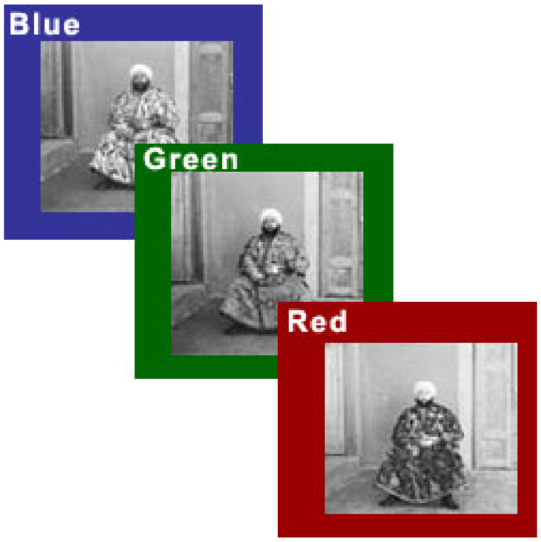
Programming Project #1 (`proj1`)

CS194-26 (CS294-26): Image Manipulation and Computational Photography



Images of the Russian Empire:

Colorizing the Prokudin-Gorskii photo collection

Due Date: 11:59pm on Tuesday, September 8, 2015

Background

Sergei Mikhailovich Prokudin-Gorskii (1863-1944) [Сергей Михайлович Прокудин-Горский, to his Russian friends] 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.

Overview

The goal of this assignment is to take the digitized Prokudin-Gorskii glass plate images and, using image processing techniques, automatically produce a color image with as few visual artifacts as possible. In order to do this, you will need to extract the three color channel images, place them on top of each other, and align them so that they form a single RGB color image. A cool explanation on how the Library of Congress created the color images on their site is available here.

Some starter code is available in MATLAB here and Python here; do not feel compelled to use it. We will assume that a simple x,y translation model is sufficient for proper alignment. However, the full-size glass plate images are very large, so your alignment procedure will need to be relatively fast and efficient.

DETAILS

A few of the digitized glass plate images (both hi-res and low-res versions) will be placed in the following directory (note that the filter order from top to bottom is BGR, not RGB!): data/. Your program will take a glass plate image as input and produce a single color image as output. The program should divide the image into three equal parts and align the second and the third parts (G and R) to the first (B). For each image, you will need to print the (x,y) displacement vector that was used to align the parts.

The easiest way to align the parts 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 (show on right), 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.

Exhaustive search will become prohibitively expensive if the pixel displacement is too large (which will be the case for high-resolution glass plate scans). In this case, you will need to implement a faster search procedure such as an image pyramid. An image pyramid represents the image at multiple scales (usually scaled by a factor of 2) and the processing is done sequentially starting from the coarsest scale (smallest image) and going down the pyramid, updating your estimate as you go. It is very easy to implement by adding recursive calls to your original single-scale implementation. Do not use Matlab's impyramid function but you can use imresize.

Your job will be to implement an algorithm that, given a 3-channel image, produces a color image as output. Implement a simple single-scale version first, using for loops, searching over a user-specified window of displacements. The above directory has skeleton Matlab code that will help you get started and you should pick one of the smaller .jpg images in the directory to test this version of the code. Next, add a coarse-to-fine pyramid speedup to handle large images like the .tiff ones provided in the directory.

BELLS & WHISTLES (EXTRA CREDIT)

Although the color images resulting from this automatic procedure will often look strikingly real, they are still a far cry from the manually restored versions available on the LoC website and from other professional photographers. Of course, each such photograph takes days of painstaking Photoshop work, adjusting the color levels, removing the blemishes, adding contrast, etc. Can we make some of these adjustments automatically, without the human in the loop? Feel free to come up with your own approaches or talk to me about your ideas. There is no right answer here -- just try out things and see what works. For example, the borders of the photograph will have strange colors since the three channels won't exactly align. See if you can devise an automatic way of cropping the border to get rid of the bad stuff. One possible idea is that the information in the good parts of the image generally agrees across the color channels, whereas at borders it does not.

Here are some ideas, but we will give credit for other clever ideas:

- Up to 4 pts: Automatic cropping. Remove white, black or other color borders. Don't just crop a predefined margin off of each side -- actually try to detect the borders or the edge between the border and the image.
- Up to 3 pts: Automatic contrasting. It is usually safe to rescale image intensities such that the

darkest pixel is zero (on its darkest color channel) and the brightest pixel is 1 (on its brightest color channel). More drastic or non-linear mappings may improve perceived image quality.
- Up to 5 pts: Automatic white balance. This involves two problems -- 1) estimating the illuminant and 2) manipulating the colors to counteract the illuminant and simulate a neutral illuminant. Step 1 is difficult in general, while step 2 is simple (see the Wikipedia page on Color Balance and section 2.3.2 in the Szeliski book). There exist some simple algorithms for step 1, which don't necessarily work well -- assume that the average color or the brightest color is the illuminant and shift those to gray or white.
- Up to 3 pts: Better color mapping. There is no reason to assume (as we have) that the red, green, and blue lenses used by Produkin-Gorskii correspond directly to the R, G, and B channels in RGB color space. Try to find a mapping that produces more realistic colors (and perhaps makes the automatic white balancing less necessary).
- Up to 3 pts: Better features. Instead of aligning based on RGB similarity, try using gradients or edges.
- Up to 5 pts: Better transformations. Instead of searching for the best x and y translation, additionally search over small scale changes and rotations. Adding two more dimensions to your search will slow things down, but the same course to fine progression should help alleviate this.
- Up to 4 pts: Aligning and processing data from other sources. In many domains, such as astronomy, image data is still captured one channel at a time. Often the channels don't correspond to visible light, but NASA artists stack these channels together to create false color images. For example, here is a tutorial on how to process Hubble Space Telescope imagery yourself. Also, consider images like this one of a coronal mass ejection built by combining ultraviolet images from the Solar Dynamics Observatory. To get full credit for this, you need to demonstrate that your algorithm found a non-trivial alignment and color correction.

For all extra credit, be sure to demonstrate on your web page cases where your extra credit has improved image quality.

Deliverables

For this project you must turn in both your code and a project webpage as described here.

- Please submit all the code you used to create your results, along with a main.m or main.py script that can be used to run your code and a README describing the contents of each code file, to the bCourses course site.
- Additionally, upload a zipped folder containting a web page to this site:
  - The folder should be named: "lastname_firstname_proj1" and then the whole folder should be zipped.
  - The folder should contain an index.html web page file that should include the following:
    - Text giving a brief overview of the project, and text describing your approach. If you ran into problems on images, describe how you tried to solve them. The website does not need to be pretty; you just need to explain what you did.
    - The result of your algorithm on **all** of our example images. List the offsets you calculated. Do not turn in the large .tiff images. Your web page should only display compressed images (e.g. jpg or png or gif if you want to animate something).
    - The result of your algorithm on a few examples of your own choosing, downloaded from the Prokudin-Gorskii collection.
    - If your algorithm failed to align any image, provide a brief explanation of why.
    - Describe any bells and whistles you implemented. For maximum credit, show before and after images.

- Remember **not to use any absolute links** to images etc on your computer, as these will not work online. Only use relative links within your folder.

FINAL ADVICE

- A lot of the suggested MATLAB code will be in the Image Processing Toolbox.
- For all projects, don't get bogged down tweaking input parameters. Most, but not all images will line up using the same parameters. Your final results should be the product of a fixed set of parameters (if you have free parameters). Don't worry if one or two of the handout images don't align properly using the simpler metrics suggested here.
- The input images can be in jpg (uint8) or tiff format (uint16), remember to convert all the formats to the same scale (seeÂ im2doubleÂ andim2uint8).
- Shifting a matrix is easy to do in MATLAB by using circshift.
- You can create the coordinates of the window you are shifting over by using meshgrid and turn that into a list of (x,y) pairs. This is not required to get the job done.
- The borders of the images will probably hurt your results, try computing your metric on the internal pixels only.
- Output all of your images to jpg, it'll save you a lot of disk space.

This assignment will be graded out of **100** points, as follows:

- **60 points** (**40** for those in the grad version of the class) for a single-scale implementation with successful results on low-res images.
- **40 points** for a multiscale pyramid version that works on the large images.
- Up to **10 points** for bells & whistles explicitly mentioned above.
- Up to **5 points** for bells & whistles you come up with on your own (and OK with course staff).