Computer vision - Final project
PANORAMIC IMAGES

Giulio Conca - ID: 2028844

Introduction

This final project consists in creating a panoramic image, starting from a set of images that capture a 360° field of view in the horizontal direction.
In order to obtain the final image, the first step is to project all the images on a cylindrical surface: after this operation, the trasformation between the images becomes a simple translation. The second step consists in extracting the keypoints of couples of adjacent images using SIFT or ORB, in order to match the corresponding features. Starting from the matches, the translation between two adjacent images is computed, both in the horizontal and in the vertical direction. Finally, the images are stitched together by exploiting the information on the translation computed in the previous step.

Cylindrical projection

As already specified, the first step consists in loading the set of images that must be stitched together, and in projecting them on a cylindrical surface. These two tasks are executed by the function *loadAndProject*, in which the arguments are the number of images to load and half of the field of view of the camera used to take the images.
For the loading operation, a vector of pictures, called *images*, has been defined. By using a for cycle, all the images have been converted to LAB colorspace first, and then inserted in the vector. After this operation, a second structure was defined, called *all_channels*. It is a vector of vectors of images, that contains in each position 3 cv::Mat objects. In particular, each image inside the vector *images* has been split in its three channels, that have been saved in another vector called *channels*, then the plane corresponding to the luminance has been equalized, and finally all the three planes of each image have been inserted inside *all_channels*. So in the first position it contains the three channels of the first image, in the second position it contains the three channels of the second image and so on (with the luminance plane already equalized).
After this operation, all the images inside *all_channels* have been projected on a cylinder using the function cylindricalProj, already provided inside the file "panoramic_utils2.cpp", in which the conversion to greyscale was commented because not needed, since now it is possible to work using the luminance plane of the images.
Finally, the vector *all_channels* has been returned by the function, and stored inside a vector of the same type, called *total_set*, inside the main function.

Figure 1: Masks used for the process

Creation of the masks

The following step consists in finding the keypoints for couples of adjacent images, in order to match them. To this aim, the method *detectAndCompute* was used, defined in the Feature2D class of OpenCV. Among the various arguments of this method, that will be explained in the next section, there is the possibility of defining a mask in order to limit the search of keypoints in a specific area of the considered image. This parameter will increase the speed of the process, that is often very computational demanding. Since all the images that must be stitched together have the same size, the masks can be computed only once and used multiple times inside the entire program.
The creation of the two masks have been performed by the *maskCreation* function, that has only one parameter, which is one image among the set (for example, the first one). Most of the keypoints are expected to be found in the overlapping region between two adjacent images. For this reason, only those parts must be considered. The mask that is used in the *detectAndCompute* function must be a 1-channel matrix of 8 bits unsigned char, that correspond to CV_8U in OpenCV. Moreover, the values of the masks must be set to 255 for the points that must be considered in the process, and to 0 for the points that shouldn't be considered.
So, first of all, two matrices of this type called *mask1* and *mask2*, with all zero values (completely black), were created. After that, two regions inside them were defined, called respectively *area1* and *area2*. The first contains the section of the original image that corresponds to three quarters of it on the right hand side, while the second contains three quarters of the original image on the left hand side. Finally, these areas inside the original image were set to 255 and the two masks have been returned by the function. The vector in which they are stored in the main function is called *masks*. The final result is shown in figure 1, where the black parts are not going to be considered in the *detectAndCompute* method, differently from the grey parts (as already explained, those parts are white in the program because they correspond to the scalar 255, however they are grey in this report only to better identify them in a white background).

Keypoints and feature matching

As already specified, the following task is about the computation of the keypoints and the matching between corresponding features inside couples of ad-

jacent images. All the process is carried out inside the function *matching*, that have different parameters: the vector of vectors where the images have been saved, called *all_channels* once again, the masks needed by the *detectAndCompute* method, descripted in the previous section, and a string that indicates which feature descriptor must be used among SIFT and ORB.
The first thing that is done by the function is the creation of another vector of images, called *panorama*, that will contain the final panorama image. The first position of the vector is taken by the image stored inside the first position of *all_channels*, that is obtained by merging its channels together and by reconverting it to the RGB color space. Of course, the image is equalized, since the luminance plane has been equalized in the *loadAndProject* function.
After that operation, all the other parameters needed by the *detectAndCompute* function have been defined. In particular, since the objective is to analyze couples of adjacent images, two descriptors and two vectors of keypoints have been defined, one of them for each image. They are called *descriptor1*, *descriptor2*, *kp1* and *kp2*. After this operation, the two pointers to the SIFT and ORB objects are initialized, using the *create* method, as well as the vector that will contain the matches between the two images and the pointers to the *BFMatcher* objects that have been used for matching the corresponding features. Since both SIFT and ORB are used in the program, two objects of this type are needed: the first uses the euclidean distance, useful for SIFT, while the other uses the Hamming distance, useful for ORB. Finally, the last structure is initialized: once again, it is a vector of vectors of float called *set_translations*, and it contains all the translations needed to merge the images together, both in the horizontal and vertical direction. For example, *set_translation*[0][0] will contain the translation along the horizontal direction needed to merge the first image with the second, while *set_translation*[0][1] will contain the translation along the vertical direction. Then in the position [1][1] and [1][2] the translation from the second image to the third are stored, and so on.
Now it's time to find the keypoints among couples of adjacent images and to merge them together. To this aim, a for loop is used, that consider all the images stored inside *all_channels*. As already specified, the operation is executed only for the luminance plane of the stored image: the other planes of the image will be translated accordingly as well. Depending on the type of feature descriptor that is used, the *detectAndCompute* method is called for both the images that constitute the first couple (i.e. the first and the second image of the set). Then the matching between the features are obtained using the *match* function, called by the correct *BFMatcher* pointer. Using the *drawMatches* function, it is possible to check which matches have been found by the program. An example of the output of this function is shown in figure 2. As shown by the figure, most of the matches are correct but there are some false detections. In order to obtain a better set of matches, a refinement based on the minimum distance between the matches is executed. In particular, first the minimum distance between matches is computed using a for loop, and then only the matches characterized by a distance that is lower than $2.5 \times d_{min}$ are kept. The other ones are erased from the vector.
After this operation it is time to compute the translation between the couple of images considered. This operation is done by the *ransac* function, called inside the *matching* function.

Figure 2: Output of the *drawMatches* function

Ransac robust estimator

The function *ransac* contains a simplified version of the Ransac robust estimator. The parameters for this function are the vector that contains the matches between the two images, the set of keypoints for each image and the first image of the couple. The algorithm to apply is the following: a random match is chosen among the entire set of matches, then for that particular match the corresponding translation is computed. After that, a search is performed among all the other matches, in order to understand how many matches have a similar translation. These operations are repeated for 50 times, selecting different random matches, and at the end only the translation with the highest number of compatible matches is considered. Finally, only the matches compatible with that translation were taken into account to compute the final translation, that is given by the average of the translations for all the compatible matches.
First of all, a random integer number between 0 and the size of the vector with the matches - 1 is generated, and it indicates the index of a random match inside the vector. Then, two vectors of floating points are initialized, called *temp_trans* and *vec_translations*. The latter will contain the final set of translations for all the compatible matches computed according to the Ransac algorithm. After that, a for loop is used in order to compute the operations for 50 times. At each iteration, the translation along the horizontal and vertical direction for that specific random match are computed, and stored inside the variables *deltax* and *deltay*. Then, a new for cycle is used to scan all the matches and, once again, the translations along the horizontal and vertical direction for each match are computed and stored inside the variables *deltaxj* and *deltayj*. A match is considered compatible if

$$|deltaxj - deltax| + |deltayj - deltay| < 5$$

If this is the case, the translations in the horizontal and vertical direction for that specific match are stored inside a cv::Point2f, that is then inserted in *temp_trans*. So, at the end of the inner for cycle, *temp_trans* will contain the translations for all the matches that have a compatible translation with respect to the random match. If the size of that vector (i.e. the number of matches that agree with the random one) is larger than max, which is an integer initialized to zero before the external for loop, then it means that a larger set of matches that are compatible with the random one has been found. In this case, max is set equal to the size of *temp_trans* for the following checks, and the *vec_translations* vector is set equal

to *temp_trans*, which is emptied to collect new translations for matches that are compatible with the following random match.
At the end of the algorithm, after 50 iterations, *vec_translations* will contain the translations of the largest set of matches that agree with one of the random matches. Finally, the ultimate translations are computed by taking the average of the translations stored inside *vec_translations*. They are stored inside the variables *final_x* and *final_y*, that are pushed inside a vector of float called *single_translation*, which is returned by the *ransac* function and stored in the vector *single_delta* inside the *matching* function.
Before pushing the values of *final_x* and *final_y* inside the *single_translation* vector, a check is performed for *final_x*. In particular, if its value is larger than the width of an image, it means that the camera is rotated from right to left, which is an issue that will be explained in the following section. In this case, the value of *final_x* is set as follows:

$$final\_x = width - final\_x \tag{1}$$

The important point, that will be exploited inside another function, is that in this case the value of *final_x* is negative but, once again, this will be explained in the following section.
However, getting back to the topic of this section, the vector *single_delta*, inside the *matching* function, will contain the translation needed to merge the couple of images considered in that iteration of the for loop. The values of this translation are stored inside another vector of vectors of float inside the function *matching*, called *set_translations*. So, at the end of the for loop, this vector of vectors will contain all the translations needed to merge together all the images, both the horizontal and vertical ones. Once again, the position [0][0] will contain the horizontal translation needed to merge the first two images, the position [0][1] the vertical one, the position [1][0] will contain the horizontal translation needed to merge the second and the third image, the position [1][1] the vertical one and so on.

Checking the order of the images

Once the *set_translation* vector is completely populated, the last thing to do is to use the information about all the translations to merge all the images together. The strategy to apply for the construction of the final panorama is the following: given a certain translation (both in horizontal and vertical direction) stored inside a certain position of the vector *set_translations*, a specific translation matrix is constructed, then the second image of the couple is translated according to the matrix, the blue columns (that are the result of the translation) on the right hand side of the translated image are deleted and the first image of the couple is merged with the translated image.
The output image is then merged to the following images by applying the same procedure, and so on. The details will be discussed in the following section.
Anyway, it's clear that these operations will work only if the camera rotates from left to right, otherwise the values of the translations are messed up and the deletion of the blue columns from the translated image, that is always applied to the right hand side of the image, will not work properly. For this reason, a check must be performed in order to understand if there are some images that

were taken rotating the camera from right to left. In this case, the images must be reordered in order to have a rotating camera only from left to right.
Just to be clear, among the provided set of images the ones with this issue are the images of the lab 2021. In particular, for the "lab-auto-challenging" dataset, the first two rotations of the camera are from the right to the left, while in the "lab-manual" dataset, only the first rotation presents this issue. The other datasets are fine, since the camera was always rotated clockwise for taking the different photos. Of course the reorder is not the only solution, but it is probably the simplest, even if is pretty computationally demanding.
The order of the images is checked inside the *wrongOrder* function, called inside the *matching* function, that takes two parameters: the entire set of translations, and the entire set of images. As already explained, the *set_translation* vector will contain a negative value of the translation in the horizontal direction only if the camera was rotated from the right to the left. In the other case, the horizontal translation will be a positive value.
Considering for example the "lab-auto-challenging" dataset, as already explained, the first two horizontal translations are negative (thanks to equation 1 inside the *ransac* function), while all the others are positive. The situation of the vector *set_translation* is the following (considering only the horizontal translations):

$$set\_translation[\,i\,][\,0\,] = \begin{bmatrix} - & - & + & + & + & ... \end{bmatrix}$$

As we can see, the first positive translation corresponds to the index 2 of the vector. But if we keep considering that dataset, we have that the third image is the last one that was obtained with a counterclockwise rotation of the camera. So it corresponds to the image that must be considered as the first in the new order, because if we start from it, all the other images are taken with a rotation of the camera from the left to the right. But the third image corresponds to the index 2 of the *all_channels* vector, which is the same index of the first positive horizontal translation.
So the operations that must be done by this function are the following: first it must find the index that corresponds to the first positive horizontal translation inside *set_translations*, that is stored inside the *first_positive* variable, then that index represents also the index of the image inside *all_channels* that must be placed in the first position of the new order. So the next step consists in swapping the image in the position 0 of *all_channels* with the image in the position given by the value of *first_positive*. Of course these operations must be executed only if the images are out of order, so only if there is at least one horizontal translation that is negative. For this reason, a variable called *out_of_order* is initialized with the value zero and set to 1 only in this case. The swapping operation is executed only if *out_of_order* is equal to 1. The same variable is then returned by the function, and stored inside the *check* variable in the *merging* function.
After these operations, the new correct order of *all_channels* is established. Please note that, for the *wrongOrder* function, the vector *all_channels* is passed by reference, so each modification inside the function *wrongOrder* is going to modify also the vector inside the *matching* function.
The value of *check* is checked inside the *merging* function, and if it is positive it means that the images have been reordered. In this case, a new call to the function *merging* is executed, in order to recompute again the translations between the images, using this time the correct order for the images.

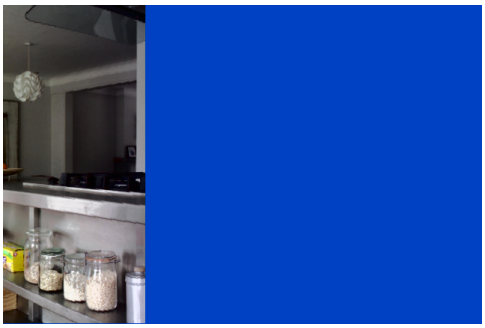

Figure 3: Translated image

It was also possible to avoid the second call by recomputing the correct values of the translations starting from the values already computed for the "wrong order" translations. Once again it wasn't easy so, even if two calls of the *matching* function are pretty computational demanding, I chose the simpler way between the two (which is probably not the simplest, but the *wrongOrder* function was the best solution I was able to think of to solve the problem in a reasonable way). Of course, in the second call the *wrongOrder* function will return the value zero and the program will skip other calls to the *matching* function.

Translating and stitching the images

As already explained, after these operations the *set_translations* vector will contain all the translations needed to merge together all the images stored inside *all_channels*. So, with a for loop, a translation matrix is defined for each couple of images, with the following shape at every iteration:

$$translation\_matrix = \begin{bmatrix} 1 & 0 & -set\_translation[i][0] \\ 0 & 1 & -set\_translation[i][1] \end{bmatrix}$$

The - sign is due to the fact that the values stored inside *set_translations* are used to merge the first image to the second, but we need to stitch the second image to the first.
The translation is applied to all the channels corresponding to a specific image, and once they have been translated, they are merged together and the image is reconverted to he RGB color space, in order to obtain the equalized version of the original image. An example of a translated image (already reconstructed from its channels) is shown in figure 3. In order to merge the first image of the couple with the second, the blue part of the image must be eliminated. To this aim, it's possible to note that the blue area has an horizontal size that is equal to the horizontal translation. To eliminate it, it's then sufficient to use the *Range* function of OpenCV, saving all the rows of the translated image and just the first N columns, where N is given by the difference between the total number of columns of the image and the horizontal translation. The final result is shown in figure 4.
To merge the first image and the translated image without the blue part, the function *hconcat* is used, and the result is saved in the *add* cv::Mat object. At

Figure 4: Translated image without the blue part

the beginning of the *merging* function, as already specified, a vector of images called *panorama* was initialized, and the first position was occupied by the first image inside the set. At every iteration of the for cycle, the image stored in panorama (starting from the first image of the set) is horizontally concatenated with the following one, then saved in an object called *add* and added to the *panorama* vector. For this reason, at the end of the for cycle, the final panorama will be saved in the last position of the vector *panorama*. The panorama image is finally stored inside an object called *final_image*, that is shown by using the function *imshow*, provided by OpenCV.
Note: actually, the *merging* function uses another vector of images, called *panorama2*. This is needed to avoid the typical problem of "shadowing" caused by the second call to the function in case of out of order images. Basically, the vector *panorama* of the (possible) second call might shadow the same vector inside the first call, so another vector is needed to have vectors with different names inside the two calls of the function.

Other strategies

The findHomography function

The descripted pipeline is not the only way to perform image stitching. Another way consists in finding the translation between couples of adjacent images by using the *findHomography* function of OpenCV, specifying the RANSAC method in the corresponding parameter. To work properly, that function requires two vectors of Points2f that should contain the coordinates of the keypoints of the first image and the coordinates of the keypoints of the second image. Anyway, the most important parameter is the mask, that is set by the function. In particular, its number of rows is equal to the number of matches found between the two images. In order to retrieve the inliers, a comparison between the mask and all the matches must be performed: when the mask value in the i^{th} rows is equal to one (after a cast to int), it means that the i^{th} match is a good match. Given the set of inliers, it is then possible to compute the translation in both directions taking the average of all the translations. Once the single translation is computed, the rest of the code is exactly the same already descripted.
The code, that has been developed for an initial version of the project, is reported in the *findhom* function, that is completely commented. This is because

after some days I decided to implement manually the Ransac algorithm, so that function was not used anymore, but since (at least at the beginning) I tried both the methods, I thought it would have been interesting to show the results of both of them. Anyway, the function doesn't work and it's commented to underline the fact that it was developed for another script, that was abandoned after some days.
The final result is very similar to the one obtained with the *ransac* function, already commented. It is possible to look at an example computed with this technique by checking the file called "homography_result.bmp" inside the folder "results" (the final image is not equalized because, as already specified, the script it comes from was an initial version, without the equalization part).

The warpPerspective function

Another variation of the project, that is strictly related to the *findHomography* function, is given by another very useful function provided by OpenCV, that is called *warpPerspective*. After the computation of the homography matrix using the *findHomography* function, it is possible to pass that matrix to the *warpPerspective* function in order to compute the transformation that is needed to match the second image of the couple to the first one.
This function also includes other types of transformation, not only the simple translation, and its results are very good. On the other hand it is more difficult to handle, at least with the pipeline that I decided to use. This is because a rotation may cause an increase of the number of rows of the image, not only of the columns such as in the final version of the project. Moreover, it's very difficult to concatenate horizontally two images without knowing the translation (that in this case is not computed manually, but it's calculated by the function) because the horizontal size of the blue part is unknown.
However, also this function was taken into account in an initial version of the project. Once again, the code is reported in the commented function called *perspective*. The output of that function, that was used inside another script (once again, another initial version of the project), is shown in figure 5. The first image represents the transformation that has been applied to the second image of the couple, while the other image shows the result of the stitching operation. As it is easy to see, the results are very good but the function is much more difficult to handle. For this reason, it was abandoned very soon.
However, it underlines somehow the fact that, even after the cylindrical projection function, the simple translation is not always the correct transformation to apply. When we are not dealing with an ideal case, a rotation and in general a change in the perspective of the image is needed, as shown in figure 5.

Results

The final panoramic images are stored inside the folder "results", one for each provided dataset (the most part of them computed with SIFT, a couple of images computed with ORB). Moreover, the result of the *findHomography* is stored inside the file "homography_result.bmp", and the result of the *warpPerspective* function is stored inside the file "perspective.png".
The final panoramic images have not been inserted inside the report because

Figure 5: Results of the *warpPerspective* function

their size is too large. However, they are shown at the end of the program by using the *namedWindow* function, specifying the NORMAL_WINDOW flag, and then using the usual *imshow* function.
Final note: to change the dataset of images loaded by the program, it is sufficient to indicate the number of images in the *num* variable inside the main, the correct parameter for the *cylindricalProj* function, stored inside the *angle* variable (half of the field of view) inside the main, and to change (if needed) the way the string *name* is composed inside the *loadAndProject* function.