Norwegian University of Science
and Technology
Department of Mathematical
Sciences

TMA4205 Numerical Linear Algebra Fall 2025

Semester project

1 Optical Flow

When a three-dimensional scene is observed through a camera lense (or an eye), the movements in the scene together with the relative movement of the lense give rise to an apparent motion within the image plane. This apparent motion is called the *optical flow* and can be used for example for object segmentation or collision detection. The goal of this project is the efficient implementation of a particular method for the estimation of this optical flow.

Denote in the following by $\Omega := [0, L_x] \times [0, L_y] \subset \mathbb{R}^2$ the image plane, and assume that we are given an image sequence (or movie) on Ω, modelled as a function $I \colon \Omega \times \mathbb{R} \to \mathbb{R}$. We are interested in estimating the optical flow at some fixed time $t_0 \in \mathbb{R}$, which we write as vector field $w \colon \Omega \to \mathbb{R}^2$ with components u and v, that is,

$$w(x, y) = \begin{pmatrix} u(x, y) \\ v(x, y) \end{pmatrix}.$$

Here u denotes the horizontal and v the vertical apparent movement in the image sequence. The basis for the estimation of this apparent movement is now the assumption that intensity values of the scene remain approximately constant along the flow. That is, for sufficiently small time steps Δt we have

$$I(x, y, t_0) = I(x + \Delta t\, u(x, y), y + \Delta t\, v(x, y), t_0 + \Delta t) + o(\Delta t).$$

Dividing by Δt and taking a limit $\Delta t \to 0$, this implies that

$$0 = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \big(I(x + \Delta t\, u, y + \Delta t\, v, t_0 + \Delta t) - I(x, y, t_0) \big) = u \partial_x I + v \partial_y I + \partial_t I.$$

That is, the functions u and v solve the optical flow equation

$$u \partial_x I + v \partial_y I = -\partial_t I. \tag{1}$$

Because this is a single equation for the two functions u and v, we cannot expect a unique solution. Roughly spoken, the equation (1) only provides a description of the optical flow across edges in the image, but not along edges. This non-uniqueness of the solution of the optical flow equation is called the *aperture problem* in optical flow. In order to tackle this problem and at the same time deal with approximation errors, it is necessary to include additional assumptions concerning the behaviour of the optical flow. One of the simplest assumption is smoothness: the values of u and v should change only slowly in space. One

possibility to realise this assumption is to require that the functions u and v solve the optimisation problem

$$\frac{1}{2}\|u\partial_x I + v\partial_y I + \partial_t I\|^2 + \frac{\lambda}{2}\big(\|\nabla u\|^2 + \|\nabla v\|^2\big) \to \min,$$

where $\lambda > 0$ is some regularisation parameter trading off smoothness of u and v against accuracy of the solution of the optical flow equation.

The Euler–Lagrange equations for this variational problem (that is, the optimality conditions for this optimisation problem) are the coupled PDEs

$$\begin{aligned}(u\partial_x I + v\partial_y I)\partial_x I - \lambda\Delta u &= -\partial_x I\,\partial_t I,\\ (u\partial_x I + v\partial_y I)\partial_y I - \lambda\Delta v &= -\partial_y I\,\partial_t I,\end{aligned} \qquad \text{in } (0, L_x)\times(0, L_y), \tag{2}$$

with either homogeneous Neumann boundary conditions both for u and v, that is,

$$\begin{aligned}\partial_x u = \partial_x v = 0 \quad &\text{on } \{0, L_x\}\times[0, L_y],\\ \partial_y u = \partial_y v = 0 \quad &\text{on } [0, L_x]\times\{0, L_y\},\end{aligned}$$

or homogeneous Dirichlet boundary conditions

$$u = v = 0 \qquad \text{on } \{0, L_x\}\times[0, L_y]\cup[0, L_x]\times\{0, L_y\} \tag{3}$$

if one wants to include the assumption that the flow at the boundary of the image plane is zero. This equation has been first proposed by Horn and Schunck [HS81] for a stable solution of the optical flow problem. More information and more general models can for instance be found in [BPS15].

See Figure 1 for an example of the resulting optical flow computed from two consecutive images in an image sequence.

2 Discretisation

For the discretisation, we assume that we are given two consecutive frames I_0 and I_1 of an image sequence at times $t = 0$ and $t = \Delta t$. The images I_0 and I_1 themselves are greyscale images given on a rectangular (pixel) grid of size $m \times n$. For simplicity, we assume that $\Delta t = 1$ and the grid size of the pixel grid is $\Delta x = \Delta y = h_0 := 1$.

In this case, the time derivative of I at $t = 0$ can be roughly approximated by

$$\partial_t I(x_i, y_j) := I_1(x_i, y_j) - I_0(x_i, y_j),$$

and the space derivatives by first approximating

$$\partial_x I_0(x_i, y_j) := \begin{cases} I_0(x_{i+1}, y_j) - I_0(x_i, y_j) & \text{if } i < m,\\ I_0(x_m, y_j) - I_0(x_{m-1}, y_j) & \text{if } i = m,\end{cases}$$

and

$$\partial_x I_1(x_i, y_j) := \begin{cases} I_1(x_{i+1}, y_j) - I_1(x_i, y_j) & \text{if } i < m,\\ I_1(x_m, y_j) - I_1(x_{m-1}, y_j) & \text{if } i = m,\end{cases}$$

Figure 1: Result of an optical flow computation. *First row:* Two consecutive frames in an image sequence. *Second row:* The resulting flow *(right)* and the first frame of the sequence overlaid with the flow *(left)*. The colourwheel on the bottom right indicates the direction and intensity of the flow field at each pixel. The test images have been taken from `http://vision.middlebury.edu/flow/`.

and then setting

$$\partial_x I(x_i, y_j) := \frac{1}{2}(\partial_x I_0(x_i, y_j) + \partial_x I_1(x_i, y_j)).$$

The y-derivative can be obtained similarly, by first setting

$$\partial_y I_0(x_i, y_j) := \begin{cases} I_0(x_i, y_{j+1}) - I_0(x_i, y_j) & \text{if } j < n, \\ I_0(x_i, y_n) - I_0(x_i, y_{n-1}) & \text{if } j = n, \end{cases}$$

and

$$\partial_y I_1(x_i, y_j) := \begin{cases} I_1(x_i, y_{j+1}) - I_1(x_i, y_j) & \text{if } j < n, \\ I_1(x_i, y_n) - I_1(x_i, y_{n-1}) & \text{if } j = n, \end{cases}$$

and then

$$\partial_y I(x_i, y_j) := \frac{1}{2}(\partial_y I_0(x_i, y_j) + \partial_y I_1(x_i, y_j)).$$

The flow field $w = (u, v)$ is discretised on the same pixel grid as the given image sequence, and for the discretisation of the Laplacian of u and v, we use the usual 5-point stencil, that is,

$$(\Delta u)_{ij} \approx (A_h u)_{ij} := \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}}{h^2}. \tag{4}$$

Moreover, we consider in the following the problem with Dirichlet boundary conditions (3). In which case you can set values outside the domain to 0.

In total, we obtain the coupled systems

$$\begin{aligned} (\partial_x I)_{ij}^2 u_{ij} + (\partial_y I)_{ij}(\partial_x I)_{ij} v_{ij} - \lambda(A_{h_0} u)_{ij} &= -(\partial_t I)_{ij}(\partial_x I)_{ij}, \\ (\partial_y I)_{ij}^2 v_{ij} + (\partial_y I)_{ij}(\partial_x I)_{ij} u_{ij} - \lambda(A_{h_0} v)_{ij} &= -(\partial_t I)_{ij}(\partial_y I)_{ij}, \end{aligned} \tag{5}$$

for $1 \le i \le m$ and $1 \le j \le n$.

3 Dealing with images and preprocessing

In PYTHON, images can be imported using the function `matplotlib.image.imread`. In order to display images, one can use the function `matplotlib.pyplot.imshow`. At the webpage of the course, a PYTHON program can be found for converting two-dimensional flow fields into RGB-images (`mycomputeColor`).

In case of fast movements in the images (or a low frame rate), the results of the optical flow computations improve, if one applies some smoothing to the input images I_0 and I_1. This can, for instance, be done by calling the function `scipy.ndimage.gaussian_filter(I0,sigma)`, which convolves the image `I0` with a Gaussian kernel with standard deviation σ. You are strongly recommended to apply this type of smoothing for all tests with natural (i.e., non-synthetic) images.

4 Problem Setting

4.1 Theoretical questions

1 Discuss the properties of the linear system (5). In particular, decide whether this system is symmetric, positive (semi-)definite, diagonally dominant, or irreducibly row diagonally dominant. What do your results imply for the convergence of the Jacobi method, the Gauß–Seidel method, and the CG method applied to this equation?

Hint: In order to show the positive definiteness, it makes sense to split up the operator on the left hand side of (5) *into two parts, the first containing only the (negative) Laplace parts, and the other part the multiplications with the derivatives of I. Both parts can be shown to be positive semi-definite, and the Laplacian part (with Dirichlet boundary conditions) is positive definite. Thus the whole operator is positive definite as well.*
Note that the situation is more complicated in the case of Neumann boundary conditions. Here, the negative Laplacian is only positive semi-definite, but no longer positive definite. Indeed, one can show that its kernel consists precisely of constant functions. In certain settings, it can then happen that specific constant functions are contained in the kernel of the other part of the operator (the part depending on the image sequence) as well, in which case the PDE either has no solution at all, or the solution is non-unique. [1]

4.2 Numerical methods

2 Implement a version of the conjugate gradient (CG) method for the solution of the optical flow problem (2) with Dirichlet boundary conditions (3). The method should contain a convergence test so that it terminates when

$$\frac{\|r_k\|_2}{\|r_0\|_2} < \text{tol},$$

where r_k denotes the residual at step k.

The implementation should consist of two functions, first a main function where the images are imported, preprocessed, and their derivatives are computed, and then a function where the actual CG method is implemented. The function header for the CG method could for instance look like this:

```
def OF_cg(u0,v0,Ix,Iy,reg,rhsu,rhsv,tol=1.e-8,maxit=2000):
'''
The CG method for the optimal flow problem
input:
    u0     - initial guess for u
    v0     - initial guess for v
    Ix     - x-derivative of the first frame
    Iy     - y-derivative of the first frame
```

[1] This non-uniqueness is strongly related to the so called *aperture problem* of motion perception, see e.g. https://en.wikipedia.org/wiki/Motion_perception#The_aperture_problem.

```
    reg    - regularisation parameter lambda
    rhsu   - right-hand side in the equation for u
    rhsv   - right-hand side in the equation for v
    tol    - relative residual tolerance
    maxit  - maximum number of iterations

output:
    u      - numerical solution for u
    v      - numerical solution for v

'''
```

3 Implement a multigrid V-cycle for the solution of the optical flow problem (2) with Dirichlet boundary conditions (3). Again, implement a main function from which the V-cycle is called, and another function containing the actual implementation of the V-cycle. This function should take as input the initial guess, the right-hand side, the current level and maximal level of the grid, the number of pre-smoothings, and the number of post-smoothings. Use the CG algorithm from **2)** to solve the problem on the coarsest level. For the smoother, a red-black Gauss–Seidel (preferably) or the weighted Jacobi method should be implemented.

It is advantageous to further break down the function where the V-cycle is performed into several parts. The main routine for the multigrid cycle could for instance look like this (different setups are possible, though):

```
def V_cycle(u0, v0, Ix, Iy, lambda, rhsu, rhsv, s1, s2, level, max_level):
'''
     V-cycle for the optical flow problem.

     input:
     u0        - initial guess for u
     v0        - initial guess for v
     Ix        - x-derivative of the first frame
     Iy        - y-derivative of the first frame
     reg       - regularisation parameter (lambda)
     rhsu      - right-hand side in the equation for u
     rhsv      - right-hand side in the equation for v
     s1        - number of pre-smoothings
     s2        - number of post-smoothings
     level     - current level
     max_level - total number of levels

     output:
     u      - numerical solution for u
     v      - numerical solution for v
'''

    u,v = smoothing(u0, v0, Ix, Iy, reg, rhsu, rhsv, level,s1)
    rhu,rhv = residual(u, v, Ix, Iy, reg, rhsu, rhsv)
```

```
    r2hu,r2hv,Ix2h,Iy2h = restriction(rhu, rhv, Ix, Iy)

    if level == max_level - 1
        e2hu,e2hv = OF_cg(np.zeros_like(r2hu), np.zeros_like(r2hv),
             Ix2h, Iy2h, reg, rhsu, rhsv, 1e-8, 1000, level+1)
    else
        e2hu,e2hv = V_cycle(np.zeros_like(r2hu), np.zeros_like(size(r2hv)),
             Ix2h, Iy2h, reg, r2hu, r2hv, s1, s2, level+1, max_level)
    ehu,ehv  = prolongation(e2hu, e2hv)
    u = u + ehu
    v = v + ehv
    u,v = smoothing(u, v, Ix, Iy, lambda, rhsu, rhsv, level, s2)
    return u, v
```

Notice that the function is defined recursively. What is left then is to implement the functions `smoothing`, `residual`, `restriction` and `prolongation`. Also, note that the CG method also requires some information of the level (or, alternatively, the current grid size), as this influences the discretisation of the Laplace term.

Since typical images have sizes that are highly divisible by powers of two, (e.g., images of size 512×768) it makes sense to define the restriction operator in such a way such that the size of the images is exactly halved in each restriction step. This can be achieved by defining the restriction operator using the stencil

$$\frac{1}{4}\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}.$$

This corresponds to using a staggered grid in the multi-level approach, that is, the grid points in the coarsened grid lie between (and not on) grid points in the fine grid.

4 Modify the routine in **2)** to solve the optical flow problem with a PCG method using a multigrid V-cycle as in **3)** as preconditioner. That is, instead of solving the linear system $Mz = r$ that is usually required in each PCG iteration, we compute z by applying the multigrid solver (with a reasonably small number of smoothing steps s_1 and s_2) with right hand side r. This preconditioner, however, has to be applied with some care. What properties must a preconditioner for the CG method have, and what does this mean for the multigrid V-cycle? Is it necessary to modify the setup of the red-black Gauß–Seidel method, and if yes, how?

5 Numerical tests

The different methods should first be tested on the synthetic test "images" that can be generated with the function `generate_test_images` downloadable on the webpage of the course. The first testcase implemented there produces the $(N \times N)$ image of a Gaussian that moves to the lower right. This can be used for testing the code, as the method should be able to almost reproduce this movement in the center of the image. Because of the Dirichlet boundary conditions, the movement tends to zero on the boundary of the image, though.

The second testcase implemented in `generate_test_images` produces images of two Gaussians circling each other. Test all the different methods that were implemented on this synthetic example with image size $2^k \times 2^k$ for $k = 6, \ldots, 9$, and discuss the convergence behaviour of the different methods for each of these cases, and also how the different methods scale with the image size (feel free to test even larger images). Use a regularisation parameter $\lambda_k = 4^{k-4}$ (depending on the image size) for these tests. How do the parameters within the multigrid method (that is, number of levels and number of pre- and post-smoothing iterations) influence your results?

Additionally, test the different methods using the images `frame10.png` and `frame11.png` from the webpage of the course. Additional test sequences can also be downloaded from `http://vision.middlebury.edu/flow/`. How do the computation times change with λ? Also, discuss how the parameters of the methods influence the number of iterations and the computation time. For which method and parameter setting can one obtain the best possible computation times?

Use the condition $\|r_k\|_2/\|r_0\|_2 < 10^{-8}$ as convergence criterion in all the computations.

A Matrix Operations

Use matrix- or vector-operations and avoid costly loops whenever possible. As an example, do not evaluate the central part of the negative Laplacian operator using the double loop

```
n, m = u.shape
laplace_u = np.zeros_like(u)
for i in range(1,n-1):
  for j = in range(1,m-1):
    laplace_u[i,j]= -(4*u[i,j] - u[i-1,j] - u[i,j-1] - u[i+1,j] - u[i,j+1])
  end
end
```

but rather with

```
  n, m = u.shape
  laplace_u = np.zeros_like(u)
  laplace_u[1:n-1,1:m-1] = -4*u[1:n-1,1:m-1] + u[0:n-2,1:m-1] + u[2:n,1:m-1] \
                         + u[1:n-1,0:m-2] + u[1:n-1,2:m]
```

An implementation with a (sparse!) matrix vector multiplication is of course also possible, and there exist many other (also more efficient) implementations than that one. The important point is, not to use double loops here.

References

[BPS15] F. Becker, S. Petra, and Ch. Schnörr. Optical flow. In *Handbook of mathematical methods in imaging. Vol. 1, 2, 3*, pages 1945–2004. Springer, New York, 2015.

[HS81] B. K. P. Horn and B. G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.