Harris Corner Detection and Keypoint Tracking

Titus Cieslewski

Contents

The goal of this laboratory session is to get you familiarized with feature detection, description and matching.

1 Preliminaries

1.1 Outline of the exercise

In this exercise, you will implement Harris corner detection and keypoint tracking, by only using the knowledge you have so far obtained in the class. You will run this keypoint tracker on the first 200 frames of the popular KITTI dataset.

Figure 1: Matching features from the first two frames in the KITTI dataset.

You will achieve this with the following steps: First, you will evaluate the Harris score for each pixel of the input image. Then, you will select keypoints based on the Harris scores. In a next step, you will evaluate simple image patch descriptors at the selected keypoint locations. Finally,

you will match these descriptors using the sum of squared differences (SSD) in order to find feature correspondences between frames.

1.2 Provided code

We provide you with skeletal Matlab code (`main.m`) which loads the images for you, provides you with good initial parameters, and does the plotting for you. Your job will be to implement the code that does the actual logic. We also provide the functions stubs with some comments about the input and output formats, so if these are not clear from this pdf, they should be clear from the function stubs. *You do not need to reproduce the reference outputs exactly*, as slight implementation differences might affect the outputs. The `main` script is subdivided into an initialization part and the parts below, you can execute the separate parts by hitting `Ctrl+Enter` while your cursor is in the corresponding part.

1.3 Conventions

In our code, we use the naming convention that `patch_size` represents the size of a square patch in one dimension and `patch_radius` the amount of pixels between the center pixel of an odd-sized patch and its border. Thus, `patch_size` = `patch_radius` $\cdot 2+1$ and a patch has `patch_size`^{2} pixels.

Furthermore, for all conversions between matrices and vectors, unless otherwise stated, we assume column-wise order. This means that e.g. a matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ gets transormed into $\mathbf{a} = \begin{bmatrix} 1 & 3 & 2 & 4 \end{bmatrix}^T$ and vice-versa. This corresponds to the conventions of Matlab commands `reshape` and `a = A(:)`.

2 Part 1: Calculate Harris scores

We detect features using the Harris corner detector. As seen in the lecture, the Harris score for a given pixel (u, v) is:

$$R_{u,v} = \det(M_{u,v}) - \kappa \text{trace}^2(M_{u,v}) \tag{1}$$

with

$$M_{u,v} = \begin{bmatrix} \sum I_{x,u,v}^2 & \sum I_{x,u,v} I_{y,u,v} \\ \sum I_{x,u,v} I_{y,u,v} & \sum I_{y,u,v}^2 \end{bmatrix}, \tag{2}$$

where the sums are performed over a fixed-size image patch around the point for which the score is evaluated, and I_x and I_y are the image derivatives in the x and y direction respectively. These can be obtained by convolving the image with the Sobel filter:

$$I_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I, \quad I_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I \tag{3}$$

In the lecture, we have observed that it is sensible to combine smoothing and derivative filters if we have noisy images. In this exercise, however, the given images are free of noise and we can directly use derivative filters (e.g. Sobel filter) without smoothing.

You should be able to evaluate R without any for-loop. The trick is to evaluate (1) without expressing M as a matrix, but directly with the coefficients. Then, if you express each coefficient of M in a matrix representing the entire image, you should be able to perform (1) with Matlab's coefficient-wise operators (`.*` and `.^`). Use `conv2` with the `valid` option (in our convention, filters are not defined at border pixels for which the mask only partially overlaps the image) for convolutions and `padarray` to ensure that a harris score is returned for each pixel. By using padarray you simply set the score to 0 for pixels for which the score is not defined. You should obtain roughly the scores shown in Fig. 2. Some hints:

- You should have intermediate matrices containing as coefficients $I_x, I_y, I_x^2, I_y^2, I_x I_y, \sum I_x^2, \sum I_y^2$ and $\sum I_x I_y$ for each pixel *where they are defined*. In particular, if the size of the image I is $h \times w$, the sizes of the first five matrices will be $(h-2) \times (w-2)$ and the sizes of the last three matrices, as well as the initial R, will be $(h-1-\text{patch\_size}) \times (w-1-\text{patch\_size})$. As written before, you can obtain the final R by applying `padarray` to the initial R.
- The image in Fig. 2 has been obtained by setting all negative scores to 0. You can do this since in the remainder of this exercise we will only use the positive values.

Figure 2: Harris scores for the first frame of the KITTI dataset.

3 Part 2: Select keypoints

In the lecture you have seen that to select keypoints, you need to find points where R exceeds a certain threshold. But how to choose that threshold? A popular approach is to select the k points with highest score instead. We will pursue this approach here, so we won't need to worry about the threshold. Note that pixels with high scores tend to have neighbors with high scores. To prevent selecting neighboring pixels as keypoints, perform non-maximum suppression around the highest-scoring pixel selected at each iteration; that is, set all pixel scores within a radius to 0 (you can use a square box to make things easier in Matlab).

Your entire function should contain only one for-loop - to iterate over single keypoint selection. Take a closer look at the ways in which the `max` function can be used. `ind2sub` might also be helpful. Fig. (3) shows what you should roughly get.

4 Part 3: Describe keypoints

We simply describe keypoints using the pixel intensity values of image patches around the keypoint. Trivial, right? Still, it's useful to verify that your output makes sense after this step. Your function should return a $d \times k$ matrix, where d is the descriptor dimension (total amount of pixels in patch) and k the amount of keypoints. The ith column of the matrix should contain the patch intensities

Figure 3: Keypoints selected in the first frame of the KITTI dataset.

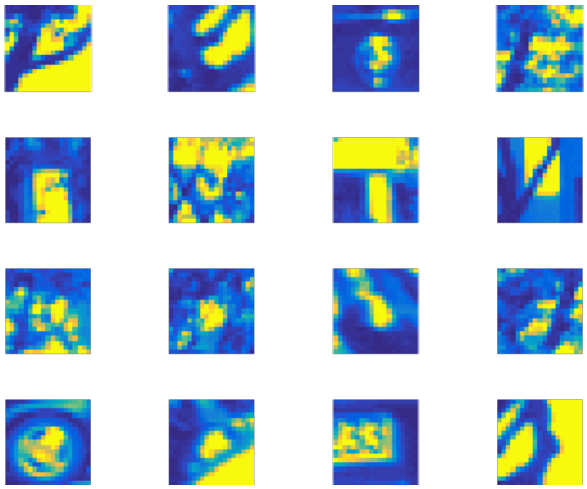

Figure 4: The 16 patch-based descriptors with the highest Harris scores with our code.

around the ith keypoint, stored in column-wise order (see section 1.3). You might want to check out the `reshape` function. Also, we recommend to `padarray` the input matrix so that you don't have to deal with special cases if a keypoint is closer to the image border than the descriptor patch radius. You should use at most one for-loop, to iterate over keypoints. Once the descriptors are evaluated, you should be able to tell where in the image they come from based on the plotting we implemented for you. Our 16 "best" descriptors (highest Harris score) are shown in Fig. (4). Assuming that the keypoint selection code started with the keypoint with highest Harris score and put it at the first place (etc.), the 16 "best" descriptors should be stored in the first 16 columns.

5 Part 4 and 5: Match descriptors

In this final step, we will match descriptors between two images. We will simply do this by matching the image patches that differ the least. While this is not quite state-of-the-art, we can already use this to track keypoints across frames! In part 4, you will use the first two frames of KITTI to develop and debug your descriptors matching. Then in part 5 you will apply your tracking to the first 200 frames of the KITTI dataset, and if all goes well, you should obtain behavior as shown in the preview video (if your code is efficient, you should be able to achieve a similar speed).

Note that you can use the Matlab function `pdist2` to compute the SSD, which will significantly boost performance. Familiarize yourself with this command and use it for this exercise. Your function should take as input two descriptor matrices, query_descriptors (descriptors from current frame) and database_descriptors (descriptors from last frame). It should then return a vector with the same length as the query_descriptors matrix, such that the i-th element of the result vector is the index

of the matching database descriptors for the i-th query descriptor. As you can imagine, not all query descriptors will be matched. A naive approach would be to look for a descriptor distance threshold, such that you only match descriptors which have a smaller descriptor distance between them. However, since we are looking at subsequent frames of a trajectory, we know that there should be at least one match, and so we will use an adaptive threshold

$$\delta = \lambda \cdot d_{\min}, \tag{4}$$

where $d_{\min}$ is the smallest non-zero distance between two descriptors from the two descriptor sets. Write code that matches a database descriptor at most once, i.e. no database descriptor should be matched to more than one query descriptor. If all goes well, the plot resulting from executing the code from part 4 should look like Fig. 1.1. You are then ready to run part 5 and compare with the preview video.

5.1 Matching quality

Note that the matching is not quite perfect - many keypoints are not matched, and there are occasional outliers (you can tweak λ to trade off between false negatives and false positives - give it a try!). In a later exercise, we will see how we can filter outliers using geometric verification. Apart from geometric verification, can you think of a simple way to filter outliers in the given dataset (hint: camera motion is not too big between frames)?

5.2 Optimizing your code the smart way - profiling

If you want to make your code run faster, you might be tempted to make a guess at what part of your code is slowest and then try to optimize that - don't do that, that's ineffective! The most effective way to optimize your code is through profiling. In Matlab, it's very simple to profile code. Type `profile on` in the console, then run part 5. Once part 5 is complete, type `profile viewer` and inspect the profile summary. Is it what you would have expected it to be? Identify one function that you have written and click on its corresponding entry. You can scroll down and see how much time each line of code has taken.