Computer Vision/Visual Perception
ESIREM Robotique & Master in Computer Vision
Renato Martins – Universite de Bourgogne, January 2023

Assignment I - Corner Detection & Feature Tracking

In this exercise, we will implement your own Harris corner detector and use it to track corners over time with patch templates and with SIFT descriptor. You will implement your own Harris corner detector and then track keypoints on the first 200 frames of the popular KITTI Visual Odometry dataset [1], as shown in Figure1.

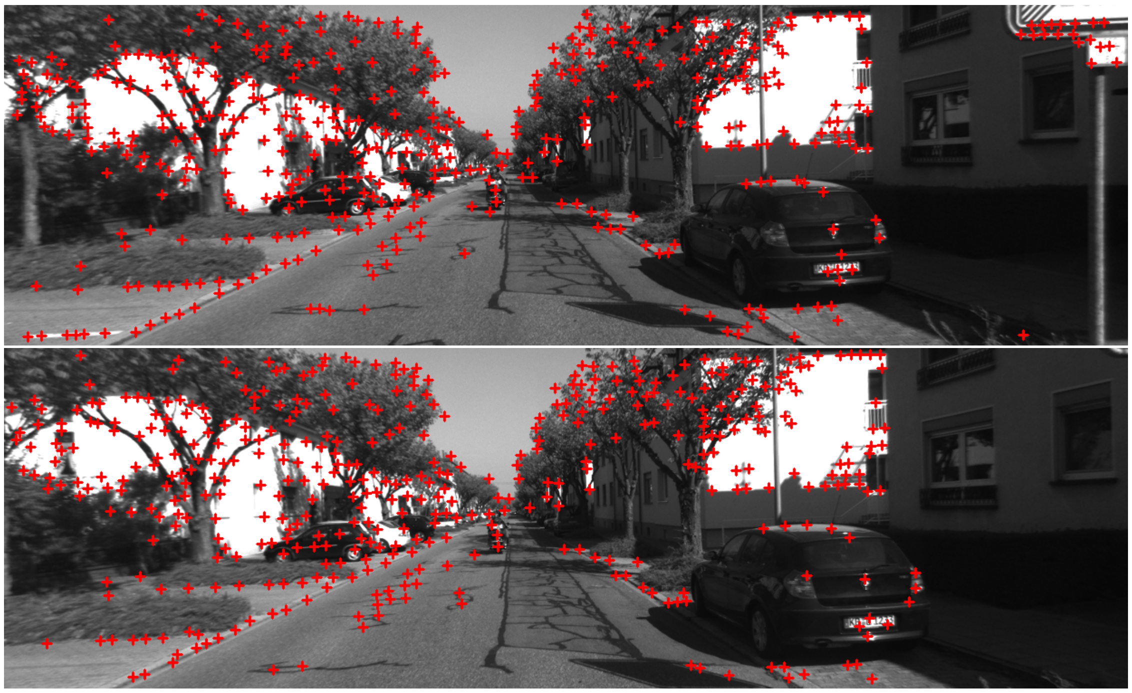

Figure 1: Detected corners in the first two frames of the KITTI sequence.

IMPORTANT: You should not use any high-level OpenCV function (unless stated otherwise explicitly) such as for finding the corners (e.g., Harris, ...) or to define the Gaussian and Sobel kernels. However, you can use the available convolution implementations in the Scipy and Scikit-Image libraries. You are also welcomed to check your detected corner results with the ones provided by OpenCV functions, as the example here[1] (please notice this example do not apply non-maximum suppression).

[1]`https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html`

Part I - Environment Setup & Inputs

Please create a notebook on Colab or follow these steps to configure your conda environment if you are working in your local machine:

```
>> conda create -n myharris_track
>> conda activate myharris_track
>> conda install jupyter
# Please go to your workspace directory of the course and run jupyter
>> jupyter-notebook
```

Again we will greatly use Numpy arrays and convolution operations during this exercise. So do not hesitate to go back in the slides of Class 2 (OpenCV/Python) to check some definitions or in this tutorial on Numpy[2]. There are also some helpful "Cheat Sheets" of basic commands for Numpy and Scipy (Linear algebra in Python)[3].

We provide the first 200 images from **Sequence 00** of KITTI in the folder **images/** in the file *CV_Assignment_1_CornerTracker.zip*. We also include a video for the 100 last frames of the expected correspondence outputs (file *corner_tracking_patch.mp4*) and the visualization functions for plotting the corresponding corners between two frames (as well as several hints along the assignment).

Part II - Corner Detection

We start by implementing your own Harris corner detector. Please check the slides of the class lectures (as well as the book references) for recalling important concepts and main steps of Harris.

1. Since derivative filters are sensitive to noise, we will start smoothing the images for reducing noise with a low-pass filter before computing the gradients. Implement a function *gaussian_kernel* that receives the size of the kernel h and the standard deviation of the Gaussian, and returns the normalized Gaussian kernel filter with dimension $(h \times h)$. Use this function to create a Gaussian kernel of size (3x3) and standard deviation of 0.1 (KITTI images are not noisy ;-)). Please check that your function will return a normalized kernel, i.e., sum of elements equal to 1.

2. Please recall the form of Sobel filters in the $(x,\ y)$ image directions with size (3x3). Write a function *gradient_image* that receives an image and returns two images of the gradients in $(x,\ y)$ using a convolution of the Sobel filter with the smoothed input image using the Gaussian kernel you defined in the previous step. Then compute the three images I_xI_x, I_yI_y and I_xIy. Please note these multiplications are element-wise!

3. Create a second Gaussian kernel with your function *gaussian_kernel* using a standard deviation of 1 and size 5 (returned kernel of size (5x5)). You will use this filter as the window w and image gradients in the previous step to compute the (2x2) matrix elements of M (as shown in the lectures' slides). Create a function *harris_response* that receives (I_xI_x, I_yI_y and I_xIy, and the kernel w) and returns an image with Harris corner response R for all pixels using $\alpha = 0.06$.

[2] https://numpy.org/devdocs/user/quickstart.html
[3] Scipy: http://datacamp-community-prod.s3.amazonaws.com/dfdb6d58-e044-4b38-bab3-5de0b825909b
Numpy: http://datacamp-community-prod.s3.amazonaws.com/ba1fe95a-8b70-4d2f-95b0-bc954e9071b0

4. Create a function *response_non_maxsup* that apply Non-Maximum-Suppression in a window of (15x15) pixels whose response is above 0.01 times the global maximum of R. For that you can use *import skimage.feature as feature* and then *feature.peak.peak_local_max* with parameters *min_distance* $= 7$ and *threshold_rel* $= 0.01$. Your function should return a list with the 2D pixel coordinates of the resulting corner keypoints.

5. Visualize and check the corners locations in the images by plotting a red '+' in the keypoint location per image. You can create a video with the 200 frames using the following function:

```
# This function plots detected keypoints with red '+' markers over a grayscale image
# inputs: image and list of keypoints coordinates in Numpy convention order
def keypoints_vis(image, list_keypoints):

    # repeat channels for creating color image from grayscale
    image_color = cv.merge([image,image,image]).astype(np.uint8)

    # add cross for each keypoint
    for keypoint in list_keypoints:
        # please note that coordinates are inverted to follow opencv convention (x,y)
            instead of (y,x)
        cv.drawMarker(image_color, (keypoint[1],keypoint[0]), (255,0,0), markerType=cv
            .MARKER_CROSS, markerSize=10, thickness=2, line_type=cv.LINE_8)

    # display image
    plt.figure()
    plt.imshow(image_color)
    plt.axis('off')
    plt.show()
    return image_color
```

Part III - Corner Tracking with Patch Templates

Now that we have detected keypoint locations (your corners), we will start with a simple description strategy of using pixel intensity values of image patches around each keypoint. While this is not quite state-of-the-art, as we have seen in the class, we can still use this strategy to track keypoints across frames! In **Part IV** you will use SIFT descriptor for computing the features.

1. Write a function that returns a list of patches for the detected keypoints. Please use a window size $N = 32$ pixels around each keypoint position that were detected with your Harris implementation in Part II. Hint: In order to avoid image border issues you can use the padding *cv.copyMakeBorder(src, top=N/2, bottom=N/2, left=N/2, right=N/2, borderType=cv.BORDER_REPLICATE)* in the image. You should then also add N/2 in the coordinates of your keypoints when extracting the intensities of the window region!

2. Extract intensity normalized patches for each image using the detected keypoints (in order to be robust to illumination changes). The normalization per patch can be done with:

$$\tilde{X} = \frac{X - np.mean(X)}{np.std(X) + \epsilon},$$

where $\epsilon = 10^{-7}$ for avoiding numerical instabilities on constant intensity patches.

3. **Correspondence:** Implement a function *distance_matrix* that takes two arrays of patches respectively of size $M_1 \times N \times N$ and $M_2 \times N \times N$ and computes a matrix of distance D, where D_{ij} is the squared distance between the the normalized patches i in the first list of patches and the patch j in the second list of patches.

4. Create a function *find_matches* that returns the correspondences between keypoints of two successive frames (k and $k+1$). The function should return a tuple of indexes pairs of matched keypoints, i.e., you should check for each column the position with minimum distance in D. Hint: You can use *index = np.argmin(D[:,i])* which is equivalent to have a matching pair $(index, i)$. What is the average number of detected and tracked keypoints per frame along the sequence?

5. Visualize the tracking results for the two first images. For the visualization of corresponding matches we provide the following function:

```python
## plot matched keypoints
# This function plots detected keypoints with red '+' markes over a grayscale image
# inputs: target image, lists of src , list of target keypoints (in Numpy convention
    order), and list of matches
def matches_vis(image_tgt, list_keypoints_src, list_keypoints_tgt, matches):

    # repeat channels for creating color image from grayscale
    image_color = cv.merge([image_tgt,image_tgt,image_tgt]).astype(np.uint8)

    # add cross for each keypoint in target image
    for keyp in list_keypoints_tgt:
        cv.drawMarker(image_color, (keyp[1], keyp[0]), (255,0,0), markerType=cv.
            MARKER_CROSS, markerSize=10, thickness=2, line_type=cv.LINE_8)

    # draw green lines of matches
    for (i,j) in matches:
        coord_src, coord_tgt = list_keypoints_src[i,:], list_keypoints_tgt[j,:]
        # please note that coordinates are inverted to follow opencv convention (x,y)
            instead of (y,x)
        cv.line(image_color,(coord_tgt[1], coord_tgt[0]),(coord_src[1], coord_src[0])
            ,(0,255,0),1)

    # display image
    plt.figure()
    plt.imshow(image_color)
    plt.axis('off')
    plt.show()
    return image_color
```

And you should have something similar Figure 2 for the correspondence results of frames 114 and 115 of the sequence:

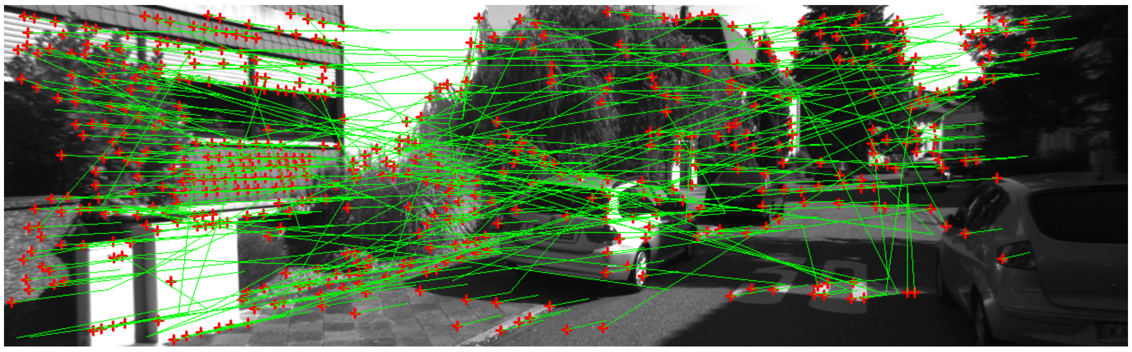

Figure 2: Tracking results between the frames 114 and 115 (the motion of keypoints are indicated by the green lines).

6. As you can notice, there are several outliers in the matching. Now we will try to improve the matching by using the 1NN/2NN ratio test and cross-validation check for removing ambiguous correspondences (please see the slides of the course). Create a function *find_matches_robust* that includes these two tests. Your results should then be similar to Figure3.

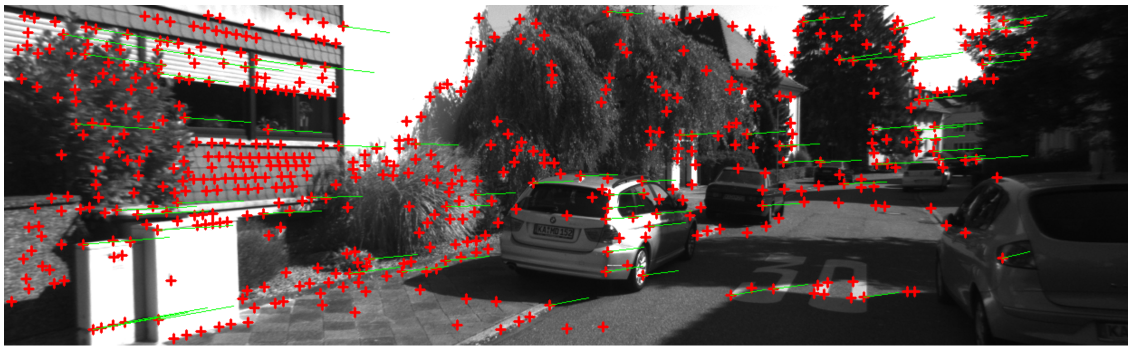

Figure 3: Corner tracker results with outlier tests between frames 114 and 115 (the motion of keypoints are indicated by the green lines).

What is the average number of tracked keypoints per frame when considering two matching outlier tests?

7. Create a video of the generated images for visualizing the tracking results with the outlier removal tests. You can use the following function:

```python
# this function creates a video from a list of paths of images
def make_video(list_image_names):
    # reads the images
    img_array = []
    for filename in sorted(list_image_names):
        img = cv.imread(filename)
        img_array.append(img)

    # video size = image size
    height, width, layers = img.shape
    size = (width,height)
    # create mp4 writer with 3 FPS
    video_wr = cv.VideoWriter('corner_tracking_patch.mp4',cv.VideoWriter_fourcc(*'mp4v
        '), 3, size)
    for i in range(len(img_array)):
        video_wr.write(img_array[i])
    video_wr.release()
```

Can you observe any improvement in matching. For how many frames in average the keypoints can be tracked without and with the test?

Part IV - Corner Tracking with SIFT Features

We will now use the SIFT descriptor for extracting features in the detected corners from **Part II**, instead of using the patches intensities directly. You are now allowed to use cv.SIFT and cv.KeyPoint for performing the description in the following steps.

1. In order to use SIFT descriptor you need to convert your detected corner keypoints list into the appropriate data structure expected by SIFT. Please check the signature of the

cv.KeyPoint class. You need to fill the fields *(pt.x, pt.y)* with the coordinates of the locations of your detected corner keypoints. Since we do not have the estimated orientation in detected corners you can set the field *angle* to zero.

2. Our detected corners also do not have the scale attribute required by SIFT (field *size* in *cv.KeyPoint*). We will then use the same scale for all keypoints, as done in the template tracking in **Part III**. What is the keypoint scale (size) if we want SIFT regions with patch size of (32x32)? (Please check the slides of the class).
3. Extract the SIFT descriptors for each image by passing your converted list of *cv.KeyPoint*.
4. Perform the steps 3-7 from **Part III** for the computed SIFT descriptors.
5. What is the average number of matched keypoints? Did the tracking results improved with SIFT by visual inspection? For how many frames in average can we track the kypoints?

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

We will now assume that the 2D keypoint motion inter-frame follows an affine transformation (this is a prior information given by an oracle). You will then explore this prior in the motion to handle outliers with RANSAC!

1. Check in the course slides the model fitting constraints for fitting an affine transformation model and to an homography one. How many corresponding 2D points are required for each case?
2. Create a function that estimates homography transformations between each pair of subsequent frames. You can use the OpenCV function *cv.findHomography*. Please notice *cv.findHomography* function comes with RANSAC robust function option that you should use in this case. Create the videos for the filtered correspondences using only the found inliers for Harris corners and SIFT.
3. Generate the same results now considering an affine motion model (affine transformation) between the views. What could you observe? Is there improvements compared to the case without the prior motion model?

Submission

Please return a PDF report explaining your reasoning, commenting your results, presenting visualizations and equation developments asked during the assignment. You should also include the two generated videos of the tracking, and your python notebook (commented) with the corresponding implementations, inside a [name]_assign1_tracker.zip file (replacing [name] with your name ;-)). The submission should be done via the teams channel of the course using teams assignment.
Deadline: 06/02/2023 at 23:59pm.

Notes:

- Plagiarism copying the work from another source (student, internet, etc.) will be awarded a 0 mark. In case there are multiple submissions with the same work (or sharing partial content), each one will receive a 0.

- **Credits:** This assignment was partially inspired by the exercises kindly provided by Davide Scaramuzza and by Martin de La Gorce.

References

[1] Andreas Geiger, Philip Lenz and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite". IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2012.

[2] Richard Hartley and Andrew Zisserman. "Multiple View Geometry in Computer Vision", 2003.