Final project: features matching

Vitetta Emanuele;
2082149

1 Introduction

This laboratory activity focuses on image reconstruction. In particular, several datasets are provided, each consisting of a source image from which certain areas have been removed. Additionally, a series of patches are given to complete the corrupted image. The patches are divided into two groups based on the complexity of the transformation required to restore them to their original position. The reconstruction process occurs in two phases. In the first phase, keypoints and descriptors are sought for both the image and the patches using various recognition algorithms, particularly SIFT and ORB (**bonus**). These pieces of information are then used to find matches between the image and each patch. In the second part, the goal is to overlay the patch onto the original image. To accomplish this, the affine transformation matrix needs to be found. Therefore, RANSAC is utilized, both through the standard OpenCV library and a manual implementation (**bonus**). Finally, the matrix found is used to bring each patch back to its original position. The last step consists in performing the fusion.

2 Files organization

In the submitted folder, the following files can be found:

1. *main.cpp*, *functions.cpp*, *functions.h*: these files contain the source code of the program. Inside them, you can find comments that explain step by step the various functions and the choice of different parameters.
2. A folder named *Dataset*: it contains all the files that can be used for the necessary tests on the program's functionality (notice that *original images* are supposed to have ".jpg" extension, while patches do not have this restriction).
3. Inside each of the folders in *Dataset*, there is a folder renamed *results* where the program saves the intermediate images that are generated and also the final result.

3 Program execution

When the program is executed, the relative path to the folder containing the set of images to work on is passed as the first input argument. Subsequently, the user is prompted to make a few decisions on some parameters of the program. Additionally, in order to progress from one step to another while viewing images, it is often necessary to press the a button on the keyboard.

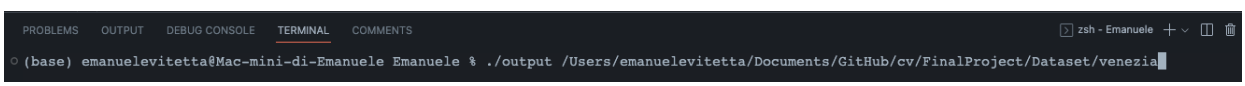


Figure 1: Example of program usage

4 Image loading

Firstly, a suitable function called `loadImage()` was implemented to load the images for analysis. This function takes the folder path and the prefix of the files to be loaded as input parameters. For simplicity, it is assumed that all the files have the same prefix followed by a different numerical value (e.g., 'patch0.jpg', 'patch1.jpg', etc.). The function supports image files with the extensions ".jpg", ".png", and ".jpeg". Additionally, the function can accept a string and a Boolean value as optional arguments, which can be used to display the loaded images with a customized title. However, the decision to activate this printing function is predetermined and cannot be controlled by the user through the terminal. It primarily serves as a debugging aid. Using the above-mentioned `loadImage()` function, the program proceeds to load the corrupted image and two different sets of patches.

5 Feature extraction

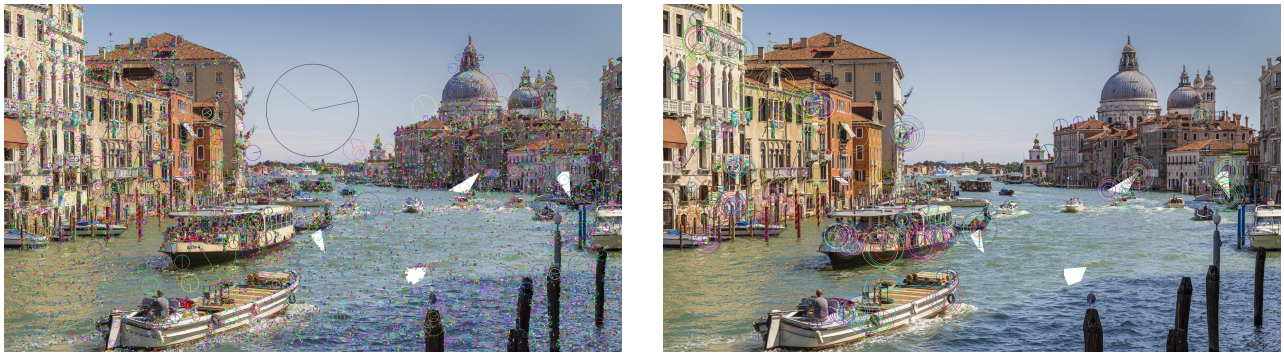
To compare the corrupted image with the patches, it is necessary to extract features from all these images. Therefore, two different algorithms are utilized, both implemented in the standard OpenCV library.

5.1 SIFT

The first algorithm considered is SIFT (*Scale-Invariant Feature Transform*). It offers excellent performance in terms of reliability and robustness. For each image, SIFT extracts keypoints and descriptors. As parameters, we can choose the number of octave layers and the maximum number of features to be extracted. In our case, this value is set to 0 (as default choice, but can be changed), meaning that the algorithm extracts all the features it finds. This choice is made because we do not have specific requirements for execution time (if real-time processing were necessary, the considerations would be different).

5.2 ORB (bonus)

During the execution of the program, it is possible to use ORB (*Oriented FAST and rotated BRIEF*) instead of SIFT. In this case the user has to choose the number of features to be extracted. To get acceptable results this value has to be really high (not smaller than ~100000).



(a) SIFT (b) ORB

Figure 2: Different feature extractions.

5.3 Code implementation

In the code, we can find the function `KeypointsFeatureExtractor()` that fulfils the before-mentioned functionalities. Specifically, it takes as input a <Feature2D> object, the image to analyze, and a Boolean value. The Boolean value determines whether to display the extracted features on the screen. We can appreciate the results of extraction in fig. 2.

6 Feature matching

This section is one of the most critical parts of the entire program. Its implementation is entrusted to two functions, namely `matchRefine()` and `flippingMatcher()`.

The first function takes as input the descriptors of the two images to be compared and some secondary parameters. First, the metric to evaluate the matches is defined. In the case of SIFT, the L2 norm is used, while in the case of ORB, the Hamming Distance is used. Preliminary filtering is performed using only this parameter. Then, two additional filtering steps are carried out. The first one is based on the distanceRatio parameter, and the second one discards all matches that have a distance between them equal to n times the minimum distance between them. Notice that n is a parameter that can be chosen by the user. By default, it is set to 3.

Before comparing the matches between the corrupted image and the patches, we want to reevaluate the patches to obtain the best possible matching. For this reason, we use the `flippingMatcher()` function. Each patch is considered in four distinct versions: its original version and three flipped versions. The original patch is replaced by the version that shows the highest number of matches with the corrupted image. This operation improves the reliability in finding matches and, most importantly, ensures that the patch undergoes the least possible transformation in the subsequent steps.

Figure 3: Example of matching between a patch and the corrupted image

7 RANSAC

In order to obtain the affine transformation that can bring the patch back to its original position, we rely on RANSAC. Below, we present the implementation using both the OpenCV library and a manual variant (**bonus**). Notice that, after finding the correct matrix for the affine transformation, the warped patch is computed, using `warpPerspective()` or `warpAffine()` (the difference between these two functions is the dimension of the matrix transformation passed as input argument).

7.1 OpenCV implementation

The function `homographySTD()` finds the homography matrix H using the OpenCV functions. Firstly, two vectors are created containing the points associated with the various matches, in the same order. Then, the necessary parameters are set to use the `findHomography()` function based on RANSAC.

7.2 Manual implementation (bonus)

The function `homographyMAN()` finds the homography matrix H using the following iterative procedure. At each iteration, three matches are randomly selected from all possible matches. Then, the matrices A and b are created as seen during the lectures, and the associated linear system is solved using least squares. At this point, all the matches are projected onto the corrupted image, and the distance between each projection and its corresponding match is evaluated. The count of how many of these distances are less than a certain threshold is calculated. In each iteration, the homography matrix found in this way is updated if more matches have distances close to their respective projections.

7.3 User refinement

As known from the theory, to obtain a valid homography matrix, three matches are sufficient. If there are not enough matches, the algorithm displays a warning and the patch is not overlaid on the image. Instead, the next patch is considered. However, it is possible to increase this threshold so that the algorithm ignores patches with too few matches (if they are too scattered, the final result may be distorted). The user can decide this threshold. For example, when using ORB, in some cases, the matches are not reliable, and therefore, some patches are ignored.

8 Image reconstruction

The last part of the code concerns the merging of transformed patches with the corrupted image in order to fill in the empty spaces. This task is performed by the function `mergeNoShape()`. The input patch is first filtered to remove all black pixels. Then, two masks are applied to crop the patch edges and create a mask that ensures a smoother transition between the patch and the image, eliminating any artefacts or black borders. Next, both the image and the patch are split into three colour channels. The patch is scaled according to the mask, and the respective colour channels are merged together. The final result is the image with the patch added to it.

It is possible that, when observing the areas of the reconstructed image where the patches have been applied, they appear slightly blurred. This is due to the fact that when the patches are scaled and transformed, there can be a loss of quality caused by the interpolation that occurs after the inverse affine transformation.

9 SIFT results

This section aims to examine the behaviour of the program, distinguishing between basic patches and more complex ones. To analyze the reconstructed image using all the patches and compare it with its original version, the `showImageDifferences()` function is utilized. This function generates a third image that highlights in white the differing pixels between the original and reconstructed versions. The fewer white areas there are, the better the final output silhouette obtained.

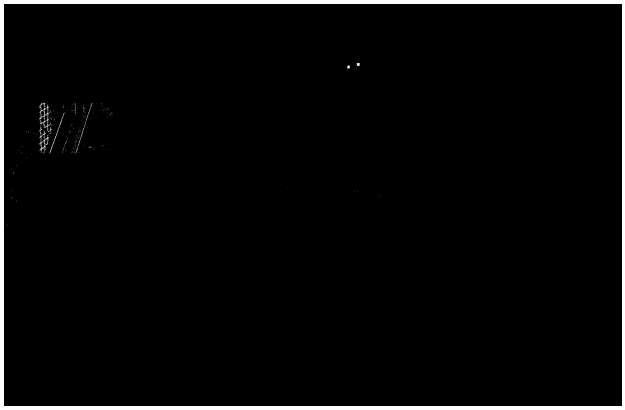
9.1 RANSAC

By analyzing some comparisons between the two implementations of RANSAC, it can be observed that there are no differences in the final result between the manual RANSAC and the one provided by OpenCV.

9.2 Base patches

Firstly, the results obtained using the basic patches are taken into consideration. In Figures 4 and 5, the final reconstruction result and the image showing the differences from the original image can be observed. As can be easily noticed, the image appears slightly more blurred in the patch areas, although this effect is somewhat difficult to discern.

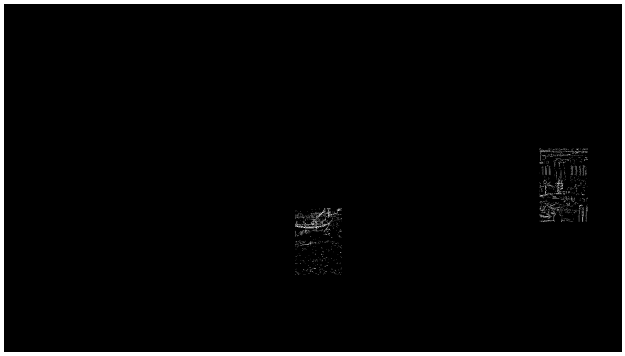
(a) Final reconstructed image



(b) Differences with original image

Figure 4: Scrovegni - filled with easy patches

(a) Final reconstructed image



(b) Differences with original image

Figure 5: Venezia - filled with easy patches

9.3 Hard patches

In this case, unlike the previous one, it is possible to notice that the difference in the area where the patch is repositioned is greater. In fact, to obtain the affine patches, certain transformations (such as resizing) were applied, making the inverse transformation difficult and sometimes irreversible.

(a) Final reconstructed image

(b) Differences with original image

Figure 6: Scrovegni - filled with hard patches

(a) Final reconstructed image

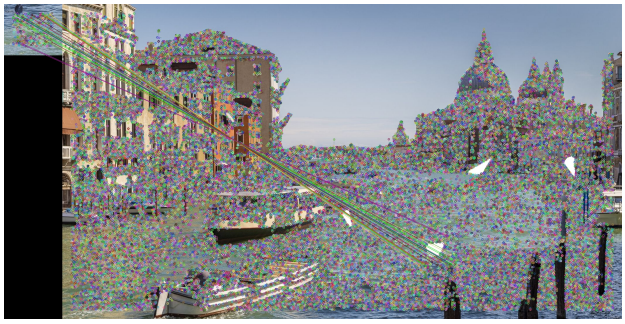
(b) Differences with original image

Figure 7: Venezia - filled with hard patches

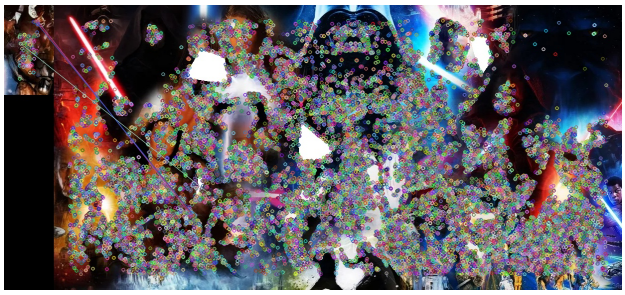
10 ORB results (bonus)

As observed from the upcoming figures, it is evident that ORB is less robust compared to SIFT. Particularly, in order to achieve acceptable results with base patches, a large number of features need to be extracted (in this example, the value is set to 100,000). Fig. 8 illustrates an example of matching with base patches in two different datasets, while fig. 9 demonstrates a challenging scenario with hard patches.

It is noticeable that ORB performs well only in certain cases and lacks robustness when confronted with more significant transformations. After conducting several tests, we can conclude that the code works perfectly with the "*Venezia*" dataset and base patches. However, in other cases, there may be issues, the most notable being the lack of good matches between the source image and the patches.



(a) Venezia, good results.



(b) Starwars, few matches.

Figure 8: Matches with ORB descriptors and base patches.

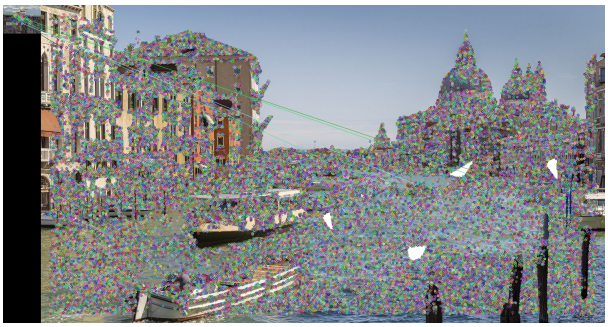


Figure 9: Matches with ORB descriptors and a hard patch.

11 Mixed images reconstruction (bonus)

In the code, there is also the capability to store multiple distinct images and multiple patches in the same folder. The program associates each image with its corresponding patches using the `showImageDifferences()` function. This function takes as input the images and all the patches. For each patch, it evaluates the matches with the considered image and calculates how scattered these matches are. If these matches meet a certain condition (a threshold that can be manually adjusted), then the patch is associated with that image. This process is iterated for all corrupted images. Once the associations have been made, the reconstruction process is performed. Please note that this section can be executed using both SIFT and ORB. Anyway, in order to get acceptable results, it is highly recommended to use SIFT (more robust).

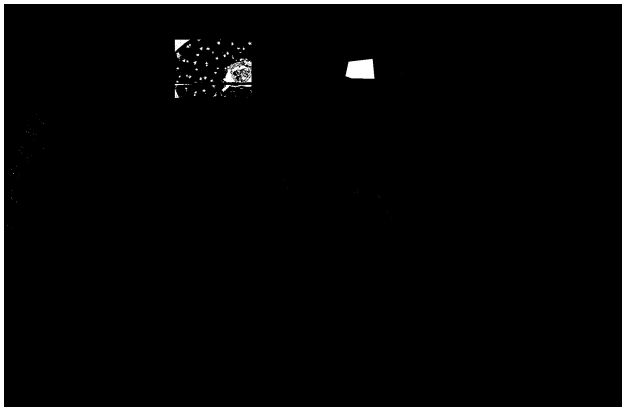
We tested the program using the "*multiple*" folder contained in "*Dataset*" containing images and easy patches both from "*pratodellavalle*" and from "*venezia*". The final images were perfectly reconstructed (but here they are not reported). The only point that may cause some errors is if a patch is assigned to the wrong image.

12 Template Matching (bonus)

Template matching is a technique used to locate and identify specific objects or patterns within an image. It involves comparing a small template image with sub-regions of a larger target image to find regions that closely match the template. The matching process typically calculates similarity or dissimilarity measures between the template and each candidate region in the target image, aiming to identify the best matching location or locations.

In the code, template matching is implemented by the function `templateMatching()`, which takes a set of patches and the corrupted image as input. We tested this function with different datasets, using only the base patches. The results are excellent for "*Venezia*" and "*Starwars*", but there are some issues with "*Scrovegni*". In fig. 10, we can see that one patch is not correctly placed. This problem may arise because the area from which the patch is derived is very similar to the surrounding area. Additionally, the presence of similarities in the bar regions contributes to the incorrect placement.

(a) Final reconstructed image

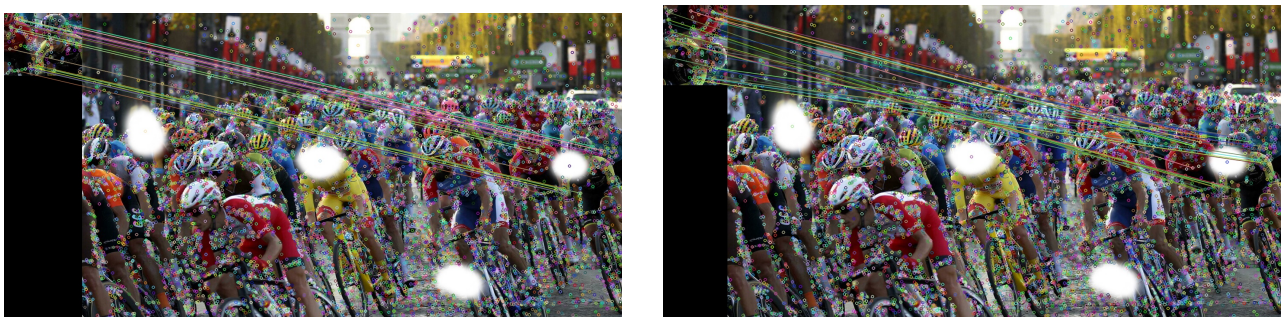


(b) Differences with original image

Figure 10: Template matching in the worst case.

13 Personal dataset (bonus)

Inside the *Dataset* folder is it also possible to find another folder called *bonus*, in which there are some extra files that can be used to test the algorithm. We can observe that the program gives perfect results both with base and affine patches. In fig. 11 we report some match examples with such patches (using SIFT)

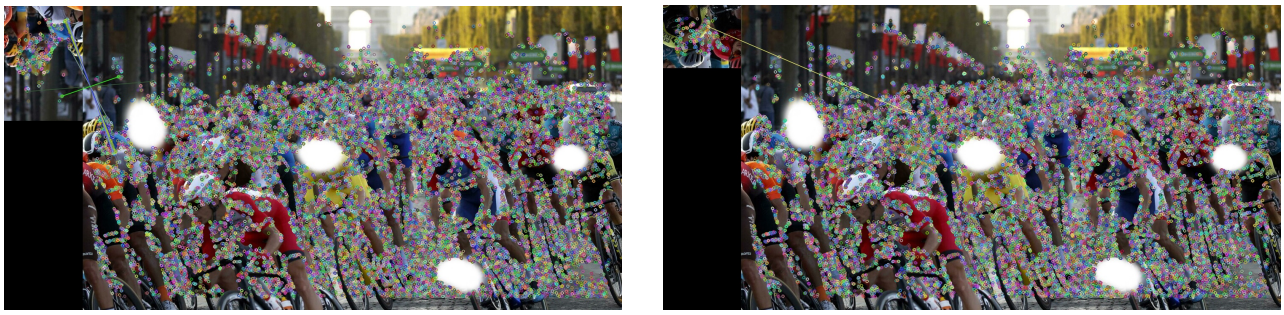


(a) Base patch

(b) Hard patch

Figure 11: Matches between the corrupted image the same patch in base and affine version.

Results using ORB are good too. In order to obtain acceptable results it is only necessary to extract a very high number of features. When reconstructing the final image the algorithm is not able to find enough matches only for one patch. In fig. 12 we can see some results



(a) Enough matches

(b) Not enough matches for reconstruction

Figure 12: Matches between the corrupted image some patches.

Finally is it possible to notice that Template Matching works perfectly and reconstructs the image with zero errors, as shown in fig. 13.

Figure 13: Template matching reconstruction