Advanced Vision Systems

AGH UST International Course

Tomasz Kryjak, PhD, Mateusz Wąsala, MSc

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*.

■

R 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.

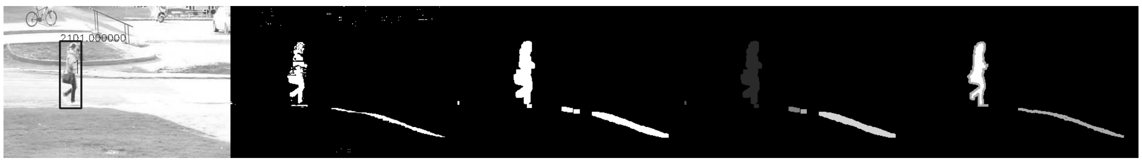


Figure 2.1: An example of the result of each stage of the algorithm. From left: input image with bounding box, image after binarisation, image after median filtering and morphological operations (erode, dilate), image representing labels after labelling, reference image – ground truth.

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.

R 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.

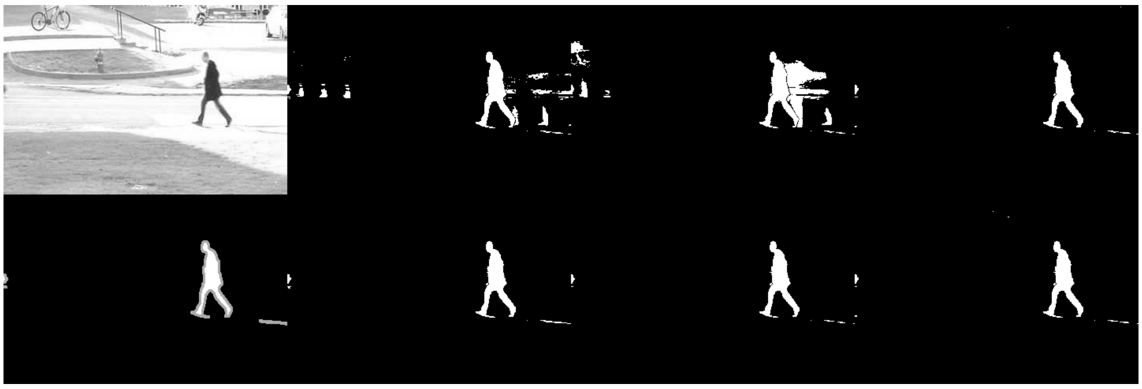


Figure 3.1: Example of the result of the different versions of the algorithm for foreground object segmentation for the *pedestrians* dataset. The frame index is 600. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the 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.

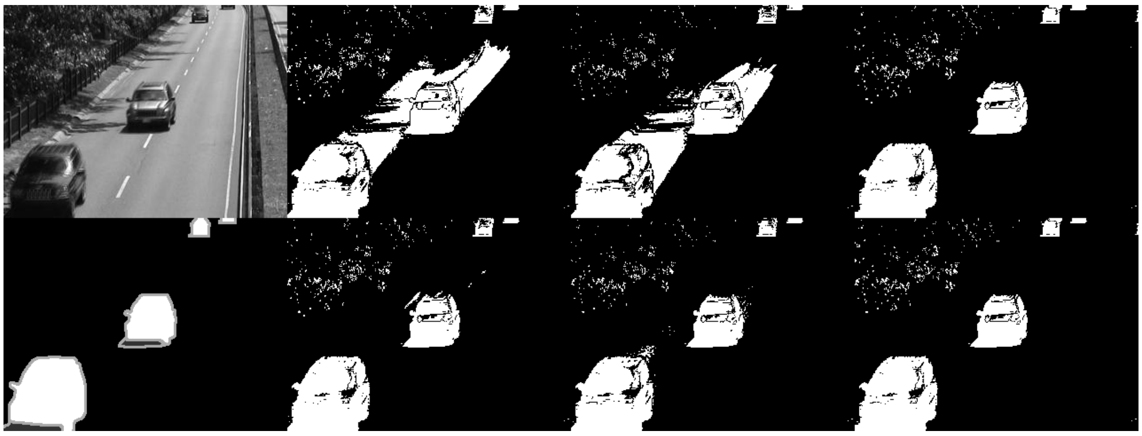


Figure 3.2: Example of the result of the different versions of the algorithm for foreground object segmentation for the *highway* dataset. The frame index is 1200. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the 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.

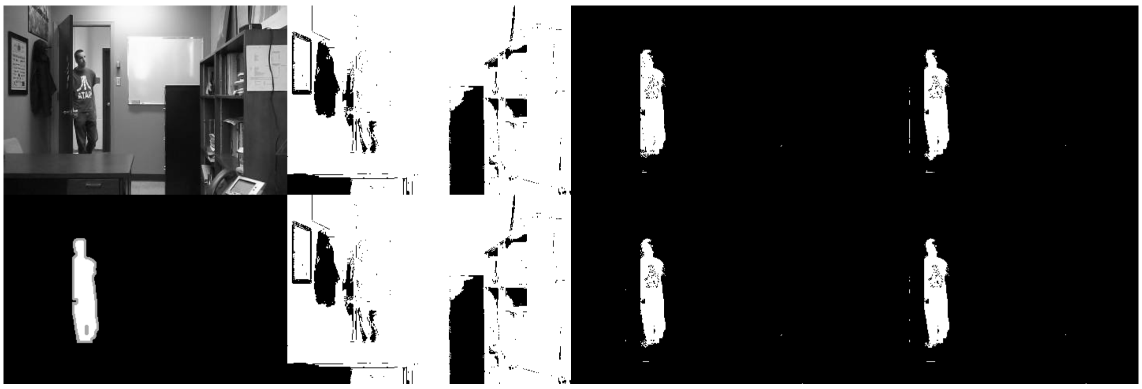


Figure 3.3: Example of the result of the different versions of the algorithm for foreground object segmentation for the *office* dataset. The frame index is 600. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the second 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.

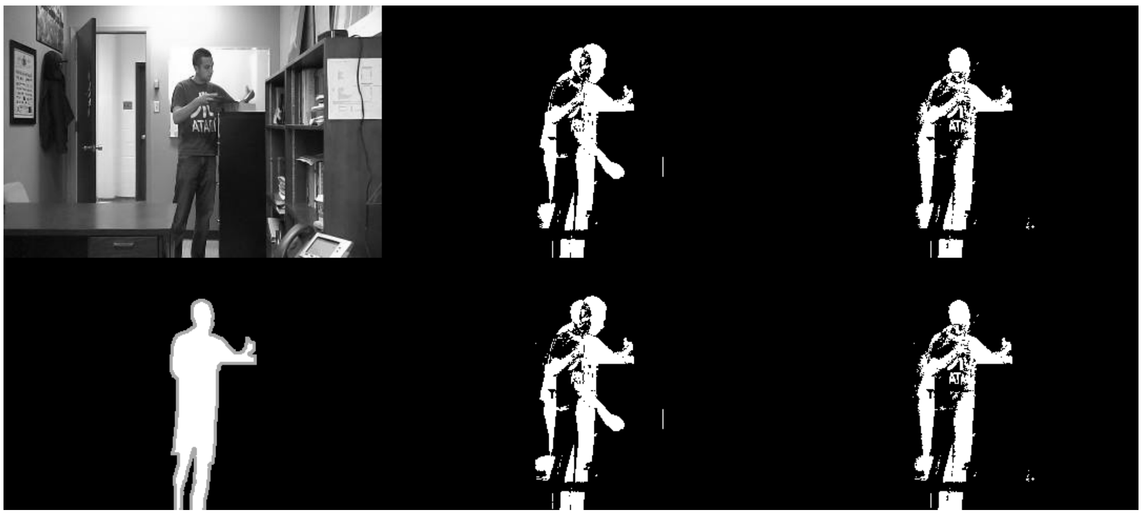


Figure 3.4: Example result of the proposed solution for *office* dataset. The frame index is 800. The first column from the left is the input image together with the reference image. The first row is for algorithms using the mean, the second row uses the median. Sequentially from the second column the 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.

■

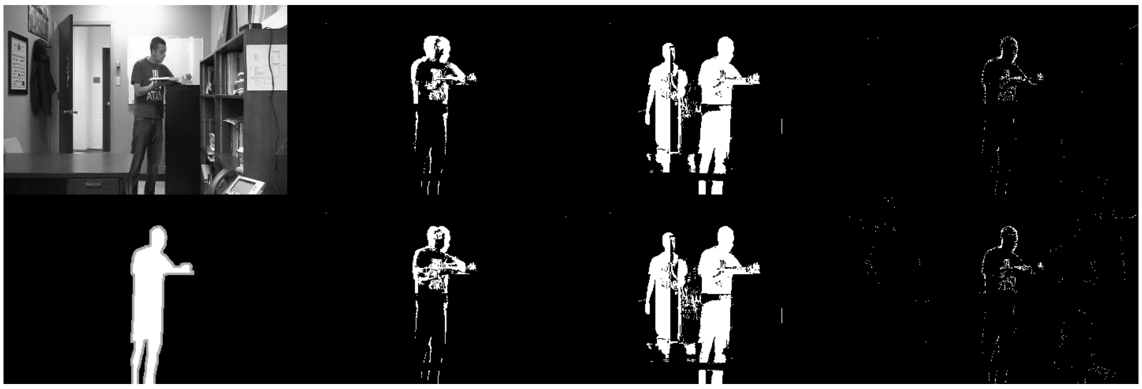


Figure 3.5: Example result of the proposed solution for *office* dataset. The frame index is 1000. The first column from the left is the input image together with the reference image. Then the approximation of the mean and median with a liberal update policy, the approximation of the mean and median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

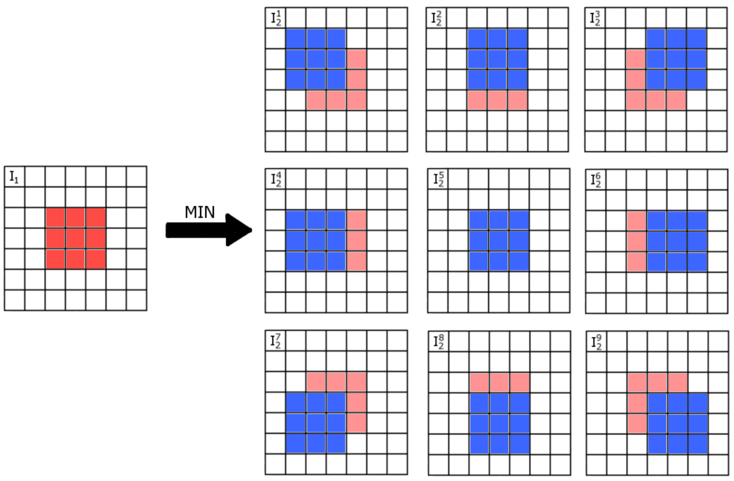


Figure 4.1: Example of searching for a matching block

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: `I0 = 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 `J0` surroundings, converting them to *float32* and then calculating the „distance" between `I0` and `J0` slices. This can be done with the instruction: `np.sqrt(np.sum((np.square(J0-I0))))`. From all the fragments of `J0` for a given `I0`, find the smallest „distance" – the location of the „closest" to the `I0` 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.

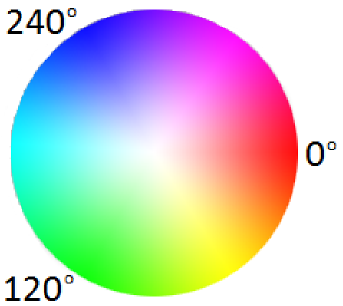


Figure 4.2: Color wheel

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

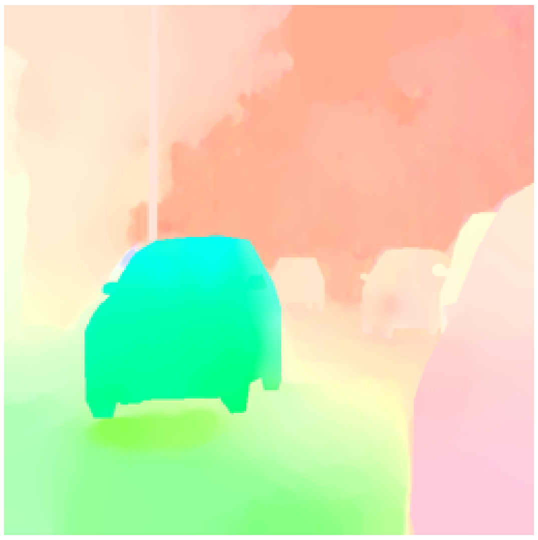


Figure 4.3: Ground truth for `cm1.png` and `cm2.png` images.

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.

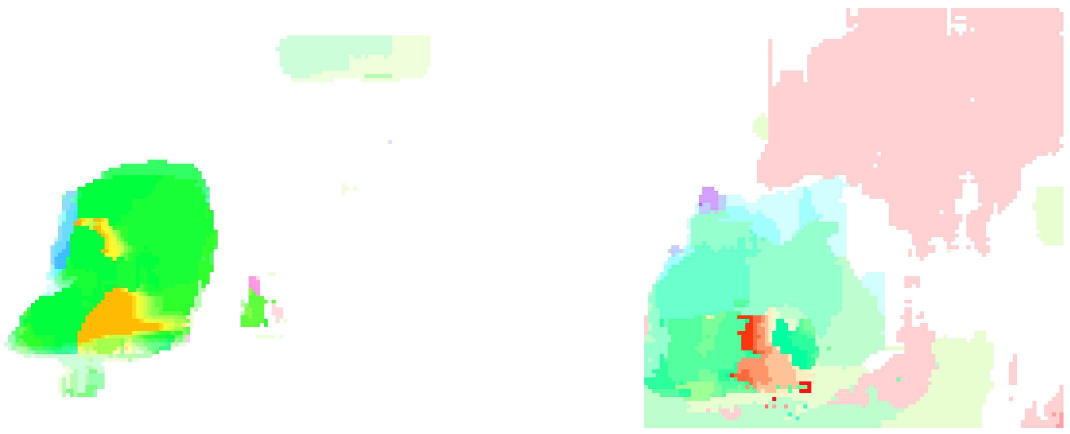


Figure 4.4: Example results of the optical flow determined by the block method: on the left for images `I.jpg` and `J.jpg`, image downscaled twice and parameters $W2 = dX = dY = 5$; on the right for images `cm1.png` and `cm2.png`, downscaled twice and parameters $W2 = dX = dY = 5$.

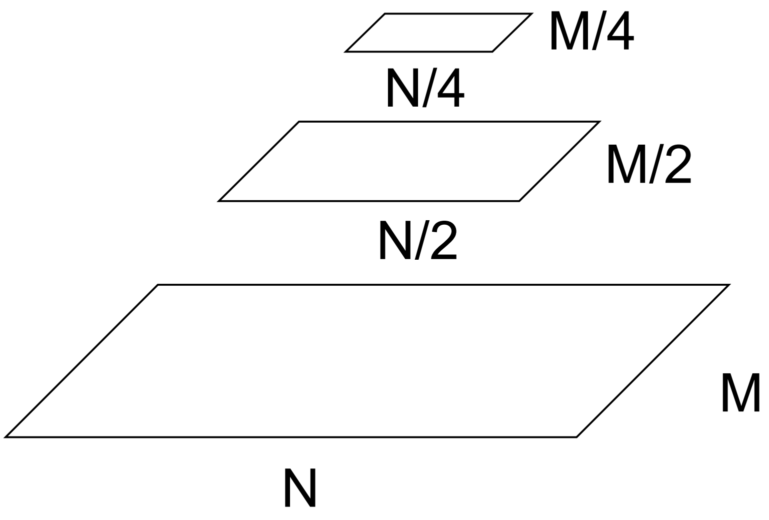


Figure 4.5: Example of pyramid image construction (multiple scales)

- 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.

   R 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.

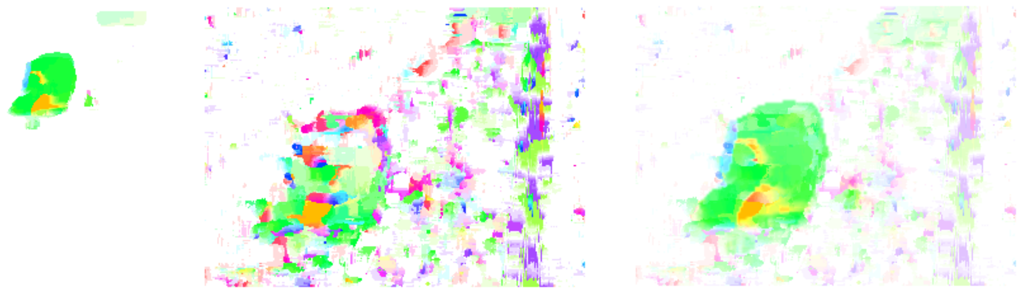


Figure 4.6: Example results of the optical flow determined by the block method in 2 scales for `I.jpg` and `J.jpg` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: flow at smaller scale, residual flow at larger scale, total flow.

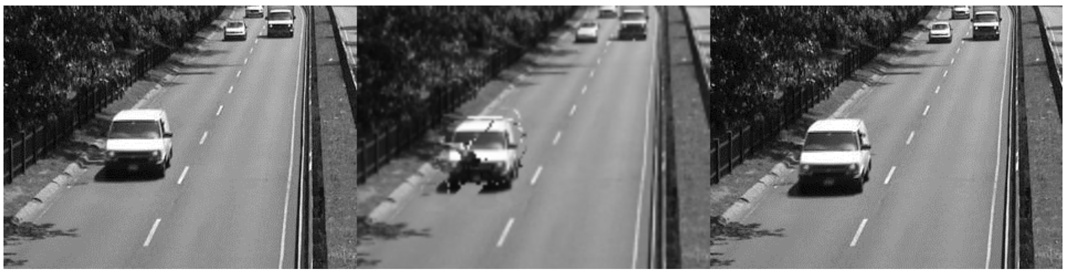


Figure 4.7: Example warping results for `I.jpg` and `J.jpg` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: the previous frame I, the frame modified according to the lower scale flow I_org, the next frame J.

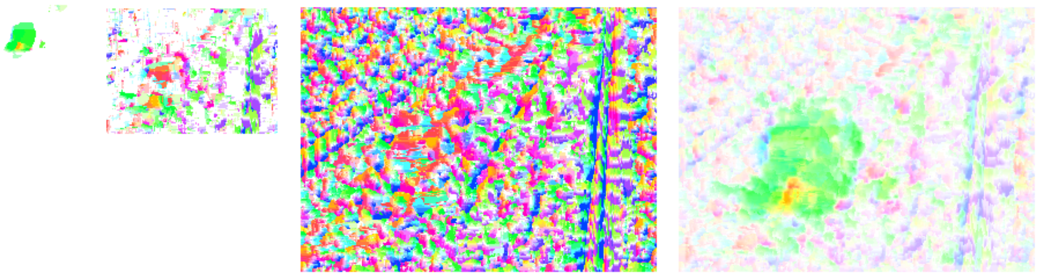


Figure 4.8: Example results of optical flow determined by the block method in 3 scales for `I.jpg` and `J.jpg` images, in original size and parameters $W2 = dX = dY = 3$. Sequentially from the left: smallest scale flow, middle scale residual flow, largest scale residual flow, total flow.

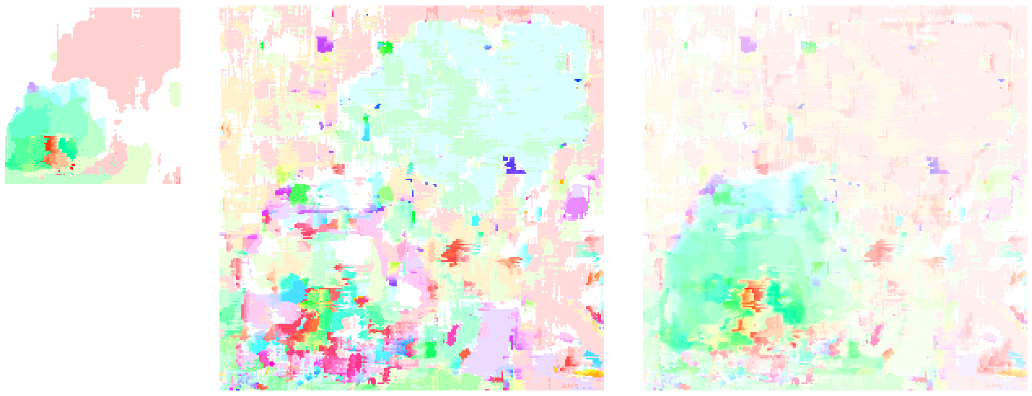


Figure 4.9: Example results of the optical flow determined by the block method in 2 scales for `cm1.png` and `cm2.png` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: flow at smaller scale, residual flow at larger scale, total flow.

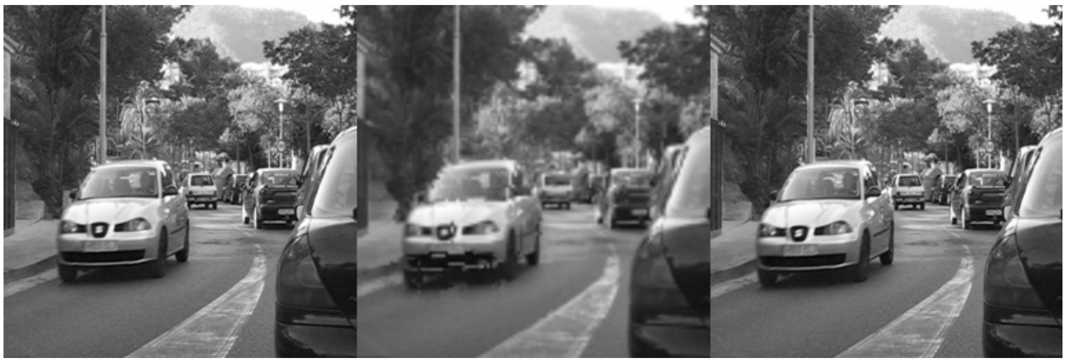


Figure 4.10: Example warping results for `cm1.png` and `cm2.png` images, in original size and parameters $W2 = dX = dY = 5$. Sequentially from the left: the previous frame I, the frame modified according to the lower scale flow I_org, the next frame J.

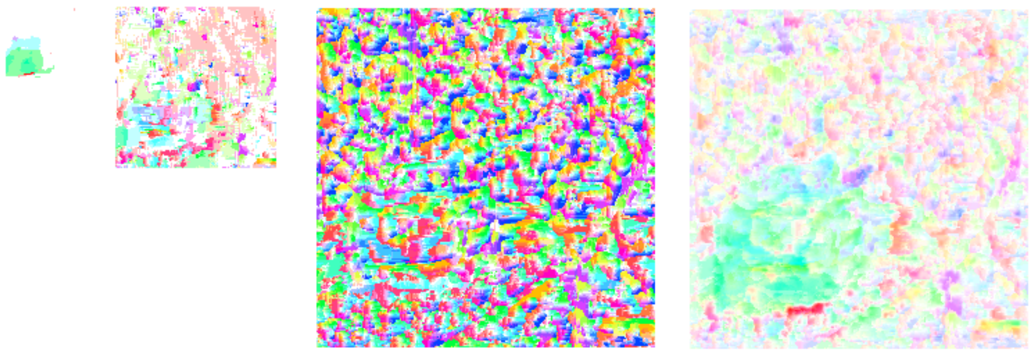


Figure 4.11: Example results of optical flow determined by the block method in 3 scales for `cm1.png` and `cm2.png` images, in original size and parameters $W2 = dX = dY = 3$. Sequentially from the left: smallest scale flow, middle scale residual flow, largest scale residual flow, total flow.

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.

■

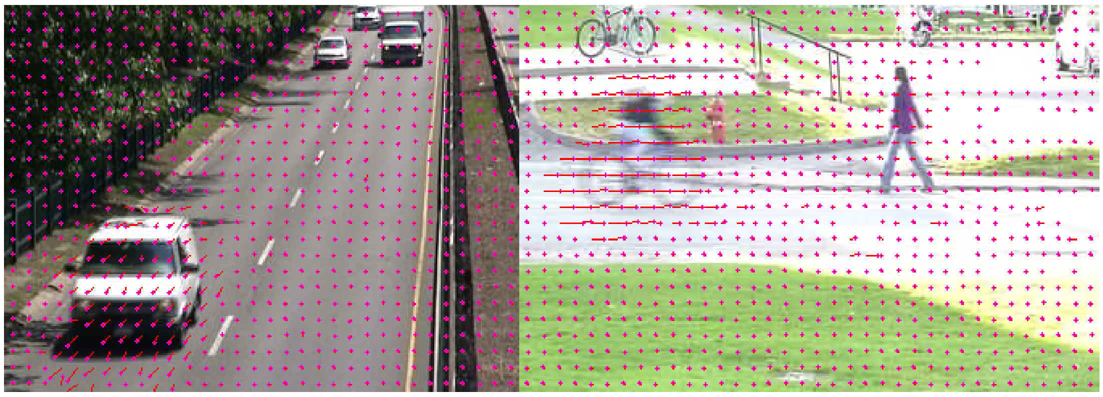


Figure 4.12: Example results of sparse optical flow determined by the LK method: left for image index 158 from the *highway* sequence; right for image index 471 from the *pedestrians* sequence.

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.

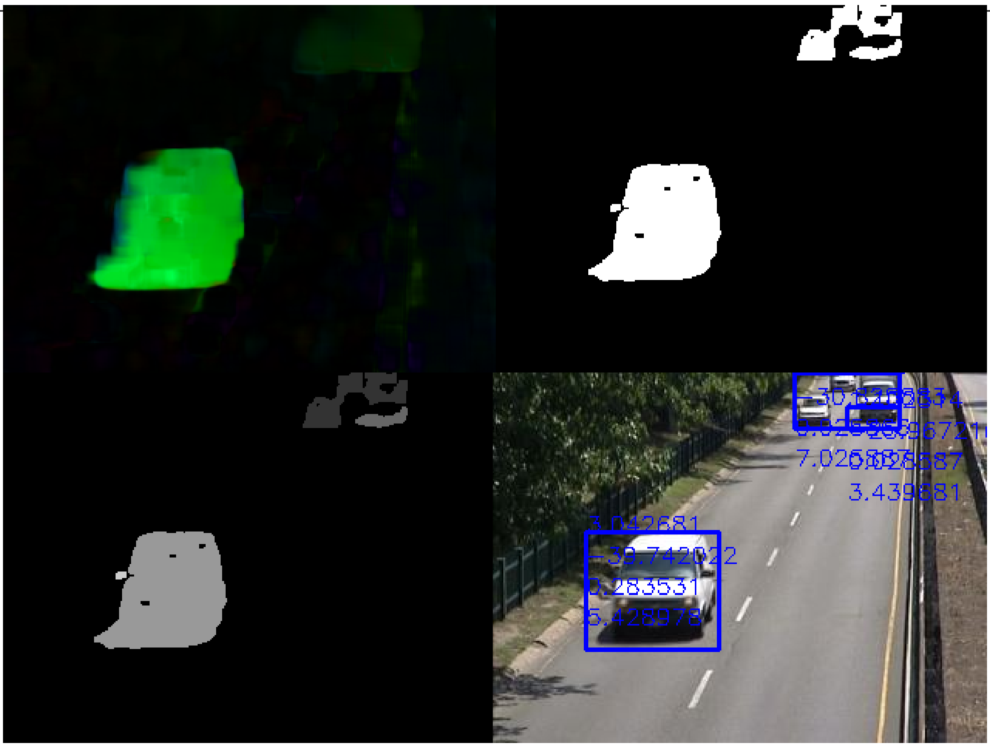


Figure 4.13: Example results for an image with index 146 from the *highway* sequence: top left – Farneback optical flow, top right – foreground object segmentation, bottom left – labelling, bottom right – detected objects with calculated parameters.

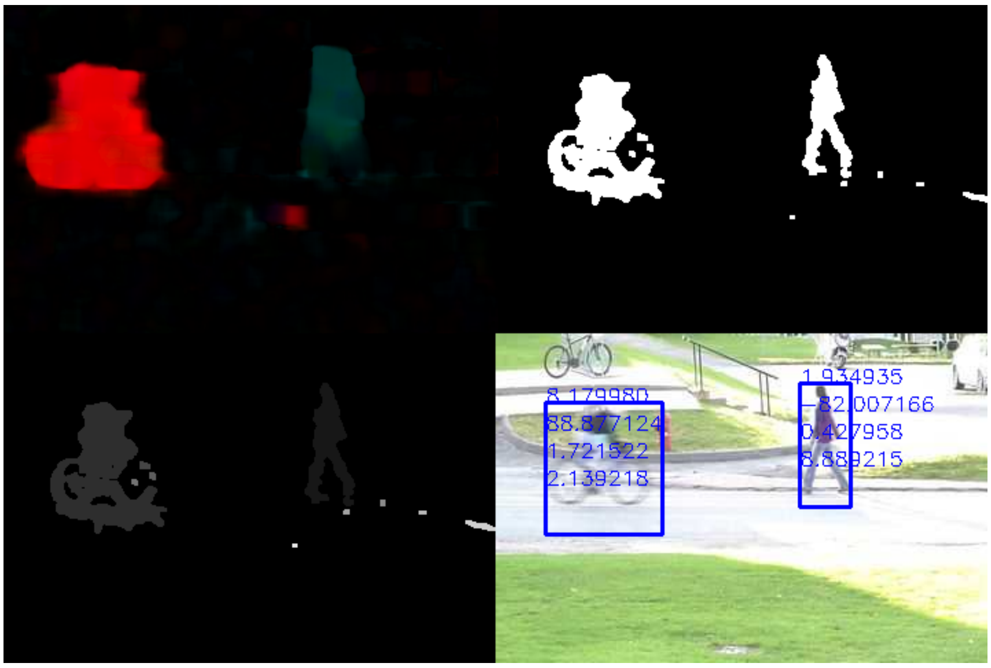


Figure 4.14: Example results for the image with index 471 from the *pedestrians* sequence: top left – Farneback optical flow, top right – foreground object segmentation, bottom left – labelling, bottom right – detected objects with calculated parameters.

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.

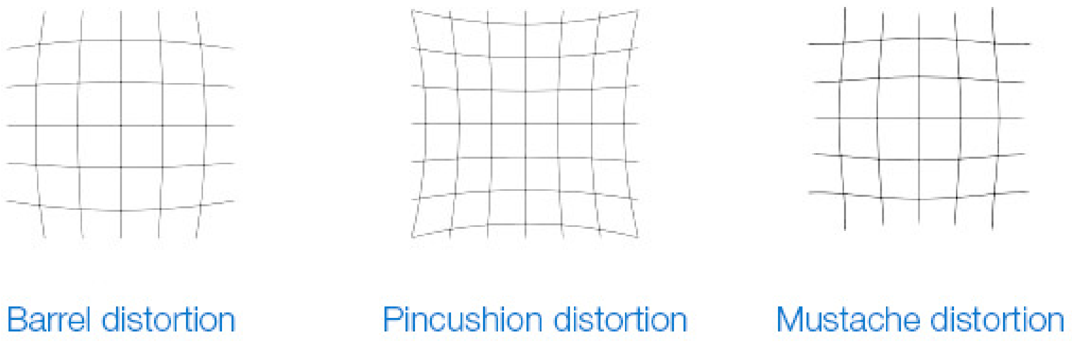


Figure 5.1: Types of distortion: (a) barrel, (b) pincushion (c) mustache. Source: https://www.panoramic-photo-guide.com/optical-distortions-panoramic-photography.html.

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

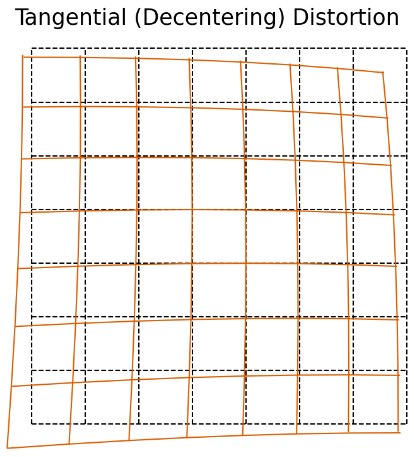


Figure 5.2: Tangential distortion. Source: https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i.

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.

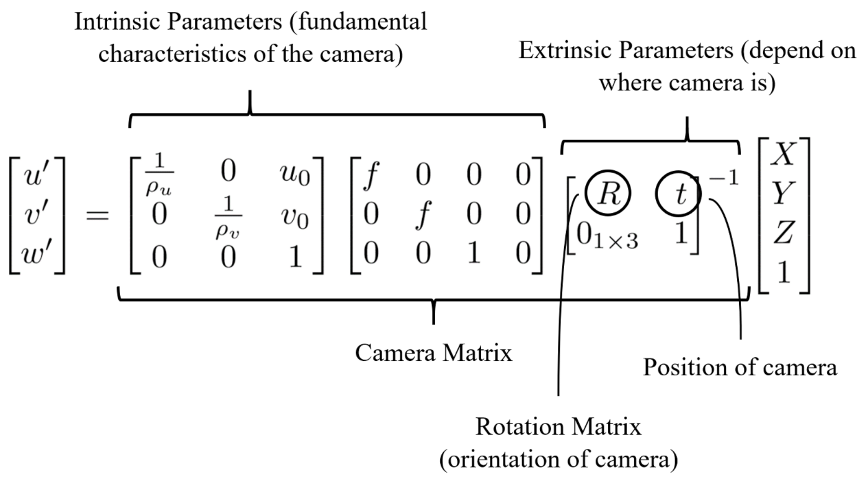
There are two camera models:

- *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


Intrinsic Parameters (fundamental characteristics of the camera)
Extrinsic Parameters (depend on where camera is)
Camera Matrix
Position of camera
Rotation Matrix (orientation of camera)

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} \frac{1}{\rho_u} & 0 & u_0 \\ 0 & \frac{1}{\rho_v} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0_{1\times 3} & 1 \end{bmatrix}^{-1} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

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.

```python
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)
```

R 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

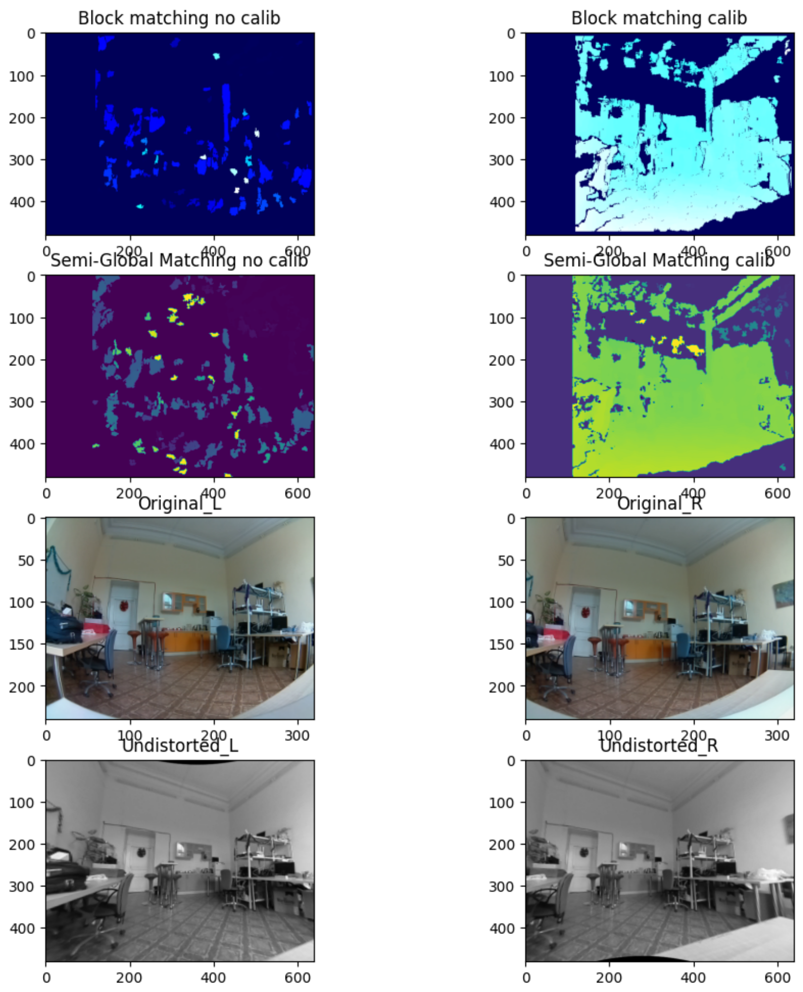


Figure 5.4: An example solution.

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. ■

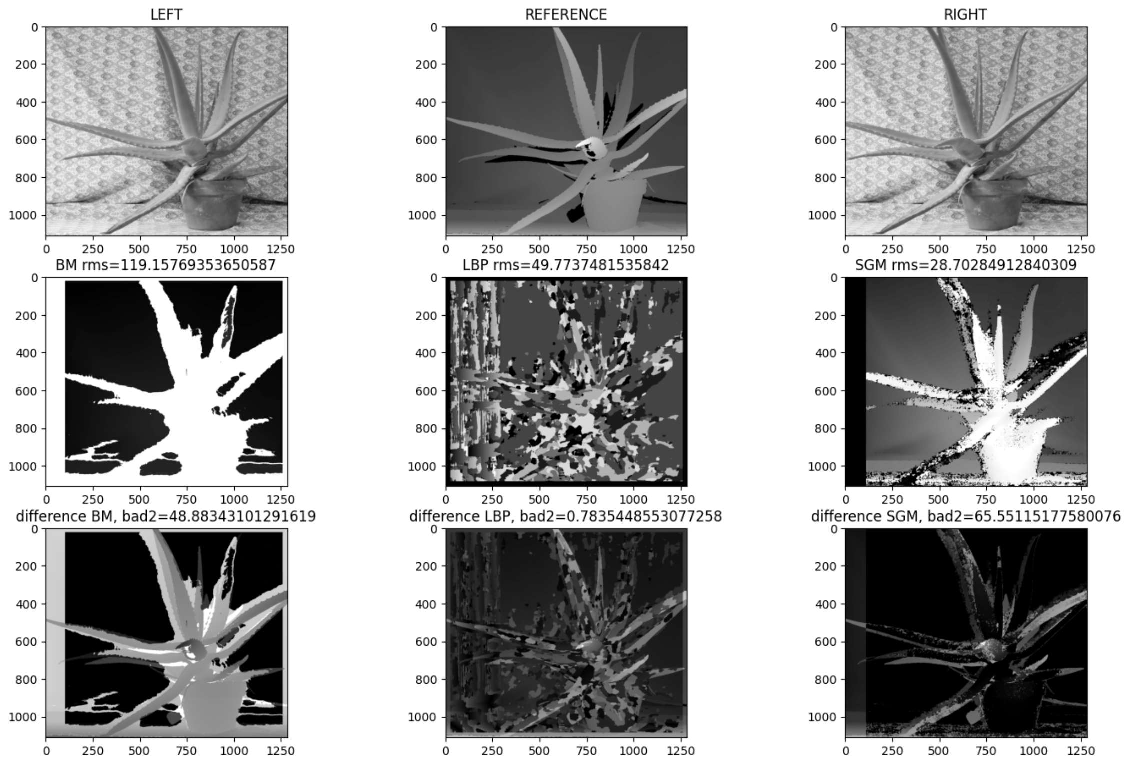


Figure 5.5: An example solution.