Computer Vision – Lab 4: Image Stitching

Vittorio Esposito 2005795

At the beginning of the program, I loaded the input images using the loadInputImages() function. They get loaded in a 3x3 matrix.

Then, the cylindricalProjection() function is invoked to perform cylindrical projection on the input images, and passing as argument the angle (double). This function calls the cylindricalProj() function provided by the professor, on each image of the matrix. I was able to get good results passing 10° as angle.

At this point, I used SIFT to detect the features and get the descriptors, by means of the extractFeaturesSIFT() function, which requires the number of features to be extracted as argument. I‘ve been able to get good results also with extractFeaturesORB() and extractFeaturesBRISK(). In this way I was able to try SIFT, ORB, and BRISK. All these functions are in charge of detecting the features and computing the descriptors on all the images of the 3x3 matrix.

Optionally a showKeypoints() function can be invoked so that the key points just extracted are visible.

At this stage, it necessary to find the matches. The strategy I adopted, to try something different from what the professor suggested, to also allow code reusability, was to find the matches row by row. For example, the first matches that will be found are those between the image (0,0) and (0,1), and then between (0,1) and (0,2). This can be repeated for each row. More in general, the matches will be found between: (row, 0) <-> (row,1) and between (row,1) <-> (row,2), where “row” is the row number (e.g., 1,2, or 3). To achieve this, I invoked the findMatchesV2() function which takes the desired number of output matches to be returned. Only the best N matches, based on the Euclidean distance, will be returned. I found good results with a fixed number of 6 matches between pairs of images. Row by row, for each pair of images, the BFMatcher has been utilized.

I also implemented the findMatches() function that returns only the matches with distance lower than a ratio, provided as argument, times the minimum Euclidean distance between the matches, as requested by the professor. I found good results with a ratio of 1.8. The BFMatcher has been utilized in this case as well.

At this point, the matches are available. It has been necessary to find the transform (homographies) between the matched key points by means of the findHomographies() function. This is in charge of calling multiple times the computeHomography() function, which takes as arguments the key points of the first image, those of the second image and the matches between these. This function uses the findHomography() function to compute the homography between an object “obj” of the first image and a “scene” (second image). (Note: for the variable naming I got inspired by an online guide, but the code is original).

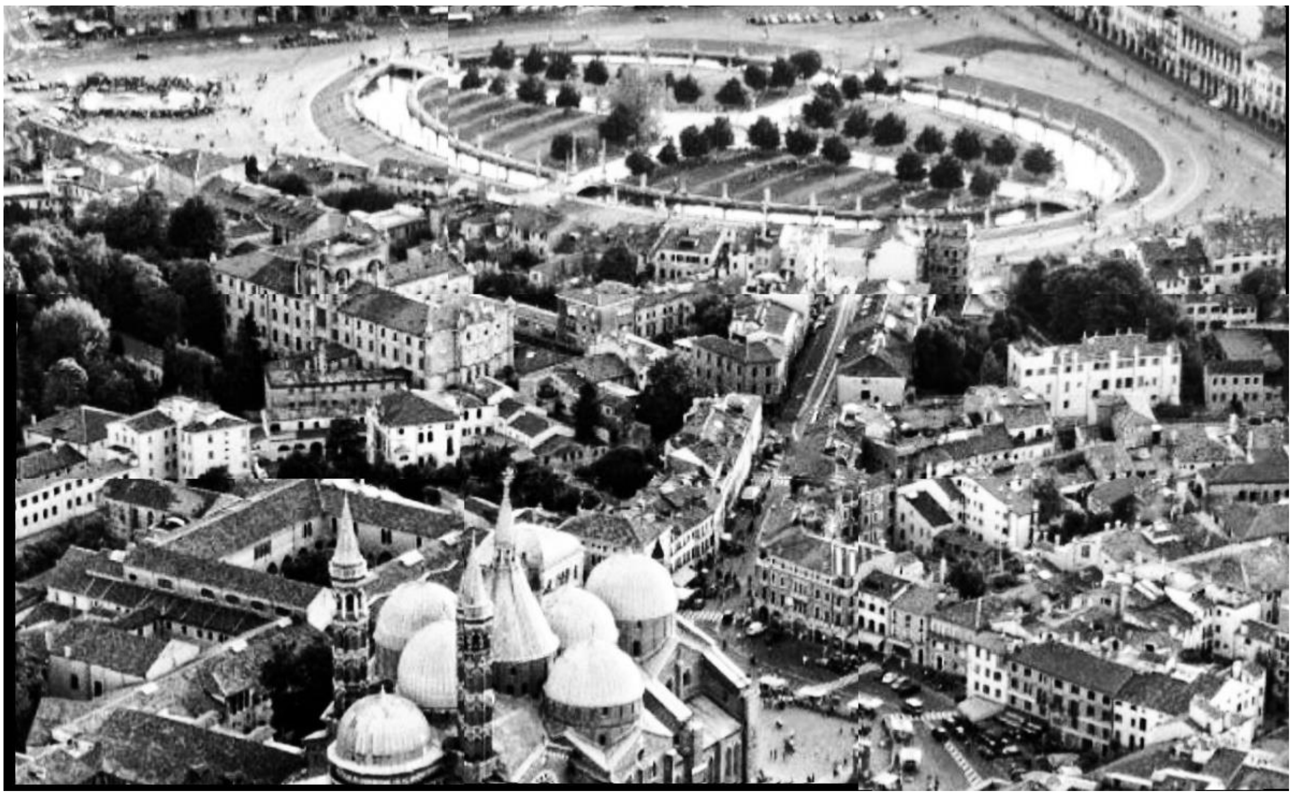
Now, homographies between each couple of images (row by row) are available. To find the distances between the input images to be used in the stitching phase, I used the findDistances() function. It just computes a “simple” average using the results (e.g., coordinates) of the function used at the previous step.

At this point I invoked the performRowStitching(). Note: there is another function with the same name, but different signature, not to be confused with this one (overloading concept has been used). The performRowStitching() function invokes, for each row, the performRowStitching(int rowIndex, vector<Mat> imgs) function. “rowIndex” is the index of the row (i.e., 0,1, or 2) while “imgs” is a vector of images (e.g., the first row of the matrix during the first iteration of the for loop). The performRowStitching(int rowIndex, vector<Mat> imgs) computes a temporary sum of coordinates of the distances found at the previous point, and the X dimension of the row stitched image. Then, it computes the Y dimension of the row stitched image as maximum Y dimension of the 3 images in the row. Then, the row stitched image is prepared, and pieces of the input images are progressively copied over it, obviously considering the previously computed coordinates (distances). At the end, the stitched row image is returned and loaded in an array.

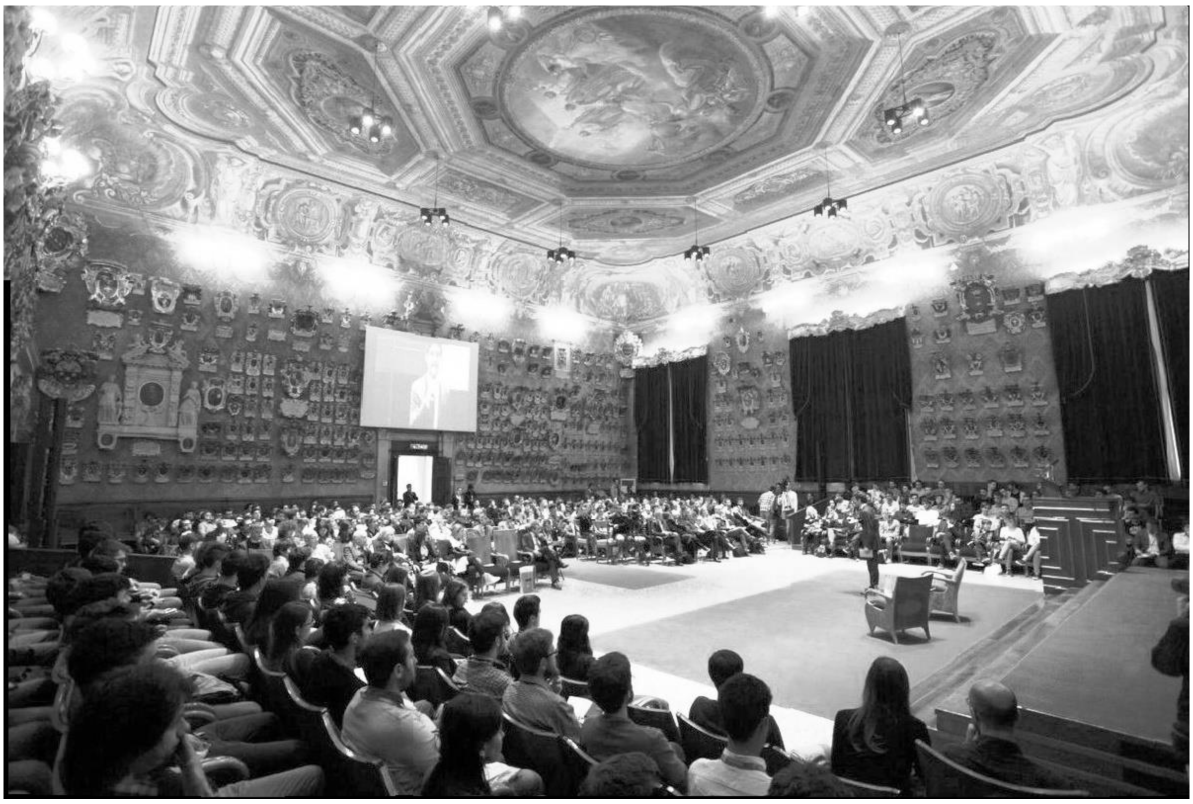
At this point, three stitched images are available, one for each row of the 3x3 matrix. The procedure now is very similar to before, with the exception that now it’s not necessary to operate on 3 rows, but just on one. As a first step, the stitched row images are rotated invoking the rotateStitchedImages() function, which takes in input a Boolean to indicate the versus of rotation (e.g., false is ROTATE_90_COUNTERCLOCKWISE and the operation is performed on the 3 stitched images). Features and descriptors are extracted with extractFeaturesSIFT() as before (ORB and BRISK can be used as well). Matches are found as before with findMatchesV2() or findMatches(). Then, the findHomographies() function computes the homographies between the image (0) and (1), and then between image (1) and (2). Distances are computed as well invoking findDistances(). Stitching is done using performRowStitching(int rowIndex, vector<Mat> imgs), passing 0 as “rowIndex” and the array of the three stitched images as “imgs”. Then, rotateStitchedImages() is called passing true as argument (i.e., ROTATE_90_CLOCKWISE and performed on the final image). Finally, to improve the image, equalizeHist() has been used to equalize the histogram, as already done in previous lab. The image is the written to “outputImage.jpg” and showed to the user.

I took my own pictures of 3 different subjects. Experimental results follow.

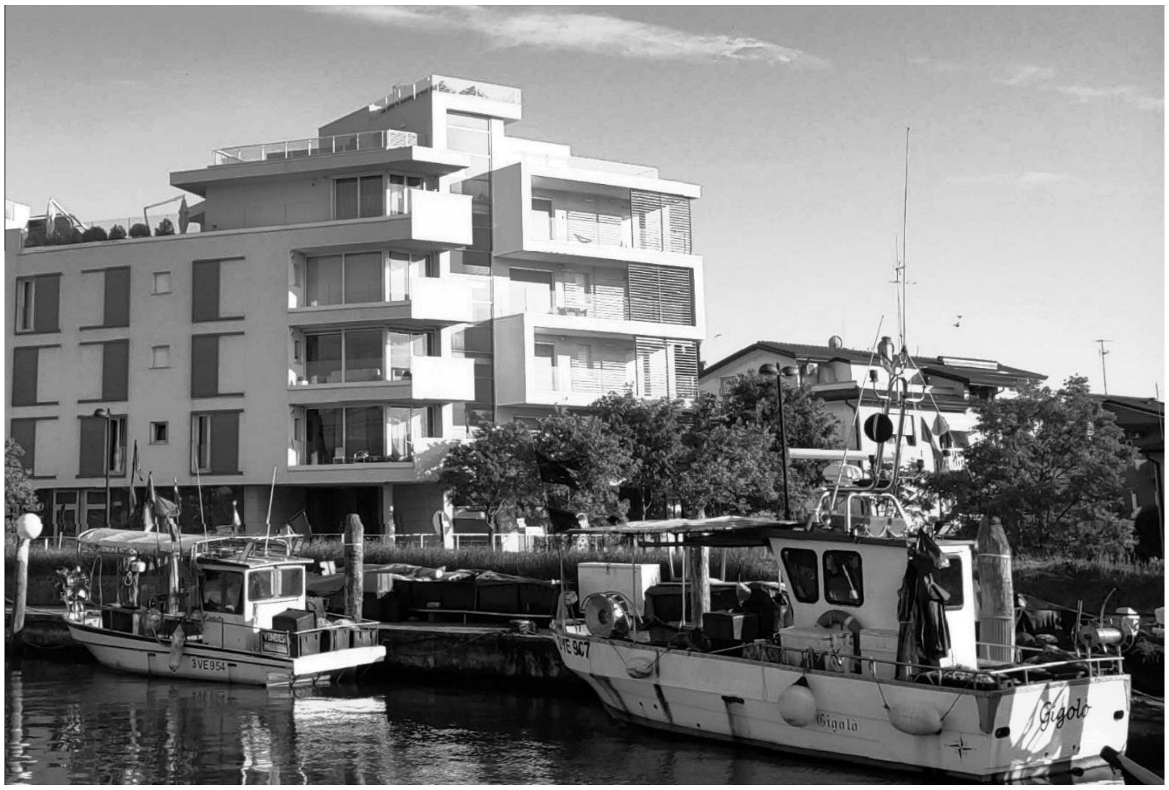
Padova – Discrete results have been achieved. Note the black portions along the outer edges of the output image due to the imperfection of the input images.



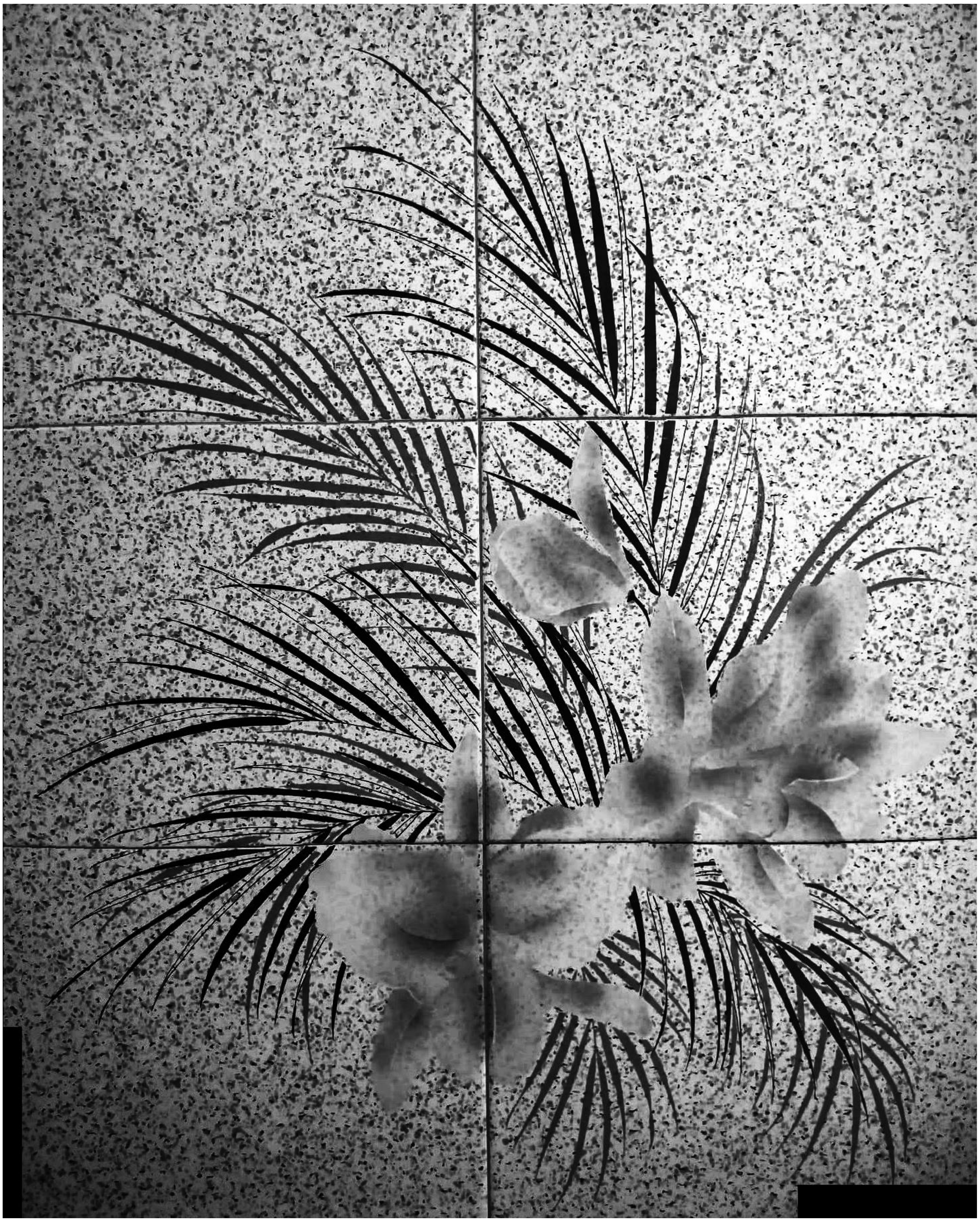
Aula Magna – A good result has been achieved. Artifact due to stitching are not visible. Note the black portions along the outer edges of the output image due to the imperfection of the input images.



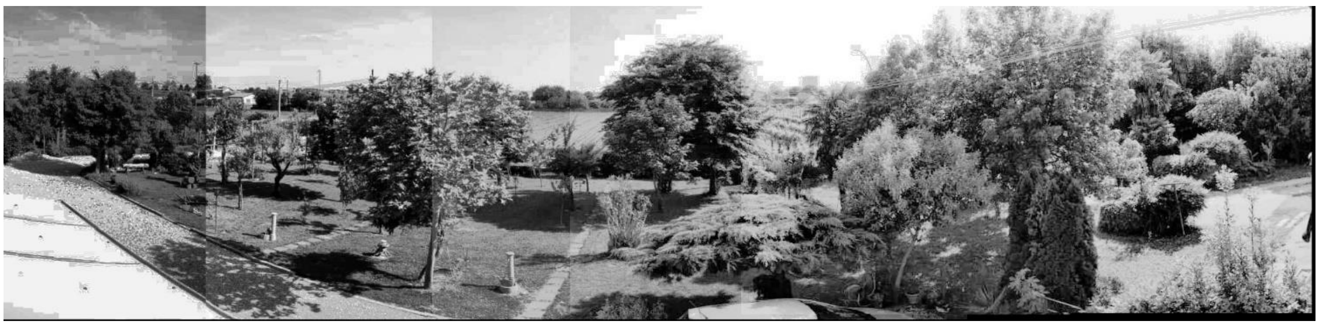
My own pictures (Caorle) – Quite good results, considering the presence of many small details and that input images were all cropped to different sizes. Note the black portions along the outer edges of the output image due to the imperfection of the input images.



My own pictures (decoration) – Quite good results, considering the presence of many small details and that input images were all cropped to different sizes. Note the black portions along the outer edges of the output image due to the imperfection of the input images.



My own panoramic picture (Garden) – Discrete results, but I wanted to try panoramic stitching as well. Note the black portions along the outer edges of the output image due to the imperfection of the input images.



For this last case, the code I used is very similar, except that most of the operations are done just in one row containing N images.

To summarize:

1. To merge the images, I load them on a 3x3 matrix, then merge row by row. Afterwards, I rotate the three stitched images and merge them as I did for the lines (this is to reuse the same portion of code 4 times).
2. I also use the professor's cylindrical projection function.
3. I have tried SIFT, ORB and BRISK.
4. I wrote a function that returns matches with distance < minDistance * ratio and one that returns the best N matches between keypoints of two adjacent images.
5. For homographies I use the findHomography function recommended by the professor.
6. I took my own pictures of 3 different subjects.
7. At the end i use equalizeHist to improve the image.
8. I tried panoramic stitching as well.