Advanced Vision Systems

AGH UST International Course

Tomasz Kryjak, PhD, Mateusz Wąsala, MSc

Tomasz Kryjak
Mateusz Wąsala

EMBEDDED VISION SYSTEMS GROUP
VISION SYSTEMS LABORATORY
WEAIIIB, AGH UNIVERSITY OF KRAKOW

Third edition, revised and corrected, Krakow, February 2024

Contents

Laboratory 1

1. Vision algorithms in Python 3.X – introduction

The exercise will present/mention basic information related to the use of the Python language to perform image and video processing operations and image analysis.

1.1 Programming software

Before starting the exercises, **please create** your own working directory in the place given by the tutor. The name of the directory should be associated with your name – the recommended form is `LastName_FirstName` without no special characters.

The Python programming environment – Visual Studio Code (VSCode) – can be used to complete the exercises. Visual Studio Code is a free, lightweight, intuitive and easy to use code editor. It is a universal software for any programming language. Configuration of the editor for exercises comes down only to choosing the appropriate interpreter, or more precisely the appropriate virtual environment in the Python language.

1.2 Python modules used in image processing

Python does not have a native array type that allows to perform elementwise operations betweens two containers (in a convenient and efficient way). For operations on 2D arrays (i.e. also on images) it is common to use the external module NumPy. This is the basis for all further mentioned modules supporting image processing and display. There are at least 3 main packages supporting image processing:

- a pair of modules: the *ndimage* module from the *SciPy* library – contains functions for processing images (including multidimensional images), and the *pyplot* module from the *Matplotlib* library – for displaying images and plots,
- *PILLOW* module (part of the non-expanded PIL module)
- *cv2* module is an frontend to the popular *OpenCV* library.

In our course we will rely on *OpenCV*, although *Matplotlib* and occasionally *ndimage* will also be used – mainly in situations where the functions in *OpenCV* do not have adequate functionality or are less convenient to use (e.g. displaying images).

1.3 Input/output operations

1.3.1 Reading, displaying and saving an image using OpenCV

Exercise 1.1 Perform a task in which you practice handling files using OpenCV.

1. From the course page, download the image *mandril.jpg* and place it in your own working directory.
2. Run the program – *spyder*, *pycharm*, *vscode* – from the console or using the icon. Create a new file and save it in your own working directory.
3. In the file, load the module `cv2` (`import cv2`). Test loading and displaying images using this library – use the function `imread` for loading, example usage: `I = cv2.imread('mandril.jpg')`
4. Displaying a picture requires at least two functions – `imshow()` creates a window and starts the display, and `waitKey()` shows the picture for a given period of time (argument 0 means wait for a key to be pressed). When the program is finished, the window is automatically closed, but on condition that it is terminated by pressing any key.

   Attention! Closing the window via the button on the window bar will loop the program and you will have to abort it:
   - VSCode – stop the program in the terminal by using the keyboard shortcut 'CTRL+Z' or closing the terminal.

   R For especially high-resolution images, it is useful to use the function `namedWindow()` with the same name as in the function *imshow()*. We declare it before the *imshow()* function.

5. An unnecessary window can be closed using `destroyWindow` with the appropriate parameter, or all open windows can be closed using `destroyAllWindows`.

```
I = cv2.imread('mandril.jpg')
cv2.imshow("Mandril",I)       # display
cv2.waitKey(0)                 # wait for key
cv2.destroyAllWindows()        # close all windows
```

   R The window does not necessarily need to be displayed on top. It is good practice to use the *cv2.destroyAllWindows()* function.

6. Saving to a file is performed by the function *imwrite* (please note the format change from *jpg* to *png*):

```
cv2.imwrite("m.png",I) # zapis obrazu do pliku
```

7. The image displayed is in colour, which means that it consists of 3 channels. In other words, it is represented as a 3D array. There is often a need to view the value of this array. This can be done in one-dimensional form, but is better done with a two-dimensional array.
   - VSCode – it is necessary to run the program in *Debug* mode and set the breakpoint. Under *Run and Debug* and in the *Variables* section you will find the variable *I*. To view the values of the *I* matrix, select *View Value in Data Viewer* under the right mouse button. In the *Watch* section it is possible to refer to a specific element of the *I* array or perform appropriate operations on it.

   In all cases a 2D array is displayed which is a slice of the 3D array, however by default

this slice is in the wrong axis – a single line in 3 components is displayed. To get the whole image displayed for one component, change the axis.
- VSCode – under *SLICING* section, select the appropriate axis – set to 2 or select the appropriate indexing and press the *Apply* button.

Other components are available:
- VSCode – *Index* field.

8. Access to image parameters can be useful at work:

```
print(I.shape)  # dimensions /rows, columns, depth/
print(I.size)   # number of bytes
print(I.dtype)  # data type
```

The print function is a display to the console.

■

1.3.2 Loading, displaying and saving an image using the *Matplotlib* module

An alternative way to implement input/output operations is to use the *Matplotlib* library. Then the handling of load/display etc. is similar to that known from the Matlab package. Documentation of the library is available online Matplotlib.

Exercise 1.2 Do a task in which you practice handling files using the Matplotlib library. To keep some order in your code, create a new source code file.

1. Load *pyplot* module.

```
import matplotlib.pyplot as plt
```

(as allows you to shorten the name to be used in the project)

2. Load the same image *mandril.jpg*

```
I = plt.imread('mandril.jpg')
```

3. The display is implemented very similarly to the Matlab package:

```
plt.figure(1)         # create figure
plt.imshow(I)         # add image
plt.title('Mandril') # add title
plt.axis('off')       # disable display of the coordinate system
plt.show()            # display
```

4. Image storage:

```
plt.imsave('mandril.png',I)
```

5. During the lab, it can also be useful to display certain elements in the image – such as points or frames.
6. Adding the display of points:

```
x = [ 100, 150, 200, 250]
y = [ 50, 100, 150, 200]
plt.plot(x,y,'r.',markersize=10)
```

Attention! Commas are necessary (unlike in Matlab) when setting array values. In the plot command, the syntax is similar to Matlab – 'r' – colour, '.' – dot, and the size of the marker.

Full list of possibilities in the documentation.

7. Adding rectangle display – drawing shapes i.e. rectangles, ellipses, circles etc. is available in *matplotlib.patches*. Please note the comments in the code below.

```
from matplotlib.patches import Rectangle # add at the top of the file

fig,ax = plt.subplots(1) # instead of plt.figure(1)

rect = Rectangle((50,50),50,100,fill=False, ec='r'); # ec - edge colour
ax.add_patch(rect) # display
plt.show()
```

1.4 Colour space conversion

1.4.1 OpenCV

The function *cvtColor* is used to convert the colour space.

```
IG = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)
IHSV = cv2.cvtColor(I, cv2.COLOR_BGR2HSV)
```

R Please note that in OpenCV the reading is in **BGR** order, not RGB. This can be important when manually manipulating pixels. For a full list of available conversions with the relevant formulas in OpenCV documentation

Exercise 1.3 Convert the colour space of the given image.

1. Convert the *mandril.jpg* image to grayscale and HSV space. Display the result.
2. Display the H, S, V components of the image after conversion.

R Please note that in contrast to e.g. the Matlab package, here the indexing is from 0.

Useful syntax:

```
IH = IHSV[:,:,0]
IS = IHSV[:,:,1]
IV = IHSV[:,:,2]
```

1.4.2 Matplotlib

Here the choice of available conversions is quite limited. Therefore, we need to use the formulas for colour space conversion.

1. RGB to greyscale.

$$G = 0.299 \cdot R + 0.587 \cdot G + 0.144 \cdot B \tag{1.1}$$

The formula can be represented as a function:

```
def rgb2gray(I):
  return 0.299*I[:,:,0] + 0.587*I[:,:,1] + 0.114*I[:,:,2]
```

R The colour map must be set when displaying. Otherwise the image will be displayed in the default, which is not greyscale: `plt.gray()`

2. RGB do HSV.

```
import matplotlib # add at the top of the file
I_HSV = matplotlib.colors.rgb_to_hsv(I)
```

1.5 Scaling, rescaling with OpenCV

Exercise 1.4 Scale the image with *mandrill*. Use the *resize* function to scale. ■

Example of use:

```
height, width =I.shape[:2] # retrieving elements 1 and 2, i.e. the corresponding
    height and width
scale = 1.75   # scale factor
Ix2 = cv2.resize(I,(int(scale*height),int(scale*width)))
cv2.imshow("Big Mandrill",Ix2)
```

1.6 Arithmetic operations: addition, subtraction, multiplication, difference module

The images are matrices, so the arithmetic operations are quite simple – just like in the Matlab package. Of course, remember to convert to the appropriate data type. Usually double format will be a good choice.

Exercise 1.5 Perform arithmetic operations on the image *lena*.

1. Download the image *lena* from the course page, then load it using a function from OpenCV – add this code snippet to the file that contains the image loader *mandril*. Perform the conversion to grayscale. Add the matrices containing Mandril and Leny in grayscale. Display the result.
2. Similarly perform the subtraction and multiplication of images.
3. Implement a linear combination of images.
4. An important operation is the module from image difference. It can be done manually – conversion to the appropriate type, subtraction, module (`abs`), conversion to *uint8*. An alternative is to use the function *absdiff* from OpenCV. Please calculate the modulus from the difference of the mandril and Lena greyscale image.

■

R To display the image correctly, the data type must be changed to uint type. Appropriate conversion:

```
import numpy as np
cv2.imshow("C",np.uint8(C))
```

1.7 Histogram calculation

Calculating the histogram can be done using the function textcalcHist. But before we get to that, let's remind ourselves of the basic control structures in Python – functions and subroutines. Please complete the following function yourself, which calculates a histogram from an image with 256 grayscale values:

```
def hist(img):
  h=np.zeros((256,1), np.float32) # creates and zeros single-column arrays
  height, width =img.shape[:2] # shape - we take the first 2 values
    for y in range(height):
...
return h
```

Attention! In Python, indentation is important as it determines the block of a function, or loop!

The histogram can be displayed using the function `plt.hist` or `plt.plot` from the *Matplotlib* library.

The function `calcHist` can count the histogram of several images (or components). However, the form of the formula is most commonly used:

```
hist = cv2.calcHist([IG],[0],None,[256],[0,256])
# [IG] -- input image
# [0] -- for greyscale images there is only one channel
# None -- mask (you can count the histogram of a selected part of the image)
# [256] -- number of histogram bins
# [0 256 ] -- the range over which the histogram is calculated
```

Please check that the histograms obtained by both methods are the same.

1.8 Histogram equalisation

Histogram equalisation is a popular and important pre-processing operation.

1. Classical histogram equalization is a global method, it is performed on the whole image. There is a ready-to-use function for this type of equalization in the OpenCV library:

   ```
   IGE = cv2.equalizeHist(IG)
   ```

2. CLAHE Histogram Equalization (*Contrast Limited Adaptive Histogram Equalization*) – is an adaptive method that improves lighting conditions in an image by using histogram equalization for individual parts of the image rather than the whole image. The method works as follows:
   - division of the image into disjoint (square) blocks,
   - calculation of the histogram in blocks,
   - performing a histogram equalisation, where the maximum height of the histogram is limited and the excess is redistributed to adjacent intervals,
   - interpolation of pixel values based on calculated histograms for given blocks (four neighbouring quadrant centres are taken into account).

   For details, see Wiki and tutorial.

```
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
# clipLimit - maximum height of the histogram bar - values above are distributed
    among neighbours
# tileGridSize - size of a single image block (local method, operates on separate
    image blocks)
```

```
I_CLAHE = clahe.apply(IG)
```

Exercise 1.6 Run and compare both equalisation methods. ■

1.9 Filtration

Filtering is a very important group of operations on images. As an exercise, please run:

- Gaussian filtration (`GaussianBlur`)
- Sobel filtration (`Sobel`)
- Laplasian filter (`Laplacian`)
- Median filtration (`medianBlur`)

R OpenCV documentation – filtration.

Please also note the other functions available:

- bilateral filtration,
- Gabor filters,
- morphological operations.

Exercise 1.7 Run and compare both equalisation methods. ■

II Laboratory 2

2. Detection of foreground moving objects

What are foreground objects ?

Those that are of interest to us (for the application under consideration). Typically: people, animals, cars (or other vehicles), luggage (potential bombs). So the definition is closely related to the target application.

Is foreground object segmentation a moving object segmentation ?

No. Firstly, an object that has stopped may still be of interest to us (e.g. a man standing in front of a pedestrian crossing). Second, there are a whole range of moving scene elements that are not of interest to us. Examples include flowing water, a fountain, moving trees and bushes, etc. Note also that usually moving objects are taken into account during image processing. So foreground object detection can and should be supported by moving object detection.

The simplest method to detect moving objects is to perform subtraction of consecutive (adjacent) frames. In this exercise we will realize a simple subtraction of two frames, combined with binarization, connected-component labeling and analysis of the resulting objects. Finally, we will try to consider the subtraction result as a result of foreground object segmentation and check the quality of this segmentation.

2.1 Image sequence loading

Exercise 2.1 Image sequence loading

1. Download the appropriate test sequence from the *UPeL* platform. In the exercise, we will focus on the sequence *pedestrians*. The sequences used come from a dataset from the changedetection.net page. The dataset contains labelled sequences, i.e. each image frame has an object mask - a reference mask (*ground truth*). On these, each pixel has been assigned to one of five categories: background (0), shadow (50), beyond the area of

interest (85), unknown motion (170) and foreground objects (255) – the corresponding greyscale levels are given in brackets.
For the exercise, we will only be interested in the split between foreground objects and the other categories. In addition, the folder contains a region of interest (ROI) mask and a text file with the time interval for which the results should be analysed (temporalROI.txt) – details follow later in the exercise.

2. The following example code allows the sequence to be loaded:

```
for i in range(300,1100):
  I = cv2.imread('input/in%06d.jpg' % i)
  cv2.imshow("I",I)
  cv2.waitKey(10)
```

R We assume that the sequence was extracted in the same folder as the source file (otherwise you need to add the appropriate path).

You must use the `waitKey` function. Otherwise the displayed image will not be refreshed.

3. Add to the loop the option to analyse every i-th frame – the *range* function used may have as a third parameter: *step*. Experiment with its value now (when displaying the video) and later (when detecting moving objects).

■

2.2 Frame subtraction and Binarization

In order to detect the moving element we subtract two consecutive frames. It should be noted that we are concerned exactly with calculating the modulus from the difference.

R To avoid problems with unsigned number subtraction (*uint8*), perform a conversion to type *int* – `IG = IG.astype('int')`.

For simplicity of consideration, it is better to convert to greyscale in advance. To begin with, you need to handle the first iteration in some way. For example – read the first frame before the loop and consider it as the previous one. Later, at the end of the loop, add the assignment previous frame = current frame. Test display the subtraction results – you should see mostly edges.

Binarisation can be performed using the following syntax:

```
B = 1*(D > 10)
```

R In that case, `1*` means converting from a logical type to a numeric type. The correct display is a separate issue – you need to change the range (multiply by 255) and convert to *uint8* (you may need to import the `numpy` library).

The threshold should be chosen so that objects are relatively visible. Please pay attention to compression artifacts.

An alternative is to use a function built into OpenCV:

```
(T, thresh) = cv2.threshold(D,10,255,cv2.THRESH_BINARY)
# D -- input array
# 10 -- threshold value
```

```
# 255 -- maximum value to use with the THRESH_BINARY and THRESH_BINARY_INV
    thresholding types
# cv2.THRESH_BINARY -- thresholding type
# T - our threshold value
# thresh - output image
```

R The first argument of the returned values by the `cv2.threshold` function is the binarization threshold (this is useful when using automatic threshold determination, e.g. with the Otsu or triangle method). See the OpenCV documentation for details.

2.3 Morphological operations

Exercise 2.2 The resulting image is quite noisy. Please perform filtering using erosion and dilation (`erode` and `dilate` from OpenCV). ■

R The aim of this stage is to obtain a maximally visible silhouette, with minimum distortions. To improve the effect it is worth adding a median filtering step (before morphology) and possibly correcting the binarisation threshold.

2.4 Labeling and simple analysis

In the next step, we will perform a filtering of the obtained result. We will use indexing (assigning labels to groups of connected pixels) and calculating the parameters of these groups. The function connectedComponentsWithStats is used for this. Function call:

```
retval, labels, stats, centroids =  cv2.connectedComponentsWithStats(B)
# retval --  total number of unique labels
# labels -- destination labeled image
# stats -- statistics output for each label, including the background label.
# centroids -- centroid output for each label, including the background label.
```

R When displaying the *labels* image, you need to set the format properly and add scaling. You can use information about the number of objects found for scaling.

```
cv2.imshow("Labels", np.uint8(labels / retval * 255))
```

Then display the surrounding rectangle, field and index for the largest object. Below is a sample solution to the task. Please run it and possibly try to optimise it.

```
I_VIS = I # copy of the input image

if (stats.shape[0] > 1):    # are there any objects

  tab = stats[1:,4] # 4 columns without first element
  pi = np.argmax( tab )# finding the index of the largest item
  pi = pi + 1 # increment because we want the index in stats, not in tab
  # drawing a bbox
  cv2.rectangle(I_VIS,(stats[pi,0],stats[pi,1]),(stats[pi,0]+stats[pi,2],stats[pi
      ,1]+stats[pi,3]),(255,0,0),2)
  # print information about the field and the number of the largest element
  cv2.putText(I_VIS,"%f" % stats[pi,4],(stats[pi,0],stats[pi,1]),cv2.
      FONT_HERSHEY_SIMPLEX,0.5,(255,0,0))
```

```
cv2.putText(I_VIS,"%d" %pi,(np.int(centroids[pi,0]),np.int(centroids[pi,1])),cv2.
    FONT_HERSHEY_SIMPLEX,1,(255,0,0))
```

Comments on the sample solution:

- `stats.shape[0]` is the number of objects. Since the function also counts the object with index 0 (i.e. background), in the conditional function check if there are objects the condition is > 1.
- the next two lines are the calculation of the maximum index from column number 4 (field).
- the resulting index should be incremented, as we have omitted element 0 (background) from the analysis.
- We use the `rectangle` function from OpenCV to draw a surrounding rectangle in the image. The syntax `''%f'' % stats[pi,4]` allows us to output the value in the appropriate format (f - float). The next parameters are the coordinates of the two opposite vertices of the rectangle. Then the colour in format (B,G,R) and finally the line thickness. See the function documentation for details.
- The function `putText` is used to output text on the image. The coordinates of the lower left vertex are given to determine the position of the text box. The next parameters are the font (full list in the documentation), size and colour.
- R `I_VIS` is the input image before conversion to greyscale.

Exercise 2.3 Use the function `cv2.connectedComponentsWithStats` to label and calculate the parameters of the resulting objects. Display the surrounding rectangle, field and index for the largest object. Based on the tips and examples in the text above, try to optimize the solution. ■

2.5 Evaluation of foreground object detection results

In order to evaluate the foreground object detection algorithm, and preferably in a relatively objective manner, the results returned by it, i.e. the object mask, should be compared with the reference mask (*groundtruth*). The comparison takes place at the level of individual pixels. If shadows are excluded (which is what we assumed at the beginning), four situations are possible:

- TP – *true positive* – a pixel belonging to a foreground object is detected as a pixel belonging to a foreground object,
- TN – *true negative* – a pixel belonging to the background is detected as a pixel belonging to the background,
- FP – *false positive* – a pixel belonging to the background is detected as a pixel belonging to a foreground object,
- FN – *false negative* – a pixel belonging to an object is detected as a pixel belonging to the background.

A series of metrics can be counted from the listed coefficients. We will use three: *precision* - P, *recall* - R and $F1$ score. These are defined as follows:

$$P = \frac{TP}{TP+FP} \tag{2.1}$$

$$R = \frac{TP}{TP+FN} \tag{2.2}$$

$$F1 = \frac{2PR}{P+R} \tag{2.3}$$

The F1 score is in the range $[0;1]$, with the higher its value, the better it is.

Exercise 2.4 Implement the computation of the metrics P, R and $F1$.

1. In the first step, define the global counters TP, TN, FP, FN, and calculate the desired values of P, R and $F1$ when the loop finished.
2. Add to the loop the loading of a reference mask – analogous to an image. It is available in the *groundtruth* folder.
3. The next step is to compare the object mask and the reference mask. The simplest solution is `for` loop over the whole image, in each iteration we check the pixel similarity. Of course, this is not a very efficient approach. You can try using Python's capabilities in implementing matrix operations. Create appropriate logical conditions, for example:

```
TP_M = np.logical_and((B == 255),(GTB == 255)) # logical product of the
    matrix elements
TP_S = np.sum(TP_M)          # sum of the elements in the matrix
TP = TP + TP_S               # update of the global indicator
```

However, we only perform the calculation if a valid reference map is available. To do this, check the dependence of the frame counter and the value from the `temporalROI.txt` file – it must be within the range described there. Example code that loads the range (by the way, please note how text files are handled):

```
f = open('temporalROI.txt','r')        # open file
line = f.readline()                    # read line
roi_start, roi_end = line.split()      # split line
roi_start = int(roi_start)             # conversion to int
roi_end = int(roi_end)                 # conversion to int
```

4. Run the calculations for the consecutive frame subtraction method. Note the value of $F1$.
5. Perform calculations for the other two sequences – *highway* and *office*.

■

Information on the following classes.

1. In further exercises we will learn more algorithms and functionalities of the Python language. However, we do not put special emphasis on learning the Python language, but on using in practice the successive algorithms appearing in the lectures. The subject Advanced Vision Systems is not a Python course!
2. The solutions presented should always be considered as examples. The problem can certainly be solved differently, and sometimes better.
3. In the next exercises the level of detail of the solution description will decrease. We therefore encourage you to actively use the documentation for Python, OpenCV and materials/tutorials found on the Internet :).

2.6 Example results of the algorithm for foreground moving object detection

In order to better verify the different steps of the proposed method for foreground moving object detection, an example result for an image with index 350 will be presented.

Figure 2.1: An example of the result of each stage of the algorithm. From left: input image with bounding box, image after binarisation, image after median filtering and morphological operations (erode, dilate), image representing labels after labelling, reference image – ground truth.

Laboratory 3

3. Foreground object segmentation

3.1 Main objectives

- You will become familiar with the issue of foreground object segmentation and the problems associated with it.
- You will be able to implement simple background modelling algorithms based on a frame buffer – analyzing their pros and cons.
- You will be able to implement the running average and approximate median method.
- You will become familiar with the methods available in *OpenCV* – Gaussian Mixture Model (also called Mixture of Gaussians) and a method based on the KNN (K-Nearest Neighbours) algorithm,
- You will become familiar with neural network architecture and an example application.

3.2 Theory of foreground object segmentation

What is foreground object segmentation?

Segmentation is the extraction of a certain group of objects present in an image. It should be noted that it is not necessarily the same as classification, where the output is the assignment of an object to a specific class: e.g. a car (or even a type), a pedestrian, etc. The definition of a foreground object is not very strict – they are all objects relevant to the application. For example for video surveillance: people, cars and maybe animals.

What is the background model?

In the simplest case, it is an image of an empty scene, i.e. without objects of interest.

In more advanced algorithms, the background model has no explicit form (it is not an image) and cannot simply be displayed (examples of such methods: GMM, ViBE, PBAS).

Initialisation of the background model

When the algorithm is started (also restarted), it is necessary to perform an initialization of the background model, i.e., to determine the values of all parameters (variables) that represent this model. In the simplest case, the first frame of the analysed sequence (the first N frames in the case of methods with buffer) is taken as the initial background model. This approach works well under the assumption that there are no foreground objects on that particular frame (sequence of frames). Otherwise, the initial model will contain errors that will be more or less difficult to eliminate in further operation - it depends on the specific algorithm. In the literature this issue is referred to as *bootstrap*.

More advanced initialisation methods rely on temporal and even spatial pixel analysis for a certain initial sequence. The assumption here is that the background is visible most of the time and is more spatially consistent than the objects.

Background modelling, background generation

The question is whether it is not enough to take an image of an empty scene, save it and use it throughout the algorithm? In the general case no, but let's start with a special case. If our system is monitoring a room where the lighting is strictly controlled and any previously unforeseen and approved changes should not take place (e.g. a forbidden zone inside a power plant, a bank vault, etc.), then the simplest approach with a static background will work.

Problems arise when the lighting of a scene changes due to a change in the time of day or when additional lights are switched on, etc. Then the actual background will change and our static background model will not be able to adapt to this change. The second, more difficult case is a change in the position of scene elements: e.g. someone moves a bench/chair/flower. This change should not be detected, or at least after a fairly short time it should be taken into account in the background model. Otherwise it will cause the generation of false detections (known as *ghost*) and consequently the generation of false alarms.

The conclusion is as follows. A good background model should be characterised by its ability to adapt to changes in the scene. The phenomenon of model updating is referred to in the literature as background generation or background modelling.

3.3 Methods based on frame buffer

The exercise will present two methods: the buffer mean and the buffer median. The buffer size will be assumed to be $N = 60$ samples. In each iteration of the algorithm it is necessary to remove the last frame from the buffer, add the current one (the buffer works on the principle of FIFO queue) and calculate the mean or median from the buffer.

1. First, declare a buffer of size $N \times YY \times XX$(`np.zeros` function), where YY and XX are the height and width of the frame from the *pedestrians* sequence.

```
BUF = np.zeros((YY,XX,N),np.uint8)
```

   You should also initialize the counter for the buffer (let it be called e.g. `iN`) with the value 0. The parameter `iN` is like a pointer that points to a place in the buffer from which the last frame should be removed and the current frame assigned.
2. The buffer handling should be as follows. In the loop under the variable `iN`, store the current frame in greyscale.

```
BUF[:,:,iN] = IG;
```

Then increment the counter `iN` and check if it does not reach the buffer size (N). If it does, it needs to be set to 0.

3. The computation of the mean or median is implemented by the functions `mean` or `median` from the *numpy* library. As a parameter of the function you specify the dimension for which the value is to be calculated (remember to index from zero). Then convert the result to `uint8`.
4. Finally, we perform background subtraction, i.e. we subtract the model from the current scene and binarise the result – analogous to the difference of adjacent frames. We also use median filtering of object masks and/or morphological operations.
5. Using the code created in the previous exercise, compare the performance of the method with the mean and median. Note the values of the $F1$ indicator for both cases.

R Calculating the median can take a long time.

Consider why the median works better. Check the performance on other sequences – in particular for the sequence *office*. Observe the phenomenon of blurring and silhouette blending into the background and the formation of *ghost*.

Exercise 3.1 Implement the algorithms (median and mean) based on the above description. ■

3.4 Approximation of mean and median (so-called sigma-delta)

Using a sample buffer is resource-intensive. There are methods for faster calculation of mean, but in case of median such methods do not exist. Therefore, methods that do not require a buffer are more likely to be used. In the case of the mean approximation the relation is used:

$$BG_N = \alpha I_N + (1-\alpha) BG_{N-1} \tag{3.1}$$

where: I_N – current frame, BG_N – background model, α – weight parameter, typically 0.01 – 0.05, although the value depends on the specific application.

The approximation of the median is obtained using the relation:

$$\begin{aligned} &if \;\; BG_{N-1} < I_N \;\; then \;\; BG_N = BG_{N-1} + 1 \\ &if \;\; BG_{N-1} > I_N \;\; then \;\; BG_N = BG_{N-1} - 1 \\ &otherwise \;\; BG_N = BG_{N-1} \end{aligned} \tag{3.2}$$

Exercise 3.2 Implement both methods.

1. Take the first frame of the sequence as the first background model. Note the value of the $F1$ indicator for the two methods (set the *alpha* parameter to 0.01). Observe the running time.
2. Experiment with the value of the *alpha* parameter. See how the parameter value affects the background model.

■

R For the running average method it is very important that the background model is floating point (*float64*).
For the pattern 3.1 the avoidance of using a loop over the whole image is obvious. For the dependency 3.2 this is also possible. It is worthwhile to use the documentation of *numpy*. Additionally, please note that in Python it is possible to perform arithmetic operations on Boolean values – False ==0 while True ==1.

3.5 Updating policy

So far we have followed a liberal update policy – we have updated everything. We will now try to use a conservative approach – we update only those pixels that have been classified as **background**.

Exercise 3.3 1. For the selected method, implement a conservative update approach. Check its operation.

R Simply remember the previous object mask and use it appropriately in the update procedure.

2. Note the value of the F1 indicator and the background model. Are there any errors in it?

■

3.6 OpenCV – GMM/MOG

One of the most popular foreground object segmentation methods is available in the OpenCV library: Gaussian Mixture Models (GMM) or Mixture of Gaussians (MoG) - both names appear in parallel in the literature. In a most brief way: a scene is modelled by several Gaussian distributions (mean of brightness (colour) and standard deviation). Each distribution is also described by a weight parameter that captures how often it was used (observed in the scene – probability of occurrence). The distributions with the largest weight parameters represent the background. Segmentation is based on calculating the distance between the current pixel and each of the distributions. Then if the pixel is similar to the distribution considered as background, it is classified as background, otherwise it is considered as object. The background model is updated in a way similar to the equation 3.1. If you are interested, we refer you to the literature, e.g. article.

Exercise 3.4 This method is an example of a multivariant algorithm – the background model has several possible representations (variants).

1. Open the documentation for OpenCV. Go to *Video Analysis->Motion Analysis*. We are interested in the class `BackgroundSubtractorMOG2`. To create an object of this class in Python3 use `createBackgroundSubtractorMOG2`. Using this object in a loop for each image we use its method `apply`.
2. To begin, we will run the code with the default values.
3. Please experiment with the parameters: *history*, *varThreshold* and disable shadow detection. It is also possible to manually set the learning rate in the `apply()` method – *learningRate*

R There is a GMM version available in the OpenCV package which is slightly different from the original – including a shadow detection module and a dynamically changing number of Gaussian distributions. Article references are available in the OpenCV documentation.

4. Determine the F1 parameter (for the method without shadow detection). Observe how the method works. Note that the background model cannot be displayed.

■

3.7 OpenCV – KNN

The second method available in OpenCV is KNN algorithm (BackgroundSubtractorKNN).

Exercise 3.5 Please run the KNN method and evaluate it. ■

3.8 Example results of the algorithm for foreground object segmentation

In order to better verify the individual steps of the proposed methods for foreground object segmentation, example results for selected image frames will be presented.

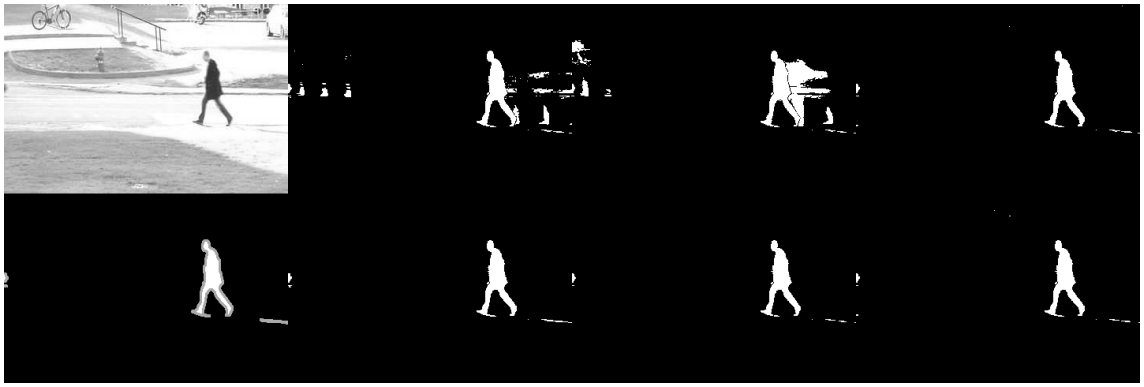


Figure 3.1: Example of the result of the different versions of the algorithm for foreground object segmentation for the *pedestrians* dataset. The frame index is 600. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the second row uses the median. Sequentially from the second column the result of the algorithm is presented: liberal update policy, then the function approximation with a liberal update policy and the function approximation with a conservative update policy.

Figure 3.2: Example of the result of the different versions of the algorithm for foreground object segmentation for the *highway* dataset. The frame index is 1200. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the second row uses the median. Sequentially from the second column the result of the algorithm is presented: liberal update policy, then the function approximation with a liberal update policy and the function approximation with a conservative update policy.

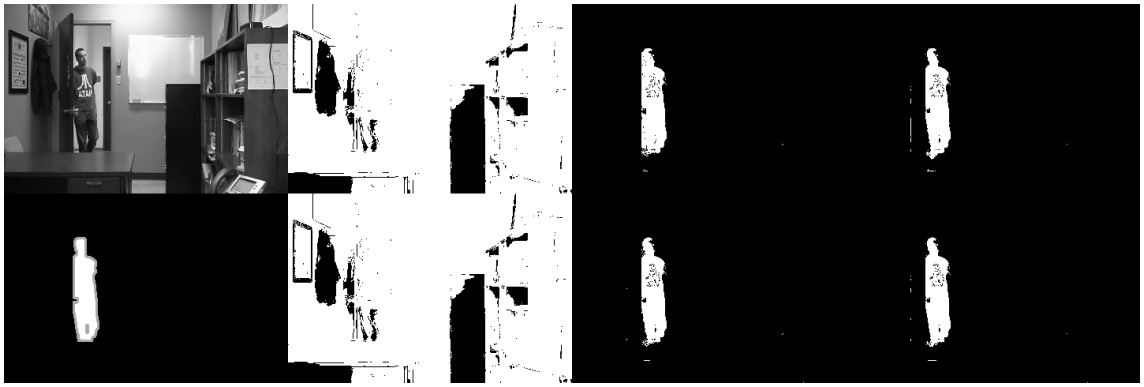


Figure 3.3: Example of the result of the different versions of the algorithm for foreground object segmentation for the *office* dataset. The frame index is 600. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the second row uses the median. Sequentially from the second column the result of the algorithm is presented: liberal update policy, then the function approximation with a liberal update policy and the function approximation with a conservative update policy.

3.9 Using a neural network to segment foreground objects.

Neural network architecture-based methods can also be used to segment foreground objects. An example of this is the work [1]. The proposed approach is to use a convolutional neural network called BSUV-Net (Background Substraction Unseen Videos Net). The architecture is based on a U-net structure with residual connections, which is divided into an encoder and a decoder. The input to network consists of the current frame and two background frames captured at different time scales along with their semantic segmentation maps. The output is a mask containing only foreground objects.

You can find details about this neural network in the article https://arxiv.org/abs/1907.11371. The architecture with sample data and trained network models is made available on Github https://github.com/ozantezcan/BSUV-Net-inference.

Exercise 3.6 Run the BSUV-Net convolutional neural network.

- Download all the files for the neural network from the UPeL platform.
- The `pedestrians` database will be used, but appropriately scaled to the size of the network input and converted to video sequences,
- Open `infer_config_manualBG.py` file and modify the paths to:
  - in class `SemanticSegmentation` specify the absolute path to the segmentation folder – variable `root_path`,
  - in class `BSUVNet` specify the path to the network model `BSUV-Net-2.0.mdl` in the `trained_models` folder – `model_path`,
  - in class `BSUVNet` specify the path to an image that contains only the background – the first image in the set `pedestrians` – variable `empty_bg_path`
- In the `inference.py` file specify the path to the network input, i.e. the `pedestrians` video sequence – variable `inp_path` and a path to where the network output is to be stored (path with the file name and file extension) – variable `out_path`.

■

3.10 Additional exercises

3.10.1 Initialization of the background model in the presence of foreground objects on the scene

Exercise 3.7 **Additional exercise 1**

If we take a method in which we have used a conservative update approach and start the sequence analysis not with frame 1 but with 1000 then we will see in practice the disadvantages of assuming that the first frame in the sequence is the initial background model (*ghosty* etc.). Propose a solution to the problem. In your application, demonstrate the advantage of your own approach over initialization based on the first frame.

Figure 3.4 shows an example of a solution to the problem.

■

R It seems a good idea to buffer, for example, 30 or more frames and perform a frequency analysis of the occurrence of each shade of brightness in the sequence – independently for each pixel. The background should be considered to be that which was observed most frequently. Other effective ideas are also welcome.

[1] Tezcan, Ozan and Ishwar, Prakash and Konrad, Janusz; BSUV-Net: A Fully-Convolutional Neural Network for Background Subtraction of Unseen Videos, https://arxiv.org/abs/1907.11371

A different sequence can be used in the task.

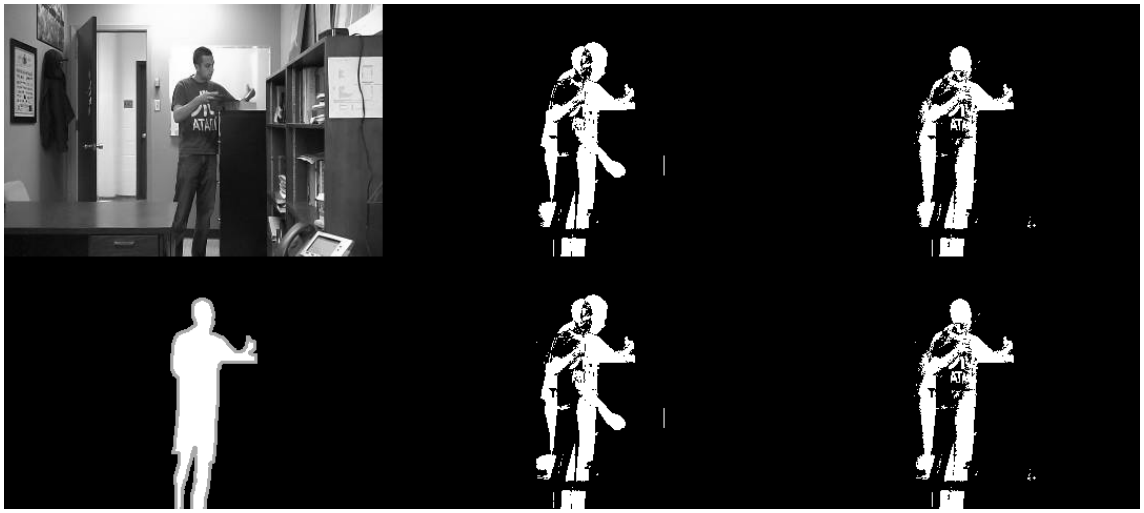


Figure 3.4: Example result of the proposed solution for *office* dataset. The frame index is 800. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the second row uses the median. Sequentially from the second column the result of the algorithm is presented: function approximation with conservative update policy without initialization buffer, function approximation with conservative update policy with initialization buffer.

3.10.2 Implementation of ViBE and PBAS methods

Exercise 3.8 Additional exercise 2

There are articles on the course website which describe the ViBE and PBAS methods. The task is to implement and evaluate them. Please follow the order, as PBAS is an extension of ViBE.

Figure 3.5 shows an example solution to the problem.

■

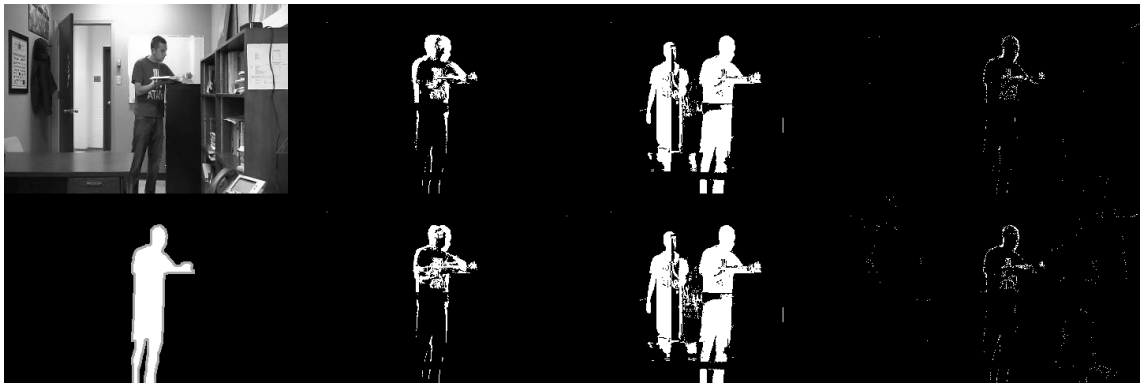


Figure 3.5: Example result of the proposed solution for *office* dataset. The frame index is 1000. The first column from the left is the input image together with the reference image. Then the approximation of the mean and median with a liberal update policy, the approximation of the mean and median with a conservative update policy and the last column is the ViBe algorithm (top) and the PBAS algorithm (bottom).

IV Laboratory 4

4. Optical Flow

4.1 Main objectives

- You will become familiar with the issue of optical flow,
- You will learn about the applicability of optical flow and its limitations,
- You will become familiar with the use of multiple scales in determining optical flow,
- You will implement the block method,
- You will become familiar with the methods available in the OpenCV library,
- You will run an example neural network for optical flow,
- You will use the optical flow to segment moving objects,
- You will be able to classify objects as rigid or non-rigid using optical flow.

4.2 Theory of Optical Flow

The optical flow is a vector field describing the movement of pixels (possibly only selected ones, i.e. feature points) between two consecutive frames from a video sequence. In other words, for each pixel on frame I_{n-1} we determine the displacement vector resulting in image I_n. It is worth noting, however, that the flow can be (and sometimes is even desirable) calculated for frames with a gap greater than '1'.

The optical flow can be *dense* or *sparse*. In the first case, the displacement is determined for each pixel, and in the second only for a limited set of them.

The basis of optical flow operation is tracking pixels (or certain local pixel configurations, i.e. surroundings) between frames. This means that on subsequent frames we look for the same pixels (or their small surroundings). This assumes that the brightness of a pixel is constant (on a greyscale, in the general case – constant colour).

Unfortunately, as the exercise will show, the approach fails in several cases, the most important of which are large, homogeneous areas. Why does this happen? Imagine a rather large carton box. It has grey, homogeneous walls with no texture and is illuminated with uniform light. We want to determine the optical flow for all pixels in the image, including the pixel from the centre of the box wall. Will we be able to find exactly this point on the next frame? Is an unambiguous assignment

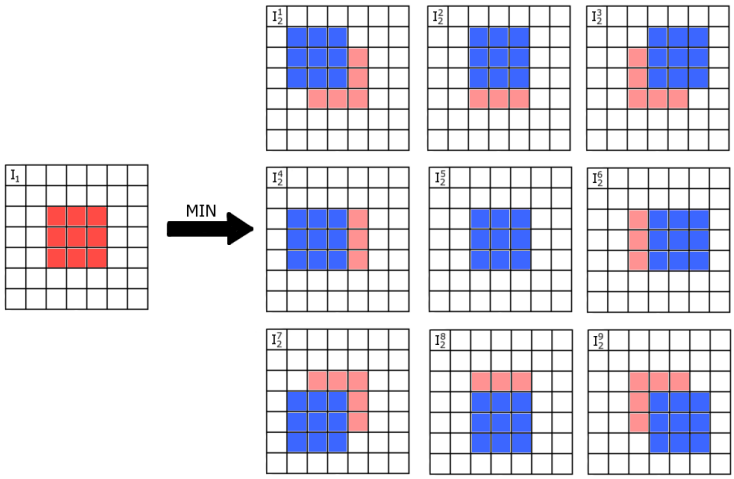


Figure 4.1: Example of searching for a matching block

possible?

For this reason, it is sometimes preferable to use a sparse flow, calculated at predefined locations. Here we have two possibilities. Either we arbitrarily select points e.g. on a square or rectangular grid (then the aim is to reduce the computation time), or we search for so-called feature points – these can be edges, corners or other distinct image elements. There are multiple algorithms for their detection, e.g. Harris corner detector, SIFT or SURF feature point detectors etc. (this will be the subject of one of the next exercises).

A very large number of methods have been described in the literature – please see the rankings available at Middlebury, KITTI, Sintel. The two most popular classical methods are Horn-Schunck (HS) and Lucas-Kanade (LK). The second one is available in the OpenCV library.

4.3 Implementation of the block method

The block method, as the name suggests, is based on matching some pixel blocks between successive frames. From one image, we crop blocks of a certain size (e.g., 3×3, 5×5, etc.) using the sliding window method across the image. For the second image, we look for the most similar block to the cropped one, with the search limited to a small neighborhood of the cropped block's coordinates. There are two reasons for doing this – usually the displacement of pixels between successive frames is small, and we also significantly reduce the execution time of the algorithm.

Exercise 4.1 Implement the block method for determining the optical flow.

1. Create a new Python script. Load *I.jpg* and *J.jpg* images. These will be two frames from the sequence, labeled `I` and `J` (earlier and later frame, respectively). Optionally, you can downscale both images during the tests, e.g. four times – the calculations will be performed faster. Convert images to grayscale – `cvtColor`. Display given images – using `namedWindow` and `imshow` is recommended. Visualize the difference with the command: `absdiff`.
2. Implement a block method for determining optical flow using the guidelines below. Make the following assumptions – we compare image patches of size 7×7. Further used symbols and values for the 7×7 window:
   - $W2 = 3$ – integer value from half the size of the window,
   - $dX = dY = 3$ – search size in both directions.

3. You should start your implementation with two `for` loops on the image. Use the $W2$ parameter taking into account the edge case – we assume that we do not calculate the optical flow on the edges.
4. Inside the loop, we cut out a part of the `I` frame. As a reminder, the necessary syntax is: `IO = np.float32(I[j-W2:j+W2+1,i-W2:i+W2+1])`.

   R In the exercise we assume that `j` – outer loop index (by lines), `i` – inner loop index (by columns).

   Please note the „+1" component – in Python the upper range is 1 greater than the largest assumed value – unlike in Matlab. Conversion to a floating point type is needed for further calculations.
5. Then we execute the next two `for` loops – searching around the pixel `J(j,i)`. They range from $-dX$ to dX and from $-dY$ to dY. Please don't forget the „+1". Inside the loop, check that the coordinates of the current context (i.e. its center) are within the allowed range. Alternatively, you can also modify the range of the outer loops – so as to exclude access to pixels beyond the acceptable range. In this case, for a slightly wider edge, the flow will not be determined, but it has no practical significance.
6. We cut out `JO` surroundings, converting them to *float32* and then calculating the „distance" between `IO` and `JO` slices. This can be done with the instruction: `np.sqrt(np.sum((np.square(JO-IO))))`. From all the fragments of `JO` for a given `IO`, find the smallest „distance" – the location of the „closest" to the `IO` patch in the image `J`.
7. The obtained coordinates of the minima found should be written in two matrices (for example `u` and `v`) – they should be created before (before the main loop) and initialized with zeros (the function `np.zeros`), image dimensions are like for `I`.
8. The determined field of the optical flow can be visualized in two ways – through the vectors (arrows) – the `plt.quiver` function from the *matplotlib* library or colors – we will implement the second approach. The idea is to find the angle and length of the vector defined by the two optical flow components `u` and `v` for each pixel. We will then get a data representation similar to the HSV color space. When displayed, the color indicates the direction in which the pixels move, while its saturation indicates the relative speed of pixel movement – analyze the color wheel in Fig. 4.2.
   Convert the determined flow to the polar coordinate system – `cartToPolar`. Create an image variable in HSV space with input image dimensions, 3 channels, and `uint8` type. The first channel is component H, equal to $angle * 90/np.pi$ (range from 0 to 180 in Python). The second channel is the S component – normalize (`normalize`) the vector length to the range 0-255. The third channel is component V, set it to 255.

   R To make no motion appear black instead of white, swap S and V channels.

   Finally, convert the image to RGB space and display it.
   Please analyze the results and experiment with the input image size and with the $W2$, dX and dY parameters (e.g. image reduced twice, parameters equal to 5). Do such parameters allow for the correct determination of the optical flow for the input image in original size? How large window is needed to get similar results?
   Check the method for a pair of images from a different sequence – *cm1.png* and *cm2.png*. The ground truth for this pair of images, used for evaluation of optical flow algorithms, is given in Fig. 4.3.

   R Due to its simplicity, the block method is quite inaccurate – the calculated horizontal

or vertical shift is integer. In other methods (e.g. HS or LK), the optical flow is mostly decimal.

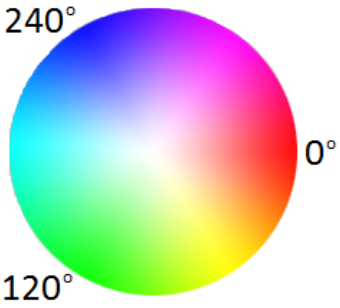


Figure 4.2: Color wheel

Example results for the analysed test sequences are shown in Figure 4.4.

Figure 4.3: Ground truth for `cm1.png` and `cm2.png` images.

4.4 Implementation of optical flow in multiple scales

The detection of large displacements by the implemented method is computationally complex. This is due to the need to use large values for the parameters dX and dY, and therefore for most locations (except edge locations) it will be necessary to check the similarity of more contexts (more SAD operations). This approach is very inefficient.

A better, and very commonly used idea, is to process the image at multiple scales. The idea is shown in Figure 4.5.

Schematic of how the method works:

- perform scaling of images (I and J) to scales: L_0 – original resolution, $L_1, \ldots, L_m$ (the smallest image) – this is referred to in the literature as an image pyramid. This type of solution usually uses scaling with a factor of 0.5.

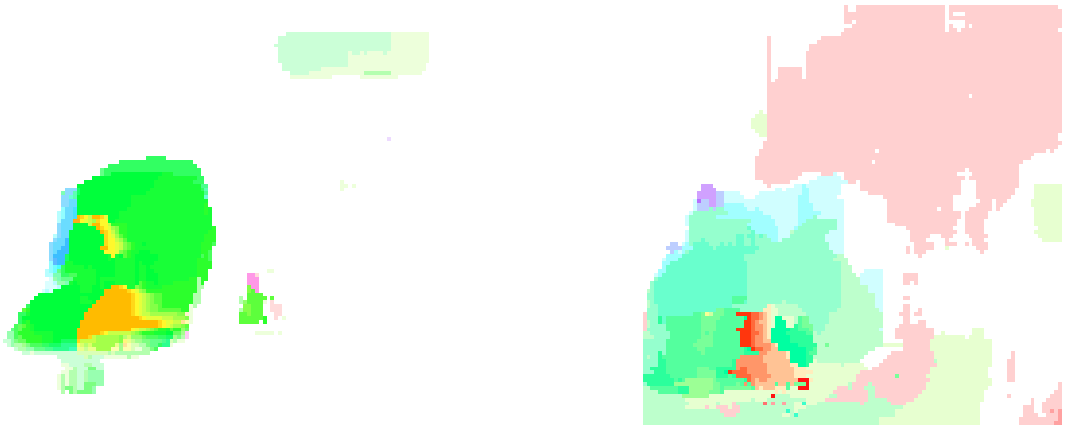


Figure 4.4: Example results of the optical flow determined by the block method: on the left for images `I.jpg` and `J.jpg`, image downscaled twice and parameters $W2 = dX = dY = 5$; on the right for images `cm1.png` and `cm2.png`, downscaled twice and parameters $W2 = dX = dY = 5$.

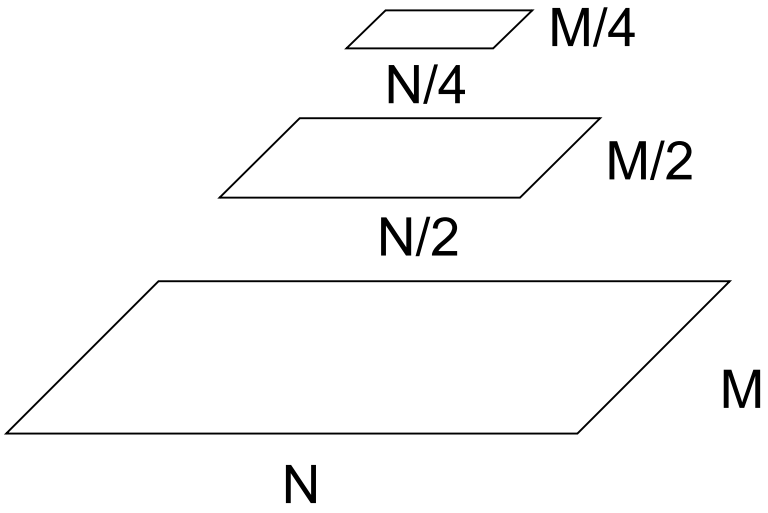


Figure 4.5: Example of pyramid image construction (multiple scales)

- calculate the optical flow for a pair of images I and J on a scale of L_m (e.g. using the block method).
- propagate the results of the calculated flow to the L_{m-1} scale.
  - in the first step the previous image frame is modified according to the flow (warping). The purpose of such a procedure is to compensate for motion, i.e. to shift pixels in such a way that on a larger scale L_{m-1} corresponding pixels in the two images are closer to each other,
  - the modified image is then upscaled to L_{m-1} (i.e. usually twice).
- we calculate more accurate flow values at L_{m-1} between the modified image I and J (e.g. using the block method),
- the procedure is carried out up to level L_0.

Exercise 4.2 Implement the multiscale version of the block method for optical flow calculation.

1. At the beginning, please create a copy of the algorithm you have created so far in Task 1. Next, we convert the part of the algorithm concerning the computation of the OF into **function** for the selected scale.
   The function should look like this:
   `def of(I_org, I, J, W2=3, dY=3, dX=3):`
   The individual parameters are `I_org` and `J` – input images, `I` – image after modification, `W2, dX, dY` – method parameters (identical to 4.3). In fact, you don't need to pass `I_org` to the function – the suggested solution is to calculate `absdiff` between images and display the results and images `I_org`, `I` and `J` for a better understanding of how the method works. In fact, the calculations are for `I` and `J`.
2. It will also be useful to convert the part of the algorithm concerning the visualisation of the optical flow into a **function**. This function should be of the form:
   `def vis_flow(u, v, YX, name):`
   The individual parameters are `u` and `v` – optical flow components, `YX` – image dimensions, `name` – window name displayed, e.g. `'of scale 2'`.
3. After writing the function, it is good to test its operation for one scale (L_0) – the result should be the same as the one obtained earlier.
4. After creating two functions based on the written algorithm, we proceed to write a function to generate a pyramid of images. The following function can be used:

```
def pyramid(im, max_scale):
  images=[im]
  for k in range(1, max_scale):
    images.append(cv2.resize(images[k-1], (0,0), fx=0.5, fy=0.5))
  return images
```

   In this case, we assume that the resolution reductions will always be double. So the input image is downscaled 2 times, 4 times, 8 times, etc. For the purposes of the experiment, we can assume the limit of 3 scales. A pyramid should be generated for each of the input images.
5. We start processing from the smallest scale, so to the `I` variable we assign the smallest-scale pyramid image: `I = IP [-1]`, where `IP` is the generated pyramid for the first image. Please note the syntax `[- 1]` – access to the last item.
6. The key component of the algorithm is the scales loop – please consider the starting and ending indexes. We start with the flow calculation and make a copy `I_new` of the first image. The next step is to modify this copy according to the flow. We perform this operation for all scales except the largest one (i.e. an image with the input's dimensions). This can be done using two `for` loops with the protection to keep the indexes in the allowed range. To individual pixels from `I_new` we assign appropriate pixels from `I`, according to the calculated flow. It is worth checking if the first image after modification is similar to the second image – if it is, the operation has been performed correctly. Finally, we need to prepare the image for the next (larger) scale. To do this, we upscale the image `I_new` and assign it to `I`. Upscaling: `I = cv2.resize(I_new, (0,0), fx=2, fy=2, interpolation=cv2.INTER_LINEAR)`.
7. The last element of the algorithm is the calculation of the total flow and its visualization. To do this, prepare an empty 2D table for each of the `u` and `v` components with the dimensions of the input image. Then, in a loop over the scales, we add the flows from different scales to each other.

A brief example to clarify. We have a pixel whose displacement is 9 (*ground*

truth), but the search is limited to 5 pixels in each direction. We reduce the image twice, now the pixel displacement should be 4.5. We find the most suitable pixel in a smaller scale – we get the result of 5, so according to it we modify the image (we move the pixel by 5 in the appropriate direction) and we increase it twice – now (in a larger scale) the analyzed pixel is shifted by 10. We count the flow again, this time in a larger scale, and we get the result of -1 (residual flow). Then we sum the results of both scales and we get the final flow value 9.

Conclusions. To get the correct final result, the `u` and `v` flows from each scale should be increased in two ways – the dimensions of these „images" must be the same as for the input `I` and the flow values must be increased 2, 4, 8 etc. times, depending on the scale. The summed flow from different scales should be visualized with the help of a function `vis_flow`.

8. Compare how the method works with and without multiple scales. Analyze the results for larger window sizes in one scale, and for smaller window sizes with multiple scales, pay attention to execution time. Was it possible to obtain the correct flow for a small window (e.g. $dX = dY = 3$) using a multi-scale method (e.g. 3 scales) for an image of original size (i.e. for a larger pixel displacement)?
   Again check the method for the image pair *cm1.png* and *cm2.png*.

   R The multiscale method works very well in theory – in practice, pixel matching is not always accurate, and scaling down and up always results in the loss of some information, so that even minor errors are propagated to subsequent scales and the final result may be partially incorrect or noisy. For this reason, in practice, 2, 3, or a maximum of 4 scales are used (but nevertheless, the results still differ slightly from the „expected" ones).

Example results for the analysed test sequences are shown in Figures 4.6-4.11.

■

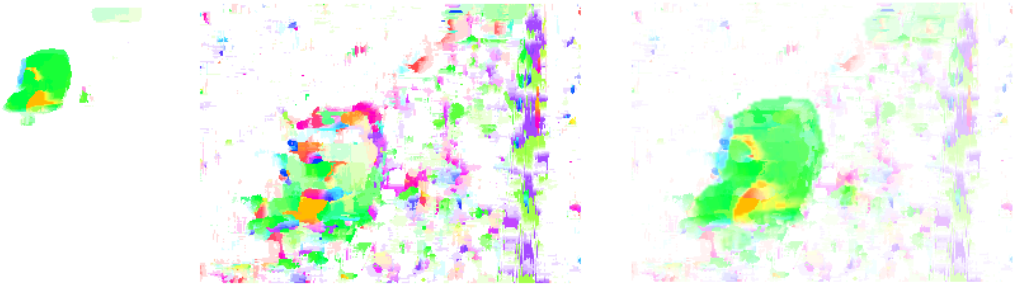


Figure 4.6: Example results of the optical flow determined by the block method in 2 scales for `I.jpg` and `J.jpg` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: flow at smaller scale, residual flow at larger scale, total flow.

Figure 4.7: Example warping results for `I.jpg` and `J.jpg` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: the previous frame I, the frame modified according to the lower scale flow I_org, the next frame J.

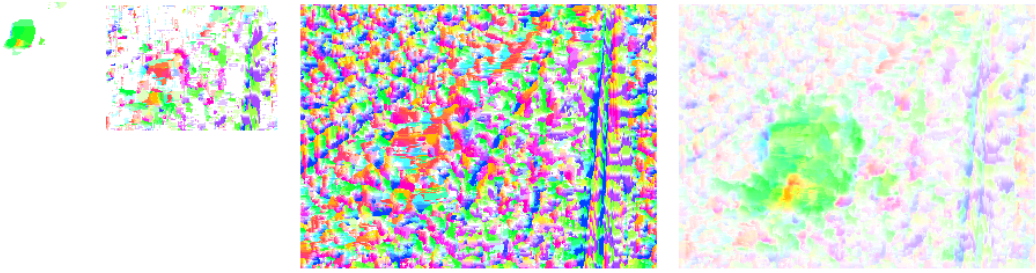


Figure 4.8: Example results of optical flow determined by the block method in 3 scales for `I.jpg` and `J.jpg` images, in original size and parameters $W2 = dX = dY = 3$. Sequentially from the left: smallest scale flow, middle scale residual flow, largest scale residual flow, total flow.

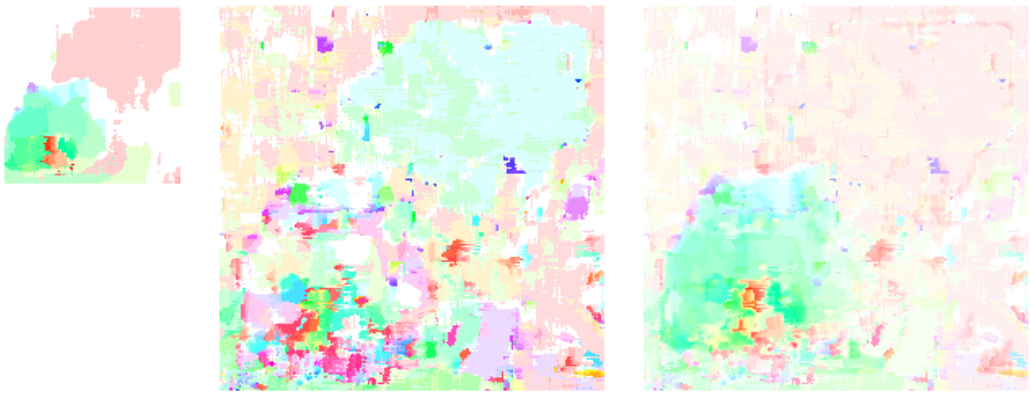


Figure 4.9: Example results of the optical flow determined by the block method in 2 scales for `cm1.png` and `cm2.png` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: flow at smaller scale, residual flow at larger scale, total flow.

Figure 4.10: Example warping results for `cm1.png` and `cm2.png` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: the previous frame I, the frame modified according to the lower scale flow I_org, the next frame J.

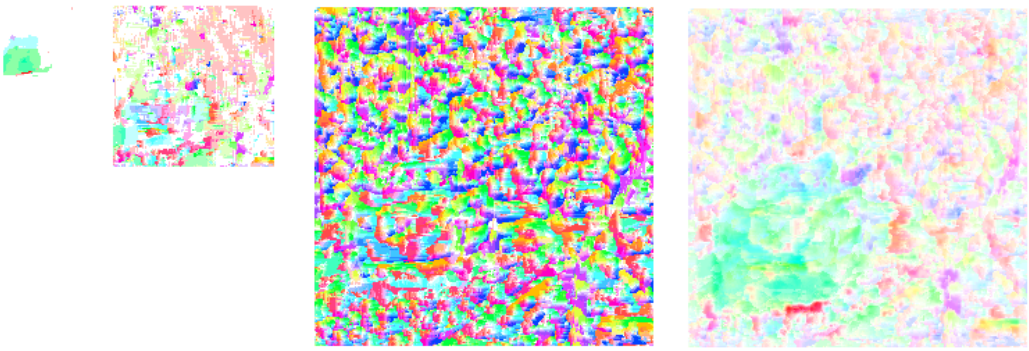


Figure 4.11: Example results of optical flow determined by the block method in 3 scales for `cm1.png` and `cm2.png` images, in original size and parameters $W2 = dX = dY = 3$. Sequentially from the left: smallest scale flow, middle scale residual flow, largest scale residual flow, total flow.

4.5 Additional exercise 1 – Other methods of determining optical flow

The following video sequences are available on the course page:

- *highway* -– motorway traffic monitoring sequence,
- *pedestrians* — campus monitoring record.

They are taken from the dataset from the changedetection.net website.

Use the script created in the previous exercises to read the image sequence. Make sure that it includes the parameter `iStep`, which allows you to set how many frames are skipped during processing.

R For simplicity, sequences in grey scale will be analysed.

Several functions are available in the OpenCV library to calculate optical flow:

- Lucas-Kanade (multi-scale, sparse and dense),
- Farneback,
- Dual TV-L1,
- PCA Flow,
- DIS Flow,
- Simple Flow,
- Deep Flow.

R Details of the specific algorithms are available in the OpenCV documentation. It is also worth taking a look at the optical flow tutorial and analysing it.

Exercise 4.3 Run sample solutions.

1. Task 1 – determine the optical flow by dense methods (i.e. all but sparse LK). The code should consist of several elements:
   - loading of input images and conversion to greyscale,
   - creating a variable `flow` with two channels (for `u` and `v`),
   - creating an instance of the given optical flow determination method and calling `calc` on it,
   - visualisation of results as in chapter 4.3.

   Note the results for different methods (including the accuracy of determining the edges of objects), as well as the time needed to perform the calculations.

   R For visualising dense LK, it may be useful to restrict the flow modulus to the interval 0-10, e.g. by `mag[mag>10]=10`.

2. Task 2 – run a sparse optical flow using the Lucas-Kanade method. The tutorial shows the tracking of feature points (methods for their detection will be part of one of the next exercises). We will calculate the optical flow for uniformly distributed points. So the code needs to be adapted accordingly. Some remarks:
   - the algorithm has a number of parameters, which should be clear after the analysis of the lecture (possibly read in the documentation) – size of the window, number of scales, criteria for terminating the calculation (solving the system of equations),
   - generate a grid of uniformly spaced points (e.g. a step of 10 pixels) – here it is useful to preview the form of `p0` from the example,
   - the second difficulty is a different visualisation. Example solution:

```
    img = I
```

```
    for jj in range(0, y, 1):
        if (st[jj] == 1):
            cv2.line(img, (points[jj,0,0],points[jj,0,1]), (new_points
                [jj,0,0],new_points[jj,0,1]), (0,0,255))
```

- please do not forget to assign the current image to the previous one at the end of the loop.

Please observe how the algorithm works – are the determined flows correct? You can add a vector direction to the display – mark the start or end of the vector with a dot. You can also remove vectors with a small modulus.

Example results for the analysed test sequences are shown in Figure 4.12.

3. Task 3 – run example neural networks for optical flow determination. As you can easily guess, many solutions have appeared in this area as well, which return much better results than classical methods. On the course page you can find implementations of two networks in PyTorch – LiteFlowNet and SPyNet, whose reimplementations in PyTorch were downloaded from GitHub and adapted for simple running. If necessary, modify the paths from which the sequences are loaded. Also modify the value of the `iStep` parameter (e.g. to 5 or 10) and check the network performance. Implementation details are left for self-analysis.

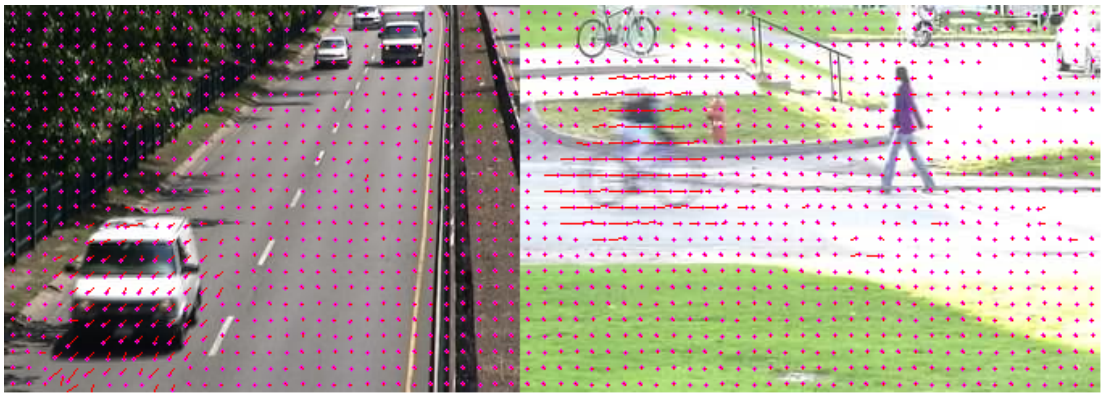


Figure 4.12: Example results of sparse optical flow determined by the LK method: left for image index 158 from the *highway* sequence; right for image index 471 from the *pedestrians* sequence.

4.6 Additional exercise 2 – Direction detection and object classification using optical flow

The idea is as follows. A car, which is a rigid solid, should be characterized by coherent motion – each of its elements should have similar displacement, at least in terms of direction. A walking person is a contrary, as the specifics of walking cause the limbs to move in different directions (depending on the phase of the walk).

So we will try to analyse the motion vectors for each object (depending on the sequence of the car or the human) and see if we can tell something from this and possibly make a classification.

Exercise 4.4 Implement the algorithm, based on the optical flow program from the previous task (e.g. Farneback's algorithm).

1. First we will add foreground object segmentation. Use the MOG/GMM algorithm used in previous labs.

2. A very important issue is mask filtration. Here again, the experience gained from previous exercises will be useful. It seems to be a good idea to disable shadow detection – `detectShadows=False` in the MOG2 constructor.
3. In the next step we add a labelling, the result of which we display.
4. We then perform an optical flow analysis. It comes down to calculating the mean and standard deviation for the modulus and angle of the vectors corresponding to a given object. The difficulty of the task depends on your level of skill in Python.A few hints (you may or may not want to use them):
   - we carry out calculations only if there are objects – `retval > 0`,
   - the key is a convenient container for averages and variances. You can use lists and the `append` instruction. Then we will get the functionality as in Matlab. It is easier to use two separate lists, as it is simpler to calculate the statistics later.
   - we perform the essential calculations in a loop on the image. If the label is different from zero then:
     - check whether the amplitude of the optical flow at this point is greater than a given threshold (e.g. 1),
     - if so, calculate the angle of the vector (`atan2` from `math`) and add the result to our list at the appropriate address.
   - in the next step we calculate the mean and standard deviation for the modulus and angle. The functions `mean` and `stdev` from the `statistics` package will be useful.

     R Before computing, you need to check that there are at least two elements in the list for the object.

   - the final stage is visualization. The easiest way to adapt the code from the moving object segmentation exercise is to display the determined parameters – two averages and two standard deviations. Optionally you can add filtering of small objects.
5. Analysis of results:
   - How do the mean and standard deviation for both sequences behave?
   - Does the average direction of movement correspond more or less to reality?
   - Is it possible to see any difference between the two sequences – standard deviation?
   - Are there problems with the segmentation? What kind of problems?

   Alternatively, you could try displaying the mean motion vector for each object.

■

Example results for the analysed test sequences are shown in Figure 4.13 and 4.14.

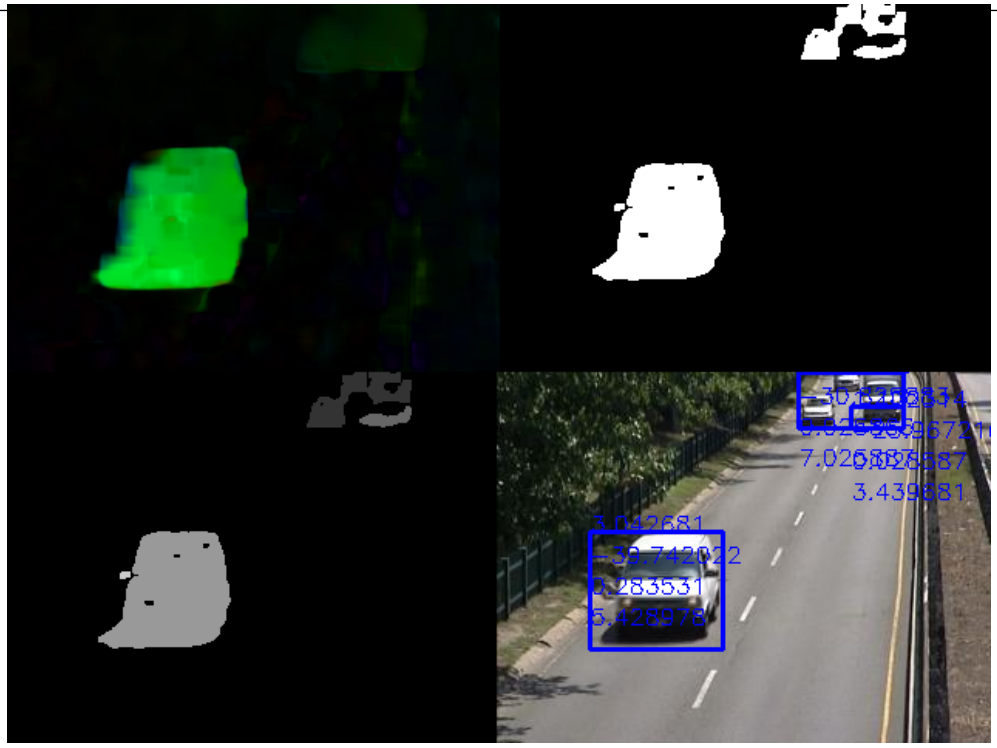


Figure 4.13: Example results for an image with index 146 from the *highway* sequence: top left – Farneback optical flow, top right – foreground object segmentation, bottom left – labelling, bottom right – detected objects with calculated parameters.

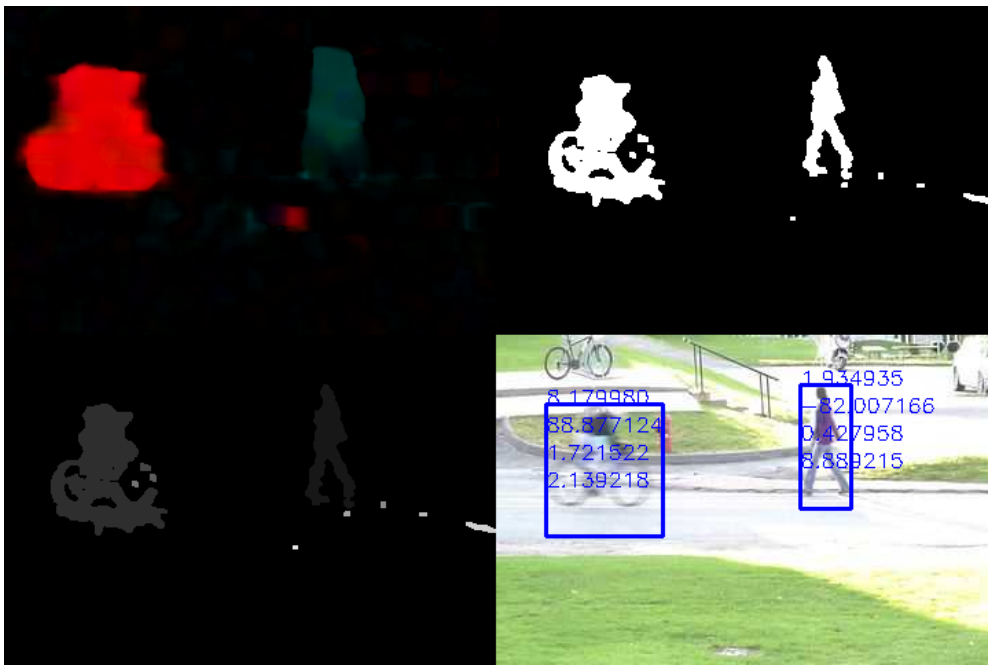


Figure 4.14: Example results for the image with index 471 from the *pedestrians* sequence: top left – Farneback optical flow, top right – foreground object segmentation, bottom left – labelling, bottom right – detected objects with calculated parameters.

V

Laboratory 5

5. Camera calibration and stereovision

A camera is a device that converts the 3D world into a 2D images. This relationship can be described by the following equation:

$$x = PX \tag{5.1}$$

where: x denotes a 2-D image point, P denotes the camera matrix and X denotes a 3-D world point.

Linear or nonlinear algorithms are used to estimate intrinsic and extrinsic parameters utilizing known points in real-time and their projections in the picture plane

Camera calibration is designed to eliminate distortions (distortion) introduced by the optical system. These distortions include: radial distortions (barrel, pincushion, mustache - related to the lens) and tangential distortions (related to the uneven installation of the lens and the matrix - Figure 5.1 and Figure 5.2.

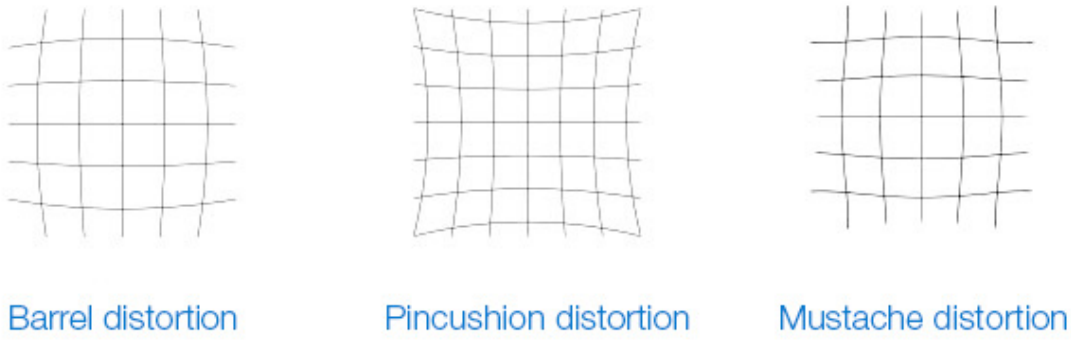


Figure 5.1: Types of distortion: (a) barrel, (b) pincushion (c) mustache. Source: https://www.panoramic-photo-guide.com/optical-distortions-panoramic-photography.html.

Distortion correction requires the calculation of some coefficients. This is possible in the calibration procedure, where an image of a pattern with known parameters (mutual distances

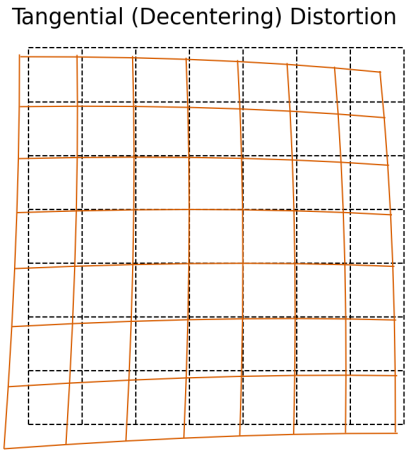


Figure 5.2: Tangential distortion. Source: https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i.

between points and their actual sizes) is recorded at different positions relative to the camera. The most commonly used pattern is a chessboard, on which corners can be easily detected, even with sub-pixel accuracy (as in the analysed image). Practice indicates that there should be at least 10 images.

The parameters determined in the camera calibration procedure are divided into two groups:

- Extrinsic or External parameters – it allows mapping between pixel coordinates and camera coordinates in the image frame, e.g. optical center, focal length, and radial distortion coefficients of the lens.
- Intrinsic or Internal parameters – tt describes the orientation and location of the camera. This refers to the rotation and translation of the camera with respect to some world coordinate system.

Figure 5.3 shows the equation of the camera model, which contains the homogeneous coordinates and the projection matrix. The calibration matrix is the product of matrices containing internal and external camera parameters. Linear or non-linear algorithms are used to estimate the internal and external parameters. Knowledge of the real parameters (distance between points, in centimetres) and their projections on a plane (in pixels) is used.

There are two camera models:

- *pinhole camera model* – basic camera model without lens, see OpenCV documentation for details: Pinhole model,
- *fisheye camera model* – primarily used in situations where distortion is extreme, models well for wide angle cameras, see OpenCV documentation for details: Fisheye model.

Performing the correct distortion correction – camera calibration – is essential if you want to measure the size of objects, e.g. in centimetres, measure distance or determine camera displacement.

5.1 Single camera calibration

The camera calibration procedure is divided into steps. Almost identical steps have to be followed for the calibration of a single camera or of stereo-vision cameras. A dataset containing image pairs from a stereovision camera will be used. To calibrate a single camera, a set from a single lens (left

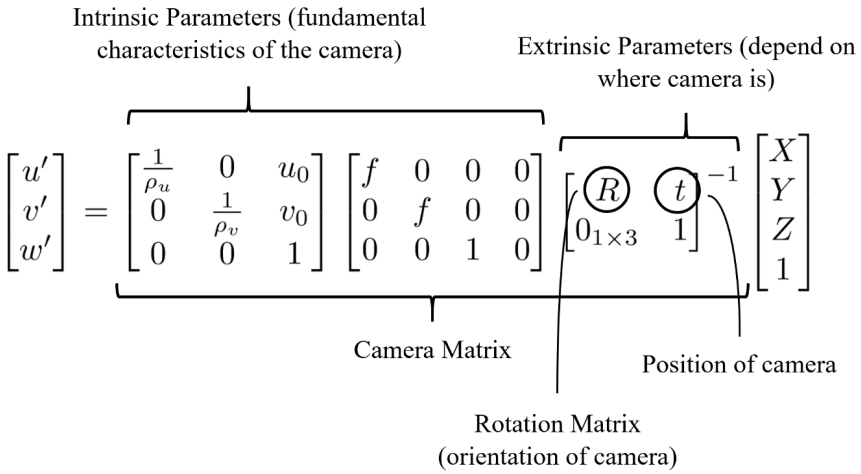


Figure 5.3: Camera model formula based on general equation 5.1. ρ_u, ρ_v is the image size x, y in pixels, u_0, v_0 is the principal point, f is the focal length. Source: https://towardsdatascience.com/understanding-transformations-in-computer-vision-b001f49a9e61

or right images respectively) should be selected. The proposed set was captured by a camera that needs to be modelled by a fisheye model.

In general, the camera calibration process can be divided into:

1. Selecting a proper calibration pattern. The most commonly used are: checkerboard, chArUco (combination of checkerboard and ArUco markers), circle grid, asymmetric circle grid.
2. Taking pictures of the calibration board from different angles and distances. The number of correctly taken images should be greater than 10.
3. Preparing the parameters of the calibration board, e.g. measuring the size of the chessboard fields in centimetres.
4. Preparing the code and running the algorithm for calibration.

The steps of the algorithm for calibrating a single camera are described below. A previously prepared set of images (available on the UPeL platform) and functions from the OpenCV library were used for this.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
calibration_flags = cv2.fisheye.CALIB_RECOMPUTE_EXTRINSIC+cv2.fisheye.
    CALIB_FIX_SKEW

# inner size of chessboard
width = 9
height = 6
square_size = 0.025  # 0.025 meters

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(8,6,0)
objp = np.zeros((height * width, 1, 3), np.float64)
objp[:, 0, :2] = np.mgrid[0:width, 0:height].T.reshape(-1, 2)

objp = objp * square_size  # Create real world coords. Use your metric.

# Arrays to store object points and image points from all the images.
objpoints = []  # 3d point in real world space
```

```
imgpoints = []  # 2d points in image plane.

img_width = 640
img_height = 480
image_size = (img_width,img_height)

path = ""
image_dir = path + "pairs/"

number_of_images = 50
for i in range(1, number_of_images):
    # read image
    img = cv2.imread(image_dir + "left_%02d.png" % i)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(gray, (width, height), cv2.
        CALIB_CB_ADAPTIVE_THRESH + cv2.CALIB_CB_FAST_CHECK + cv2.
        CALIB_CB_NORMALIZE_IMAGE)

    Y, X, channels  =  img.shape

    # skip images where the corners of the chessboard are too close to the edges of
         the image
    if (ret == True):
        minRx = corners[:,:,0].min()
        maxRx = corners[:,:,0].max()
        minRy = corners[:,:,1].min()
        maxRy = corners[:,:,1].max()

        border_threshold_x = X/12
        border_threshold_y = Y/12

        x_thresh_bad = False
        if (minRx < border_threshold_x):
            x_thresh_bad = True

        y_thresh_bad = False
        if (minRy < border_threshold_y):
            y_thresh_bad = True

        if (y_thresh_bad==True) or (x_thresh_bad==True):
            continue

    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)

        # improving the location of points (sub-pixel)
        corners2 = cv2.cornerSubPix(gray, corners, (3, 3), (-1, -1), criteria)

        imgpoints.append(corners2)

        # Draw and display the corners
        # Show the image to see if pattern is found ! imshow function.
        cv2.drawChessboardCorners(img, (width, height), corners2, ret)
        cv2.imshow("Corners", img)
        cv2.waitKey(5)
    else:
        print("Chessboard couldn't detected. Image pair: ", i)
        continue
```

With the coordinates of the points calculated, you can proceed to calibration:

```
N_OK = len(objpoints)
K = np.zeros((3, 3))
D = np.zeros((4, 1))
rvecs = [np.zeros((1, 1, 3), dtype=np.float64) for i in range(N_OK)]
tvecs = [np.zeros((1, 1, 3), dtype=np.float64) for i in range(N_OK)]
```

```
ret, K, D, _, _ = \
    cv2.fisheye.calibrate(
        objpoints,
        imgpoints,
        image_size,
        K,
        D,
        rvecs,
        tvecs,
        calibration_flags,
        (cv2.TERM_CRITERIA_EPS+cv2.TERM_CRITERIA_MAX_ITER, 30, 1e-6)
    )
# Let's rectify our results
map1, map2 = cv2.fisheye.initUndistortRectifyMap(K, D, np.eye(3), K, image_size,
    cv2.CV_16SC2)
```

R Description of the parameters obtained during the calibration process:

- `ret` – the value of the mean squared errors of the back projection (describes the quality of the matching),
- `K` – matrix of internal camera parameters in the form:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

  where: (f_x, f_y) – focal length, (c_x, c_y) – principal points.
- `D` – distortion coefficients,
- `rvecs` – rotation vectors,
- `tvecs` – translation vectors (together with the rotation vectors allow to transform the position of the calibration standard from the model coordinates to the real coordinates)

Having obtained all the necessary parameters in the calibration process, it is now necessary to select any image from the given set and remove the distortion. To do this, the function `cv2.remap()` will be used:

```
undistorted_image = cv2.remap(image, map1, map2, interpolation=cv2.INTER_LINEAR,
    borderMode=cv2.BORDER_CONSTANT)
```

Exercise 5.1 Using the above description to calibrate a single camera, complete the task.

- Determine the camera parameters and display them.
- Check the correction for one of the images in the set (straight line test).
- Perform distortion correction for the entire set of images. Display and compare the image before and after calibration.

■

5.2 Stereo camera calibration

In the most simplistic terms, the calibration of a camera system (for simplicity, two cameras) is intended to allow easy calculation of depth maps. This issue can be understood in different ways: setting identical parameters of the two cameras, eliminating distortions, converting to a common coordinate system and rectification. The last issue is particularly interesting. In the general case, the corresponding pixels in both images lie on epipolar lines. Rectification consists in transforming the images in such a way that the lines are parallel to one of the image edges (usually horizontal). This makes it much easier to determine the correspondence (depth map) - you only have to search in one plane (and not in two dimensions or along a diagonal line (necessary interpolation)).

The initial operations are carried out similarly to those for a single camera. It is best to copy the created code to a new file, add the loading of the second image and make the appropriate modifications. As a result, we should get the camera parameters for the left and right images (output from `cv2.fisheye.calibrate`).

Then we execute the three functions that are responsible for stereo calibration, rectification and calculation of the mapping coefficients for the `remap` function:

```
imgpointsLeft = np.asarray(imgpointsLeft, dtype=np.float64)
imgpointsRight = np.asarray(imgpointsRight, dtype=np.float64)

(RMS, _, _, _, _, rotationMatrix, translationVector) = cv2.fisheye.stereoCalibrate(
        objpoints, imgpointsLeft, imgpointsRight,
        K_left, D_left,
        K_right, D_right,
        image_size, None, None,
        cv2.CALIB_FIX_INTRINSIC,
        (cv2.TERM_CRITERIA_EPS+cv2.TERM_CRITERIA_MAX_ITER, 30, 0.01))

R2 = np.zeros([3,3])
P1 = np.zeros([3,4])
P2 = np.zeros([3,4])
Q = np.zeros([4,4])

# Rectify calibration results
(leftRectification, rightRectification, leftProjection, rightProjection,
    dispartityToDepthMap) = cv2.fisheye.stereoRectify(
        K_left, D_left,
        K_right, D_right,
        image_size,
        rotationMatrix, translationVector,
        0, R2, P1, P2, Q,
        cv2.CALIB_ZERO_DISPARITY, (0,0) , 0, 0)

map1_left, map2_left = cv2.fisheye.initUndistortRectifyMap(
        K_left, D_left, leftRectification,
        leftProjection, image_size, cv2.CV_16SC2)

map1_right, map2_right = cv2.fisheye.initUndistortRectifyMap(
        K_right, D_right, rightRectification,
        rightProjection, image_size, cv2.CV_16SC2)
```

Description of the parameters used in the above methods:

- `imgpointsLeft, imgpointsRight` – list of detected checkerboard corners for the left and right image, the values come from the function `cv2.cornerSubPix`,
- K_left, K_right – matrix of internal camera parameters for a set of image pairs, the values come from the function `cv2.fisheye.calibrate`,
- D_left, D_right – camera distortion coefficients, values are derived from the function `cv2.fisheye.calibrate`.

Finally, it remains for us to load two images from the set (any one) and perform the mapping:

```
dst_L = cv2.remap(img_l, map1_left, map2_left, cv2.INTER_LINEAR)
dst_R = cv2.remap(img_r, map1_right, map2_right, cv2.INTER_LINEAR)
```

To check how the rectification works, it is a good idea to put the two images side by side and draw horizontal lines to see if the same pixels are actually on the same lines. This can be done, for example, as follows:

```
N, XX, YY = dst_L.shape[::-1] # RGB image size
```

```
visRectify = np.zeros((YY, XX*2, N), np.uint8) # create a new image with a new size
    (height, 2*width)
visRectify[:,0:XX:,:] = dst_L       # left image assignment
visRectify[:,XX:XX*2:,:] = dst_R    # right image assignment

# draw horizontal lines
for y in range(0,YY,10):
    cv2.line(visRectify, (0,y), (XX*2,y), (255,0,0))

cv2.imshow('visRectify',visRectify)  # display image with lines
```

R It is important to know that there is also a function in OpenCV to rectify uncalibrated images `stereoRectifyUncalibrated`. We encourage you to enable it.

Exercise 5.2 Perform calibration and rectification process for stereo camera. Display the image with distortions removed and with horizontal lines to confirm that the algorithm is working correctly. ■

5.3 Stereo correspondence problem

The calculation of stereo correspondences (depth maps, disparity maps) is relatively straightforward (at least in theory and for suitable images). The idea is to find by how much a pixel $I_L(x,y)$ is shifted (d – disparity) in the second image $I_R(x+d,y)$. As a reminder – we are only looking in horizontal lines, since we assume that the images have been rectified. You may also notice that the task is similar to optical flow in a sense. Only there we had two consecutive frames from a sequence recorded with one camera, and here two images from two cameras at the same time. By the way, it is useful to know that depth can be reconstructed from a sequence from a single camera too – as long as this camera is moving – the issue of *Structure from Motion*.

There are two methods available in the OpenCV library for computing depth maps: block matching (*Block Matching*) and SGM (*Semi-Global Matching*). The operation of the first one is quite obvious, in the case of the second one the optimization of the global disparity map is also performed (in a simplified form), which allows to improve the continuity of the map.

Exercise 5.3 Please, based on the documentation, determine the disparity map for one of the images from the *example* folder before and after the calibration process. Please choose the parameters of the individual functions empirically, following the constraints. Display the original image before and after calibration and the corresponding disparity maps (SGM and BM).

An example solution is presented in Figure 5.4. ■

R The disparity map should be normalized to the range $0-255$ before display. You can also apply a function to convert the disparity to a heatmap: `cv2.applyColorMap(map, cv2.COLORMAP_HOT)`.

5.4 Using a neural network for the task of image depth analysis

As an exercise, we propose to analyse a deep convolutional neural network that allows to obtain depth maps from **single-view depth** method. In addition, the network is trained in an unsupervised manner (i.e. without a teacher – in this case without reference depth maps). Furthermore, the network implements camera position estimation. The details of the method used can be read in the

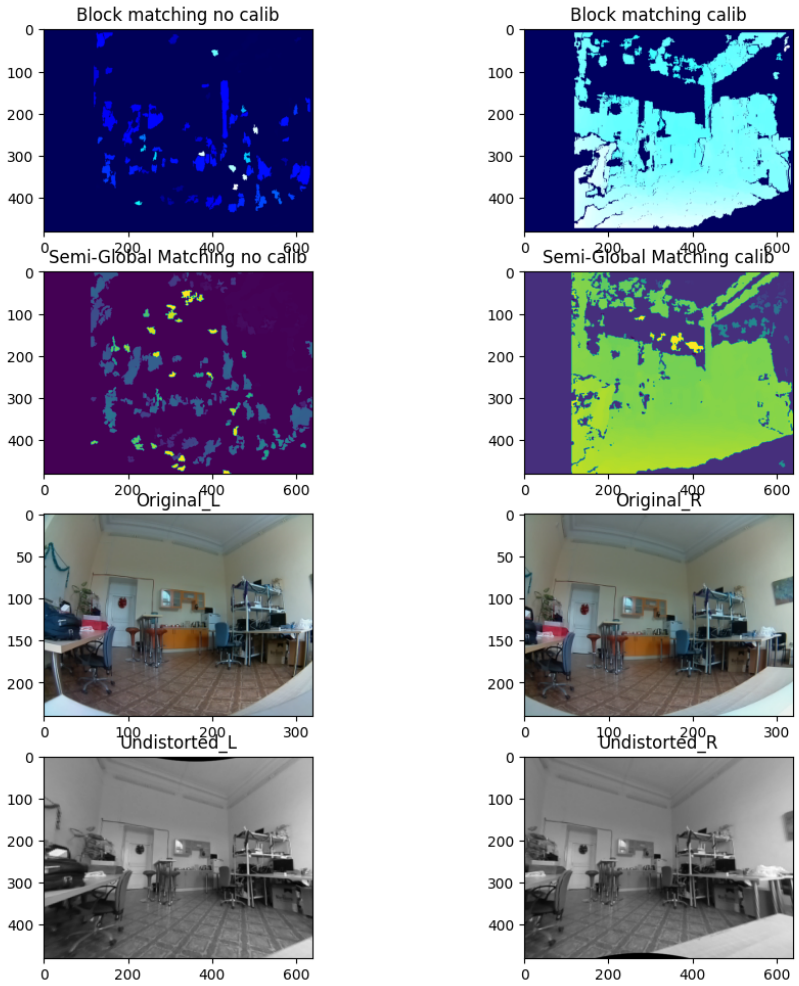


Figure 5.4: An example solution.

paper ZHOU, Tinghui, et al., Unsupervised Learning of Depth and Ego-Motion from Video. We will use the network model provided by the authors. The code is available on the GitHub repository – SfmLearner-Pytorch.

5.5 Additional exercise – Census transformation

A simple but quite good method is the so-called Census transformation (also known as *Local Binary Patterns*). We take a block of $N \times N$ such as 5×5 (test larger sizes up to 100). We binarise it with a threshold equal to the median value from the context. All values less receive 0, greater or equal to 1. And then we read the values from the context line by line and we get the binary string. In the second image we do the same, but in d positions (from 0 to $d_{max} - 1$). Comparing the two contexts is the operation $xor(C1, C2)$. We count the number of differences and look for the minimum – this will be our d (disparity) value.

Exercise 5.4 Implement the described method and compare the operation with BM, SGM and DCNN (at least visually). Use the images in the *aloes* folder for this.

An example solution is presented in Figure 5.5. ■

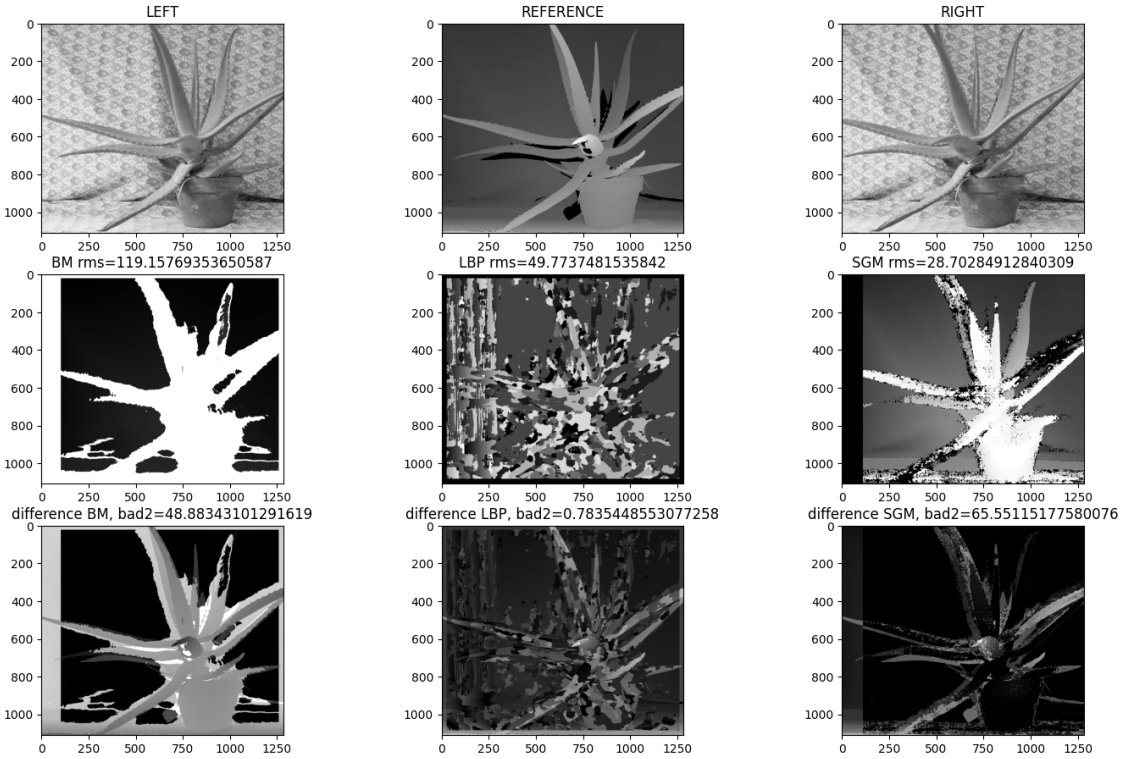


Figure 5.5: An example solution.

VI

Laboratory 6

6. Feature points

6.1 Main objectives:

- introduction to the problem of detecting characteristic points,
- implementation of a simple detection algorithm - Harris method,
- learning how to describe a feature point,
- the surroundings of a point as a descriptor,
- comparison of the Harris, SIFT and ORB (FAST+BRIEF) methods.

R In today's exercise, please use the functions from `matplotlib.pyplot` instead of OpenCV to display images.

6.2 Harris corner detection – theory

Harris corner detection is based on finding pixels for which the vertical and horizontal gradient modulus has a significant value.
The method is described by the following relation:

$$H(x,y) = det(M(x,y)) - k * trace^2(M(x,y)) \tag{6.1}$$

where: det – matrix determinant, $trace$ – matrix trace, k – constant value, (x,y) – image coordinates.
The autocorrelation matrix M is:

$$M(x,y) = \begin{bmatrix} < I_x^2(x,y) > & < I_{xy}(x,y) > \\ < I_{xy}(x,y) > & < I_y^2(x,y) > \end{bmatrix} \tag{6.2}$$

where each symbol means:

$$< I_x^2(x,y) > = I_x^2(x,y) \otimes h(x,y) \tag{6.3}$$
$$< I_y^2(x,y) > = I_y^2(x,y) \otimes h(x,y) \tag{6.4}$$
$$< I_{xy}(x,y) > = I_x I_y(x,y) \otimes h(x,y) \tag{6.5}$$

where: $I_x(x,y)$ oraz $I_y(x,y)$ are partial derivatives in the x and y directions, values determined by using for example the Sobel gradient, the symbol $\otimes$ denotes convolution, and $h(x,y)$ is a Gaussian function:

$$h(x,y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{6.6}$$

where: σ – standard deviation. Gaussian filtering is used to reduce the impact of noise, which has a particularly strong effect on the calculation of gradient values.
The determinant and trace of a matrix 2×2 (as a reminder):

$$det(M(x,y)) = < I_x^2(x,y) >< I_y^2(x,y) > - < I_{xy}(x,y) >^2 \tag{6.7}$$

$$trace(M(x,y)) = < I_x^2(x,y) > + < I_y^2(x,y) > \tag{6.8}$$

Finally, a given pixel is classified as a corner if the calculated value $H(x,y)$ is greater than a certain threshold.

As a final filtering it is useful to use the search for local maxima. We consider as a corner candidate only those pixels that are local maxima in their neighborhood (e.g. $7times7$).

6.3 Implementation of the Harris method

Exercise 6.1 Based on the above theory, implement the Harris method.

1. From the course page, download the archive with the data for the exercise and unpack it in your own working directory.
2. **Load images** *fontanna1.jpg* and *fontanna2.jpg*.
3. **Implement a function** that calculates the value of H from the autocorrelation matrix. The function should receive as parameters the grayscale image and the size of the Sobel and Gauss filter masks used by it (this can be the same size for both filters). Filter the image with a vertical and horizontal Sobel filter (function `cv2.Sobel`) – this will be the implementation of the derivative of the image after x and y. Specify `cv2.CV_32F` as the Sobel result type. Compute the corresponding products of the directional derivatives and blur them using a Gaussian filter (function `cv2.GaussianBlur` – example call: `cv2.GaussianBlur(image, (size, size), 0)`, where *size* is the filter mask size). Blurring with the Gaussian filter corresponds to calculating a weighted sum of elements from the neighborhood of each matrix point (weighted by the Gaussian curve). Enter the H values (from the formula using the calculated determinants and traces) into the resulting image. Assume a coefficient value of `K = 0.05`. Return the output image (it is useful to normalise it e.g. to the range 0-1 to make it easier to select the threshold in the next section)
4. **Use** the following function to find local maxima in the array received as a parameter:

```
import scipy.ndimage.filters as filters

def find_max(image, size, threshold) : # size - maximum filter mask size
    data_max = filters.maximum_filter(image, size)
    maxima = (image==data_max)
    diff = image>threshold
    maxima[diff == 0] = 0
    return np.nonzero(maxima)
```

The maxima are searched here using a maximum filter. The method is simplistic (it can

return several maxima that lie within the mask), but sufficient for our purposes. The function returns the found maxima as lists of coordinates – one list for each dimension. For a 2D array, these will be lists of y coordinates and x coordinates, respectively. The function also works for 3D arrays – in which case it will return three lists, with the first list being the z coordinates (i.e., the height in an octave of a pyramid, for example).

5. For both loaded images, apply the above functions (the second is to search for local maxima in the result returned by the first). You can take 7 as mask size in both functions. Display the results from the second function as marks (e.g. *) plotted on the initial images. It is a useful approach to create a separate drawing function for this purpose, which will create `figure()`, display the image using `plt.imshow`, and then apply characters to it using the `plt.plot` function.
   Example `plt.plot(5, 10, '*', color='r')` will plot a red star in (x,y) coordinates equal to $(5,10)$. Multiple stars can be drawn simultaneously using this function: `plt.plot([1,2,3], [4,5,6], '*')` draws three blue stars in (x,y) coordinates equal to $(1,4)$, $(2,5)$ and $(3,6)$. The lists $[1,2,3]$ and $[4,5,6]$ can be obtained from the second function – but note the order of the coordinates!
6. **Display and check** whether the detected points in both images are (more or less) in the same positions (i.e. whether the detector is repeatable)
7. **Repeat** the above steps for the images *budynek1.jpg* and *budynek2.jpg*.

■

6.4 Simple descriptor of feature points

Exercise 6.2 This exercise is a continuation of the previous task. After the detection of the characteristic points, their description is still required.

1. Add a function to the function from the previous task **add function** that creates descriptions of feature points. The description will be image patch from around the feature point with the size given by the parameter. As parameters, the function should receive an image, a list of coordinates of feature points (the result of the function from the previous task) and the size of the neighborhood. Remove from the list of points all points whose neighborhoods do not fit into the image. Here you can use the function *filter* with an anonymous lambda function – for example (X,Y is the size of the image, *size* is the size of a neighbourhood/patch):

```
pts = list(filter(lambda pt: pt[0] >= size and pt[0]< Y-size and    pt[1]
    >= size and pt[1] < X-size, zip(pts[0],pts[1])))
```

   Then create a list of point descriptions after filtering. At the same time it will be good to convert it to a vector. If e.g. `p` is an array, a vector is obtained from it using `p.flatten()`. As a result of the function, a list of neighbourhoods/patches supplemented with the coordinates of their central points should be returned.
2. At this stage the filtered list of coordinates can be zipped with the list of neighbourhoods/-patches (zip function) – for example:

```
wynik_funkcji = list(zip(list_patches, l_fp_coords))
```

3. **Add a function** that compares feature point descriptions from two images and finds the most similar descriptions. The function should receive as parameters two lists of feature point descriptions (from the function from 1) and the number of `n` most similar

descriptions. The function can use the 'any-to-any' method (brute force) – each point description in the first list is compared with all descriptions in the second list, and the most similar description is selected. The measure of similarity can be specified differently. It can be their vector distance, the sum of the absolute values of their differences, or the result of their scalar product. The coordinates of the pair of points corresponding to the pair of most similar descriptions/vectors are to be included in the resulting list together with the similarity measure. The function should return a list of `n` of the most similar descriptions. To do this, one can, for example, sort the list (sort method) and take its first `n` elements (this can also be done manually by selecting the most similar match n times). The sort method works 'in place', that is, it replaces the list with a sorted list. To use the sort method to sort a list of tuples, you need to pass a function to the method that specifies by which element of the tuple you want to sort. For example, sorting by the second element of the tuple would be:

```
lst.sort(key=lambda x: x[1])
```

In other words, the key parameter expects a function that will return consecutive items for sorting. Note also that `x[1]` – is a reference to the **second** element of the tuple, since we number from 0.
By default, the `sort` method sorts in ascending order. If you want to sort in descending order, set the `reverse` parameter to texttttrue.

4. **Load images** *fontanna1.jpg* and *fontanna2.jpg*. Find feature points for them using the Harris method using the function from the previous task. For the points you find, create lists of descriptions using the functions from 1. Set the size of the neighborhood to `15` (you can experiment with other settings). Search for the best matches using the function in 3. The parameter `n` can be set to `20`.
5. **View results.** Among the data for the exercise is the file *pm.py*. You can import it into your file (*import pm*) and use the *pm.plot_matches* function in it. However, you may need to adapt this function to your list returned by the function from 3 (the provided implementation assumes that each pair is stored as $([y1,x1],[y2,x2])$, the images should be in grayscale).
6. **Repeat the operations** from the previous points for the images *budynek1.jpg* and *budynek2.jpg*. Notice that in these images the buildings are located at a slightly different angle. Has this affected the quality of the fit?
7. **Repeat the operations** from the previous points for the images *fontanna1.jpg* and *fontanna_pow.jpg*. The images differ significantly in scale. Has Harris dealt with such a difference?
8. **Repeat the operations** from the previous points for the images *eiffel1.jpg* and *eiffel2.jpg*. The images differ in brightness and somewhat in scale. Has Harris dealt with such a difference?
9. **Modify the function** that determines feature point descriptions (from 1) to account for affine changes in brightness:

$$w_{aff} = \frac{w - \bar{w}}{|w - \bar{w}|} \tag{6.9}$$

Whereby the mean $\bar{w}$ can be determined as `np.mean(w)`, and `|w - w̄|` as the standard deviation using `np.std(w)`.
Now try repeating the operations for the images listed above. Has there been any improvement?

6.5 ORB (FAST+BRIEF)

Currently, there are three most popular algorithms and their modifications for feature point detection and descriptor, i.e. SIFT, SURF, ORB. The SIFT method (*Scale-Invariant Feature Transform*) has high efficiency and accuracy, but requires very high computational effort. The SURF method (*Speed Up Robust Features*) is an approach that requires less computational complexity, but the quality of the results obtained in selected cases is significantly worse than SIFT. The ORB (*Oriented FAST and Rotated BRIEF*) algorithm is an efficient alternative to SIFT and SURF.

The Features from Accelerated Segment Test (FAST) detector is used to detect feature points. It is designed to find corners, similar to the Moravc and Harris algorithms. It performs this task by comparing the brightness I_c of a given image point with 16 pixels located on the surrounding circle. This is illustrated in figure 6.1.

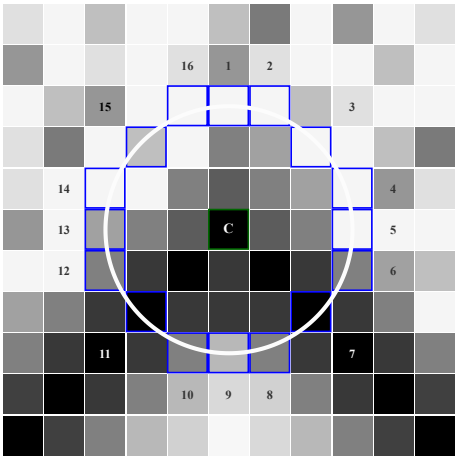


Figure 6.1: Position of the pixels whose brightness is compared with the centre point c in the FAST detector.

A central point is considered a vertex in this case if n of consecutive pixels on the circle has a brightness lower than $I_c - t$, or higher than $I_c + t$, where t defines some threshold – a parameter of the algorithm. The best quality results can be obtained by taking $n = 9$. However, the risk of unwanted edge detection inherent in vertex detectors may be a problem. In view of this, it is necessary to rank all obtained vertices with respect to the Harris measure (6.1), and then choose N of the best results. All the vertices thus indicated are unfortunately not independent of the angle, rotation of the image and its scale. In order to solve the first problem, the idea of *intensity centroid* was used. It is based on the determination of geometric moments of an image patch around a given feature point, defined by the equation:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \tag{6.10}$$

where: x, y -– local coordinates relative to the detected feature point, $I(x,y)$ – brightness of the pixel at point (x,y).

In this case, the summation proceeds over all pixels within a circle of radius r and centre at a given feature point. From this, the coordinates of the centroid of C can be determined:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \tag{6.11}$$

The orientation of a given feature point is determined by the following:

$$\theta = \arctan 2(m_{01}, m_{10}) \tag{6.12}$$

In order to make the ORB algorithm resistant to scale, a pyramid of images should be used.

Once a set of feature points has been obtained, the next step is to locally descript them. For this purpose, the BRIEF algorithm (*Binary Robust Independent Elementary Features*) is used. It consists in constructing n-element bit strings, according to the relation:

$$f_n(p) := \sum_{i=1}^{n} 2^{i-1} \tau(\mathbf{p}; u_i, v_i) \tag{6.13}$$

whereby the i-th binary test τ is defined by Eq:

$$\tau(\mathbf{p}; u_i, v_i) := \begin{cases} 1 & \text{dla } I(u_i) < I(v_i) \\ 0 & \text{dla } I(u_i) \geq I(v_i) \end{cases}$$

where:
p – a image patch around a given feature point,
u_i, v_i – some two pixels inside $\mathbf{p}$ $(u_i \neq v_i)$,
$I(w)$ – brightness of the image at a point $w = (x, y)$.

The image patch $\mathbf{p}$ should be blurred in this process to reduce the impact of noise. The blurring is performed by integral images in which a given feature point was represented by the summed brightness of the pixels surrounding it (assumed for this purpose to be a square neighbourhood of 5×5 at a image patch size of 31×31). Fundamental to the performance of the whole algorithm is also the appropriate selection of pixel pairs for the individual τ tests. The location of the test pairs is defined randomly – Figure 6.2 or deterministically based on a specially developed learning algorithm proposed by the authors of the ORB algorithm – [1]. The output of the descriptor is a 256-element binary vector. The Hamming metric is used to determine the similarity between two binary vectors.

Exercise 6.3 The aim of the exercise is to learn and implement the elements of the ORB algorithm in a simplified version. The theory described above will be helpful. The results should be compared with the results of the previous exercise.

1. Use the pair of images used in the previous task, e.g. *fontanna1.jpg* and *fontanna2.jpg*.
2. **Implement the FAST detector** to detect feature points and determine the Harris measure for each point. The FAST detector determines the feature point based on a image patch of size 7×7 px.
3. Then, using the *non-maximum suppression* method, we perform the first filtering. It consists in removing feature points that are close to each other. We check if there are multiple detected feature points in the neighborhood of 3×3 px, if so leave the one with the highest measure.
4. The next step is to remove feature points that do not have the full environment 31×31

[1] Rublee, Ethan, et al. "ORB: An efficient alternative to SIFT or SURF." 2011 International conference on computer vision. IEEE, 2011.

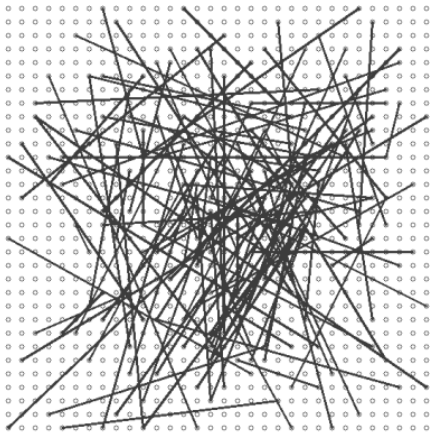


Figure 6.2: Randomly generated point pairs for the BRIEF descriptor.

used by the descriptor.

5. Sort the received points according to the Harris measure. Select the N of the best points.
6. Determine the coordinates of the centroid C and the orientation of the feature point based on the idea of *intensity centroid* for each point.
7. **Implement the BRIEF descriptor**. A 31×31 image patch should first be blurred with a gauss of 5×5 px. Pairs of points can be determined randomly from an appropriate interval or ready pairs of points can be loaded from a file `orb_descriptor_positions.txt`. However, these are the points defined for the orientation 0°. Depending on the angle α obtained from the *intensity centroid*, pairs of points should be rotated, according to Eq:

$$x' = \cos\alpha * x - \sin\alpha * y \tag{6.14}$$
$$y' = \sin\alpha * y + \cos\alpha * x \tag{6.15}$$

   We perform a binary test for all pairs of points, obtaining a 256-bit vector.
8. To compare feature points from two images, the Hamming metric must be used. A brute force method can be used here – we determine the similarity for each pair of points from two images.
9. We then sort by the best similarity and select the N of the best points.
10. **Display the results** and **compare** with the results from the previous task. Note that the ORB algorithm, compared to the previous task, should deal with rotation.

■

6.6 Using feature points to combine images into a panorama

Exercise 6.4 The task will use ready-made functions for feature point detection and descriptors from the OpenCV library.

1. Load images: *left_panorama.jpg* i *right_panorama.jpg*. Convert images to greyscale.
2. Find the feature points in the images. Any descriptor can be used for this: `cv2.SIFT_create()`, `cv2.xfeatures2d.SURF_create()` or `cv2.ORB_create()`. However, note that only ORB returns a binary description of the feature point.
3. Display the feature points found using the function: `cv2.drawKeypoints`.
4. We then match the feature points from both images. This is what the class is for:

cv2.BFMatcher(NORM). For SIFT and SURF the norm will be `cv2.NORM_L2` and for ORB `cv2.NORM_HAMMING`.

5. There are two matching functions: Brute Force (BF) or KNN. If we choose BF then we call the `.match()` method, if KNN then the `.knnMatch()` method.
6. If BF then we sort the obtained connections by distance (the smaller the distance between the descriptor vectors the better the similarity) and select the N best points. If KNN then we use the following code:

```
best_matches = [ ]
for m,n in matches: # m i~n - best and second best matches
    if m.distance < 0.5*n.distance:  # the best is twice as good as the
        second
        best_matches.append([m])
```

or the same using list comprehension:

```
best_matches = [ [m] for m,n in matches if m.distance < 0.5*n.distance]
```

7. To check, you can view the received matches – `cv2.drawMatches`.
8. The next step is to determine the rotation and translation of the images (points) with respect to each other, i.e. to determine the homography matrix – function `cv2.findHomography`. Before passing feature points to this function, we need to convert them appropriately. We convert the set of points to a numpy array:

```
keypointsL = np.float32([kp.pt for kp in keypointsL])
keypointsR = np.float32([kp.pt for kp in keypointsR])
```

and then create sets of corresponding pairs already matched with each other:

```
ptsA = np.float32([keypointsL[m.queryIdx] for m in matches])
ptsB = np.float32([keypointsR[m.trainIdx] for m in matches])
```

9. The final step is to call the function:

```
result = cv2.warpPerspective(image_left, H, (width, height)
```

where: (width, height) is the sum of the dimensions of the two processed images. and the connection to the other image:

```
result[0:image_right.shape[0], 0:image_right.shape[1]] = image_right
```

10. View results.
11. In addition, the actual dimension of the merged images can be detected and the excess black background removed.

■

6.7 Additional exercise – SIFT

Exercise 6.5 In the following, we will use one of the most popular algorithms for feature point detection and feature descriptor - the Scale-invariant feature transform (SIFT) method.

1. Using the OpenCV implementation of the SIFT method, perform operations analogous to those in the 6.4 section (finding similar neighborhoods of feature points). To do this, call

the function *cv2.xfeatures2d.SIFT_create()* which creates an object that performs various operations on images using the SIFT method. Then use the *detectAndCompute* method of this object twice for the two images: *fontanna1.jpg* and *fontanna2.jpg*. The parameters of this method should be a grayscale image and textNone. The method returns two lists – a list of feature points and a list of their descriptions.

2. To determine the similarity of feature point descriptions, use the comparison method from OpenCV – *BFMatcher* and its method for `k` nearest neighbours (`k=2`, since we are interested in the two most similar neighborhoods according to the similarity measure – this will be used when drawing the results). For example, for lists of descriptions `desc1` and `desc2`, the list of matching points `matches` is obtained as follows:

```
bf = cv2.BFMatcher()
matches = bf.knnMatch(desc1,desc2, k=2)
```

   The list of matches thus obtained can be displayed as an image returned by the function `cv2.drawMatchesKnn` called with the following parameters: first image, feature points of the first image, second image, feature points of the second image, list of matched points (with `knnMatch`), output image (set to `None`, the output image is also returned by this function), `flags=2` (without this argument all feature points will be displayed).

3. To keep only **best matches**, one can remove from the list of `matches` before calling `drawMatchesKnn` those whose second neighbour (of knn) has a fairly similar neighborhood. This is implemented by the example loop:

```
best_matches = [ ]
for m,n in matches: # m i~n - best and second best matches
    if m.distance < 0.5*n.distance:  # the best is twice as good as the
        second
        best_matches.append([m])
```

   or the same using list comprehension:

```
best_matches = [ [m] for m,n in matches if m.distance < 0.5*n.distance]
```

4. **Repeat** the above steps for the remaining images processed in section 6.4. How SIFT performs in comparison to a patch-based description (in particular for images *fontanna1.jpg* and *fontanna_pow.jpg*)?
   If there are too many lines and the image is unreadable as a result - simply increase the requirement for how much better the best fit is than the other (i.e. **increase** a value of 0.5).

■

VII Laboratory 7

7. Object tracking – Correlation Filter

7.1 Correlation Filter

Another approach to the tracking algorithm is to use methods from the correlation filter family. In these algorithms a certain filter is used with which the object to be tracked is represented. In order to predict the new position of the object, a correlation is performed between the tracking area and the filter. The calculation is performed in the frequency domain, as it is faster to calculate the Fourier transform with the *Fast Fourier Transform* algorithm and use the convolution theorem than to calculate the correlation directly. The tracking algorithms of the correlation filter family have low computational complexity (i.e. real-time tracking is possible) while maintaining competitive tracking quality.

The simplest version of the algorithm described in the paper will be presented: Visual Object Tracking using Adaptive Correlation Filters.

In the first frame of the image, the filter is initialized based on the grayscale image patch f that contains the object to be tracked.

Figure 7.1: Wskazany obiekt do śledzenia.

$$H_t^* = \frac{A}{B} \tag{7.1}$$

$$A = \sum_i^N G_i \odot F_i^* \tag{7.2}$$

$$B = \sum_i^N F_i \odot F_i^* \tag{7.3}$$

where F_i is the discrete Fourier transform of the image patch f_i, while G is the Fourier transform of the two-dimensional Gaussian distribution g of the size of the tracked area:

Figure 7.2: Fourier transform of the two-dimensional Gaussian distribution.

The operation of *odot* denotes element-by-element multiplication, and X^* denotes complex conjugate (the result of the Fourier transform is represented in complex numbers).

The sum over i means that from one object sample in the first tracking frame, a set of N learning samples consisting of random rotations of the base sample should be created:

Figure 7.3: Learning sample.

Then, a position prediction is made in each image frame:

$$\mathscr{F}^{-\infty}\{F \odot H^*\} \tag{7.4}$$

where $\mathscr{F}^{-\infty}$ denotes the inverse discrete Fourier transform, while F here is the transform of the image patch centred on the last known position of the object. The correlation map obtained

in this way represents the displacement of the object with respect to the previous image frame, as predicted by the filter:

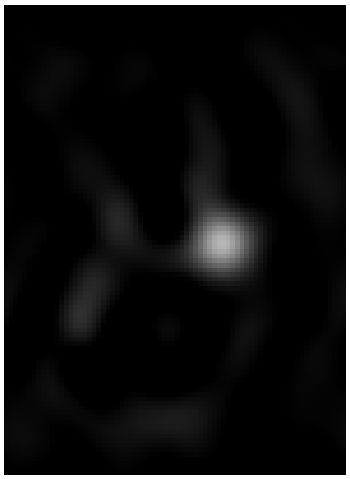


Figure 7.4: Correlation map.

After each prediction, the filter is updated with a portion of the current image frame cropped at the new object position:

$$A_t = \eta G \odot F_t^* + (1-\eta)A_{t-1} \tag{7.5}$$
$$B_t = \eta F_t \odot F_t^* + (1-\eta)B_{t-1} \tag{7.6}$$

Where $\eta \in [0,1]$ is the assumed learning rate.

7.2 Object tracking - exercise

Exercise 7.1 The purpose of the exercise is to learn a method that uses correlation filters to track objects.

1. Download the materials and the exercise template from the UPeL platform.
2. The exercise has been prepared in Jupyter notebook. The file `.ipynb` can be opened and run using Google Colab (www.colab.research.google.com). It is also possible to run the file using the VS Code editor. You will probably be required to install the appropriate libraries. If we choose VS Code we skip loading the library:

```
from google.colab import drive
drive.mount('/content/drive')
```

3. Follow the instructions in the file.
4. The solution to the task is the completed and working .ipynb file, which should be sent to the UPeL platform.

■

VIII

Laboratory 8

8. Visual Odometry

8.1 Visual Odometry

Visual Odometry is defined as the process of estimating the robot's motion (translation and rotation with respect to a reference frame) by observing a sequence of images of its environment. VO is a particular case of a technique known as Structure From Motion (SFM) that tackles the problem of 3D reconstruction of both the structure of the environment and camera poses from sequentially ordered or unordered image sets. SFM's final refinement and global optimization step of both the camera poses and the structure is computationally expensive and usually performed off-line. However, the estimation of the camera poses in VO is required to be conducted in real-time. In recent years, many VO methods have been proposed which those can be divided into monocular and stereo camera methods. These methods are then further divided into feature matching (matching features over a number of frames), feature tracking (matching features in adjacent frames) and optical flow techniques (based on the intensity of all pixels or specific regions in sequential images).

Simultaneous localization and mapping is an extension of the VO method that deals with drift, but also SLAM is a process in which a robot is required to localize itself in an unknown environment and build a map of this environment at the same time without any prior information with the aid of external sensors (or a single sensor).

Vision odometry consists of several steps:

1. Feature detection (usually ORB - FAST+BRIEF)
2. Feature matching (Brute-force or FLANN)
3. 3D point cloud generation - triangulation
4. Inliers detection, RANSAC filtering
5. Motion estimation - determination of rotation and translation

8.2 Object tracking - exercise

Exercise 8.1 The purpose of the exercise is to implement simple visual odometry, i.e. to determine the trajectory of a car moving through the streets on the basis of a KITTI dataset.

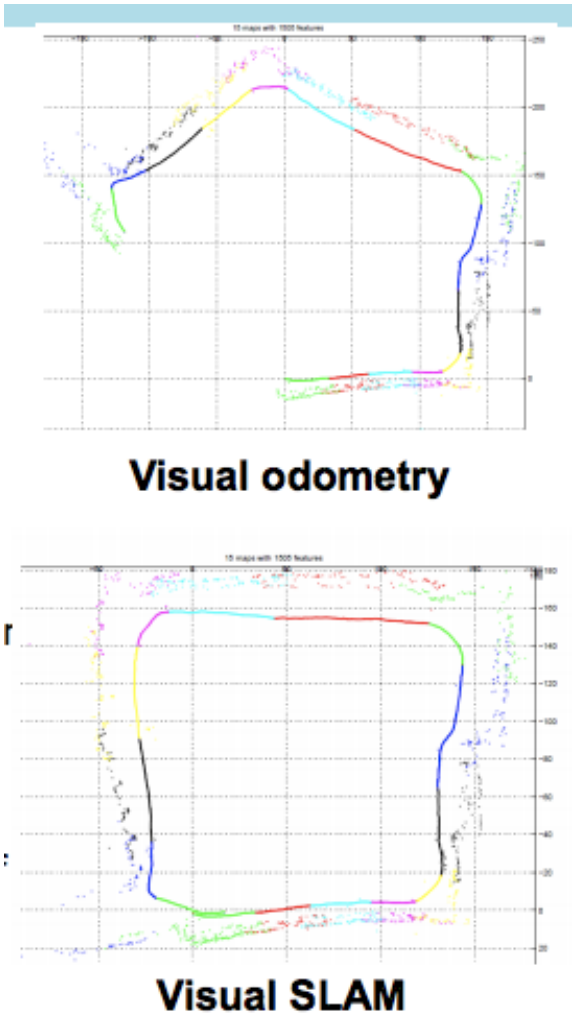


Figure 8.1: Comparison of VO and SLAM.

1. Download the materials and the exercise template from the UPeL platform.
2. The exercise has been prepared in Jupyter notebook. The file `.ipynb` can be opened and run using Google Colab (www.colab.research.google.com). It is also possible to run the file using the VS Code editor. You will probably be required to install the appropriate libraries. If we choose VS Code we skip loading the library:

```
from google.colab import drive
drive.mount('/content/drive')
```

3. Follow the instructions in the file.
4. The solution to the task is the completed and working .ipynb file, which should be sent to the UPeL platform.

■

IX

Laboratory 9

9. Generalised Hough transform

9.1 Main objectives:

- implementation of the generalised Hough transform,
- pattern search using the generalised Hough transform.

9.2 Implementation of the generalised Hough transform

Exercise 9.1 Pattern searching.

1. From the course page, download the data archive for the exercise and unzip it in your own working directory.
2. Create a new script. Based on the image with the pattern `trybik.jpg`, we will create an array `R-table`. To do this, determine the **contours** and **gradients** in the pattern image.
3. In order to **determine the contour**, one has to first perform image conversion to grayscale and binarization with an appropriate threshold. Next, use the function `cv2.findContours` to get the contours present in the image.

   R Negate the values of the image – `cv2.bitwise_not` – before sending it to the function, because it contains a black contour on a white background, and the function `findContours` expects the opposite.

   Note to get lists of all contour points – use the parameter `CHAIN_APPROX_NONE`. Suggested call:

```
contours, hierarchy = cv2.findContours(bin_img, cv2.RETR_TREE, cv2.
    CHAIN_APPROX_NONE)
```

   The result of the function is, among others, a list of contours (theoretically, there can be more objects in the image) – in our case it should be a one-element list. However, it can happen that the contours get split – then there are several solutions. First, you can shrink the object by erosion to prevent splitting. Alternatively, you can select the longest contour.

The easiest way to use when finding a contour is to use the parameter `RETR_TREE` – then the contours are arranged in a hierarchy and the longest one should be first (have index 0).

4. The resulting contour can be plotted on any image with the function:

```
cv2.drawContours(img, contours, 0, color)},
```

where: `image` is the target image, `contours` – contour list, 0 – contour number, `color` – brightness or color component tuple (depending on image type).

5. Sobel filters (for a grayscale image) can be used to **calculate gradients**, for example:

```
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)
```

We need two matrices containing the values of the gradient amplitude (this is the root of the sum of the squares of the vertical and horizontal Sobel, and the orientation of the gradient (using the function `np.arctan2`). The matrix of gradient amplitude values is worth normalizing by dividing it by its maximum value (`np.amax()`).

6. Before filling in `R-table`, **choose a reference point** – let it be the center of gravity of the pattern determined from a binarized image of the pattern using moments. The central moments of `m00`, `m10`, and `m01` (function `cv2.moments(bin, 1)`) can be used to determine the center of gravity.
7. Vectors connecting the contour/contours points to the reference point will be needed to fill the `R-table`. Write into `R-table` the lengths of these vectors and the angles they form with the OX axis (here again the function `np.arctan2` will be useful).
   The entry point of the `R-table` is determined by the orientation of the gradient at the contour point (determined from Sobel mask filtering) – **convert radians to degrees** – `R-table` will have 360 rows. We use an accuracy of 1 degree. *R-table* can be implemented as a list of 360 lists:

```
Rtable =  [[] for _ in range(360)]
```

Then, for example. `Rtable[30]` will be a list of polar coordinates of the contour points whose orientation calculated from the gradients is about 30°.

8. Using the image `trybiki2.jpg` and the `R-table` from the previous section, fill the two-dimensional Hough space – recalculate the gradient at each point and for points whose normalized gradient value exceeds 0.5, determine the orientation at that point, then increase the value by 1 in the accumulation space at the points stored in `R-table` according to the dependence:

```
x1 = -r*np.cos(fi) + x
y1 = -r*np.sin(fi) + y
```

where r,fi – values from the corresponding row in the `R-table`, x,y – coordinates of the point for which the gradient is greater than 0.5.
The figure of Hough's space is also worth displaying.

9. Search for the maximum in Hough space and mark it on the input image – `trybiki2.jpg`. Determination of the coordinates of the maximum can be done using the function `np.argmax` or by construction:

```
np.where(hough.max() == hough)
```

Where: `hough` is of type `np.array`.
In turn, a function can be used to mark on an image:

```
plt.plot([m_x], [m_y],'*', color='r'))
```

On the other hand, if you don't want to use `pyplot` then you can simply draw a circle on the image:

```
cv2.circle(I,(int(M[1]),int(M[0])),2,(0,0,255))
```

Where: I is the image, M is the coordinates of the found object, 2 – the radius, and the last parameter is the color.

10. The result of the above algorithm is to mark the found maximum on the image and overlay the pattern contour around that point. An example solution is shown in Figure 9.1, additionally in Figure 9.2 illustrates the Hough space.

■

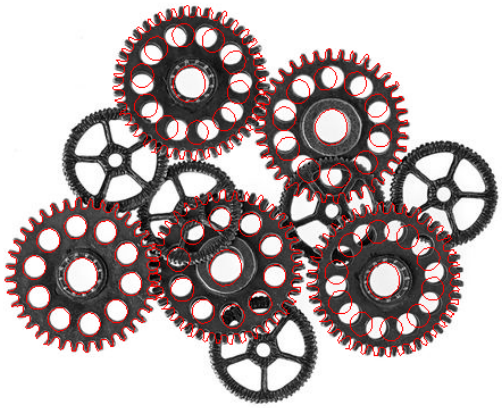


Figure 9.1: Example Solution.

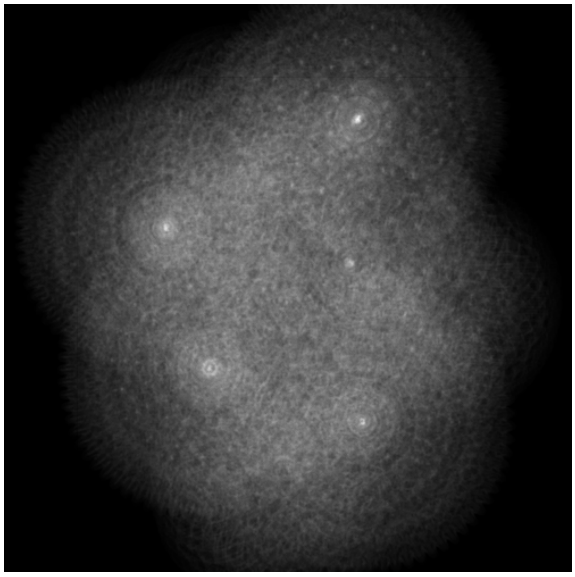


Figure 9.2: Hough space.

X

Laboratory 10

10. Thermovision

Thermal images have the advantage that objects, such as people or animals, are usually quite visible (i.e. stand out from the background). The exception is when the background temperature is close to that of the object – for example, on a very warm day. Another problematic situation is clothing that insulates well from outside conditions – for example, a down jacket.

The exercise will explore three approaches to the issue:

- simple object detection using thresholding and object analysis,
- the use of a deep convolutional neural network and fusion of RGB and thermal imaging data,
- using a probabilistic model (additional exercise).

10.1 Simple thermal image analysis

In the first stage, we will use the following algorithms for human silhouette detection:

- thresholding,
- filtration,
- labelling,
- analysis of labelling results.

Exercise 10.1 Instructions for the task:

1. From the course page, download the test sequence.
2. Loading sequences in OpenCV is relatively simple:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

cap = cv2.VideoCapture('vid1_IR.avi')

while(cap.isOpened()):
  ret, frame = cap.read()
  G = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  cv2.imshow('IR',G)
```

```
    if cv2.waitKey(1) & 0xFF == ord('q'):  # break the loop when the 'q' key
        is pressed
        break
cap.release()
```

R Instead of a file name you can give a number (e.g. 0) – then the application will try to connect to the camera (USB, built-in), of course if it is available in the system. You can also enter the address of an IP camera that works in the `rtsp` protocol.

3. Binarisation – in the first approximation, please use a fixed threshold. As a reminder: `cv2.threshold` (see documentation for details). The threshold value should be chosen experimentally, so that the silhouettes are extracted as well as possible, while avoiding noise. It should be noted right away that this cannot be done perfectly.
4. Filtering – the whole known set of operations – median filters and morphological operations – can be used.
5. Labelling – available functions in the OpenCV library `cv2.connectedComponents` and `cv2.connectedComponentsWithStats`. It is advisable to use the second version (the area, surrounding rectangle and centre of gravity are calculated at once). The order of the returned values and the type of variables should be found in the documentation.
6. Analysis of labelling results:
   (a) in the first step, display the surrounding rectangles for all objects - function `cv2.rectangle`,
   (b) filter objects based on size – objects should be larger than X (choose threshold yourself),
   (c) filter objects on the basis of shape – we are looking for vertical silhouettes (e.g. appropriate proportions of height and width of the surrounding rectangle),
   (d) The last and most difficult element is the attempt to merge silhouettes that are split. This is a common phenomenon in thermal images of people who are dressed (warmly, thickly). Exposed parts of the body such as the head, hands are the brightest and the waist and leg areas are the darkest. In this case, it is worth considering the heuristic of connecting the surrounding rectangles lying one below the other – a method to be designed independently.

   R Please describe the proposed solution as a comment in the code.
7. An example result of the algorithm is shown in Figure 10.1.

R Please assume that the implementation of this stage of the exercise should not take more than 30-35 minutes. In other words, please do not spend too much time developing an algorithm for combining silhouettes (even at the cost of its effectiveness).

10.2 Using a neural network based detector for RGB image analysis + Thermal imaging (YOLOv3)

The first approach to thermal image analysis has provided an introduction to the appearance and characteristics of such images. Thermal imaging provides a very good source of information that can serve as an additional information channel when analysing RGB images. The fusion of data from RGB and thermal images provides opportunities to improve the efficiency and accuracy of detection, which can be exploited using a detector based on a deep convolutional neural network.

When using neural networks, data fusion can be carried out in a number of ways:

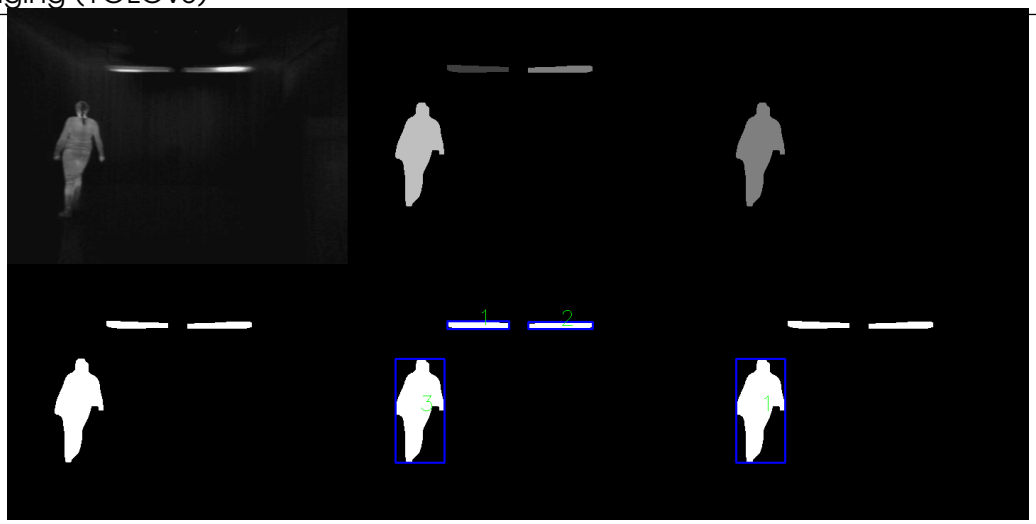


Figure 10.1: An example of the result of the algorithm.

1. `early fusion`) – image fusion at the beginning of the detection process. The thermal image and the RGB image are combined into a single image, which is then fed to the detector, which determines the rectangles surrounding the objects and their classes.
2. `middle fusion` – at the beginning, the detector processes the images separately, but at some stage in the structure of the neural network the features representing both images (feature maps) are combined and finally at the output of the network we get uniquely determined surrounding rectangles with their corresponding classes.
3. `late fusion` – in this case, the two detectors receive one image as input. At their output, separate results are obtained for each detector, which then need to be combined. This can be done by finding corresponding surrounding rectangles and averaging their parameters, such as position or size.

This exercise will use and compare approaches based on early and late fusion. In order to perform this task, it is necessary to download a set of images from the test base, the program template, as well as all files containing the weights and configurations of the learned YOLOv3 detectors, which are available for download on the UPeL platform.

Exercise 10.2 How to run the neural network.

1. At the very beginning, the paths to the folders `test_rgb` and `test_thermal` must be set in the program template (file `thermo_detection.py` available on the UPeL platform). (section `TODO0`).
2. When executing individual fusion methods, set the variable `FUSION = "EARLY"`, or `FUSION = "LATE"` in the script in the `METHOD` section.
3. Early fusion will be realised by adding the thermal image to the red channel in the RGB image. This will be implemented as follows. In OpenCV the channels of the image are loaded by default in BGR order, so the first two channels will be left unchanged and the value of the third channel will be for each pixel the maximum value from the value of the R channel and the thermal image converted to grayscale (OpenCV loads a 3 channel image by default, so we choose one arbitrary channel). The merging of the images should be done in the textTODO1 section.
4. Late fusion will be realised by loading colour and thermal images, which will be fed to two separate detectors (in this case thermal imaging will also be converted to greyscale,

as the detector that implements detection on the thermal image has been set up to have one input channel).

5. Then, corresponding surrounding rectangles will be found for both detections, whose parameters will be averaged to finally obtain one surrounding rectangle for one pedestrian detected in both images. In order to find the corresponding surrounding rectangles, the coefficient `IoU` (*Intersection over Union*) will be used – Figure 10.2. This is a coefficient calculated by dividing the area of the common part of the two surrounding rectangles by the total area of the figure formed by their union, as shown in the image below. The function that implements IoU, has been placed in the code, it will be necessary to determine the corresponding rectangles, which should be implemented in the section `TODO2`.
6. In order to handle all possible situations including the detection by a given detector of several frames one inside the other, the following will show how this task is implemented.
7. The input will give us two lists of surrounding rectangles: `Rect1` and `Rect2`. Each rectangle is also stored as a list (4 elements: coordinates of the top left vertex, width and height).
8. First we need to determine the IoU value for each pair of surrounding rectangles, one rectangle from each list per pair. We can refer to the surrounding rectangles by indices, e.g. a pair of 3 rectangles from the first list and 4 rectangles from the second list can be written as a tuple `(3, 4)`.
9. For each tuple we determine the IoU value, to do this we can create a list (let's call it `boxes_iou`) which will contain lists consisting of the tuple and the IoU value for each pair for which this value is greater than 0 (if the IoU value of a pair is 0, we do not add it to the list). It will look like this `[[(3, 4), 0.75], [(2,3), 0.43], ...]`.
10. We sort the resulting list, in descending order, by the value of IoU, which can be done using the function `sorted` with the parameters `reverse=True` and `key = lambda x : x[1]`.
11. The next step is to explicitly choose which pairs of rectangles will be merged, to do this we create lists (`Rect1_paired` and `Rect2_paired`) in which we will record which of the surrounding rectangles already have pairs. We will write the indices of the surrounding rectangles into them.
12. We then iterate over the array `boxes` and if none of the rectangles in the pair are paired, we save such a pair to a new list `paired_boxes` and add the corresponding indexes to the lists `Rect1_paired` and `Rect2_paired`. Otherwise, we skip the pair and iterate further.
13. With the pair indices, we return to the lists containing the coordinates of the surrounding rectangles and determine the average of each element, then convert to integer. We save the connected rectangles to the list `boxes`.
14. Then we follow the directions in the code – given a solution template to complete.
15. Finally, we run the program and compare the performance of the two detection methods.

■

10.3 Additional exercise – Object detection using a probabilistic pattern

Another approach to pedestrian detection is to observe that the silhouettes of standing people have a fairly distinctive and repeatable appearance. This makes it possible to create a silhouette pattern and then search for it in the image using the sliding window technique.

Exercise 10.3 Additional exercise – probabilistic pattern.

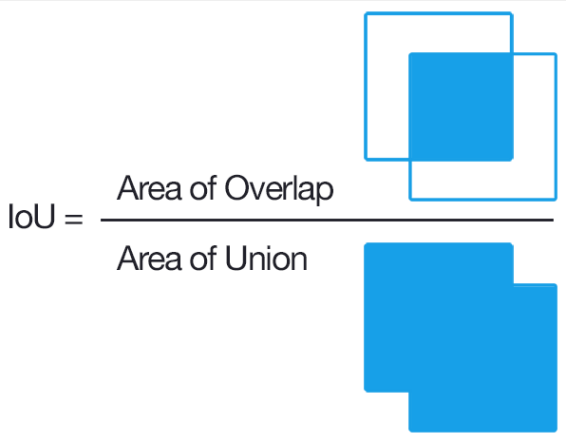


Figure 10.2: *Intersection over Union*. Source: https://en.wikipedia.org

A. Saving human silhouettes to a .png file.

In the first step, you will need a database of silhouette snippets with specific sizes – 192x64 px. To get them, we will use the application from the first part of the exercise – let's add to it the saving to the *png* file of the verified silhouettes.

1. We implement the slice base by using ROI:

```
ROI = G[y1:y2, x1:x2]
```

2. We save the extracted ROI:

```
cv2.imwrite('sample_%06d.png' % iPedestrian,ROI)
```

iPedestrian is a global counter – to be incremented every write,

3. We manually select about 30 – 50 images. In general they should be frontal, but photos taken from the side are also acceptable.

B. Creating a silhouette pattern.

1. In the next step, in a separate script, the selected images should be loaded, resized and the pattern determined. Scaling is done to 192×64 pixels using the function `cv2.resize`.
2. Change the assigned output value from 255 to 1. If necessary, filtering may also be considered.
3. Please check the correctness of the silhouettes you have received.
4. Finally, the probabilistic pattern is formed by the rule $PDM = PDM + B$ (sum of binary images), where B must be binary. We normalise the resulting pattern by dividing by the number of images.
5. The resulting pattern is saved to file. The pattern should look similar to the one in Figure 10.3.

C. Object detection.

1. we perform the loading of the sequence (as in the first step), but at the beginning it is best to practice the method on a selected frame. An example is provided on the course page

and shown in Fig. 10.4, others can be obtained by saving the frames to the file *png* – stage 1.

2. Load the pattern (you may need to convert it to grayscale). Convert to type *float32*, divide by the number of images from which it was created (obtain scaling to the range 0–1). We denote the result of the division as PDM_1 and calculate the negation $PDM_0 = (1 - PDM_1)$ i.e. the probability of belonging to the background.
3. We binarise the image from the test sequence. We then run the analysis in a 192x64 window (sliding window technique). We scan consecutive parts of the image with the window (a step larger than 1 can be considered). For the binary mask (B) we count the probability that it is a human silhouette:

$$\sum_y \sum_x (B(y,x) * PDM_1(y,x) + (1 - B(y,x)) * PDM_0(y,x)) \tag{10.1}$$

   We store the result in an auxiliary image: `result = np.zeros((360,480), np.float32)`.
4. To visualise the results correctly, it is necessary to normalise:

```
result = result / np.max(np.max(result))
result_uint8 = np.uint8(result*255)
```

5. The last stage is finding local maxima. The simplest solution is to look for a maximum in each iteration – during the analysis of individual windows. However, this approach only finds one maximum. Please check and draw a surrounding rectangle for the character found.
6. In further development of your application, please develop a method for finding multiple maxima. Two things should be kept in mind when doing so:
   (a) set a lower threshold,
   (b) apply the elimination of areas already visited (this is often referred to as *Non-Maximal Suppression*).
7. To be considered independently. The implemented solution allows detection of objects only in one scale (characters must be approximately 192x64 pixels in size). What should be done to enable detection of objects of other sizes as well ? Write your thoughts and solutions to the problem as a comment in the code or implement a proper solution (it is neither difficult nor time consuming, although it may significantly increase the computation time).

Figure 10.3: Example pattern.

Figure 10.4: Example frame.

XI

Laboratory 11

11. Human silhouette detection

11.1 Main objectives:

- familiarisation with the detection of human silhouettes,
- familiarisation with the Histogram of Oriented Gradients descriptor (HOG),
- familiarisation with the Support Vector Machine classifier(SVM),
- basics of machine learning (methodology).

11.2 Histogram of Oriented Gradients – description

Histogram of oriented gradients (HOG)[1] is a feature descriptor widely used in image processing and image recognition. It describes the local appearance and shape of an object through the distribution of local intensity gradients and edge directions. The HOG algorithm can be divided into several steps:

1. Gradient calculation,
2. Computation of histogram of gradient orientations in cells,
3. Normalization of histograms in blocks of cells,
4. Determination of the feature vector.

Gradient calculation

For each pixel, a horizontal and vertical gradient is calculated using two one-dimensional masks: $\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}^T$. As a result, two components of the gradient are obtained, with which its amplitude and orientation can be determined.

For a grayscale image, the amplitude of the gradient is calculated based on the formula (11.1):

$$m(x,y) = \sqrt{G^x(x,y)^2 + G^y(x,y)^2} \tag{11.1}$$

where:

(x,y) – pixel coordinates,

[1] Dalal, Navneet, and Bill Triggs. "Histograms of oriented gradients for human detection." 2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05). Vol. 1. Ieee, 2005.

$G^x(x,y)$ – horizontal component of the gradient,
$G^y(x,y)$ – vertical component of the gradient,
$m(x,y)$ – gradient amplitude.

When processing a colour image, the gradient amplitude for each channel of the RGB colour space is determined and the highest value selected – formula (11.2).

$$m(x,y) = max(m_R(x,y), m_G(x,y), m_B(x,y)) \tag{11.2}$$

where:
$m_R(x,y), m_G(x,y), m_B(x,y)$ – the amplitude of the gradient for each RGB channel.

The orientation of the gradient is calculated from the formula (11.3). When the detector window contains pixels in RGB colour space, in each cell the gradient angle is determined only for the gradient components for which the amplitude is the highest.

$$\theta(x,y) = \arctan\left(\frac{G^y_{max}(x,y)}{G^x_{max}(x,y)}\right) \tag{11.3}$$

where:
$\theta(x,y)$ – gradient orientation,
$G^y_{max}(x,y), G^x_{max}(x,y)$ – gradient components for the highest amplitude among the gradients of individual RGB channels.

Histogram calculation

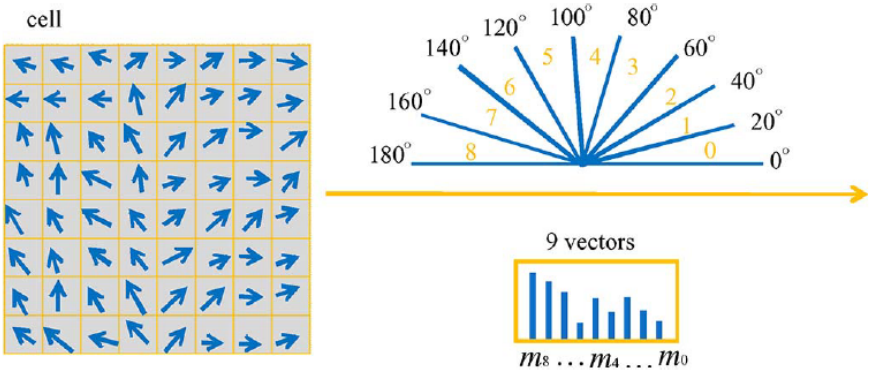


Figure 11.1: View of one cell of 8×8 pixels in the detection frame. Distribution of histogram intervals in the range 0° to 180° and an example histogram.

The detector window is divided into non-overlapping cells. The most commonly used cell size is 8×8 pixels due to easy hardware implementation. For each cell, a histogram consisting of evenly spaced intervals is determined, representing gradient orientations ranging from 0° to 180° or 0° to 360°. In fig. 11.1) an example of the histogram most commonly used in pedestrian detection is given. In the histogram, the amplitude values of the gradients are accumulated on the basis of the direction and sometimes also the return in individual intervals. The adjacency of an individual gradient is accurately determined by the equation (11.3). To compensate for aliasing, i.e. different assignments of gradient values to two adjacent intervals with a small change in gradient orientation,

linear interpolation was used. Consequently, the amplitude of the gradient is linearly assigned to adjacent intervals according to the formulas (11.4) and (11.5).

$$\upsilon_{UB}(x,y) = m(x,y)\left(\frac{\theta(x,y)-\theta_{LB}}{BW}\right) \tag{11.4}$$

$$\upsilon_{LB}(x,y) = m(x,y)\left(\frac{\theta_{UB}-\theta(x,y)}{BW}\right) \tag{11.5}$$

where:

$\upsilon_{UB}(x,y), \upsilon_{LB}(x,y)$ – amplitude value coming into one of two adjacent predefined intervals of the histogram,

BW – the width of the histogram interval,

$m(x,y)$ – gradient amplitude,

$\theta(x,y)$ – gradient orientation,

θ_{LB} – the centre of the lower bin of the histogram,

θ_{UB} – the centre of the upper bin of the histogram.

Normalization of histograms and determination of the feature vector

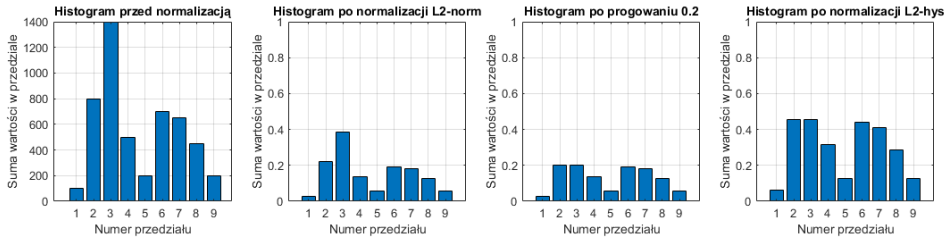


Figure 11.2: An example of a histogram normalisation procedure.

The norm of the gradient within the detection window can vary significantly. For this reason, local normalization of histograms is introduced. Each block consists of four contiguous cells in the form 2×2. The areas of consecutive blocks overlap in 50 in both directions (vertical and horizontal). This ensures that edges close to the edges of the cells are given the same consideration as those in the centre. The resulting histograms are grouped into a 36 element vector within one block. To reduce the influence of uneven illumination in the detection frame, double normalisation was applied: L2 (formula (11.6)) and L2-hys (formula (11.7)). In Fig. (11.2) an example is shown.

In the first stage, the feature vector is reduced to a value between $[0;1]$. In this way, the various contrasts present in the image are eliminated. Additional noise may appear on the image in the form of variable illumination caused by the mapping of colour intensity by the camera and reflections from various types of surfaces. As a result, the amplitudes of some parts of the image are significantly higher than in other parts. To reduce the influence of these areas on the shape of the whole histogram, upper thresholding is used. Vector values above 0.2 are truncated and aligned to this number. The vector is then re-normalized to the interval $[0;1]$. As a result, the orientation of the gradient is more influential than its amplitude. The value of the threshold was determined experimentally in work D. Lowe[2].

[2]David G Lowe. „Distinctive image features from scale-invariant keypoints". W: International journal of computer vision 60.2 (2004), s. 91–110.

- norm L2:

$$f_1 = \frac{f}{\sqrt{\|f\|_2^2 + \varepsilon^2}} \tag{11.6}$$

- norm L2-hys:

$$f_2 = \frac{f_{th}}{\sqrt{\|f_{th}\|_2^2 + \varepsilon^2}}, f_{th} = \min(f_1, 0.2) \tag{11.7}$$

where:

f – unnormalised feature vector,
f_1 – normalised feature vector (norm L2),
f_2 – normalised feature vector (norm L2-hys),
f_{th} – feature vector after thresholding,
$\|f\|_k$ – k-th norm of the vector f,
ε – an appropriate constant (avoidance of division by 0).

Finally, the vectors created in each block are concatenated into a single one-dimensional vector, which reaches a size of $7 \times 15 \times 36 = 3780$ values. This is then passed to the classifier to assign the object to the appropriate class.

11.3 Support Vector Machines – description

The aim of the SVM algorithm is to find a (hyper)plane in a multidimensional space defined by features, derived from a descriptor, which is maximally distant from all points of both classes. It is described by the following formula (11.8) and illustrated by the example in Fig. 11.3.

$$f(x) = w^T x + b = 0 \tag{11.8}$$

where:
x – feature vector,
w – vector of weights, hyperplane parameters,
b – bias.

There are two versions of the SVM algorithm:
- *soft-marigin* – a classifier with a soft margin that allows for misclassification of parts of the data,
- *hard-marigin* – the classifier unambiguously separates the data from the two classes.

However, due to the appearing computational inaccuracies and deformations in the image, feature sets are usually not linearly separable. For this reason, a penalty function with a constant coefficient C is introduced into the optimisation task, which allows a certain group of points to be misclassified. This makes it possible to find a plane that separates the significantly different features of an object, and the remaining points, close to the separation boundary, will not be affected. The introduction of a weakened classifier or in other words a soft margin classifier has an additional advantage. This algorithm is not sensitive to data far from the point of densest clustering and it follows that it generalizes the problem better and is resistant to over-fitting. The important issue is to find such a coefficient of C to obtain the lowest percentage of misclassified test samples. Most

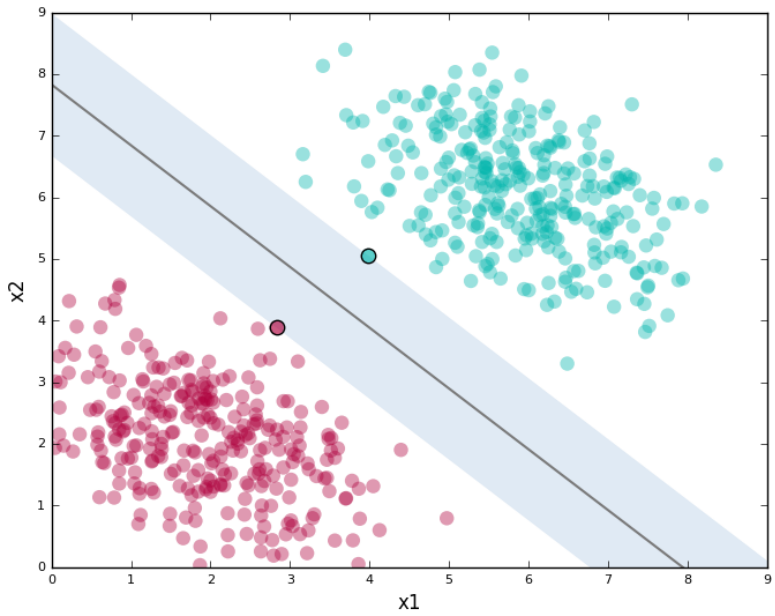


Figure 11.3: Representation of a one-dimensional hyperplane separating samples of two classes for a hard margin classifier.

often cross-validation is used to determine the best value. The larger the parameter, the narrower the margin of the classifier. In Fig. 11.4 shows the dependence of the value of C on the width of the classifier margin.

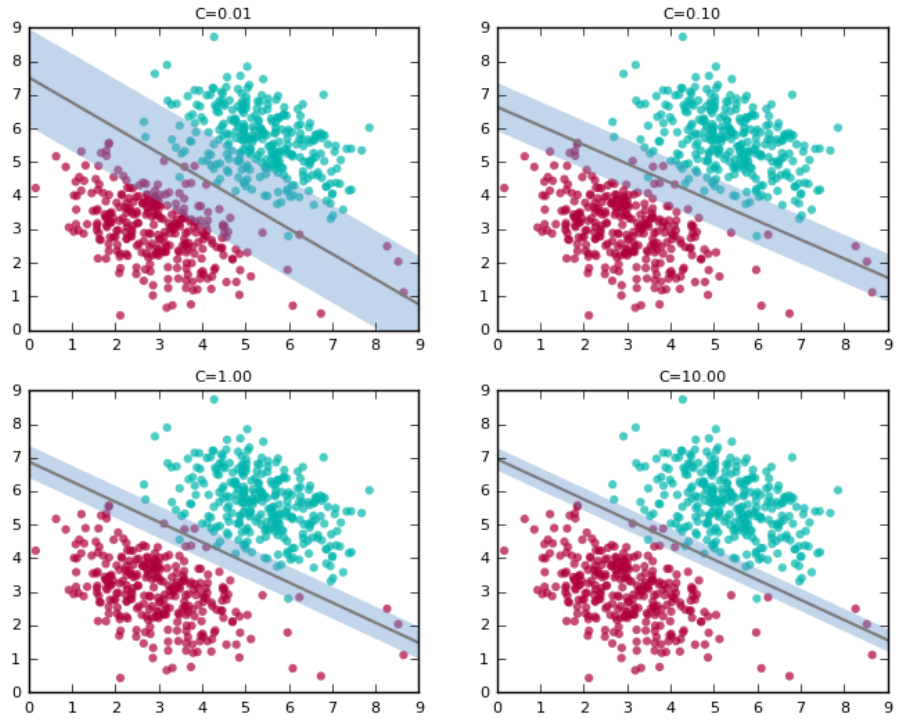


Figure 11.4: Dependence of the parameter C as a function of penalty on margin width.

By solving the equation (11.9), a hyperplane is obtained that best generalizes the pedestrian

classification problem. The set of detection windows, derived from a single image frame, is expressed by the set of pairs: $(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)$, where x_i is the feature vector describing the detection window and $y_i \in \{-1,1\}$ are class labels (non-pedestrian, pedestrian).

$$\min_{w,b,\xi}\left(\frac{1}{2}w^T w + C\sum_j \xi_j\right) \tag{11.9}$$

on the assumption that:

$$\left(w^T x_i + b\right) y_i \geq 1-\xi_i, \quad \xi_i \geq 0, \quad (i=1,\ldots,n) \tag{11.10}$$

where:

w – vector of weights, hyperplane parameters,
C – a constant factor for the penalty function,
ξ_i – slack variable, distance of a point (x_i,y_i) from the margin on the corresponding side of the hyperplane.

The parameter ξ_i reduces the separation margin of the samples. There are three intervals that define the position of a given sample relative to the hyperplane. If the condition $0 \leq \xi_i < 1$ is true then the point (x_i,y_i) lies on the correct side of the hyperplane, the classification will be correct. On the other hand, if $1 < \xi_i$ is true then point (x_i,y_i) lies on the wrong side, so the class membership will be incorrectly determined. If the value of ξ_i equals 1 then it is not possible to correctly assign a label to the sample.

R It is worth mentioning again that the above operations are performed iteratively for each image frame (64×128 px) using the *sliding window* method. The list of all parameters of the *HOG+SVM* algorithm is as follows:
bins – the number of bins in the histogram (usually 9, in 20°) steps,
cell – the number of pixels in the cell (usually 8×8 px),
block – number of cells in the block (usually 4×4 cells),
overlap – the number of cells common to two neighbouring blocks or the percentage of overlap between blocks (2 cells, 50%),
frame_width – detection frame size (usually 64×128 px),
svm_weights – the number of SVM classifier weights (3780 for 64×128 px),
bias_weight – bias value of SVM classifier,
stride – number of pixels per which the detection frame is moved (usually 8 or a multiple for the frame 64×128 px),

11.4 HOG descriptor

In a very simple way, the algorithm for feature detection can be divided into three steps:
- calculation of gradients,
- determination of histograms of gradients,
- normalisation of histograms.

At the beginning, please pay attention to the parameters of the algorithm:
- colour space – RGB,
- no gamma correction (or other preprocessing),
- form of gradient operator,
- number of histogram bins, orientations analysed,

- cell size,
- norm type – instead of *norm L2-Hys*, we will use *norm L2*,
- block size, cell overlap – stride,
- detection window size.

Exercise 11.1 The aim of the exercise is to implement the HOG descriptor.

1. From the course page, download a set of image snippets with silhouettes of people (positive samples) and without people (negative samples).
2. Load any positive image – we will develop a HOG implementation for it. You can implement the above three steps in separate functions.
3. We start by calculating the gradient. Vertical and horizontal edge detection should be realised and the amplitude and orientation (in degrees) in each RGB channel should be counted separately. It is best to adapt the function to both grayscale and RGB images. A function can be used to calculate gradients with a mask (from the 11.2 section):

```
dx = scipy.ndimage.filters.convolve1d(np.int32(im), np.array([-1, 0, 1]),1)
dy = scipy.ndimage.filters.convolve1d(np.int32(im), np.array([-1, 0, 1]),0)
```

   After calculating the gradients for the three components, select the one with the largest value from them. This can be done in a loop, but it will be inefficient. It is better to use Python's capabilities where values that fulfil a condition are selected from an array. For example, use the function `np.logical_and()`:

```
max_B = np.logical_and(magnitude[:, :, 1] < magnitude[:, :, 0], magnitude
    [:, :, 2] < magnitude[:, :, 0])
```

   We then analogously find the highest gradients from the remaining components. By selecting the highest gradients from the individual components, we obtain an array containing the maximum values of the gradients. The same mechanism should be used to determine the orientation matrix of the gradients.
4. The second stage – determining histograms in blocks is the most difficult part of the implementation (technically, because conceptually it is quite simple). First, you need to define the cell size (8) and determine the number of cells in the X and Y axis (e.g. `YY_cell= np.int32(YY/cellSize)`). We then create a container for the histograms – one histogram for each cell. We loop through the cells and perform the following operations:
   - We retrieve the matrix values with amplitude and orientation for the corresponding cell.
   - Firstly, you need to handle the case of negative angles (by adding 180 if you are considering values in degrees).
   - Secondly, we need to determine to which interval of the histogram the angle belongs. For this purpose, the interval $[0;180]$ is divided into 9 parts, and for each of them the middle element should be determined. Then we need to determine to which interval each gradient belongs and to which part of the interval, e.g. for the interval $0-20^\circ$, the middle element is 10. So the gradient with orientation 11° belongs to the interval $0-20^\circ$ and the subinterval $10-20^\circ$. This is important because of the necessary use of bilinear interpolation between two adjacent intervals. **Please note** the special treatment of the transition $180^\circ - 0^\circ$ (wrapping).
   - The correct result at this stage is to get the corresponding interval number (ideally assumed to be the lower interval). Another solution is to store the compartment

numbers for the lower and upper interval.

- Then we need to calculate the distance from the angle to the centre of the intervals according to the formulas (11.4) and (11.5). Here again, remember to handle values close to 180° in a special way.
- Finally, we add the amplitude values from the cell to the corresponding intervals of the histogram.

The final step is to normalise the cells in the blocks. Loop through the cell coordinates (size minus 1) to retrieve 4 histograms belonging to one block. These should then be concatenated into a single vector (`np.concatenate`), the norm calculated (function `np.linalg.norm`) and normalized using L2. The normalized vector is part of the final feature vector.

```
# Normalization in block
e = math.pow(0.00001,2)
F = []
for jj in range(0,YY_cell-1):
    for ii in range(0,XX_cell-1):
        H0 =  hist[jj,ii,:]
        H1 =  hist[jj,ii+1,:]
        H2 =  hist[jj+1,ii,:]
        H3 =  hist[jj+1,ii+1,:]
        H = np.concatenate((H0, H1, H2, H3))
        n = np.linalg.norm(H)
        Hn = H/np.sqrt(math.pow(n,2)+e)
        F = np.concatenate((F,Hn))
```

In the end we should get a vector of 3780 numbers. The first 10 values for the image *per00060.ppm*:

0.34384495
0.15958051
0.10338961
0.13061814
0.13671541
0.09076598
0.09694051
0.19526622
0.19388454
0.16014419

You can also use the code below to display the histogram gradients (before normalising them).

```
def HOGpicture(w, bs): # w - histograms, bs - cell size (8)
    bim1 = np.zeros((bs, bs))
    bim1[np.round(bs//2):np.round(bs//2)+1,:] = 1;
    bim = np.zeros(bim1.shape+(9,));
    bim[:,:,0] = bim1;
    for i in range(0,9):  #2:9,
      bim[:,:,i] = scipy.misc.imrotate(bim1, -i*20,'nearest')/255

    Y,X,Z = w.shape
    w[w < 0] = 0;
    im = np.zeros((bs*Y, bs*X));
    for i in range(Y):
      iisl = (i)*bs
      iisu = (i+1)*bs
```

```
      for j in range(X):
        jjsl = j*bs
        jjsu=(j+1)*bs
        for k in range(9):
          im[iisl:iisu,jjsl:jjsu] += bim[:,:,k] * w[i,j,k];
    return im
```

Finally, please convert the created code (convenient at the implementation stage) into a function, whose argument is the image (matrix), and return the determined vector of features (normalized gradient histograms).

■

11.5 SVM classifier

The determined feature vectors using the HOG descriptor should then be classified accordingly in order to assess whether or not there is an object (pedestrian) in the analysed image frame.

Exercise 11.2 The aim of this exercise is to implement an SVM classifier and its learning.

1. We start by preparing the dataset. From the course page, download the `pedestrians` folder. It contains positive (with pedestrians) and negative (without pedestrians) samples.
2. For the images in both sets, we calculate the HOG descriptor (for the first 100).

```
HOG_data = np.zeros([2*100,3781],np.float32)

for i in range(0,100):
    IP = cv2.imread('pos/per%05d.ppm' % (i+1))
    IN = cv2.imread('neg/neg%05d.png' % (i+1))
    F = hog(IP)
    HOG_data[i,0] = 1;
    HOG_data[i,1:] = F;
    F = hog(IN)
    HOG_data[i+100,0] = 0;
    HOG_data[i+100,1:] = F;
```

   The data contains a class index (1 – pedestrian, 0 – non pedestrian) in addition to the vector.
3. Before learning, we separate the input data into a vector of labels (first column) and data.

```
  labels = HOG_data[:,0];
  data = HOG_data[:,1:]
```

   We include the module `from sklearn import svm`.
   We create a linear SVM object: `clf = svm.SVC(kernel='linear', C = 1.0)`.
   We carry out learning: `clf.fit(data, labels)`.
   We test: `lp = clf.predict(data)`.
4. We analyse the results of learning. Classification is performed on the learning set. We determine the confusion matrix by counting the coefficients TP, TN, FP, FN. The results of good learning should be close to 100 %.
5. We carry out a validation of the solution. We basically repeat the above experiment. This is the moment to make any corrections:
   - change of parameter C,
   - change the kernel, etc.

   Of course, after making changes, you should always check their effect on the test and validation sets.

6. Finally, once we are sure that a particular version of the classifier achieves the best results, we run a check on the test set. We consider its results as final for the classifier. This is at least methodologically correct.

■

11.6 Human silhouette detection

At this point, we already have all the tools to implement a complete detection system: the HOG descriptor and the learned SVM classifier. We can therefore implement the detection on the real image.

Exercise 11.3 Pedestrian silhouette detection using a HOG descriptor and a learned SVM classifier.

1. Four test images are provided on the course page: (`testImage1-4.png`). But you can just as well use any images you like.
2. At first, it is worth checking if the height of the person in the image roughly corresponds to the size of our detection window. This can be done manually (measure the height in pixels) or automatically (draw the detection window on the image). The second method is better as we will still need the rectangle drawing function on the image (to visualize the detection). As a reminder *cv2.rectangle*.
3. Next we need to adjust the size of the image (function `cv2.resize`) to the height of the figure. It is worth browsing for ourselves to see what the silhouettes look like in the learning set – this will give us an idea of "how the classifier might have learned".
4. We implement a loop over the image. In each iteration we take a patch of size 64×128, calculate the HOG descriptor for it, and then do the classification. If it is positive then we apply a rectangle to the image that is copy to be displayed (we make a copy using the copy method).

   R The step (i.e. how much we move the window vertically and horizontally) should not be 1, because we may not get the result. It is best to choose it empirically, but values of 8 or 16 seem appropriate.

5. Try if the detection works on the attached images.
6. **Optional**. Multi-scale processing.
   Previously we selected the appropriate scale empirically/experimentally (manually). Now we process the image at multiple scales. For example, we start with a resolution of 1 and each time decrease it by 10 % or set the scaling factor to 1.2 or 1.5. The pyramid construction method should be chosen empirically. At each scale, we implement the detection.

   R It is important to realise that it is possible to build a pyramid in which the resolution of the input image is increased. This allows the detection of small silhouettes (although the effectiveness of this approach is not very high).

   R For testing, please choose a relatively small image with two silhouettes of different sizes. For example `testImage4.png`. However, please do not expect completely error-free operation of the system. Primarily because the set of negative samples was not very large.

7. Finally, the issue of visualisation and possibly integration of detections needs to be

resolved. It is best to memorise the coordinates of the windows where silhouettes have been detected, scale them to base resolution and plot them.

8. In addition, Non-maximum suppression (NMS) methods are used to eliminate multiple detections of the same silhouette.

■

XII

Laboratory 12

12. Object tracking – Siamese neural network

12.1 Tracking with Siamese neural network

A Siamese network is a Y-shaped network with two branches joined to produce a single output. Irrespective of the branches' structure, a Siamese network can be considered as a similarity function that measures the resemblance between two inputs. We can define the basic Siamese architecture using Equation (12.1), where ϕ stands for extracting deep features (e.g. via network's branches), z represents the template (e.g. exemplar, object to track), x represents the patch that is compared to the exemplar (e.g. search region, ROI) and γ is a cross-correlation.

$$y = \gamma(\phi(z), \phi(x)) \tag{12.1}$$

Its scheme is presented in Fig. 12.1. The input z represents the tracked object selected in the first frame and scaled to 127×127 pixels. In the next frames, the ROI around the previous object's location is selected and presented to the network as the input x, scaled to 255×255 pixels. Both patches (exemplar and ROI) result in two blocks of feature maps, of sizes $17 \times 17 \times 32$ (for z) and $49 \times 49 \times 32$ (for x) respectively. Finally they are cross-correlated and a heat map of size 33×33 is obtained. Its highest peak indicates the object's location. The ROI is selected approximately 4 times greater than object's size, over five scales to handle object's size variations.

The original description of the algorithm can be found at: Fully-Convolutional Siamese Networks for Object Tracking

12.2 Object tracking - exercise

Exercise 12.1 The purpose of the exercise is to learn about the architecture of a Siamese neural network for object tracking. This is an example in which a neural network will be used.

1. Download the materials and the exercise template from the UPeL platform.
2. The exercise has been prepared in Jupyter notebook. The file `.ipynb` can be opened and run using Google Colab (www.colab.research.google.com). It is also possible to run the

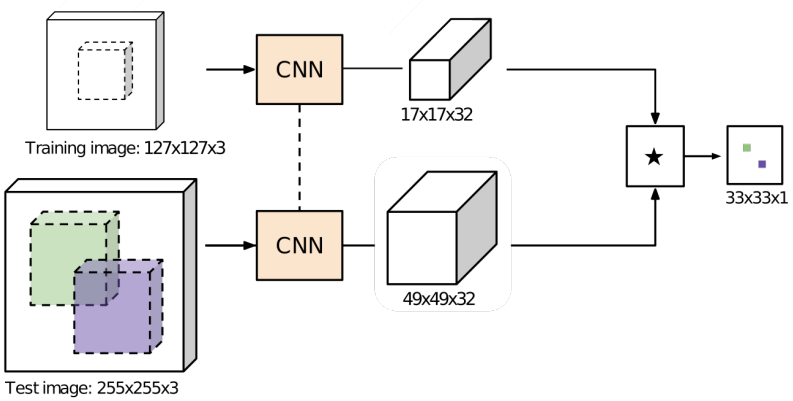


Figure 12.1: A fully-convolutional Siamese Network for tracking

file using the VS Code editor. You will probably be required to install the appropriate libraries. If we choose VS Code we skip loading the library:

```
from google.colab import drive
drive.mount('/content/drive')
```

3. Follow the instructions in the file.
4. The solution to the task is the completed and working .ipynb file, which should be sent to the UPeL platform.

■

XIII

Laboratory 13

13. Dynamic Vision Sensors

13.1 Theoretical part

13.1.1 What are event cameras?

An event camera (also known as Dynamic Vision Sensor – DVS) is a neuromorphic sensor that takes its inspiration from the human eye. Unlike classical cameras, which record the brightness (colour) level for a given pixel every specified time interval (frame per second parameter), a DVS records brightness changes independently (asynchronously) for individual pixels. Consequently, the data captured by the camera does not depend on the clock but the dynamics of the scene. As a result, a stream of events is available on the output, where each is described by 4 values:

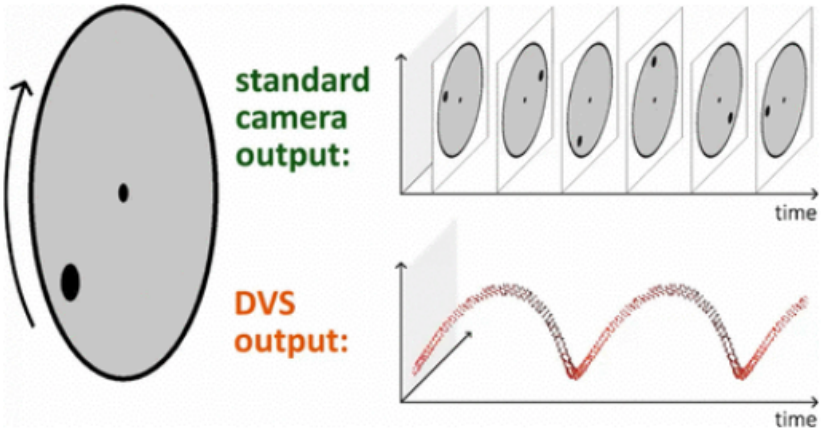
$$e = \{t, x, y, p\} \tag{13.1}$$

where: t is the event capture time (timestamp), x and y are the pixel coordinates, and p is the value 0 or 1 corresponding to the event polarity (positive or negative change in pixel value reaching a fixed threshold).

13.1.2 Why do We use event cameras?

Such an approach has a number of interesting consequences. First, a single pixel consists of an analogue and a digital part. Changes are detected in the analogue part, which results in low latency (fast response time) and energy efficiency (insufficient change means that the digital part will not be triggered). In addition, the analysis of the logarithm of the change in brightness of each particular pixel independently allows one to obtain a high dynamic range (above 120 dB). Consequently, DVS sensors are relatively resistant to large differences in the brightness of the recorded scene, which for classical cameras becomes a problem (part of the image is overexposed or too dark). This property also enables the correct detection of objects even under very high or limited light conditions.

Second, the transmission of information only about the occurring changes eliminates the usually unwanted redundancy (a traditional camera sends information about the status of all pixels, even those whose brightness has not changed). Another characteristic of event cameras is their high sampling rate – the timestamp has a resolution of microseconds. In addition, the camera



Mueggler, Elias, Basil Huber, and Davide Scaramuzza. "Event-based, 6-DOF pose tracking for high-speed maneuvers." *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014.

Figure 13.1: Visualization of how the event camera works

data stream can reach up to 1200 MEPS (million events per second) depending on the chip and hardware interface used. The limitation of redundant information and the high frequency enable the processing of key information even under conditions of high variability, i.e. when the objects registered by the camera move in relation to it at high speed.

13.2 Understanding of event data

During this class we will use Python and DAVIS 240C dataset. Read about it on this website: `https://rpg.ifi.uzh.ch/davis_data.html`. You can download dataset yourself with this link: `https://rpg.ifi.uzh.ch/datasets/davis/shapes_rotation.zip`.

Exercise 13.1 Visualising Event Data.

1. Open (extract) the dataset.
2. Examine images from `images` folder – this is a scene recorded with an event camera.
3. Open the `events.txt` file, analyze the data format. What are the particular data values ?
4. Create a Python file (use Visual Studio Code). Save it to the same folder, where the `events.txt` is located.
5. Read `events.txt` and save it to a variable (if you do not know how, check "google").
6. Now we will analyse the file line by line:
   - Each file line should be saved as a separate list member.
   - Event values (timestamp, coordinates and polarity) should be splitted into vectors (`split` function could prove useful).

   R Save only those events, where timestamp < 1. Saving all events would take a lot of time!

   To achieve this, there are different options possible. Check in "google" how the function `readline` works. It could be used in a `while` loop to iterate over the whole file. Use the `split` function to separate the column is a given line. To limit the number of samples, check if the timestamp is > 1 (if so – break the loop). In the end we should have a list o events.

7. In the next step we should split the events into particular variables: timestamp, x, y and polarity. Create 4 empty lists, do a for loop over the created events and extract the information. Again. If you do not know how – check how to iterate over a list in Python.
8. Analyze the events:
   - Print number of events.
   - Print first and last timestamp.
   - Print maximum and minimum values of pixel coordinates (x and y, respectively) and compare it to image resolution of 240x180.
   - Which events are there more of? Those with positive or negative polarity?

   Use the print function. Provide a description for each filed. If you do not know .... :) For the polarity you can sum the the whole list to get the number of positive ones and for the negatives subtract the number of positive from the total.
9. Visualize event data with 3D chart:
   - Chart dimensions should be x,y coordinates and timestamp.
   - Positive and negative events should have different colors.
   - The chart should have labels for each dimension.
   - Show the source code and the generated chart to your teacher.

   R `scatter3D` function and `matplotlib` library could prove useful.

   R Hints. You have to split the data into positive and negative lists (so in total 6 list). The scatter plot is described well in the documentation.
10. Analyse the data.

■

Exercise 13.2 Analysing Event Data

1. Plot 3D chart (as in exercise 1.1) but only for the first 8000 events.
   - Rotate the chart so that you can see the object's shapes clearly.
   - Compare the rotated chart to the first image from the `images` folder.
2. Plot the 3D chart but only for events with timestamp between 0.5 and 1
   - Rotate the chart so that you can see the direction of movement of objects over time.
3. Answer following questions (as comments in Your code):
   - How long is the sequence used during exercise 1.1 (in seconds)?
   - What's the resolution of event timestamps?
   - What does the time difference between consecutive events depend on?
   - What does positive/negative event polarity mean?
   - What is the direction of movement of objects in exercise 1.2?
4. Show the source code and the generated charts to your teacher.

■

13.3 Event frame representations

13.3.1 Why do we use event frame representations?

The subject of vision systems is one that has been developed for many years. So far, computer vision algorithms are based on classic image frames - 2D matrices representing the values (brightness) of individual pixels (it is not uncommon to use more channels, for example, three - R (red), G (green) and B (blue)).

The event camera data stream, which can be described as a sparse point cloud in space-time,

presents significant challenges in terms of designing analysis and image processing algorithms. It is difficult to use them with state-of-the-art computer vision algorithms and the previous achievements of over 60 years of computer vision cannot be easily applied. Three main approaches are used: an attempt to analyse the data as a point cloud (in a way similar to, e.g. LiDAR data), transformation (accumulation) of event data into various types of event frames, and reconstruction. In today's class, we will focus on the representation of event data in the form of frames, which are analogous to traditionally used matrices containing pixels brightness value.

13.3.2 Event frame

The simplest form of event data representation is the "event frame". Each pixel is assigned a polarity value of the recorded events. Therefore, it takes into account only information about coordinates and polarisation of pixel brightness changes, ignoring temporal information. A representation defined in this way can be described as:

$$f(u,t) = \begin{cases} P_e(u) & for\ t - \Sigma_e(u) \in (0;\tau) \\ 0 & for\ t - \Sigma_e(u) \in (\tau;+\infty) \end{cases} \tag{13.2}$$

Any pixels for which no event have been registered are marked with the value 127, and events are marked as 0 or 255 depending on the polarity.

13.3.3 Exponentially decaying time surface

An extension of the idea of an event frame is a representation that also takes into account the time of occurrence of an event in a time window. This method assumes that the weight of information decreases exponentially with time to zero. This representation is denoted by the Equation (13.3).

$$f(u,t) = \begin{cases} P_e(u) * e^{\frac{\Sigma_e(u)-t}{\tau}} & for\ \Sigma_e(u) \leq t \\ 0 & for\ \Sigma_e(u) > t \end{cases} \tag{13.3}$$

13.3.4 Event frequency

The sigmoid representation uses information about the frequency of events for a given pixel, which is ignored by the approaches presented so far. It is defined as:

$$f(x) = 255 * \frac{1}{1+e^{-x/2}} \tag{13.4}$$

The sum of the polarisation values present in a given pixel is denoted as x.

Exercise 13.3 Event frame

1. Create a Python file (use Visual Studio Code). Save it to the same folder, where the `events.txt` is located.
2. Import `cv2` and `numpy` libraries - we are going to need those during this class.
3. Read `events.txt` and save it to a variable (as You did last week).
4. Analyse the file line by line, save values, split them into particular variables: timestamp, x, y and polarity. (Points 6-7 from Basic Exercise 1.1 with some minor changes).
   - Remember to convert strings to proper variable types - timestamps should be floating points, coordinates and polarities should be integers
   - During this exercise, it would be easier to represent polarity as {1,-1} instead of {1,0}. Update negative polarity values during splitting events into variables.

R Save only those events, where timestamp > 1 and < 2. Saving all events would take a lot of time! This interval was chosen because of the significant movement of objects relative to the camera, which will facilitate the task.

5. Implement function that creates event frame representation.
   - Define function *event_frame* with following input arguments - event coordinates lists, polarities list and image shape.
   - Inside the function, create 2D matrix with image shape. Initialize this matrix with value 127 (as mentioned in the Theoretical part of this class). This matrix will represent our image.

     R `numpy` python library could prove useful. Its function `np.ones()` can be used to create a matrix with ones as all its elements. This matrix can be later multiplied by 127.

   - In order to properly visualize matrix as image, change its type to uint8 (values are represented by integers of 0-255 where 0 is black and 255 is white).

     R You can do it with `image.astype(np.uint8)` function (if you imported numpy library first).

   - Loop through all events and set pixel values for each coordinates to 255 (for positive polarity) and to 0 (for negative polarity).
   - This function should return image, that is, the completed pixel matrix.
6. Each event frame represents the events that were recorded in a given time window. Thus, it is necessary to select the duration of data aggregation, which will then be used to create a representation. Create a variable `tau` and set it to 10 ms (remember that timestamp is expressed in seconds, so this variable should also be expressed in seconds).
7. Loop through all saved events. Create temporary lists to store aggregated event data. Append saved events to temporary lists untill the difference between the first and last timestamp is greater than tau. Then call the function you prepared earlier and pass the aggregated event data (temporary lists) - timestamps, coordinates and polarity - to it as arguments.
8. Show image with `opencv` functions.

   R All `opencv` functions are well documented - just google it. You should use `imshow()` function to show created image. You should also use `waitKey()` function just after `imshow()` - This ensures that the image will be displayed until you press any key on the keyboard.

9. After generating event frame you should keep looping through saved events in order to prepare next event frame. Clear temporary lists and keep aggregating data. The event frame should be generated every 10 ms. Since the length of entire sequence is 1s, you should get 100 images. Compare generated images to those in *images* folder.

   R You should use one *for* loop with *if* to verify the timestamp difference. Whenever the condition is met, you call *event_frame*, clear temporary list and show image. Then - the loop can be continued as soon as you press any key on the keyboard.

10. Try running the same code with different values of `tau` (1ms, 10ms, 100ms). How does the aggregation time `tau` affect the generated event frames? Answer as comment to your code.
11. Show the source code and the generated images to your teacher.

■