Assignment_Feature_based_Image_Alignment_Solution

October 30, 2019

1 Feature Based Image Alignment in OpenCV

|

Figure 1. Left: An image from the Prokudin-Gorskii Collection. Right : The same image with the channels aligned.

|

The image on the left is part of a historic collection of photographs called the **Prokudin-Gorskii** collection. The image was taken by a Russian photographer in the early 1900s using one of the early color cameras. The color channels of the image are misaligned because of the mechanical nature of the camera. The image on the right is a version of the same image with the channels brought into alignment using a function available in OpenCV 3.

In this post we will embark on a fun and entertaining journey into the history of color photography while learning about image alignment in the process. This post is dedicated to the early pioneers of color photography who have enabled us to capture and store our memories in color.

2 A Brief and Incomplete History of Color Photography

|

Figure 2. Ribbon by Maxwell and Sutton was the first color photo ever taken by superimposing 3 grayscale images.

|

The idea that you can take three different photos using three primary color filters (Red, Green, Blue) and combine them to obtain a color image was first proposed by James Clerk Maxwell (yes, the Maxwell) in 1855. Six years later, in 1861, English photographer Thomas Sutton produced the first color photograph by putting Maxwell's theory into practice. He took three grayscale images of a Ribbon (see Figure 2), using three different color filters and then superimposed the images using three projectors equipped with the same color filters. The photographic material available at that time was sensitive to blue light, but not very sensitive to green light, and almost insensitive

to red light. Although revolutionary for its time, the method was not practical.

By the early 1900s, the sensitivity of photographic material had improved substantially, and in the first decade of the century a few different practical cameras were available for color photography. Perhaps the most popular among these cameras, the *Autochrome*, was invented by the Lumière brothers.

A competing camera system was designed by Adolf Miethe and built by Wilhelm Bermpohl, and was called *Professor Dr. Miethe's Dreifarben-Camera*. In German the word "Dreifarben" means tri-color. This camera, also referred to as the Miethe-Bermpohl camera, had a long glass plate on which the three images were acquired with three different filters. A very good description and an image of the camera can be found here.

In the hands of Sergey Prokudin-Gorskii the Miethe-Bermpohl camera (or a variant of it) would secure a special place in Russian history . In 1909, with funding from Tsar Nicholas II, Prokudin-Gorskii started a decade long journey of capturing Russia in color! He took more than 10,000 color photographs. The most notable among his photographs is the only known color photo of Leo Tolstoy.

Fortunately for us, the Library of Congress purchased a large collection of Prokudin-Gorskii's photographs in 1948. They are now in the public domain and we get a chance to reconstruct Russian history in color!

|

Figure 3 : Three images captured on a vertical glass plate by a Miethe-Bermpohl camera.

|

It is not trivial to generate a color image from these black and white images (shown in Figure 3). The Miethe-Bermpohl camera was a mechanical device that took these three images over a span of 2-6 seconds. Therefore the three channels were often mis-aligned, and naively stacking them up leads to a pretty unsatisfactory result shown in Figure 1.

Well, it's time for some vision magic!

3 Reconstructing Prokudin-Gorskii Collection in Color

The above image is also part of the Prokudin-Gorskii collection. On the left is the image with unaligned RGB channels, and on the right is the image after alignment. This photo also shows that by the early 20th century the photographic plates were sensitive enough to beautifully capture a wide color spectrum. The vivid red, blue and green colors are stunning.

Computer Vision in real world is tough, things often do not really work out of the box.

The problem is that the red, green, and blue channels in an image are not as strongly correlated if in pixel intensities as you might guess. For example, check out the blue gown the Emir is wearing in Figure 3. It looks quite different in the three channels. However, even though the intensities are different, something in the three channels is similar because a human eye can easily tell that it is the same scene.

Let's see if Computer Vision can match the powers of human eye in this case.

4 Assignment Instructions

In this assignment, you will implement **Feature Matching based Image Alignment**. This assignment carries **30 marks** and you will be allowed a maximum of **5 submissions**.

The assignment will be **manually graded** by Staff members.

4.1 Grading Rubric

1. **Step 1**: Read Image - **No Marks**
2. **Step 2**: Detect Features - **6 Marks**
3. **Step 3**: Match Features - **6 Marks**
4. **Step 4**: Calculate Homography - **12 Marks**
5. **Step 5**: Warping Image - **6 Marks**
6. **Step 6**: Merge Channels - **No Marks**

4.1.1 *You can refer to code given in the previous sections for completing this assignment*

4.2 Step 1: Read Image

In [1]:
```
#include "includeLibraries.h"
#include <iostream>
#include <opencv2/opencv.hpp>
#include "matplotlibcpp.h"
#include "displayImages.h"
```

In [2]:
```
using namespace std;
```

In [3]:
```
using namespace cv;
```

In [4]:
```
using namespace matplotlibcpp;
```

In [5]:
```
// Read 8-bit color image.
// This is an image in which the three channels are
// concatenated vertically.
Mat img =  imread(DATA_PATH + "images/emir.jpg", IMREAD_GRAYSCALE);
```

In [6]:
```
// Find the width and height of the color image
Size sz = img.size();
int height = sz.height / 3;
int width = sz.width;
```

In [7]:
```
cout << sz;
```

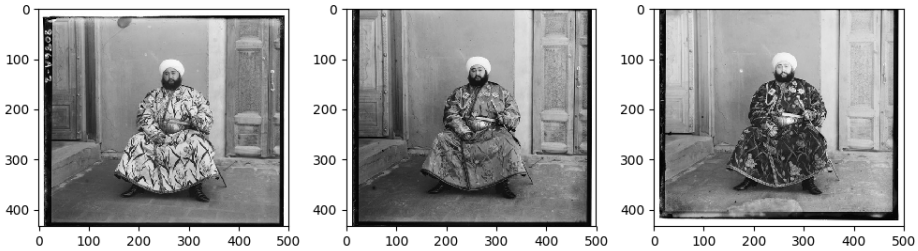
```
[500 x 1300]
```

In [8]:
```
// Extract the three channels from the gray scale image
vector<Mat>channels;
channels.push_back(img( Rect(0, 0,         width, height)));
channels.push_back(img( Rect(0, height,    width, height)));
channels.push_back(img( Rect(0, 2*height,  width, height)));
```

```
In [9]: Mat blue = channels[0];
        Mat green = channels[1];
        Mat red = channels[2];
```

```
In [10]: plt::figure_size(1200,300);
         plt::subplot(1,3,1);
         plt::imshow(blue);
         auto pltImg = displayImage(blue);
         plt::subplot(1,3,2);
         plt::imshow(green);
         pltImg = displayImage(green);
         plt::subplot(1,3,3);
         plt::imshow(red);
         pltImg = displayImage(red);
         pltImg
```

Out[10]:



4.3 Step 2: Detect Features - 6 Marks

We will align Blue and Red frame to the Green frame. Take a minute and think about a valid reason.

If you align blue and red channels, it might not give very good results since they are visually very different. You may have to do a lot parameter tuning (MAX_FEATURES, GOOD_MATCH_PERCENT, etc) to get them aligned. On the other hand, Blue and Green channels are reasonably similar. Thus, taking green as the base channel will produce best results.

We detect ORB features in the 3 frames. Although we need only 4 features to compute the homography, typically hundreds of features are detected in the two images. We control the number of features using the parameter `MAX_FEATURES` in the C++ code.

Set MAX_FEATURES and GOOD_MATCH_PERCENT (You may need to play around with these parameters to get the desired result)

```
In [11]: ///
         int MAX_FEATURES = 650;
         float GOOD_MATCH_PERCENT = 0.1055f;
         ///
```

Detect ORB features and compute descriptors.

You need to find the keypoints and descriptors for each channel using ORB and store them in respective variables

e.g. keyPointsBlue and descriptorsBlue are the keypoints and descriptors for the blue channel

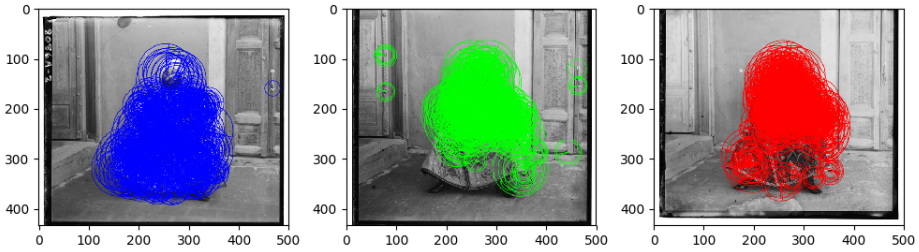
In [12]:
```
///
// Variables to store keypoints and descriptors
std::vector<KeyPoint> keypointsBlue, keypointsGreen, keypointsRed;
Mat descriptorsBlue, descriptorsGreen, descriptorsRed;

// Detect ORB features and compute descriptors.
Ptr<Feature2D> orb = ORB::create(MAX_FEATURES);
orb->detectAndCompute(green, Mat(), keypointsGreen, descriptorsGreen);
orb->detectAndCompute(blue, Mat(), keypointsBlue, descriptorsBlue);
orb->detectAndCompute(red, Mat(), keypointsRed, descriptorsRed);
///
```

4.3.1 Step 3: Match Features

We find the matching features in the two images, sort them by goodness of match and keep only a small percentage of original matches. We finally display the good matches on the images and write the file to disk for visual inspection. We use the hamming distance as a measure of similarity between two feature descriptors. The matched features are shown in the figure below by drawing a line connecting them. Notice, we have many incorrect matches and thefore we will need to use a robust method to calculate homography in the next step.

In [13]:
```
plt::figure_size(1200,300);
Mat img2;
drawKeypoints(blue, keypointsBlue, img2, Scalar(255,0,0), DrawMatchesFlags::DRAW_RICH
plt::subplot(1,3,1);
plt::imshow(img2);
pltImg = displayImage(img2);

drawKeypoints(green, keypointsGreen, img2, Scalar(0,255,0), DrawMatchesFlags::DRAW_RI
plt::subplot(1,3,2);
plt::imshow(img2);
pltImg = displayImage(img2);

drawKeypoints(red, keypointsRed, img2, Scalar(0,0,255), DrawMatchesFlags::DRAW_RICH_K
plt::subplot(1,3,3);
plt::imshow(img2);
pltImg = displayImage(img2);

pltImg
```

Out[13]:



4.4 Step 3: Match Features - 6 Marks

You need to find the matching features in the Green channel and blue/red channel, sort them by goodness of match and keep only a small percentage of original matches. We finally display the good matches on the images and write the file to disk for visual inspection. Use the hamming distance as a measure of similarity between two feature descriptors.

Let's first match features between blue and Green channels.

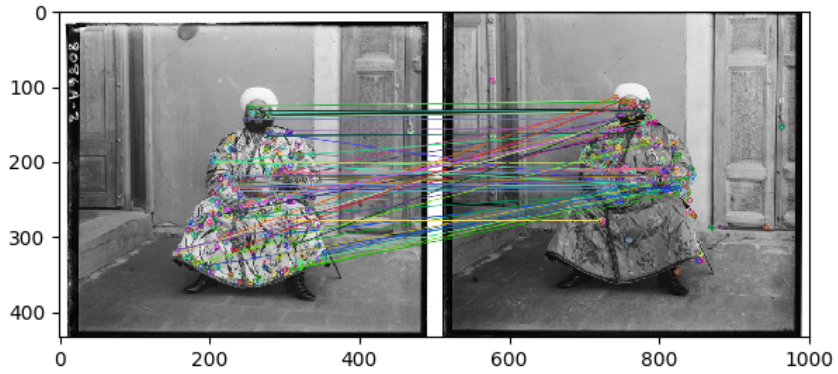
Find the matches between Blue and Green channels and save them in matchesBlueGreen variable

In [14]:
```
// Match features.
///
std::vector<DMatch> matchesBlueGreen;
Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");

///

// Match features between blue and Green channels
///
matcher->match(descriptorsBlue, descriptorsGreen, matchesBlueGreen, Mat());
///

// Sort matches by score
std::sort(matchesBlueGreen.begin(), matchesBlueGreen.end());

// Remove not so good matches
int numGoodMatches = matchesBlueGreen.size() * GOOD_MATCH_PERCENT;
matchesBlueGreen.erase(matchesBlueGreen.begin()+numGoodMatches, matchesBlueGreen.end()

// Draw top matches
Mat imMatchesBlueGreen;
drawMatches(blue, keypointsBlue, green, keypointsGreen, matchesBlueGreen, imMatchesBlu
```

In [15]:
```
plt::figure_size(800,300);
plt::imshow(imMatchesBlueGreen);
```

```
pltImg = displayImage(imMatchesBlueGreen);
pltImg
```

Out[15]:



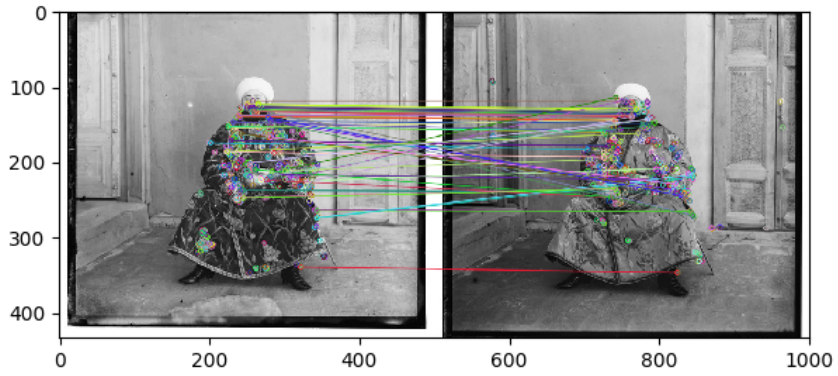
We will repeat the same process for Red and Green channels this time.

Find the matches between Red and Green channels and save them in matchesRedGreen variable

In [16]:
```
// Match features between Red and Green channels
///
std::vector<DMatch> matchesRedGreen;
matcher->match(descriptorsRed, descriptorsGreen, matchesRedGreen, Mat());
///

// Sort matches by score
std::sort(matchesRedGreen.begin(), matchesRedGreen.end());

// Remove not so good matches
numGoodMatches = matchesRedGreen.size() * GOOD_MATCH_PERCENT;
matchesRedGreen.erase(matchesRedGreen.begin()+numGoodMatches, matchesRedGreen.end());

// Draw top matches
Mat imMatchesRedGreen;
drawMatches(red, keypointsRed, green, keypointsGreen, matchesRedGreen, imMatchesRedGr
```

In [17]:
```
plt::figure_size(800,300);
plt::imshow(imMatchesRedGreen);
pltImg = displayImage(imMatchesRedGreen);
pltImg
```

Out[17]:



4.5 Step 4: Calculate Homography - 12 Marks

Next, you need to compute the homography between the green and red/blue channels using the matches and keypoints computed in the previous step.

Let's first calculate the homography between Blue and Green channels.

Find the homography matrix between the Blue and Green channel and name it hBlueGreen

```
In [18]: // Extract location of good matches
         ///
         std::vector<Point2f> points1, points2;
         for (size_t i = 0; i < matchesBlueGreen.size(); i++)
         {
             points1.push_back(keypointsBlue[matchesBlueGreen[i].queryIdx].pt);
             points2.push_back(keypointsGreen[matchesBlueGreen[i].trainIdx].pt);
         }
         ///

         // Find homography
         ///
         Mat hBlueGreen = findHomography(points1, points2, RANSAC);
         ///
```

Similarly, we can calculate the homography between Green and Red channels.

Find the homography matrix between the Red and Green channel and name it hRedGreen

```
In [19]: // Extract location of good matches
         ///
         std::vector<Point2f> points3, points4;

         for (size_t i = 0; i < matchesRedGreen.size(); i++)
         {
```

```
        points3.push_back(keypointsRed[matchesRedGreen[i].queryIdx].pt);
        points4.push_back(keypointsGreen[matchesRedGreen[i].trainIdx].pt);
    }
    ///

    // Find homography
    ///
    Mat hRedGreen = findHomography(points3, points4, RANSAC);
    ///
```

4.6 Step 5: Warping Image - 6 Marks

Once an accurate homography has been calculated, the transformation can be applied to all pixels in one image to map it to the other image. This is done using the `warpPerspective` function in OpenCV.

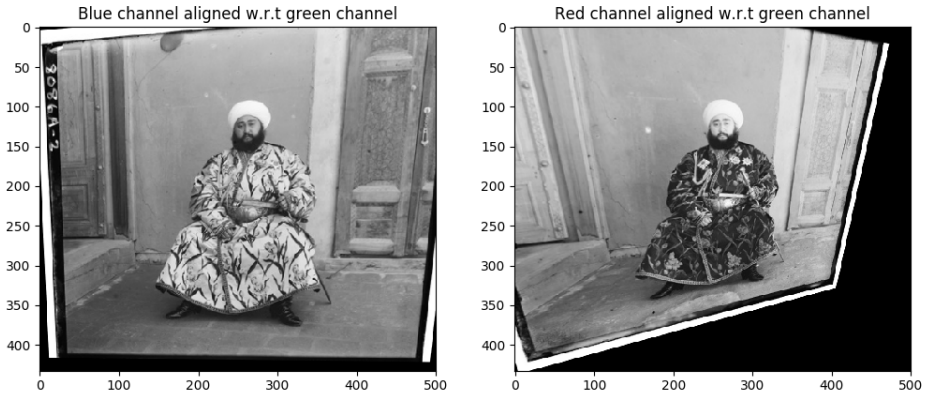
We map the Blue and Red channels to Green channel pixels.

Find the warped images blueWarped and redWarped using the warpPerspective function

In [20]:
```
// Use homography to find blueWarped and RedWarped images
///

int imgHeight = blue.rows;
int imgWidth = blue.cols;

Mat blueWarped;
warpPerspective(blue, blueWarped, hBlueGreen, Size(imgWidth, imgHeight));

Mat redWarped;
warpPerspective(red, redWarped, hRedGreen, Size(imgWidth, imgHeight));

///
```

In [21]:
```
plt::figure_size(1200,500);
plt::subplot(1,2,1);
plt::imshow(blueWarped);
plt::title("Blue channel aligned w.r.t green channel");
pltImg = displayImage(blueWarped);
plt::subplot(1,2,2);
plt::imshow(redWarped);
plt::title("Red channel aligned w.r.t green channel");
pltImg = displayImage(redWarped);
pltImg
```

Out[21]:

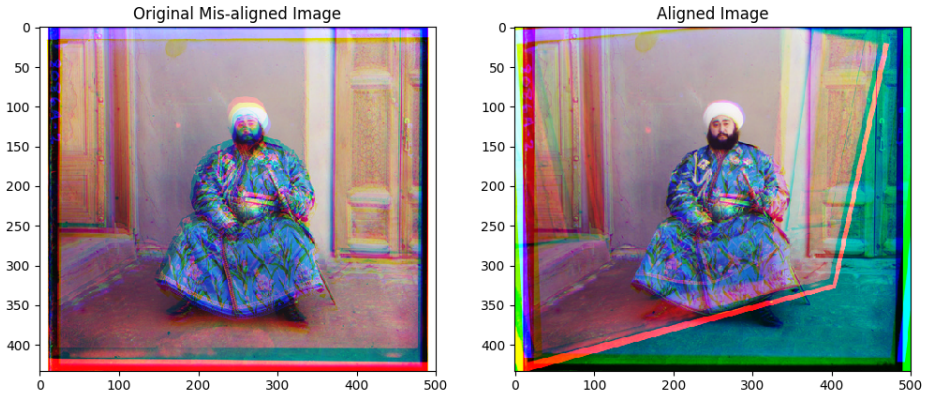


4.7 Step 6: Merge Channels

Finally, let's merge the channels to form the final colored image.

```
In [22]: Mat colorImage;
         vector<Mat> colorImageChannels {blueWarped, green, redWarped};
         merge(colorImageChannels,colorImage);

In [23]: Mat originalImage;
         merge(channels,originalImage);

In [24]: plt::figure_size(1200,500);
         plt::subplot(1,2,1);
         plt::imshow(originalImage);
         plt::title("Original Mis-aligned Image");
         pltImg = displayImage(originalImage);
         plt::subplot(1,2,2);
         plt::imshow(colorImage);
         plt::title("Aligned Image");
         pltImg = displayImage(colorImage);
         pltImg
```

Out[24]:


Original Mis-aligned Image
Aligned Image
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500