Homework 3

CMPSCI 370 Spring 2019, UMass Amherst

Due: March 20, 11:55 PM

Instructor: Subhransu Maji

TA: Tsung-Yu Lin

Guidelines

Submission. Submit your answers via Gradescope that includes a pdf with your solutions and source code. You have a total of 3 late days for all your homework assignments which you can use in any way you like. However note that delay by even one minute counts as a full late day and submissions beyond the late days will not be given *any* credit. Please contact the instructor and obtain permission ahead of time if you seek exceptions to the policy.

Plagiarism. We might reuse problem set questions from previous years, covered by papers and webpages, we expect the students not to copy, refer to, or look at the solutions in preparing their answers. We expect students to want to learn and not google for answers. Any instance of cheating will lead to zero credit for the homework, and possibly a failure grade for the entire course.

Collaboration. The homework must be done individually, except where otherwise noted in the assignments. 'Individually' means each student must hand in their own answers, and each student must write their own code in the programming part of the assignment. It is acceptable, however, for students to collaborate in figuring out answers and helping each other solve the problems. We will be assuming that you will be taking the responsibility to make sure you personally understand the solution to any work arising from such a collaboration.

Matlab requirements. The code should work any Matlab version ($\geq$ 2011a) and relies on the Image Processing Toolbox for some functions.

Python requirements. We will be using Python 2.7. The Python starter code requires `scipy`, `matplotlib` and `pillow` for loading images and `numpy` for matrix operations (at least v1.12). If you are not familiar with installing those libraries through some package manager (like `pip`), the easiest way of using them is installing Anaconda. Contact the course staff if you are having trouble installing these.

Using other programming languages. We made the starter code in Python and Matlab. You are free to use other languages such as Octave or Julia with the caveat that these will not be supported by the course staff.

1 Image Filtering [10 points]

Answer these questions in one or two sentences. Each one is worth 2 points.

- Why is filtering with a Gaussian kernel preferable over a box kernel for denoising an image?

- What is the effect of increasing the σ of the Gaussian kernel on the result of filtering?

- When is median filtering preferable over Gaussian kernel filtering for denoising?

- Why is it a good idea to smooth an image before filtering with a derivative filter?

- What does filtering an image with a Laplacian of Gaussian filter do?

2 Image Gradient and Orientation Histogram [15 points]

Write a function to compute the gradient magnitude and angle (orientation) for each pixel in an image. The function `[m, a] = imageGradient(im)` should take a grayscale image `im` and return two arrays, m and a, that indicate the magnitude and angle of the gradient at each pixel in the image. The arrays m and a should be of the same size of the image.

Note that the gradient of the image gx and gy can be obtained by filtering the image with derivative filters. In this homework compute the gradients using the following filters $fx = [-1\ 0\ 1]$ and $fy = [-1\ 0\ 1]^T$. The gradient magnitude is given by $m = \sqrt{gx^2 + gy^2}$ and the angle $\theta = \tan^{-1}(gy/gx)$. In this case the angle $\theta \in [-\pi/2, \pi/2]$ radians or $[-90, 90]$ degrees.

It is important to treat the boundary of the images carefully. The simplest way is to pad the image using the option "replicate" where you repeat the sides for padding. Padding with zeros will most certainly add a strong gradient along the four sides. Alternatively, you can zero out the gradient responses for pixels that are near the boundary after computing gradients with zero padding.

- *(5 points)* Use your implementation to compute the gradient magnitude and angle on the `parrot.jpg` image included in the data directory (Figure 1). Convert this image to grayscale and double format before applying the image filters. Visualize the magnitude as a grayscale image and angle image using the "parula" colormap. The is the default colormap for "imagesc()" in Matlab. See an example in Figure 2.
- *(5 points)* Recompute the the above quantities by first applying a Gaussian filter of $\sigma = 2$ pixels to the image. Display that gradient magnitude and orientation for the smoothed image. How is the gradient magnitude affected by smoothing?
- *(5 points)* Visualize the distribution of angles by computing a histogram of orientations as follows:
  1. Bin the range of angles between [-90 90] degrees into 9 equal sized bins. For example the first bin corresponds to angles [-90 -70) degrees [1], the second bin corresponds to angles [-70 -50), till the last bin that corresponds to angles [70 90].
  2. Assign each pixel to a bin based on the angle. For example the angle -85.2 degrees gets assigned to bin 1, while angle 0 degrees gets assigned to bin 5. This step gives you a number between 1 to 9 for each pixel.
  3. For each bin compute the total magnitude by summing the magnitudes of all the pixels that belong to the bin. This gives you 9 numbers that indicates the total magnitude in 9 different orientation. Weighing by the magnitude reduces the contributions of the flat regions in the image where the magnitude is close to zero and the angle is unreliable. Plot the orientation histogram for this image with and without smoothing using a "bar" plot. See Figure 2.

Figure 2 shows the gradients for the butterfly image included in the data directory.

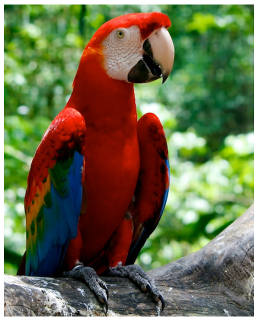

Figure 1: Compute the gradient magnitude, angle, and gradient histogram for the "parrot" image.

[1] The notation means that -90 is included and -70 is excluded

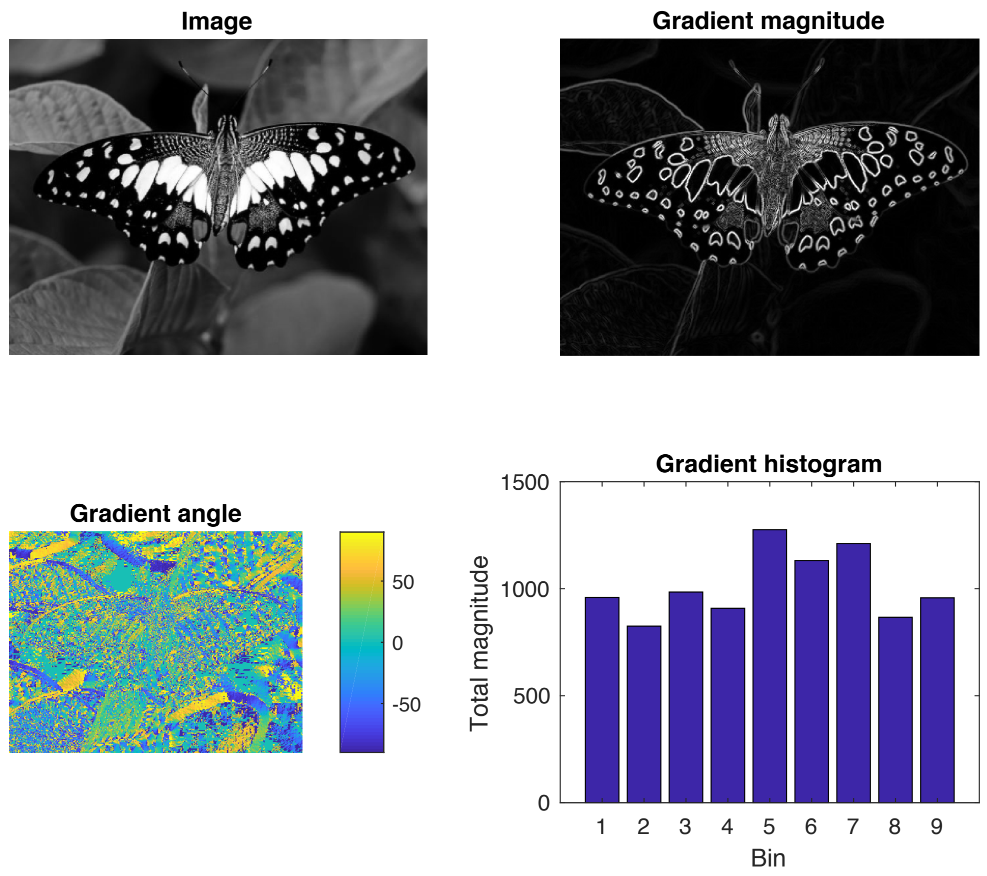

Figure 2: Gradient magnitude, angle and gradient histogram for the butterfly image.

3 Corner Detection

In this part you will implement two corner detection techniques as discussed in class. The first is a "simple corner detector" that is based on how much a patch changes when you move in eight different directions. The second is the Harris corner detector which is a more accurate way of measuring the same change. Recall that the steps for detecting corners are:

1. Compute a per-pixel "cornerness" score.
2. Threshold the score and compute peaks of the score (or non-maximum suppression).

The codebase contains an implementation of Step 2. Your will implement Step 1. The entry code for this part is in `testCornerDetector`. The code loads a checkerboard image and calls the `detectCorners` function with `isSimple=true` and `isSimple=false`, and diplays the detected corners. If you run this code you should see the output in Figure 3-top. Once you implement the corner detectors described in the next two sections the corners should be correctly detected as shown in Figure 3-bottom.

Take a look inside the file `detectCorners` which now contains a dummy implementation of the two corner detectors. Both these currently return a `cornerScore` image indicating how corner-like each pixel is. This is thresholded and non-maximum suppressed to produce the location of peaks. Sorting these locations by score produces the ranked list of corner locations. If you are curious take a look at the `nms` function. You will replace the detectCorners with your own implementation.

3.a A simple corner detector [25 points]

Recall, that the simple corner detector computes the `cornerScore` by computing the weighted sum-of-squared differences between pixels within a window and its shifted version in eight possible directions.

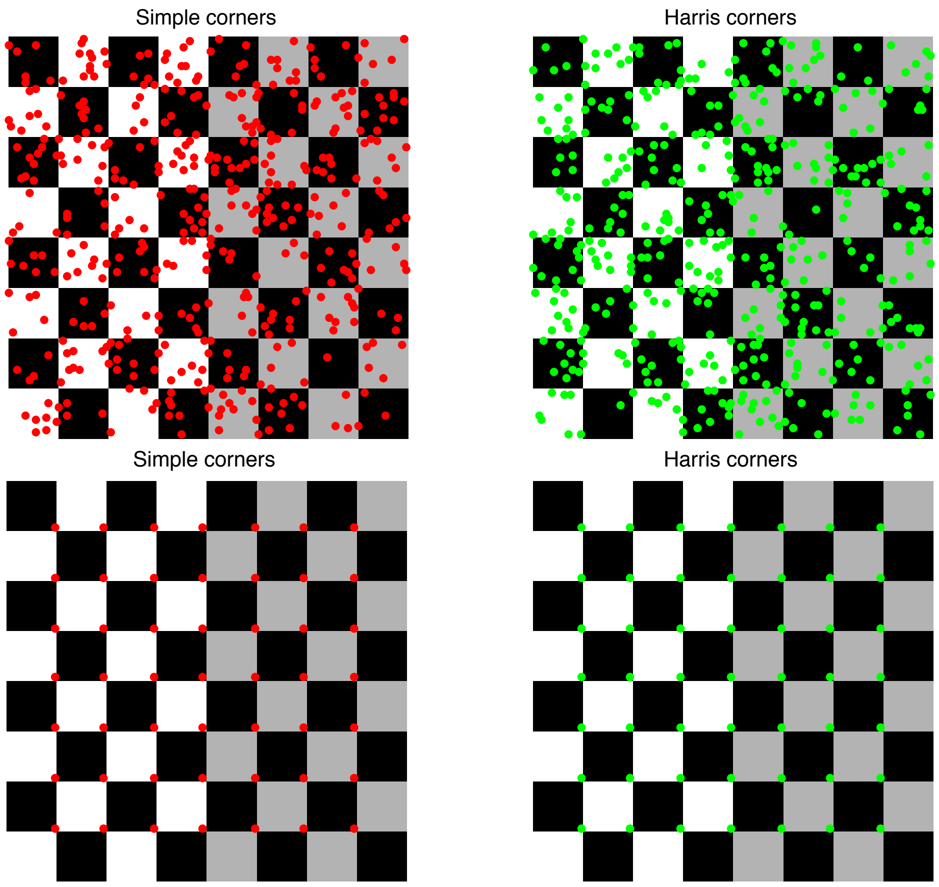

Figure 3: **(Top)** Initial output of `testCornerDetector`. **(Bottom)** Output after the correct implementation of `detectCorners`.

The parameter `w` specifies the σ of the Gaussian used to compute the weighting kernel.

Implement the function `[cx, cy, cs] = detectCorners(I, isSimple=true, w, th)`. Here, cx, cy, and cs is a list of x-coordinate, y-coordinate, and corner score of the corners sorted in the decreasing order of their scores, w is the window size, and th is a user-specified threshold.

You can do this by implementing the helper function `cornerScore = simpleScore(I, w)` inside the file, which returns a image with per-pixel scores. Right now it simply returns a random score for each pixel. The basic algorithm is to compute

$$E(u,v) = (I * f(u,v))\,.^2 * G_\sigma,$$

for each of the eight possible shifts (u,v). Here $f(u,v)$ is the filter corresponding to the difference between pixel at the center and pixel at (u,v), $.^2$ is the *element-wise squaring* operation[2], and G_σ is a Gaussian kernel with standard deviation σ. The `cornerScore` is the sum of $E(u,v)$ across all of (u,v), i.e.,

$$\text{cornerScore} = \sum_{u,v} E(u,v)$$

[2]Note the difference between element-wise squaring and matrix squaring in Matlab

3.b Harris corner detector [25 points]

The Harris corner detector computes the `cornerScore` by looking at behavior of the function $E(u, v)$ for all possible values of the shift (u, v). The 2×2 matrix,

$$M = \sum_{x,y} G_\sigma(x, y) \left[\begin{array}{cc} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{array} \right],$$

characterizes how $E(u, v)$ behaves for small shifts (u, v), from which the `cornerScore` is computed as,

$$\text{cornerScore} = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2.$$

Here, λ_1 and λ_2 are the eigenvalues of the matrix M, and k is a parameter usually set between $0.04 - 0.06$ (you can set $k = 0.04$ in your implementation). I_x and I_y are gradients in x and y directions respectively which can be computed using a derivative filter.

Instead of computing the eigenvalues of the matrix you can use the following identities to compute the score. If M is a matrix,

$$M = \left[\begin{array}{cc} a & b \\ c & d \end{array} \right],$$

then, $\lambda_1 \lambda_2 = det(M) = ad - bc$, and $\lambda_1 + \lambda_2 = trace(M) = a + d$. Thus you can compute the score as,

$$\text{cornerScore} = (ad - bc) - k(a + d)^2.$$

Using these identities implement `cornerScore = harrisScore(I,w)` inside the `detectCorners()` function. The corners detected by this function will be similar to the output of the first method which may help you debug.

Tip: Try visualizating intermediate outputs of the code to debug by adding breakpoints in your code. Another useful operation in matlab is that A.*B does element-wise multiplication of A and B, which might save you a few "for" loops.

3.c What to submit?

To obtain full credit include:

- A self contained implementation of `detectCorners` function.
- The corner score visualized as a colormap and the output of running `testCornerDetector` on the checkerboard image for both the simple and Harris corner detector.
- The same outputs (score, corner detections) on the `polymer-science-umass.jpg` image provided in the data directory and one additional image of your choice.

Note that you may have to adjust the thresholds on the corner score for each image to display a sensible number of corners. Alternatively you can detect the top 100 corners for each image based on the corner score. Feel free to modify the testCornerDetector function to account for this and include the modified version in your submission.

4 Submission and rubric

- Follow the outline on Gradescope for your submission.
- A fraction of the points for each coding assignment will be for readability and efficiency of your code. Thus it is important to properly document parts of the code you implemented.