UNIVERSITY OF PADOVA

Lab 4 Report

Subject: INP9087844 - Computer Vision
Professor: Zanuttigh Pietro
Student: Menglu Tao **Nº**: 2041389

1. Introduction

This report describes the process of creating an application using C++ and OpenCV to create a mosaic image by joining together a group of images. Since the application considers the SIFT method, to compile and run this code is necessary to have OpenCV version 4.5.2 installed. To compile the code, just run **make** in the root folder of the project and to execute:

```
./lab4 <ratio>
```

where *ratio* = 0.5 if not provided, i.e. by just running **./lab4**.

To configure the number of features to be used by the SIFT detector, or to resize the images to a common size, change the defined values at the top of the file like shown in Fig. 1.1:

```
16  #define DO_RESIZE 0
17  #define DEFAULT_RATIO 0.5 // if the user does not provide ratio
18  #define NUM_FEATURES 20000
```

Figure 1.1: Constants that can be configured.

The test directories from the *data* folder are defined in a list called *datasetNames* - Fig. 1.2.

```
31  int main(int argc, char const *argv[])
32  {
33      // srand(time(NULL));
34      if (argc > 1) {
35          matchesRatio = stod(argv[1]); // stod = String TO Double
36      }
37      string datasetNames[] = {"T1", "T2", "T3", "RT1"}; // test files
```

Figure 1.2: Dataset names list - change to test files from *data* folder.

In this work I implemented some of the optional steps from the homework description:

- 5) Work with color images instead of grayscale
- 7) Use some blending/mixing technique to better merge adjacent images
- 9) Try to automatically guess which images are linked to which

The rest of this document is organized as follows: section 2 presents the main points of the development, section 3 shows and discusses the results, section 4 ends with the conclusion and finally the main references used for this work.

2. Mosaic Image Creator - Development process

To create this application, the provided images were analyzed to build a logic that could be used for any other scenarios different from the presented ones. So, in order to create an application that receives a **random set of images** and evaluate to stitch them **regardless of the order** they were read, the following were considered:

1. Load a set of images from disk;
2. Extract SIFT features from the images;
3. Build a set of best matches;
    - for each image, compute the best matches for each other image in the set.
4. Sort the set of best matches in descending order by the number of good matches;
5. Find homography for the best matches and use it to determine the position of the images to be stitched;
6. Create a new set of images that are generated by stitching each pair of images;
7. Repeat the whole process from step 2 by using the new set of images until we have only one image in the end, i.e. the final result.

After analyzing the 3x3 images from the data provided, it was observed that a simple logic could be implemented by solving only 2 images at a time. Fig. 2.1 shows how the algorithm works implementing this idea. At each step the algorithm extracts the keypoints, descriptors and best matches for all the images and pairs of images, then tries to join all pairs of horizontal images (steps 1 and 2). It keeps trying to join all pairs of horizontal imagens until there are no more images to be joined horizontally. If there are no more images to be joined horizontally, then the algorithm tries to join all pairs of vertical images (steps 3 and 4). After that, it is expected that the remaining image is the final result.

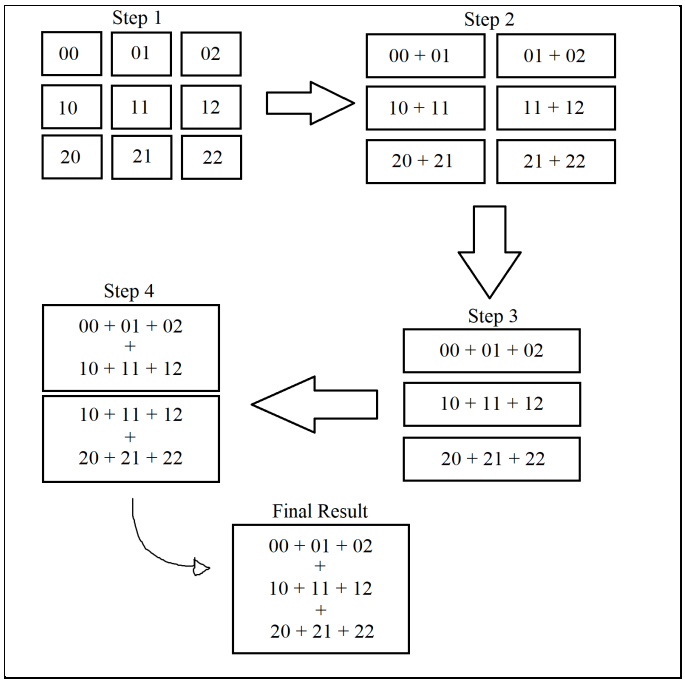

Figure 2.1: How the algorithm works for an image composed of 9 pieces.

It is important to mention that the images can come in any order - the algorithm tries to guess the images to be stitched together. To validate that, according to Fig. 2.2, it sorts the images just after reading them from disk:

```
43      for (string datasetName : datasetNames) {
44          vector<Mat> imgList = loadImages("data/" + datasetName);
45          auto rng = default_random_engine {};
46          shuffle(begin(imgList), end(imgList), rng);
```

Figure 2.2: Shuffling the images before processing them.

However, to guess the images to be stitched together is not an easy task - the method *findHomography* depends on the order of the images. In this case, if there is any negative number in its output, then an easy solution is to swap the evaluated images and run the method again. After that, it is possible to call the methods *warpPerspective* and *copyTo* to create the stitched image. The next section shows some results by running the created algorithm.

3. Results

The results below were generated for the datasets T1, T2, T3 and RT1 using *ratio* = 0.5:

Figure 3.1: Generated image for dataset T1.

Figure 3.2: Generated image for dataset T2.

Figure 3.3: Generated image for dataset T3.

Figure 3.4: Generated image for dataset RT1.

By observing Figs. 3.1, 3.2 and 3.3, it is possible to see that the implemented algorithm works very well for the datasets T1, T2 and T3. Some missing details in the pieces are also improved in the final result by blending some parts - the function *inpaint* from OpenCV was

used. From Fig. 3.4, the algorithm still works for a more complex dataset, but can be improved to have better results.

4. Conclusion

The presented work showed an application built using C++ and OpenCV, which is capable of creating a mosaic image by joining together a group of images. This application was made to be a generic solution for a random group of images in any order - it tries to guess the images to be stitched together based on the number of good matches using SIFT algorithm. It is based on the idea of joining a pair of images at a time until it has only one image in the image list, that is the final result.

This project provided a better understanding and practice of the language C++, the library OpenCV and the knowledge acquired during the classes. The algorithm showed excellent performance for datasets T1, T2 and T3, and an acceptable result for RT1, showing that it still can be improved to solve more complex scenarios.

References

[1] Features2D + Homography to find a known object. Link:
<https://docs.opencv.org/3.4/d7/dff/tutorial_feature_homography.html>
[2] Brute-Force Matching with SIFT Descriptors and Ratio Test. Link:
<https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html>
[3] SIFT in OpenCV + detect and compute keypoints and descriptors. Link:
<https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html >
[4] Image Stitching with OpenCV C++. Link:
<https://medium.com/@pavanhebli/image-stitching-with-opencv-c-3e42a626c75c >
[5] Perspective transformation. Link:
<https://theailearner.com/tag/cv2-getperspectivetransform/# >
[6] Image mosaic and panorama construction using Python and opencv. Link:
<https://pythonmana.com/2021/08/20210803083628400F.html >
[7] Reconstruct stitched image. Link:
<https://answers.opencv.org/question/73645/reconstruct-stitched-image/ >
[8]Type of mat object. Link:
<https://stackoverflow.com/questions/10167534/how-to-find-out-what-type-of-a-mat-object-is-with-mattype-in-opencv >