Exercise – Optical Flow

Mathematical Modelling
02526

The goal with this exercise is to use mathematical modelling to estimate movement in sequences of images (video). This will be done using optical flow. Optical flow has a broad range of applications in computer vision, robot navigation, artificial slow motion and more.

You will:

- Identify significant issues and use cases for optical flow
- Formulate a mathematical model for optical flow
- Implement a solution to the optical flow using either MATLAB or PYTHON
- Collect your own datasets for analysis
- Create a visualization of the estimated motion in the data.
- Report the results from the analysis and discuss the validity of the model

The report for this exercise must be 5 pages at most, including graphs, tables, and images (excluding front page and appendices). While this exercise is broken into a number of sub-tasks and questions, the report is not a list of answers, but instead be a coherent documentation and discussion of the analysis performed.

The exercise consists of three primary components, which should also be reflected in the report.

- **Image filters:** This bread-and-butter operation in image analysis can do most of the processing needed to compute the optical flow.
- **Optical flow:** The mathematical model and this practical technique to image processing that you should implement and test. Seeking out more information is highly encouraged, to determine the relevance of the problem and putting it in perspective.
- **Practical experiment:** Test your optical flow implementation on real videos. You can acquire video-data using your laptop-camera or mobile phones. Under what conditions does optical flow work well, and when does it break down.

Use MATLAB or PYTHON for implementation.
Suggested packages for PYTHON: NumPy, SciPy, Matplotlib, scikit-image.

Optical flow / Optic flow

Optical flow can be thought of as a generic concept for determining apparent motion of objects in a video. It has wide range of applications and the methodology come in countless variations, concerning both the mathematical model, implementations for speed and efficiency etc.

Preparation: Watch the following two videos from Computerphile, which should take roughly 20 minutes in total, [YouTube link 1] and [YouTube link 2]. (They are quite decent, which is a generally valid statement for lots of concepts within computer science.) Watch these videos with the following study guides in mind:

When you have seen these videos you should be able to:

- Explain what the optical flow model allows you to do in lay-term terms
- State the (primary) mathematical operation used for optical flow estimations
- List at least 3 use-cases or applications of optical flow
- List at least 3 potential limitations of optical flow

You can also look up additional information yourself. Here are some suggestions on where to start:

- This blog post [link] is also quite good, with demonstration/inspiration for potential extensions and recent developments within the field.
- For a more 'classic' published source [1][link].

The mathematical model: A sequence of image frames (a video) can be represented as a 3D structure/matrix, $\mathbf{V}$, where the third (temporal) dimension represents the forward increments in time.

An explanation of optical flow starts with considering one point, $\mathbf{p} = [p_x, p_y, p_t]^T$, within in the image domain and the problem of finding a displacement $\mathbf{u} = [x, y, 1]^T$ such that

$$\mathbf{V}(\mathbf{p}+\mathbf{u}) - \mathbf{V}(\mathbf{p}) = 0\,.$$

This can be thought of as a **brightness constancy constraint**.

Under the assumption of a **small displacement**, a Taylor expansion (keeping only first order components) can be shown to yield,

$$\mathbf{V}(\mathbf{p}+\mathbf{u}) = \mathbf{V}(\mathbf{p}) + [\mathbf{V_x}(\mathbf{p}) \quad \mathbf{V_y}(\mathbf{p}) \quad \mathbf{V_t}(\mathbf{p})]\ \mathbf{u}$$

where we use notation $\mathbf{V_x} = \frac{\partial \mathbf{V}}{\partial x}$, and correspondingly for partial derivatives in y and t direction.

This reduces to

$$\mathbf{V_x}(\mathbf{p})x + \mathbf{V_y}(\mathbf{p})y + \mathbf{V_t}(\mathbf{p}) = 0$$

or,

$$\mathbf{V_x}(\mathbf{p})x + \mathbf{V_y}(\mathbf{p})y = -\mathbf{V_t}(\mathbf{p}) \tag{1}$$

This is ONE equation with TWO unknowns and a solution can not be determined without providing further constraints or regularization. Different solutions have been proposed,

- Horn-Schunk [2]
- Lucas-Kanade [3][link]
- and many more ...

For this exercise, we suggest using the Lucas-Kanade solution. This involves considering a small neighborhood and assuming that all pixels experience **the same displacement**. In that case, every pixel $\mathbf{p}_i \in N$ contributes with one equation. This gives an overdetermined system,

$$\begin{bmatrix} \vdots & \vdots \\ \mathbf{V_x}(\mathbf{p}_i) & \mathbf{V_y}(\mathbf{p}_i) \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = - \begin{bmatrix} \vdots \\ \mathbf{V_t}(\mathbf{p}_i) \\ \vdots \end{bmatrix} \tag{2}$$

Finding the linear least squares solution to such an overdetermined system, $\mathbf{Ax} = \mathbf{b}$, is quite simple.

Problem 1 - Loading and displaying a toy problem: This is a warm-up exercise, assuming that you have not worked with video processing previously. A good place to start, is to load and use a small toy problem for developing your implementation of optical flow.

The toy problem is simply a sequence of 64 (.png) images located on *DTU LEARN*. The dimension of each frame: [256, 256]. Load it into a 3D matrix or numpy array. Remember to convert from RGB `uint8` to grayscale `float` in range [0,1]. Display all or a subset of the frames to get a sense of the scene. Try to display the frames on the same figure with a slight update delay (i.e. an animation of the frames).

- MATLAB: `shg` and `pause()`
- PYTHON: `matplotlib.pause()`

Part 2 - Image Gradients

The primary processing step is the calculation of the image gradients $\mathbf{V_x}, \mathbf{V_y}, \mathbf{V_t}$.

Problem 2.1 - Low level gradient calculation: The most simple way is to approximate the gradient as the difference between neighboring pixels/voxels.

In a 1D signal, $\mathbf{x}$, this would be:
MATLAB: `dx = x(2:end) - x(1:end-1)`
PYTHON: `dx = x[1:]  - x[0:-1]`

Obtain $\mathbf{V_x}, \mathbf{V_y}, \mathbf{V_t}$ for the entire volume and display for some selected frames.
Consider what happens with the image/volume size, when performing gradient calculations like this?

Problem 2.2 - Simple Gradient Filters: Image filtering is an efficient and convenient way of doing some form of computation on a local neighborhood of the image and repeating it across the entire domain. For a more complete introduction - check this [video]. This also includes calculating derivatives.

Generally, image filtering (`imfilter()` and `fspecial()` in MATLAB and `scipy.ndimage` in PYTHON) requires a least two inputs:

- The image/volume, $\mathbf{V} \in [N_x \times N_y \, (\times N_z)]$.
- The kernel (or simply filter), which is a small vector, patch or cube of size N defining the processing to be done.
- This operation comes with some options for boundary handling, output size etc. Check the documentation for more information.

The kernel: For the problem of calculating gradients, one can use the classic *prewitt* or *sobel* kernels. Generate these kernels and plot them. Does it return the horizontal or vertical gradient? How do you flip the kernel to be in other directions?

Image filtering: The filtering operation works by convolving the kernel with the current overlap of the image, and moves the kernel iteratively across the entire image. Run the image filtering separately(!) in each dimension to obtain $\mathbf{V_x}, \mathbf{V_y}, \mathbf{V_t}$. Plot and compare to the results from problem 2.1.

Problem 2.3 - Gaussian Gradient Filters: The *prewitt* and *sobel* filters are nice and simple because they have a fixed size of $[3 \times 3]$. It is however easy to imagine cases, where one might want to calculate the gradient using a larger region of the image. The Gaussian kernel is a compelling alternative for such a task - see this [Wiki page] for more information.

In 1D, the formula for the Gaussian is,

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \, e^{-\frac{x^2}{2\sigma^2}} \tag{3}$$

The only control parameter is σ, which handles the spread/variance (i.e. allowing pixels closer to the kernel center to be more influential). The patch size of a Gaussian kernel is not fixed, but can also be controlled by σ. I.e. the patch need to be no bigger than some multiplier of σ - typically 4-6 σ.

Extending the Gaussian to more dimensions is quite simple. In 2D, it has the following form,

$$G(x, y) = \frac{1}{2\pi\sigma^2} \, e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Justify that $G(x, y) = G(x) \cdot G(y)$ and extend to 3D. This property is called *separability*, and proves very handy when doing image filtering - see this [Wiki page] for more information. Basically, instead of creating a 2D or 3D kernel, one can simply use 1D kernels consecutively. I.e. filter the original image with the 1D kernel ($G(y)$) as a column vector, then filter the resulting output image with the (same) kernel ($G(x)$) as a row vector), and so forth.

Finally, it should be noted that the normal Gaussian kernel is a smoothing filter! In order to make it into a gradient filter, you will need the derivatives.

Start in the 2D case and make the derivative with respect to x, i.e. $\frac{d}{dx}\left(G(x, y)\right)$. Use the separability property and the product rule for differentiation.

Extend to 3D, and repeat for $\frac{d}{dy}$ and $\frac{d}{dt}$.

Justify that in the end, obtaining estimates for $\mathbf{V_x}$, $\mathbf{V_y}$ and $\mathbf{V_t}$ boils down to a series of image filter operations with the 1D Gaussian kernel (Equation 3) and its derivative.

Finally, plot and compare to the image gradient results from problem 2.1 and problem 2.2. Vary σ to see the effect on the gradient estimation.

Part 3 - Lucas-Kanade solution to optical flow:

NB: You are of course welcome to work with a different solution to the optical flow problem! The suggestion is to try with Lucas-Kanade solution (Equation: 2). Start simple (problem 3.1) and make it work for just ONE single local region of ONE frame. Then expand to ONE full frame, and finally calculate a solution for ALL frames (problem 3.2).

Problem 3.1 - Local and low level solution:

- Select one pixel in a single frame, $\mathbf{p} = (x, y, t)$
- Extract all gradients in a $N \times N$ region around $\mathbf{p}$ from $\mathbf{V_x}$, $\mathbf{V_y}$ and $\mathbf{V_t}$.
- Set up matrices $\mathbf{A}$, $\mathbf{b}$ (see Equation 2) and find the least squares solution.

Plot the resulting displacement vector on the image frame. Consider if N should be uneven or even. Consider how to handle the boundary of the image (or ignore the boundary).

MATLAB hint: "\" operator
PYTHON hint: `numpy.linalg.lstsq()`

Problem 3.2 - Apply densely to full volume: You should now be ready to add some `for`-loops around problem 3.1, to find a solution for "all" voxels in a frame. It is possible to include a 'stride' in the $x-$ and $y-$direction, i.e. making small jumps such that only every n'th pixel is visited.

Strategy:

- Start by processing a single frame, calculate the solution "for each visited voxel" and plot the vector field.
- Loop through all frames. End by displaying the image sequence animation (as in Part 1, problem 1) but with the vector field included.

Does it work well? Equally in all regions of the image?
Try selecting different window sizes and see the effect?

Problem 3.3 - Visualizing the resulting motion fields: As there is no ground truth motion available for the data, the assessment is purely qualitative, and thus relies on good visualization. Plotting a vector on every pixel quickly becomes too dense to really make sense of the motion. There is room for creativity here, but you could consider: to only show a subset of vectors; to filter away outlier or insignificant motion vectors, based on e.g. vector length; making a direction colormap, as an alternative to plotting the result as a vector field.

Optional Problem 3.4 - Expanding the solution: There are countless options for expanding the implementation. Consider implementing an interest-point-detector (as done here), thus going in the direction of object tracking throughout the frames.

Part 4 - Experimental part: Make videos and test optical flow

You should now have a simple working implementation.

The purpose of this part, is for you to create your own data/video. You can get far with your smart phone camera! Exploring the limitations of a method or model is often a great way to deepen your understanding. There is a lot of proper engineering and insights in data acquisition and practical experiments.

Figuring out how to import the video to your computer and process it may take a few tries. Remember that processing time scales with resolution (spatial and temporal) and video-length!
MATLAB: You can use `videoReader()`. Check the documentation.
PYTHON: There are numerous tools for reading videos. You can see some suggestions here. As an alternative, look up online tools for turning a video into an image sequence.

Problem 4.1. - An optical flow friendly video: The optical flow method is by no means perfect. Can you create a video where it actually works decently?

Problem 4.2. - An optical flow adverse video: Can you create a video where the optical flow model (the one you implemented) "fails"? It is easy to severely break some of the underlying assumptions. The purpose is more for you to create a challenging scenario, where you try to find one or more of the model limits.

Hints: Speed vs. frame rate, the aperture problem, lighting conditions and object textures.

Part 5 - Optional: Optical flow via Gaussian filters

Note: You should have solved problem 2.3, before continuing with this bit.

In problem 3.2, we moved a window across the image domain to select pixels in which to solve the optical flow (Lucas-Kanade) problem. This operation has the distinct smell of an image filter! Except that image filters don't pick pixels for you. They will also "sum" them together ...

On the other hand, when we do a least squares solution the content is also "summed" together, so there must be a link!

Generically, the Lucas-Kanade solution boils down to solving a linear system of equations (in a least square sense): $\mathbf{Ax} = \mathbf{b}$

Now, let us expand the equation with the system matrix transposed: $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$

Try to work out what $(\mathbf{A^TA})^{-1}$ and $\mathbf{A}^T\mathbf{b}$ becomes once you insert $\mathbf{V_x}, \mathbf{V_y}, \mathbf{V_t}$ accordingly.

Justify that the system to solve is

$$\begin{bmatrix} s_{xx} & s_{xy} \\ s_{xy} & s_{yy} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = - \begin{bmatrix} s_{xt} \\ s_{yt} \end{bmatrix} \tag{4}$$

where s_{xx} for can be computed by image filtering (with a Gauss kernel for example) of the volume $\mathbf{V}_x^2$ (i.e. element wise multiplication of $\mathbf{V}_x \cdot \mathbf{V}_x$).

Reporting

Contents of the report

1. Describe the problem and the background – what is modelled and why?
2. Describe the data and the experiments
3. Describe the mathematical model and the image processing methodology?
4. Describe and visualize the results.
5. Discuss the results – how good are the they? To what degree do they reflect the true problem?
6. Conclude – what is the contribution of the analysis?

Note, that the page-limit of 5 can quickly be reached if you want to present graphical results from all of the videos (toy problem and your own). It it fine to include some processed frames in the appendix, and you can in principle also upload your processed videos to DTU Learn. Just make sure that the story in the report is consistent and self-contained.

Hand in: The report is uploaded to Peergrade via DTU Learn. The code must be included in an appendix in the report.

afhar, 2023
Special thanks to Vedrana Andersen Dahl and Hans Martin Kjer.

References

[1] John L Barron, David J Fleet, and Steven S Beauchemin. Performance of optical flow techniques. International journal of computer vision, 12(1):43–77, 1994.

[2] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. Artificial Intelligence, 17(1):185–203, 1981.

[3] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. Vancouver, British Columbia, 1981.