

VISION: Practical work – implementation of several optical flow methods

D. Béréziat
Sorbonne Université - Master IMA/DIGIT

February 4, 2025

Start up From the course web page: <http://pequan.lip6.fr/~bereziate/cours/master/vision/>, get the practical work archive, unzip it in your home directory.

Due date By Tuesday at noon, February 11 via Moodle. Provide an archive containing the following files in a directory having your names. You are strongly encouraged to work in pair. The archive will contain the following files:

- `horn.py`, `lucas.py`, `nagel.py`, `gradhorn.py`, `horn_xxx.py`, `lucas_xxx.py`, `nagel_xxx.py`, where `xxx` stand for one of seven data to process,
- you can also provide Jupyter notebooks or Matlab files (if you prefer Matlab),
- a short report where you discuss the results (do not detail the methods), and hyper-parameters. Each experiment would be illustrated by figures and a table of statistics when the ground truth is available. It also would be useful to compare the two methods between them,
- do not provide data sets in the archive unless it is new data.

1 Horn-Schunck method

Algorithm

1. Read two images I_1, I_2 (same size and dimensions)
2. Compute I_x, I_y , and I_t (see next subsection)
3. Choose N (number of iterations) and α (regularization)
4. $u^0 = v^0 = 0$, two images having same size than I_1
5. For $k = 0$ to $N - 1$:

(a) compute $\bar{u}^k = A \star u^k, \bar{v}^k = A \star v^k$ with $A = \begin{pmatrix} \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \\ \frac{1}{6} & 0 & \frac{1}{6} \\ \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \end{pmatrix}$

(b) compute: for i line index and j row index

$$\begin{aligned} u^{k+1}(i, j) &= \bar{u}^k(i, j) - I_x(i, j) \frac{I_x(i, j)\bar{u}^k(i, j) + I_y(i, j)\bar{v}^k(i, j) + I_t(i, j)}{\alpha + I_x^2(i, j) + I_y^2(i, j)} \\ v^{k+1}(i, j) &= \bar{v}^k(i, j) - I_y(i, j) \frac{I_x(i, j)\bar{u}^k(i, j) + I_y(i, j)\bar{v}^k(i, j) + I_t(i, j)}{\alpha + I_x^2(i, j) + I_y^2(i, j)} \end{aligned}$$

6. Visualization with function `flowToColor()`

7. If available: read the ground truth with the function `readFlowFile()` and compare with (u^N, v^N) (see next subsection).

Determination of image gradient

Horn and Schunck proposed the following filter:

$$\begin{aligned}I_x(i, j) &\simeq \frac{1}{4} [I_1(i, j+1) - I_1(i, j) + I_1(i+1, j+1) - I_1(i+1, j) \\&\quad + I_2(i, j+1) - I_2(i, j) + I_2(i+1, j+1) - I_2(i+1, j)] \\I_y(i, j) &\simeq \frac{1}{4} [I_1(i+1, j) - I_1(i, j) + I_1(i+1, j+1) - I_1(i, j+1) \\&\quad + I_2(i+1, j) - I_2(i, j) + I_2(i+1, j+1) - I_2(i, j+1)] \\I_t(i, j) &\simeq \frac{1}{4} [I_2(i, j) - I_1(i, j) + I_2(i+1, j) - I_1(i+1, j) \\&\quad + I_2(i, j+1) - I_1(i, j+1) + I_2(i+1, j+1) - I_1(i+1, j+1)]\end{aligned}$$

Using provided functions

Functions `computeColor()` and `readflow()` take as parameter or return an array of dimension 3. The two first dimensions are the image coordinates, the third dimension is of size 2, and describe the 2 components of the velocity vector. Here an example in Python that reads an image and display the image gradient map with `computeColor()`:

```
from PIL import Image
nasa = Image.open('nasa9.png')
from numpy import gradient
Ix, Iy = gradient(nasa)
import matplotlib.pyplot as plt
from middlebury import computeColor
plt.imshow(computeColor(Ix, Iy))
plt.show()
```

Or in Matlab:

```
nasa = imread('nasa9.png');
[Ix, Iy] = gradient(nasa);
w(:, :, 1) = Ix;
w(:, :, 2) = Iy;
imagesc(FlowToColor(w))
```

Expected work

1. Write the function `gradhorn(I1, I2)` that takes two images as parameter and returns the spatio-temporal gradient of this pair of images.
2. Write the function `horn(I1, I2, alpha, N)` that returns compute the velocity map between images I1 and I2 according to the Horn and Schunck method.
3. Test your function on available data. For each data, you would provide a specific script that typically loads the data, computes the optical flow, displays the velocity map with `computeColor()`. Moreover, if a ground truth is available, you would compute several statistics (mean, standard deviation) of End

Point error, angular error, norm error. Angular error (in space-time, see lecture) at pixel (i, j) is given by:

$$\theta(i, j) = \arccos \left(\frac{1 + w_r(i, j) \cdot w_e(i, j)}{\sqrt{1 + \|w_r(i, j)\|^2} \sqrt{1 + \|w_e(i, j)\|^2}} \right)$$

where w_r is the ground truth (reference) and w_e is the estimated map.

Tips:

- First, verify that the function `gradhorn()` is correct (check on data `square` for instance)
 - Test la fonction `horn()` on data `mysine`. Find the optimal value `alpha` minimizing the angular error or EPE.
 - Verify the velocity map obtained by your algorithm is visually coherent with the change between I_1 and I_2
4. **Bonus:** with the help of function `quiver()` visualize the velocity map as an under-sampled field of vectors (for instance, display 1 vector among 10). A scaling parameter can also be applied if the vectors has a low magnitude. Another data are available on web site mentioned in the lecture.

2 Lucas-Kanade method

Algorithm

1. Read two images I_1 and I_2
2. Determine I_x , I_y and I_t (with the Horn-Schunck gradient filter)
3. Choose a window size n
4. For each pixel p in the image domain (excepted pixels on domain border, i.e. those at distance $\frac{n}{2}$ from the border), do:
 - (a) Center a squared window W , of size $n \times n$, on pixel p
 - (b) Form the column vector $B = (-I_t(p'))_{p' \in W}$ (of length n^2)
 - (c) Form the matrix $A = ((I_x(p'))_{p' \in W} \quad (I_y(p'))_{p' \in W})$ (of dimension $n^2 \times 2$)
 - (d) Compute velocity at pixel p : $w = (A^T A)^{-1} A^T B$

Expected work

Repeat the questions from the previous section, but considering Lucas-Kanade method. For each experiment, try to find the best parameter (n) and compare to the best run of Horn-Schunck. Test alternative neighboring system, for instance a Gaussian window W . The same tips given in the previous section also apply here.

3 Nagel method

This method is similar to Horn and Schunck ones. However it requires second spatial derivatives from images I_1 and I_2 : I_{xx} , I_{yy} , and I_{xy} . Theses derivatives can be obtained by applying twice the `gradhorn()` procedure. The iterative scheme (see lecture II, slide 8) also applies spatial derivatives on scalar image f : f_x , f_y , $f_{xy} = \frac{\partial f_x}{\partial y}$. We suggest to use the following operators: $f_x(i, j) \simeq \frac{f(i, j+1) - f(i, j-1)}{2}$ for horizontal derivative, and $f_y(i, j) \simeq \frac{f(i+1, j) - f(i-1, j)}{2}$ for the vertical derivative.

Algorithm (see slides 6, 7, and 8 lecture II)

1. Read two images I_1, I_2 (same size and dimensions)
2. Compute $I_x, I_y, I_t, I_{xx}, I_{yy},$ and I_{xy}
3. Set N (number of iterations), α (regularization), δ (oriented regularization)
4. $u^0 = v^0 = 0$, two images having same size than I_1
5. For $k = 0$ to $N - 1$:
 - (a) compute $\tilde{u}^k = \eta(u^k), \tilde{v}^k = \eta(v^k)$ with:
 - η a function applying on a scalar image f such as $\eta(f) = \bar{f} - 2I_x I_y f_{xy} - q \nabla f$
 - $q = \frac{1}{I_x^2 + I_y^2 + 2\delta} \nabla I^T \left[\begin{pmatrix} I_{yy} & -I_{xy} \\ -I_{xy} & I_{xx} \end{pmatrix} + 2 \begin{pmatrix} I_{xx} & I_{xy} \\ I_{xy} & I_{yy} \end{pmatrix} V \right]$
 - $V = \frac{1}{I_x^2 + I_y^2 + 2\delta} \begin{pmatrix} I_y^2 + \delta & -I_x I_y \\ -I_x I_y & I_x^2 + \delta \end{pmatrix}$
 - and $\bar{f}$ is as in Horn and Schunck algorithm step 5.(a)
 - (b) compute: for i line index and j row index

$$\begin{aligned} u^{k+1} &= \tilde{u}^k - I_x \frac{I_x \tilde{u}^k + I_y \tilde{v}^k + I_t}{\alpha + I_x^2 + I_y^2} \\ v^{k+1} &= \tilde{v}^k - I_y \frac{I_x \tilde{u}^k + I_y \tilde{v}^k + I_t}{\alpha + I_x^2 + I_y^2} \end{aligned}$$

4 Use automatic differentiation and optimizer (bonus)

One can leverage the capability of deep learning framework (such as Pytorch) to automatically determine the gradient of a cost function and optimizer to minimize the cost function.

- You will be able to write in Python a function taking as parameters the velocity field u and v , and the image gradient I_x, I_y, I_t and returning the Horn and Schunck cost function (i.e. $\|I_x u + I_y v + I_t\|_2^2 + \alpha(\|\nabla u\|_2^2 + \|\nabla v\|_2^2)$).
- The optimization procedure is similar to that used in deep learning training:
 - initialize the optimizer, the parameters being the values of images u and v . We **strongly** suggest to use LBFGS (see example below),
 - iterate the following procedure: set to zero the gradient buffer, compute the cost function, retro-propagate the gradient, perform a steepest descent.

- Example of steepest descent with Pytorch LBFGS:

```
optimizer = torch.optim.LBFGS(...)
def closure():
    optimizer.zero_grad() # nullify gradient
    L = costfunction(...) # compute the cost
    L.backward()           # retropropagate and one step of descent direction
    optimizer.step(closure) # perform the optimization steps
```

- Experiment with L_2 norms and compare with Horn and Schunck. Experiment with L_1 norms and discuss the results.