Computer Vision: Neighborhood Processing & Filters

Arsen Sheverdin, Mátyás Schubert, Ard Snijders

November 23, 2020

Introduction

In order to efficiently compare images and match objects on them, we need to recognize unique points which we can find on both images. These points of interest are called features and have several applications in the field of computer vision, such as image alignment, motion tracking and object recognition. Because of their usefulness, several algorithms have been developed to detect them. One of them is the Harris Corner Detector which we will implement and test in this report. Aside from identifying features, it can be useful to track their motion. This can be achieved via estimating the so-called optic flow of a set of images. In this report, we will perform optic flow estimation with the Lucas-Kanade method. Finally, we explore the concept of feature tracking, where we employ both corner detection and optic flow estimation algorithms to identify features and track their motion over time.

Harris Corner Detector

Features need to have 2D structure in order to be localised. This means flat surfaces and edges are not considered features. Instead we try to detect corners. The implementation of the corner detector is made up of two main parts. First we compute the corner responses for each pixel which generates a heat map for points of interest. Then we select the local maximas in reasonable windows which are larger than a certain threshold to get the exact points where we detect corners. The detected corners for different images can be seen on figure 1 and figure 2.

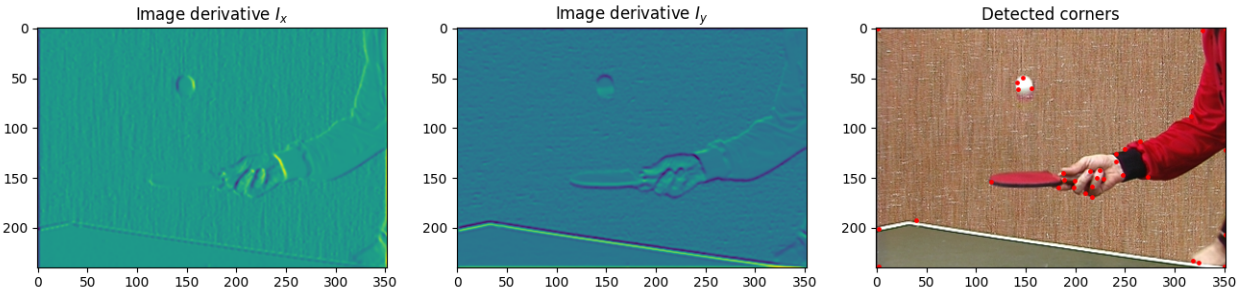


Figure 1: Corners detected by using a threshold of 70000

Using lower thresholds resulted in misclassified corners on the rough surface of the walls appearing on both images. Unfortunately there were corners which had lower response rate than this noise so increasing the threshold implied a trade-off between removing both misclassified and rightly classified corners.

The Harris Corner Detector algorithm should be rotation invariant as the shape of corners, and thus the eigenvalues generated are not impacted by rotation. While this stands in theory, by only approximating the

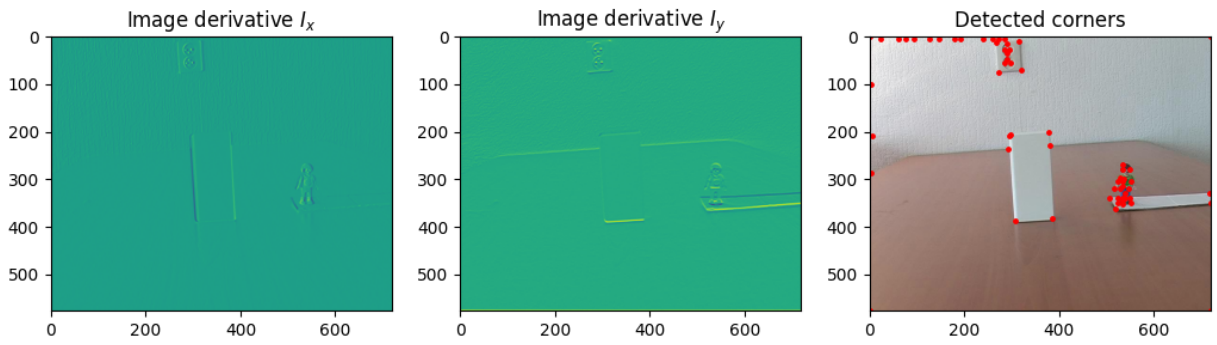


Figure 2: Corners detected by using a threshold of 10000

first order Gaussian derivative on the horizontal and vertical axis, some points get a different cornerness response rate in different rotation angles. This difference is almost negligible when rotating by 90^o degrees. The effects of rotation on detected corners can be seen on figure 3.

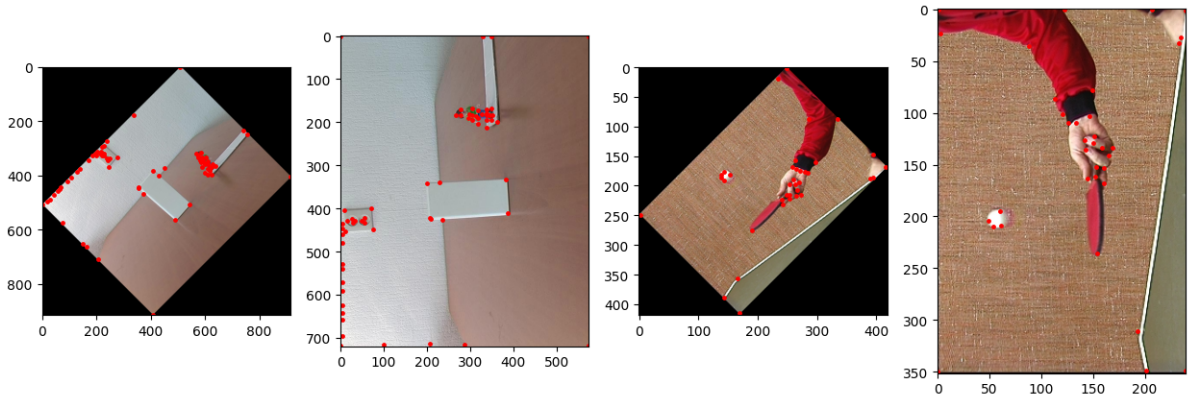


Figure 3: Corners detected after rotating the images by 45^o and 90^o degrees

Shi and Tomasi Cornerness

Shi and Tomasi define cornerness in a similar fashion to the Harris Corner Detector [3]. For the latter, the cornerness $H(x, y)$ is defined as:

$$H = \lambda_1\lambda_2 - 0.04(\lambda_1 + \lambda_2)^2 = det(Q) - k * (trace(Q))^2 \tag{1}$$

Shi and Tomasi simplify this notion of cornerness, instead only considering the minimum of the two eigenvalues λ_1 and λ_2.

$$H = min(\lambda_1, \lambda_2) \tag{2}$$

It can then be determined whether something is considered a corner if the resulting value for H surpasses some threshold. When the two eigenvalues are small, this corresponds to a somewhat constant intensity profile, given a particular window. When one of the eigenvalues is large and the other small, this suggests a unidirectional texture pattern. Finally, when both eigenvalues are large this could suggest the presence of

corners, salt-and-pepper textures, and any other profile that lends itself well for feature-tracking. As the Shi Tomasi method builds on Harris corner detection, we would reasonably expect to see the same invariance properties; agreeable invariance to translation and rotation, but lesser invariance for scale. To verify this, we run a brief experiment using the off-the-shelf `cv2.goodFeaturesToTrack()` method, an algorithm that follows the detection procedure as outlined by Shi & Tomasi [1]. Results can be seen in figure 4. As can be observed, the Shi-Tomasi method appears to perform well for all of the described types of invariances, insofar that roughly speaking, the same set of corners is preserved across the three different transformations.

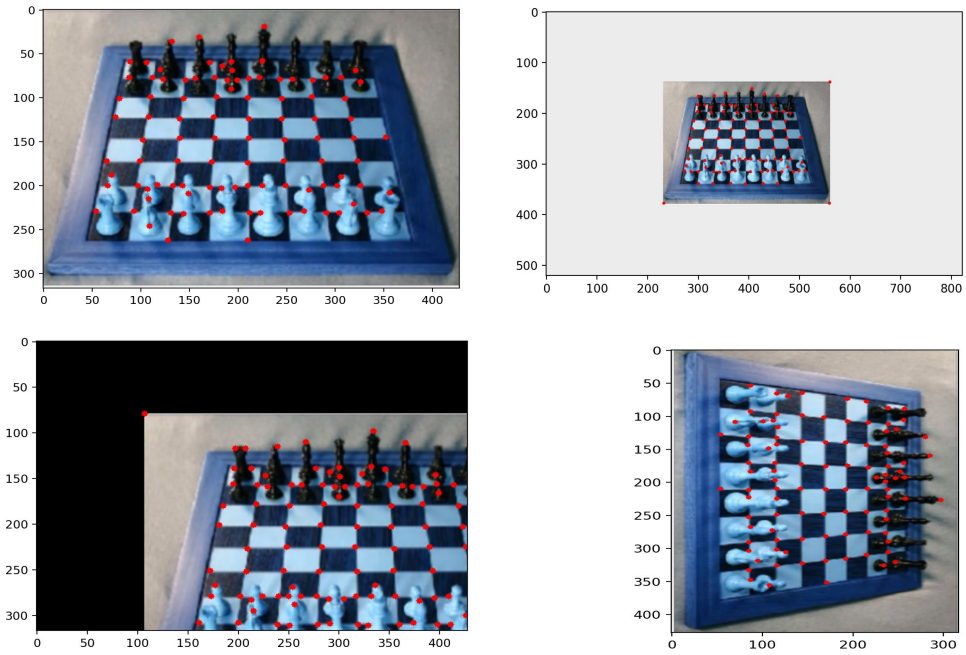


Figure 4: Testing Shi-Tomasi Corner detection for scale, translation and rotation invariance. Top row, from left to right: original image, down-scaled image. Bottom row: Translated image, and image rotated by 90 degrees.

Optical Flow - Lucas-Kanade

Optical flow is the perceived motion of image features, or pixels, over time, thereby giving rise to movement of objects between images. Optical flow usually builds on the so-called *brightness constancy* assumption, in which we assume pixel intensities to remain constant over a change in time t, even if their x, y position in the image might change. We can leverage this assumption to estimate the optical flow, by constructing a vector field which captures the x, y movement for a certain pixel window. In this report we will do so by use of the Lucas-Kanade optic flow estimation algorithm. An important assumption of this algorithm is that optical flow can be assumed to be constant for a region of pixels. This implies that we can compute the optical flow for a particular region by solving the following system of equations:

$$
\begin{aligned}
I_x(q_1)V_x + I_y(q_1)V_y &= -I_t(q_1) \\
I_x(q_2)V_x + I_y(q_2)V_y &= -I_t(q_2) \\
&\vdots \\
I_x(q_n)V_x + I_y(q_n)V_y &= -I_t(q_n)
\end{aligned} \tag{3}
$$

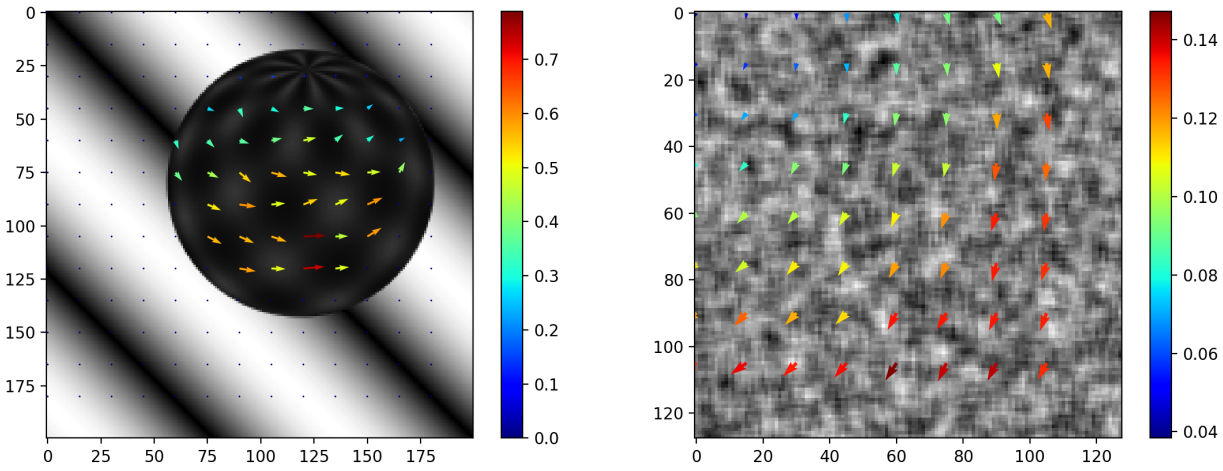
Here I_x denotes the gradient in the x-direction, I_y the gradient in the y-direction, and I_t the gradient over time. We can approximate the x, y gradients by convolving the image with 3×3 Sobel kernels. The change over time can simply be computed by taking the difference in pixel intensity between two frames. n denotes the total number of pixels for a given window. We can then extend this system to be expressed in matrix notation in the form $Av = b$, as described below:

$$
A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix}, v = \begin{bmatrix} V_x \\ V_y \end{bmatrix}, \text{ and } b = \begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ \vdots \\ -I_t(q_n) \end{bmatrix} \tag{4}
$$

We can then rewrite this into the form:

$$
v = (A^T A)^{-1} A^T b \tag{5}
$$

The resulting system can be solved via a least-squares method. Doing so for all regions, we can obtain the optical flow estimates, provided that we have two frames of each image. Results for optical flow estimation for `sphere{1,2}.ppm` and `synth{1,2}.pgm` can be observed in Figure 5.



(a) Optic Flow estimated from `sphere{1,2}.ppm`

(b) Optic Flow estimated from `synth{1,2}.pgm`

Figure 5: Optic flow estimations via Lucas-Kanade. Note that vector fields are plotted over the first time frame of both images.

As can be observed, employing the Lucas-Kanade methods leads to agreeably accurate results; for the synthetic sphere, it accurately captures the rightward rotational movement of the sphere, whilst not noting any movement in the background, which is indeed fixed on both images (this can also be seen from the dots on the background, which are really just infitely small vectors, implying no movement in those regions).

Similarly, we get a decent result for the blurred, noisy `synth` image, suggesting a large, rotational movement towards the horizontal axis.

We can note a number of things here. First, the algorithm seems to perform best on the synth image, as there is comparatively less variation among vector angles; Whilst we can still approximate the movement on the sphere image, the larger variations between vector angles can be interpreted as the algorithm making less accurate estimations per region. This can be explained by multiple factors. The `synth` image is a synthetically generated, 2D surface, with no apparent light source and/or shading present. Thus, the differences in pixel brightness over time can be solely attributed to a translation of the surface with respect to the image window. Given these 'easy' conditions, the algorithm is more likely to approximate the 'true' optic flow as it is not hindered by variations in lighting as a result of e.g. the shape of the object or surface, as the image is translated. Furthermore, the synthetic surface in `synth` is very heterogeneous, allowing the algorithm to leverage the many differences in pixel values to estimate the optic flow between them. In case of `sphere`, the algorithm can still deduce variation via the alternating dark and light patches present on the image, but the 'perceived' surface of these patches will vary as the sphere is rotated (for instance, as the sphere rotates, a white patch will become less round and more elliptical w.r.t. the lens of the camera). The thereby induced distortion of brightness over time gives rise to less accurate estimations, possibly explaining the comparatively larger variations between vector angles.

The aperture problem is a recurring difficulty in all kinds of optical flow estimating algorithms. While the Lucas-Kanade algorithm solves this by operating on local regions and assuming constant flow. The Horn–Schunck method instead assumes a smoothness [2] in the flow over the whole scene, thus it is regarded as a global method. We can formulate the flow with an energy function and add a global constraint that penalizes the total variation of the flow field. The algorithm tries to minimize this function, which can be written as the following equation:

$$E = \int\int (I_x V_x + I_y V_y + I_t)^2 + \lambda(||\nabla V_x||^2 + ||\nabla V_x||^2)\ dx\ dy \tag{6}$$

Where λ is the weight of the regularization.

Both algorithms estimate flow by comparing pixels on two different frames. This comparison can only provide us valuable information about the movement if there is a change in the pixel values. However on flat surfaces with homogeneous intensity, pixel values do not change regardless of motion. Because of this, both methods fail to recognize movement on such surfaces.

Feature Tracking

We can now combine the algorithms discussed previously to construct a feature tracking algorithm. Here, for a given set of frames, we can first compute the corner features using the Harris Corner Detector for the first frame in the series of frames. After that we applied the Lucas - Kanade algorithm for every corner pixel, which we found before, with the window size of 15×15 pixels. Application of this algorithm on the first frame gave us V_x and V_y values for each corner pixel, using which, we can predict the position of the features in the next frame. For second and subsequent frames, the process of predicting feature positions for the next frame repeats, however, without the utilization of Harris Corner Detector.

In principle, one could compute features for every single frame, and subsequently estimate optic flow between these two features. The downside of this approach is that it is much more computationally costly to re-compute features for every frame. However, the disadvantage of only computing features once, and tracking those for the remaining frames (as described above), is that potential features that were not yet present in the first frame will also not be identified later. Possibly, a trade-off between feature detection and computational resources could be achieved by re-computing features for every n frames. This would also mitigate the compounding bias of estimating feature positions in subsequent frames based on motion vectors alone.

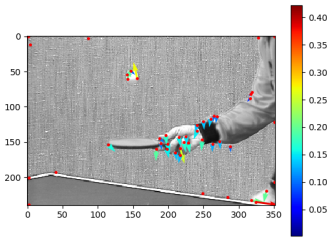


Figure 6: Frame #1

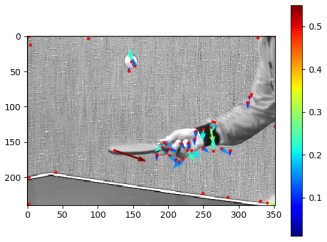


Figure 7: Frame #3

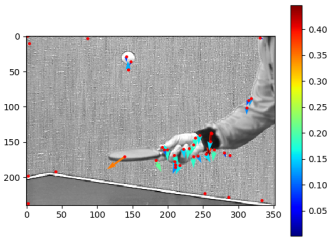


Figure 8: Frame #5

Figure 9: Optical flow of feature points (red dots)

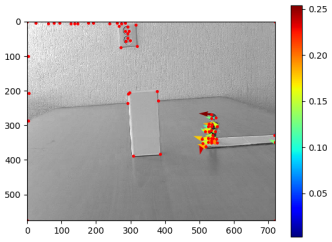


Figure 10: Frame #1

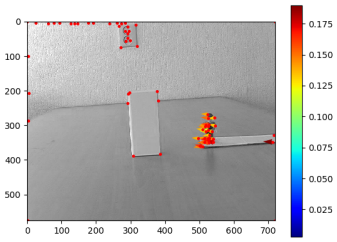


Figure 11: Frame #6

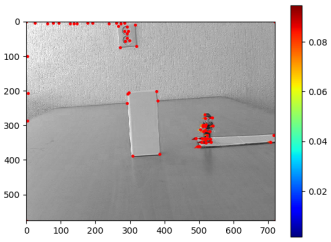


Figure 12: Frame #11

Figure 13: Optical flow of feature points (red dots)

Main implementation of tracking is contained in **tracking.py**. The alternative implementation with the corner detection at each frame combined with optical flow depiction is implemented in **tracking_alternative.py**. Demonstrative code, which generates videos and pictures, is named **tracking_demo.py**, containing implementations for both of the methods (regular and alternative). Generated **videos are stored in the main folder**, while corresponding image sequences are located in *person_toy_slideshow* and *pingpong_slideshow* folders.

Conclusion

In this report we discussed several methods to detect points which are unique on a given picture. These algorithms can detect these features regardless of image rotation. We used approximations in our implementation of the Harris Corner Detector and because of this, our implementation gave somewhat different results for different orientations while still being able to detect sufficient amount of corners. Additionally, we attempted to estimate optical flow by use of the Lucas Kanade method on two synthetically generated images, with agreeable results. When applying this method on natural images for part 3, the produced vectors were more noisy, albeit still useful for feature tracking purposes. Solely relying on Lucas Kanade to estimate the positions of features, as was done in the final section, was shown to be susceptible to noise in the optic flow estimations, yielding moderately successful feature tracking. It is important to note that, even though the optical flows were not successful for the whole image sequence, it performed considerably well for the first 10-25 images, which could be potentially improved by application of corner detection once for each 20-30 images. In contrast to that, *alternative* approach utilizing Harris Corner Detector at each frame,

combined with optical flows, produced images, which are somewhat closer to the expectations of the feature tracking algorithms.

References

[1] Feature detection — opencv 2.4.13.7 documentation.

[2] B. K. Horn and B. G. Schunck. Determining optical flow. *Techniques and Applications of Image Understanding*, 281, 1981.

[3] S. Jianbo and C. Tomasi. Good features to track. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, page 593–600, 1994.