University of Bucharest
Faculty of Mathematics and Computer Science
Department of Computer Science

Alexandra Diaconu
Ciprian-Mihai Ceauşescu
Bogdan Alexe
MSC in Computer Science

Computer Vision - Laboratory class 1

Introduction to Computer Vision

1.1 Introduction to Computer Vision using OpenCV

In the first part of this laboratory class we will introduce the OpenCV library in Python. During the next laboratory classes we will rely on OpenCV for solving different exercises. The script *Lab1-intro-opencv.ipynb* contains basic information regarding installing the OpenCV library and using several functions from this library for loading and displaying an image, accessing individual pixels, array slicing and cropping, resizing images, rotating an image, etc.

1.2 Aligning color channels from digitized glass plate images

Sergei Mikhailovich Prokudin-Gorskii (1863-1944) was a photographer ahead of his time. He saw color photography as the wave of the future and came up with a simple idea to

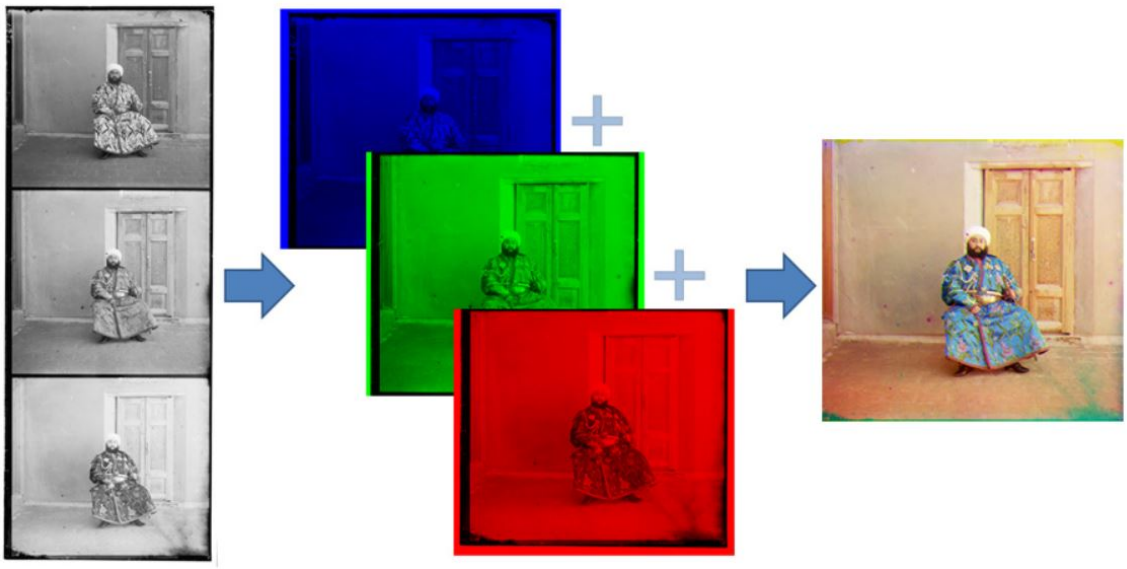

Figure 1: *We divide the input glass plate image (left) into three equal parts (middle) corresponding to the blue, green and red channels. Properly aligning the three color channels produces the color image depicting the Emir of Bukhara (right).*

produce color photos: record three exposures of every scene onto a glass plate using a red, a green, and a blue filter and then project the monochrome pictures with correctly coloured light to reproduce the color image; color printing of photos was very difficult at the time. Due to the fame he received from his color photos, including the only color portrait of Leo Tolstoy (a famous Russian author), he won the Tzar's permission and funding to travel across the Russian Empire and document it in 'color' photographs. His RGB glass plate negatives were purchased in 1948 by the Library of Congress. They are now digitized and available online.

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 automatically produce a single color image as output with as few visual artifacts as possible. The program should extract the three color channel images and align them so that they form a single RGB color image. The easiest way to align the color channels 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 metric is the normalized cross-correlation (NCC), which is simply a dot product between two normalized vectors: $(image1/||image1||$ and $image2/||image2||)$

Important! 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 more clever 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.

Take the digitized Prokudin-Gorskii glass plate images and automatically produce a color image with as few visual artifacts as possible. Your program should:

- Divide the image into three equal plates (corresponding to the three image color channels) and align the three color channels (blue is top, green is middle, red is bottom);

- Implement a single-scale sliding window alignment search with an appropriate metric, and record the displacement vector used to align the parts;
- The high-resolution images are quite large—for efficiency, implement a multiscale image pyramid alignment algorithm. Report any difference between the single- and multi-scale alignment vectors, plus the speedup factor of the multi-scale approach;
- Try your algorithm on other images from the collection *(data folder)*.

Creating the pyramid: In the multi-level implementation we first build an image pyramid for each color channel. This means the original image is stored at a number of different scales, each one half the size of the previous. The shrinking can be done by resizing the image. Once we have the pyramid the alignment operation is first performed on the *smallest scale image* and the *resulting offset used as the starting point for alignment on the next size up*. The process is repeated all the way down the pyramid until we have the optimal offset for the original image. This allows the region of offsets we consider at each level to be much smaller, in this case ±4 pixels, as we have an approximate guess from the previous level and at the smallest level ±2 pixels is a significant portion of the image. This is most significant at the larger levels where there are lots of pixels to consider when calculating the alignment score. Overall this method is much faster than the naive implementation on high-resolution images.

1.3 Demosaicing

In this exercise, we are going to 'demosaic' an image encoded with the Bayer Pattern. There are some cameras that use the Bayer Pattern in order to save an image. Using this encoding only 50% of green pixels, 25% of red pixels and 25% of blue pixels are kept. The Bayer encoding takes a RGB image and encodes it in one of the four possible ways as depicted in Figure 2.

Figure 2: *Bayer pattern, specified as one of the four possible values. Each value represents the order of the red, green, and blue sensors by describing the four pixels in the upper-left corner of the image (left-to-right, top-to-bottom).*

In this exercise, we are going to 'demosaic' an encoded image assuming the encoding with the RGGB pattern. We will implement a very simple algorithm which, for each pixel, fills in the two missing channels by averaging the values of their nearest neighbors (1, 2 or 4)

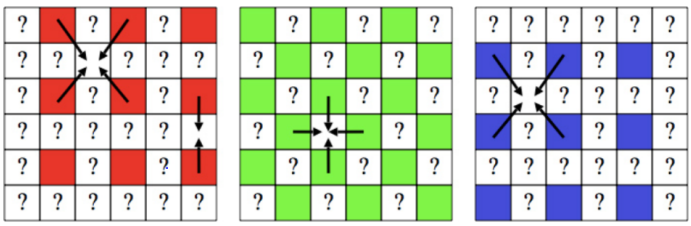

Figure 3: *For each pixel we fill in the two missing values by averaging the values of their nearest neighbors (1, 2 or 4) in the corresponding channel.*

in the corresponding channel.

To complete this task, we have to do:

- read the encoded image (*crayons_mosaic.bmp*);
- recreate the green, red and blue channel by copying the values into the corresponding positions of each channel;
- interpolate the missing values in each channel.