Darius Rückert
Prathmesh Madhu
16.7.2020
LME
LGDV COMPUTER GRAPHICS • ERLANGEN

The deadline for this exercise is on Thursday 30.07.2020 at 12:00

Optical Flow

1 Introduction

In this exercise you will implement the Lucas-Kanade method for optical flow estimation. The optical flow describes the movement of objects in a video stream between two consecutive frames. We will compute a `sparse` optical flow at some distinctive keypoints on the screen. At each of these keypoints the optical flow Δu is the displacement of this point from the last frame I^{prev} to the next frame I^{next}.

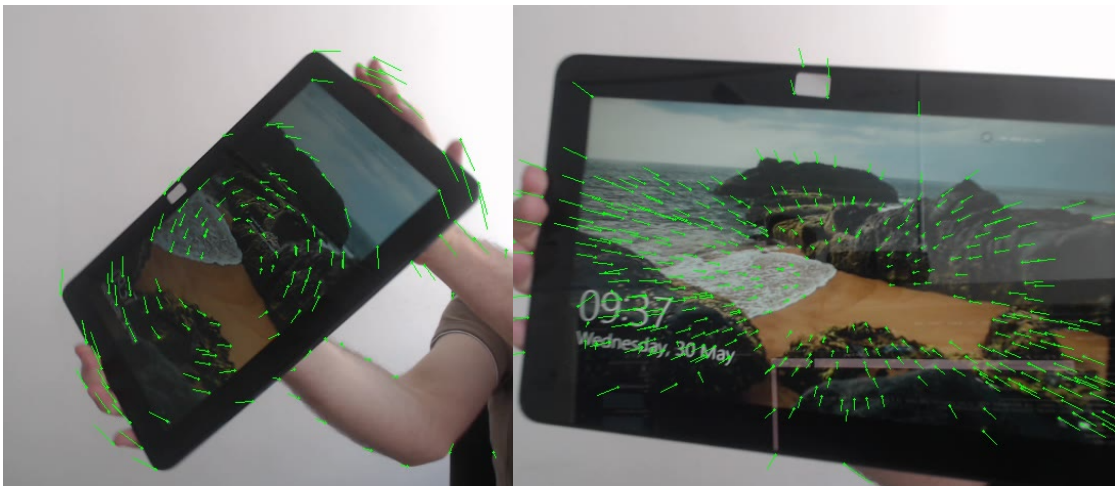


Figure 1: Optical flow for a rotational movement on the left and a movement towards the camera on the right. The green lines display the movement of the corresponding keypoints.

Assuming that the displacement of a pixel between two consecutive frames is small, the intensity of I^{next} at position u can be approximated by the Taylor-Expansion

$$I^{\text{next}}(u) = I^{\text{prev}}(u) + \begin{bmatrix} I_x^{\text{prev}}(u) & I_y^{\text{prev}}(u) \end{bmatrix} \begin{bmatrix} \Delta u_x \\ \Delta u_y \end{bmatrix},$$

where $I_x^{\text{prev}}(u)$ and $I_y^{\text{prev}}(u)$ are the spatial derivatives in x and y direction. In this form the system is under determined, because we have two unknowns $\Delta u_x, \Delta u_y$ and only one equation. To solve this problem, we assume that the optical flow is constant for a patch Ω centered around u. Each pixel $u_i \in \Omega$ now generates one equation resulting in the following system of linear equations:

$$WA\Delta u = W(b_{\text{next}} - b_{\text{prev}})$$

$$W \begin{bmatrix} I_x^{\text{prev}}(u_1) & I_y^{\text{prev}}(u_1) \\ I_x^{\text{prev}}(u_2) & I_y^{\text{prev}}(u_2) \\ \vdots & \vdots \\ I_x^{\text{prev}}(u_n) & I_y^{\text{prev}}(u_n) \end{bmatrix} \begin{bmatrix} \Delta u_x \\ \Delta u_y \end{bmatrix} = W \begin{bmatrix} I^{\text{next}}(u_1) - I^{\text{prev}}(u_1) \\ I^{\text{next}}(u_2) - I^{\text{prev}}(u_2) \\ \vdots \\ I^{\text{next}}(u_n) - I^{\text{prev}}(u_n) \end{bmatrix} \tag{1}$$

The diagonal matrix W contains per pixel weights. These are later used to give more importance to pixels close to the patch center. We can solve the linear system above with a least square approach:

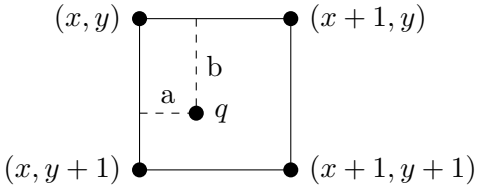
$$A^T W A \Delta u = -A^T W (b_{\text{next}} - b_{\text{prev}})$$

$$\Delta u = -(A^T W A)^{-1} A^T W (b_{\text{next}} - b_{\text{prev}})$$

2 Preparations

2.1 Bilinear Interpolation

In this exercise (and for many other computer vision applications) we will need to access an image at positions between two pixels. The value is computed from the surrounding pixels using bilinear interpolation:



$$\begin{aligned} I(q) =& (1-a)(1-b) \cdot I(x, y) \\ & a(1-b) \cdot I(x+1, y) \\ & (1-a)b \cdot I(x, y+1) \\ & ab \cdot I(x+1, y+1) \end{aligned}$$

Figure 2: Bilinear interpolation of image I at position $q = (x, y)$

Implement the function `computeBilinearWeights` in `util.py`.

2.2 Gaussian Weights

The LK-method assumes that the optical flow is equivalent for an entire patch centered around the current pixel. However, the further a pixel is away from the center the lesser they should contribute to the result.

Implement the function `computeGaussianWeights` in `util.py` that evaluates the two dimensional Gaussian distribution in a window of size `winsize`.

$$G(x, y) = e^{-\frac{\hat{x}^2+\hat{y}^2}{2\sigma^2}},$$

where $\hat{x}, \hat{y}$ are the centered and normalized coordinates of the pixels in this window:

$$\hat{x} = \frac{c_x - x}{\text{width}}$$

$$\hat{y} = \frac{c_y - y}{\text{height}}$$

2.3 Matrix inversion

Later in this exercise, you have to solve a 2×2 system of linear equations. For such a small matrix the inverted matrix is easily computed:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$
$$A^{-1} = \frac{1}{ad - cb} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Implement the matrix inversion in the function `invertMatrix2x2` of `util.py`.

2.4 Spatial Derivatives

The final piece of preparation is the computation of the image gradients $I_x^{\text{prev}}(u)$ and $I_y^{\text{prev}}(u)$ at the beginning of `compute` function of class `OpticalFlowLK` in the file `opticalFlowLK.py`. Use the function `Scharr` provided by Opencv and store the derivatives in the variables `prevDerivx` and `prevDerivy`.

Make sure to use the normalized (!!!) Scharr filter for image convolution.

$$\frac{1}{32} \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

3 Iterative Lucas-Kanade Solver

Computing the optical flow with equation (1) is not stable and only works for very small displacements. In this task we will use an iterative method to find the correct displacement vector u in the next image:

1. Initialize the target point u with the keypoint in the previous image $u = u_0$.
2. Compute $\Delta u = -(A^T W A)^{-1} A^T W (b_{\text{next}} - b_{\text{prev}})$, where
   A and b_{prev} is evaluated at u_0 and
   b_{next} is evaluated at u.
3. Update the target point $u = u + \Delta u$
4. If $|\Delta u|^2 > \epsilon$ go to 2.

Everything from here on has to be implemented in the function `compute` of class `OpticalFlowLK` of `opticalFlowLK.py`, which executes the algorithm above for every keypoint.

3.1 Precomputations

In step 2 of the algorithm, the variables A and b_{prev} are constant over all iterations. Therefore, we can compute these beforehand. Compute the following variables before the iterative loop in in the `compute` function:

- `bprev` $= b_{\text{prev}}$
- `A` $= A$
- `AtWA` $= A^T W A$
- `invAtWA` $= (A^T W A)^{-1}$

Note that the logic to iterate over the image patch is already provided. The position of the pixel relative to the top left corner of the patch is (x, y) and the position in the complete image is stored in (gx, gy). Make sure to use the bilinear interpolation weights `bw`. The Gaussian weights for the pixels in the image patch are stored in the variable `weights`. Note: Don't use matrix multiplication with `weights`, because this is not a diagonal matrix!!!

3.2 Iterative Solver

Compute the remaining part of step 2 in the iterative loop of `compute` of class `OpticalFlowLK`:

- `AtWbnbp` $= A^T W(b_{\text{next}} - b_{\text{prev}})$

As in the task before, make sure to use the bilinear weights `bw` for sampling in the next image.

After that, compute Δu and add it to u. Break out of the loop, if the condition of step 4. fails (ϵ is given in the variable `epsilon`).