments from the original image (i.e. not infinite). Briefly explain your algorithm in the README.

```
function cropped_lines_image_out = p8(image_in,
hough_image_in, edge_thresholded_in, hough_thresh)
```