1 Introduction

This project is about the development of an algorithm that creates panoramic views from sequential images. We are going to delve in computer vision techniques such as keypoints extraction, matching, and image alignment, the algorithm aims to merge images and overcome challenges in scale, rotation, and lighting variations. Then we are going to see a few insights found during this project. And finally we are going to state the algorithm constraints and conclusions.

It is important to mention that while the language used in the report uses plural pronouns for a more formal or inclusive tone, it's important to note that the entirety of the project, including research, development, and documentation, was made individually.

2 Algorithm Design

The approach taken in this project follows the strategy seen in the theory of this course. Image stitching is build upon three steps: (**1**) keypoints extraction, (**2**) keypoint matching and (**3**) image alignment. So, lets answer the first question based in this three steps. The design was inspired by [4].

2.1 Which techniques are used to solve this problem?

1. **Keypoint extraction**: Local descriptors in computer vision are compact representations of key features within a small region of an image, capturing distinctive patterns or textures. And this can enable us to create feature matching between two images. The descriptors used in this solution are SIFT descriptors. SIFT descriptors encode gradient information in a spatial neighborhood [6]. Thanks to its nature are invariant to scale, rotation, illumination and rotation. Property that enables us to use them to create panoramic views over images that are slightly rotated or with illumination changes. A very helpful characteristic in this problem.
2. **Feature matching**: Feature matching is the process of finding correspondences between distinctive points by establishing relationships between key features [5]. Often is based on similarity or distance metrics. In this case we are going to use a brute-force approach, by calculating the distance from each descriptor of an image to each one of all the descriptors of another image, and for the minimum distance pair of descriptors, a match will be created. In this way the time complexity of this approach is $O(m \cdot n)$ where m are the descriptors found in one image and n the descriptors of the other one.
3. **Image alignment**: Image alignment is the process of adjusting the position and orientation of one image to match another. This alignment will be done by computing an homography matrix, that in the computer vision field defines the relationship of two images of the same scene, applying geometric corrections such as rotation, translation and perspective distortion all in one matrix [3],[2]. In order to change the image perspective we are going to Warp the perspective of the image according to the homography matrix. This warping will adjust the coordinates of each pixel to achieve the desired viewpoint in which the two images, when we join them together, appear like a single panoramic image.

2.2 Which constraints has this approach?

The approach explained in the previous section has some restrictions. Firstly, the success of the method relies on the effectiveness of SIFT descriptors for keypoint extraction, and it may not perform well in **dynamic scenarios**, in which there are objects in movement, because from one sequential image to another it will be changes that will affect the correct keypoint matching. Additionally, the brute-force approach for feature matching may be **computationally expensive**, making it less suitable for real-time applications or large image resolutions where plenty of smartphones nowadays makes use of, like 2K or 4K images. Finally, the accuracy of image alignment using homography depends on the assumption that the scene undergoes planar transformations, limiting the applicability in non-planar or complex 3D scenes, for instance if the user makes photos **changing dramatically of position and orientation**.

2.3 Which alternative techniques could be used?

Probably there are plenty of options within and outside this approach. Lets mention some of them:

- **Keypoint extraction**: Instead of using SIFT descriptors, we can consider other local descriptors such as SURF (Speeded Up Robust Features) or ORB (Oriented FAST and Rotated BRIEF). These may provide a balance between speed and accuracy in different scenarios.

- **Feature matching**: Maybe we could think of a more clever way to compute the matches between descriptors and this will lead us to a better performance algorithm. There are some efficient nearest neighbours search algorithms as FLANN [7].
- **Image alignment**: We could explore deep learning-based methods for keypoint matching and image alignment, such as in [9]. These approaches may offer improved performance in certain contexts, although they may require big computational resources, training data and knowledge on the field.

3 Algorithm Implementation

3.1 Which technologies are used in the algorithm implementation? Show the algorithm implementation function

I will answer both questions while explaining the implementation of the algorithm through this section. The code is written in Python, the computer vision functions are from OpenCV library and the multi-dimensional array managing is from NumPy library.

3.2 Algorithm of Pair Image stitching

First we developed an algorithm that implements the design mentioned in the previous section for only two images. We can divide the algorithm structure in the following 8 steps:

- **Convert to grayscale**: converting images to grayscale simplifies the image stitching process by reducing computational complexity, as matching and aligning features based on intensity values is enough for offering a good result. Grayscale representation can also enhance the robustness of feature extraction, making the algorithm less sensitive to variations in color across images. Fact that will be useful because with sequential images taken with a phone often have changes in colors. We have used a OpenCV function for changing the color space.
- **Resize images**: as we have commented before the brute-force approach is computationally expensive, and with large images the algorithm will take even minutes to finish. To solve this problem we established a resizing criteria before searching descriptors. The user can offer by parameter a factor of scaling, but by default we set the higher dimension of the image to 1000 pixels and then we resize the other dimension conserving the same aspect ratio. We assume that we don't need 2K and 4K to find good descriptors. We have used the OpenCV function for resizing images.
- **Keypoint region selection**: since we have to join more than 2 images to create the panoramic view not all the keypoints of the images are relevant. In a three image stitching scenario, the matches of the left image and the middle one are relevant, but the matches for the left and right images are what is called false matches. See an example of this in the figure (1). Moreover, this can help to reduce the performance of the algorithm, since now, we will have reduced the searched that brute-force procedure has to deal with. To prevent extracting keypoints from the non-relevant regions, in the `detectAndCompute` function of the SIFT object (that will be explained in the next item) we can add a mask by parameter. The mask is used to specify a the pixels where keypoint detection should be performed.

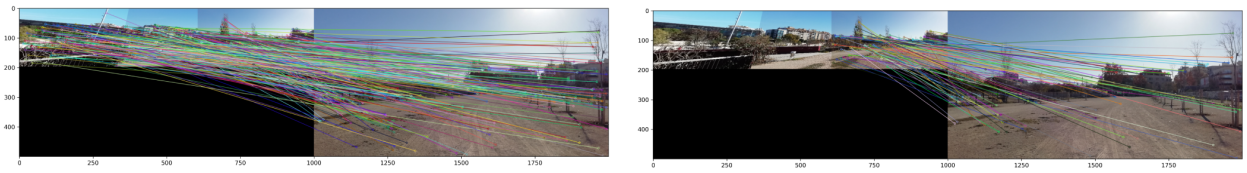

Figure 1: Top image: all matches found, Bottom image: matches of the adjacent image. Only a part of all the matches in this case are relevant, the ones belonging to rightmost image that was stitched.

- **Descriptors and keypoints**: for keypoint extraction and descriptor generation it is used SIFT. As we mentioned before, we used SIFT because it is invariant to rotation, scale, illumination and perspective changes. This characteristics enables good results when matching 2 sequential images that can be affected by this effects. Although this technology seems appropriate for this task, as we have commented, there are many more that could lead to good results too. We have used the OpenCV implementation for the SIFT descriptors extraction. Of all the computed matches, the user can select to only a percentage of the best matches to be used in the next computation. This percentage is controlled by a hyperparameter.

- **Descriptor matching**: we implemented it with a brute-force approach, it has been explained before so we are not going to delve in to it. I could implemented it myself but since OpenCV has its own implementation I am going to use it.
- **Computing homography matrix**: now from the keypoints of the remaining matches we extract the source points and the destination points for computing the homography matrix. In this case, the algorithm will warp the perspective of the second (right) image, thus the source points are from the left image and the destination from the right one. A relevant fact to mention is that although we resized the image at the beginning of the algorithm, **we have to return back a result with the original size**. Hence, we divide the points to the scale factor used in order to scale them back to its original size. We used the find homography implementation of OpenCV with the RANSAC method, this is used to robustly estimate a model from a set of data points by iteratively fitting models to random subsets, mitigating the impact of outliers and producing more accurate results in the presence of noise or erroneous data [8].
- **Warp Perspective**: with the homography matrix we warp the perspective of the right image, using the OpenCV function and specifying an output size of the sum of the 2 widths and the same height of the left image, in order to not fail the insertion of the left image (next item).
- **Join Images**: finally we insert the left image in the left side of the outputed image. In this point we have finished with the stitching, but usually we find a black rectangle in the right part of the image, due to the change of perspective. As we don't want it this way we have cropped this rectangle. See the diference in the figure (2)

Figure 2: Difference between the image with the black rectangle that can appear result of the warping and joining the images, and the image without it.

This function is named `computePairPanoramic` and you can see a full example of its structure and flow in the figure (3).

Figure 3: computePairPanoramic algorithm structure, **sizes of images in the diagram do not represent the actual sizes of the algorithm's output**)

3.3 Algorithm for image stitching of n images

At this point, since we have an algorithm that stitches two images, it came to my mind the idea that **not only join three sequential images, but n sequential images**. And that is what has been done in this section.

At first, the idea was to given a list of images, stitch the `last` with the `last - 1` and then the result stitch it to the `last - 2` and so on. However, with this approach there was an accumulation of warp perspectives on the right side of the image. This accumulation occurred because each stitching operation introduced a new perspective transformation, causing distortions to accumulate along the image width. In order to deal with this circumstance, we decided to start the stitching process from the middle image. First, we stitched the images to the left of the middle image (including the middle image itself if the total number of images is odd). Then, we stitched the images to the right. Finally, we stitch the results from both directions, creating the final output. You can see the results differences between this two approaches in the figure (4).

However, we have a problem, `computePairPanoramic` stitches the right image to the left image (it warps the perspective of the right image), and we need the opposite for stitching the images to the left of the middle image, we need to warp perspective of the left image. To avoid building another function we applied a horizontal mirror flip to both images, and we inserted them into the function reversed, the left image in the right position and the right image in the left position, and then we flipped horizontally the result:

```
fliped( computerPairPanoramic(left: flipped(img_right), right: flipped(img_left)) )
```

In this way we obtain the desired result by using the same function to both sides.

Yo can see a full example of this algorithm in the figure (5).

Figure 4: Comparison of the result of stitching the images always to the right (top images), and the result of stitching half the images to the left and half to the right (bottom images).

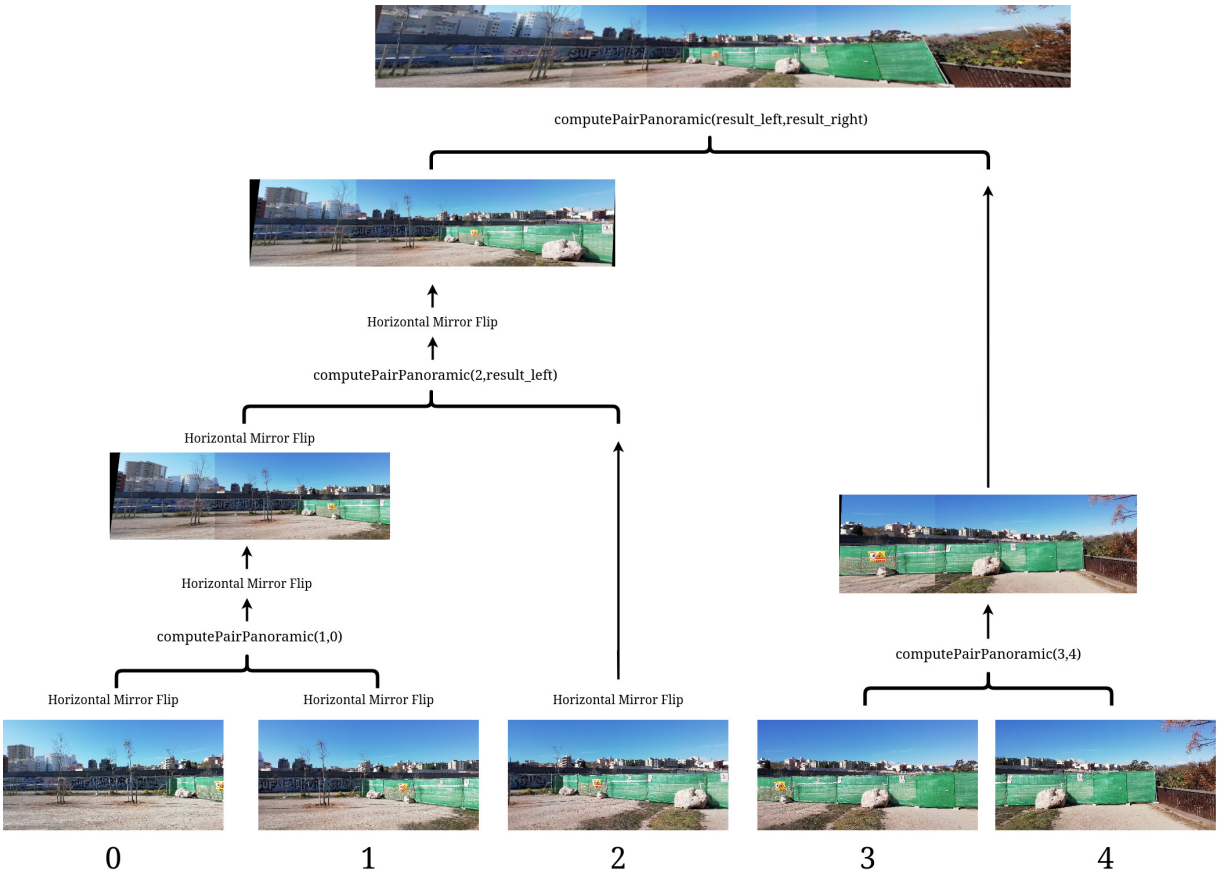

Figure 5: General Algorithm structure diagram (**sizes of images in the diagram do not represent the actual sizes of the algorithm's output**).

3.4 Mandatory and optional algorithm parameters

3.4.1 Mandatory parameters

- `img_list`: A sequential list of images (in ndarray format), arranged from left to right. The leftmost image is in the [0] position, while the rightmost image is represented by [len - 1] position in the list. The data type is List.

3.4.2 Optional parameters

- `scale_factor`: Factor in which the images are resized in order to compute the operations (SIFT, BF-matcher, Homography finding...) that build the panoramic image. Range between (0,1], if None (the default value) the scale factor is set automatically, as mentioned before. This parameter has to be a float.
- `p_best_matches`: Percentage of matches that will be used in finding the homography matrices. Range (0,1], if 1 all the matches will be used, 1 is the default value. This parameter has to be a float.
- `cut_black_part`: Indicates wether if the leftover black parts of the image, when warp perspective is applied, are removed or not. The default value is `True` and has to be a Boolean data type.

- `ransacReprojThreshold`: it is a hyperparameter of finding the homography matrix using the RANSAC method. A lower value restricts more the algorithm considering only very close matches as not outliers, and a higher value makes the algorithm more permissive (recommended with scenes with larger variations between points and more noise). The default value is 50.0 and has to be a float. Further explanation in the section (4.1.7).

3.5 Source Code

The source code is available in a folder called `/src/` that is in the same directory as this document.

4 Experimentation

4.1 Tuneable parameters

We have selected 7 tuneable parameters: 4 optional parameters seen in the section (3.4.2) and 3 more related to OpenCV functions used in the algorithm. Lets change its values and see how the algorithm behaviour changes:

4.1.1 Scale Factor

We have tried with the values: default (maximum dimension size = 1000, see section (3.2), 0.05, 0.5, 0.8 and 1. The execution time can bee seen in the table (1) and the algorithm output in the figure (6). The input is a list of 5 images of size 2080x4160.

As we can see the best option is to use the default value, that in the implementation simply a `None`, and this resizes the image to 1000xA or Ax1000 and then A is set automatically conserving the aspect ratio. This gives a really decent result and with a low execution time. The 0.05 value has also a low execution time, taking into account that we are stitching 5 images, but the result is much worse; the right side of the image is not joined properly. From default to 1 the output image result is pretty much the same, with slight differences, but the execution time is getting increased dramatically. As we can see, processing images with its original size the execution times arrives at 10 minutes. This is definitely not acceptable.

Scale Factor	Default	0.05	0.5	0.8	1.0
Time (seconds)	2.89	2.20	11.23	144.10	622.48

Table 1: Running time of the algorithm with different scale factor values.

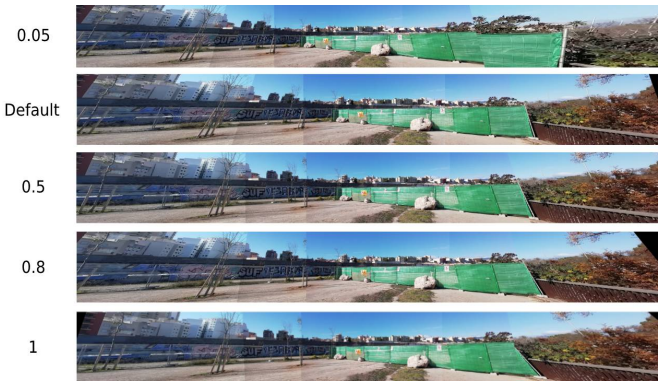

Figure 6: The output images of the algorithm with different scale factor values.

4.1.2 Percentage of best matches

In this case we have selected three different percentage values: 0.01, 0.5, 1 (default). The changes in this parameter reflect minor changes in the time performance of the algorithm (order of milliseconds). So we have not taken this metric into the comparison. You can see the results in the figure (7). As you can see if we select a tiny part of all the best matches, although those are the best matches the result is not well stitched because there is a lack of information. However, we can achieve remarkable results with only using the or even less

of the matches. Even though, I recommend using all the matches because it does not have a great impact on performance and in this way you assure the algorithm's potential is maximized.

Figure 7: The output images of the algorithm with different percentages of best matches

4.1.3 Cut black part parameter

The reader can see the behaviour of setting `False` or `True` this parameter in section (3.2) item ("Join Images"). I recommend setting it to `True` (default value) in order to remove unwanted black rectangles that enlarge unnecessarily the resulting image. Is fair to mention that, in some scenarios could happen that there is a real black stripe in the scene and the algorithm section in charge of cropping the black rectangle detects this stripe as the beginning of the rectangle, resulting an unwanted cropping of the scene. For this reason we keep the value of this parameter on the user hands.

4.1.4 Computing all the keypoints

As we have seen in section (3.2) in item ("Keypoint region selection") the keypoints are only computed in the relevant regions, but how the algorithm behaves if we take into account all the keypoints. You can see the behaviour in the figure (8). In the figure you can see that computing all the keypoints for finding the corresponding matches can lead to an erroneous result due to false matches, in the below image you can see that using the mask for computing only the relevant descriptors is the way to achieve the correct result.

Figure 8: Difference of the output image when using all the keypoints and using only the relevant ones

4.1.5 Brute Force cross check

In the context of `cv2.BFMatcher`, the parameter `crossCheck` is a flag that affects the behavior of the matching process. When `crossCheck` is set to True, the matcher performs a two-way matching, meaning that it considers a match between two keypoints only if the match is mutual. In other words, if the descriptor of keypoint Z in the first set is the best match for keypoint X in the second set, and in reverse, then the match is considered valid. In the figure (9) you can see the results. As the reader can see when crossCheck is set to False, since we don't ensure a correct match from the both sides, can outcome erroneous results. In the left result image the algorithm has only been capable of stitching correctly 2 images and in the second result image the algorithm has stitched correctly 4. For that reason, we are going to keep the `crossCheck` flag to True.

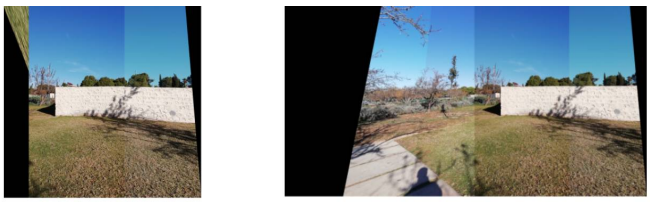

Figure 9: Difference of the output image when setting the `crossCheck` flag to True or False.

4.1.6 Homography Ransac

The `cv2.findHomography` function in OpenCV provides different methods for estimating the homography matrix. Instead of using RANSAC, we can choose from the following methods: `cv2.LMEDS` (Least Median of Squares) and `cv2.RHO` (RHO Method). Lets try them all and see the results, you can see them in the figure (10). As you can see, using RHO method the algorithm can not even stitch 2 images correctly. LMEDS seems to work a little bit better, as you can see it can stitch correctly the right image to the middle one, but it fails trying to stitch the left to the middle one. We have tried with several images and the behaviour is pretty much the same. Thus, we can conclude that using the RANSAC method is the best option for this problem.

Figure 10: Difference of the output image with different methods to compute the homography matrix.

4.1.7 RANSAC reprojection error

In `cv2.findHomography` there is a hyperparameter called `ransacReprojThreshold`. This parameter represents the maximum distance (in pixels) between the observed point and the point predicted by the estimated homography for the point to not be considered an outlier. If the distance exceeds this threshold, the point is treated as an outlier [1].

- A lower `ransacReprojThreshold` value makes RANSAC more strict, considering only very close matches as inliers. This can lead to a more accurate homography but may also exclude valid matches, especially in the presence of noise or complex scenarios.
- A higher `ransacReprojThreshold` value makes RANSAC more permissive, considering a broader range of matches as inliers. This can be useful when dealing with noisy data or when you expect larger variations between corresponding points.

To show the behaviour of this hyperparameter we are going to use the most complex scenario in the dataset of images that I used during the experimentation. This is the subway wagon scene. This is the most difficult environment of the dataset because: it has changing light conditions due to movement of the wagon; it has plenty of light reflections trough the windows, the ground, walls, etc; and it presents more strong rotations due to the movement of the wagon while I was taking the photos. Hence, this is the reason why we have chosen this scene to experiment with RANSAC hyperparameter.

The output results are in the figure (11). Until now, we were using a `ransacReprojThreshold` of 5.0, but as the reader can see, for this scenario is way better using a threshold of 50.0. This is because, as mentioned before, with complex scenes and environments with larger variations between points, a higher reprojection threshold makes more permissive the algorithm and can consider more inliners in the computation. Thus, we decided to put this hyperparameter as an optional parameter for the algorithm, in order to give the user the freedom to select this value regarding the scenario his is dealing with.

Figure 11: Difference of the output image with different RANSAC reprojection error thresholds. The scene is made of 4 sequential images of a subway wagon

4.2 Potential improvements of the solution

- **Stitched Images boundaries**: can be seen, in most of the results, clear joints and imperfect color and pixel intensity boundaries in the stitched images. This may arise due to variations in lighting conditions, color differences, and exposure levels between the individual images being stitched. Would be nice to add some color correction techniques in order to solve this problem and get smoother and more real results.
- **Black triangles**: In some resulting images, can be noticed a black triangle on one or both sides. This happens because of the warp perspective applied in the stitching process. Despite removing black rectangles, these triangular areas remain. An enhancement could involve removing them by detecting the hypotenuse and cropping orthogonally from a vertex, deleting the presence of these leftover triangles for a cleaner output.
- **Impact on performance of flipping**: the algorithm's performance may be impacted by the frequent use of horizontal mirror flips. As it can be seen in the figure (5), for stitching a scene of 5 images, the algorithm needs to compute 6 horizontal flips. The purpose of doing so was avoiding the creation separate functions, one for stitching to the left and one for stitching to the right. Nevertheless, perhaps building two separate functions would result a cleaner result regarding performance impact.
- **Improve performance by changing brute-force approach**: as we have mentioned previously the brute-force approach employed in the algorithm could be changed for another one less computationally demanding. Instead of calculating the distance between each descriptor in one image and all descriptors in another image, we could explore more efficient nearest neighbors search algorithms such as FLANN (Fast Library for Approximate Nearest Neighbors) [7].
- **Handle Image Scaling Effectively**: we could try to optimize the scaling factor used in resizing images to balance computational efficiency and matching accuracy. Experimenting with different scaling strategies to find the optimal balance for each specific use case.
- **Automate RANSAC threshold**: I don't really know how I would approach the following idea, but would be extraordinary to find a way to set the RANSAC reprojection threshold hyperparameter automatically. To create an algorithm that is capable of adapt this parameter according to the scene conditions. This, if it works properly, would probably improve the results of the algorithm.

5 Conclusions

5.1 Algorithm Constraints

- **Images of the same scene**: obviously the images must be of the same scene, it makes no sense to create a panoramic view with images that are not related to one another.
- **Sequential Images**: as it has been said, the images have to be sequential, otherwise probably the matches found will be false matches.
- **Order importance**: not only have to be sequential, but also go from left to right in the list that has to be passed by parameter to the algorithm function. `list[0]` will be the leftmost image and `list[len(list) - 1]` will be the rightmost image.

- **Shared elements within adjacent images**: between two adjacent images in the list must be shared elements: a tree, a rock, a field, building, etc. If not, it will be impossible to find true matches between the two adjacent images.
- **Planar transformation Assumption**: the accuracy of image alignment using homography depends on the assumption that the scene undergoes planar transformations. This constraints the use of the algorithm in non-planar or complex 3D scenes. For instance, if the user takes photos changing dramatically in position and orientation, the algorithm's accuracy may be affected drastically.
- **Tiny images**: if the images are too small, there might not be enough information for finding matches, and this can lead to the failure of the find homography matrix step. This is because the algorithm requires a minimum of four points to accurately compute the homography matrix.
- **Sensitivity to algorithm parameters**: the algorithm involves several parameters, and the success or failure of it could be attached to the parameter values. For instance: if we use a `p_best_matches` of 0 there would be no matches and the algorithm will fail. The same with the `scale_factor` parameter, if it's 0, there is no image to compute and if its 1000 the algorithm could last days to finish.
- **Static scene**: the algorithm success is tied on the assumption of a static scene. The presence of moving elements such as people, cars, or leaves can significantly impact correct keypoint matching. Dynamic elements introduce variations in the scene between consecutive images, leading to mismatches in keypoints during feature extraction and compromising the accuracy of subsequent steps.
- **Panoramic View of a large amount of images**: I have not checked it but probably there is a point where if we introduce a list with a great quantity of images the algorithm will end up failing. For instance: 100 images.
- **Destroying illumination changes**: although SIFT descriptors are invariant to illumination changes, if those are too strong that destroy the corresponding matches will probably be affected and consequently not detected. Thus, the algorithm is constrained by its sensitivity to destroying illumination changes between sequential images, leading to bad image alignment and warping perspective. Important to mention that those illumination changes must be exceptionally hard, as you can see in the figure (12), even with remarkably hard illumination changes, the algorithm still works fine.

Figure 12: Example of the algorithm dealing with strong illumination changes. This shows the correct effectiveness of the SIFT descriptors with hard illumination variance.

5.2 Project's Opinion and learning

This project has created the space not only to learn new things about computer vision, but also to apply techniques already seen in the subject theory. It has showed us the usefulness of the methods and strategies applied in computer vision problems that we are learning in the theory.

Moreover, thanks to this, I have developed a computer vision project that performs really well and maybe could be part of a portfolio in a time. I already did plenty of computer vision projects during my academic life, so this has helped me to refresh computer vision concepts and additionally new concepts, for instance I learned what homography matrix really mean.

If I had to say something negative about this project, is that I would wanted to have complete freedom on the memory report, instead of having to answer pre-defined questions. Because I had to answer those questions while at the same time ensuring that the report structure was coherent.

5.3 Conclusion

In conclusion, the developed image stitching algorithm shows a robust approach to creating panoramic views from sequential images. The project follows a three-step strategy: keypoints extraction, keypoint matching,

and image alignment. Through the use of SIFT descriptors, the algorithm ensures invariance to scale, rotation, illumination, and perspective changes, making it suitable for a variety of scenarios.
The experimentation phase revealed that the RANSAC was the best method for this particular problem. Also, we confirmed the success use of the reprojection threshold hyperparameter of RANSAC, proved by the positive outcome of using a high value in a larger variance scene.

The project also has potential improvements, including solving clear stitched image boundaries, handling left-over black triangles, optimizing the impact of horizontal mirror flips, exploring alternative keypoint matching algorithms, etc. Additionally we stated its constraints, such as the assumption of planar transformations and sensitivity to dynamic elements.

Finally, this project provided a valuable learning experience in computer vision, allowing practical application of theoretical concepts. It not only showed the effectiveness of learned methods but also contributed to the development of a successful computer vision project. Through this task, I refreshed existing concepts and learned new ones, revealing the success of the final purpose of this project and course, that is to learn the bases of the computer vision field.

References

[1] cv.findhomography. http://amroamroamro.github.io/mexopencv/matlab/cv.findHomography.html. Last accessed: 25/12/2023.

[2] Homography (computer vision). https://en.wikipedia.org/wiki/Homography_(computer_vision). Last accessed: 29/12/2023.

[3] Homography transform — image processing. https://mattmaulion.medium.com/homography-transform-image-processing-eddbcb8e4ff7. Last accessed: 26/12/2023.

[4] Image stitching using opencv — a step-by-step tutorial. https://medium.com/@paulsonpremsingh7/image-stitching-using-opencv-a-step-by-step-tutorial-9214aa4255ec. Last accessed: 24/12/2023.

[5] Introduction to feature detection and matching. https://medium.com/@deepanshut041/introduction-to-feature-detection-and-matching-65e27179885d. Last accessed: 24/12/2023.

[6] Introduction to sift(scale invariant feature transform). https://medium.com/@deepanshut041/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40. Last accessed: 24/12/2023.

[7] Python opencv – flannbasedmatcher() function. https://www.geeksforgeeks.org/python-opencv-flannbasedmatcher-function/. Last accessed: 27/12/2023.

[8] Ransac. https://en.wikipedia.org/wiki/Random_sample_consensus. Last accessed: 29/12/2023.

[9] Canwei Shen, Xiangyang Ji, and Changlong Miao. Real-time image stitching with convolutional neural networks. In *2019 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 192–197, 2019.