REPORT LABORATORY 4 COMPUTER VISION

Davide Peron 2082148

Academic Year 2022/2023

1 Introduction

The main goal of this laboratory experience was to complete some images; I was provided with some dataset containing an image with some parts that were missing and the patches that correspond to the missing part; obviously the patches were bigger than the missing part to allow feature detection outside of the white areas and subsequently place the patch on top of the corrupted image to cover the missing spot. I was provided with two different sets of patches, the first one consisting of parts of the original image, without the missing spot, that were not modified whatsoever, in the following they will be referred as **basic patches**. The second set instead consisted on the same patches as before but with some transformations e.g. rotations, flippings, resizing, added noise etc; they will be referred as **transformed patches**. As previously mentioned the main tasks is to reconstruct the original image, this task can be divided into four main parts:

1. Feature detection;
2. Feature matching;
3. Finding homography / affine transformation;
4. Image reconstruction;

These four parts will be explained in depth in their respective sections with all the design choices made.
Other than the main assignments I decided to implement some extra points that don't directly fall under the previous tasks subdivision, in fact I decided to implement template matching, only for the basic patches, and also a mode that takes as input multiple corrupted images and patches that corresponds to those images or not and the program is required to identify which patch belongs to which image and then reconstruct them; the procedure will be explained in depth in a dedicated section.
Lastly I developed a user interface (UI) to make it easy to chose the mode we want to use and give appropriate values to the necessary inputs.

2 How to Use

In this Section I will explain how to use the program developed and the specific requirements of the folders and the images to make everything work. The program has been developed using Windows as OS, and it also includes compatible functions with other OS, but has never been tested with them, so I cannot guarantee that everything will work, although it should. Another important thing to note is that since I used the *filesystem* library the program need to be compiled with a compiler that supports C++ 17 or higher standards.

2.1 User Interface

In this paragraph I will briefly explain how the user interface works in choosing the desired mode to run the program, in Figure 1 it is possible to see a flow chart representing the possible modes that has been included in the program.
After compiling and running the executable we will be prompted with some questions that have only two possible answers the user will be required to answer 1 or 2 depending on the desired response; if the input is not valid meaning it is something other than 1 o 2 the program will say that the input is not valid and the user will be required to insert a new input until a valid one is given. It will not be possible to come back to a question previously answered without closing the program and re-running it.
After a mode has been chosen the user will see a confirmation of the selected mode and will be asked to insert the specific inputs required for that specific mode e.g. path of the images, thresholds, whether to save the results or not etc., these inputs have some requirements, too, that will be specified in the following paragraph and if those are not satisfied the user will be required to insert a new value until a valid one is given.
After all the inputs have been inserted the main part of the code will run and it will behave differently based on the selected mode.

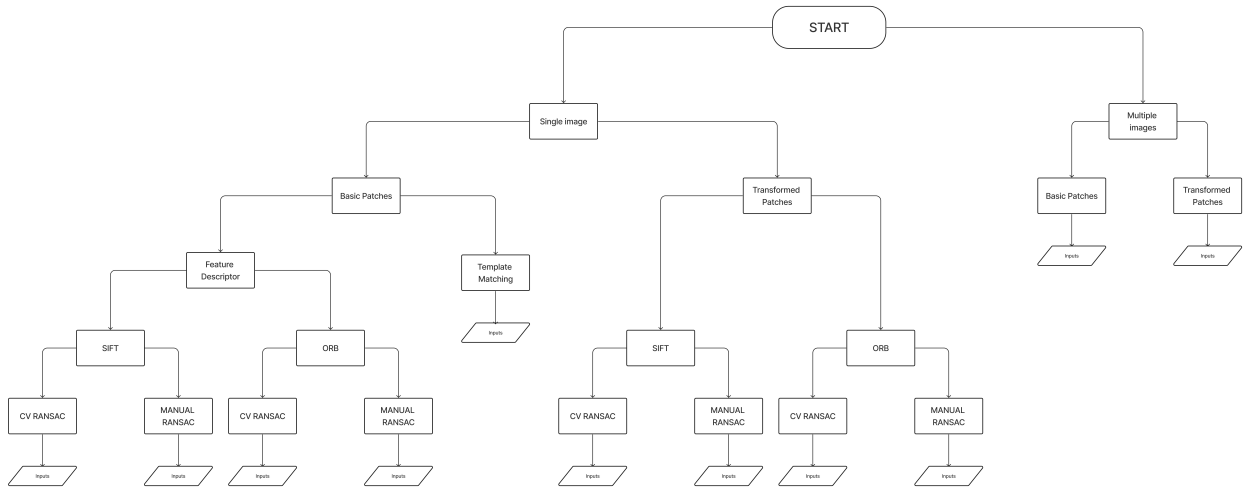


Figure 1: Flow chart representing all the possible modes of use

2.2 Folder and Input Requirements

While running the program we need to have the folder that contains the images of a specific dataset that respects some requirements that are different from the base datasets given in the assignment.

Single image mode: while running signle image mode the folder containing the dataset is similar to the one given in the assignment i.e. the corrupted image and the patches must be named as the one provided and the extension must be *.jpg* and the patches must have a progressive number starting from zero (when the first number in the sequence is missing the functions will stop loading images), although there is not a limit on the number of patches that a dataset can contain.
The only difference is that the folder must contain the original image namely the image provided without the black spaces of the corrupted image named *original_image.jpg*, this image is only used to show the differences between it and the final result namely the reconstructed image.

Multiple image mode: in this second case the name structure of the patches is the same as in the single image case instead the corrupted image and the original image must be named, respectively, *image_to_complete_**i**.jpg* and *original_image_**i**.jpg* with **i**=0,1,... up to the number of images we want to use. In the provided zip folder there is already a folder called *multiple* that contains some images already formatted as required.
As far as the inputs concern, based on the mode the user is required to insert:

- path : the path of the folder containing the dataset the user desires to use, it must be a valid directory.
- octave_layers: used only with SIFT and it specifies the number of octave layers the user decided to use, I decided to include this parameter since some transformed patches did not produce satisfying results with the default value. It must be an integer greater than zero.
- distance_ratio: it is used for both SIFT and ORB to refine the matches found. it will be described more in depth in the next sections. It must be a double greater than 1.
- match_distance_threshold and sparsity_ratio_threshold: these two variables are used only in multiple image mode and are used to match the image to the patches the first one is a double greater than 1 and the second one is also a double but in the interval$[0, 1]$.
- draw_matches: it is a boolean, used only while using a feature detector, that indicates if the user desires to see the matches between the corrupted image and the patch during the execution.
- save_results: it is a boolean that indicates if the user wants to save the final results and also the matches, if available.

2.3 What is included in the zip folder

In the zip folder submitted we can find a folder called *dataset* that contains the test images provided with the modifications required to the images as described in the previous paragraph, inside each dataset there is a folder called *Results* which contains some results of the test done on the images provided.
Finally there are the source code files respectively:

- **Peron_Davide_Project.cpp** which contains the main that consists in the UI and the calls to the main parts of the program.
- **main_parts.cpp** and the respective header that contains the three functions used for running the program, respectively single image mode, multiple image mode and template matching.
- **Functions.cpp** and the respective header that contains all the basic functions used throughout the program.

3 Feature Detection

In this section we are going to see the approach used for feature detection, specifically I decided to implement both SIFT and ORB; as we will see in the results section they provided good results with the basic patches but when using the transformed one, as expected SIFT had better performances due to its robustness although it being a bit slower than ORB.

3.1 SIFT

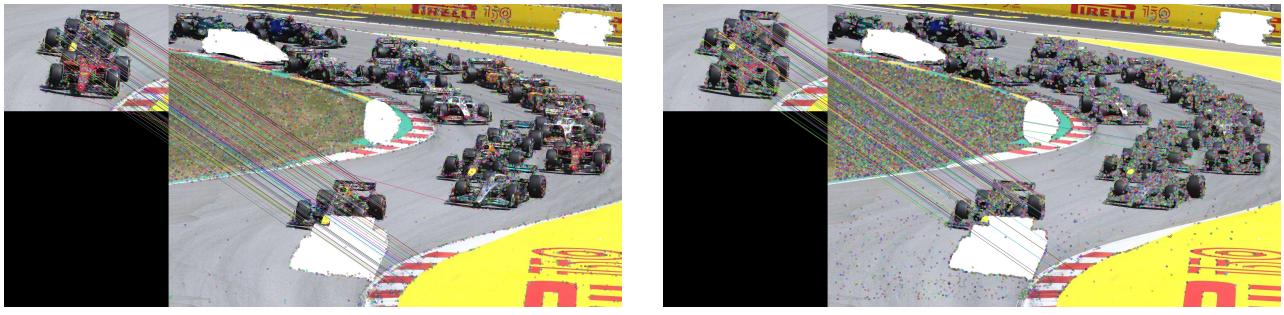
As already said SIFT provided excellent results with both basic and transformed patches, this algorithm is implemented in the *OpenCV* library and between the possible parameters to set I decided to ask the user for the number of octave layers to use for the specific set of images since some of them required a different number of octave layers to obtain a satisfying results, especially with the transformed patches; another design choice I made was to to consider 0 as the number of feature parameter that the algorithm finds, which means that there is no limit on them, therefore running the algorithm will be more computationally expensive but its robustness increases a lot.

3.2 ORB

Another feature descriptor I decided to implement was ORB, it provided similar results to SIFT while considering the basic patches but when considering the transformed patches it provided really bad results, this is probably due the transformations that made more difficult to find the matches. A possible solution could have been to play around with the initialization parameters of ORB but I decided not to do that, and as a matter of fact I decided to set only the maximum number of feature returned by the algorithm, following a similar line of thought to what I have done with SIFT I decided to set that value to one million since there is no direct way to not set a limit and with a lower value we would obviously get a faster algorithm but way worse results. As we can see in Figure 2 setting such a high value will amount to a lot more useless features than the one found using SIFT.

3.3 Code Implementation

Both feature descriptor were used in a really basic way just by creating the feature descriptor and running the *detectAndCompute* function provided, this function returns the key-points and descriptors of the image we are analyzing that will be then used in the feature matching part. These part has been implemented though a function I created called *Detect_Compute* that takes as input a Flag (the list of all the flags used can be found at the beginning of *Functions.h* file) I created to indicate if we want to use SIFT or ORB.



(a) Matches using SIFT (b) Matches using ORB

Figure 2: Difference of the matches found by SIFT and ORB using the F1 dataset

4 Feature Matching

After obtaining all the key-points and descriptor from both the corrupted image and the patches we need to find the matches between the patches and the corrupted image, to do that I used the brute force matcher provided in the *OpenCV* library where the only parameter specified was the norm used for the matching which is the $L2$ norm for SIFT and the hemming norm for ORB.

As a first try I used the function *match* which provides all the matches between the patch and the corrupted image, then I filtered those matching using the $distance\_ratio$ input which specifies a ratio between the distance of the match we are considering and the minimum distance of all the matches; this approach gave good result with an accurate tuning of the $distance\_ratio$ for each set of images, but I it still had some problems with some images. To simplify this process and obtain better results I decided to implement an approach we saw in class which consisted in using the function *knnMatch* that returns the 2 best matches for each descriptor, in increasing order of distance, and then keeping only the matches where the ratio between the distance of the first element and the one of the second is lower than 0.8 as suggested in slide 47 of the SIFT notes; then I still did the same filtering as before and found that the value of $distance\_ratio$ was more constant than before even to the point that I could use for all the images the same value namely $distance\_ratio = 3$, I still maintained the user input as requested in the assignment but using above mentioned value worked very well for all the images.

4.1 Best Flipping

Since some of the transformed patches presented some sort of flipping, e.g. the central patch in *pratodellavalle* dataset, and the matcher was not able to find enough or adequate matches I decided to implement a function called *Best_Flipping* that takes as input a patch and the descriptor of the corrupted image and then tries to find matches between the descriptors and all the possible flippings of the patch, namely horizontal, vertical, and diagonal, and gives as output the patch flipped with the highest number of matches, supposing that this is the correct flipping of that patch. This function was able to solve all the problems I used to have with some transformed patches, but obviously it made the code much more computationally expensive, because of this this method is applied only if the patches the user decides to use are transformed ones. In Figure 3 we can see the difference between the same transformed patch with and without using the function described; it is interesting to note that in Figure 3c the matches found for the patch are few and very far apart from each other.

(a) Results best flipping

(b) Results without best flipping

(c) Matches without using best flipping

Figure 3: Application of best flipping to pratodellavalle dataset

5 Finding homography / affine transformation

The third step is to find the transformation matrix that allows us to position and transform the patch to fill the blank space in the image. To do that I decided to implement both the library function that returns the homography matrix and the manual version that instead returns the affine transformation matrix since all the patches are affine transformed. Both approaches used the RANSAC algorithm.

5.1 CV homography

This first approach can be found in the function *find_homography_CV* and it simply uses the library function *findHomography* that uses the points where the matches were found to calculate the homography matrix (3×3) using the RANSAC algorithm. Then to transform the patch I use the *warpPerspective* function since I obtained a homography matrix. Lastly it is important to notice that the homography or affine transformation cannot be found if there are less than 4 matches, to avoid the program from crashing I added a control where if we don't have enough matches error message will appear in the command windows and the program will skip to the next patch, without adding the one that gave poor results.

5.2 Manual affine transformation

The second approach, can be found in the function *find_affine_manual* where I developed the procedure explained in the LAB 4 slides, the result of this procedure is an affine transformation matrix (2×3) which implies that *warpPerspective* is not a valid function since the dimensions of the matrix are not the required ones, a possible solution was adding as last row of the matrix obtained the following row $[0, 0, 1]$ but instead I decided to use the *warpAffine* function that takes as input exactly an affine transformation matrix.

5.3 Differences between the two methods

There were no major differences between the results provided by the two approaches, the two methods had similar computational times and similar results. Some differences, although nothing that the naked eye could perceive, were present, as it is possible to see in Figure 4 that depicts the difference between the original image and the reconstructed one for both the CV and manual approach using SIFT method for basic patches on Venezia dataset, a black image means no difference between original image and the reconstructed one.

(a) CV Homography (b) Manual affine transform

Figure 4: differences between original image in Venezia dataset, basic patches

6 Image Reconstruction

This last task is implemented in the function *merge_filter* which takes as input the image we need to patch and the modified patch obtained as described previously then creates a mask that is the same size as the patch, containing only the points that are not black; it converts the mask to grayscale and erodes it, to remove any noise and black borders. Subsequently, it resizes the mask to match the size of the image then splits both the image and patch into their color channels; finally it blends the patch onto the image using the eroded mask for each color channel and lastly it merges the blended color channels back into the final

image.

7 Additional Parts

It this section I will explain how I implemented some extra features that do not directly fall into the scheme described previously.

7.1 Template Matching

To implement the template matching technique to reconstruct the images I took heavy inspiration from the examples present in *OpenCV*'s documentation; this tecnicque is only used for basic patches since it relies on passing the patch over the whole corrupted image and then placing it where the algorithm finds more matches; so I was not able to make it work with transformed patches. Probably using more complex machine learning algorithms it may be possible to make this type of technique work even with transformed patches, but I decided not to do that and limit my implementation to the basic ones.

7.2 Multiple Images Recognition

As explained in the introduction in this mode, the program takes as input one or more images to patch and some patches (which may or may not be part of the images) and then assigns to each image a subset of patches that the algorithm thinks are part of that image. This matching part is the core of this mode since after having found the subset of patches for each image the program does the same tasks seen previously to each image and the respective subset of patches.
The matching algorithm can be seen in the function *multiple_image_match* and computes the matches between the images and the patches. It then decides, through a sort of cost function based on the sparsity of the matches, whether or not to assign that patch to the image. The parameters of the cost function are user defined since the algorithm can work on both basic and transformed patches but needs to be tuned to obtain really good results. In the Results section, I will provide some value that gave really good results on the dataset I used.

8 Results

In this last section, it will be possible to see some main results obtained with the developed program; I will highlight both the good aspects and the consistency of the algorithms used while also pointing out where they may fail and the possible cause for that. Since there are no major differences when using the version of RANSAC implemented manually of by the *OpenCV* library all the following results will be shown using the library function, since it has slightly more robust results, as shown in Figure 4.

8.1 SIFT

In this paragraph it will be possible to see some results regarding the use of the SIFT algorithm, if not specified otherwise I considered the following parameters:

- octave_layers: 5;
- distance_ratio: 3;

8.1.1 Basic Patches

Considering the basic patches every dataset used provided visually perfect results, as saw before, in Figure 4 some difference could be present, when evaluating the absolute difference between the original image and the patched one, but nothing really noticeable.

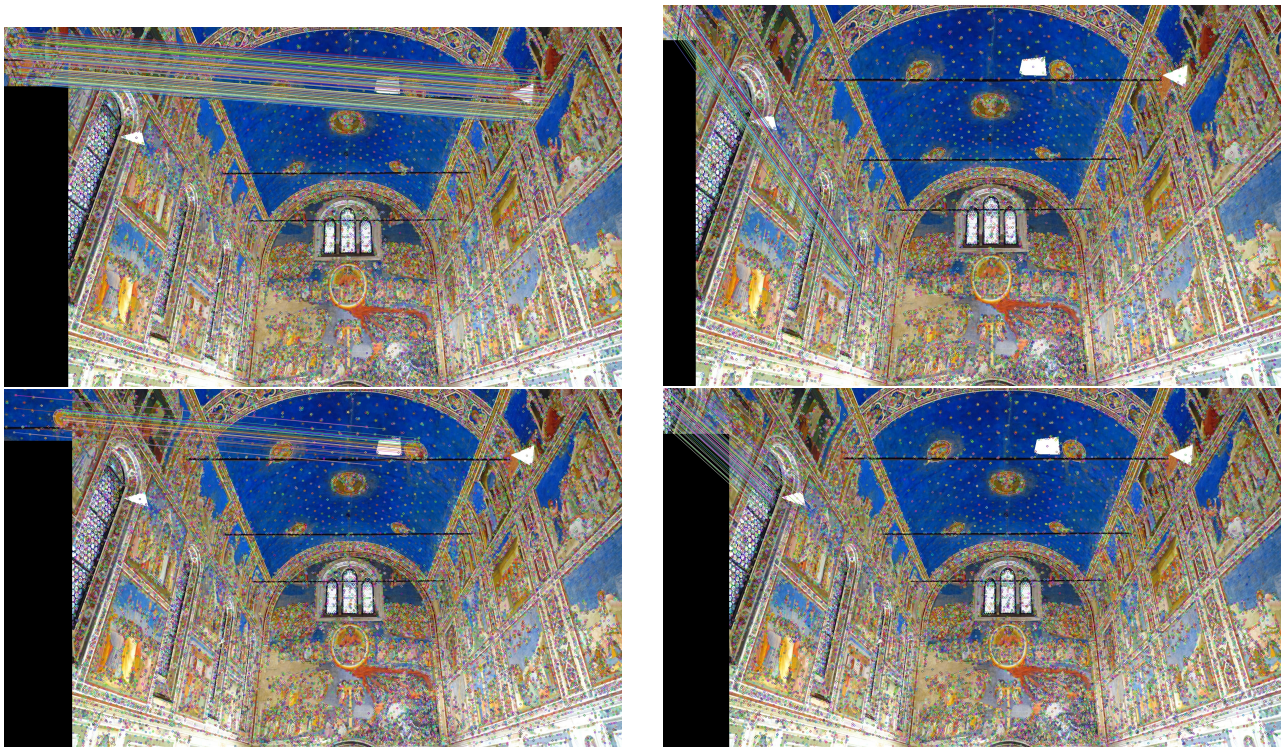


Figure 5: Matches on scrovegni dataset, SIFT

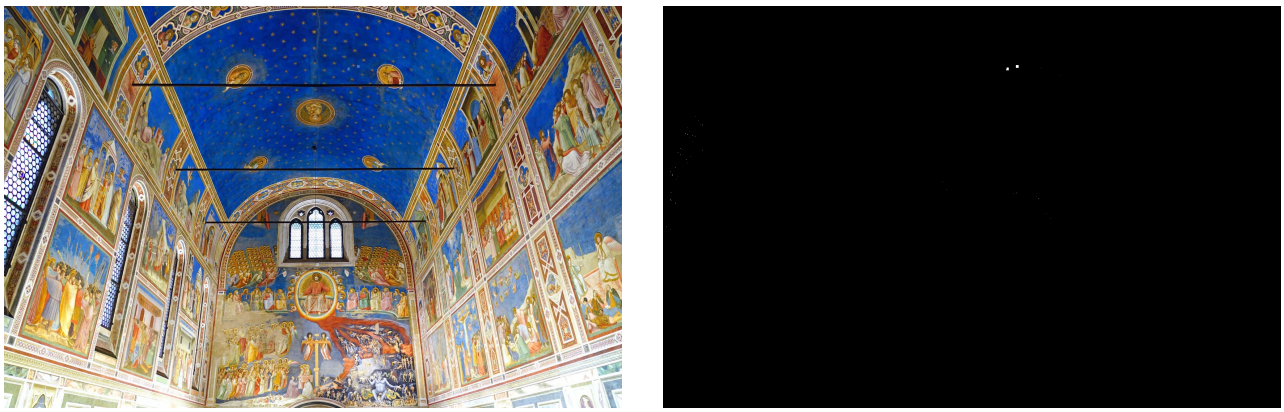


Figure 6: Results on scrovegni dataset, SIFT

8.1.2 Transformed Patches

Using the transformed patches, as expected, proved to be more difficult, therefore I needed to add some extra steps to obtain good results with all datasets; the main problem regarded the flipped patches that, as said before, was solved introducing the *Best_Flipping* function whose effects can be seen in Figure 3. The other big problem regarding the transformed patches was due to the dimensions of the patches itself, which were too small to find enough matches, this is clearly visible in Figure 7; the problem can be solved using a higher number of octave layers for the sift algorithm so when evaluating the patches feature we

go more in depth in the image despite its reduced dimensions. This problem presented itself only with the international dataset and to solve it I used $octave\_layers = 20$, obviously this makes everything more computationally expensive. The last thing it is important to point out is that we don't obtain visually perfect results, since some of the patches has added noise and changes in equalization that I decided not to remove, so some differences between the original image and the patched one are expected.

Figure 7: Results on transformed international dataset, SIFT $octave\_layers = 5$

Figure 8: Results on transformed international dataset, SIFT $octave\_layers = 5$

8.2 ORB

In this paragraph it will be possible to see some results regarding the use of the ORB feature descriptor; I considered the following parameter:

- distance_ratio: 3;

Before showing the results it is important to point out that as expected form theory the ORB descriptor is far less robust than SIFT but at the same time is much faster, and as we will see for easy images it provides the same results, so it is a good alternative, especially for real time applications but it doesn't guarantee the same robustness of SIFT and depending on the specific use-case ORB can be a valid alternative.

8.2.1 Basic Patches

While using the basic patches the results obtained were flawless or almost flawless with all the datasets, as we can see from Figure 10; the only exception is the international datasets that gave the same problem it gave with the transformed patches in SIFT, Fiugre 7, since the results are similar, so I won't add that result. Looking at the patches that produced bad results, we can conclude, that the main problem is the dimension of the patch itself, indeed a really small patch is really difficult to analyze for ORB without the proper initialization of all the parameters, which was outside of the scope of the experience.

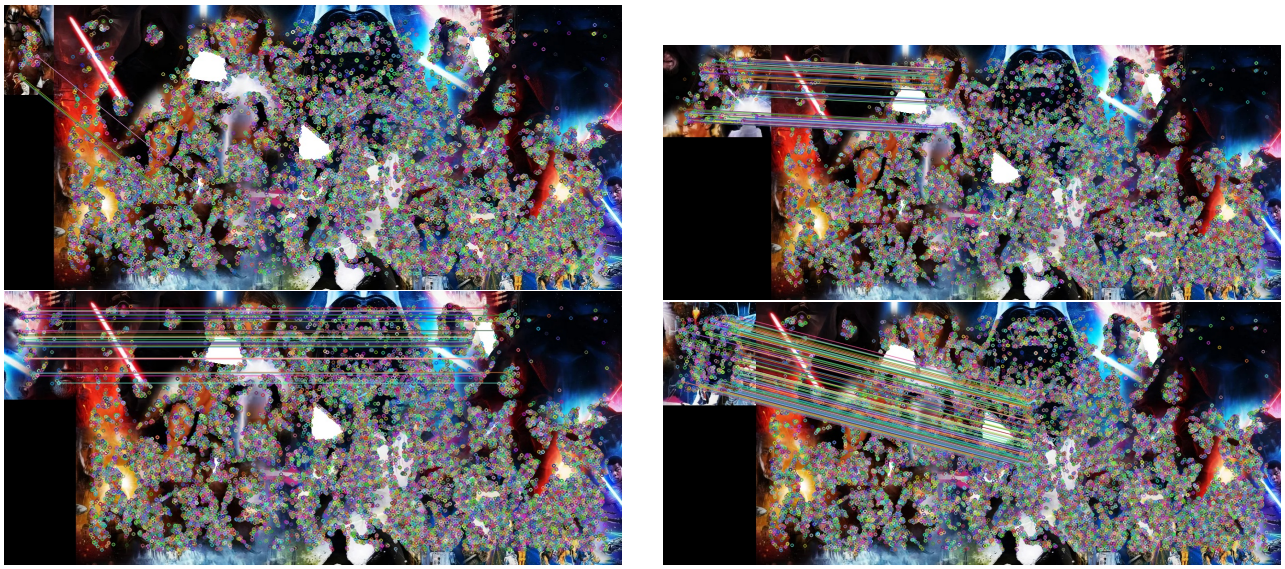


Figure 9: Matches on starwars dataset, ORB

Figure 10: Results on venezia dataset, transformed patches, ORB

8.2.2 Transformed Patches

As previously said, ORB is much less robust than SIFT and this is clearly visible with the transformed patches that do not produce any meaningful result, meaning that none of the dataset worked; as its possible to see in Figure 11 the general results were not very good since not enough matches were found or the one found were not good.

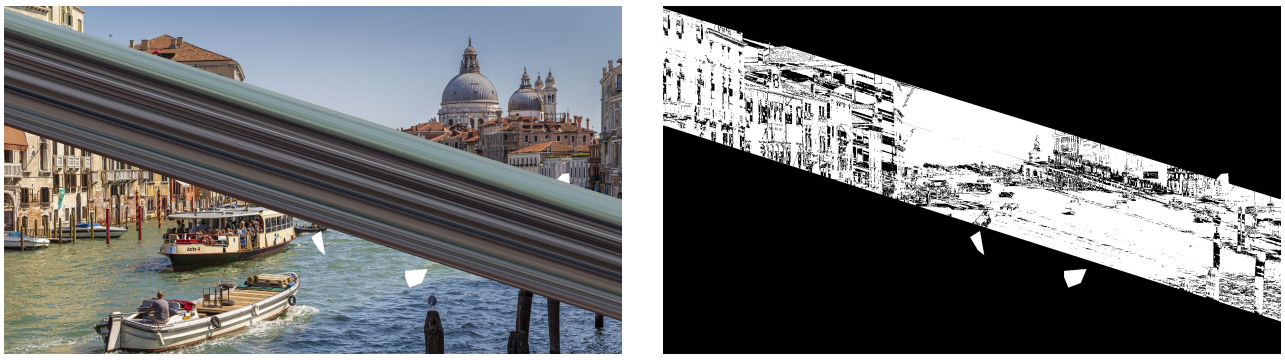


Figure 11: Results on venzia dataset, ORB

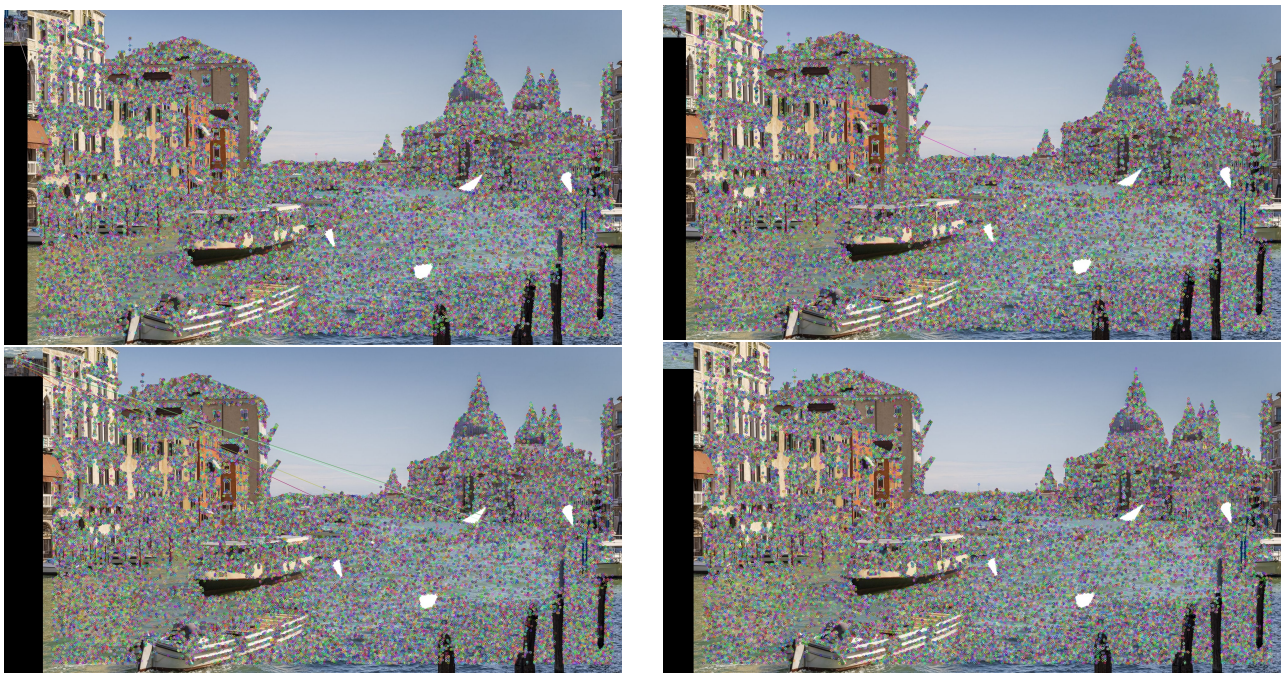


Figure 12: Matches on venezia dataset, ORB

8.3 Template Matching

In this paragraph I will analyze the result obtained while using the template matching algorithm; as previously said it is a very simple algorithm that relies on passing the patch through all the image and placing it where it finds most matches, due to its simplicity it is not very robust, as we can see from the results in Figure 14, but the mistake can be easy in this case since both the patch and the position where it was placed are very similar, indeed in both there is a Red Bull behind another car and so this mistake is comprehensible; but we can see that when a patch really stands out, as it's the case with the other three patches they are placed in the correct position.

Figure 13: Original image and differences with patched one using Template Matching on F1 dataset

Figure 14: Patches progression using Template Matching on F1 dataset

8.4 Multiple Images

In this last section it will be possible to see how the program will behave when working in multiple image mode, the most important thing about this mode is finding the right threshold to correctly classify each patch only to the corresponding image; the threshold used for the *multiple* dataset can be found in the following table:

Patches Type	Match Distance	Sparsity Ratio
Basic	40	0.3
Transformed	130	0.3

Table 1: Parameters for multiple image recognition

The Final results with both basic and transformed patches are flawless as its is possible to see in Figures 15 and 16; the main problem of this mode, as previously said, is the computational time required to associate the patches to the images.

Figure 15: Results of multiple image recognition on basic patches

Figure 16: Results of multiple image recognition on transformed patches