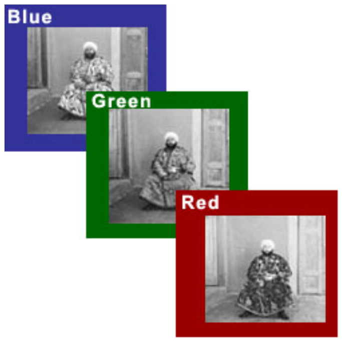
Day 1, Section 4: Colorizing the Prokudin-Gorskii Photographs



1 Introduction

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 the 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 digitized the negatives and made them available online[1].

2 Overview

The goal of this lab 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. There are a few steps in this process:

1. Split the image into three color channel images
2. Align the images to each other
3. Composite the channels into a single RGB image.

We've provided you with a function called `colorize` that does exactly this, but there's a catch - it calls another function called `align_image` to accomplish Step 2, which you'll need to fill in before `colorize` can work.

Exercise 1 *Read through* `colorize.m` *to familiarize yourself with how the code works, and how it uses the* `align_image` *function.*

[1]We've given you a few examples, but many more are available at `http://www.loc.gov/pictures/collection/prok/`

3 Image Alignment

Your main task will be to implement the `align_image` function. We've provided a file `align_image.m` that contains the outer parts of the function and help text describing what it should do. The easiest way to align the parts is to exhaustively search over a window of possible displacements (shifts) of one image, score each one using some image matching metric, and take the displacement with the best score.

There are two functions you'll find useful when implementing this. First, we've given you a function called `similarity_score` that does image matching. Use `help` to see how to use it. Second, we'll keep things simple by using the built-in `circshift` function to shift an image by a certain number of rows and columns.

Exercise 2 *Implement the missing part of* `align_image`*. Test your implementation by running* `colorize` *on several of the example images in the images directory (monastery.jpg, settlers.jpg, tobolsk.jpg). Try calling it with different values of* `window_size`*; how big do you need to make it for the alignment to work well?*

4 Optional: Better image matching

If you open up `similarity_score.m`, you can see that our image matching function isn't very complicated. The first line of the function body selects all but a 15-pixel border around the edge of each image, then subtracts them. This is to help get rid of the colored borders we see from the shifting. But if we shift by more than 15 pixels, this code won't necessarily crop the borders correctly. It would work better if we could specify how much to crop based on `window_size`.

Exercise 3 *Modify* `similarity_score` *to take an additional* `crop` *argument, and use that to determine the ranges of A and B that are compared. Then, modify* `align_image` *to call* `similarity_score` *with* `window_size` *to crop according to the maximum possible shift.*

5 Optional: High-resolution images

The course website also contains a link to a file `highres.zip`, which contains several much larger images. Download these images (it might take a few minutes) and put them in the images directory. While it's downloading, read on and try to devise an algorithm.

Because these images have much higher resolution, doing operations on them takes longer. Moreover, the pixels are smaller, so we need to search a much larger window. This makes our brute force approach problematically slow on images where each color channel is over 3000x3000 (9 megapixels).

See if you can think of a way to speed up the alignment for high-resolution images. A couple ideas:

- Shift the image by more than one pixel at a time.
- Do the alignment on smaller versions of the images (the `imresize` function will come in handy), then use the small-image result to align the full-sized images.
- Come up with an approximate alignment using one of the above methods, then refine it using a pixel-by-pixel search over a smaller window.

Exercise 4 *Come up with a way to speed up alignment of high-resolution images. If you find something that works efficiently for high-resolution images, see if you can make your code automatically work well regardless of the input resolution.*