The KLT Tracker

Exercise 2

1 Introduction

During this exercise, you will derive and implement the original version and the OpenCV version of the widely used Lucas-Kanade feature tracker (LK tracker) [2]. The LK tracking algorithm is likely to be slow if we try to track a region around every pixel in the image so we will make the restriction to track only one or a few pixels, i.e. only the regions centered around one or a few pixels. Such pixels are typical chosen by sampling regularly or by choosing them based on the eigenvalues of the data matrix [3]. The most common name for this tracker is therefore Kanade-Lucas-Tomasi tracker (KLT tracker).

2 Derivations

The task is to derive the forward-additive and inverse-additive versions of the KLT for shift operations. The more general cases are treated in the literature [1].

Define the dissimilarity between two local regions, one in image I and one in image J:[1]

$$\epsilon = \iint_W [J(\mathbf{x}+\mathbf{d}) - I(\mathbf{x})]^2 w(\mathbf{x})\, d\mathbf{x} \tag{1}$$

where $\mathbf{x} = [x, y]^T$, the displacement $\mathbf{d} = [d_x, d_y]^T$. The integration region W is a local region around a pixel. The weighting function $w(\mathbf{x})$ is usually set to the constant 1, and we will for simplicity ignore the weight in the derivation from now on.

2.1 Forward-additive scheme

Equation 1 is identical to the equation given in the original work [2]. In that work, the Taylor series expansion of $J(\mathbf{x}+\mathbf{d})$ about the point $\mathbf{x}$, truncated to the linear term, is considered.

Task 1: Write down this expansion, the resulting approximation of (1), and its gradient with respect to $\mathbf{d}$. Use $\nabla J = [\frac{\partial J}{\partial x}, \frac{\partial J}{\partial y}]^T$ and ignore w.

Task 2: Set the derivative to zero and reorder terms to obtain the equation $\mathbf{Zd} = \mathbf{e}$. Give $\mathbf{Z}$ and $\mathbf{e}$. What is required of $\mathbf{Z}$?

---

[1] In optical flow you may think of images I and J as frames I_t and I_{t+1}.

The idea of the KLT algorithm is then to solve the original non-linear problem (1) by steepest descent iterations using the Taylor expansion from above:

1. Set $\mathbf{d}_{\text{tot}} = 0$.
2. Compute $\mathbf{Z}$ and $\mathbf{e}$ to get $\mathbf{d}$.
3. Add $\mathbf{d}$ to $\mathbf{d}_{\text{tot}}$ and compute the shifted image $J(\mathbf{x} + \mathbf{d}_{\text{tot}})$ and its gradient $\nabla J(\mathbf{x} + \mathbf{d}_{\text{tot}})$ by interpolating the original data.
4. If method has not yet converged, go to step 2.

Task 3: Draw a block diagram that outlines the main steps in the algorithm.

Due to the warping of J in step 3 in the iteration above, the expansion in the subsequent steps will be about the point $\mathbf{x} + \mathbf{d}_{\text{tot}}$. This is called the forward-additive scheme.

2.2 Inverse-additive scheme

In the OpenCV version of the KLT, the expansion is done about the point $\mathbf{x}$ in all steps. To avoid moving the expansion point with J, the expansion is computed for $I(\mathbf{x} - \mathbf{d})$ instead. The image J is still warped by $\mathbf{d}_{\text{tot}}$ in each iteration. This is called the inverse-additive scheme.

Task 4: Repeat the expansion above, write down the modified algorithm, and draw the modified block diagram.

Task 5: Discuss the advantages and drawbacks of the forward and inverse schemes in terms of computational complexity and noise in the images I and J.

---

3 The KLT Tracker

Here you will implement the blocks needed to create your own KLT tracker. We recommend to perform these tasks in Python/NumPy/SciPy, as this simplifies the use of OpenCV and ROS later.

3.1 Gradient Function

To be able to estimate the matrix **Z** needed by the KLT tracker we need a function that estimates the horizontal and vertical derivatives for all pixels.

Task 6: Implement a function that returns the regularized (lowpass-filtered) derivatives for an image.

As filter mask, use the Scharr filter with the following coefficient matrix (x-derivative, scaled by 32):

3	0	-3
10	0	-10
3	0	-3

3.2 Estimating Z

Task 7: Implement a function that estimates the matrix **Z** for a specified region.

Make this function general, i.e. make it possible to use different window sizes and also non-square windows. Use a default of 21×21 pixels.

3.3 Difference Function

To update the displacement with the KLT equations we need to estimate **e** for a specified region.

Task 8: Implement a function that estimates **e**.

Make this function general, i.e. make it possible to use different window sizes and also non-square windows. Use a default of 21×21 pixels.

3.4 Interpolation Function

During the gradient descent iterations, it is apparent that we need to obtain intensity values for non-integer pixel values. You need to implement a function to access these values.

Task 9: Create a function that returns the interpolated intensity values for all sub-pixels specified by a region of interest.

You might want to implement more than one interpolating function (default: bilinear interpolation) but that is not required to pass the exercise. Depending on your implementation, your region of interest might be the whole image. **Hint:** See the Python/SciPy function `interp2`.

3.5 Finalizing the KLT Tracker

Now when you have all the building blocks (the gradient function, the coefficient matrix function, the difference function, and the interpolating function) you should be able to finalize the inverse-additive KLT tracker for single scale tracking:

```
outputPoints = calcKLT(oldImage, newImage, inputPoints, None, winSize, maxIter, minDisp)
```

Task 10: Implement an iterative method to solve the displacement equation and to update the displacement vector until some stopping criterion is met, e.g. small enough displacement (default 0.01) or maximum number of iterations (default 30).

3.6 Test Implementation

You may use the provided two images `view0.png` and `view1.png` and consider the point $(320, 336)$. You will find OpenCV documentation under http://docs.opencv.org/3.1.0/.

Task 11: Check that your implementation is working correctly by comparing it to single scale tracking using OpenCV `calcOpticalFlowPyrLK()`.

Note that the method in OpenCV might make smaller steps in each iteration. Compare the result after several iterations. Use some image that you displace by some few pixels.

References

[1] S. Baker and I. Matthews. Lucas-Kanade 20 years on: A unifying framework. *International Journal of Computer Vision*, 56(3):221–255, 2004.

[2] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1981.

[3] Carlo Tomasi and Takeo Kanade. Detection and tracking of point features. Technical report, International Journal of Computer Vision, 1991.