Danmarks Tekniske Universitet

02526 - Mathematical Modeling

Optical Flow

Aksel Buur Christensen - s203947
Karen Witness - s196140
Morten Westermann - s203965
Viktor Isaksen - s203957

February 19, 2023

1 Loading and displaying

Optical flow (OF) is the study of apparent motion in an image or frame. This can be used in many different fields for example in object detection through images and videos[1]. However, OF is hard to implement, so a few assumptions need to be made in order to calculate the OF and this naturally imposes limits to the use cases. In this OF project the problem will be solved by the Lucas-Kanade method[2] which assumes lighting is constant and that flow is constant in the local neighbourhood for the considered pixel.

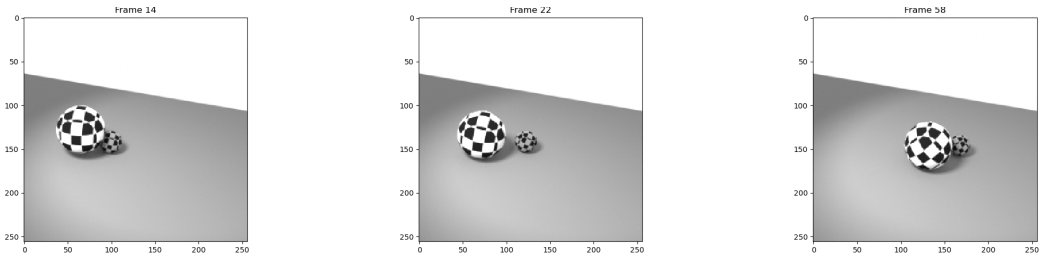


Figure 1: *Three selected frames from the "Toy-problem" animation, it contains in total 64 frames of 256x256 pixels. These frames are grayscaled to obtain pixel values between 0 and 1.*

2 Image Gradients

Here are a few different methods to calculate the image gradients $\mathbf{V_x}$, $\mathbf{V_y}$, $\mathbf{V_t}$.

2.1 Low-level gradient calculation

The first method is to approximate the low-level gradient by calculating the difference between rows of the entire volume (3D image stack), using a simple kernel (-1,1). See appendix (8.1). Obtaining $\mathbf{V_x}$, $\mathbf{V_y}$, $\mathbf{V_t}$, selected frames is displayed in figure (2).

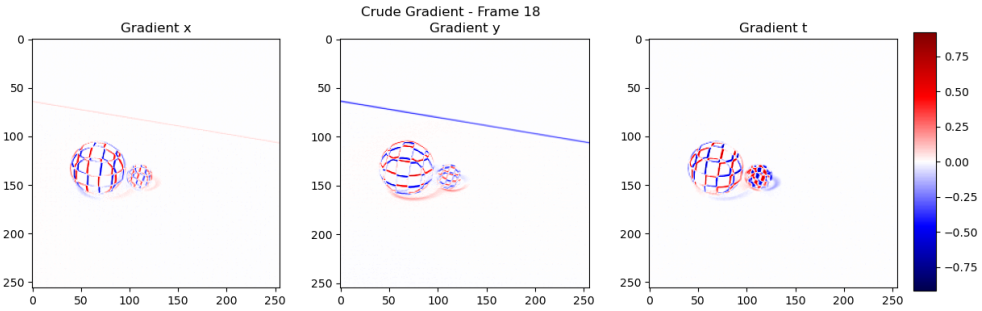


Figure 2: *Selected frame of* $\mathbf{V_x}$, $\mathbf{V_y}$, $\mathbf{V_t}$

Depending on the direction, we notice that the visibility of the table is different respectively to x,y, and t-directions. The table is very visible in the y-direction as we expected. This is because of the big contrast between the background and the table in the kernel. The x-direction displays a smoother transition in comparison. When performing gradient calculations like this, the volume size minimizes. This method reduces the dimensionality of the volume, but implementation is easier.

[1] `https://www.sciencedirect.com/science/article/pii/S0042698912000508?via%3Dihub`
[2] `https://cseweb.ucsd.edu//classes/sp02/cse252/lucaskanade81.pdf`

2.2 Simple Gradient Filters

This second method uses a larger kernel when doing computations on a local neighborhood of the image, which is then repeated on the entire volume. To calculate the gradients we use scipy.ndimage.prewitt to generate our 'Classic' gradient kernel, which defines the processing to be done.

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

***Fig. 3a**: Prewitt kernel*

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

***Fig. 3b**: Sobel kernel*

j

As shown in figure 3a we have a horizontal gradient. If the desired gradient is in other directions, we simply transpose the matrix accordingly. To do the filtering operation, this kernel 3a is simply convolved with the image. When the image is filtered by gradient methods, the noise is amplified. Meaning the gradient generates a mean value in the kernel, and when doing this the entire image becomes more smooth. We obtain gradients for each dimension $\mathbf{V_x}$, $\mathbf{V_y}$, $\mathbf{V_t}$.

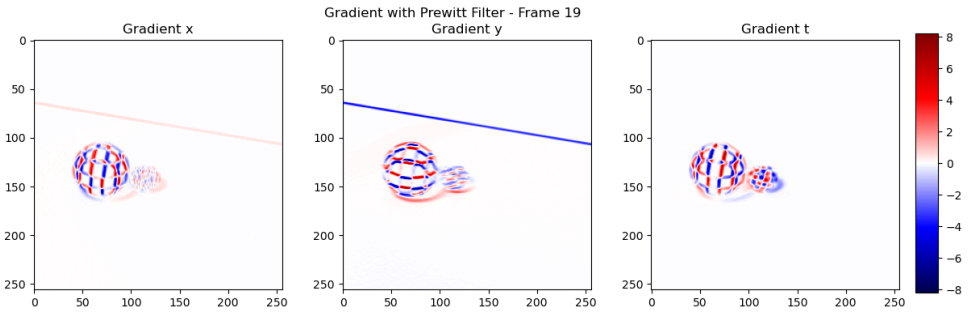


***Figure 4:** Frame of Simple Gradient Filters with Prewitt filter. See appendix 8.2 to see Sobel filter.*

2.3 Gaussian Gradient Filters

Here, we use separability to justify that $G(x,y) = G(x) \cdot G(y)$. Instead of creating 2D and, by extension, 3D kernels, we can simply use 1D kernels consecutively. 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.
It should be noted that the normal Gaussian kernel is a smoothing filter. In order to make it into a gradient filter, we will need the first order derivatives.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \quad (2.1) \qquad G(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}} \quad (2.2) \qquad G(t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{t^2}{2\sigma^2}} \quad (2.3)$$

See appendix 8.3.1 to see how we derived the equations using separability and the product rule. In the end, we obtained the Gaussian gradients as estimated for $\mathbf{V_x}$, $\mathbf{V_y}$, $\mathbf{V_t}$, where we see the gradients boils down to a series of image filter operations with the 1D Gaussian kernel and its derivative.
The smoothing in the Gaussian filter is controlled by the sigma parameter. We try to vary the sigma parameter to see the effect on the gradient estimation. The image becomes more smooth when increasing the sigma parameter. For a visual understanding of how the sigma value changes the Gaussian filter see figure 5a. For a more practical understanding of the effect it has on an actual picture see appendix (10 and 11).

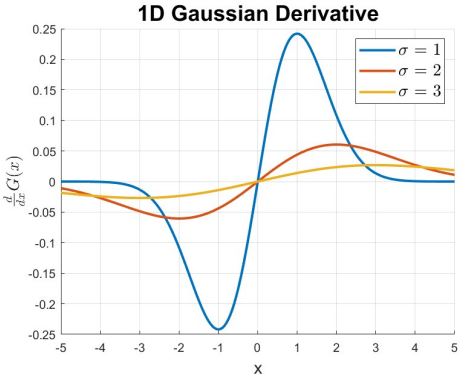


Fig. 5a: The derivative of the 1D Gaussian filter can be seen for different standard deviations.

$$\frac{\partial}{\partial x}G(x,y,t) = \frac{x}{(2\pi)^{3/2}\sigma^5}e^{-\frac{y^2+x^2+t^2}{2\sigma^2}} \quad (2.4)$$

$$\frac{\partial}{\partial y}G(x,y,t) = \frac{y}{(2\pi)^{3/2}\sigma^5}e^{-\frac{y^2+x^2+t^2}{2\sigma^2}} \quad (2.5)$$

$$\frac{\partial}{\partial t}G(x,y,t) = \frac{t}{(2\pi)^{3/2}\sigma^5}e^{-\frac{y^2+x^2+t^2}{2\sigma^2}} \quad (2.6)$$

3 Lucas-Kanade solution

3.1 Theory

As mentioned in section 1, the Lucas-Kanade solution was used to solve the OF problem. The whole sequence of frames can be represented by a three-dimensional matrix $\mathbf{V}$, where x and y is the pixels in each frame and the third dimension t is the increments in time (or frames).
The Lucas-Kanade solutions gives the following over-determined system:

$$\begin{bmatrix} \vdots & \vdots \\ \mathbf{V}_x(\mathbf{p}_i) & \mathbf{V}_y(\mathbf{p}_i) \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \vdots \\ -\mathbf{V}_t(\mathbf{p}_i) \\ \vdots \end{bmatrix} \quad (3.1)$$

Which can easily be solved by the least squares solution, $\mathbf{Ax} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{x} = \mathbf{A}^\top \mathbf{b} \left(\mathbf{A}^\top \mathbf{A}\right)^{-1}$.

3.2 Solving the Toy Problem

To solve the Toy Problem by the Lucas-Kanade method an initial point is chosen: $\mathbf{p} = [x_0, y_0, t_0]^\top$. This is solved by least-square methods in Python and the motion vector-field can be determined from this solution. The process is iterated over all voxel points in the volume, but it is seen that applying the method on the dense volume is computationally expensive.

3.3 Visualizing resulting motion fields

A padding solution is therefore introduced to solve the system, by means of matrix operations, for all voxels in the volume. As the solution to each voxel relies on its neighbours in every direction the edges are discarded. This is also used in the expanded solution and a sketch of how the padding is done can be seen in appendix, figure 12.

After the padding solution is implemented and quivers with length less than 0.1 are removed, we get following vector fields in frame 14, 22 and 58.

3.4 Expanding solution: Live camera feed

Assuming the use of a Prewitt or Sobel kernel, we created a live camera feed. Rather than processing all frames at once, this program works with a 3-voxel tall matrix. As a new frame is loaded, it is added to the top of the matrix, pushing the others down and dropping the "oldest" frame. This matrix is tiny compared to the full volume toy problem, hence the computational requirements

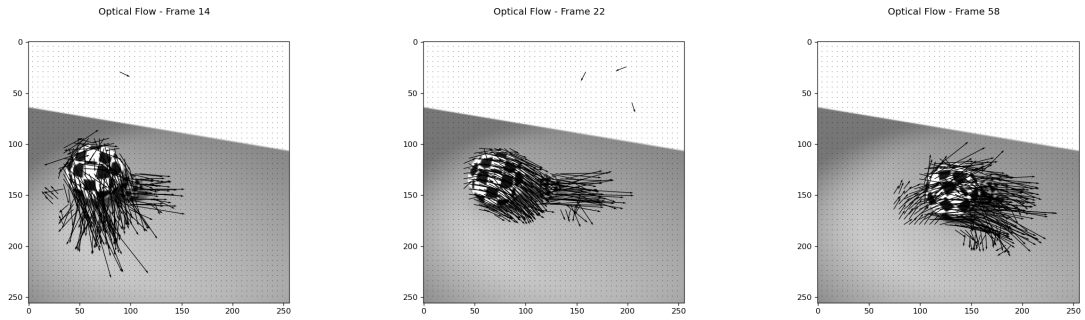


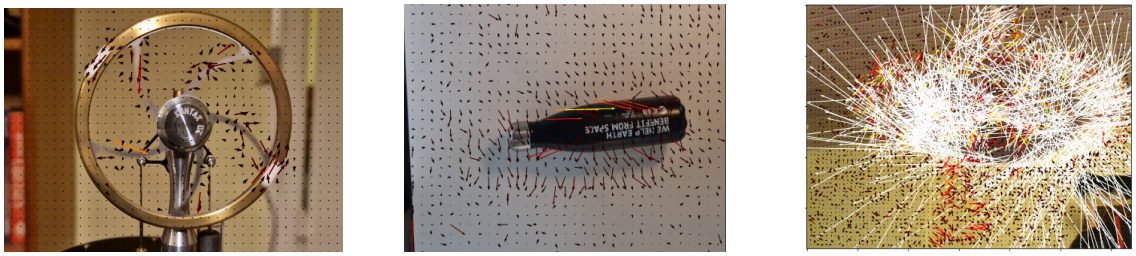
Figure 6: *The same frames displayed in figure 1, but displayed with the vector fields, where insignificant motions are removed.*

drops about 20-fold to an order of 100 ms. Set the frame input to a live webcam feed or similar and you have a movie-like output with real-time motion vectors.

4 Experimental Test of OF

A video which is suited for OF needs to be roughly in line with the assumptions made in the Lucas-Kanande solution. This entails that the objects should be moving slowly - relative to the frame rate at least - on a static background with fixed lighting and additionally there can be a need for the object to be textured.

In fig. 7a the two white markings clearly indicate the direction of motion and the spokes also show motion. Notice that the brass wheel itself is shown as motionless as the surface is smooth and featureless.
Figure 7b shows a clear direction of motion for the bottle as well as the shadow. The top of the bottle suffers from the aperture problem, seen as strong orthogonal motion on the boundary. Slight changes in the background lighting also introduces some noise.
Figure 7c shows a lamp that is turned on and shaking violently. This breaks the basic assumptions made in the Lucas-Kanande model and therefore the overflow model breaks down.



(a) ***(b)*** ***(c)***

Figure 7: *Here are some selected frames from our own videos. (a): A slowly spinning wheel on static background shot at 60FPS. (b): A rolling bottle on a plain background with moving shadows shot at 30FPS. (c): A fast swinging lamp on a background with fast moving shadows shot at ≈10FPS.*

5 OF via Gaussian Filters

In the first part the script iterated over each voxel individually, which is workable, but quite slow. In the latter of part 3 we changed the method to a series of matrix operations over each plane in

the voxel space. This is about 7-10 times faster. But it can be done in other ways too. First, let's understand the $\mathbf{Ax} = \mathbf{x}$ a bit better. Multiply each side by A^T; the LHS becomes:

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix} \dots & \mathbf{V}_x(\mathbf{p}_i) & \dots \\ \dots & \mathbf{V}_y(\mathbf{p}_i) & \dots \end{bmatrix} \cdot \begin{bmatrix} \vdots & \vdots \\ \mathbf{V}_x(\mathbf{p}_i) & \mathbf{V}_y(\mathbf{p}_i) \\ \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \sum_i \mathbf{V}_x(\mathbf{p}_i)^2 & \sum_i \mathbf{V}_x(\mathbf{p}_i)\mathbf{V}_y(\mathbf{p}_i) \\ \sum_i \mathbf{V}_y(\mathbf{p}_i)\mathbf{V}_x(\mathbf{p}_i) & \sum_i \mathbf{V}_y(\mathbf{p}_i)^2 \end{bmatrix} = \begin{bmatrix} s_{xx} & s_{xy} \\ s_{yx} & s_{yy} \end{bmatrix} \tag{5.1}$$

Notice here that $s_{xy} = s_{yx}$. On the RHS this becomes

$$\mathbf{A}^T\mathbf{b} = \begin{bmatrix} \dots & \mathbf{V}_x(\mathbf{p}_i) & \dots \\ \dots & \mathbf{V}_y(\mathbf{p}_i) & \dots \end{bmatrix} \cdot \begin{bmatrix} \vdots \\ -\mathbf{V}_t(\mathbf{p}_i) \\ \vdots \end{bmatrix} = \begin{bmatrix} -\sum_i \mathbf{V}_x(\mathbf{p}_i)\mathbf{V}_t(\mathbf{p}_i) \\ -\sum_i \mathbf{V}_y(\mathbf{p}_i)\mathbf{V}_t(\mathbf{p}_i) \end{bmatrix} = \begin{bmatrix} -s_{xt} \\ -s_{yt} \end{bmatrix} \tag{5.2}$$

From this the final equation can be set up

$$\begin{bmatrix} s_{xx} & s_{xy} \\ s_{yx} & s_{yy} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = - \begin{bmatrix} s_{xt} \\ s_{yt} \end{bmatrix} \tag{5.3}$$

Here, the matrix products in (5.1) and (5.2) is easily calculated as the element-wise product of the respective matrices. The summation could be done with a convolution of an averaging image filter of size N.
On the other hand, a Gaussian filter applied to the whole volume is feasible as well. This would in essentially create a weighted sum, akin to the above equations. See attached video "Rolling Bottle" as an example.

6 Discussion

We compare the different methods analyzed in section 2; the low level gradient, Prewitt filter, Sobel filter, and Gaussian filter. See figure (13) in the appendix to observe the filters in the same time frame. The crude gradient has weak edges compared to the other filters. Prewitt and Sobel filters give edges that are more visible because of the weighted pixel intensities around the edges. In the Gaussian filter, we notice a smoother transition, which appears because the edges are calculated from the Gaussian distribution. Because of the given kernels for weighted pixel intensities, Prewitt and Sobel filters are determined beforehand, and can not adapt to the given image. However, the Gaussian filter has a parameter (σ) that can be adjusted, and therefore one can control the effectiveness of the filter, which is a great ability for OF.
The results from our analysis show good results when we use videos suited for analysing OF, but bad results when lighting is rapidly changed. This accords very well with our expectations since we are assuming through the Lucas-Kanade solution that the lighting is constant.

Another way to try and challenge our analysis which we did not do, is to use videos with fast-rotating elements for example a spinning top.

7 Conclusion

In this project, we have learned that OF is a feasible and usually practical method to determine movement in an image series. However, it works best in optimal conditions and is borderline useless if the lighting conditions or background changes quickly.
The simpler implementations of OF can be quite computationally expensive, but more refined approaches can reduce this by at least an order of magnitude.
In total, this shows that OF can be circumstantially useful, if a suitable method is used. Which method is optimal for any given situation will depend on the validity of the assumptions in said method. Based on our finding, we would recommend Lucas-Kanade solution based on matrix operations for accuracy, and the Gaussian Filter method for speed.

8 Appendix

8.1 Low-level gradient calculation

```
"""
Problem 2.1: Low Level Gradient
"""

# Computing Vx, Vy and Vt
Vy = im_3d[1:, :, :] - im_3d[:-1, :, :]
Vx = im_3d[:, 1:, :] - im_3d[:, :-1, :]
Vt = im_3d[:, :, 1:] - im_3d[:, :, :-1]

plot_3_gradients(Vx, Vy, Vt, title = "Crude Gradient")
```

8.2 Simple Gradient Filters

```
Problem 2.2: Simple Gradient Filters
"""
#%%

# Using the Prewitt method

Vy_prewitt = ndimage.prewitt(im_3d, axis=0)
Vx_prewitt = ndimage.prewitt(im_3d, axis=1)
Vt_prewitt = ndimage.prewitt(im_3d, axis=2)

# Displaying the gradient
plot_3_gradients(Vx_prewitt, Vy_prewitt, Vt_prewitt, title =
"Gradient with Prewitt Filter")

#----------------------------------

# Using the Sobel method
Vy_sobel = ndimage.sobel(im_3d, axis=0)
Vx_sobel = ndimage.sobel(im_3d, axis=1)
Vt_sobel = ndimage.sobel(im_3d, axis=2)

# Displaying the gradient
plot_3_gradients(Vx_sobel, Vy_sobel, Vt_sobel, title =
"Gradient with Sobel Filter")
```

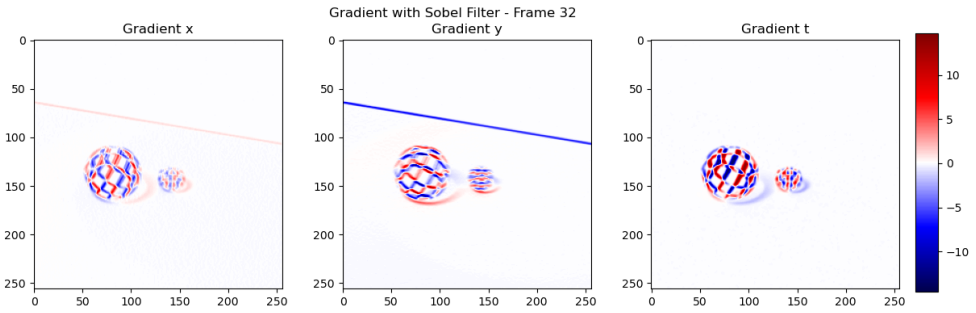


Figure 8: *Frame of Simple gradient filter with Sobel filter*

8.3 Gaussian Gradient Filters

8.3.1 Derivation of Gaussian gradients

Starting in 2D and making the Gaussian derivative kernels by using the separability property, and subsequently using the product rule for differentiation in equation 8.1.

$$\frac{\partial}{\partial x}G(x,y) = G(y)\cdot\frac{\mathrm{d}}{\mathrm{d}x}G(x) = G(y)\cdot\frac{\mathrm{d}}{\mathrm{d}x}G(x) + G(x)\cdot\frac{\mathrm{d}}{\mathrm{d}x}G(y) = G(y)\cdot\frac{\mathrm{d}}{\mathrm{d}x}G(x) \tag{8.1}$$

Inserting values and derives the equation.

$$= \frac{1}{2\pi\sigma^2}e^{-\frac{y^2}{2\sigma^2}}\cdot\frac{\mathrm{d}}{\mathrm{d}x}\left(e^{-\frac{x^2}{2\sigma^2}}\right) = \frac{1}{2\pi\sigma^2}e^{-\frac{y^2}{2\sigma^2}}\cdot\left(-\frac{x}{\sigma^2}e^{-\frac{x^2}{2\sigma^2}}\right) = -\frac{1}{2\pi\sigma^2}\frac{x}{\sigma^2}e^{-\frac{y^2+x^2}{2\sigma^2}} = -\frac{x}{2\pi\sigma^4}e^{-\frac{y^2+x^2}{2\sigma^2}}$$

Extending to 3D

$$G(x,y,t) = (2\pi\sigma^2)^{-\frac{3}{2}}e^{\frac{y^2+x^2+t^2}{2\sigma^2}} \tag{8.2}$$

Following the same arguments as in 2D, starting with differentiation of $G(x)$.

$$\frac{\partial}{\partial x}G(x,y,t) = G(t)G(y)\frac{\mathrm{d}}{\mathrm{d}x}G(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\frac{x}{2\pi\sigma^4}e^{-\frac{y^2+x^2+t^2}{2\sigma^2}} = \frac{x}{(2\pi)^{3/2}\sigma^5}e^{-\frac{y^2+x^2+t^2}{2\sigma^2}} \tag{8.3}$$

Repeating same procedure for G(y) and G(t).

$$\frac{\partial}{\partial y}G(x,y,t) = G(x)G(t)\frac{\mathrm{d}}{\mathrm{d}y}G(y) = \frac{y}{(2\pi)^{3/2}\sigma^5}e^{-\frac{y^2+x^2+t^2}{2\sigma^2}} \tag{8.4}$$

$$\frac{\partial}{\partial t}G(x,y,t) = G(x)G(y)\frac{\mathrm{d}}{\mathrm{d}t}G(t) = \frac{t}{(2\pi)^{3/2}\sigma^5}e^{-\frac{y^2+x^2+t^2}{2\sigma^2}} \tag{8.5}$$

8.3.2 Code for plotting Gaussian gradients

```
    """
Problem 2.3: Gaussian Gradient Filters
"""
# Using the Gaussian Kernel

sigma = 3

Vy_gauss = ndimage.gaussian_filter1d(im_3d, sigma=sigma, order = 1, axis=0)
Vx_gauss = ndimage.gaussian_filter1d(im_3d, sigma=sigma, order = 1, axis=1)
Vt_gauss = ndimage.gaussian_filter1d(im_3d, sigma=sigma, order = 1, axis=2)
```

```
# Displaying the gradient
plot_3_gradients(Vx_gauss, Vy_gauss, Vt_gauss, title =
"Gradient with Gaussian Filter")
```

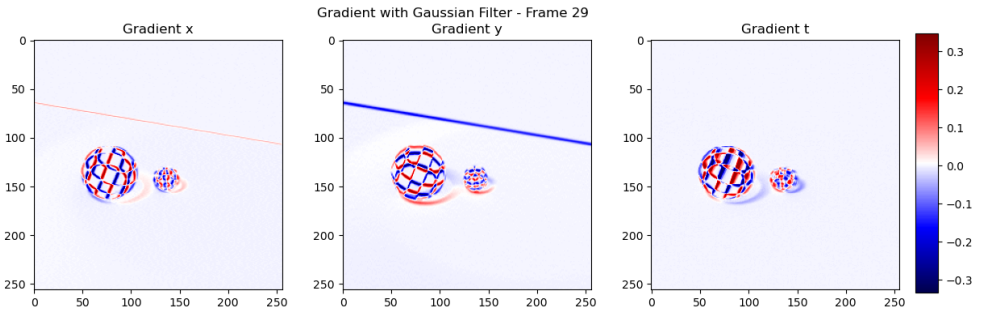


Figure 9: *Frame of Gaussian with sigma = 1*

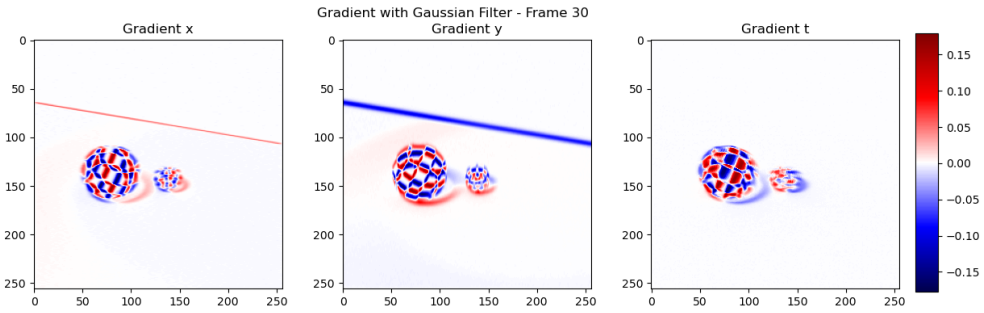


Figure 10: *Frame of Gaussian with sigma = 2*

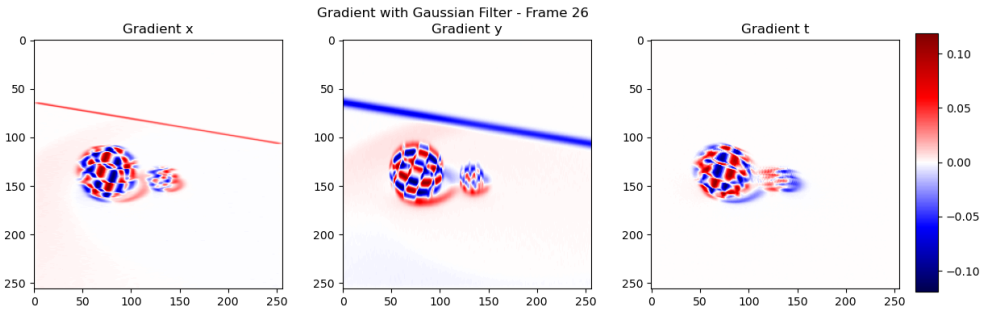


Figure 11: *Frame of Gaussian with sigma = 3*

8.4 Lucas-Kanade solution

Appendix with code for the Lucas-Kanade solution which is implemented through a padding solution. Here 0-pixel values are introduced around the frame/image to make the calculation possible.

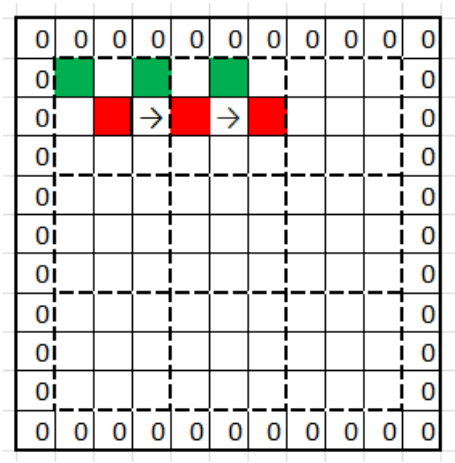


Figure 12: *Simplified image where we have padded around the initial image with zeros. Red indicate the pixel considered.*

Take the gradient matrices $\mathbf{V}_x$, $\mathbf{V}_y$, and $\mathbf{V}_t$ of size m by n. In the case of a system of equations of size N by N this method exploits the relative position of corresponding pixels.
Each proper sub-matrix[3] of size $n-N+1$ by $m-N+1$ will represent a specific entry in each system matrix A. From here the matrices in (5.3) can be obtained more efficiently, and with only a single call to the linear algebra solver, the speed of this algorithm is increased by an order of magnitude.

8.5 Discussion

Figure to compare the different filters.

[3]A proper sub-matrix shall here refer to any sub-matrix that result from removing only edges from the original matrix

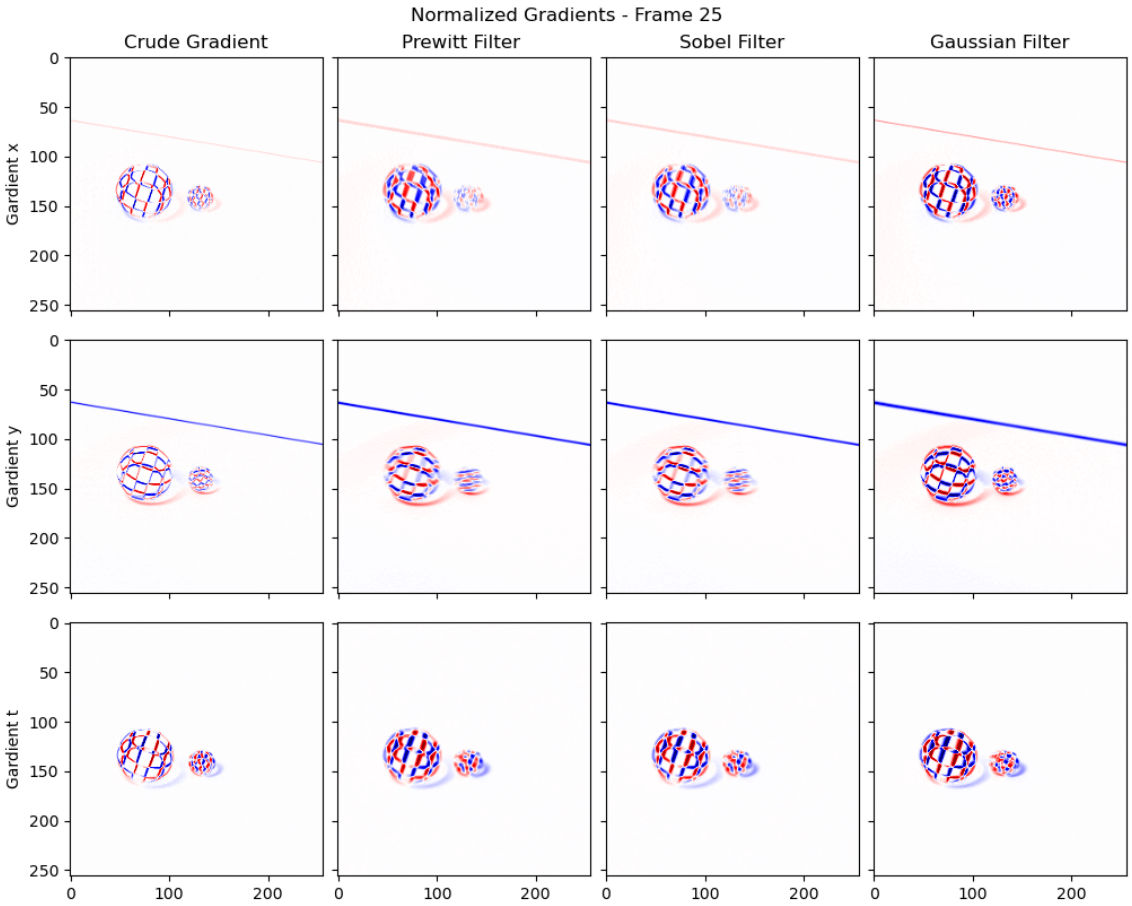


Figure 13: *Frame of all methods*

9 Appendix - Code files

Optical flow python script.

```
import matplotlib.pyplot as plt
import os
import skimage
#from skimage import io, color
from scipy import ndimage, signal
import numpy as np
import time
from tensor_solve import *
import cv2


def plot_3_gradients(Vx, Vy, Vt, cmap = "seismic", FPS = 24, title = "Gradient"):
    ↪

    # Set sensible colormap scale
    vmin = np.min([np.min(Vy),np.min(Vx),np.min(Vt)])
    vmax = np.max([np.max(Vy),np.max(Vx),np.max(Vt)])
```

```
    # Account for diverging array size in Vt
    N_im = np.min([np.shape(Vx)[2],np.shape(Vy)[2],np.shape(Vt)[2]])
    if N_im == 0: return None

    # Set up figure parameters
    fig, [axx,axy,axt] = plt.subplots(1, 3, figsize=(14, 4))
    cb_ax = fig.add_axes([0.91, 0.1, 0.02, 0.8])

    # Initiate the figure
    axx.set_title("Gradient x"); axy.set_title("Gradient y"); axt.set_title("
        ↪ Gradient t")
    imx = axx.imshow(Vx[:,:,0], vmin = vmin, vmax = vmax, cmap = cmap)
    imy = axy.imshow(Vy[:,:,0], vmin = vmin, vmax = vmax, cmap = cmap)
    imt = axt.imshow(Vt[:,:,0], vmin = vmin, vmax = vmax, cmap = cmap)
    fig.colorbar(mappable=imt, cax=cb_ax)

    # Iterate the plot
    for i in range(N_im):
        fig.suptitle(f"{title} - Frame {i+1}")
        imx.set_data(Vx[:,:,i])
        imy.set_data(Vy[:,:,i])
        imt.set_data(Vt[:,:,i])
        plt.pause(1/FPS)
    plt.close(fig)


# ----------------------------------------------------------------

"""
Problem 1.1: Making the video
"""

# Loading all 64 images into a 3D array as grayscale
w = 640
h = 480

print(" Loading Files...", end="\r")
start = time.time()

image_name_list = np.sort(os.listdir('Optical_flow/toyProblem_F22'))
N_im = np.size(image_name_list)
im_3d = np.zeros((h, w, N_im))
cam = cv2.VideoCapture(1)

for i, image_location in enumerate(image_name_list):
    _, image = cam.read()
    im_gray = skimage.color.rgb2gray(image)
    im_3d[:,:, i] = im_gray

# Displaying the image sequense
imm = skimage.io.imshow(im_3d[:,:,0])
# for i in range(N_im):
# imm.set_data(im_3d[:,:,i])
    # plt.title(f"Frame {i+1}")
```

```
    # plt.pause(1/24)

plt.close()

"""
Problem 2.1: Low Level Gradient
"""

print("␣Computing␣gradients...␣", end="\r")

# Computing Vx, Vy and Vt
Vy = im_3d[1:, :, :] - im_3d[:-1, :, :]
Vx = im_3d[:, 1:, :] - im_3d[:, :-1, :]
Vt = im_3d[:, :, 1:] - im_3d[:, :, :-1]

# plot_3_gradients(Vx, Vy, Vt, title = "Crude Gradient")


"""
Problem 2.2: Simple Gradient Filters
"""


# Using the Prewitt method

Vy_prewitt = ndimage.prewitt(im_3d, axis=0)
Vx_prewitt = ndimage.prewitt(im_3d, axis=1)
Vt_prewitt = ndimage.prewitt(im_3d, axis=2)

# Displaying the gradient
# plot_3_gradients(Vx_prewitt, Vy_prewitt, Vt_prewitt, title = "Gradient with
    ↪ Prewitt Filter")

#---------------------------------

# Using the Sobel method
Vy_sobel = ndimage.sobel(im_3d, axis=0)
Vx_sobel = ndimage.sobel(im_3d, axis=1)
Vt_sobel = ndimage.sobel(im_3d, axis=2)

# Displaying the gradient
# plot_3_gradients(Vx_sobel, Vy_sobel, Vt_sobel, title = "Gradient with Sobel
    ↪ Filter")


"""
Problem 2.3: Gaussian Gradient Filters
"""
# Using the Gaussian Kernel

sigma = 2

Vy_gauss = ndimage.gaussian_filter1d(im_3d, sigma=sigma, order = 1, axis=0)
Vx_gauss = ndimage.gaussian_filter1d(im_3d, sigma=sigma, order = 1, axis=1)
```

```
Vt_gauss = ndimage.gaussian_filter1d(im_3d, sigma=sigma, order = 1, axis=2)

# Displaying the gradient
# plot_3_gradients(Vx_gauss, Vy_gauss, Vt_gauss, title = "Gradient with Gaussian
    ↪ Filter")


#Let's plot all the gradients in one plot to compare.
test_frame = 25
vmin = -1; vmax = 1; cmap = "seismic"

imm = skimage.io.imshow(im_3d[:,:,0])
fig, ax = plt.subplots(3, 4, figsize=(10, 8), constrained_layout=True, sharex =
    ↪ True, sharey=True)
#cb_ax = fig.add_axes([0.97, 0.1, 0.01, 0.8])

# Initiate the figure
#crude gradient
km = ax.flat[0].imshow(Vx[:,:,test_frame]/np.max(Vx[:,:,test_frame]), vmin =
    ↪ vmin, vmax = vmax, cmap = cmap)
ax.flat[0].set_title("Crude␣Gradient")
ax.flat[0].set_ylabel("Gardient␣x")
ax.flat[4].imshow(Vy[:,:,test_frame]/np.max(Vy[:,:,test_frame]), vmin = vmin,
    ↪ vmax = vmax, cmap = cmap)
ax.flat[4].set_ylabel("Gardient␣y")
ax.flat[8].imshow(Vt[:,:,test_frame]/np.max(Vt[:,:,test_frame]), vmin = vmin,
    ↪ vmax = vmax, cmap = cmap)
ax.flat[8].set_ylabel("Gardient␣t")

# Prewitt Filter
ax.flat[1].imshow(Vx_prewitt[:,:,test_frame]/np.max(Vx_prewitt[:,:,test_frame]),
    ↪  vmin = vmin, vmax = vmax, cmap = cmap)
ax.flat[1].set_title("Prewitt␣Filter␣")
ax.flat[5].imshow(Vy_prewitt[:,:,test_frame]/np.max(Vy_prewitt[:,:,test_frame]),
    ↪  vmin = vmin, vmax = vmax, cmap = cmap)
ax.flat[9].imshow(Vt_prewitt[:,:,test_frame]/np.max(Vt_prewitt[:,:,test_frame]),
    ↪  vmin = vmin, vmax = vmax, cmap = cmap)

# Sobel Filter
km = ax.flat[2].imshow(Vx_sobel[:,:,test_frame]/np.max(Vx_sobel[:,:,test_frame]),
    ↪  vmin = vmin, vmax = vmax, cmap = cmap)
ax.flat[2].set_title("Sobel␣Filter")
ax.flat[6].imshow(Vy_sobel[:,:,test_frame]/np.max(Vy_sobel[:,:,test_frame]),
    ↪ vmin = vmin, vmax = vmax, cmap = cmap)
ax.flat[10].imshow(Vt_sobel[:,:,test_frame]/np.max(Vt_sobel[:,:,test_frame]),
    ↪ vmin = vmin, vmax = vmax, cmap = cmap)

# Gaussian Filter
km = ax.flat[3].imshow(Vx_gauss[:,:,test_frame]/np.max(Vx_gauss[:,:,test_frame]),
    ↪  vmin = vmin, vmax = vmax, cmap = cmap)
ax.flat[3].set_title("Gaussian␣Filter")
ax.flat[7].imshow(Vy_gauss[:,:,test_frame]/np.max(Vy_gauss[:,:,test_frame]),
    ↪ vmin = vmin, vmax = vmax, cmap = cmap)
ax.flat[11].imshow(Vt_gauss[:,:,test_frame]/np.max(Vt_gauss[:,:,test_frame]),
```

```
    ↪ vmin = vmin, vmax = vmax, cmap = cmap)

#fig.colorbar(mappable=km, cax=cb_ax)
fig.suptitle(f"Normalized Gradients - Frame {test_frame}")
fig.savefig("Compare_gradients.png", dpi = 300, format = "png")


"""
Problem 3.1
"""
# N = 3 # N has to be uneven because of the r definition below
# r = int((N-1)/2)
# x0=100; y0=100; t0 = 30

# # print(np.shape(Vy_prewitt))
# Vy_p = Vy_prewitt[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
# Vx_p = Vx_prewitt[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
# Vt_p = Vt_prewitt[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
# A = np.stack((Vy_p, Vx_p))

# sol = np.linalg.lstsq(A.T, -Vt_p, rcond = None)
# print(sol[0])


"""
Problem 3.2
"""

# pos = np.mgrid[r:256-r, r:256-r]
# x_list = pos[0,:,:].flatten()
# y_list = pos[1,:,:].flatten()

# vector_field = np.zeros(np.shape(pos))
# start = time.time()
# for i in range(np.size(x_list)):
# N = 7 # N has to be uneven because of the r definition below
# r = int((N-1)/2)
# x0=x_list[i]; y0=y_list[i]; t0 = 30

# # print(np.shape(Vy_prewitt))
# Vy_p = Vy_prewitt[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
# Vx_p = Vx_prewitt[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
# Vt_p = Vt_prewitt[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
# A = np.stack((Vy_p, Vx_p))

# sol = np.linalg.lstsq(A.T, -Vt_p, rcond = None)
# vector_field[0, x0-1, y0-1] = sol[0][0]
# vector_field[1, x0-1, y0-1] = sol[0][1]
# print(f"There passed {np.round((time.time()-start), 3)} seconds")

# plt.close()
# plt.figure()
# plt.imshow(im_3d[:,:,30], cmap = 'gray')
# plt.quiver(pos[0,::10,::10], pos[1,::10,::10], vector_field[0,::10,::10],
```

```
    ↪ vector_field[1,::10,::10])
# plt.show()


############ GIF in 3.2
N = 9 # N has to be uneven because of the r definition below
r = int((N-1)/2)
tmax = N_im
pos = np.mgrid[r:h-r, r:w-r, 0:tmax]
# x_list = pos[0].flatten()
# y_list = pos[1].flatten()
# t_list = pos[2].flatten()

vector_field = np.zeros((2, h-2*r, w-2*r, tmax))


print("Computing␣flow...␣␣␣␣␣␣")

for i in range(tmax):
    vector_field[:, :, :, i] = tensor_solve(Vx = Vx_prewitt[:,:,i], Vy =
        ↪ Vy_prewitt[:,:,i], Vt = Vt_prewitt[:,:,i], N = N)[:, r:-r, r:-r]


for i in range(np.size(x_list)):
    if i%1e4 == 0: print(f"␣Operations:␣{np.round(i*1e-6,2)}␣million␣", end="\r")
    x0=x_list[i]; y0=y_list[i]; t0 = t_list[i]

    Vy_p = Vy_sobel[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
    Vx_p = Vx_sobel[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
    Vt_p = Vt_sobel[y0-r:y0+r+1, x0-r:x0+r+1, t0].flatten()
    A = np.stack((Vx_p, Vy_p))

    sol = np.linalg.lstsq(A.T, -Vt_p, rcond = None)
    vector_field[0, x0-r, y0-r, t0] = sol[0][0]
    vector_field[1, x0-r, y0-r, t0] = sol[0][1]

# print(f"\nDone in {time.strftime('%-M minutes and %-S seconds', time.gmtime(
    ↪ time.time()-start))}")



#print(np.shape(pos))

N_a = 7 # This is the distance in pixels between each quiver arrow
average_filter = np.ones([3, 3])/(9)

# Initialize the plot
ax1 = plt.figure(figsize = (6,6))
background = plt.imshow(im_3d[:,:,0], cmap = 'gray')
opt_flow = plt.quiver(pos[0,::N_a,::N_a, 0], pos[1,::N_a,::N_a, 0], vector_field
    ↪ [0,::N_a,::N_a, 0], -vector_field[1,::N_a,::N_a, 0], figure = ax1)
plt.pause(1)

for i in range(tmax):
```

```
    # Lets try to average the vector field
    quiver_field_x = signal.convolve2d(vector_field[0,:,:,i], average_filter,
        ↪ mode = "same")
    quiver_field_y = signal.convolve2d(vector_field[1,:,:,i], average_filter,
        ↪ mode = "same")

    # Lets remove small values
    amplitude_field = quiver_field_x**2 + quiver_field_y**2
    neglect_value = 0.15
    quiver_field_x[amplitude_field <= neglect_value] = 0
    quiver_field_y[amplitude_field <= neglect_value] = 0

    # Plot the result
    ax1.suptitle(f"Optical␣Flow␣-␣Frame␣{i+1}")
    background.set_data(im_3d[:,:,i])
    opt_flow.set_UVC(quiver_field_x[::N_a,::N_a], -quiver_field_y[::N_a,::N_a])
    # Check the sign of quiver_field_y...
    # It seems to be invereted...
    #ax.arrow(10,100,50,50)

    plt.pause(1/10)
    #plt.savefig(f'Optical_flow/toyOpticalFlow/image_flow_{i}.png', dpi = 120)
```

Tensor-solve python script used in the optical flow script to solve the problem through padding.

```
import numpy as np
import time

# def tensor_solve(Vx, Vy, Vt, N = 3):
# """
# Vx, Vy, and Vt must all be the same shape numpy array
# """
# # Assume N = 3

# (n,m) = np.shape(Vx)
# vector_field = np.zeros((2,n,m))

# if (np.shape(Vy) != (n,m)) or (np.shape(Vt) != (n,m)): raise Exception("
    ↪ Gradients Must be the same shape!")
# if N%2 == 0: raise Exception("N must be odd!")

# # Somehow this method only works for square matrices
# # Padding is therefore nessersary
# k = max(n,m)
# Vx = np.pad(Vx,((0,k-n),(0,k-m)), mode="constant")
# Vy = np.pad(Vy,((0,k-n),(0,k-m)), mode="constant")
# Vt = np.pad(Vt,((0,k-n),(0,k-m)), mode="constant")

# # Matrix dimensions
# r = (N-1)//2; d = 2*r
# si = (k-d)**2; sj = N**2

# # Predefinitions
```

```
# A0 = np.zeros((si,sj,2))
# b0 = np.zeros((si,sj,1))

# # Grab all proper submatrices of size (k-d) by (k-d)
# for i in range(sj):
# x = i%N; y = i//N
# u = x-d; v = y-d
# if u == 0: u = None
# if v == 0: v = None

# A0[:,i,0] = Vx[y:v,x:u].T.flatten()
# A0[:,i,1] = Vy[y:v,x:u].T.flatten()
# b0[:,i,0] = Vt[y:v,x:u].T.flatten()


# # All matricies in A must be square. This is done by 3D matrix multiplication
# AT = np.transpose(A0, (0,2,1))
# A = np.matmul(AT, A0)
# b = np.matmul(AT, b0)

# # Make sure trivial zeros does not kill the solver :)
# # 'A' cannot be singular, so this is fixed here
# trivial_zeros = np.argwhere(np.all(A[..., :] == 0, axis=(1,2)))
# A[trivial_zeros] = np.array([[1,1],[0,1]])
# b[trivial_zeros] = np.array([[0],[1]])
# # Magic!
# try:
# sol = np.linalg.solve(A,-b)
# except np.linalg.LinAlgError:
# sol = np.zeros((si,2,1))

# # Reconstruct the vector field to the original size
# vector_field[:,r:-r,r:-r] = np.reshape(sol,(k-d,k-d,2))[:n-d,:m-d].transpose
    ↪ ((2,0,1))

# return vector_field


def fake_solve(Vx, Vy, Vt, N):

    """
    This is a test function
    Returns:
        vector_field: An upper triangle matrix with vector length (1,1)
    """

    r = (N-1)//2
    (n,m) = np.shape(Vx)
    vector_field = np.zeros((2,n,m))
    vector_field[:,r:-r,r:-r] = np.triu(np.ones((2,n-2*r,m-2*r)))

    return vector_field
```

```
#-----------------------------------------------------------------------

def tensor_solve(Vx, Vy, Vt, N = 3):
    """
    Vx, Vy, and Vt must all be the same shape square numpy array
    """

    (n,m) = np.shape(Vx)

    vector_field = np.zeros((2,n,m))

    if (np.shape(Vy) != (n,m)) or (np.shape(Vt) != (n,m)): raise Exception("
        ↪ Gradients Must be the same shape!")
    if N%2 == 0: raise Exception("N must be odd!")

    # No padding nessersary :)
    # Matrix dimensions
    r = (N-1)//2; d = 2*r
    si = (n-d)*(m-d); sj = N**2
    A0 = np.zeros((si,sj,2))
    b0 = np.zeros((si,sj,1))

    # grab all submatrices of size (k-d) by (k-d)
    for i in range(sj):
        x = i%N; y = i//N
        u = x-d; v = y-d
        if u == 0: u = None
        if v == 0: v = None

        A0[:,i,0] = Vx[y:v,x:u].flatten()
        A0[:,i,1] = Vy[y:v,x:u].flatten()
        b0[:,i,0] = Vt[y:v,x:u].flatten()


    # All matricies in A must be square. This is done by 3D matrix multiplication
    AT = np.transpose(A0,(0,2,1))
    A = np.matmul(AT, A0)
    b = np.matmul(AT, b0)

    # Make sure trivial zeros does not kill the solver :)
    # 'A' cannot be singular, so this is fixed here
    singular_matrix = (np.linalg.matrix_rank(A, hermitian=True) != 2)
    A[singular_matrix] = np.array([[1,1],[0,1]])
    b[singular_matrix] = np.array([[0],[0]])

    # Magic!
    sol = np.linalg.solve(A,-b)

    # Reconstruct the vector field to the original size
    vector_field[:,r:-r,r:-r] = np.reshape(sol,(n-d,m-d,2)).transpose((2,0,1))

    return vector_field
```

```
#-----------------------------------------------------------------


if __name__ == "__main__":

    print("Testing␣Linalg␣Solve␣..")

    sample_size = (480,640)
    n_samples = 500
    N = 7

    r = (N-1)//2
    if n_samples >= 10:
        n_percent = n_samples/100
    else:
        n_percent = n_samples

    result1 = np.zeros(n_samples); result2 = np.copy(result1)
    output1 = np.zeros((2,sample_size[0],sample_size[1])); output2 = np.copy(
        ↪ output1)

    Vx = np.random.rand(sample_size[0],sample_size[1])
    Vy = np.random.rand(sample_size[0],sample_size[1])
    Vt = np.random.rand(sample_size[0],sample_size[1])


    for i in range(n_samples):
        if i%n_percent == 0: print(f"Completion:␣{i//n_percent}%␣␣", end = "\r")
        Vx[:,:] = np.random.rand(sample_size[0],sample_size[1])
        Vy[:,:] = np.random.rand(sample_size[0],sample_size[1])
        Vt[:,:] = np.random.rand(sample_size[0],sample_size[1])

        start = time.time()
        output1[:,:,:] = tensor_solve(Vx, Vy, Vt, N = N)
        result1[i] = time.time() - start

    print("\nDone!\n")

    mu1 = np.mean(result1)
    sigma1 = np.std(result1)

    print("Testing␣iterative␣lstsq␣loop␣..")

    pos = np.mgrid[0:sample_size[0],0:sample_size[1]]
    vector_field = np.zeros((2,sample_size[0],sample_size[1]))
    x_list = pos[0].flatten()
    y_list = pos[1].flatten()

    for i in range(n_samples):
        if i%n_percent == 0: print(f"Completion:␣{i//n_percent}%␣␣", end = "\r")
        Vx[:,:] = np.random.rand(sample_size[0],sample_size[1])
        Vy[:,:] = np.random.rand(sample_size[0],sample_size[1])
        Vt[:,:] = np.random.rand(sample_size[0],sample_size[1])
```

```
    start = time.time()

    for j in range(np.size(x_list)):
        # Try to implement np.tensordot
        x0 = x_list[j]; y0 = y_list[j]
        u1 = x0-r; u2 = x0+r+1;
        v1 = y0-r; v2 = y0+r+1
        if u1 == 0: u1 = None
        if v1 == 0: v1 = None

        Vx_p = Vx[v1:v2, u1:u2].flatten()
        Vy_p = Vy[v1:v2, u1:u2].flatten()
        Vt_p = Vt[v1:v2, u1:u2].flatten()

        A = np.stack((Vx_p,Vy_p))

        sol = np.linalg.lstsq(A.T, -Vt_p, rcond=None)
        vector_field[0, x0, y0] = sol[0][0]
        vector_field[1, x0, y0] = sol[0][1]

    output2[:,:,:] = vector_field

    result2[i] = time.time() - start

print("\nDone!\n")
mu2 = np.mean(result2)
sigma2 = np.std(result2)


print("Comparing...")

output1[:,:,:] = tensor_solve(Vx, Vy, Vt, N = N)

# print(np.shape(output1))
# print(np.shape(output2))

# print(output1[:,r:-r,r:-r])
# print(30*"-")
# print(output2[:,r:-r,r:-r])

working_precision = 6
n = 0
nn = 0

for k in zip(output1[:,r:-r,r:-r].flatten(), output2[:,r:-r,r:-r].flatten()):
    if np.round(k[0], working_precision) != np.round(k[1], working_precision):
        ↪
        if k[1]: nn += 1; print(f"Error! {np.round(k[0], working_precision)} 
            ↪ != {np.round(k[1], working_precision)}")
        n += 1

if n == 0:
    print("Results compare OK!\n")
else:
```

```
        print(f"Found␣{n}␣errors␣of␣which␣{nn}␣is␣nontrivial!\n")

    print(f"Tensor␣Solve:␣Average␣is␣{np.round(mu1*1000,1)}ms␣and␣std␣is␣{np.
        ↪ round(sigma1*1000,1)}ms")
    print(f"Lstsq␣Loop:␣Average␣is␣{np.round(mu2*1000,1)}ms␣and␣std␣is␣{np.round(
        ↪ sigma2*1000,1)}ms\n")

    if mu1 > mu2:
        print(f"Lstsq␣Loop␣is␣{np.round((mu1/mu2␣-␣1)*100,1)}␣percent␣faster")
    else:
        print(f"Tensor␣Solve␣is␣{np.round((mu2/mu1␣-␣1)*100,1)}␣percent␣faster")
```

detect-cameras python script used to detect USB-camera to use for a optical flow livefeed.

```
"""
Test the ports and returns a tuple with the available ports and the ones that
    ↪ are working.
Copied from https://stackoverflow.com/questions/57577445/list-available-cameras-
    ↪ opencv-python
 at 18:19 09/02/2023


"""
import cv2


def list_ports():

    is_working = True
    dev_port = 0
    working_ports = []
    available_ports = []

    print(20*"-")

    for i in range(10):
        camera = cv2.VideoCapture(dev_port)
        if not camera.isOpened():
            print("Port␣%s␣is␣not␣working." %dev_port)
        else:
            is_reading, img = camera.read()
            w = camera.get(3)
            h = camera.get(4)
            if is_reading:
                print("Port␣%s␣is␣working␣and␣reads␣images␣(%s␣x␣%s)" %(dev_port,h
                    ↪ ,w))
                working_ports.append(dev_port)
            else:
                print("Port␣%s␣for␣camera␣(␣%s␣x␣%s)␣is␣present␣but␣does␣not␣reads.
                    ↪ " %(dev_port,h,w))
                available_ports.append(dev_port)
        dev_port +=1
```

```
    print(20*"-")
    return available_ports,working_ports


if __name__ == "__main__":

    list_ports()
```

camera python script used to watch livefeed from the detected USB-camera and show the vector field simultaneously.

```
import numpy as np
import skimage
import matplotlib.pyplot as plt
import matplotlib.animation as anim
# import matplotlib.quiver
from scipy import ndimage
import cv2
import time
from tensor_solve import *


# Conditions
N = 7 # Same N as other script...
scale_factor = 4 # Scale factor for optical flow. Lower is better but slower
figsize = (8,4.5)
N_a = 8 # Distance between arrows
sigma = 4

fps = 15
name = "Spinning Wheel"


image_source = "Optical_flow/Videos/Spinny_wheel.mp4"

r = (N-1)//2

# Loading camera
cam = cv2.VideoCapture(image_source)

w = int(cam.get(3))
h = int(cam.get(4))
n_frames = int(cam.get(7))


test_frame = np.random.rand(h,w,3)
frame = np.copy(test_frame[:,:,0]); downscaled_image_old = skimage.transform.
    ↪ downscale_local_mean(frame,(scale_factor,scale_factor));
    ↪ downscaled_image_new = np.copy(downscaled_image_old); downscaled_image =
    ↪ np.copy(downscaled_image_old)
fig, ax = plt.subplots(1, 1, figsize = figsize)
background = plt.imshow(test_frame)
fig.suptitle("Camera")
```

```
N_im = (n_frames-1)*(n_frames > 0) + (n_frames == -1)*100

ret, frame = cam.read()

pos = np.mgrid[0:h:scale_factor,0:w:scale_factor]
vector_field = np.zeros((2,h//scale_factor,w//scale_factor))


opt_flow = plt.quiver(pos[1,::N_a,::N_a], pos[0,::N_a,::N_a], vector_field[0,::
    ↪ N_a,::N_a], vector_field[1,::N_a,::N_a], vector_field[0,::N_a,::N_a], cmap
    ↪  = "hot", scale = 10000) #


movie_writer = anim.writers['ffmpeg']
metadata = dict(title=name, authors = 'Aksel␣Buur␣Christensen,␣Karen␣Witness,␣
    ↪ Morten␣Westermann,␣Viktor␣Isaksen')
movie = movie_writer(fps=fps, metadata=metadata)


with movie.saving(fig, name + ".mp4", n_frames):
    for i in range(N_im):
        start = time.time()
        ret, new_frame = cam.read()
        if not ret:
            print("failed␣to␣grab␣frame")
            break

        img = skimage.color.rgb2gray(new_frame)
        downscaled_image_new = skimage.transform.downscale_local_mean(img,(
            ↪ scale_factor,scale_factor))


        image_stack = np.stack((downscaled_image_old, downscaled_image,
            ↪ downscaled_image_new))
        # Shape: (3,y,x)
        Vy = ndimage.gaussian_filter1d(image_stack, axis=1, order = 1, sigma =
            ↪ sigma, mode="constant", cval=0)[1,:,:]
        Vx = ndimage.gaussian_filter1d(image_stack, axis=2, order = 1, sigma =
            ↪ sigma, mode="constant", cval=0)[1,:,:]
        Vt = ndimage.sobel(image_stack, axis=0)[1,:,:]

        vector_field[:,:,:] = tensor_solve(Vx = Vx, Vy = Vy, Vt = Vt, N = N)

        # Lets remove small values
        amplitude_field = np.sqrt(vector_field[0,::N_a,::N_a]**2 + vector_field
            ↪ [1,::N_a,::N_a]**2)

        # Update Plot
        background.set_data(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
        opt_flow.set_UVC(vector_field[0,::N_a,::N_a], -vector_field[1,::N_a,::N_a
            ↪ ], amplitude_field ) #
        plt.draw()
        movie.grab_frame()
```

```
        downscaled_image_old = np.copy(downscaled_image)
        downscaled_image = np.copy(downscaled_image_new)
        frame = np.copy(new_frame)

        print(f"␣Frame:␣{i},␣Frametime:␣{int(np.ceil(1000*(time.time()␣-␣start))
            ↪ )}ms,␣max␣movement:␣{np.round(np.sqrt(np.max(amplitude_field)),2)},
            ↪ ␣zero-points:␣{(vector_field[:,:,:]␣==␣0).sum()}␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
            ↪ ␣␣␣␣␣␣␣␣", end="\r")

print("\nReleasing␣Camera")
cam.release()
```

gauss_camera python script used to watch livefeed from the detected USB-camera and show the vector field through a gaussian filter simultaneosly.

```
import numpy as np
import skimage
import matplotlib.pyplot as plt
import matplotlib.animation as anim
from scipy import ndimage
import cv2
import time
from tensor_solve import *


# Conditions
scale_factor = 6 # Scale factor for optical flow. Lower is better but slower
figsize = (8,4.5)
N_a = 8 # Distance between arrows
sigma_1 = 2
sigma_2 = 2
sigma_3 = 1

fps = 15
name = "Rolling␣Bottle"

image_source = "Optical_flow/Videos/Good_test_video.mp4"

start = time.time()

# Loading camera
cam = cv2.VideoCapture(image_source)

w = int(cam.get(3))
h = int(cam.get(4))
n_frames = int(cam.get(7))

frame_stack = np.zeros((h,w,3,n_frames), dtype = np.int16)
image_stack_downscaled = np.zeros((h//scale_factor,w//scale_factor,n_frames),
    ↪ dtype=np.float32)
vector_field = np.zeros((h//scale_factor,w//scale_factor,n_frames,2),dtype=np.
    ↪ float32)

for i in range(n_frames):
```

```
    ret, frame = cam.read()
    if not ret:
        print("Failed to grab frame!!")
        quit()
    frame_stack[:,:,:,i] = np.int16(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
    image_stack_downscaled[:,:,i] = skimage.transform.downscale_local_mean(
        ↪ skimage.color.rgb2gray(frame),(scale_factor,scale_factor))

cam.release()

Vx = ndimage.gaussian_filter1d(image_stack_downscaled, sigma = sigma_1, order =
    ↪ 1, axis = 0, mode = 'nearest')
Vy = ndimage.gaussian_filter1d(image_stack_downscaled, sigma = sigma_1, order =
    ↪ 1, axis = 1, mode = 'nearest')
Vt = ndimage.gaussian_filter1d(image_stack_downscaled, sigma = sigma_1, order =
    ↪ 1, axis = 2, mode = 'nearest')

sxx = ndimage.gaussian_filter(Vx*Vx, sigma = sigma_2, mode = 'nearest')
sxy = ndimage.gaussian_filter(Vx*Vy, sigma = sigma_2, mode = 'nearest')
syy = ndimage.gaussian_filter(Vy*Vy, sigma = sigma_2, mode = 'nearest')

sxt = ndimage.gaussian_filter(Vx*Vt, sigma = sigma_3, mode = 'nearest')
syt = ndimage.gaussian_filter(Vy*Vt, sigma = sigma_3, mode = 'nearest')

A = np.zeros((np.size(sxx), 2, 2))
b = -np.zeros((np.size(sxx), 2))

A[:,0,0] = sxx.flatten()
A[:,0,1] = sxy.flatten()
A[:,1,0] = sxy.flatten()
A[:,1,1] = syy.flatten()

b[:,0] = -sxt.flatten()
b[:,1] = -syt.flatten()

sol = np.linalg.solve(A,b)

vector_field = sol.reshape((h//scale_factor,w//scale_factor,n_frames,2))

fig, ax = plt.subplots(1, 1, figsize = figsize)
background = plt.imshow(frame_stack[:,:,:,0])
fig.suptitle("Camera")

movie_writer = anim.writers['ffmpeg']
metadata = dict(title=name, author = 'Aksel Buur Christensen, Karen Witness,
    ↪ Morten Westermann, Viktor Isaksen')
movie = movie_writer(fps=fps, metadata=metadata)

amplitude_field = vector_field[::N_a,::N_a,:,0]**2 + vector_field[::N_a,::N_a
    ↪ ,:,1]**2

pos = np.mgrid[0:h:scale_factor,0:w:scale_factor]
opt_flow = plt.quiver(pos[1,::N_a,::N_a], pos[0,::N_a,::N_a], vector_field[::N_a
    ↪ ,::N_a,0,0], vector_field[::N_a,::N_a,0,1], amplitude_field[:,:,0])
```

```
print(f"\nDone in {time.strftime('%-M minutes and %-S seconds', time.gmtime(time.
    ↪ time()-start))}")

with movie.saving(fig, name + ".mp4", n_frames):
    for i in range(n_frames):
        #Update Plot
        background.set_data(frame_stack[:,:,:,i])
        opt_flow.set_UVC(vector_field[::N_a,::N_a,i,0], vector_field[::N_a,::N_a,
            ↪ i,1], amplitude_field[:,:,i])
        if i == 200: print(np.max(amplitude_field[:,:,200]))
        # write frame to video
        plt.pause(0.1)
        movie.grab_frame()

print("\nReleasing Camera")
```