Mini-project 1: Color images

CMPSCI 670, Fall 2016, UMass Amherst
Due: September 29, 1:00 PM
Instructor: Subhransu Maji
TA: Tsung-Yu Lin

Guidelines

Submission. Submit a *single pdf document* via moodle that includes your solutions, figures and printouts of code. For readability you may attach the code printouts at the end of the solutions. 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.

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. All of the starter code is in Matlab which is what we expect you to use. You are free to use other languages such as Octave or Python with the caveat that we won't be able to answer or debug non Matlab questions.

1 Aligning Prokudin-Gorskii images [25 points]

Sergei Mikhailovich Prokudin-Gorskii (1863-1944) was a man well ahead of his time. Convinced, as early as 1907, that color photography was the wave of the future, he won Tzar's special permission to travel across the vast Russian Empire and take color photographs of everything he saw including the only color portrait of Leo Tolstoy. And he really photographed everything: people, buildings, landscapes, railroads, bridges... thousands of color pictures! His idea was simple: record three exposures of every scene onto a glass plate using a red, a green, and a blue filter. Never mind that there was no way to print color photographs until much later – he envisioned special projectors to be installed in "multimedia" classrooms all across Russia where the children would be able to learn about their vast country. Alas, his plans never materialized: he left Russia in 1918, right after the revolution, never to return again. Luckily, his RGB glass plate negatives, capturing the last years of the Russian Empire, survived and were purchased in 1948 by the Library of Congress. The LoC has recently digitized the negatives and made them available on-line.[1]

Figure 1: Example image from the Prokudin-Gorskii collection. On the left are the three images captured individually. On the right is a reconstructed color photograph. Note the colors are in B, G, R order from the top to bottom (and not R, G, B)!

Your goal is to take photographs of each plate and generate a color image by aligning them (Figure 1). The easiest way to align the plates is to exhaustively search over a window of possible displacements (say [-15,15] pixels), score each one using some image matching metric, and take the displacement with the best score. There is a number of possible metrics that one could use to score how well the images match. The simplest one is just the L2 norm also known as the Sum of Squared Differences (SSD) distance which is simply `sum(sum((image1-image2).^2))` where the sum is taken over the pixel values. Another is normalized cross-correlation (NCC), which is simply a dot product between two normalized vectors: `(image1./||image1||` and `image2./||image2||)`. Note that in the case of the Emir of Bukhara (Figure 1), the images to be matched do not actually have the same brightness values (they are different color channels), so you might have to use a cleverer metric, or different features than the raw pixels.

1.a Code

Download the `p1_code.tar.gz` and `p1_data.tar.gz` from the moodle page. The latex sources are also available as `p1_latex.tar.gz`. Move them to your homework directory and extract the files (e.g. by typing `tar -xvf p1_code.tar.gz`).

[1] This description and assignment is courtesy Alyosha Efros @ UC Berkeley

Before you start aligning the Prokudin-Gorskii images (in `data/prokudin-gorskii`), you will test your code on synthetic images which have been randomly shifted. Your code should correctly discover the inverse of the shift.

Run `evalAlignment.m` on the Matlab command prompt inside the code directory. This should produce the following output. Note the actual 'gt shift' might be different since it is randomly generated.

```
    Evaluating alignment ..
    1 balloon.jpeg
   gt shift: ( 1,11) ( 4, 5)
 pred shift: ( 0, 0) ( 0, 0)
    2 cat.jpg
   gt shift: (-5, 5) (-6,-2)
 pred shift: ( 0, 0) ( 0, 0)
     ...
```

The code loads a set of images, randomly shifts the color channels and provides them as input to `alignChannels.m`. Your goal is to implement this function. A correct implementation should obtain the shifts that is the negative of the ground-truth shifts, i.e. the following output:

```
    Evaluating alignment ..
    1 balloon.jpeg
   gt shift: ( 13, 11)   ( 4, 5)
 pred shift: (-13,-11)   (-4,-5)
    2 cat.jpg
   gt shift: (-5, 5) (-6,-2)
 pred shift: ( 5,-5) ( 6, 2)
     ...
```

Once you are done with that, run `alignProkudin.m`. This will call your function to align images from the Prokudin-Gorskii collection. The output is saved to the `outDir`. Note: if this directory already exists then the results will be overwritten.

1.b What to submit?

To get full credit for this part you have to

- include your implementation of `alignChannels.m`,
- verify that the `evalAlignment.m` correctly recovers the color image and shifts for the toy example and include the output results.
- include the aligned color images from the output of `alignProkudin.m`, *including* the computed shift vectors for each image.

1.c Some tips

- Look at functions `circshift()` and `padarray()` to deal with shifting images.
- How to include code in latex? Answer: The "verbatim" package provides ways to include pasted code, or code from a file, in its raw format. For example you can include the code in a file `alignChannels.m` by typing:

  ```
  \verbatiminput{../solution/alignChannels.m}}.
  ```

- Shifting images can cause ugly boundary artifacts. You might find it useful to crop the image after the alignment.

2 Color image demosaicing [30 points]

Recall that in digital cameras the red, blue, and green sensors are interlaced in the Bayer pattern (Figure 2). The missing values are interpolated to obtain a full color image. In this part you will implement several interpolation algorithms. 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 2, i.e., top left `im(1,1)` is red, `im(1,2)` is green, `im(2,1)` is green, `im(2,2)` is blue, etc. Your goal is to create a single color image C from these measurements.

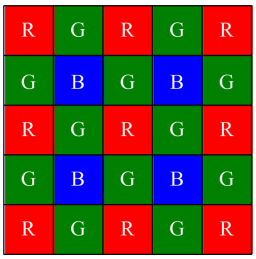

Figure 2: Bayer pattern

2.a Code

Your entry point for this part of the homework is in `evalDemosaic.m`. The code loads images from the data directory (in `data/demosaic`), artificially mosaics them (`mosaicImage.m` file), and provides them as input to the demosaicing algorithm (`demosaicImage.m` file). By comparing the result with the input we can also compute the reconstruction error measured as the distance between the reconstructed image and the true image. This is what the algorithm reports. Once you have implemented the `mosaicImage` function run the `evalDemosaic.m` and you should see the output below.

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. All the other methods call the baseline algorithm 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. You should report the results of all your methods in the form of the table 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
-------------------------------------------------------------------------
```

Tips: You can visualize at the errors by setting the display flag to true in the `runDemosaicing.m`. Avoid loops for speed in MATLAB. Be careful in handling the boundary of the images. It might help to think of various operations as convolutions. Look up MATLAB's conv2() function that implements 2D convolutions.

2.b What to submit?

To get full credit for this part you have to

- include your implementation of `demosaicImage.m`,
- include the output of `evalDemosaic.m`,
- clearly describe the implementation details.

3 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?
- **Coarse-to-fine alignment.** Searching over displacements can be slow. Think of a way of aligning them faster in a coarse-to-fine manner. For example you may align the channels by resizing them to half the size and then refining the estimate. This can be done by multiple calls to your alignChannels() function. If you implement this part then you can try your method on the high resolution .tif images in the http://inst.eecs.berkeley.edu/~cs194-26/fa16/hw/proj1/data/ directory.
- **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)?