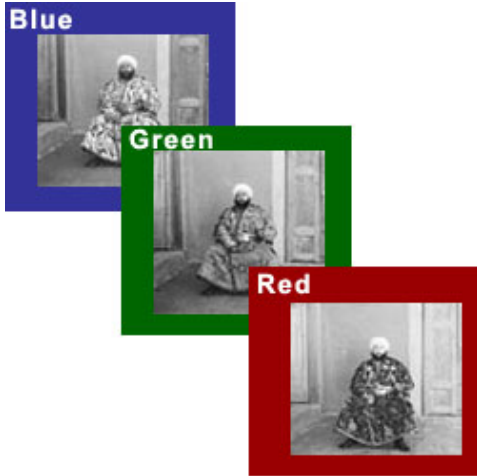




Programming Project #1 (proj1)

CS180: Intro to Computer Vision and Computational Photography



IMAGES OF THE RUSSIAN EMPIRE:

Colorizing the Prokudin-Gorskii photo collection

Due Date: September 12, 2025 at 11:59 PM

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. [This](#) is a cool explanation on how the Library of Congress composed their color images.

Some starter code is available in [Python](#); do not feel obligated 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 (i.e. .tif files) are very large, so your alignment procedure will need to be relatively fast and efficient. When you begin your naive implementation, you should start with the smaller files `monastery.jpg` and `cathedral.jpg` provided, or by downsizing the larger files. Your submission should be ran on the full-size images.

DETAILS

A few of the digitized glass plate images (both hi-res and low-res versions) will be placed in the following zip file (note that the filter order from top to bottom is BGR, not RGB!): [data.zip](#) ([online gallery for preview](#)). 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 (e.x. 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 **Euclidean Distance** which is simply $\sqrt{\text{sum}(\text{sum}((\text{image1}-\text{image2}).^2))}$ 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: $(\text{image1} ./ ||\text{image1}|| \text{ and } \text{image2} ./ ||\text{image2}||)$. Note that in the case like 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. You should implement the pyramid functionality yourself using appropriate downsampling techniques.

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 Python 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 .tif ones provided in the directory.



BELLS & WHISTLES (EXTRA FOR CS180, MANDATORY FOR CS280A)

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?

Bells and whistles that CS280A students **need** to implement, but CS180 students are welcome to do for a bragging right. *You can use any libraries to solve bells and whistles as long as you can explain what it is doing and why you used it.*

- **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.
- **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.
- **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.
- **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).
- **Better features.** Instead of aligning based on RGB similarity, try using gradients or edges.

(Optional) **Feel free to come up with your own approaches.** 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.

- **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.
- **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, this [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 truly show that your algorithm works, you should demonstrate a non-trivial alignment and color correction that your algorithm found.

DELIVERABLES

For this project you must turn in both your code and a project webpage as [described here](#).

- 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 upload the large .tif 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.
- **Submit your webpage URL to the class gallery via [Google Form](#).**

FINAL ADVICE

- You will be constructing an image pyramid in project 2 too, so writing nice reusable code will pay dividends.
- For the main assignment, you need to implement almost everything from scratch (except the functions for reading, writing, resizing, shifting and displaying images: e.g. imread, imresize, cirshift.). In particular, you are not allowed to use high level functions, such as these for constructing Laplacian/Gaussian pyramids, automatically aligning images, etc. If in doubt, ask on Ed.
- The average running time is expected to be less than 1 minute per image. If it takes hours for your program to finish, you should further optimize the code.
- Try to vectorize/parallelize your code, and avoid using too many FOR loops. (See more details for [Python](#) and [NumPy Broadcasting](#))
- 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.
- Convert images to floats. The input images can be in jpg (uint8) or tiff format (uint16), remember to convert all the formats to the same scale (see im2double and im2uint8).
- Shifting a matrix is easy to do in Python with np.roll.
- 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:

GRADING RUBRIC

CS180	CS280A
<ul style="list-style-type: none">• 60 points 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.	<ul style="list-style-type: none">• 40 points for a single-scale implementation with successful results on low-res images.• 30 points for a multiscale pyramid version that works on the large images.• 30 points for implementing the mandatory bells and whistles:<ul style="list-style-type: none">◦ Automatic cropping◦ Automatic contrasting◦ Automatic white balance◦ Better color mapping◦ Better features