Università degli Studi di Padova

ICT for Internet and Multimedia

COMPUTER VISION

LAB-4 PROJECT REPORT

IMAGE COMPLETION WITH FEATURE DESCRIPTORS

Written By

Umut Berk CAKMAKCI
2071408

Supervisor
Prof. Pietro ZANUTTIGH

June 14, 2023

Contents

1 Introduction

Image completion using feature descriptors is a technique used to fill in missing or damaged parts of an image by leveraging the information from the surrounding regions. Feature detection algorithms aim to identify regions or points in an image that possess certain characteristics, such as edges, corners, blobs, or texture variations.

Feature descriptors, such as SIFT (Scale-Invariant Feature Transform), SURF (Speeded-Up Robust Features) or ORB (Oriented FAST and Rotated BRIEF), are used to describe local features in an image, which can then be matched to find corresponding features in other parts of the image or in another image.

Image completion with feature descriptors is just one approach among many for image inpainting. Other techniques, such as deep learning-based methods using generative models like GANs (Generative Adversarial Networks) or CNNs (Convolutional Neural Networks), have gained popularity in recent years due to their ability to learn complex image structures and produce high-quality completions.

We are going to use some of feature detection techniques which they can work on different scales, transforms and even in mirrored images such as SIFT, SURF, GFTT, FAST, AKAZE, ORB, and BRISK. Also we are going to use some works done in the past years on this subject [1][2][3].

2 Main Tasks

In this study, we are required to do feature detection, extraction and matching between 2 images using some algorithms, find the 'good' matches between them, calculate the homography and overlay the patch images onto the image needs to be completed with the coordinates we found in homography.

2.1 Feature Detection

Feature detection algorithms aim to find salient points that stand out from their surroundings and are robust to variations in scale, rotation, lighting conditions, and viewpoint changes. Some popular feature detection algorithms include Harris corner detector, FAST (Features from Accelerated Segment Test), Difference-of-Gaussians (DoG), or Maximally Stable Extremal Regions (MSER).

2.2 Feature Extraction

The goal in this process is to extract meaningful information or descriptors from the detected features. This extraction may involve quite considerable amounts of image processing. The result is known as a feature descriptor or feature vector. Examples of feature extraction methods include SIFT, SURF, ORB, or HOG (Histogram of Oriented Gradients).

2.3 Feature Matching

The goal of feature matching is to establish correspondences between features in different images, allowing for various analysis and understanding of the visual data. Various techniques can be used to match features between different images, such as Nearest Neighbor, K-nearest Neighbors or RANSAC.

3 Studies

3.1 Theory

At the beginning of the project, I used SIFT algorithm for feature detection and extraction with the parameters in Table 1. After that, I used BFMatcher for feature mathing with the parameter NORM-L2. We don't need all the matches we found, so we got 'best' matches by doing ratio*min-distance to eliminate most of the matches and get the good ones. Then, with the good matches we found, I calculated the coordinates for 2 pictures to align best by using findHomography function. With these coordinates, we overlay the patch image onto incomplete image in the missing part. To do that, we used warpPerspective function.

	SIFT
ratioThreshold	0.8
distanceThreshold	100
nFeatures	0
nOctaveLayers	3
contrastThreshold	0.08
edgeThreshold	10
sigma	2

Table 1: SIFT Algorithm Parameters

The SIFT algorithm worked well with the normal images, scaled images and rotated images. However, it doesn't work well with the mirrored images. So I start searching what is the best algorithm for mirrored images and I found a study about this topic. In this study, they compared 10 feature detection algorithm including SIFT in the horizontally reflected (mirrored) images [1].

I didn't use all the algorithms they compare but I choose FAST and GFTT (HARRIS is already included in GFTT algorithm in the OpenCV). I also tried AKAZE and ORB algorithms for mirrored images.

```
//Ptr<SIFT> detector = SIFT::create();
//Ptr<FastFeatureDetector> detector = FastFeatureDetector::create();
//Ptr<AKAZE> detector = AKAZE::create(AKAZE::DESCRIPTOR_KAZE, 1, 3, 0.001, 4, 4, KAZE::DIFF_PM_G2);
//Ptr<GFTTDetector> detector = GFTTDetector::create(100, 0.02, 1, 3, true, 0.04);
Ptr<SIFT> extractor = SIFT::create();
```

Figure 1: Feature Detection Algorithms

For alternative working, I focused on my working to AKAZE algorithm. After long trials, I found the optimum parameters for AKAZE algorithm, which they are in Table 2.

I tried to use FlannBasedMatcher instead of BFMatcher and used another matcher function, which is called knnMatch (Figure 2). SIFT and GFTT algorithms work with the L2-norm and AKAZE and ORB algorithms work with the Hamming distance [3][4].

I also tried first getting the edges of images using Canny edge detector manually and after that try to get the features from the edges. It does work good on the normal images but didn't work out with mirrored image.

	AKAZE
ratioThreshold	0.7
distanceThreshold	200 (for transposed images)
	120 (for normal images)
descriptor-type	AKAZE::DESCRIPTOR-KAZE
descriptor-size	1
descriptor-channels	3
threshold	0.001
nOctaves	4
nOctaveLayers	4
diffusivity	KAZE::DIFF-PM-G2

Table 2: AKAZE Algorithm Parameters

```
// Step 3: Match detected keypoints between 2 images
FlannBasedMatcher matcher;
//BFMatcher matcher(NORM_L2);
//BFMatcher matcher(NORM_HAMMING);

vector<DMatch> matches;
matcher.match( queryDescriptors: descriptors1, trainDescriptors: descriptors2, [&] matches);

vector<std::vector<DMatch>> matches2;
matcher.knnMatch( queryDescriptors: descriptors1, trainDescriptors: descriptors2, [&] matches2, k: 2);
```

Figure 2: Feature Matching Code Snippet

Some of the patch images have some noise, mostly salt-pepper noise. So before do the processes, I needed to filter the images using MedianBlur or BilateralFilter. Bilateral filter did a good job than Median blur, it cleared the images with preserving most of the edges in the image. Also I used another function rather than standart blurring. Firstly, I convert BGR to YUV image, split the image into channels and filter only the channel 0. After filtering process, I merged the channel back and return the final filtered image.

```
Mat yuvImage;
cvtColor( inputImage, yuvImage, code: COLOR_BGR2YCrCb);
vector<Mat> yuvChannels;
split( yuvImage, yuvChannels);
Mat denoisedChannel;
Mat denoisedChannel2;
bilateralFilter( yuvChannels[channelIndex], denoisedChannel, d: kernelSize, sColor, sSpace);

// Replace the original color channel with the denoised channel
yuvChannels[channelIndex] = denoisedChannel;
//yuvChannels[2] = denoisedChannel2;

Mat outputYUVImage;
merge( yuvChannels, outputYUVImage);

Mat outputImage;
cvtColor( outputYUVImage, outputImage, code: cv::COLOR_YCrCb2BGR);
```

Figure 3: Denoise Color Channel

The parameters for denoising vary for each image and you can see the optimum parameters for each image I found in the below code snippet.

```
// Apply the noise filter on the specified color channel
Mat f_incompImage = denoiseColorChannel(r_incompImage, channelIndex: 0, kernelSize: 5, sColor: 15, sSpace: 20);
Mat f_patch_t_1 = denoiseColorChannel(flippedt1, channelIndex: 0, kernelSize: 8, sColor: 45, sSpace: 15);
Mat f_patch_t_0 = denoiseColorChannel(patch_t_0, channelIndex: 0, kernelSize: 9, sColor: 45, sSpace: 25);
Mat f_patch_t_2 = denoiseColorChannel(patch_t_2, channelIndex: 0, kernelSize: 8, sColor: 45, sSpace: 45);
```

Figure 4: Denoising Function Parameters

Some of the images in datasets are too big that it extends the screen of pc and failed to detect features in the image. So before detection, I need to resize the image and after that perform the feature detection operation.

In addition, I tried to do the matching function without using any kind of algorithms I mentioned. For this, I used Template Matching method. This method worked well with the normal images but it failed to overlay the rotated, scaled or mirrored images. It requires a pre-operation, flipping or rotation, before doing the template matching.

```
// Methods of Template Matching: "SQDIFF", "SQDIFF NORMED", "TM CCORR", "TM CCORR NORMED", "TM COEFF", "TM COEFF NORMED";
matchTemplate( image: img, templ, result, method: TM_CCOEFF_NORMED);
normalize( src: result, result, alpha: 1, beta: 0, norm_type: NORM_L2, dtype: -1, mask: Mat());
minMaxLoc( src: result, &minVal, &maxVal, &minLoc, &maxLoc, mask: Mat());

matchLoc = maxLoc;
//rectangle(img_display, matchLoc, Point(matchLoc.x + templ.cols-2, matchLoc.y + templ.rows-2), Scalar(255, 255, 255, alpha), 2, 8, 0);
//rectangle(result, matchLoc, Point(matchLoc.x + templ.cols, matchLoc.y + templ.rows), Scalar(255, 255, 255), 2, 8, 0);
```

Figure 5: Template Matching Function

I added another function for good matches, to find better matches. To do that, first I did ratio*min-distance and after that, I removed the mathces that exceeds the distance threshold I defined. By adding extra elimination method, I aim to eliminate possible irrelevant matches.

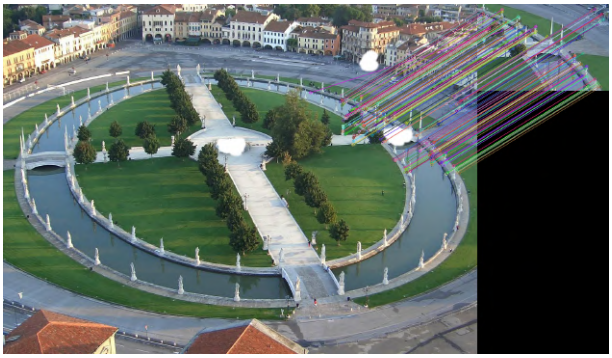
```
// Apply ratio test
for (size_t i = 0; i < matches.size(); i++)
{
    if (matches2[i][0].distance < ratio * matches2[i][1].distance)
    {
        good_matches2.push_back(matches2[i][0]);
    }
}

// Elimination with Distance Threshold
std::vector<DMatch> filteredMatches;
for (const auto& match : good_matches2)
{
    if (match.distance < distanceThreshold)
    {
        filteredMatches.push_back(match);
    }
}
```

Figure 6: Ratio Test and Distance Threshold

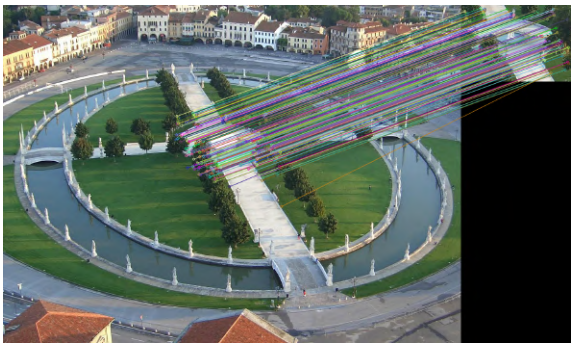
3.2 Results

3.2.1 Using SIFT Algorithm

The first results were obtained with the SIFT algorithm, BFMatcher using NORM-L2 distance parameter. The ratio test parameter was 3 and the distance threshold was not added yet. Next step was calculating the homography with findHomography function and using this result, overlaying the images onto one another with warpPerspective function.



(a) Incomplete and Patch-0 Image Matches



(b) Incomplete and Patch-1 Image Matches

(c) Incomplete and Patch-2 Image Matches

(d) Merged Image with Normal Images

Figure 7: SIFT with Normal Images

The SIFT algorithm did good job on regular images, also scaled and rotated images. The only problem in SIFT was mirrored images. When it comes to mirrored images, the algorithm couldn't find enough proper matches.

(a) Incomplete and Patch-t-0 Image Matches

(b) Incomplete and Patch-t-2 Image Matches

(c) Incomplete and Patch-t-1 Image Matches

Figure 8: SIFT with Transposed Images

But if we flipped the image horizontally before processing it, we got better results and we can overlay the image onto incomplete image.

(a) Incomplete and Patch-t-1 Image Matches

(b) Merged Image with Transposed Images

Figure 9: SIFT with Transposed Images

3.2.2 Using Other Algorithms (FAST, GFTT)

In the previous part, I mentioned that SIFT algorithm works good on scaled and rotated images, but not in mirrored images. After some research, I found some algorithms that could work on mirrored images [1]. I decided to use 2 of them; FAST and GFTT. Also I figured out that GFTT algorithm uses Harris Corner detector inside so we could get better results if we use them both. On the contrary to the study, these algorithms fail to detect much useful features. The results are shown below.

(a) Incomplete and Patch-t-0 Image Matches (b) Incomplete and Patch-t-1 Image Matches

Figure 10: Image Matches using FAST Algorithm

(a) Incomplete and Patch-t-0 Image Matches (b) Incomplete and Patch-t-1 Image Matches

Figure 11: Image Matches using GFTT Algorithm

These results may be affected from the patch images because the patch images were noisy and I did some filtering process for getting more clear images. But also I figured out that these 2 algorithms were not effective in scaled and rotated images. They require pre-operation (like scaling, rotating, flipping) more than SIFT or AKAZE.

3.2.3 Using AKAZE Algorithm

Compared with SIFT algorithm, AKAZE algorithm has a faster calculation speed. Compared with ORB and Brisk algorithm, its repeatability and robustness are also significantly improved [3][5]. With the normal images, AKAZE is effective as much as SIFT or any other method but it is faster than SIFT.

(a) Incomplete and Patch-0 Image Matches

(b) Incomplete and Patch-1 Image Matches

(c) Incomplete and Patch-2 Image Matches

(d) Merged Image with Normal Images

Figure 12: AKAZE with Normal Images

The AKAZE algorithm works better than SIFT in the mirrored images where SIFT didn't find any correct matches with the flipped image, AKAZE finds 3 correct matches. But these correct matches weren't enough for flipping the image because there were other wrong matches in the picture.

Figure 13: Incomplete and Patch-t-1 Image Matches without Flipping

In this case, I need to use flip function to flip horizontally the mirrored image to get the results. Next images will show the better results with filtering and flipping processes included.

(a) Incomplete and Patch-t-0 Image Matches

(b) Incomplete and Patch-t-1 Image Matches

(c) Incomplete and Patch-t-2 Image Matches

(d) Merged Image with Transposed Images

Figure 14: AKAZE with Transposed Images

3.2.4 Using Template Matching Method

Template matching method is very successful with locating the object's correct position in the image. When we try to point out the normal patches, this method correctly draws a frame around the location where the patch should be placed. I didn't show each patch but the final result is below. As you can see fromm the result, this method overlay the patch images onto incomplete one with precision.

Figure 15: Merged Image Using Template Matching Method

However, the this method does not work well with transformed images, like rotated or scaled. It locates the objects in the image but it doesn't have rotation or scaling capabilities so it just placed the patch incorrectly. In general, before matching processes, it requires some additional operations like rotation or scaling or flipping the image.

4 Conclusion

In this study, we try to use different feature detecting algorithms on different images with different transposes. We try to use SIFT as the most recommended one and we see that it worked out good with normal, scaled and rotated images. But we see that it didn't deal with the transposed image by itself so it required a pre-process, which is flipping.

Also we saw that the noisier the image, the difficult to detect the features. In our dataset, we had some noisy images so we applied some denoising functions. We prefer to use Bilateral Filter because it is effective against salt-pepper noises and it is a good filtering method to preserve the details in the image while decreasing the noise.

As an alternative, we use Template Matching method. This method also works good with normal images but when it comes to detect and place transposed images, it didn't work good as SIFT.

After doing some research, I detect 3 feature detector algorithm as an alternative to SIFT; FAST, GFTT with Harris and AKAZE. With these algorithms, we tried to use FlannBasedMatcher instead of BFMatcher and with that, knnMatch to improve our results. FAST and GFTT was successful in normal images but failed in transposed images. AKAZE, on the other hand, did good job with transposed images and even with the flipped image. It didn't successfully place the patch in the image but it managed to detect 3 correctly matched feature while SIFT didn't detect any correct match. AKAZE also required flipping operation but it is still better than SIFT.

SIFT and AKAZE are robust and accurate feature detection algorithms but can be slower. FAST and GFTT are faster algorithms but may be less robust. Also FAST and GFTT have failed to detect mirror image features. So they didn't seem correct algorithms for this dataset.

Between the SIFT and AKAZE; SIFT and AKAZE are robust feature detection and description algorithms, with SIFT being accurate but slower and AKAZE offering a faster alternative while maintaining good performance. So looks like the best option for this dataset is AKAZE.

References

[1] Henderson, C. and Izquierdo, E., 2016. Symmetric stability of low level feature detectors. Pattern Recognition Letters, 78, pp.36-40.

[2] Su, M., Ma, Y., Zhang, X., Wang, Y. and Zhang, Y., 2017. MBR-SIFT: A mirror reflected invariant feature descriptor using a binary representation for image matching. PloS one, 12(5), p.e0178090.

[3] S. A. K. Tareen and Z. Saleem, "A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK," 2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), Sukkur, Pakistan, 2018, pp. 1-10, doi: 10.1109/ICOMET.2018.8346440.

[4] `https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html`

[5] Xu, J., Ji, X. and Wang, Y. (2021) 'A Matching Algorithm T-AKAZE for Image Recognition of Hydroelectric Equipment Failure', Journal of physics [Preprint]. Available at: https://doi.org/10.1088/1742-6596/2010/1/012003.