CS83/183, Fall 2015, Problem Set # 1

October 2, 2015

Due: October 16, 11:59pm

This problem set requires you to provide written answers to a few technical questions. In addition, you will need to implement some learning algorithms in MATLAB and apply them to the enclosed data sets. The questions requiring a written answer are denoted with **W**, while the MATLAB programming questions are marked with **M**.

You must provide your written answers in paper form and leave them in the course mailbox located near the lobby of the Sudikoff building (the mailbox is tagged with the label CS83/183). Since each problem will be graded by a different person, we ask that you provide a separate submission for each individual problem. For example, for problem 2 of this homework assignment you should turn in a single set of pages stapled together containing your answers to question 2(b). **Please use the cover pages included at the end of this document. For each problem, fill out the corresponding cover page by writing your name, date and time of submission and use it as front page for your problem submission. A penalty of 5 points will be applied if you fail to follow these instructions.**

Instead, your MATLAB code must be submitted via Canvas. We have provided MATLAB function files with an empty body for all the functions that you need to implement. Each file specifies in detail the input and the output arguments of the function. A few of the functions include some commands to help you get started with the implementation. We have also provided scripts that check the sizes of the inputs and outputs of your functions. Do not modify these scripts. In order to allow us to test your code, it is imperative that you do not rename the files or modify the syntax of the functions. Make sure to include all the files necessary to run your code. **If we are unable to run one of your functions, you will receive 0 points for that question.** Please place all your files (data, .fig files automatically generated by the provided scripts, and function files) in a single folder (**without subfolders**). Zip up the entire folder, and upload the compressed file to Canvas.

Please do not use functions from Matlab toolboxes (e.g., the Image Processing Toolbox or the Optimization Toolbox) for your homework code unless instructed to do so. If in doubt, please ask. While student discussion is permitted, each student must write his/her homework solution and code *independently*.

1. **[30 points]** This problem is designed to help you familiarize yourself with the MATLAB environment and some of its useful built-in functions for image manipulation. You will implement a simple face detection algorithm using normalized cross correlation. You will use the image in file "template.jpg" as a template to find faces in the photo "soccer_team.jpg".

(a) [**M**, 8 points] Implement a function `q1_drawrectangles(I,x,y,width,height)` that produces a modified copy of the input color image `I` (a 3D matrix) by drawing red (unfilled) rectangles over it. The inputs `x, y, width and height` are K-dimensional vectors containing the coordinates and sizes of the K rectangles that must be drawn on the image. Specifically, the `j`-th rectangle should start from top-left corner at pixel location `(x(j),y(j))`, and should have width `width(j)` and height `height(j)` (in pixels). Note that you should leave the rectangles unfilled and draw only their edges (with line thickness equal to 1 pixel) using color [255 0 0] (red). Thus, the output should be another color image that is identical to `I` except at the pixels belonging to the *edges* of the rectangles.

You will receive 4 points for implementing the function correctly and 4 additional points if you code your function efficiently using a single `for` loop (i.e., you function should contain a loop over the K rectangles but should not loop over the individual pixels of each rectangle). To achieve this efficiency you may want to use the function `sub2ind`.

When you are done implementing your function you can run the script `q1a.m`. If your function works correctly you will see the image "soccer_team.jpg" with the faces of all players framed in red rectangles.

(b) [**M**, 12 points] You need to implement a function `q1_detectfaces(I,template,threshold)`, where `I`, and `template` are color images (i.e., 3D matrices) and `threshold` is a scalar value. This function must return two matrices as output. The first is a color image containing red rectangles of the template size at all pixel locations where the cross correlation between grayscale versions of `I` and `template` is greater than `threshold`. The second output is the actual correlation map (i.e., the result of normalized cross correlation). In detail, here is what you need to do:

- Start by implementing a function `q1_normalized_correlation(I, template)` performing normalized cross correlation with the built-in function `normxcorr2` using a grayscale image `I` (a 2D matrix) and a grayscale template `template`. Note that this command will produce as output a matrix of size $(m_I + m_T - 1) \times (n_I + n_T - 1)$ where $(m_I \times n_I)$ is the size of the image and $(m_T \times n_T)$ is the size of the template. Eliminate the first $(m_T - 1)$ rows and the first $(n_T - 1)$ columns of this matrix as they contain irrelevant information for our purpose. You should do this by using the MATLAB colon notation (type `help colon` for help) and *without loops*. You are now left with an $(m_I \times n_I)$ matrix containing the result of normalized cross correlation.
- Now implement `q1_detectfaces`. You function must perform the following steps:
  i. Convert the input photo and the template to grayscale images by calling the function `rgb2gray`.
  ii. Invoke your function `q1_normalized_correlation` by passing as arguments the grayscale versions of `I` and `template`.
  iii. Use the argument `threshold` (a value between -1 and 1), and the function `q1_drawrectangles` (from part (a)) to draw rectangles at all locations x, y with normalized cross correlation value larger than your threshold (draw rectangles having size equal to the template size and top-left corner at detected locations). You can use the function `find` to identify all locations satisfying this condition.
- After you have complete the tasks outlined above, run the script `q1b.m` to check the output of your function. If your function works correctly you will see the image

"soccer_team.jpg" with faces framed in red rectangle. This script uses a threshold of 0.33 which provides a good tradeoff in terms of detecting a large number of faces while keeping the number of false positives (i.e., locations incorrectly detected as faces) to a minimum.

(c) [**M**, 3 points] Implement a function `q1_detectfacesHSV(I,template,threshold)`, having input and output argument as described in the point above. However, this time your function must convert the color image and template to the HSV format by using `rgb2hsv`. Then it should run cross correlation with the *hue channels* of the image and the template (you should reuse your `q1_normalized_correlation` function for this purpose).

After completing this function, run the script `q1c.m` to check how your function works.

(d) [**W**, 3 points] Think of a strategy that uses the two normalized cross correlation output matrices of (b) and (c) together to produce an improved detection result. Explain your strategy.

(e) [**M**, 4 points] Write a function `q1_detectfacesColorAndGray(I,template,threshold)` that implements this strategy. This function should invoke `q1_detectfaces` and `q1_detectfacesHSV` and then jointly use their normalized cross correlation output matrices to perform the final (improved) face detection. Note that now `threshold` is defined to be a 5-dimensional vector to give you enough space to store any parameters that you want to pass to your function.

After completing this part run the script `q1e.m` to check how your function works. Remember to set the vector `threshold` in this script to the parameter values you need but please keep its dimensionality equal to 5 in order to pass our input-output size tests (just set the parameter values you need in the top positions of this vector and leave the other entries set to zero).

2. **[30 points]** In this problem you will implement a function that computes radial distortion parameters given a set of grid points (x, y) (assuming no distortion) and their actual locations (x_d, y_d) with distortion.

The enclosed Matlab function `squaregrid` takes as input two arguments m, n and returns as output two matrices x, y containing the coordinates of $2mn$ points on a square grid with m horizontal and m vertical lines, with n points on each line. The output arrays `x` and `y` have n rows and $2m$ columns, and are arranged so that the `plot` command does the right thing. For instance, to see a 7×7 grid with 30 points on each line, type

```
[x,y] = squaregrid(7, 30);
plot(x, y, 'b')
axis equal
axis off
```

(a) [**M**, 10 points] Write a Matlab function with header

```
[xd, yd] = q2_applyradialdistortion(x, y, k1, k2)
```

that takes as input a grid as computed by `squaregrid` and applies the lens radial distortion model that we have discussed in class. The first two arguments are the grid coordinates while the third and the fourth arguments are the radial distortion parameters. The output arguments `xd,yd` should have the same size as the inputs `x,y`.

Remember that according to our distortion model a point (x, y) moves because of lens distortion to point (x_d, y_d) where

$$x_d = x(1 + k_1 r^2 + k_2 r^4) \quad (1)$$
$$y_d = y(1 + k_1 r^2 + k_2 r^4) \quad (2)$$

and $r^2 = x^2 + y^2$.

After you are done with your implementation, run the script `q2a.m` to see the output of your radial distortion function with parameters $k1 = 0.15$, $k2 = -0.04$ applied to the 7×7 point grid described above. You get 5 points for writing the function correctly, and the remaining 5 points for doing it without loops.

(b) [**W**, 2 points] By bringing the terms with the distortion parameters to the left-hand side and all the remaining terms to the right-hand side, the lens distortion equations can be rewritten in the form

$$a_{11}k_1 + a_{12}k_2 = b_1 \quad (3)$$
$$a_{21}k_1 + a_{22}k_2 = b_2 \quad (4)$$

Give the suitable definitions of coefficients $a_{11}, a_{12}, a_{21}, a_{22}, b_1, b_2$ in terms of x, y, x_d, y_d, r so as to satisfy this equation.

(c) [**M**, 18 points] With N points one can write $2N$ equations of this form, which can be packaged into an over-constrained linear system of the form

$$A\mathbf{k} = \mathbf{b} \quad (5)$$

where A is a $2N \times 2$ matrix, $\mathbf{k}$ is a 2-dimensional vector containing k_1 and k_2, and $\mathbf{b}$ is a $2N$-dimensional vector. In Matlab, such a system can be solved with the single line

```
k = A \ b;
```

so as to obtain the vector `k` containing the least-squares estimates of k_1 and k_2. Write a function with header `[k1, k2] = q2_computedistortionparams(x, y, xd, yd)` that takes arrays of coordinates as computed by `squaregrid` and `q2_applyradialdistortion` and estimates the distortion parameters k_1, k_2. Check that running your function on the data points (x, y) and (x_d, y_d) produced in (b) returns the correct distortion parameters $k1 = 0.15$, $k2 = -0.04$.

Finally, if you completed everything correctly you should be able to run the enclosed script `q2c.m` which runs your function for parameter estimation on noise-corrupted versions of x_d, y_d simulating the measurement noise present in real data. This script plots the deviation of the parameter estimates from the true parameters as a function of the amount of Gaussian noise added to the coordinates.

This time you get 12 points for doing this right, and the remaining 6 points for not using loops in your implementation of `q2_computedistortionparams`. [Hint: to transform a matrix `x` into a vector in Matlab, just type `x = x(:);`]

3. **[M, 20 points + 20 bonus points]** Traditional Gaussian filtering of a color image I can be expressed as follows:

$$H(i,j) = \frac{\sum_{(k,l)\in\mathcal{N}(i,j)} I(k,l)w(i,j;k,l)}{\sum_{(k,l)\in\mathcal{N}(i,j)} w(i,j;k,l)} \tag{6}$$

where $\mathcal{N}(i,j)$ denotes a neighborhood of pixel (i,j) and $w(i,j;k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}\right)$. The weight $w(i,j;k,l)$ is based on the spatial distance from the neighborhood center (i,j) to pixel (k,l). Instead, for a *bilateral* filter, the weight is determined based on two distances: an image-space distance and a color-space distance. The intuitive idea is that if a pixel is far from the neighborhood center either in image space or in color space then it will have little influence on the result. Formally, the weight $w(i,j;k,l)$ for the bilateral filter is defined as follows:

$$w(i,j;k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2} - \frac{||I(i,j)-I(k,l)||^2}{2\sigma_r^2}\right) \tag{7}$$

where $||I(i,j)-I(k,l)||^2 = (I_r(i,j)-I_r(k,l))^2 + (I_g(i,j)-I_g(k,l))^2 + (I_b(i,j)-I_b(k,l))^2$, and I_r, I_g, I_b denote the r,g,b channels of image I. Figure 1 shows the result of applying bilateral filtering to a color image: note how the filter successfully smoothes the colors while preserving the edges.

Figure 1: Left: input image. Right: result of bilateral filtering.

(a) [**M**, 20 points] Bilateral filtering is a bit tricky to code correctly. Thus, we ask you to first implement it in the most naive way, by using a double `for` loop over the pixels (one loop over rows, one loop over columns) containing two additionally inner `for` loops over the coordinates of the individual pixels in each neighborhood. This will result in a function that is very slow but that should be easy to code. You should write this slow-but-safe implementation in the function stub file `q3_bilateralfiltering_vanilla(I,`

`h_d, sigma_d, sigma_r)` which we have provided to you. After your are done coding it, run the script `q3a.m` to make sure that it does the right thing. You should see an output identical to that shown in Figure 1. You will get 20 points if your implementation with the 4 `for` loops is correct. In order to facilitate our job when checking the correctness of your implementation, please use zero padding for the image borders. Your function should perform the computation in double precision and output a 3D matrix of doubles.

(b) [**M**, 20 bonus points] We now ask you to write the function `q3_bilateralfiltering_optimized(I, h_d, sigma_d, sigma_r)` which should represent a much faster implementation of bilateral filtering. This time you will receive bonus points based on how fast your function runs (but, obviously, you will get 0 bonus points if your function does not compute the correct output, regardless of how fast it is). You can use any trick that you can think of in order to make your code fast. We suggest that you compare the output of your optimized code with that produced in (a) to make sure that your efficient implementation computes things correctly (again, it is easy to get bilateral filtering wrong). Your function `q3_bilateralfiltering_optimized` can be tested by running the script `q3b.m`.

Here is how the bonus points will be assigned. We will compute the average running time of your function over 10 runs for a given test image (all homework submissions will be evaluated on the same machine). You can use the provided scripts `q3a.m` and `q3b.m` to compute the average runtime of the vanilla and optimized versions of your algorithm respectively. We will compare the average runtime of your optimized function to that obtained with our own (efficient) implementation. Let t denote the average runtime of your function and $\hat{t}$ the average runtime of our implementation. Then you will receive 20 bonus points if $t < 1.25\hat{t}$, 10 points if $t \in [1.25\hat{t}, 2\hat{t}[$, 5 points if $t \in [2\hat{t}, 3\hat{t}]$, 0 points otherwise. *Hint*: The key to a fast software implementation is to reduce the number of loop iterations to a minimum. But many different strategies may lead to fast software.

4. **[W, 10 points]** If we use a pinhole camera with focal length 5 mm to view a scene point at $[X, Y, Z] = [9m, -6m, 15m]$, at what coordinates does that point appear in a 640-by-480 image? Assume that pixels have width and a height of 0.1 mm and that the principal point is at $[x, y] = [320, 240]$. Explain your solution.

5. *Problem for graduate credit only: if you are enrolled in COSC83 (instead of COSC183) you do not need to solve this problem.*
[W, 10 points] Suppose that a circle in the scene is known to be centered along the optical axis and to lie in a plane that is parallel to the image projection plane. *Concisely but formally* prove that the perspective projection of this circle forms another circle in the image.

HW1 | CS83/183, Fall 2015

Problem 1

First name _____ **Last name** _____

Date and time of submission _____

HW1 | CS83/183, Fall 2015

Problem 2

First name _____ **Last name** _____

Date and time of submission _____

HW1 | CS83/183, Fall 2015

Problem 4

First name _____ **Last name** _____

Date and time of submission _____

HW1 | CS83/183, Fall 2015

Problem 5

First name _____ **Last name** _____

Date and time of submission _____