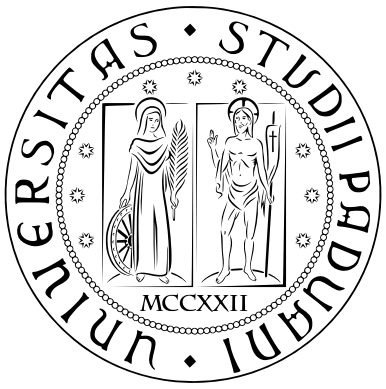

Università degli Studi di Padova

DEI - Dipartimento di Ingegneria dell'Informazione UniPD
Computer Vision Course

Computer Vision Final Project

Student: Zuccolo Giada, matr. 2055702

Date: June 12, 2022

Contents

1 Introduction

This document explains the development of the Computer Vision Final Project, about *Image Mosaicing*, based on Lab 4. The goal of this project is to develop a code that permit you to stitch multiple images together.
To understand the procedure and how the code is been developed, it is necessary some theory tips that have been applied for this project.

2 Theoretical Part

> *Image stitching is the process of combining multiple photographic images with overlapping fields of view to produce a segmented panorama or high-resolution image.*
> – Wikipedia

The porpouse is to overlap the images to produce seamless results, that is without seeing where an image stops and another one begins, to seems an only one picture. For this reason, it is also better have identical exposures, to avoid to see the difference from the images that compose the main picture.
To do this, our algorithm will do these steps:

1. Detecting `keypoints` and extracting local invariant `descriptors`.
   There operation will be done with the **SIFT (Scale-invariant feature transform)**, which is a detector is used in the detection of interest points on an input image, that we will use them to stitch the images between them.

2. Matching descriptors between images with **BFMatcher**, which is a **Brute-Force Matcher** that takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation.

3. Now, the goal is to found the best matches from these points. To do it we use the principle of the **homography**. Homography is an isomorphism of projective spaces, induced by an isomorphism of the vector spaces from which the projective spaces derive. In a simply way, it is a one-to-one correspondence between two projective spaces of the same dimension.
   With the Homography, given set of points from both the images, we will find the perspective transformation. To do it, we will use the **RANSAC (Random Sample Consensus)**, an iterative and robust algorithm for robust estimation of the model parameters in a presence of outliers- that is, data points which are noisy and wrong. Good matches which provide correct estimation are called inliers and remaining are called outliers. A function will return us a mask which specifies the inlier and outlier points.

4. using the generated **homography matrix**, now we will be able to merge the two images, applying the **warping perspective transformation**, which finds the transformation matrix that best describes the projective transform from reference image to target image and applies the projection that will align the images.

3 Development

3.1 The idea behind the algorithm

The assigned goal was, starting from a 3x3 grid of images, to be able to reconstruct the original image, as in the puzzle game, as shown in the example below (figure 3.1 and figure 3.2).

Figure 3.1: Original Image

Figure 3.2: 3x3 grid of Images

The developed algorithm has the task of stitching the contiguous images together, two images at a time. It is therefore necessary to manage the image grid as a two-dimensional matrix, where each image is named according to the position it occupies in the original image:

00.jpg	01.jpg	02.jpg
10.jpg	11.jpg	12.jpg
20.jpg	21.jpg	22.jpg

The algorithm reads this grid starting first from the rows, and for each row it scrolls the columns: it will start from the image occupying position 00 and stitching the image occupying position 01.
The problem lies in the fact that the grid of images is of the 3x3 type, and it is not possible to create a chain of concatenation of three images with the procedure adopted.
So, the algorithm first stitches the first two images of each row and saves the resulting image. Then use this resulting merged image to stitch with the image occupying position 02, then the last of the row, as shown in image 3.3.
In this way the algorithm is deceived and it is always made to work on two images, when instead in the second turn inside the row, it will operate with an image that is the result of the stitching operations of the previous step. For each row then you get a panoramic image, with 3 images that will merge into a single image.
This procedure is repeated for all three lines of the grid, obtaining at the end 3 panoramic images, one for each line.

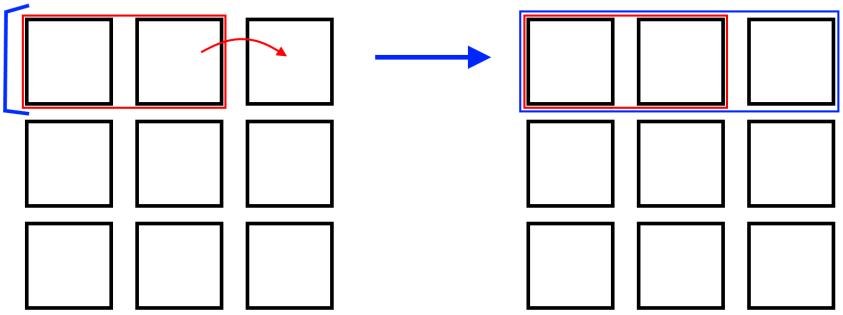

Figure 3.3: Rows operations

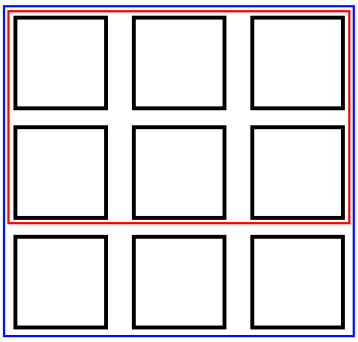

Figure 3.4: Merging operations

At this point, stitching must also be carried out vertically, joining row 0 with row 1 which in turn must be joined to row 2.
As shown in the figure 3.4, the procedure adopted is the same: first the operations that lead to the stitching of the panoramic image obtained for row 0 are carried out with the panoramic image obtained for row 1, obtaining an image that will then be used as a starting image to stitch with the panoramic image obtained for row 2.
The result obtained must be the original image.

3.2 Coding part

The code is now explained, showing the functions and the flow of instructions adopted.

Horizontal operations

Two `for` loops were used to iterate through the rows and, for each row, all the columns. In the innermost `for` loop, a condition determines whether we have to take into account the first two images of the row (e.g. 00-01, or 10-11) or if we are already at the second step, and therefore the The image 1 to choose is the result of the stitching that took place in the previous step. The path to the image to be fetched is then defined within the condition: to "build" the path, the indexs of the two `for` are used.

```
for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            if (j == 0) {
                path = dir_path +"patch/" + to_string(i) +
                ↪  to_string(j) + ".jpg";
                img1 = load_img(path);
                path2 = dir_path + "patch/" + to_string(i) +
                ↪  to_string(j + 1) + ".jpg";
                img2 = load_img(path2);
            }

            if (j == 1) {
                string val = to_string(i) + to_string(j - 1) + "-" +
                ↪  to_string(i) + to_string(j);
                path = dir_path + "partial_merge/merge_" + val +
                ↪  ".jpg";
                img1 = load_img(path);
                path2 = dir_path + "patch/" + to_string(i) +
                ↪  to_string(j + 1) + ".jpg";
                img2 = load_img(path2);
            }
```

Listing 3.1: For cycles and conditions to load the correct images

After this step, it's time to work with SIFT. Two functions have been created to facilitate subsequent steps.
The first function called `"sift_img_ret_descr"` performs the SIFT, saves the sifted image (i.e. with the keypoints, as you can see in the figure 3.5 and in the figure 3.6) and returns the descriptors (`Mat` type).
The second function called `"sift_img_ret_keypoint"` performs the SIFT and returns the vector of keypoints (`vector<KeyPoint>`).

```
int valSift = 0;
Mat sift_img_ret_descr(Mat img, int x, int y) {
    Ptr<SiftFeatureDetector> detector =
    ↪  SiftFeatureDetector::create();
    vector<KeyPoint> keypoints;
    Mat descriptors;
    detector->detectAndCompute(img, Mat(), keypoints, descriptors);
    Mat output;
    drawKeypoints(img, keypoints, output);
    string val = to_string(x) + to_string(y);
    string path_sifted_img = dir_path + "sifted_img/img_" +
    ↪  to_string(valSift) + "_sift.jpg";
    valSift++;
    imwrite(path_sifted_img, output);
    cout << "saved " + path_sifted_img << endl;
    return descriptors;
}

vector<KeyPoint> sift_img_ret_keypoint(Mat img, int x, int y) {
    Ptr<SiftFeatureDetector> detector =
    ↪  SiftFeatureDetector::create();
    vector<KeyPoint> keypoints;
    Mat descriptors;
    detector->detectAndCompute(img, Mat(), keypoints, descriptors);
    Mat output;
    drawKeypoints(img, keypoints, output);
    return keypoints;
}
```

Listing 3.2: SIFT functions

Now it's the turn of BFMatcher, with the (default value) `NORM_L2` parameter, which specifies the distance measurement to be used. BFMatcher uses the descriptor that is was fetched before during the sift.
As in the case of the sifted images, we choose to save the resulting images, in order to see the result obtained (figure 3.7).

```
Ptr<BFMatcher> matcher = BFMatcher::create(NORM_L2);
vector<DMatch> matches, better_matches;
matcher->match(sift_desc_1, sift_desc_2, matches);
```

Listing 3.3: BFMatcher code

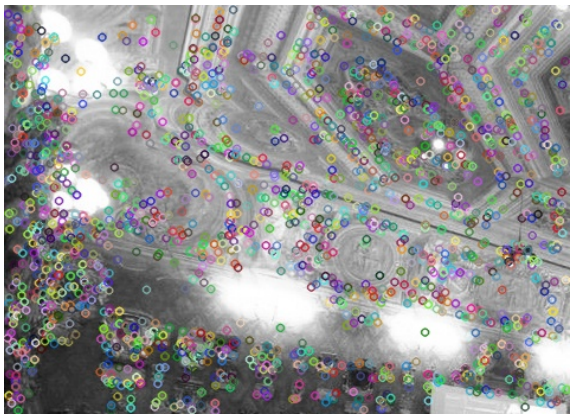

Figure 3.5: Sifted img 00

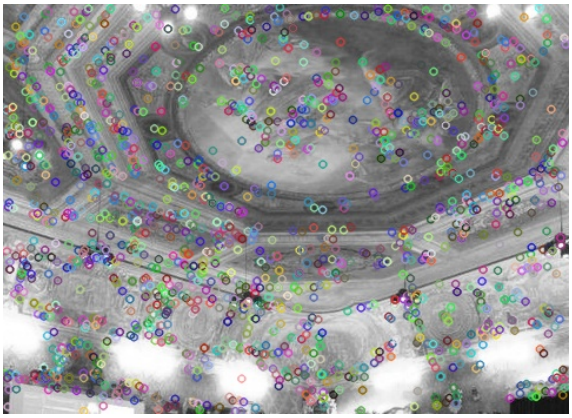

Figure 3.6: Sifted img 01

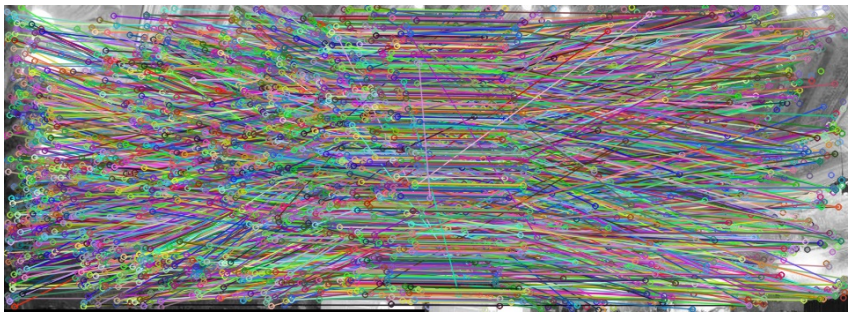

Figure 3.7: BFMatcher between sifted img 00 and sifted img 01

After a redefinition of the matches found above by selecting the matches with distance less than *ratio* * *min-distance*, where *ratio* is a user-defined threshold and *min-distance* is the minimum distance found among the matches, it's time to define the `homography` operation, where, like we said before, we use the `RANSAC` algorithm, and in the end we applied the `warpPerspective`.

```
Mat homography = findHomography(good_points_2, good_points_1,
↪  RANSAC);
Mat outwrap;
warpPerspective(img2, outwrap, homography, Point(img1.cols +
↪  img2.cols, img1.rows));
Mat half(outwrap, Rect(0, 0, img1.cols, img1.rows));
img1.copyTo(half);
```

Listing 3.4: Homography and warpPerspective code

We have reached the end of the internal for loop. A stable condition if the image obtained is the stitching of the first and second image of the row, or if it is the panoramic image that forms the first row of the original image. This condition will save the resulting image with the appropriate name and in the appropriate folder.

Vertical operations

Now we have 3 panoramic images that represent the 3 rows of the original images. As we operated before, the procedure is:

1. a `for` cycle scroll the rows.

2. If we are in the first step, the panoramic image of row 0 and the panoramic image of row 1 are loaded. If we are in the second (and final) step, the panoramic image of row 2 is loaded with the resulted image of stitching operation in the past step.

3. the procedure continues with the sifted part, calling the sift functions.

4. BFMatcher works with the resulted sifted descriptors.

5. After a redefinition of a better matches, homography and the consequently `warpPrespective` are applied.

In the end of this `for`, if we are in the first step, the resulting stitched image is saved with the appropriate name in the appropriate directory, if we are in the second (final) step, the resulting image should be the original image, and you have come to the conclusion.

4 Conclusion

The same procedure was tested with two data-sets of images: the only one difference between them was in the fact that in the first one was implemented with a gray-scale conversion of color.

4.1 A small unresolved problem

After this procedure, the resulting images are shows:

Figure 4.1: Resulting image 1 (in this case the procedure was carried out working with gray-scale images)

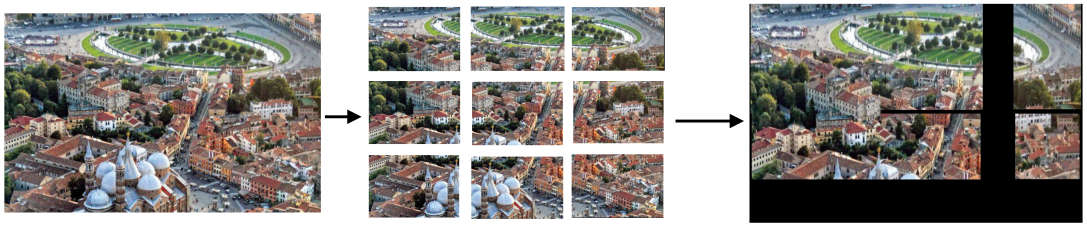

Figure 4.2: Resulting image 2

It is possible to see that the resulting work carries some imperfections, most likely dictated by the homography and warpPrespective operations, as the resulting images were provided with a lateral black space which I was not able to remove. This created a concatenation of incorrect black spaces which resulted in a resulting wrong final image.