Image Completion
with Feature Descriptors

Naoures Atidel Hamrouni

:

Contents

Objective

The objective of this lab is to develop an algorithm that can automatically find and correct missing or corrupted areas in an input image. The algorithm should use a set of provided patches that contain the missing information to identify the correct patches and their locations in the image. By utilizing the patches data, the algorithm should perform image restoration by replacing the missing or corrupted areas with the appropriate patches.

- Load the corrupted image and patches.
- Extract SIFT features from the image and patches.
- Compute matches between image and patch features.
- Refine the matches using the distance ratio test.
- Compute the affine transformation matrix using RANSAC.
- Overlay the patches on the image using the homography matrix

1. Step 1: Load Images

- First step is to load the images, in this project we have 2 types of images: Corrupted images, with some missing parts, and the second type are the patches containing the missing information of the same corrupted images. All the images are saved in vectors.

```cpp
String path = "C:/Users/Hamrouni/Desktop/work/HW2/MyHW/image_to_complete/*.jpg";
vector<Mat> lists;
vector<string> f;
glob(path, f, true);
for(size_t i =0 ; i<f.size();i++) {
    Mat corruptedImage = imread(f[i]);
        if (corruptedImage.empty()) {
            //always check if the image is emty or not
            cout << "Failed to load the corrupted image" << endl;
            return -1; // Return an error code or handle the error accordingly
        }
        lists.push_back(corruptedImage);
```

```cpp
// Load the corrupted images

Mat corruptedImage = imread("C:/Users/21629/Desktop/cours/S2/computer_vision/projet/HW2/pratodellavalle/pratodellaval
```

Figure 1 : Loading images

```cpp
vector<Mat> patches;
String path_patches = "C:/Users/Hamrouni/Desktop/work/HW2/MyHW/venezia/venezia/patch*.jpg";
vector<String> f1;
glob(path_patches, f1, true);
for (size_t i = 0; i < f.size(); ++i) {
    Mat im = cv::imread(f1[i]);
    if (im.empty()) continue;
    patches.push_back(im);
}
```

Figure 2 : Loading Patches

2. Preprocessing images

Before other steps, images needed to be processed. I called equalization function for avoiding color jumps, this function was used in Lab2. This step helps to enhance some blurred images.

```cpp
vector<Mat> new_lists_corruptedImage;

for (size_t i = 0; i < lists_corruptedImage.size(); ++i)
{
    Mat res = equalization(lists_corruptedImage[i]); // equalization function
    new_lists_corruptedImage.push_back(res);
}
```

Figure 3:Apply equalization to the corrupted images

```cpp
    // Apply equalization function to the patches
    Mat equalizedPatch = equalization(im_patches);

    // Apply Gaussian blur to the equalized patch
    Mat blurredPatch;
    GaussianBlur(equalizedPatch, blurredPatch, Size(0, 0), 2.0);

    // Initialize the sharpened patch for pre-processing
    Mat sharpenedPatch = equalizedPatch.clone();

    // Apply the sharpening operation
    addWeighted(equalizedPatch, 1.5, blurredPatch, -0.5, 0, sharpenedPatch);

    patches.push_back(sharpenedPatch);
```

Figure 4 : Apply equalization, Gaussian blur and sharpening operations to enhance the patches

3. Step2: Extract SIFT Features from the image

- Figure 2 shows the creation of a SIFT detector object and its application to detect keypoints in the corrupted image. The detected keypoints are visualized by drawing them on a new image using the drawKeypoint function, which returns an image containing the keypoints. The steps are :
- Creating a SIFT detector and descriptor extractor.
- Detect keypoints using the detector on the equalized corrupted image.
- Compute descriptors for the detected keypoints using the extractor.
- Draw keypoints on the equalized corrupted image.

- Create a new window and display the image with keypoints on it as showed in figure 6.

```
// Create SIFT detector and descriptor extractor
Ptr<SIFT> detector = SIFT::create();
Ptr<SIFT> extractor = SIFT::create();

vector<KeyPoint> keypoints;
detector->detect(equalizedcorruptedImage, keypoints);
Mat descriptors;
extractor->compute(equalizedcorruptedImage, keypoints, descriptors);
Mat corruptedImage_keypoints;
drawKeypoints(equalizedcorruptedImage, keypoints, corruptedImage_keypoints);
namedWindow("Corrupted Image with Keypoints", WINDOW_AUTOSIZE);
imshow("Corrupted Image with Keypoints", corruptedImage_keypoints);
```

Figure 5 : Extract SIFT Features from the image

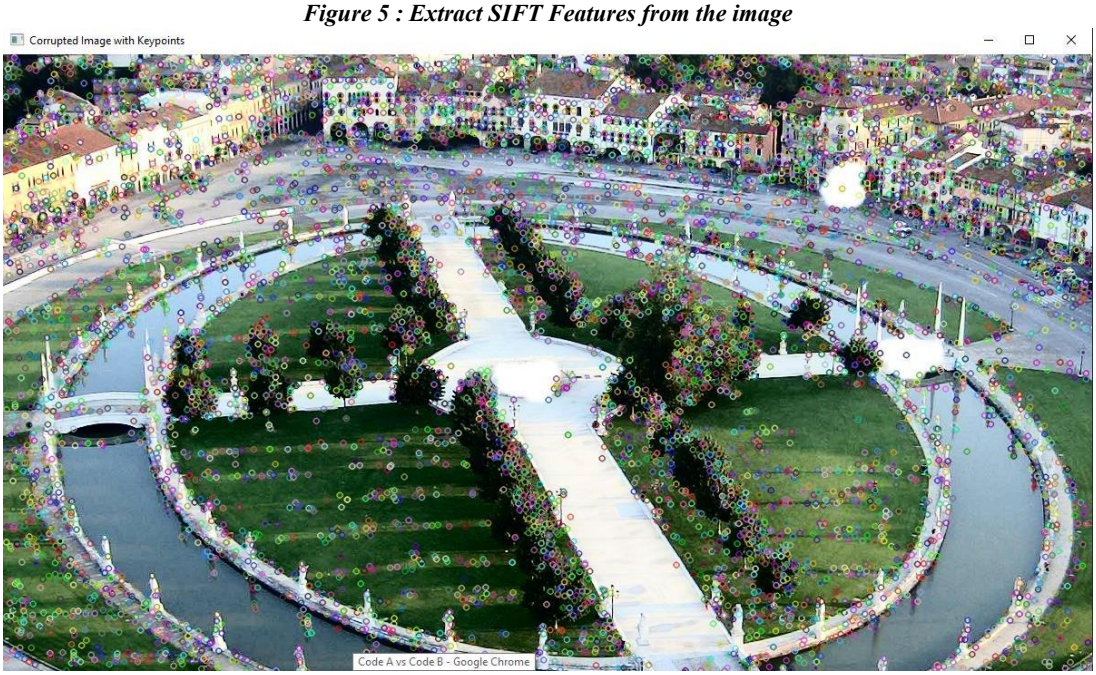


Figure 6 : Result of Using SIFT to extract Key points

 The utility of descriptors lies in their ability to capture discriminative and robust information about the visual characteristics of an image. Descriptors allow for the compact and meaningful representation of specific parts of an image, such as corners and edges.

4. Step 3: Extract SIFT Features from the patches

• At first the extraction of Sift feature was done in the main for the images and for the patches.

This may be a duplication for the same task. So I made function called "**extractSiftFeaturesFromPatches"** function that will takes a vector of images, a SIFT detector, and a descriptor extractor as input. It iterates through the patches, detects keypoints, computes descriptors, and displays the images with keypoints and their descriptors.After, the main function loads the corrupted images and patches, creates a SIFT detector and descriptor extractor, and then calls the "**extractSiftFeaturesFromPatches"** function twice - once for the corrupted images and once for the patches.

```cpp
//extractSiftFeaturesFromPatches is a function to extract Sift Features From all the Patches
void extractSiftFeaturesFromPatches(const vector<Mat>& patches, Ptr<Feature2D> detector, Ptr<DescriptorExtractor> extractor,
    vector<vector<KeyPoint>>& keypoints, vector<Mat>& descriptors)
{
    for (int i = 0; i < patches.size(); i++) {
        Mat patch = patches[i];

        // Detect keypoints and compute descriptors for the patch
        vector<KeyPoint> patchKeypoints;
        Mat patchDescriptors;
        detector->detectAndCompute(patch, Mat(), patchKeypoints, patchDescriptors);

        // Store the keypoints and descriptors for the patch
        keypoints.push_back(patchKeypoints);
        descriptors.push_back(patchDescriptors);

        // Display keypoints on the patches
        Mat patchWithKeypoints;
        drawKeypoints(patch, patchKeypoints, patchWithKeypoints);
        String titre = "Patch number " + to_string(i + 1) ;
        imshow(titre, patchWithKeypoints);
    }
}
```

Figure 7: The extractSiftFeaturesFromPatches function

```cpp
//_____ Step 3 : Extract SIFT features from the patches _____

// Define the input vectors for the extractSiftFeaturesFromPatches function
vector<vector<KeyPoint>> patches_keypoints;
vector<Mat> patches_des;

// call the extractSiftFeaturesFromPatches function
extractSiftFeaturesFromPatches(patches, detector, extractor, patches_keypoints, patches_des);
```

Figure 7 : calling the sift function

Figure 8:Extract SIFT Features from the patches

Compute the match between Images Using ORB

For this step I performed the ORB feature extraction on the corrupted image, and on each patch individually.

- Starting by creating the ORB object with specified parameters.

```cpp
// Start by creating the ORB object
    Ptr<ORB> orb = ORB::create(7000, 1.05f, 8, 10);
```

So here:

- 7000: Specifies the maximum number of keypoints to be detected.
- 1.05f: Specifies the scale factor between the levels of the Gaussian pyramid.
- 8: Specifies the number of levels in the Gaussian pyramid.
- 10: Specifies the edge threshold. It is the size of the border where keypoints are not detected due to edge effects

- After we start performing ORB feature extraction on the corrupted image:

o Define vectors to store the ORB keypoints and descriptors for the corrupted image. o Detect and compute ORB keypoints and descriptors on the equalized corrupted image. o Draw ORB keypoints on the equalized corrupted image.
o Create a named window and display the corrupted image with ORB keypoints.

- And perform ORB feature extraction on the patches:

```
// Perform ORB feature extraction on the Corrupted image
vector<KeyPoint> orb_keypoints_corruptedImage;
Mat orb_descriptors_corruptedImage;
Mat corruptedImage_with_ORB_keypoints;
orb->detectAndCompute(equalizedcorruptedImage, Mat(), orb_keypoints_corruptedImage, orb_descriptors_corruptedImage);
drawKeypoints(equalizedcorruptedImage, orb_keypoints_corruptedImage, corruptedImage_with_ORB_keypoints);
namedWindow("Corrupted Image with ORB feature extraction", WINDOW_AUTOSIZE);
imshow("Corrupted Image with ORB feature extraction", corruptedImage_with_ORB_keypoints);
```

o Define vectors to store the ORB keypoints and descriptors for the patches. o Define vectors to store the ORB keypoints and descriptors for each patch. o Detect and compute ORB keypoints and descriptors on the current patch.
o Store the ORB keypoints and descriptors in the respective vectors. o Draw ORB keypoints on the current patch. o Finally display the current patch with ORB keypoints.

```
// Perform ORB feature extraction on the patches
vector<vector<KeyPoint>> orb_patches_keypoints;
vector<Mat> orb_patches_descriptors;

for (int j = 0; j < patches.size(); j++) {
    vector<KeyPoint> orb_keypoints;
    Mat orb_descriptors;
    orb->detectAndCompute(patches[j], Mat(), orb_keypoints, orb_descriptors);
    orb_patches_keypoints.push_back(orb_keypoints);
    orb_patches_descriptors.push_back(orb_descriptors);
    Mat patch_with_keypoints;
    drawKeypoints(patches[j], orb_keypoints, patch_with_keypoints);
    imshow("Patches with ORB feature extraction", patch_with_keypoints);
```

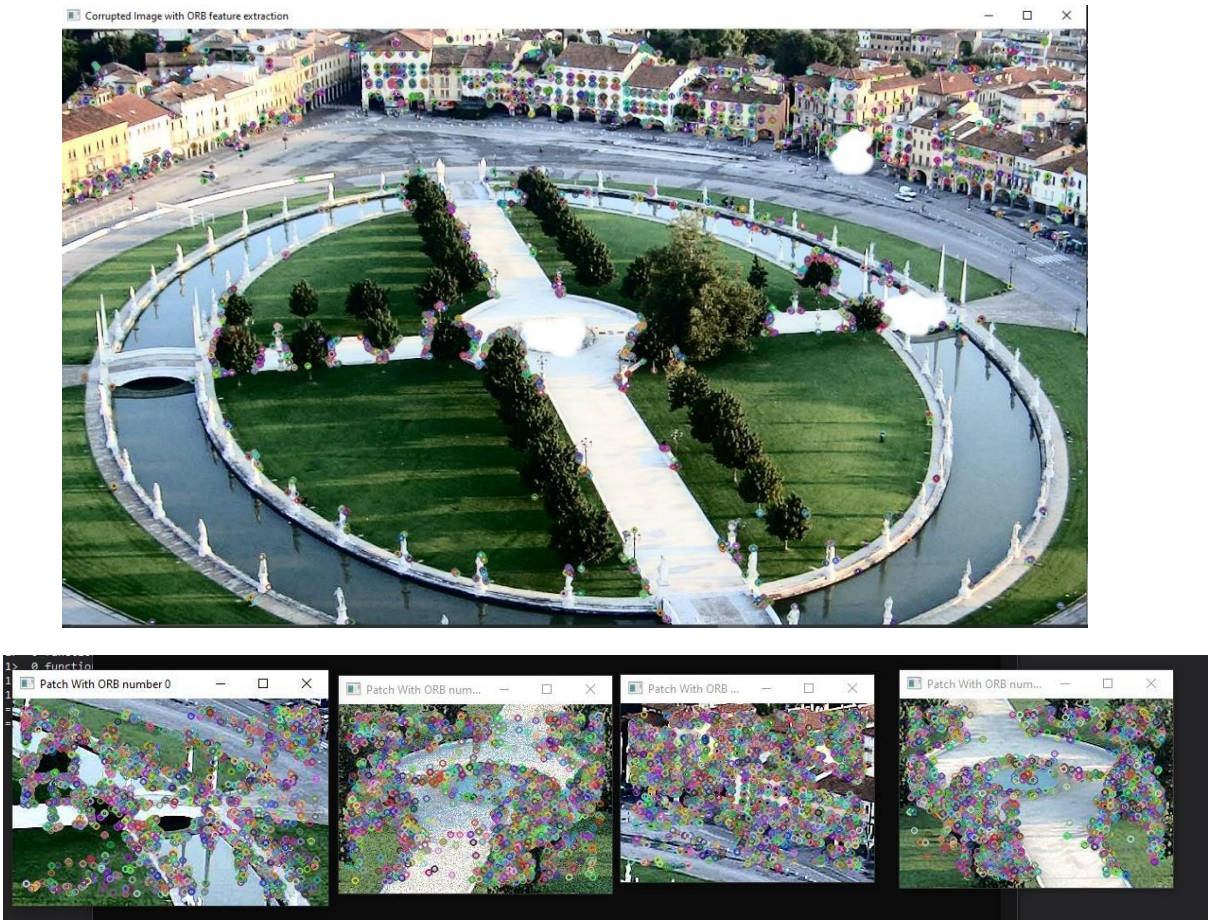


Figure 9 : Corrupted images and patches with ORB

5. Step 4 :

a. Compute the match between Images Using SIFT

After extracting SIFT features from the corrupted images and patches, I proceed to compute the match between the image and patch features.To calculate the matches between the features of an image and a patch using the "BFMatcher" and L2 distance, follow these steps:

- First, create a Brute-Force matcher object using "BFMatcher(NORM_L2)" to compute matches based on the L2 distance metric.

```
BFMatcher matcher(NORM_L2);  // Create a cv::BFMatcher matcher to compute matches using L2 distance

for (int k = 0; k < patches.size(); k++)
{
    vector<DMatch> new_matches;
    matcher.match(descriptors, patches_des[k], new_matches);  // Match the descriptors of the image and patch

    sort(new_matches.begin(), new_matches.end(), [](const DMatch& x, const DMatch& y)
    {
        return x.distance < y.distance;
 });

    Mat Image_with_Matchers;
    drawMatches(equalizedcorruptedImage, keypoints, patches[k], patches_keypoints[k],
        vector<DMatch>(new_matches.begin(), new_matches.begin() + 20), Image_with_Matchers);

    //Displaying top matches using i choose to display the first 20 matches of each patch and the corr_image
    String titre = " Matches between patches and the corrupted Image number " + to_string(k) ;
    namedWindow(titre, WINDOW_NORMAL);
    //Let's adjust the window size
    resizeWindow(titre, 800, 600);
    moveWindow(titre, 150, 150);
    imshow(titre, Image_with_Matchers);
```

- Next, match the descriptors of the image and the patch by using the "matcher.match()" function.
- Sort the matches based on their distance using the "sort()" function.

- Finally, display the top matches using the "drawMatches()" function. In this case, we are displaying the first 20 matches of each patch and the image.

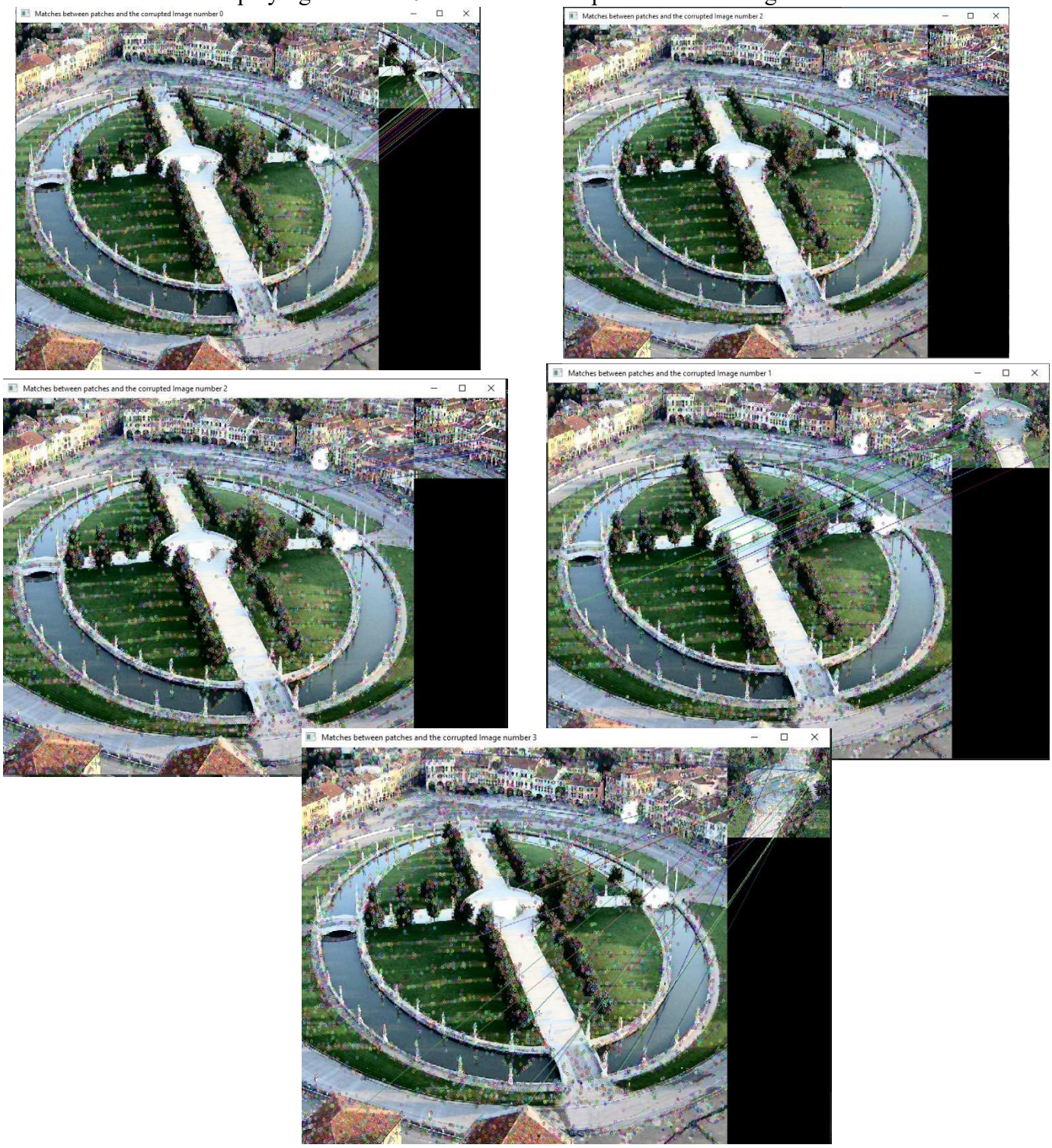


Figure 10 : Match between Image and Patches Using SIFT

b. Refine the matches

- To improve the matches, we refine the previously obtained matches by selecting only those matches with a distance less than a user-defined threshold, which is the ratio multiplied by the minimum distance among all the matches.

```
// refine the matches based on distance ratio
vector<DMatch> refineMatches;
double ratio = 1.5;  // defining the distance ratio , i start with 0.8 but i m getting an error that the destinationPoints and Sour
// is under 4 , so i can't calculate the Homography matrix , thats why i try to get a high ratio

//This part i get it from stackoverflow because i was getting lots of errors
for (int i = 0; i < new_matches.size(); i++)
{
    if (new_matches[i].distance <= (new_matches[0].distance * ratio))
    {
        refineMatches.push_back(new_matches[i]);
    }
}

matches_vector.push_back(refineMatches);

// Display filtered matches using drawMatches
Mat new_Image_with_Matchers;
drawMatches(equalizedcorruptedImage, keypoints, patches[k], patches_keypoints[k],refineMatches, new_Image_with_Matchers);

//Displaying top matches using i choose to display the first 20 matches of each patch and the corr_image
String titre1 = " The new matches " + to_string(k);
namedWindow(titre1, WINDOW_NORMAL);
resizeWindow(titre1, 800, 600);
moveWindow(titre1, 150, 150);
imshow(titre1, new_Image_with_Matchers);
```

- I start by Creating an empty vector refineMatches to store the refined matches.
- Set the ratio threshold to 1.5, which is a user-defined value.
- Iterate through each match in the new_matches vector.
- Check if the distance of the current match is less than or equal to ratio * min_distance, where min_distance is the minimum distance among all the matches.
- If ratio * min_distance is satisfied, add the match to the refineMatches vector.
- After after we store the refineMatches vector in the matches_vector.
- Create a new Mat named new_Image_with_Matchers to display the new matches.
- Using the drawMatches function to draw the refined matches on the equalizedcorruptedImage, keypoints, patches[k], and patches_keypoints[k]. After, creating a named window to display the new refine matches

Figure 11 : The refine matches

6. Step5: RANSAC algorithm:

- At this point we gone assume that there is an affine transformation between the images and patches, and we can utilize the refined matches to determine the transformation.
- Here we introduce the RANSAC algorithm, that it can be employed for this purpose. By applying the findHomography() function with CV_RANSAC as the third parameter, we can obtain the set of inliers. The mask argument can be used to retrieve the inliers from the computation. This is the steps to calculate the homography matrix:
  - Create two vectors, "source" and "destination", to store the corresponding points from the refined matches.
  - Iterate over each refined match found in the previous step, and for each match, add the point from the "keypoints" vector (image) to the "source" vector, and add the point from the "patches_keypoints" vector (patch) to the "destination" vector.

- Finaly , I use the findHomography() function with the "destination" and "source" vectors as inputs, along with the RANSAC algorithm specified.

```cpp
vector<Point2f> source;
vector<Point2f> destination;
for (const DMatch& match : refineMatches)
{
    source.push_back(keypoints[match.queryIdx].pt);
    destination.push_back(patches_keypoints[k][match.trainIdx].pt);
}


//cout << "The source points size: " << source.size() << endl;
//cout << "The destination points size: " << destination.size() << endl;

/*
if (source.size() != destination.size()) {
    cout << "Error: srcPoints and dstPoints should have the same size." << endl;
    return -1;
}
*/

Mat homography_matrix = findHomography(destination, source, RANSAC);
cout << "Homography Matrix:\n" << homography_matrix << endl;
```

```
-0.003378577060131456, 1.031810868597713, 217.7014552416846;
-1.079480813450947e-05, 6.567810591548286e-05, 1]
OpenCV: terminate handler is called! The last OpenCV error is:
OpenCV(4.7.0-dev) Error: Unknown error code -28 (The input arrays should have at least 4 corresponding point sets to calculate Homography) in cv::fin
dHomography, file C:\GHA-OCV-2\_work\ci-gha-workflow\ci-gha-workflow\opencv\modules\calib3d\src\fundam.cpp, line 385

C:\Users\21629\Desktop\cours\S2\secure digital healthcare\New folder\ComputervisionHW\x64\Release\ComputervisionHW.exe (process 27608) exited with co
```

At this point, I encountered a problem while calculating the Homography matrix due to the size mismatch between the source and destination vectors used to store the corresponding points from the refined matches. To address this issue, I attempted to improve the image quality through preprocessing steps. Additionally, I set a high ratio value (although this may result in more matches being included in the refined matches, it can potentially reduce precision)

```
The source points size: 19
The destination points size: 19
The source points size: 6
The destination points size: 6
The source points size: 99
The destination points size: 99
The source points size: 1686
The destination points size: 1686
```

- The size of the source and destination point sets for the findHomography function should ideally be 4 or more.

```
Homography Matrix:
[0.9890325833757063, 0.02703779683403149, 842.8464893326111;
 -0.004504602394943803, 1.011253285319104, 218.8063449374075;
 -1.300615244991349e-05, 2.799374420978466e-05, 1]
Homography Matrix:
[1.366030028918534, 0.003799246177814822, 440.6302478274388;
 0.2354173146335986, 1.014719792507201, 271.1369085533221;
 0.0005611868124886399, -1.159051692170721e-05, 0.9999999999999999]
Homography Matrix:
[0.9870974267165992, -0.006259630632764973, 851.3679225093857;
 -0.001103168035487715, 0.9968329035671587, 44.1795075983196;
 -1.012341699731666e-05, -6.854477538681902e-06, 1]
Homography Matrix:
[-2.474159042681783, 5.474737524190457, 414.7707597349892;
 -2.021902493620804, 8.432476655921219, 102.7053774511;
 -0.00358007348427135, 0.006754645395039046, 1]
```

Figure 12 : The Homography Matrix

Step 6 : Overlay the patches over the image:

Once the homographies have been calculated, it can be utilized to overlay the patches onto the image and correct the corrupted regions. The homography matrix describes the transformation between the source and destination points, allowing for precise alignment. By warping the patches using the homography matrix and then blending them with the original image, the corrupted regions can be seamlessly replace homography matrix and then blending them with the original image, the corrupted regions can be seamlessly replace seamlessly replace homography matrix and then blending them with the original image, the corrupted regions can be seamlessly replace regions can be seamlessly replace seamlessly replace homography matrix and then blending them with the original image, the corrupted regions can be seamlessly replace.

Figure 13 : The fixed images

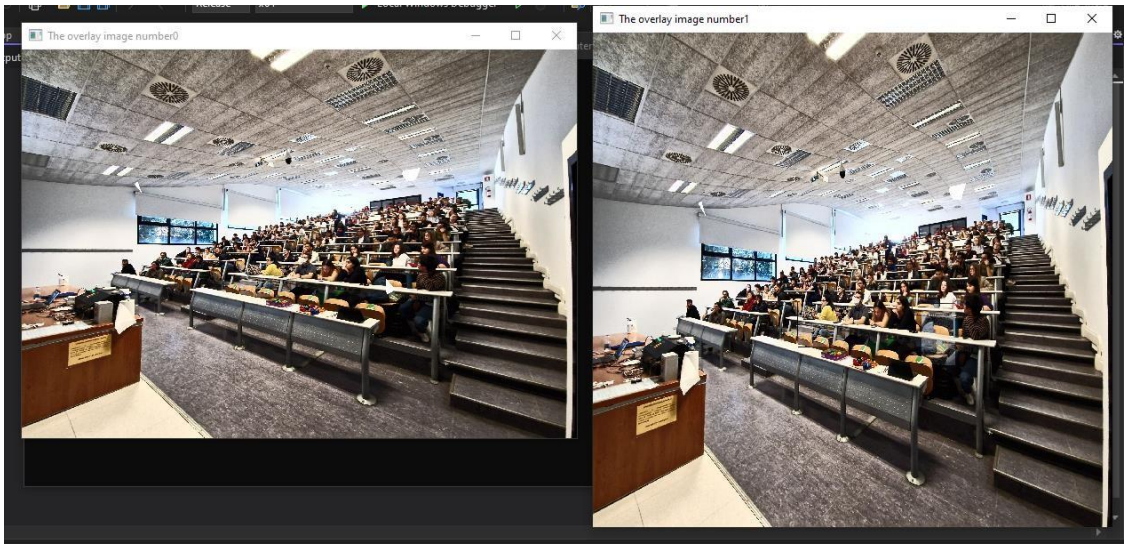
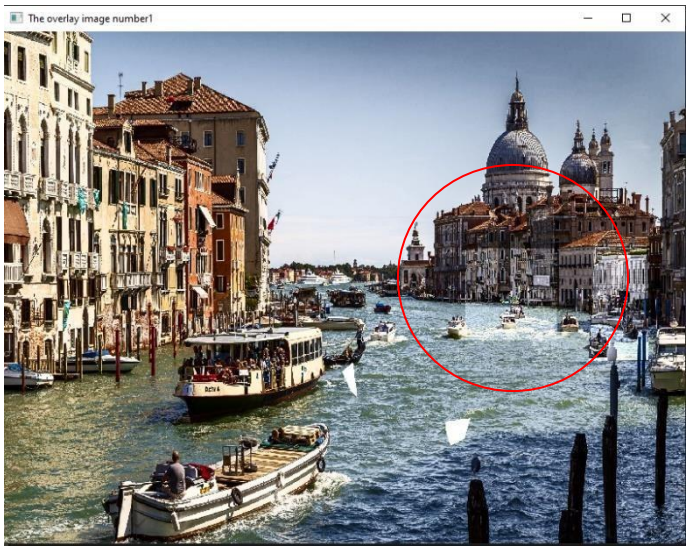
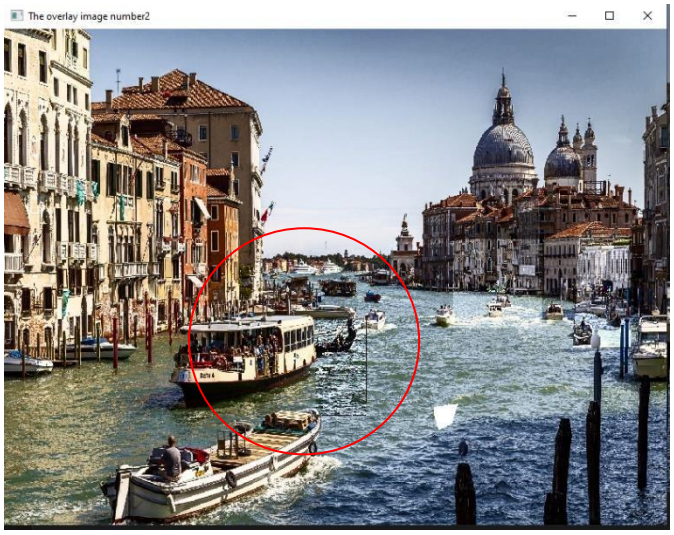


Figure 14 : The fixed images international

- ***Trying the Venezia images:***


The overlay image number1


The overlay image number2

```
Homography Matrix:
[1.050636427387188, -0.9921645866175662, 1666.360854362366;
 0.02619004088356584, 0.6809377584104214, 449.0586089021895;
 4.789700462863612e-05, -0.0005554425557921527, 1]
Homography Matrix:
[1.010218547986647, -0.01077746927462324, 1301.046567235365;
 0.004620490140402484, 0.9945146214127035, 458.0673908541486;
 7.604646002874167e-06, -7.911527373628231e-06, 1]
Homography Matrix:
[1.001452913170374, -0.002592637729065817, 902.092513851667;
 0.001151609569760579, 0.9983615289141633, 630.9891840785695;
 1.732583717138536e-06, -2.350908813369445e-06, 1]
Homography Matrix:
[0.9886225610570489, 0.001836563596398855, 1181.029364556582;
 -0.006965452408988881, 1.00000513541483, 757.0546904297028;
 -8.45539435763819e-06, 8.607936360680913e-07, 1]
Homography Matrix:
[-253.1339173759086, -187.5435320159635, 2645.730657001981;
 -98.06954667102839, -11.96673256151845, 248.0241906608811;
 -0.2292706367570922, -0.01961983807536269, 1]
Homography Matrix:
[-8.056599708922546, -11.6973190710261, 1400.11779431469;
 -3.614440379012739, -5.844295299709324, 675.4055678948228;
 -0.005306839595769499, -0.008843268101015017, 1]
Homography Matrix:
[114.6424963452005, -75.89352170705565, 781.3100746496424;
 71.23352923618977, -47.1780037841645, 484.2938710988964;
 0.04295308641418395, -0.03650043204687854, 1]
Homography Matrix:
[-16.59315645034178, 4.342595599480014, 1233.561828218006;
 -10.48568521389352, 2.921004248446162, 770.1360914380448;
 -0.01368988020219577, 0.003834487046751235, 1]
```

Figure 15 : The Homography Matrix of Venezia

Conclusion

In this project, a C++ software was developed to address the task of image completion using feature descriptors. The objective of the software was to automatically find and correct missing or corrupted areas in an input image by utilizing a set of provided patches containing the missing information.

Annex: all the keypoints on the corrupted images

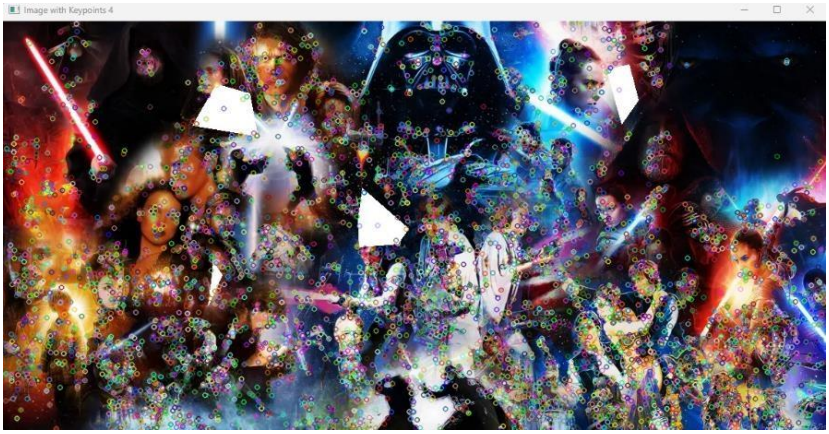


Image with Keypoints 4