COMPSCI 370: Intro to Computer Vision
University of Massachusetts, Amherst
Spring 2023
Instructor: Subhransu Maji
TA: Aaron Sun

Homework 3

1 Image denoising [15 points]

In this part you will evaluate Gaussian filtering and median filtering for image denoising. You are provided with

- `peppers.png` – reference image without noise
- `peppers_g.png` – image with Gaussian noise.
- `peppers_sp.png` – image with "Salt and Pepper" noise.

The `evalDenoising.py` script loads three images, visualizes them, and computes the error (squared-distance) between them as shown below. Your goal is to denoise the images which should result in a lower error value.

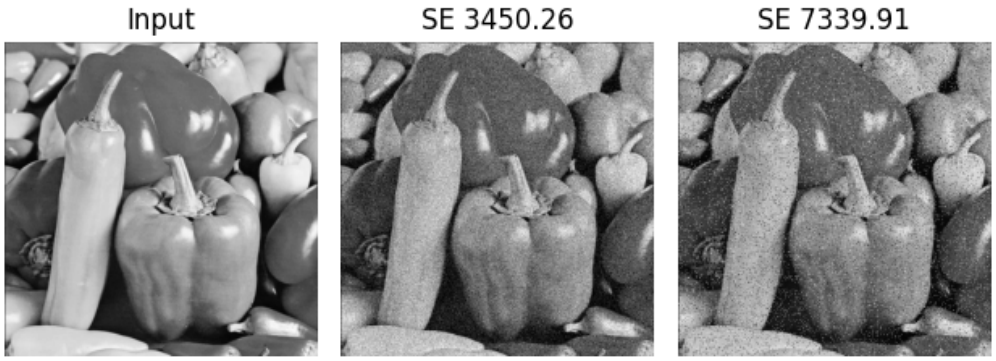

Figure 1: Input images for denoising and their errors (squared-distance).

1. *[5 points]* Using `scipy.ndimage.gaussian_filter`, apply Gaussian filter on the above images. Experiment with different values of the standard deviation parameter σ. Report the optimal error and σ for each of these images. For reference, our solution gets an error of < 500 for Gaussian noise and < 800 for salt and pepper noise. Try to get σ within 0.1 of our solution.
2. *[5 points]* Using `scipy.ndimage.median_filter` apply a median filter on the above images. Report the optimal error and neighborhood size for each of the images. For reference, our solution gets < 600 for Gaussian noise and < 200 for salt and pepper noise. Try to get the same window as our solution.
3. *[5 points]* Qualitatively compare the outputs of these results. You should include the outputs of the algorithms side-by-side for an easy comparison. Which algorithm works the best for which image?

2 Hybrid images [20 points]

A hybrid image is a sum of blurry image and a sharpened image that looks like one or the other depending on the perceived resolution of the image. In this part you will try to create such images. Recall that one can obtain a sharp image by subtracting the blurry version of an image from itself. Mathematically we have

$$I = \texttt{blurry}(I) + \texttt{sharp}(I).$$

A blurry image can be obtained by filtering an image with a Gaussian. Thus, a hybrid image of I_1 and I_2 can be obtained by

$$I_{hybrid} = \texttt{blurry}(I_1, \sigma_1) + \texttt{sharp}(I_2, \sigma_2) = I_1 * g(\sigma_1) + I_2 - I_2 * g(\sigma_2) \quad (1)$$

where, $g(\sigma_1)$ and $g(\sigma_2)$ are Gaussian filters with parameters σ_1 and σ_2 and * denotes the filtering operator. Figure 2 shows the result of convolution with Gaussian filters with two different values of σ.

dog image $\sigma = 4$ cat image $\sigma = 10$

Figure 2: Effect of filtering with a Gaussian. The bigger the sigma the more blurry it is.

[20 points] Implement the function `hybridImage(im1, im2, sigma1, sigma2)` that computes the hybrid image using Equation 1. Use your code to generate at least two examples of hybrid images: one should be of the dog and cat, and the other should be combining two images of your choosing. You will have to tune the `sigma1` and `sigma2` to make it work on specific images.

In order to visualize the hybrid image it is useful to show multiple copies of the image at different resolutions. The codebase includes a helper function `vis_hybrid_image` which can be used to create such a figure. In Python, `matplotlib` does not handle negative values in images, so use `np.clip` to make sure that image values are between 0 and 1 (assuming floating point representation). Include the code, the four source images, the hybrid images displayed in the format below, as well as the parameters that worked for you. The top three submissions (judged based on creativity and how compelling it is) will be announced in class.

Figure 3: Hybid image of the dog and cat. The large image looks like the cat while the small image looks like the dog. The image was created with $\sigma_1 = 4$ and $\sigma_2 = 10$.

3 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 `imageGradient` 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, you should 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. You are free to use 'scipy.ndimage.filters.convolve' for this task.

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 4). 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 "viridis" colormap. This is the default colormap for matplotlib. See an example in Figure 5.

- *(5 points)* Recompute the the above quantities (gradient magnitude and angle images) 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 on both the original parrot and the smoothed parrot 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 5.

Figure 5 shows the gradients for the butterfly image included in the data directory.

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

Figure 4: Compute the gradient magnitude, angle, and gradient histogram for the "parrot" image.

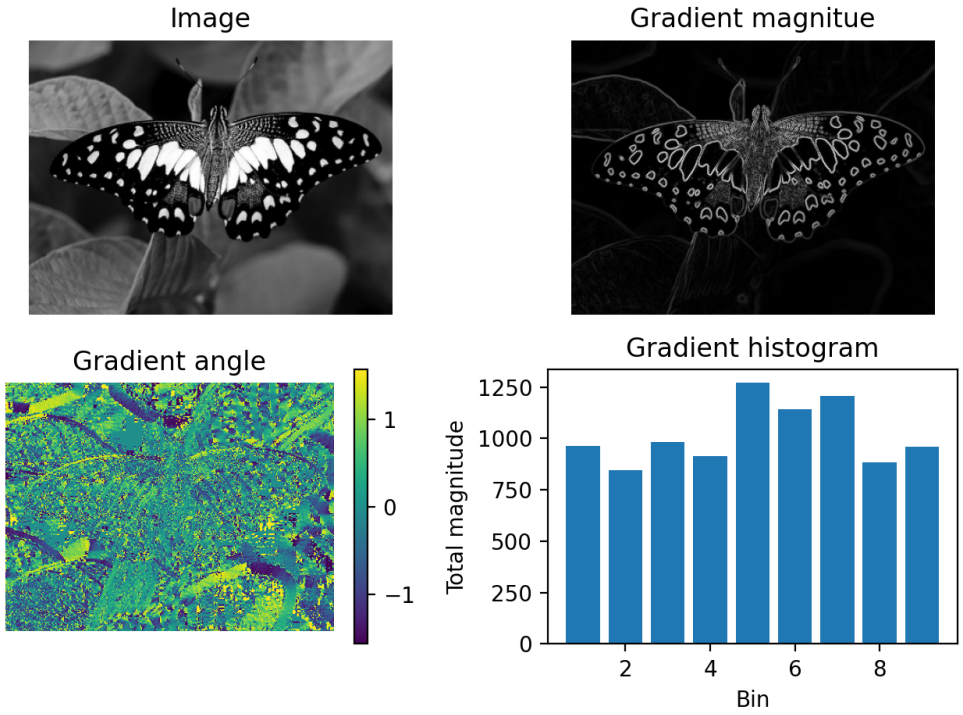

Figure 5: Gradient magnitude, angle and gradient histogram for the butterfly image.

4 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 6-top. Once you implement the corner detectors described in the next two sections the corners should be correctly detected as shown in Figure 6-bottom.

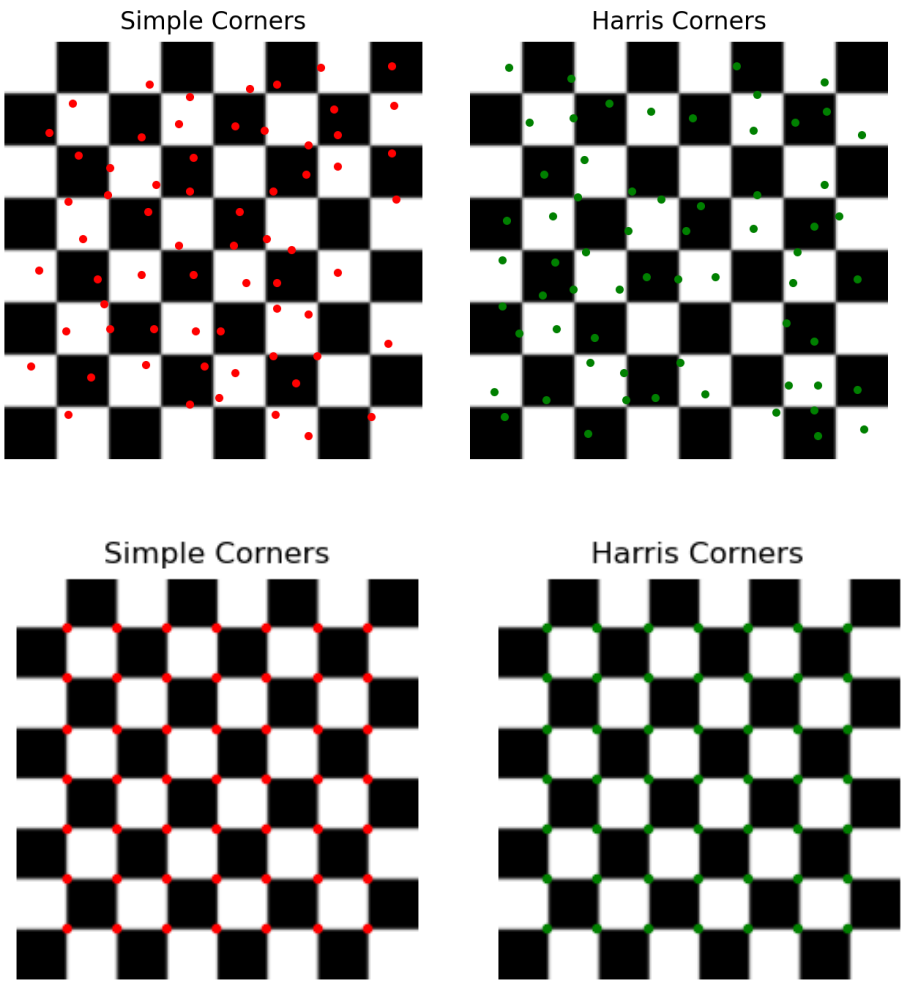

Figure 6: **(Top)** Initial output of `testCornerDetector`. **(Bottom)** Output after the correct implementation of `detectCorners`.

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.

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

Implement the function `detectCorners(I, isSimple=true, w, th)`. It returns 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 `simpleScore(I, w)` inside the file, which returns `cornerScore`, an 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), $(\cdot)^2$ 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.,

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

4.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,

$$M = \sum_{x,y} G_\sigma(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix},$$

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,

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

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 Python is that A*B does element-wise multiplication of A and B, which might save you a few "for" loops.

4.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.

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