Department of Computer Science
Technical University of Cluj-Napoca

**Image Processing**
*Laboratory activity*

# Movement detection from a gray-scale images sequence by computing the optical flow

Name: Hudisteanu Mihai
Name: Aierizer Samuel
Group: 30232

Teaching Assistant: Delia Alexandrina Mitrea
delia.mitrea@didatec.onmicrosoft.com

# Contents

# Chapter 1

# Introduction

## 1.1 The context

As the technology advances the demand for automating repetitive tasks risese more and more. In this paper we will explore the methods of automating the detection of movement on a series of consecutive images oar videos.

These types of algorithms are especially useful in monitoring and security systems, beacause they eliminate the need for another sensor that can communicate with the software and deciding when it should keep the data for the next few minutes.

## 1.2 The issues that should be solved

The issue at hand sounds like a simple one, we have a succession of very similar frames that show a number of objects moving through a background. We want to isolate those objects and determine their velocity in order to predict where they will be in the next frames. Those objects can later be identified or the result of the algorithm can be used to trigger a response in other software or hardware.

## 1.3 Proposed objectives

Our objectives for this paper is to explore a number of different algorithms that can perform movement detection on a succession of images or a continues stream of video, in order to compare them and decide wheather they are suitable for certain types of applications.

An algorithm that processes a 1920x1080 in a second is definitely not suitable for data collection from a continuous stream but may very well be perfect for creating a very detailed map with the differences between to very alike images.

# Chapter 2

# Theoretical background

## 2.1 Bibliographical study

## 2.2 Methods that can be applied

First we will have to define our terms and first of all we need to define the term optical flow.

Optical flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer and a scene.

Optical flow can also be defined as the distribution of apparent velocities of movement of brightness pattern in an image.

Optical flow works on several assumptions:

1) The pixel intensities of an object do not change between consecutive frames.

2) Neighbouring pixels have similar motion.

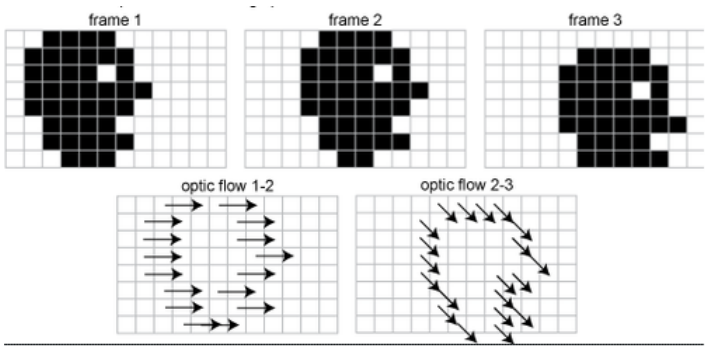


Figure 2.1: OpticalFlow

Pixel Correspondence Problem From this picture, it's easy to figure out the velocity vector (u,v). But when we look at two real images, we'd first need to solve what's called the pixel correspondence problem. That is, we need to know which pixels in image 2 correspond to which pixels in image 1. To solve this problem we make two assumptions. Assumption 1: the motion is small: this means we can look in the vicinity of where the pixel was to try to determine where it now is. Assumption 2: the appearance doesn't change from t to t+1: this assumption is best expressed mathematically.

There are two main type of optical flow : dense optical flow and sparse optical flow.

Sparse optical flow type algorithms like Lucas-Kanade algorithm only track a number of pixels in the image. These types of algorithms need a little preprocessing of the data before we can apply the algorithm itself in order to determine what pixels would be suitable for tracking.

A sparse optical flow algorithm should first of all recieve (usually as a parameter) a number of "interesting"/relevant points in an image. depending on the implementation they can be

corner pixels, very bright pixels or pixels that contrast greatly with they neighbours. They can be extracted with a multitude of algorithms such as Shi-Tomashi and Harris. Next our algoritm needs to track those points in 2 images that should be slightly different, to be more exact 2 consecutive frames of a video clip and it should output a velocity vector for each of the pixels we give as parameters.

Figure 2.2: Car tracking sparse optical flow

Dense optical flow on the other hand doesn't need the vector of relevant pixels in the first place because it will consider all the pixels in the image. This greatly increases the accuracy of the algorithm, with the downside that it requires great computational power. Those types of algorithms are better served by a GPU, using a very high number of relatively simple processes for each pixel that can be executed efficiently by the GPU cores. Without GPU acceleration these algorithms don't have a lot of useful applications especially on larger images or on videos with a high framerate.

Figure 2.3: Tom And Jerry

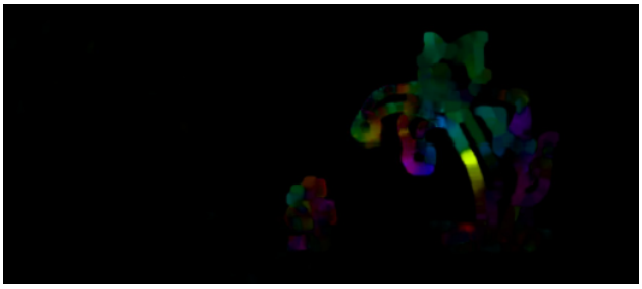

Figure 2.4: TomAndJerry dense optical flow

## 2.3 Possible solutions

Let's assume that we have a gray-scale image – the matrix with pixel intensity. We define the function $I(x, y, t)$ where $x, y$ - pixel coordinates and $t$ - frame number. The $I(x, y, t)$ function defines the exact pixel intensity at frame $t$.

To start with, we assume that the object displacement doesn't change the pixels intensity that belongs to the exact object, it means that $I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$. In our case $\Delta t = 1$. The major concern is to find the motion vector $(\Delta x, y + \Delta y)$.

Using Taylor series expansion we can write $I(x, y, t) - I(x + \Delta x, y + \Delta y, t + \Delta t) = 0$ as $I'_x u + i'_y v = -I'_t$ where $u = \frac{dx}{dt}, v = \frac{dy}{dt}$, and $I'_x, I'_y$ are image gradients. It is important that here we assume that parts of higher-order Taylor series are negligible, so this is a function approximation using only first-order Taylor's expansion. The pixel motion difference between two frames $I_1$ and $I_2$ can be written as $I_1 - I_2 \approx I'_x u + I'_y v + I'_t$. Now, we have two variables $u, v$ and only one equation, so we can't solve the equation right now, but we can use some tricks which will be disclosed in the following algorithms. [2]

For our application we can take a video frame by frame, convert each image to gray-scale if it isn't yet. then we can parse the entire video while we apply different algorithms to the images for determining the optical flow. Then the algorithm result can be displayed in a visible form so that we can see the results.

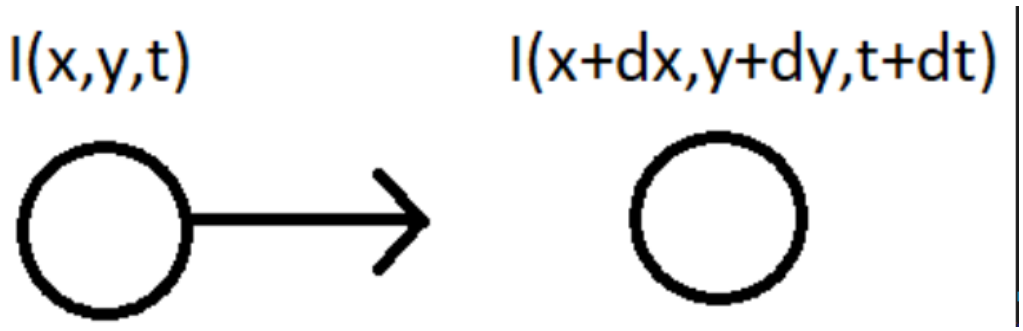


Figure 2.5: Optical Flow

# Chapter 3

# Design and implementation

For the implementation of our project we have chosen to use python, as the OpenCV library is accessible for python and this made the development faster and easier. We have implemented our own method of computing the optical flow. Our implementation is based on the basic method of calculating optical flow. Then we used the already existing algorithms, which can be found in the OpenCV library.

## 3.1 Description of the chosen solution

**Our algorithm**: We take the video frame by frame and convert each frame to gray-scale. We initialize output matrices to store the resulting optical flow and track the changes in the $x, y, t$ directions (or in other words $\Delta x, \Delta y, \Delta t$). We parse the image length and width wise and we make the initial calculations needed. Then we traverse all pixels of the image. For each pixel we create a 15x15 border and we compute the change of direction for the given pixel range. After making all the calculations we use a predefined function to draw arrow lines based on the computations and then we display the result frame by frame.

Our algorithm doesn't make use of the internal GPU. As a result the computations are quite expensive and the resulted image frames are not loaded instantaneously. We have also found OpenCV functions which implement optical flow algorithms and can be used if given the right parameters. We have chosen to make our implementation and to also use the methods found in the library to better understand and exemplify optical flow.

## 3.2 Description of the implementation flow

We have a main function in which we we define the options that we have. The user has to choose which algorithm he want's to use, and he has to state the video path. We define the possible choices and we take the user input as command line arguments. This is a useful decision because if we were to call our implementation from another function or application we could dynamically state the inputs and we are not reliant on manual user input from a file or the command line.

We parse the user input based on the algorithm chosen and then we call the correct method for the optical flow calculation.

## 3.3 Description of the functionalities

We have the following options to choose from:

- Our own optical flow algorithm.
- Lucaskanade sparse optical flow.
- Lucaskanade dense optical flow.
- Farneback dense optical flow algorithm.
- RLOF dense optical flow algorithm.

The syntax for running these is the following:

```
'py project.py --algorithm vector --video_path Videos/banana.mp4'
'py project.py --algorithm lucaskanade --video_path Videos/people.mp4'
'py project.py --algorithm lucaskanade_dense --video_path Videos/people.mp4'
'py project.py --algorithm farneback --video_path Videos/people.mp4'
'py project.py --algorithm rlof --video_path Videos/people.mp4'
```

## 3.4 Description of the implemented algorithms

First of all we implemented the Lucas-Kanade method. There is a controversy regarding this algorithm of weather it is a sparse or a dense optical flow algorithm. As far as the scope of this paper goes we implemented two versions of this algorithm a dense and a sparse one.

In computer vision, the Lucas–Kanade method is a widely used differential method for optical flow estimation developed by Bruce D. Lucas and Takeo Kanade. It assumes that the flow is essentially constant in a local neighbourhood of the pixel under consideration, and solves the basic optical flow equations for all the pixels in that neighbourhood, by the least squares criterion.

By combining information from several nearby pixels, the Lucas–Kanade method can often resolve the inherent ambiguity of the optical flow equation. It is also less sensitive to image noise than point-wise methods, making it the perfect candidate for real time tracking.

To run our project with the sparse method the value for the algorithm argument must be "lucaskanade". For the dense version the argument is "lucaskanade_dense". You have the commands to run the application on any video here :

py project.py –algorithm lucaskanade –video_path Videos/people.mp4

py project.py –algorithm lucaskanade_dense –video_path Videos/people.mp4

Gunnar Farneback Optical Flow

In dense optical flow, we look at all of the points(unlike Lucas Kanade which works only on corner points detected by Shi-Tomasi Algorithm) and detect the pixel intensity changes between the two frames, resulting in an image with highlighted pixels, after converting to hsv format for clear visibility.

It computes the magnitude and direction of optical flow from an array of the flow vectors, i.e., (dx/dt, dy/dt). Later it visualizes the angle (direction) of flow by hue and the distance (magnitude) of flow by value of HSV color representation.For visibility to be optimal, strength of HSV is set to 255. OpenCV provides a function cv2.calcOpticalFlowFarneback to look into dense optical flow. Command to run with Gunnar Farneback :

py project.py –algorithm farneback –video_path Videos/people.mp4

Robust Local Optical Flow The Robust Local Optical Flow (RLOF) is a sparse optical flow and feature tracking method. It is on of the most advanced methods of sparse optical flow and it is already implemented. We included it into the paper in order to have a baseline in the coparison between what we implemented and what is already implemented and used in the industry.

Command to run with RLOF :

py project.py –algorithm rlof –video_path Videos/people.mp4

VectorFlow Vector flow is a line by line implementation of a dense optical flow at pixel level running soley on the CPU. This serves as an example to future readers that want to implement their own version. It calculates a velocity vector for every pixel in the image and outputs it as a dynamic vector map in another window.

Command to run with VectorFlow :

py project.py –algorithm vector –video_path Videos/banana.mp4

The algorithm itself requires a lot suplimentary data structures that are used for the calculus and that are highly dependent on the resolution of the image. For this reason we do not recommand running this algorithm on anything larger than a 500x500 video with more than 15FPS or a 200x200 at 30FPS.

# Chapter 4

# Experimental results

As results we would like to share a couple of pictures with the code running. We have achieved what we wanted to create and the result will show that the implementation, works. We were able to test and compare the solutions. We found that The Lucas-Kanade sparse optical flow was the quickest and was able to handle 4K images in real time. The dense optical flow algorithms were slower. The slowest was our own implementation because we didn't write it to use hardware acceleration with the use of the GPU.

## 4.1 Practical results

**Our algorithm**:

Figure 4.1: Our implementation

**Lucaskanade sparse**:

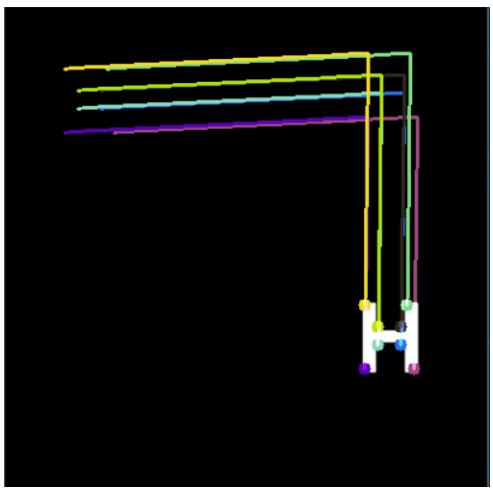

Figure 4.2: Lucaskanade sparse optical flow

**Lucaskanade dense**:

Figure 4.3: Lucaskanade dense optical flow

**Farneback dense**:

Figure 4.4: Farneback dense optical flow

**RLOF dense**:

Figure 4.5: RLOF dense optical flow

# Chapter 5

# Conclusions

## 5.1 Degree of accomplishing the objectives

We have accomplished everything that we intened, that is to explore as many algorithms of optical flow as possible in order to create a context for future projects, either personal or at he university.

## 5.2 Personal contributions

The both of us used Visual Studio Code a a programming enviroment for this project and we worked at the same time on the same files using the Live Share extention for VsCode.

## 5.3 Observations about the obtained results

We would like to explore a more in depth model of the dense optical flow and to try to make it as efficient as possible using dedicated hardware, to establish if it can be reliably used in as a real time motion detecting solution.

If the dense optical flow is too much for the hardware that we have at hand, sparse optical flow remains a good solution especially RLOF and for a version that works reliably on almost ant hardware, any resolution, almost any frame-rate and in real time Lucas-Kanade remains the way to go.

## 5.4 Future development directions

For the reader that would like to continue our work, we propose optimisation and migrating part of the code in GSLS (beeing more user firendly than Vulcan). Writing a process that does all the calculus for a single pixel and using the CPU just to manage what data is recieved and shared between these processes (using uniforms) would be a great advantage in making this program more efficient, possibly so much so that algorithms that we considered too resources intensive to run on real time data would be reliably used for the task.

# Bibliography

[1] Movement Direction Estimation on Video using Optical Flow Analysis on Multiple Frames, International Journal of Advanced Computer Science and Applications, Vol. 9, No. 6, 2018

https://thesai.org/Downloads/Volume9No6/Paper_25-Movement_Direction_Estimation_on_Video.pdf

[2] Optical Flow in OpenCV, Maxim Kuklin, JANUARY 4, 2021

https://learnopencv.com/optical-flow-in-opencv/

[3] Introduction to Motion Estimation with Optical Flow, Chuan-en Lin, 2019

https://nanonets.com/blog/optical-flow/

[4] Accelerate OpenCV: Optical Flow Algorithms with NVIDIA Turing GPUs, Aruna Medhekar, Vishal Chiluka and Abhijit Patait, Dec 05, 2019

https://developer.nvidia.com/blog/opencv-optical-flow-algorithms-with-nvidia-turing-gpus/

[5] Farneback Optical flow Opencv 3.1 Old town square Prague, vladakuc, 25 Feb 2016

https://www.youtube.com/watch?v=Tqf2EzG0djQt=8s

# Appendix A

# Our original code

```
def vector_flow(video_path):
    cap = cv2.VideoCapture(video_path)
    # Take first frame and find corners in it
    ret, old_frame = cap.read()
    #old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
    image = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
    mask = np.zeros_like(old_frame)
    while True:
        maxMag = 0
        ret, frame = cap.read()
        if not ret:
            break
        #frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        image2 = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
           for i in range(0,image.shape[0],1):
            for j in range(0,image.shape[1],1):
                if(j>0 and j<image.shape[1]-1):
                    ix[i][j] = int(image[i][j-1]) - int(image[i][j+1])
                if(i>0 and i<image.shape[0]-1):
                    iy[i][j] = int(image[i-1][j]) - int(image[i+1][j])
                ixx[i][j] = ix[i][j]*ix[i][j]
                iyy[i][j] = iy[i][j]*iy[i][j]
                ixy[i][j] = ix[i][j]*iy[i][j]
                #magnitude of change for one pixel
                it[i][j] = int(image2[i][j]) - int(image[i][j])
                ixt[i][j] = ix[i][j] * it[i][j]
                iyt[i][j] = iy[i][j] * it[i][j]
        #print("iamge size :", image.shape[0], image.shape[1])
        pos = 0
        # traversal of the image pixel by pixel
        for i in range(0, image.shape[0], 5):
            for j in range(0, image.shape[1], 5):
                ixxsum = 0
                iyysum = 0
                ixysum = 0
                ixtsum = 0
                iytsum = 0
                # for each pixel create a 15x15 border
                # and add all of them into a single variable
                for l in range(-6,7):
                    for m in range(-6,7):
                        #clamp in image
                        if((i+l>0 and i+l<image.shape[0])
                        and (j+m>0 and j+m<image.shape[1])):
                            ixxsum += ixx[i+l][j+m]
```

```
                            iyysum += iyy[i+l][j+m]
                            ixysum += ixy[i+l][j+m]
                            ixtsum += ixt[i+l][j+m]
                            iytsum += iyt[i+l][j+m]
                    pos += 1
                    if ixxsum*iyysum == ixysum**2:
                        u[i][j] = 0
                        v[i][j] = 0
                        theta[i][j] = 0
                    else:
                        u[i][j] = ((-iyysum*ixtsum) + ixysum*iytsum)
                            /((ixxsum*iyysum)-(ixysum*ixysum))
                        v[i][j] = ((ixysum*ixtsum) - (ixxsum*iytsum))
                            /((ixxsum*iyysum)-(ixysum*ixysum))
                        theta[i][j] = math.atan2(v[i][j], u[i][j])
                    #distanta intre u[i][j] si v[i][j]
                    mag[i][j] = math.sqrt((u[i][j]**2) + (v[i][j]**2))
                    if (mag[i][j] > maxMag):
                        maxMag = mag[i][j]
                    if (maxMag == 0):
                        maxMag = 1
        #print("End of giant for")
        #Makes steps 5 by 5 pixels to calculate movement
        for i in range(0,image.shape[0],5):
            for j in range(0,image.shape[1],5):
                scaleflow = 15 * (mag[i][j] / maxMag)
                cv2.arrowedLine(arrows,(j,i),
                ((int(((scaleflow*u[i][j]) + j))),
                (int((i+(scaleflow*v[i][j])))),(0,255,0))
        cv2.imshow("arrows",arrows/255)
        #img = cv2.add((arrows/255), mask)
        #cv2.imshow("frame", img)
        k = cv2.waitKey(25) & 0xFF
        if k == 27:
            break
        if k == ord("c"):
            mask = np.zeros_like(old_frame)
        # Now update the previous frame and previous points
        image = image2.copy()
```

Listing A.1: Our code