Mini-project 2

CMPSCI 670, Fall 2019, UMass Amherst

Instructor: Subhransu Maji

TAs: Aruni Roy Chowdhury, Archan Ray

Guidelines

Submission. Submit a *single pdf document* via Gradescope that includes your solutions, figures and code. The latex source file for the homework is provided which you can modify to produce your report. You are welcome to use other typesetting software as long as the final output is a pdf. For readability you may attach the code printouts at the end of the solutions within the same pdf. Note that we will not run your code. Similarly figures should be included in a manner which makes it easy to compare various approaches. Poorly written or formatted reports will make it harder for the TA to evaluate it and may lead to a deduction of credit.

Late policy. You could have 24 hours late submission with a 50% mark down. Late submission beyond 24 hours will not be given *any* credits.

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. See the Universities' guidelines on academic honesty (`https://www.umass.edu/honesty`).

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.

Using other programming languages. We made the starter code available in Python and Matlab. You are free to use other languages such as C, Java, Octave or Julia with the caveat that we won't be able to answer or debug language-questions.

Python requirements. We will be using Python 2.7. The Python starter code requires `scipy`, `numpy` (at least v1.12), and `scikit-image`. If you are not familiar with installing those libraries through some package manager (like `pip`), the easiest way of using them is installing Anaconda.

1 Color image demosaicing [30 points]

Recall that in digital cameras the red, blue, and green sensors are interlaced in the Bayer pattern (Figure 1). The missing values are interpolated to obtain a full color image. In this part you will implement several interpolation algorithms.

Your entry point for this part of the homework is in `evalDemosaicing`. The code loads images from the data directory (in `data/demosaic`), artificially mosaics them (`mosaicImage` file), and provides them as input to the demosaicing algorithm (`demosaicImage` file). The input to the algorithm is a single image im, a NxM array of numbers between 0 and 1. These are measurements in the format shown in Figure 1, i.e., top left `im(1,1)|im[0,0]` is red, `im(1,2)|im[0,1]` is green, `im(2,1)|im[1,0]` is green, `im(2,2)|im[1,1]` is blue, etc. Your goal is to create a single color image C from these measurements.

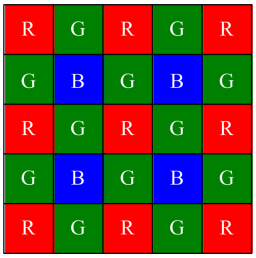

Figure 1: Bayer pattern

By comparing the result with the input we can also compute the error measured as the distance between the estimated image and the true image. This is what the algorithm reports. Running `evalDemosaic` from the Matlab command prompt produces the output shown below:

```
-------------------------------------------------------------------------
#   image              baseline      nn       linear     adagrad
-------------------------------------------------------------------------
1   balloon.jpeg       0.179239   0.179239   0.179239   0.179239
2   cat.jpg            0.099966   0.099966   0.099966   0.099966
3   ip.jpg             0.231587   0.231587   0.231587   0.231587
4   puppy.jpg          0.094093   0.231587   0.231587   0.231587
5   squirrel.jpg       0.121964   0.231587   0.231587   0.231587
6   pencils.jpg        0.183101   0.183101   0.183101   0.183101
7   house.png          0.117667   0.117667   0.117667   0.117667
8   light.png          0.097868   0.097868   0.097868   0.097868
9   sails.png          0.074946   0.074946   0.074946   0.074946
10  tree.jpeg          0.167812   0.167812   0.167812   0.167812
-------------------------------------------------------------------------
    average            0.136824   0.136824   0.136824   0.136824
-------------------------------------------------------------------------
```

Right now only the `demosaicImage(im, 'baseline')` is implemented which simply replaces all the missing values for a channel with the average value of that channel (while using python codes, if you're getting any warnings regarding **iCCP**, do `mogrify *.png` in the `data/demosiac` folder). All the other methods call the baseline algorithm and hence they produce identical results. Implement the following functions in the file:

- **[10 points]** `demosaicImage(im, 'nn')` – nearest-neighbour interpolation.
- **[10 points]** `demosaicImage(im, 'linear')` – linear interpolation.
- **[10 points]** `demosaicImage(im, 'adagrad')` – adaptive gradient interpolation.

In class we discussed how to interpolate the green channel. For the red and blue channels the algorithms will be different since there are half as many pixels with sampled values. For the adaptive gradient method start by interpolating only the green channel and using linear interpolation for the other two channels. Once the overall performance is better, think of a way of improving the interpolation for the red and blue channels. You can even apply different strategies to different pixels for the same channel!

For reference, the baseline method achieves an average error of 0.1392 across the 10 images in the dataset. Your methods you should be able to achieve substantially better results. The linear interpolation method achieves an error of about 0.017. To get full credit for this part you have to

- include your implementation of `demosaicImage`,
- include the output of `evalDemosaic` in a table format shown earlier,
- clearly describe the implementation details.

Tip: You can visualize at the errors by setting the display flag to true in the `runDemosaicing`. Avoid loops for speed in MATLAB/Python. Be careful in handling the boundary of the images.

2 Depth from disparity [35 points]

In this part of the homework, you will use a pair of images to compute a depth image of the scene. You will do this by computing a disparity map. A disparity map is an image that stores the displacement that leads every pixel in an image X to its corresponding pixel in the image X'. The depth image is inversely proportional to the disparity, as illustrated in Figure 2.

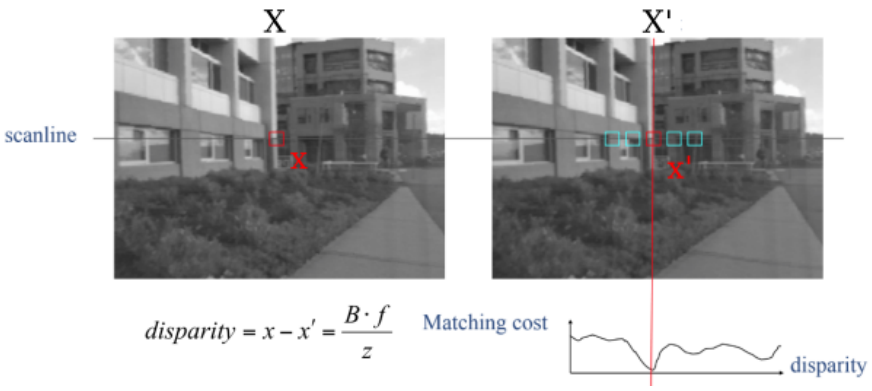

Figure 2: Visual description of depth map computation from a pair of images.

In order to compute the disparity map, you will have to associate each pixel x in X with the corresponding pixel x' in X'. This association can be computed by finding the image patch centered in x' that that has the smallest SSD to the patch centered in x. However, it is not necessary to compute SSD for every patch in X': for this homework, you can assume that the images planes are parallel to each other and have the same focal length, which means that you only need to search for the patches computed along the same epipolar line, as illustrated in Figure 2.

1. **[20 points] Compute depth image.** For this part, you will implement the function `depthFromStereo`. This function receives two images of the same scene and a patch size for SSD computation. It returns an image containing the relative depth. You can assume that the images planes are parallel to each other and have the same focal length. Once you compute the disparity image by finding patch correspondences along the same epipolar line, you can compute the depth through the following equation:

$$x - x' = \frac{Bf}{\text{depth}} \tag{1}$$

where B is the baseline and f is the focal length. Notice that we don't have access to B and f values, but those are constant throughout both images. This means that you will not be able to compute the true depth image, just the relative depth.

The entry code for this part is `evalDisparity`. The code loads a pair of images and you will implement a function that estimates the depth of each pixel in the first image, which you can visualize as a grayscale image. Assume $Bf = 1$ in your calculation. Show results for both pair of images included in the data directory.

To get good results you might have to experiment with the window size and the choice of patch similarity measure. Try a few different ones and report what worked well for you. Include a short discussion of on what parts of the image does the approach work well and where it fails.

2. **[5 points] Discussion.** In the first mini-project you implemented a method for depth estimation using photometric stereo. Discuss advantages and disadvantages of stereo over photometric stereo.

3. **[10 points] Informative patches**. Not all regions are equally informative for matching. As we discussed in class, one way to measure how informative a patch is using its second-moment matrix. For a pixel (u, v) this is defined as:

$$M = \begin{bmatrix} \sum_{x,y} I_x I_x & \sum_{x,y} I_x I_y \\ \sum_{x,y} I_x I_y & \sum_{x,y} I_y I_y \end{bmatrix}, \tag{2}$$

where the summation is over all pixels (x, y) within a window around (u, v). I_x and I_y are the gradients of the image and can be computed using pixel differences, e.g., $I_x(u, v) = I(u + 1, v) - I(u, v)$ and $I_y(u, v) = I_y(u, v + 1) - I(u, v)$. The eigenvalues of M indicate if the window is uniform, an edge, or a corner.

Let λ_1 and λ_2 be the eigenvalues of the matrix M. Write a function to compute the quantity $R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$ for $k = 0.03$ for each pixel given an image and window radius r. Visualize this quantity for the images `tsukuba_im1` and `poster_im2` for radius $r = 5$ and $r = 10$ as a heatmap (or grayscale image). What does the heatmap indicate? Describe this qualitatively.

Include your implementation in the submission (call the method `visualizeInformation`.) You may find the following identities useful for calculating R. Given a 2×2 matrix $M = [a\ b; c\ d]$, the $det(M) = \lambda_1\lambda_2 = ad - bc$ and $trace(M) = \lambda_1 + \lambda_2 = a + d$.

To get full credit for this part:

- include your implementation of `depthFromStereo`.
- include the computed depth image of the two image pairs we provided `poster` and `tsubuka`. Report which window size, similarity/distance function (if you found a better one over SSD) you used as well a discussion of failure modes.
- include the discussion comparing stereo and photometric stereo for depth estimation.
- include the implementation of `visualizeInformation`
- show a visualization of R for each image for two window sizes
- include discussion what does R indicate.

Tip: The relative depth for the `tsubuka` scene was shown in one of the lectures. The window-based is the one you are implementing. You can use that as reference for debugging purposes.

3 Report Writing and Presentation [10 points]

Please follow the guidelines for writing a good report. Graders will penalize reports that are poorly written and fail to present the results in a reasonable manner.

4 Extensions [10 points]

Implement at least one of the following to get up to 10 points. You can implement multiple for extra credit!

- **Transformed color spaces for demosaicing.** Try your demosaicing algorithms by first interpolating the green channel and then transforming the red and blue channels R $\leftarrow$ R/G and B $\leftarrow$ B/G, i.e., dividing by the green channel and then transforming them back after interpolation. Try other transformations such as logarithm of the ratio, etc (note: you have to apply the appropriate inverse transform). Does this result in better images measured in terms of the mean error? Why is this a good/bad idea?
- **Evaluate alternative sampling patterns.** Come up with a way of finding the optimal 2x2 pattern for color sampling. You can further simplify this problem by fixing the interpolation algorithm to linear and only using non-panchromatic cells (i.e, no white cells). A brute-force approach is to simply enumerate all the $3^4 = 81$ possible patterns and evaluate the error on a set of images and pick the best. What patterns work well? What are their properties (e.g., ratio of red, blue, green, cells)?