86735: Computer Vision
Optical flow

Due date: Wed 12:00, 25.11.15

Prof. F. Odone, Prof. F. Solari, M. Chessa, N. Noceti

Rabbia Asghar, BEng, Ernest Skrzypczyk, BSc

Optical flow

Motion analysis is a process of analysing two or more consecutive images in the image sequence to detect motion. An image is a 2D array of pixels, depicting a scene projected from the 3D world. In order to extract information like movement of an object, more than one image is needed.

Among the various available techniques for motion analysis, there is the detection with background subtraction and Lucas-Kanade method. The scope of this assignment is to analyse image sequences taken from stationary camera and with fixed background. A very important assumption in this analysis is that the illumination is constant in the observation interval. Thus, the source of change in pixel value in image sequences is either noise or relative motion.

The goal of the practical laboratory session was to perform tasks using those methods and describe the results.

Change detection with background subtraction

A very simple technique to detect moving region in an image sequence is to compare it with a background model. Background model is the capture of a scene with no moving objects. It must be noted that two images of the very same scene can almost never be same. The images may appear similar to naked eye, but if they are compared pixel by pixel, the difference is evident. This is due to the noise recorded by a camera when it captures images. Following image shows the difference between two grey scale images of the background model. It can be observed that the difference in pixel values is very small, ranging from 0 to 20.

Task 1 takes image sequences with motion, one by one, and subtracts them with the background image. Once binarizing process through thresholding is applied, regions of motion can be highlighted. The value for thresholding needs to be carefully selected so that information is not lost but significant differences in image are detected. Following figures $(1 \div 4)$ represent selected results acquired using different values for thresholding.

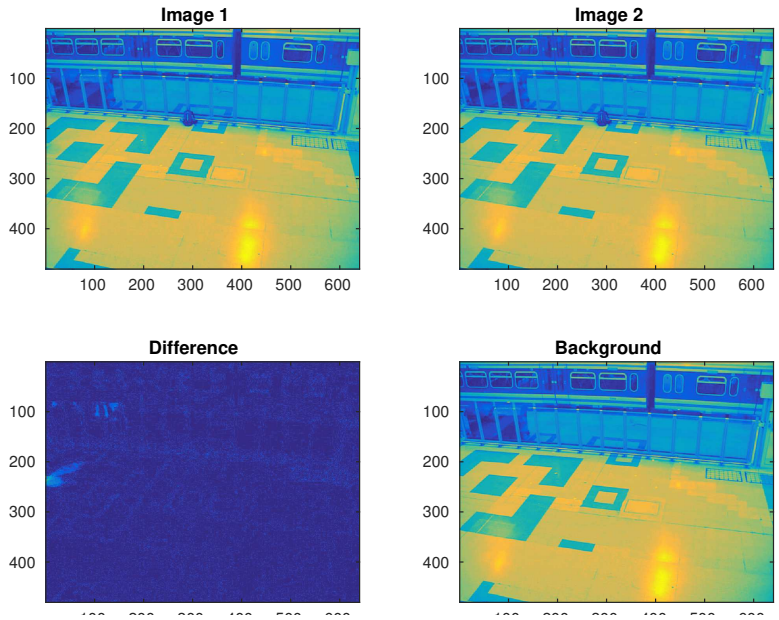

Figure 1: Background subtraction of two empty images shows the influence of noise from the image acquisition equipment.

This technique is simple and easy to detect motion, however it lacks optical flow information, which would be required for the purpose of tracking objects for example.

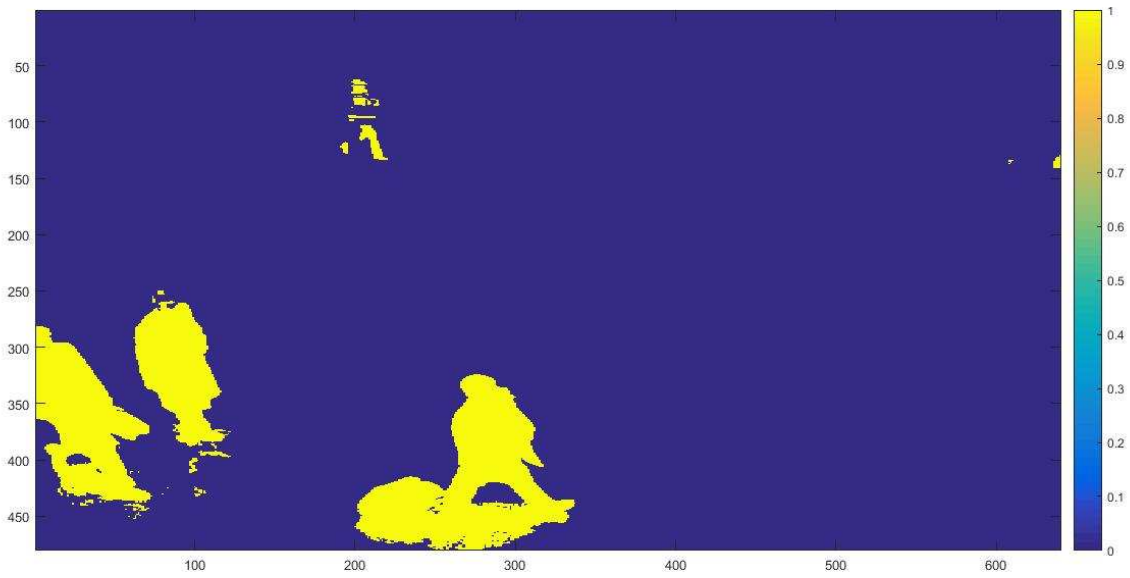

Figure 2: Background subtraction with threshold = 30

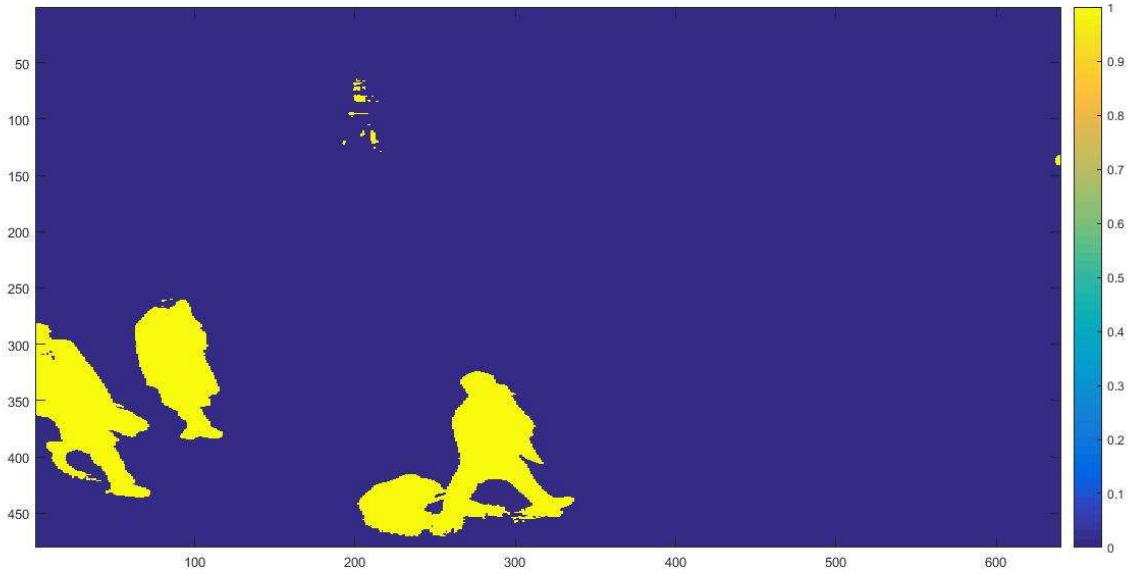

Figure 3: Background subtraction with threshold = 50

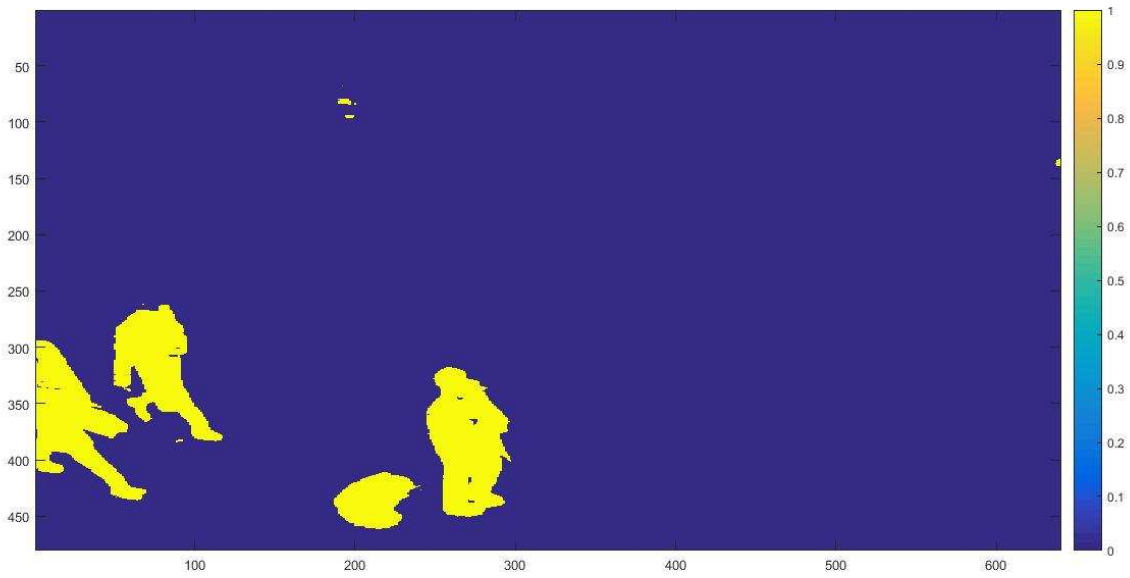

Figure 4: Background subtraction with threshold = 70

With lower thresholds more of the image movement is acquired, however the algorithm is most likely to also pick up more noise and other irrelevant data. The critical assumption of constant lighting seems not to be really fulfilled. Of course the acquisition equipment introduces its own noise. The question is which of those sources has more impact on the overall noise. It is assumed, that the lighting has no fluctuations and stays constant.

With a higher threshold the movement of the woman in the top part of the image is almost not recognizable. This is also the result of the specific environment, like the dividing glass or other structures, but also of the acquisition process itself, since the moving object is further away, so it is smaller and has a lower contrast in comparison to objects moving directly in front of the camera.

This simple method of background subtraction is not only low on resources consumption, since there are only two significant operation of subtraction and binarizing involved. Script (1) shows the implementation of this method. Additional certain functionalities have been expanded. It is possible to render a video of the sequence of images. This could be very useful for further processing from the point of view of computer vision, image processing or other science disciplines. The videos can be rendered by setting the option `videorender` to 1 and `test` to 0.

Script 1: L04_1.m: Task 1 - Visualizes noise in background subtraction and effect of different threshold (a video rendering is possible).

```
%% Computer Vision - L04 - Optical Flow
%% Rabbia Asghar, Ernest Skrzypczyk
%% 21.10.2015
%% Matlab 8.4.0.150421 (R2014b)
%% TASK 1: Change detection with background subtraction

close all; clear all; clc;

%Script parameters
imageshow = 1; %Option for showing images
imagerender = 1; %Option for rendering images
imagesave = 1; %Option for saving images
lowmem = 1; %Option for saving memory through less open figures at a time
imageleave = 1; %Option for waiting after an image has been processed
imageextension = 'pdf'; %Image file format
videorender = 0; %Option for rendering a video

%Setting basic script options
screensize = get(0,'ScreenSize'); %Screen size
set(groot, 'defaultFigurePosition', [screensize(3)/6, screensize(4)/6,  ▼20
    20▲ screensize(3)/1.5, screensize(4)/1.5]);
set(groot, 'defaultFigurePaperUnits', 'points', 'defaultFigurePaperSize',  ▼21
    21▲ [1366 1024])

%%Task 1 - Change detection with background subtraction
if ispc
  Image1 = imread('task1\EmptyScene01.jpg');
  Image2 = imread('task1\EmptyScene02.jpg');
  ImageB = imread('task1\Background.jpg');
elseif isunix
  Image1 = imread('task1/EmptyScene01.jpg');
  Image2 = imread('task1/EmptyScene02.jpg');
  ImageB = imread('task1/Background.jpg');
end
Image1GS = rgb2gray(Image1);
Image2GS = rgb2gray(Image2);
Background = rgb2gray(ImageB);

Difference = abs(double(Image1GS) - double(Image2GS));
imagesc(Difference); colorbar;
if imagesave == 1
  saveas(gcf, ['L04_1_1'], imageextension);
end

figure;
subplot(2, 2, 1); imagesc(Image1GS); title('Image 1'); colormap('parula');
subplot(2, 2, 2); imagesc(Image2GS); title('Image 2'); colormap('parula');
subplot(2, 2, 3); imagesc(Difference); title('Difference'); colormap('parula') ▼46
    46▲ ;
subplot(2, 2, 4); imagesc(Background); title('Background'); colormap('parula') ▼47
    47▲ ;
```

```
if imagesave == 1
  saveas(gcf, ['L04_1_A'], imageextension);
end


if imageleave == 1; pause; end;
if lowmem == 1; close all; end

%Image sequence from survaillance material
figure;
%~ for Threshold = 1:8:255
for Threshold = [1, 8, 64, 128, 224]
  if videorender == 1;
    outputVideo = VideoWriter(fullfile(pwd, ['L04_1_', num2str(Threshold), '. ▼61
    61▲ avi']));
    outputVideo.FrameRate = 10;
    open(outputVideo);
  end
  for i = 250:320 %Images range
  if ispc
    filename = sprintf('task1\videosurveillance\frame%4.4d.jpg', i);
  elseif isunix
    filename = sprintf('task1/videosurveillance/frame%4.4d.jpg', i);
  end
    InputImage = double(rgb2gray(imread(filename)));
    Difference = abs(InputImage - double(Background));
    Frame = Difference > Threshold;
    imagesc(Frame); title(['Difference binarized with threshold = ', num2str( ▼74
    74▲ Threshold)]); colormap('gray');
    pause(0.1); % Pseudo framerate of less than 10 fps
    if videorender == 1;
      writeVideo(outputVideo, double(Frame));
      writeVideo(outputVideo, double(Frame));
    end
  end
  if videorender == 1;
    close(outputVideo);
  end
end
```

The above script is commented in a self explanatory way and according to requirements of the laboratory session and beyond. Additional functionality has been added.

Lucas-Kanade algorithm to compute optical flow from a sequence of images

Lucas–Kanade method is a widely used technique for optical flow estimation. It was developed by Bruce D. Lucas and Takeo Kanade.

Optical flow is a velocity vector field which represents the motion of objects in a given image plane. A strong assumption is made to be able to compute optical flow using image brightness constancy equation. The assumption is that in all image sequences, illumination is constant. Another assumption is that the motion in consecutive image frames is very small and points do not move far. With these assumptions, the equation (1) is derived as following:

$$\begin{gathered}\frac{dI}{dt} = 0 \\ \frac{dI(x;y;t)}{dx} = \frac{\delta I}{\delta x}\frac{dx}{dt} + \frac{\delta I}{\delta y}\frac{dy}{dt} + \frac{\delta I}{\delta t} = 0 \\ \frac{\delta I}{\delta x}V_x + \frac{\delta I}{\delta t}V_y + \frac{\delta I}{\delta t} = 0 \\ (\nabla I)^T u + I_t = 0 \end{gathered} \tag{1}$$

Where ∇I consists of partial derivative of Image with respect to position x and y, u is the optical flow vector in direction of x and $y(Vx, Vy)$ as well as I_t, is partial derivative of the image with respect to time t.

This equation has two unknowns and so cannot be solved at individual pixels. This is known as the aperture problem of the optical flow algorithms. The component of the flow in the gradient direction is determined, while the component of the flow parallel to an edge is unknown.

In order to find the optical flow another set of equations is needed, given by some additional constraint. Lucas-Kanade is one of the techniques to compute u, optical flow vector. For an additional constraint, it assumes that the flow is essentially constant in a local neighbourhood of the pixel under consideration. For example, if we use a $3x3$ window, this would mean that the value of V_x and V_y is constant in the neighbourhood. That gives us 9 equations in total, one for each pixel. The equation for optical flow can be re-written as presented by equation (2)

$$(\nabla I(x;t))^T u + I_t(x_i;t) = 0, \quad x_i \in N(neighbourhood) \tag{2}$$

This technique ensures that there are more relevant information than unknowns. These equations can be written in matrix form (3) and can be solved using least square method.

$$Au = b \tag{3}$$

Where

$$A = \begin{bmatrix} \nabla I(x_1;t)^T \\ \nabla I(x_2;t)^T \\ \vdots \\ \nabla I(x_m;t)^T \end{bmatrix}, \quad b = \begin{bmatrix} -I_t(x_1;t) \\ -I_t(x_2;t) \\ \vdots \\ -I_t(x_m;t) \end{bmatrix}$$

and m corresponds to the total number of elements in neighbourhood N. Equation (3) can be solved using minimum least square solution as presented in equation (4).

$$\begin{aligned} \left(A^T A\right) u &= A^T b \\ u &= \left(A^T A\right)^{-1} A^T b \\ u &= A^{\dagger} b \end{aligned} \tag{4}$$

where $A^{\dagger}$ is the pseudoinverse of A

Implementation

LucasKanade function defined in `LucasKanade.m` file takes two consecutive image sequences in grey scale as input, along with the window size. Window size is the size of the neighbourhood matrix N.

```
%% Computer Vision - L04 - Motion Analysis in Image Sequences
%% Created by: Rabbia Asghar, Ernest Skrzypczyk
%% 11 November 2015

%~ function[U, V] = LucasKanade(ImageFrame1, ImageFrame2, WindowSize, EPS);
function[U, V, Condition] = LucasKanade(ImageFrame1, ImageFrame2, WindowSize);
```

The function `LucasKanade()` first computes the partial derivative ∇I of the `ImageFrame2` with respect to x and y. It must be noted that in order to compute I_x and I_y, separate horizontal and vertical derivative filters must be used. Gradient vector is computed at each pixel to compute gradient by convolving the image with filters. The function uses a simple forward difference filter to compute partial derivatives: $[1, -1]$

```
% Computing partial derivatives in x and y dimensions
Kernel = [1, -1];

Ix = conv2(ImageFrame2, Kernel, 'same');
Iy = conv2(ImageFrame2, Kernel', 'same');
```

The function then computes I_t, partial derivative of image with respect to time. This is simply computed by subtracting `ImageFrame1` from `ImageFrame2`.

```
It = ImageFrame2 - ImageFrame1;
```

The function then prepares matrix A and b, as discussed previously, for every pixel in the image. The size of matrix A and b, depends on the window size selected. For example window size of 3, gives $A = 9\text{x}2$ matrix and for window size of 5, $A = 25\text{x}2$ matrix.

```
    %extract N points in neighborhood: WindowSize x WindowSize
    A_x_elements = (Ix(i - WindowSizeHalf:i + WindowSizeHalf, j - WindowSizeHalf:j + WindowSizeHalf))';
    A_x_vector = A_x_elements(:)';

    A_y_elements = (Iy(i - WindowSizeHalf:i + WindowSizeHalf, j - WindowSizeHalf:j + WindowSizeHalf))';
    A_y_vector = A_y_elements(:)';

    b_elements = (It(i - WindowSizeHalf:i + WindowSizeHalf, j - WindowSizeHalf:j + WindowSizeHalf))';
    b_vector = -b_elements(:);

```

```
    A = [A_x_vector', A_y_vector']; % A = [A_x A_y]
```

Vector u can be computed by finding pseudo inverse of A and multiplying with b. MATLAB offers the inbuilt function `pinv()` to compute pseudoinverse of a matrix.

```
50      A_PI = pinv(A); % A_PI = Pseudo inverse of A
        %~ A_PI = pinv(A, EPS); % A_PI = Pseudo inverse of A with tolerance EPS
        SmallU = A_PI * b_vector;
        U(i, j) = SmallU(1);
        V(i, j) = SmallU(2);
```

The function returns matrix U and V, where U is optical flow vector in direction of X and V is in direction of Y.

Main script

The script `L04_2_2.m` runs a loop to compute and display optical flow for 2 consecutive image frames in image sequences. The function reads image and converts them into grayscale and casts them to double.

```
ImageFrame1 = double(rgb2gray(ImageFrame1_rgb));
ImageFrame2 = double(rgb2gray(ImageFrame2_rgb));
Optical flow is computed by calling LucasKanade function.
[U,V] = LucasKanade(ImageFrame1,ImageFrame2,windowSize);
```

U and V components are displayed on top of the image using quiver function. Since U and V are computed for every pixel, it would be too concentrated to display all vectors and thus very difficult to analyse results and heavy on resources. Because of this, U and V are first subsampled, so every SS-th coordinate is used only for the flow vectors.

Commented out is an alternative call for the function `LucasKanade()` involving passing a tolerance EPS, which is an optional argument for the function `pinv()`. By raising the tolerance an increase in performance under the same load is expected, however this has not been extensively tested. Under specific circumstances a higher EPS could introduce inconsistencies in the code and lead to overall failure of the algorithm, therefore caution is advised and further testing required.

```
84    end
85    % Subsample u and v
      u_q = U(1:SubSampleStep:end, 1:SubSampleStep:end);
```

```
90    end
      if imagerender == 1
        if lowmem == 0
          quiver(X, Y, U, V, 'g'); % Draw optical flow vectors
        end
95      quiver(X_q, Y_q, u_q, v_q, 'r'); % Draw subsampled optical flow vectors
      end
      if imagesave == 1 % Save image
        saveas(gcf, ['L04_2_2_IF', num2str(i), 'WS', num2str(WindowSize), 'SS', num2str ▼98
       98▲ (SubSampleStep)], imageextension);
```

X and Y coordinates to be used for display of U and V are prepared outside and before loop. Since, it is the same for every iteration; this is done to increase computational efficiency.

```
54  % Get coordinates for u and v in the original frame
55  [X, Y] = meshgrid(1:Columns, 1:Rows);
    X_q = X(1:SubSampleStep:end, 1:SubSampleStep:end);
```

```
Y_q = Y(1:SubSampleStep:end, 1:SubSampleStep:end);
```

Script 2: L04_2_3.m: Script with hierarchical approach on video surveillance sequence.

```
%% Computer Vision - L04 - Optical Flow
%% Rabbia Asghar, Ernest Skrzypczyk
%% 21.10.2015
%% Matlab 8.4.0.150421 (R2014b)
%% TASK 2: Lucas-Kanade algorithm to compute optical flow
%% Hierarchical approach on video surveillance sequence

close all; clear all; clc;

% Script parameters
imageshow = 0; % Option for showing images
imagerender = 1; % Option for rendering images
imagesave = 1; % Option for saving images
lowmem = 1; % Option for saving memory through less open figures and less grahical objects at a time
imageleave = 1; % Option for waiting after an image has been processed
imageextension = 'pdf'; %Image file format
videorender = 0; % Option for rendering a video
flip = 0; % Option for using fliped quiver results
comparison = 0; % Option for comparison between LK and HLK
test = 1; % Option for test run
% Setting basic script options
screensize = get(0, 'ScreenSize'); % Screen size
set(groot, 'defaultFigurePosition', [screensize(3)/6, screensize(4)/6, screensize(3)/1.5, screensize(4)/1.5]);
set(groot, 'defaultFigurePaperUnits', 'points', 'defaultFigurePaperSize', [1366 1024]);
if imagerender == 1; set(groot, 'defaultFigureVisible', 'on', 'defaultFigureRenderer', 'opengl'); end %painters might be very slow
if imagesave == 1; set(groot, 'defaultFigureRenderer', 'opengl');%painters');
if imageshow == 0; set(groot, 'defaultFigureVisible', 'off'); end; end
if test == 1; loopend = 3; else loopend = 70; end % Range of images to be processed

Images = [];

% Reading first frame
if ispc
  filename = ['Material_Task1\videosurveillance\frame0250.jpg'];
elseif isunix
  filename = ['Material_Task1/videosurveillance/frame0250.jpg'];
end
disp(['Loaded first image: ', filename]);
Images{1} = imread(filename, 'jpg');

% Dimensions of the image and sequential images
[Rows, Columns] = size(double(rgb2gray(Images{1})));

%% Main parameters
% Further parametrizing possible with another set of 'for' loops
WindowSize = 5; % Window size for Lucas Kana
Iterations = 2; % Number of refine iterations
```

```
SubSampleStep = 8; % Subsampling step -- (5, 10)

%~ for WindowSize = [3, 5, 9, 12, 24, 48, 64]
%~ for Iterations = [2, 3, 5, 10, 20, 50]
%~ for SubSampleStep = [5, 8, 10, 16, 32, 64, 128]

%% Main loop
for i = 2:loopend
% Concating filenames / using dir('*.jpg') instead possible
if ispc
  filename = ['Material_Task1\videosurveillance\frame0', int2str(249 + i), '. ▼58
   58▲ jpg'];
elseif isunix
  filename = ['Material_Task1/videosurveillance/frame0', int2str(249 + i), '. ▼60
   60▲ jpg'];
end
  disp(['Processing image: ', filename]);
  Images{i} = imread(filename, 'jpg');

  ImageFrame1 = double(rgb2gray(Images{i - 1}));
  ImageFrame2 = double(rgb2gray(Images{i}));

  if comparison == 1
    %% Invoking Lucas-Kanade algorithm
    disp(['Parameters: IF = ', num2str(i), ', WS = ', num2str(WindowSize), ',  ▼70
    70▲ IT = ', num2str(Iterations), ', SS = ', num2str(SubSampleStep)]);
    disp('Invoking full resolution and subsampled Lucas-Kanade algorithm on  ▼71
    71▲ image sequence');
    [U, V, Condition] = LucasKanade(ImageFrame1, ImageFrame2, WindowSize);
    [X, Y] = meshgrid(1:Columns, 1:Rows);
    if imagerender == 1
      figure;
      title(['IF', num2str(i), 'WS', num2str(WindowSize), 'LV', num2str(Levels ▼76
    76▲ ), 'IT', num2str(Iterations), 'SS', num2str(SubSampleStep)]);
      imshow(Images{i});
      hold on;
    end
    % Subsample u and v
    u_q = U(1:SubSampleStep:end, 1:SubSampleStep:end);
    v_q = V(1:SubSampleStep:end, 1:SubSampleStep:end);

    % Get coordinates for u and v in the original frame
    [X, Y] = meshgrid(1:Columns, 1:Rows);
    X_q = X(1:SubSampleStep:end, 1:SubSampleStep:end);
    Y_q = Y(1:SubSampleStep:end, 1:SubSampleStep:end);

    if flip == 1 % Flip the vectors
      U = flipud(U); V = flipud(V);
    end
    if imagerender == 1
      if lowmem == 0
        quiver(X, Y, U, V, 'g'); % Draw optical flow vectors
      end
```

```
    quiver(X_q, Y_q, u_q, v_q, 'r'); % Draw subsampled optical flow vectors
  end
  if imagesave == 1 % Save image
    saveas(gcf, ['L04_2_3_IF', num2str(i), 'WS', num2str(WindowSize), 'LV',  ▼99
  99▲ num2str(Levels), 'IT', num2str(Iterations), 'SS', num2str( ▼99
  99▲ SubSampleStep)], imageextension);
  end
end
%% Hierarchical Lucas-Kanade algorithm
% Subsampling
if comparison == 0
  uv = 10; % Manual estimation in pixels
else
  uv = max([U(:); V(:)]); % Determine largest optical flow vectors from  ▼107
  107▲ full resolution Lucas-Kanade results
  clear X, Y, X_q, Y_q, u_q, v_q;
end
Levels = ceil(log2(uv)); % Pyramind levels
disp(['Parameters: IF = ', num2str(i), ', WS = ', num2str(WindowSize), ',  ▼111
  111▲ LV = ', num2str(Levels), ', IT = ', num2str(Iterations), ', SS = ',  ▼111
  111▲ num2str(SubSampleStep)]);
disp('Invoking hierarchical Lucas-Kanade algorithm on image sequence');
disp(['Anticipated optical flow change:', num2str(uv)]);

if imagerender == 1
  figure;
  imshow(Images{i});
  hold on;
end
[U, V, Condition] = LucasKanadeHierarchical(ImageFrame1, ImageFrame2,  ▼120
  120▲ Levels, WindowSize, Iterations, 0);

% Subsample u and v
u_q = U(1:SubSampleStep:end, 1:SubSampleStep:end);
v_q = V(1:SubSampleStep:end, 1:SubSampleStep:end);
% Get coordinates for u and v in the original frame
[X, Y] = meshgrid(1:Columns, 1:Rows);
X_q = X(1:SubSampleStep:end, 1:SubSampleStep:end);
Y_q = Y(1:SubSampleStep:end, 1:SubSampleStep:end);

if flip == 1 % Flip the vectors
  U = flipud(U); V = flipud(V);
end
if imagerender == 1
  title(['IF', num2str(i), 'WS', num2str(WindowSize), 'LV', num2str(Levels)  ▼134
  134▲ , 'IT', num2str(Iterations), 'SS', num2str(SubSampleStep)]);
  if lowmem == 0
    quiver(X, Y, U, V, 'g'); % Draw optical flow vectors
  end
  quiver(X_q, Y_q, u_q, v_q, 'r'); % Draw subsampled optical flow vectors
end
if imagesave == 1 % Save image
  saveas(gcf, ['L04_2_3_IF', num2str(i), 'WS', num2str(WindowSize), 'LV',  ▼141
```

```
    141▲ num2str(Levels), 'IT', num2str(Iterations), 'SS', num2str( ▼141
    141▲ SubSampleStep)], imageextension);
  end
end
%~ end % SubSampleStep
%~ end % Iterations
%~ end % WindowSize
```

The above script is sufficiently commented in a self explanatory way and according to requirements of the laboratory session and beyond.

Results

Sphere

Lucas-Kanade method is first implemented to image sequences of a sphere in main script `L04_2_1.m`. 'Sphere' available for analysis consists of computer generated image with black and yellow background of slanting lines and green and black sphere in the front. If the image sequences are run in order, it can be observed that the sphere is rotating in clockwise direction along its axis. Following images ($5 \div 8$) display the optical flow for image frame 10 in form of a vector field plotted with `quiver()`.

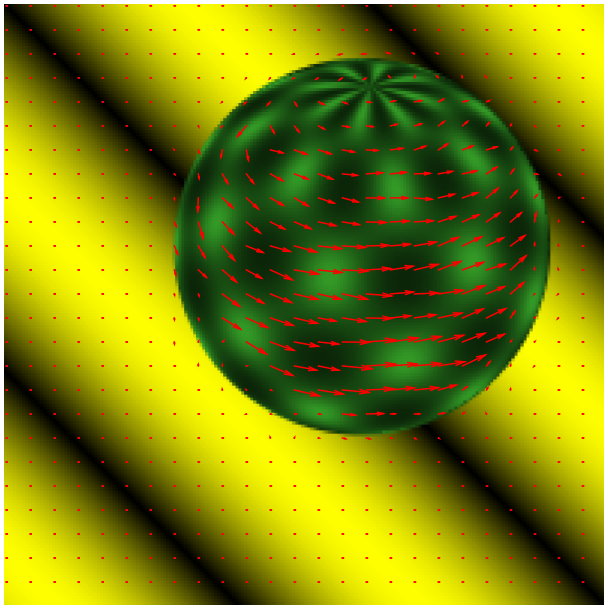

Figure 5: Optical flow for a sphere IF10WS13SS8

In most of the selected images, there is almost no optical flow detected in the background. This is because the image is computer generated and there is no noise. When zoomed on to the image result to study the optical flow, it can be observed that the vectors are pointing in the direction of rotation of sphere. Also, the optical flow vectors seem to have a higher gradient on the right and left side of the sphere. The following results compare the optical flow computed using different window sizes WS, which is indicated in the reference name, namely: 13, 3, 47 and 63.

Legend on file name reference:

- WS – **W**indow **s**ize
- IT – **I**teration
- LV – **L**ev**v**el
- SS – **S**ub**s**ampling step

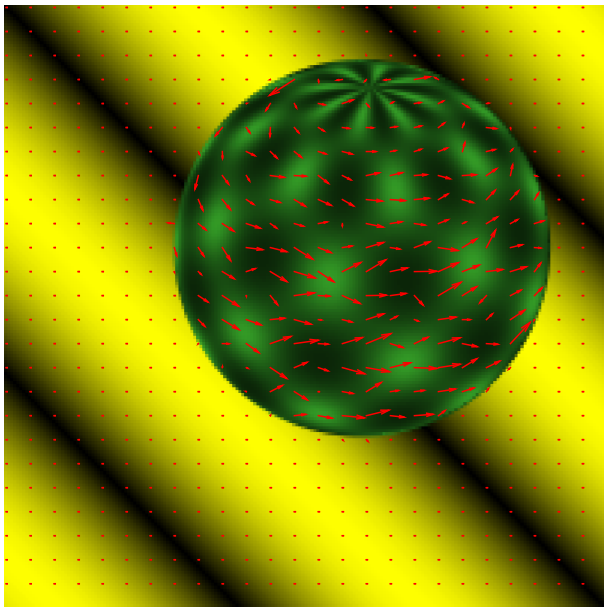

Figure 6: Optical flow for a sphere IF10WS3SS8

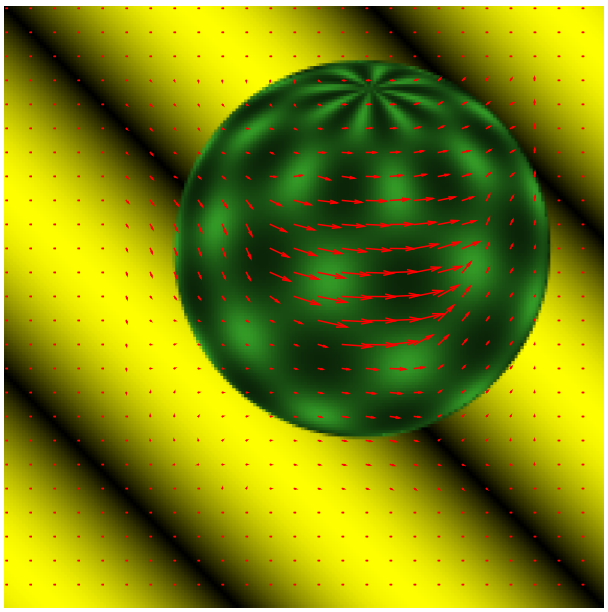

Figure 7: Optical flow for a sphere IF10WS47SS8

It can be observed that with a larger window size, the optical flow is more uniform, however there is a limit. If the window size is too big, it seems that *dots* are appearing, which indicate insufficiency of the algorithm and movement where there is none, namely the background. However one should consider technical comments in the last section "Conclusions and comments" on page 26. With other experiments it has been established that window size of 3, optical flow is more scattered but it is more sensitive and detailed. Figure (5) seems to represent the expected optical flow in the best manner with a window size of 13. This is because for a certain window size, Lucas-Kanade method assumes that the all pixels in the neighbourhood hold same optical flow vector, u and v. This way, for a window size of 9, the neighbourhood pixels of $9\text{x}9 = 81$ pixels hold the same u and v vector. Even though a larger window size offers less information about local pixels, it is useful to identify distinguished direction of motion of one complete object. Too large window size introduces distortion into the optical flow as shown in the image (8), too little window size saturates local movement more, however it can introduce more fluactations in the overall flow as seen on image (6).

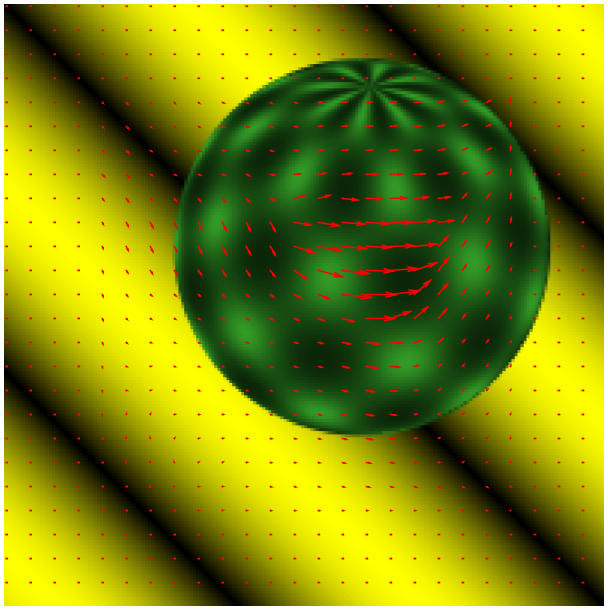

Figure 8: Optical flow for a sphere IF10WS63SS8

Video surveillance

The same script with minor alterations `L04_2_2.m` was then applied to image sequences "video surveillance". Unlike "sphere", this image sequence was acquired from a fixed camera. The images were previously analysed using background detection technique.

Following images ($9 \div 11$) show image frame 3 with window size 3, 5 and 9. It can be observed that many optical flow vectors can be found in the background. This is because images received from the camera are recorded with intrinsic noise. Another cause could be the fluctuations in lighting of the scenery, but it was already stated, that the assumption of constant luminosity is fulfilled.

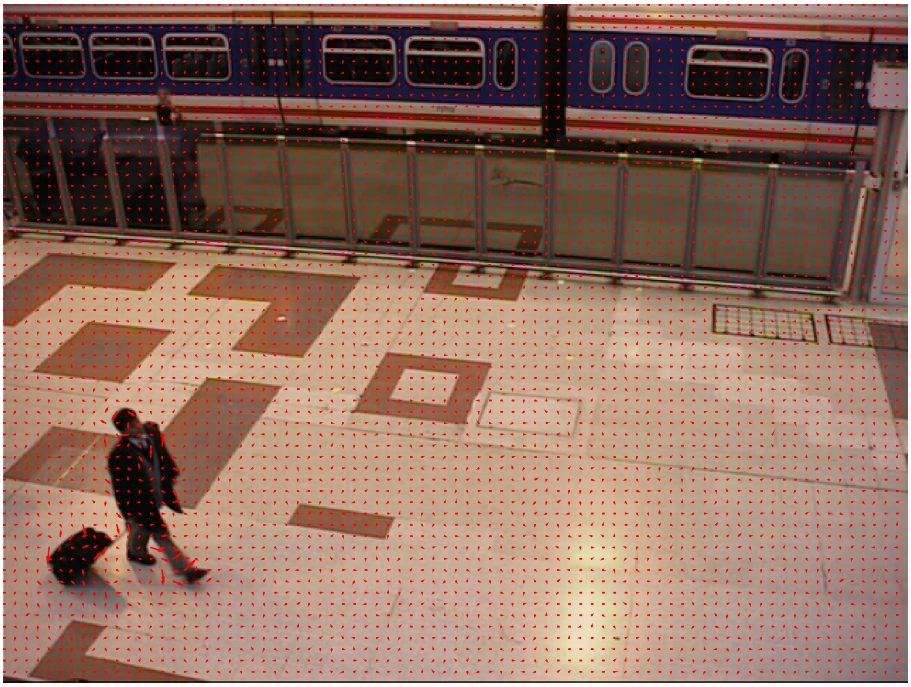

Figure 9: Optical flow for video surveillance sequence - image IF3WS3SS8

Figure 10: Optical flow for video surveillance sequence - image IF3WS5SS8

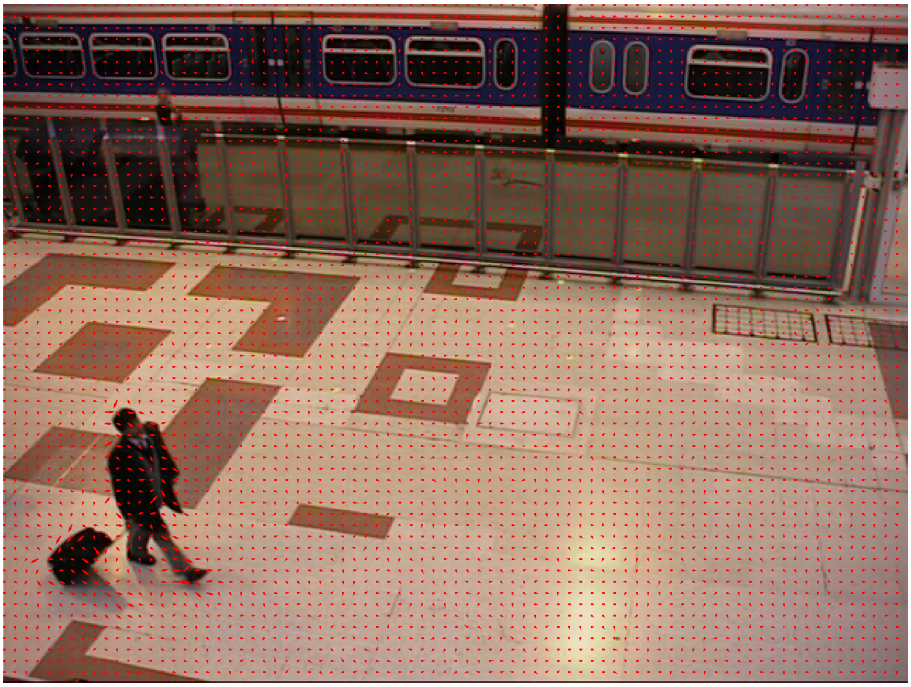

Figure 11: Optical flow for video surveillance sequence - image IF3WS9SS8

Also with a larger window size, more noisy optical flow is recorded in the background. This is again because of the erroneous assumption here that neighbourhood pixels have the same optical flow vector. However, the advantage of larger window size in this case is that optical flow vectors are more enhanced in the regions of motion. When the window size exceeds the size of the object of interest it might introduce additional distortions in the optical flow. As image (12) shows, the movement of the man and his bag in front of camera are recognized correctly. Also the flow of the shadows has been recognized. Please see last section "Conclusions and comments" on page 26 for further explanation.

What should be also noted are technical limitations of image acquisition, which can be seen here also. There is a distinct *aura* seen in the zoomed image `IF3WS9SS8` presented in image (12) around the man moving in front of the camera. Those artefacts are most likely because of the exposure time and the fact that a dynamic image is acquired. If the exposure time is long enough and the acquired moving object fast enough, there might be a significant reduction in light reflecting from the background behind the moving object, so that there is a significant difference between the covered and uncovered background by the object. This would explain the *aura effect* around moving objects, which introduce further noise for the Lucas-Kanade algorithm as seen in the images.

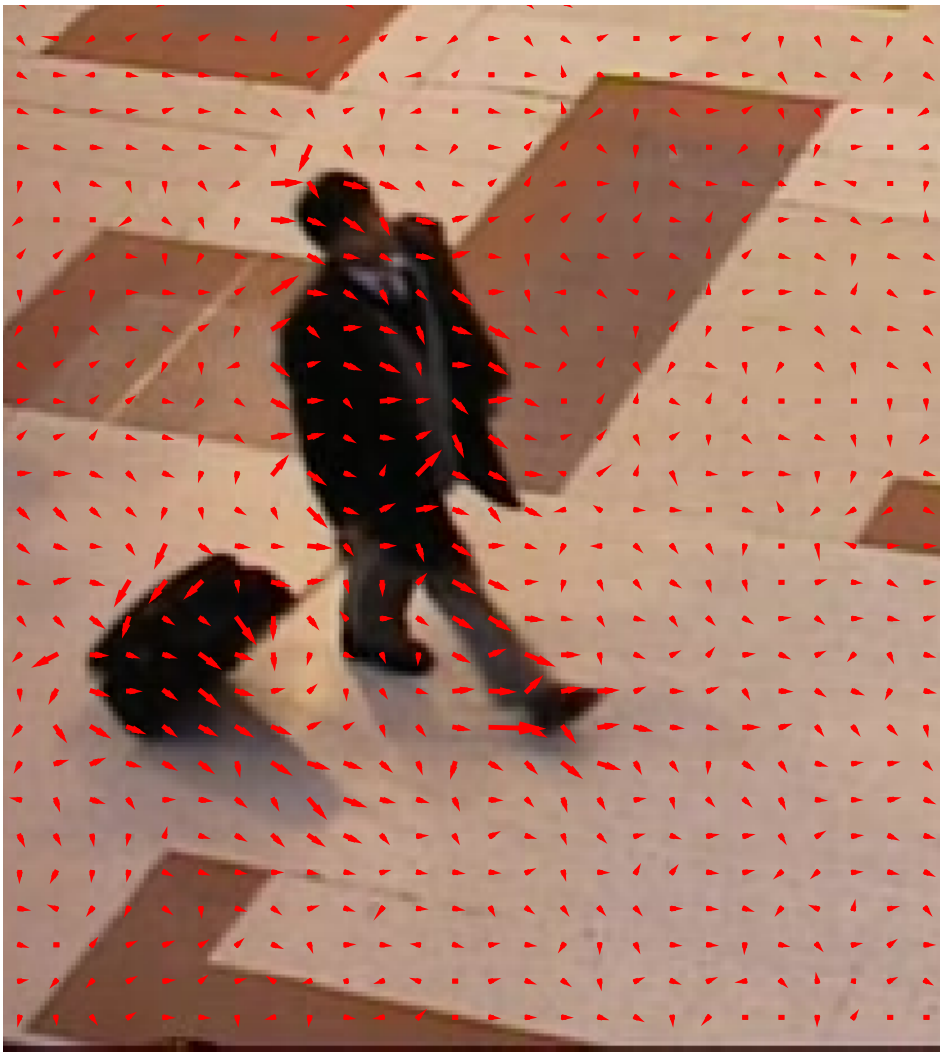

Figure 12: Optical flow for video surveillance sequence - image IF3WS9SS8

Hierarchical Lucas-Kanade algorithm

The main reason for using a hierarchical approach with Lucas-Kanade algorithm is to overcome local disturbances of the optical flow. To determine the overall optical flow of the image, one can use multiple levels of the image, each as a downsample level. As seen with the sphere, if the window size is too small, local *movement*, so influence of noise and structural limits of the algorithm, cause distortions. Those can under circumstances be so dominant, that the optical flow for the entire image is not accurate at all. By downsampling the image and calculating the optical flow for it, the influence of local movement can be lowered and the magnitude of the overall movement in the image can become more dominant. If one downsamples multiple times and performs the calculations, then all of acquired images forming a pyramid can be composited into one image where the influence of the overall movement is more significant and local disturbances in optical flow are decreased.

Each downsampling, if used with factor 2 results in a number of levels of the image pyramid and can be represented by equation (5) and the following excerpt `110` of script code `L04_2_3.m`. The variable uv is the maximum pixel movement received from directional optical flow vectors U and V. This anticipated optical flow change can be also estimated manually based on experience. So for a flow change of `10 px` there should be not more than 4 levels.

$$LV = ceil(log_2(max(U, V))) \tag{5}$$

```
104   if comparison == 0
105     uv = 10; % Manual estimation in pixels
      else
        uv = max([U(:); V(:)]); % Determine largest optical flow vectors from full ▼107
       107▲ resolution Lucas-Kanade results
        clear X, Y, X_q, Y_q, u_q, v_q;
      end
110   Levels = ceil(log2(uv)); % Pyramind levels
```

Implementation of the hierarchical Lucas-Kanade algorithm was realized by modifying a provided solution. The difference between the original and the used version in script `L04_2_3.m` can be seen in the following file comparison `LKH.diff`. Other than a few estetic changes, only the function calls have been replaced.

```
1  --- LK_hierarchical_code/HierarchicalLK.m 2015-11-09 14:00:58.000000000 +0100
   +++ LucasKanadeHierarchical.m 2015-11-24 17:48:02.361601724 +0100
   @@ -48,19 +48,20 @@
    end;
5
    % Elaborazione al livello base (piu' basso)
   -disp('Livello 1');
   +disp('Level 1');
    baseIm1 = pyramid1(1:(size(pyramid1,1)/(2^(numLevels-1))), 1:(size(pyramid1,2) ▼9
       9▲ /(2^(numLevels-1))), numLevels);
10  baseIm2 = pyramid2(1:(size(pyramid2,1)/(2^(numLevels-1))), 1:(size(pyramid2,2) ▼10
       10▲ /(2^(numLevels-1))), numLevels);
    [u,v] = LucasKanade(baseIm1, baseIm2, windowSize);

    % raffino la soluzione
    for r = 1:iterations
15 -  [u, v] = LucasKanadeRefined(u, v, baseIm1, baseIm2, windowSize);
   +  %~ [u, v] = LucasKanadeRefined(u, v, baseIm1, baseIm2, windowSize);
   +  [u, v] = LucasKanade(baseIm1, baseIm2, windowSize);
```

```
 end

 % propago la soluzione ai livelli superiori
 for i = 2:numLevels
-    disp(['Livello ', num2str(i)]);
+    disp(['Level ', num2str(i)]);
     % inizializzo il flusso al livello superiore a partire dal livello inferiore

     % riscalo il flusso rispetto alle dimensioni del livello superiore
@@ -76,7 +77,8 @@

     % raffino ulteriormente la soluzione
     for r = 1:iterations
-        [u, v, cert] = LucasKanadeRefined(u, v, curIm1, curIm2, windowSize);
+        %~ [u, v, cert] = LucasKanadeRefined(u, v, curIm1, curIm2, windowSize);
+        [u, v, cert] = LucasKanade(curIm1, curIm2, windowSize);
     end
 end;

@@ -84,4 +86,4 @@
 %
 if (display==1)
     figure, quiver(reduce((reduce(medfilt2(flipud(u),[5 5]))))), -reduce((reduce( ▼40
   40▲ medfilt2(flipud(v),[5 5]))))), 0), axis equal
-end
\ No newline at end of file
+end
```

The main difference between the hierarchical and conventional Lucas-Kanade approach is the relation to background noise or the significance of movement in the whole image. The image pairs (13÷14) and (15÷16) show clearly that the hierarchical Lucas-Kanade algorithm (LKH) determines background much better than the conventional Lucas-Kanade approach (LK). However one has to take into consideration technical aspects mentioned in the last section "Conclusions and comments" on page 26. The zoomed pair of images showing the man with his bag walking in front of the camera has the same flow with LK and LKH. The slight difference is introduced by using different subsampling step.

Figure 13: Optical flow for video surveillance sequence - image IF3WS9SS8

Figure 14: Optical flow using hierarchical method for the video surveillance sequence - image IF3WS9LV4IT2SS5

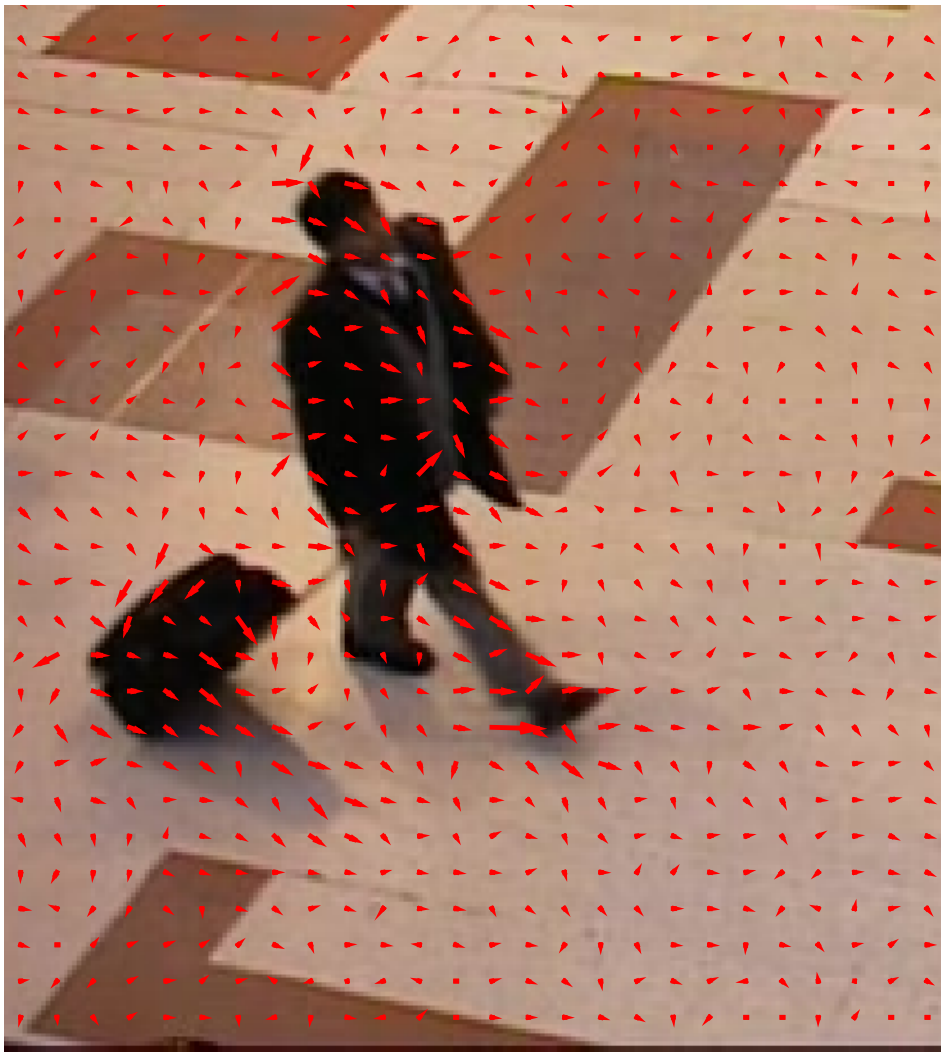

Figure 15: Optical flow for video surveillance sequence - image IF3WS9SS8

Figure 16: Optical flow using hierarchical method for the video surveillance sequence - image IF3WS9LV4IT2SS5

Conclusions and comments

Background subtraction can be used to determine presence of movement depending on the setup and environment. If the environment is controlled or fixed with known constraints and properties, background subtraction could be expanded for other purposes. However it is limited in such way, that it holds no further information on motion other than its presence.

Lucas-Kanade and hierarchical Lucas-Kanade approach are possible algorithms for determining optical flow for an image sequence. Depending on the parameter values of window size, refining iterations and subsampling, the computed overall optical flow might get distorted, influenced by noise, especially coming from lighting fluctuations. Local and global flow magnitude can be influenced by using the hierarchical approach. Based on the calculated or estimated flow change the number of levels of the image pyramid can be determined. Those downsampled images shift the magnitude of optical flow towards global movement and thus reduce what could be considered *local movement noise*.

Influence of parameters like window size, iterations and subsampling step, can change the calculated results of optical flow significantly. Usually a lower window size results in more influence of *local pixel movement*, causing possible distortions in form of random vector directions in the optical flow vector plane. This might be desirable in specific applications, but also can distort the overall information about movement. A larger window size, makes the flow more uniform, but if the used window size gets too large, it can also introduce distortions of another nature, similar to optical distortions of images with objects placed behind lenses. Optimal results have been acquired when the window size was corresponding to the area that had a more uniform flow or when the window size was either the same or magnitude of the size of the object of interest. Subsampling basically increases the size of flow vectors at the price of losing detail information, which can however filter out local pixel movement noise. Another method would include calculating an average or norm of the given subsample window, very similar to the neighbourhood N from Lucas-Kanade method implementation, this would most definitely increase the computation time.

Major limitations of presented algorithms for optical flow include no interpretation of the nature of the movement itself, especially with connection to objects that introduce arbitrary movement, for example a cylinder rotating around its height axis, so performing a horizontal movement of its surface, with a specific pattern suggesting vertical movement. Depending on the used image acquisition equipment, the observer object and environment, certain artefacts like the described *aura effect* can occur. Further distortions can be introduced by other objects in the environment, like reflections or partial coverage of moving objects of interest. As usual to overcome these problems a combination of techniques could be used and depending on the application further modification might be necessary.

Power consumption and calculation time are practical aspects that, especially when using Lucas-Kanade algorithm, can be of high significance. The background subtraction can be reduced to two simple operations of subtraction and binarizing. Lucas-Kanade method however, especially when used in the hierarchical approach with multiple refining iterations, can be not only computation power intensive, but also time consuming. Depending on the parameters used and further modifications implemented, it might not be suitable for mobile robot applications that would in most cases require low latency between image acquisition and determining significant information, used for decision making for example.

Optimization of presented algorithms is basically a numerical computation problem, which can be addressed using conventional numerical optimization techniques. One of the computing intensive tasks in Lucas-Kanade algorithm is the pseudo inversion. An attempt has been made to reduce the computation load by using the optional tolerance argument of `pinv()` function. This has not been extensively tested however, so no further comments can be made here. Of course using the Matlab calculation environment is part of the problem with computation time, since this environment is very slow. An alternative would be to at least create a C/C++ equivalent to Lucas-Kanade implementation and then call it from the Matlab or compatible environment.

Technical aspects that should be taken into account include the usage of PDF format for storing images,

in this case vector fields. Since PDF is capable of storing lossless information about primitives like lines and vector, it is very much capable of saving information that would be lost using a lossy bitmap format like png or bmp. This results in *dots* or very small vectors that might be confusing to the user. Therefore it should be taken into consideration to either use lossy formats for further analysis using image processing or to negate those small vectors through other means. If the `imageshow` options is set to 1, the generated figures will not include those *dots*, they are however there. The PDF format seems to force its own marker size on vectors.

The video rendering function would increase the capability of showing the optical flow information even further, at least for humans. It might make it more readable and allow for an easier interpretation of the overall movement in the image sequence.