University of Burgundy

Assignment I

Corner Detection & Feature Tracking

Kimberly Grace Sevillano Colina

1 Introduction

In this report, we aim to implement a Harris corner detector for detecting and tracking corners in image sequences, and to compare the performance of different feature descriptors, such as patch templates and SIFT. The assignment is divided into four parts: corner detection, corner tracking with patch templates, corner tracking with SIFT features, and corner tracking with prior motion fitting model (optional). The dataset used in this assignment is the first 200 frames of the popular KITTI Visual Odometry dataset [1], which provides a sequence of grayscale images to test our corner detection and tracking implementations.

2 Part I - Environment Setup & Inputs

2.1 Environment Setup

To set up the conda environment for working on the local machine, follow the steps below:

1. Create a new conda environment: `conda create -n myharris_track`
2. Activate the environment: `conda activate myharris_track`
3. Install Jupyter Notebook: `conda install jupyter`
4. Navigate to the workspace directory of the course and run Jupyter: `jupyter - notebook`

2.2 Input Images

The input images for this assignment are taken from Sequence 00 of the KITTI Visual Odometry dataset. The first 200 images are provided in the `images/` folder within the `CV Assignment 1 CornerTracker.zip` file. In addition, a video file (`corner tracking patch.mp4`) is provided to demonstrate the expected correspondence outputs for the last 100 frames. Visualization functions for plotting the corresponding corners between two frames and several hints are also included to assist with the assignment.

3 Part II - Corner Detection

3.1 Gaussian Kernel

The Gaussian kernel is a symmetric kernel function that is used to smooth a signal or an image. It is defined as:

$$G_{(x,y)} = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

Where σ is the standard deviation of the kernel and x and y are the coordinates of the pixels in the image. When a Gaussian kernel is applied to an image, the effect is similar to passing a mean filter, but with the additional advantage that the Gaussian kernel preserves the edges of the image better. As the standard deviation σ is increased, the Gaussian kernel becomes smoother, and its smoothing effect is intensified.

The implementation of the Gaussian kernel function is as follows:

Algorithm 1 Gaussian Kernel

INPUT: h (size of the kernel), sigma (standard deviation)
OUTPUT: kernel (2D numpy array containing the Gaussian kernel)

Initialize kernel ← np.zeros((h, h))
Compute denom ← $2 * \pi * \sigma^2$
Create samples ← np.arange(-int(h/2), int(h/2) + 1)
for all i, j in samples **do**
    x ← samples[i]
    y ← samples[j]
    Compute num ← $\exp(-1 * ((x^2 + y^2)/(2 * \sigma^2)))$
    Compute val ← $\frac{num}{\sqrt{denom}}$
    kernel[i][h - j - 1] ← val
end for
Normalize kernel ← kernel / kernel.sum()
return kernel

(a) (b)

Figure 1: In (a), we can observe the output of the gaussian_kernel function with a kernel size of 3 and sigma value of 0.1, while in (b), the output is shown with a kernel size of 5 and sigma value of 1.

3.2 Gradient Image

The gradient_image function computes the gradient of a given input image using the Gaussian kernel for smoothing and the Sobel operators for detecting the gradients in the x and y directions. The gradient of an image represents the rate of change in intensity or color.

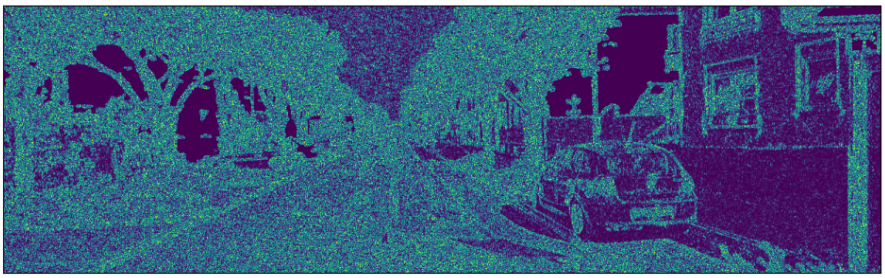


Figure 2: This is how it looks without any weighting to the Sobel coefficients, with a lot of "noise".

The implementation of the gradient_image function is as follows:

Algorithm 2 Gradient Image

INPUT: img - input grayscale image, gs - Gaussian kernel
OUTPUT: IxIx - elementwise product of the x-gradient with itself,
IyIy - elementwise product of the y-gradient with itself,
IxIy - elementwise product of the x-gradient and y-gradient

```
Apply Gaussian smoothing to the input image:
        smoothed_img = convolve(img, gs)
Define the Sobel operators:
        sobel_x = (1/128) * np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
        sobel_y = (1/64) * np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
Compute x-gradient and y-gradient of the smoothed image:
        Ix = convolve(smoothed_img, sobel_x)
        Iy = convolve(smoothed_img, sobel_y)
Compute elementwise products of the gradients:
        IxIx = Ix * Ix
        IyIy = Iy * Iy
        IxIy = Ix * Iy
return IxIx, IyIy, IxIy
```

In the notebook, you can see that different combinations of Sobel filters were tested, as shown in the commented lines. This experimentation helps to find the best balance between detecting vertical and horizontal lines in the image. Ultimately, the chosen Sobel filters, sobel_x and sobel_y, are used to calculate the x and y gradients, respectively. The function then returns the element-wise products of the gradients, IxIx, IyIy, and IxIy, which can be used for further processing or analysis of the image.

Figure 3: Plotting the gradient function with the final weights chosen for the Sobel coefficients.

3.3 Harris Response

The Harris Corner Detector algorithm uses the Harris corner score (also known as the Harris response) to determine whether a particular pixel in an image corresponds to a corner or not.

The **Harris corner score** is calculated based on the second moment matrix, which is a 2x2 matrix that describes the local image structure in the vicinity of the pixel. The second moment matrix is defined as:

$$M = \sum_{x,y} w\,(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \text{ (Harris Corner Detector Matrix)} \tag{1}$$

where I_x and I_y are the partial derivatives of the image intensity with respect to the x and y axes, respectively. Using the second moment matrix, the Harris corner score is calculated as:

$$R = det(M) - k \cdot (trace(M))^2 \tag{2}$$

where $det(M)$ and $trace(M)$ are the determinant and trace of the second moment matrix, respectively. The parameter k is an empirical constant that controls the sensitivity of the detector.

The Harris corner score R is a measure of the corner response at the given pixel. If R is large, then the pixel is likely to be part of a corner, while if R is small, then the pixel is part of a flat region.

Algorithm 3 Harris Response

INPUT: IxIx (x-derivative of the image), IyIy (y-derivative of the image), IxIy (product of x- and y-derivatives of the image), w (Gaussian kernel for smoothing)
OUTPUT: R (Harris corner response for each pixel in the image)

1: Compute g_dx2 ← convolve(IxIx, w)
2: Compute g_dy2 ← convolve(IyIy, w)
3: Compute g_dxdy ← convolve(IxIy, w)
4: Compute R ← g_dx2 * g_dy2 - (g_dxdy)^2 - 0.06 * (g_dx2 + g_dy2)^2
5: **return** R

3.4 Non-Maximum Suppression

The non-maximum suppression (NMS) algorithm compares the score of each pixel or feature to its neighboring pixels or features, keeping only the ones with the highest score in the local neighborhood. This can be implemented using various methods, such as sorting, thresholding, or clustering. In the context of Harris corner detection, a simple implementation of NMS involves selecting pixels with a score higher than a certain threshold and then selecting the local maxima within a certain window size. The threshold and window size can be adjusted based on the characteristics of the image and the desired level of accuracy.

Here is the non-maximum suppression function in the algorithm format:

Algorithm 4 Non-Maximum Suppression

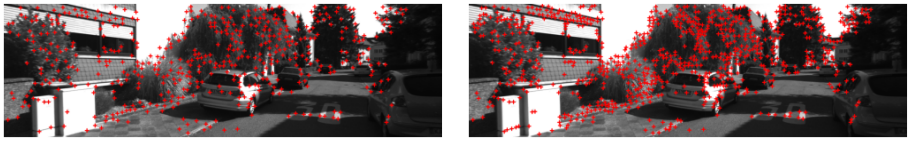
INPUT: R (Harris corner response map), window_size (size of the sliding window), threshold (minimum threshold for a pixel to be considered a corner)
OUTPUT: keypoints (array of corner points after non-maximum suppression)

1: Find local maxima of R using peak_local_max, store in keypoints ← peak_local_max(R, min_distance = window_size // 2, threshold_rel = threshold)
2: Compute the global maximum of R ← max_response = np.max(R)
3: Compute the threshold value ← threshold_value = max_response * threshold
4: Filter keypoints based on threshold value ← keypoints = keypoints[R[keypoints[:, 0], keypoints[:, 1]] ≥ threshold_value]
5: **return** keypoints

3.5 Keypoint Visualization

In this section, the `keypoints_vis` function visualizes detected corners in the images by overlaying red $'+'$ markers on a grayscale image. The function takes the image and a list of keypoints' coordinates as inputs. It converts the grayscale image into a three-channel color image, draws red $'+'$ markers at the keypoint coordinates, and displays the resulting image. This visualization provides a clear representation of the corner detection results.

When adjusting the parameters in the `non_max_suppression(R_harri, 20, 0.05)` function, the number and distribution of detected keypoints change. Increasing the window size to 20 reduces the total keypoints, but raises keypoints per area. Balancing these parameters optimizes performance and efficiency for specific applications.



(a) (b)

Figure 4: In (a), we can visualize the plotting of the keypoints with w =20 and in (b) with a w=15.

4 Part III - Corner Tracking with Patch Templates

4.1 Patch Extraction

The patch extraction function, `get_keypoint_patches`, is designed to extract square patches of size N centered around each keypoint from an input image. The function accepts an image, a list of keypoints, a patch size (N), and a small value (eps) to avoid division by zero when normalizing patches.

To handle keypoints near the image edges, the input image is first padded with a border of width `N//2` using the `cv2.copyMakeBorder` function. The keypoints' coordinates are then adjusted to account for the added border, and the N x N patches are extracted from the padded image. Each patch is subsequently normalized by subtracting the mean and dividing by the standard deviation (plus the small eps value to prevent division by zero).

Finally, the function returns an array of normalized patches as numpy arrays, providing a representation of the local image structure around each keypoint. Example of patch extraction from an image is shown as follows:

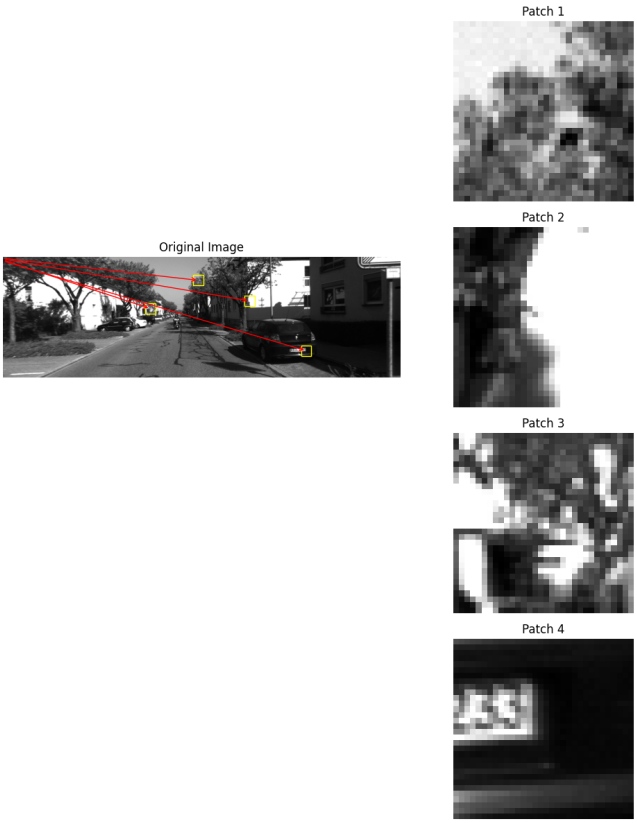


Figure 5: In the original image you can see the location of the extracted patches (shown on the right).

4.2 Distance Matrix

In this section, we describe a function that calculates the distance matrix between two sets of image patches. The purpose of this matrix is to compare the similarity between patches using the Sum of Squared Differences (SSD) as a similarity measure. **Similarity measures** The Sum of Squared Differences (SSD) is a simple similarity measure used to compare two patches of an image. It measures the sum of the squared pixel differences between two patches, and a smaller value indicates a greater similarity. The formula for computing the SSD between two patches A and B of the same size is:

$$SSD_{(A,B)} = \sum_{(i,j)} (A_{(i,j)} - B_{(i,j)})^2$$

Here, $A_{(i,j)}$ and $B_{(i,j)}$ are the pixel values of the patches at position (i, j).

To compare two sets of patches P_1 and P_2, we can create a distance matrix D of shape (M_1, M_2), where M_1 and M_2 are the number of patches in P_1 and P_2, respectively. The element $D(i, j)$ in the matrix represents the distance (SSD) between patch i in P_1 and patch j in P_2.

The implementation for computing the distance matrix function is as follows:

Algorithm 5 Distance Matrix

INPUT: patches1 (first array of patches), patches2 (second array of patches)
OUTPUT: D (distance matrix between the two arrays of patches)

```
Get M1, N, _ ← shape of patches1
Get M2, _, _ ← shape of patches2
Initialize D as a zero matrix of size (M1, M2)
for i in range(M1) do
    for j in range(M2) do
        Compute dist ← sum of squared differences between patches1[i,:,:] and patches2[j,:,:]
        Set D[i,j] ← dist
    end for
end for
return D
```

4.3 Finding Matches

In this section, we discuss the *find_matches* function, which is used to establish correspondences between keypoints of two successive frames. The function takes the distance matrix D as input, which is computed using the *distance_matrix* function we discussed earlier. The distance matrix contains the similarity measure (SSD) between all pairs of patches from two images.

The *find_matches* function returns an array of matched keypoints, where each element is a tuple representing the index pair of matched keypoints in the two input images. To do this, the function employs a simple greedy strategy, where for each keypoint in the second image, it searches for the best-matching keypoint in the first image based on the minimum SSD value. The implementation for finding matches function is as follows:

Algorithm 6 Find Matches

INPUT: D (distance matrix between two arrays of patches)
OUTPUT: matches (index pairs of matched keypoints)

```
Get M1, M2 ← shape of D
Initialize matches as an empty list
for i in range(M2) do
    Compute index ← argmin(D[:, i])
    Append (index, i) to matches
end for
return matches
```

4.4 Visualizing Tracking Results

In this section, we visualize the tracking results for the first two images using the matches_vis function. It takes the target grayscale image, lists of keypoints for source and target images, and the index pairs of matched keypoints as inputs. The function creates a color version of the target image, draws cross markers for keypoints, and lines for matches between keypoints in source and target images. After that the resulting image is displayed. The average number of detected keypoints per frame is 613.165, and the average number of tracked keypoints per frame is 612.04.

4.5 Robust Matching

In this section, we describe the robust matching function *find_matches_robust*, which enhances the matching process by incorporating robust techniques such as the ratio test and cross-validation check.

The ratio test is based on Lowe's method, which compares the distance of the best match to the second-best match. If the ratio is below a certain threshold, the match is considered valid. The equation for the ratio test is:

$$\frac{d_1}{d_2} \leq t$$

where d_1 and d_2 are the distances of the best and second-best matches, and t is the threshold.

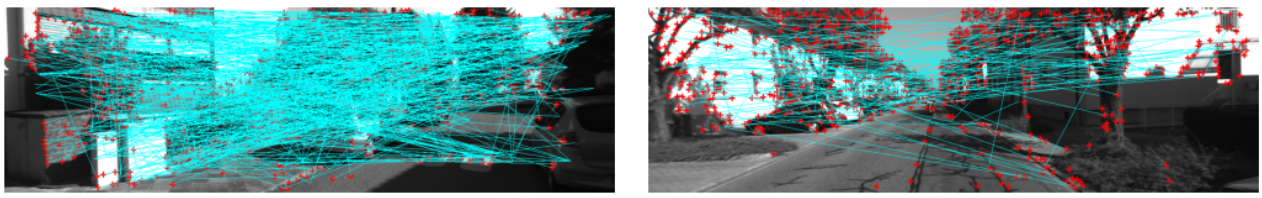
The cross-validation check ensures that the match is consistent in both directions, i.e., the best match of the first keypoint should also have the first keypoint as its best match.

Algorithm 7 Robust Matching

INPUT: D (distance matrix), ratio_test (apply ratio test), cross_validation (apply cross-validation check), threshold (ratio test threshold)
OUTPUT: matches (index pairs of matched keypoints)

```
Initialize matches ← empty list
for each column i in D do
    Compute index ← argmin of D[:, i]
    Compute dist1 ← D[index, i]
    if ratio_test then
        Compute dist2 ← second smallest value in D[:, i]
        if dist1 / dist2 > threshold then
            continue
        end if
    end if
    if cross_validation then
        Compute j ← argmin of D[index, :]
        if j ≠ i then
            if argmin of D[:, j] ≠ index then
                continue
            end if
        end if
    end if
    Append (index, i) to matches
end for
return matches
```

In the new robust matching function, the average number of keypoints tracked per frame is now 157.76, as the difference can be seen in the figure below.



(a) (b)

Figure 6: In (a), we can visualize the output of the matches_vis function but as input the find_matches function and in (b) as input using the robust matches function.

As illustrated in the figure, the number of keypoints tracked per frame has significantly decreased, resulting in a cleaner and more focused visualization of the tracked features.

4.6 Video Generation

In this section, we present a video visualizing tracking results after applying outlier removal tests. The video is generated by iterating through a list of images, comparing and adding different frames. The video, saved in a specified format, provides a clearer visualization of the tracked features with improved tracking results due to the outlier removal tests applied. The generated video can be seen in the attached documents with the name : tracker_robust.mp4

5 Part IV - Corner Tracking with SIFT Features

5.1 Converting Keypoints

The process of converting detected corner keypoints into the appropriate data structure for SIFT begins with the intention to utilize SIFT descriptors instead of directly using patch intensities. To achieve this, we use OpenCV's $cv2.SIFT$ and $cv2.KeyPoint$ classes.

For each image in the dataset, we convert the corner keypoints into SIFT keypoints using $cv2.KeyPoint()$. The keypoints are then processed using the $cv2.SIFT\_create()$ function to compute the corresponding SIFT descriptors. The computed SIFT keypoints and descriptors are stored in separate lists for further processing.

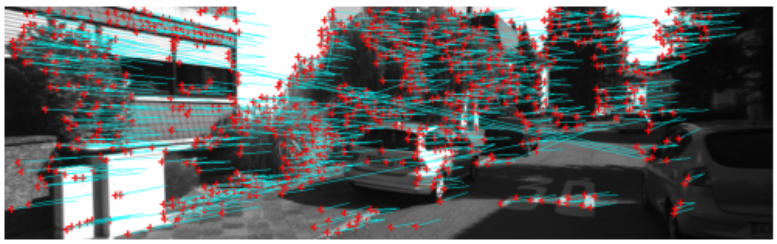
This conversion process allows us to leverage the power of SIFT for feature description, providing a more robust and reliable method for corner tracking.

5.2 Tracking with SIFT

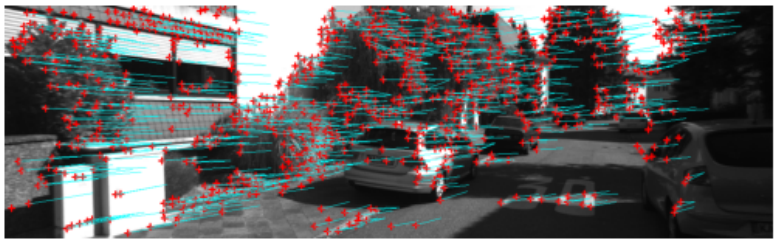
In this section, the results of tracking with the SIFT descriptor demonstrate improved performance compared to previous methods. SIFT descriptors provide a more distinctive and robust representation of keypoints, which improves the matching process.

Using SIFT descriptors and the $find\_matches()$ function, the average number of tracked keypoints per frame increases to 612.5377, as observed in the corresponding figure (a). This improvement indicates that the SIFT-based approach is capable of providing more reliable tracking of keypoints.

When combining SIFT descriptors with the $find\_matches\_robust()$ function, the average number of tracked keypoints per frame is 479.9749, as seen in the related figure (b). This result shows that incorporating robust matching techniques further refines the tracking process, reducing the number of false matches and improving the overall quality of the tracked keypoints.



(a)



(b)

Figure 7: In (a) the SIFT descriptors and in (b) the robust SIFT descriptors are shown. The improvement can be clearly seen.

6 [Extra Points] Part V - Corner Tracking with Prior Motion Fitting Model

6.1 Model Fitting Constraints

In this section, we discuss the model fitting constraints for affine and homography transformations, which are essential for establishing correspondence between keypoints in successive frames and refining the tracking process.

For an affine transformation model, at least 3 corresponding 2D points are needed. The affine transformation is represented by a 2x3 matrix, which has 6 degrees of freedom. Given that each 2D point has 2 degrees of freedom (x and y coordinates), a minimum of 3 corresponding points is required to solve for all 6 unknowns in the affine transformation matrix.

For a homography transformation model, at least 4 corresponding 2D points are necessary. The homography transformation is represented by a 3x3 matrix, which has 8 degrees of freedom. However, as the matrix is homogeneous, it can be scaled by a scalar factor without affecting the transformation. This results in 8-1=7 actual degrees of freedom. Considering that each 2D point has 2 degrees of freedom, at least 4 corresponding points are needed to solve for all 8-1=7 unknowns in the homography matrix.

6.2 Homography Estimation

The homography estimation function calculates the transformation matrix between each pair of consecutive frames in a video. This transformation represents a projective mapping between two images, enabling the alignment of keypoints and enabling the creation of panoramas, image stitching, and video stabilization.

The homography estimation proves particularly beneficial for addressing perspective distortions or significant depth variations within the video sequence, as it accommodates complex transformations. By aligning subsequent frames based on shared features, the homography estimation function enhances the visual quality and consistency of the video.

Algorithm 8 Homography Estimation

INPUT: frames (list of consecutive video frames)
OUTPUT: homographies (list of homography matrices between each pair of consecutive frames)

```
Initialize homographies ← empty list
Create SIFT object
Create BFMatcher object
Define RANSAC threshold
for each pair of consecutive frames in frames do
    Detect keypoints and compute descriptors for each frame
    Match keypoints using BFMatcher and apply ratio test to keep good matches
    Estimate homography using RANSAC
    Append homography to the list of homographies
end for
return homographies
```

6.3 Affine Motion Model

In this section, we discuss the results obtained using the affine motion model. The affine motion model is a simplified approach to image alignment and transformation, which assumes that the motion between consecutive frames can be approximated using a linear transformation. This model involves translation, rotation, scaling, and shearing operations while preserving parallelism between lines in the image.

The results obtained with the affine motion model are generally satisfactory for scenes with small depth variations and limited perspective distortions. It is capable of providing reasonable alignment and stabilization for videos featuring planar scenes or objects moving in a plane. However, its performance may degrade when dealing with more complex scenarios, such as those involving significant depth variations, non-planar scenes, or substantial perspective distortions.

Algorithm 9 Homography Estimation

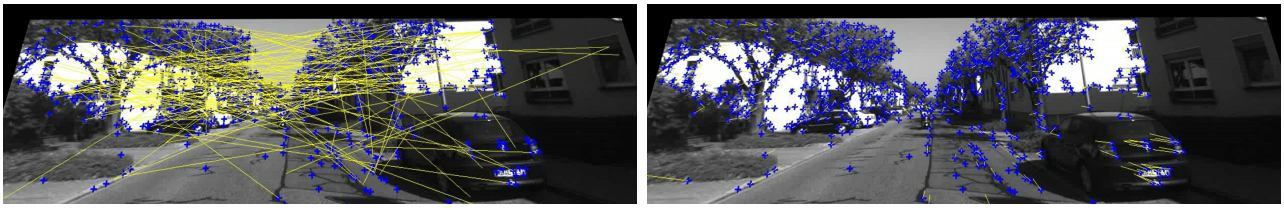
INPUT: frames (list of consecutive video frames)
OUTPUT: homographies (list of homography matrices between each pair of consecutive frames)

```
Initialize homographies ← empty list
Create SIFT object
Create BFMatcher object
Define RANSAC threshold
for each pair of consecutive frames in frames do
    Detect keypoints and compute descriptors for each frame
    Match keypoints using BFMatcher and apply ratio test to keep good matches
    Estimate homography using RANSAC
    Append homography to the list of homographies
end for
return homographies
```

7 Conclusion

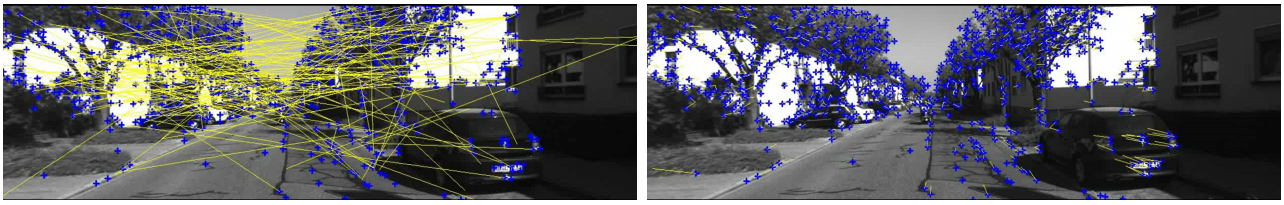
7.1 Comparing Results

After analyzing the results, it can be observed that using the affine transformation provides a clear improvement over using homography. The resulting video using homography appears to be incomplete, with parts of the image missing (Figure 8). In contrast, the video generated using affine transformations appears to be much smoother and complete (Figure 9). This improvement is likely due to the fact that the affine transformation provides a more flexible motion model compared to homography, which assumes a planar motion model.



(a) (b)

Figure 8: In (a), we can visualize the resulting video using homography with the Harris inliers and in (b) with the SIFT inliers.



(a) (b)

Figure 9: In (a), we can visualize the resulting video using affine transformations with the Harris inliers and in (b) with the SIFT inliers.