Optical Flow based Object Tracking

Ce Xu and Valentin Ott

December 2021

1 Introduction

In this work the aim was to design and make an implementation of a technique to track moving objects based on optical flow techniques. Therefore, basic optical flow algorithms were researched and implemented as well as an object detection methodology. Furthermore, different approaches to compute optical flow were tested to compare their outcome.

The different techniques and their results are described in this report. Examples in form of videos were used for better understanding and to show the differences between the diverse algorithms. Because there are many different methodologies nowadays, only a few selected algorithms are considered in this work.

2 Optical flow

Optical flow is defined as the apparent motion of individual pixels on the image plane. It is usually measured between consecutive frames t and $t + dt$ and the displacement of the pixel in the image is expressed as (dx, dy).

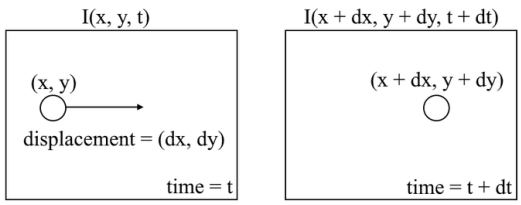


Figure 1: Optical flow problem[2]

The basic idea to calculate optical flow according to Peter O'Donovan [3] is that we assume the pixel intensities of an object are constant between consecutive frames, even though the position of the object may change:

$$I(x, y, t) \approx I(x + dx, y + dy, t + dt)$$

Using this assumption we can take the Taylor series approximation and remove terms of higher order since we are only taking a small neighborhood into account:

$$I(x+dx, y+dy, t+dt) \quad = \quad I(x,y,t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt + \dots$$

$$\Rightarrow \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt \approx 0$$

Now, if we differenciate over the time variable, we can obtain the following approximation:

$$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} \approx 0$$

Here we can rename the $\frac{dx}{dt}$ and $\frac{dy}{dt}$ terms as velocities u and v respectively which are components of the optical flow vector $\boldsymbol{v_{OF}} = (u, v)$. The derivative of the image brightness in x and y direction can be written as I_x and I_y which leads to the optical flow equation:

$$I_x \cdot u + I_y \cdot v + I_t = 0$$

Hence, we have the optical flow equation which is used as starting point for most optical flow algorithms. By solving this equation we receive the optical flow vectors. Thereafter, tracking methods can be applied to compute the movements of objects.

In this work two different optical flow approaches are described where each one belongs to a different group of algorithms. The first group calculates the optical flow vectors for the pixels in an image considering the entire image.They are called global methods. The second group uses only a small neighborhood around each pixel for its calculation of the optical flow vector. The descriptions of these methods in the following section are based on the explications of O'Donovan [3]. Well known algorithms for global techniques are

(a) Optical flow using the Horn-Schunck method

(b) Optical flow using the Gunnar Farnebäck method

(c) Optical flow using the Lucas-Kanade method

Figure 2: Comparison of different Optical Flow methods

for example the Horn and Schunck method (HS) or the Gunnar Farnebäck method (GF). A popular representative for local methods is the computation presented by Lucas and Kanade (LK). In Figure 2 the three mentioned techniques for computing optical flow are applied on a video of a traffic surveillance camera. A clearly difference in between the methods can be seen, which will be looked at in the following sections of this work.

2.1 Horn and Schunck

One of the classical global methods is the Horn and Schunck methodology. It is called a global method, because for the calculations of the optical flow vectors of each pixel it considers the entire image.

Their method uses the optical flow constraint equation but in addition also a smoothness constraint. They both are combined in a system of equations to minimize the error of the optical flow vectors, where the error consists of the difference of the optical vector compared to its neighbors and the error of the optical flow constraint equation. This is described in the following equation:

$$E = \iint (I_x u + I_y v + I_t)^2 dxdy + \alpha \iint \left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 \right) dxdy$$

Using some mathematical transformations Horn and Schunck found a large system with two equations for each pixel. To solve this large system an iterative method is used:

$$u^{n+1} = \bar{u}^n - \frac{I_x[I_x\bar{u}^n + I_y\bar{v}^n + I_t]}{\alpha^2 + I_x^2 + Iy^2}$$

$$v^{n+1} = \bar{v}^n - \frac{I_y[I_x\bar{u}^n + I_y\bar{v}^n + I_t]}{\alpha^2 + I_x^2 + Iy^2}$$

In this system $\bar{u}$ and $\bar{v}$ are the averages of that component of the optical flow vector calculated in a small neighborhood around each pixel. Here, α is a smoothing factor.

2.2 Lucas and Kanade

In this work we consider the Lucas-Kanade algorithm. It works within a local field of pixels and for small motions. The flow is assumed to be constant, but only in the considered area Ω. In this local neighborhood the optical flow equation is solved for every pixel with a weighted least square method to obtain the optical flow vector. For big motions there are existing methods to describe them as small motions, so the algorithm can be used.

To solve the optical flow constraint equation for u and v, this equation is minimized:

$$\sum_{p\in\Omega} = W(p)^2 \left[I_x(p)u + I_y(p)v + I_t(p)\right]^2$$

This equation is applied to each pixel p of the considered area. W hereby describes the weights associated to the neighbor of pixel p. The solution of this minimization problem is as followed:

$$\begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum W^2 I_x I_t \\ \sum W^2 I_y I_t \end{bmatrix}$$

The method from Lucas and Kanade is less sensitive to noise and according to O'Donovan [3] performing consistently and robustly. In comparison to the iterative Horn and Schunck method it gains good estimation of the optical flow vector without relying on the entire image which leads to faster computation time.

3 Implementation

In this section we present the steps in our algorithm to obtain the optical flow based object tracking. The implementation was done in *Matlab*.

3.1 Image pre-processing

The first thing we have to do is to pre-process the image. To do so, we need to obtain the frame, change it to gray scale, pass a median filter to clear the noise and a Gaussian filter to smooth the image.

We have employed the Matlab functions *rgb2gray*, *medfilt2* and *imgaussfilt* due to the time performance those functions bring us compared to self implemented ones. As we will see, in a real-time detector, we need to choose the most efficient functions we can get.

3.2 Optical Flow

To obtain the optical flow vectors we used different methods. The one we computed our self in *Matlab* was the Horn and Schunck technique. The code therefor is seen in the following section.

```
function [OpticFlow] = calcFlowHS(currentFrame,previousFrame,Smoothness)
% Derivatives of Brightness
%mask
hx = -fspecial('sobel');
hy = hx';
%derivatives
Ix = double(imfilter(currentFrame,hx));
Iy = double(imfilter(currentFrame,hy));
It = double(currentFrame)-double(previousFrame);

% Compute Optical Flow
hvel = [0,1,0;1,0,1;0,1,0];
for iter = 1:10
    % Estimation for Velocities
    u_bar = imfilter(u,hvel);
    v_bar = imfilter(v,hvel);

    % Estimate Optical Flow
    u = u_bar - (Ix.*(Ix.*u_bar+Iy.*v_bar+It))./(alpha^2+Ix.^2+Iy.^2);
    v = v_bar - (Iy.*(Ix.*u_bar+Iy.*v_bar+It))./(alpha^2+Ix.^2+Iy.^2);
end
% Compute Optical Flow Vectors
OpticFlow = opticalFlow(u,v);
```

As seen in the previous code section the steps are:

- Computing the spatial and temporal derivatives I_x, I_y and I_t using the current and previous video frame.

- Estimate the average of the neighborhood flow vectors using the previous iteration (for first iteration average is zero).
- Calculate the optical flow vectors using the iterative formula given by Horn and Schunck.

The obtained optical flow vectors are the basis for the following computations. To compare different methods and play with the possibilities Matlab provides we implemented other optical flow techniques. This was done utilizing the Matlab builtin function *estimateFlow* and defining the specific algorithm before. In addition to Horn-Schunck the two methods Lucas-Kanade and Gunnar Farnebäck were implemented

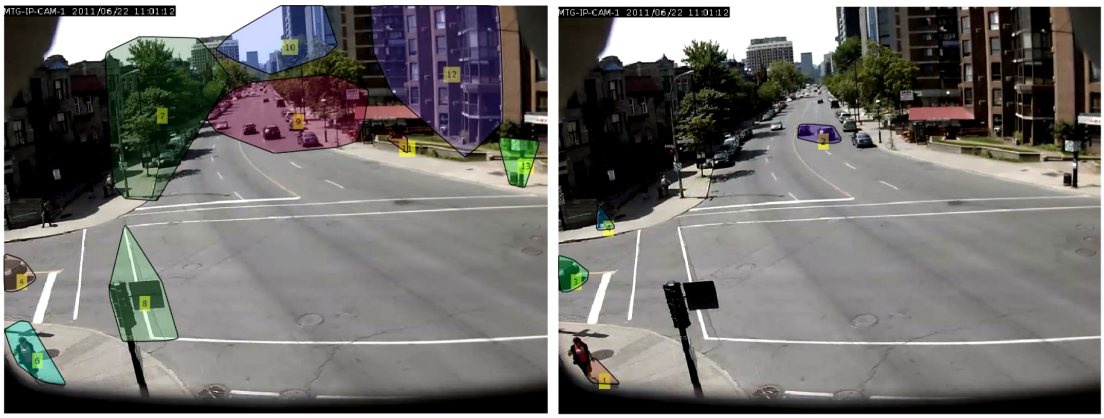
3.3 Stabilizer

We have noticed that the optical flow is very sensitive to camera movements. Slight shakiness of the camera due to wind (in the traffic surveillance) or the effector hand can create optical vectors for all the pixels of the image occluding the real detections of moving objects. To avoid this problem, we assumed that most part of the picture will belong to steady objects, so if we perform a corner detector, most of the corners will belong to the steady background and only a few of them to the moving foreground. This assumption will avoid the computationally expensive identification of the real steady objects that is employed in most of the image stabilization methods.

Once we have the corners, we will compute the median of the velocities in the x and the y directions of these corners. If the image is steady, we expect that the median will fall in one background point so the correction of the flow will be zero. But if the image is shaking, we expect that the background corners will behave similarly and, therefore, the optical flow created by the camera movement is the same as the optical flow of the background corners. We have chosen the median instead of other functions because other features like the mean will always take into account the vectors of the foreground (moving objects) and therefore contaminating the correction.

Finally, we just have to shift the optical flow velocities by the velocities of the median. We can see the created function in the following code section.

```
function [OpticFlow] = stabilize_flow_image(im_grey,in_flow)
    C = detectHarrisFeatures(im_grey);
    % Obtain the velocities images
    im_Vx = in_flow.Vx;
    im_Vy = in_flow.Vy;
    % Make the corner points readable for indexing
    points = ...
        round((C.Location(:,1))-1)*size(im_grey,1)+round(C.Location(:,2));
    % Correct the velocity image with the median
    im_Vx = im_Vx-median(im_Vx(points));
    im_Vy = im_Vy-median(im_Vy(points));
    % Create the new optical flow structure
    OpticFlow = opticalFlow(im_Vx,im_Vy);
end
```



(a) Object detection without stabilization. (b) Object detection with stabilization.

Figure 3: Importance of the Stabilizer

As we can see in Fig. 3, in the same frame without stabilization, we will detect many of the steady background as moving objects due to a camera movement detected in the optical flow vectors. After computing the median of the optical flow and the correction, we are going to detect the objects that are actually moving in the image.

3.4 Detection

The detection of the moving objects will be performed after the computation of the optical flow. As the orientation of the velocity vectors will depend on the movements of single pixels, we can not rely on this information to detect whole objects. Instead, we are going to use the magnitude of the optical flow vectors.

The first step in this process is to binarize the image. To obtain the pixels that are actually moving and therefore of interest we need to define a fixed threshold – we have chosen that the pixels with a velocity magnitude less or equal than 0.5 will not be considered moving. With this binarized image with ones in the moving pixels, we are going to split convex objects. We will close the detections, fill the possible holes and erase the elements containing less than 100 pixels – this threshold is also empirically chosen. After that, we can use the function *regionprops* with the option *'ConvexHull'* to convert the detections into continuous convex objects. Finally, we will erase the elements that are smaller than one thousand of the image – also empirically chosen – and the moving objects that cover more than 80% of the entire image – assuming that in the image we have a background and the moving objects are in the foreground, so they will not cover all the lens.

```
function [objFrame, obj] = segment_objects(OFvariable,threshlim)
    % Convert absolute velocity into binary image
    Magnitude = OFvariable.Magnitude;
    bw = Magnitude > threshlim;

    % Perform morphological operations for image optimization
    structElem = strel('disk',10);          % structuring element
    bw_closed = imclose(bw,structElem);     % close objects
```

```
    bw_filled = imfill(bw_closed,'holes');    % fill closed objects
    bw_removed = bwareaopen(bw_filled, 100); % remove objects smaller ...
        100px
    bw_convhull = bwconvhull(bw_removed,'objects');
    objFrame = bw_convhull;

    % Information about detected objectsx
    props = regionprops(objFrame,'ConvexHull');

    % Polyshape objects for plotting
    obj = [];
    im_area = size(Magnitude,1)*size(Magnitude,2);
    for iter = 1:numel(props)
        aux_obj = polyshape(props(iter).ConvexHull);
        aux_area = area(aux_obj);
        if(aux_area> 0.001*im_area && aux_area < 0.8*im_area)
            obj = [obj, aux_obj];
        end
    end
end
```

The results of the function deliver the image representing the convex hulls of each moving object. The second output is the polyshape of each object which is later used for highlighting the detected objects in the video frames.

3.5 Tracking

Finally, we have to do the tracking of the objects. This will be performed through the identification of previous detections and the indexing of new ones.

For all the objects detected in the current frame, we are going to compute the centroid and we will move it one step backwards with the velocity of the object in this point – note that for big uniform objects, the velocity of the center is usually zero, but we assume that this object will not move too fast and the center in the current pixel will still belong to the convex hull of the same object in the previous frame. This correction is only critical for small moving objects whose velocity could exceed this criteria.

After that, we will look if the centers of the moving objects belonged to a previous detected object. If this was named, we will assign the same name to the new object, but if this object was not named, we will give it a new name. This condition will ensure that we are going to name the objects detected in two consecutive frames – avoiding many of the false detections – and that we will keep the names of the previously detected objects.

To perform these conditions, we needed to look if one set of points belonged to each one of the convex hulls of the previous frame. This can be done in a plane with the function that given a set of points to consider and the corner points of a convex hull, returns a vector of booleans indicating if the point belonged to the hull created by John D'Errico [1] in a Matlab repository.

And now we can define the tracking function, that needs the current detections, the previous detections, the names of those detections, the flow structure and the naming index as input parameters.

```
function [curr_indexes, obj_counter, points] = ...
    tracking_function(curr_obj, prev_objs, prev_indexes, flow_str, ...
    obj_counter)

    curr_indexes = zeros(1,numel(curr_obj));
    points = zeros(numel(curr_obj), 2);

    % look for the searching points
    for i_obj = 1:numel(curr_obj)
        [cy, cx] = centroid(curr_obj(i_obj));
        cx = round(cx); cy = round(cy);
        Vx = flow_str.Vx(cx,cy); Vy = flow_str.Vy(cx,cy);
        points(i_obj,:) = [cy-Vy, cx-Vx];
    end

    % if cx-Vx, cy-Vy belongs to a previous object, then we consider it
    % it a consistent detection
    for j_obj = 1:numel(prev_objs) % look for all the previous objects
        aux = inhull(points, prev_objs(j_obj).Vertices);
        if(any(aux)) % some point belonged to this object
            aux = find(aux);
            if prev_indexes(j_obj) ≠ 0
                % if it was aready named
                curr_indexes(aux(1)) = prev_indexes(j_obj);
            else
                % if not, we will name it now
                obj_counter = obj_counter+1;
                curr_indexes(aux(1)) = obj_counter;
            end
        end
    end
end
```

4 Application and Experiments

To ensure that the designed algorithms to track moving objects are working, we used some video footage. We also used the algorithms on self made videos to identify their boundaries. In this section there are some images shown to compare the outcome of the different algorithms as well as a tracking benchmark. There also can be found a description of some difficulties we discovered during the process.

4.1 Tracking benchmark

The first thing we did is to benchmark the methods to see if a real-time processing was possible. To do so, we timed each one of the processes involved in the tracking and naming of the objects, but not the time required to draw the figures. We did the experiment with a traffic surveillance video of 600×800 pixels with the three available optical flow methods (Lucas-Kanade, Gunnar Farnebäck and the custom Horn-Schunck) along the 4000 frames of the video to obtain the following statistics:

(s)	pre-processing	segmentation	tracking	LK	GF	HS	stabilize
mean	0.00172	0.02265	0.00095	0.01133	0.08225	0.00798	0.02654
std	0.00027	0.01395	0.00164	0.00979	0.00322	0.00090	0.00757

As we can see, most of the variance of timing will be due to the segmentation part, as we may find many moving objects that we have to analyze at the same time. To ensure a 97.5% of confidence of processing an image on time, we have to give each frame at least $mean(processing_time) + 1.96 * std(processing_time)$ seconds to each frame, so the frame rate per second will be its inverse.

For the Lucas-Kanade method, we managed to ensure 14 fps without stabilization (`https://youtu.be/p65gR02pXaQ`) and 9 fps with stabilization (`https://youtu.be/Q5NWCcp-5h8`). For the Gunnar Farnebaäck method we arrived only to 7 fps without stabilization (`https://youtu.be/ZANPj5w61mA`) and 5 with stabilization (`https://youtu.be/2t61psL1j5o`). For the Horn-Schunck method, we reached 15 fps without stabilization (`https://youtu.be/FWNwXoFJ9ko`) and 9 fps with stabilization (`https://youtu.be/9AY_zi-1uhc`). The links to YouTube videos for each of the methods are provided for visual inspection.

Due to the time performance and the consistency of the optical flow generated, we have chosen the Lucas-Kanade algorithm to search the optical flow and added an stabilization algorithm of the image to clear part of the shakiness of the camera. This gave us a performance of about 15 fps in mean, but to ensure a correct performance of the method, we should not exceed 9 fps (97.5% of confidence).

We have also validated our method in a traffic surveillance video and in self made videos (`https://youtu.be/nVWM59Pejsc`) althoug not in real time as the videos are captured at 30 fps. To employ this on a real time aplication, we have to downgrade the capturing rate of the camera (to 9 fps) or try to take advantage of the parallelization of a graphics card, as we have seen that the CPU performance nowadays in a Matlab environment is not enough.

4.2 Downsides of the optical flow tracking

In the videos that we made in the street, we found that steady objects that occlude partially moving objects will also split the detection of the objects like the trees in Fig. 4 that split vertically the cars in the background.

Due to the definition of optical flow, we can only use the magnitude of the vectors to determine the moving objects. This means that we are not going to be able to separate objects that are moving one next to another, because the convex hull algorithm will merge them together (Figure 5a).

Another downside of this method is that moving flat surfaces will not have any optical flow. Therefore, we are only going to detect the movement of the object in the edges (Figure 5b). One way of addressing this problem could be by object recognition of the detected objects and looking near the shifted vectors given by the optical flow of the edges. But this could be very computationally expensive.

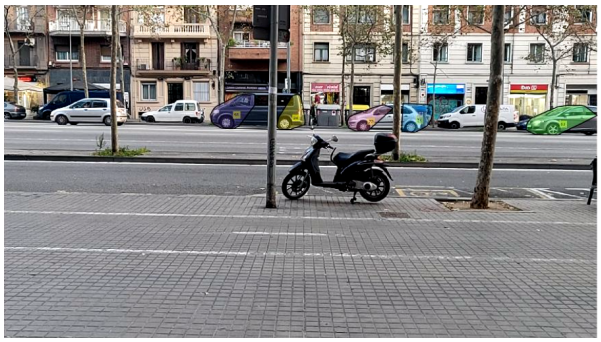


Figure 4: Trees occluding moving objects

(a) Two different moving objects that cannot be differentiated based on the optical flow.

(b) Big flat moving object split in three elements due to the lack of optical flow in the interior of the object.

Figure 5: Issues in the object detection

5 Summary and Conclusion

In general, using optical flow for tracking moving objects in videos seems to be a good method. Depending on the technique of optical flow, the implementation is simple. After obtaining the vector field, the object detection and maybe even identification is a process, which can take lots of computational time and also bring new difficulties with it.

As we can see in the videos there are some clear differences between the used optical flow techniques. The Horn-Schunck method is very sensible to small noises that cannot be stabilized with our method. The Gunnar Farnebäck algorithm is very sensitive to small camera movements and tend to generate bigger optical flow vectors which cover more uniformly the image. This results in a overwhelming detection of objects and is much more likely to generate bigger detections and therefore tending to merge different objects. In the other hand, the Lucas-Kanade algorithm has proven to be pretty stable from frame to frame and responds pretty well to the stabilization algorithm that we proposed.

In conclusion, for the sample videos and the implemented detection algorithm the Lucas-Kanade method has led to the best overall results.

References

[1] Jhon D'Errico. Inhull, Sep 2012.

[2] Chuan-en Lin. Introduction to motion estimation with optical flow, Apr 2020.

[3] Peter O'Donovan. Optical flow: Techniques and applications. *International Journal of Computer Vision*, pages 1–26, 2005.