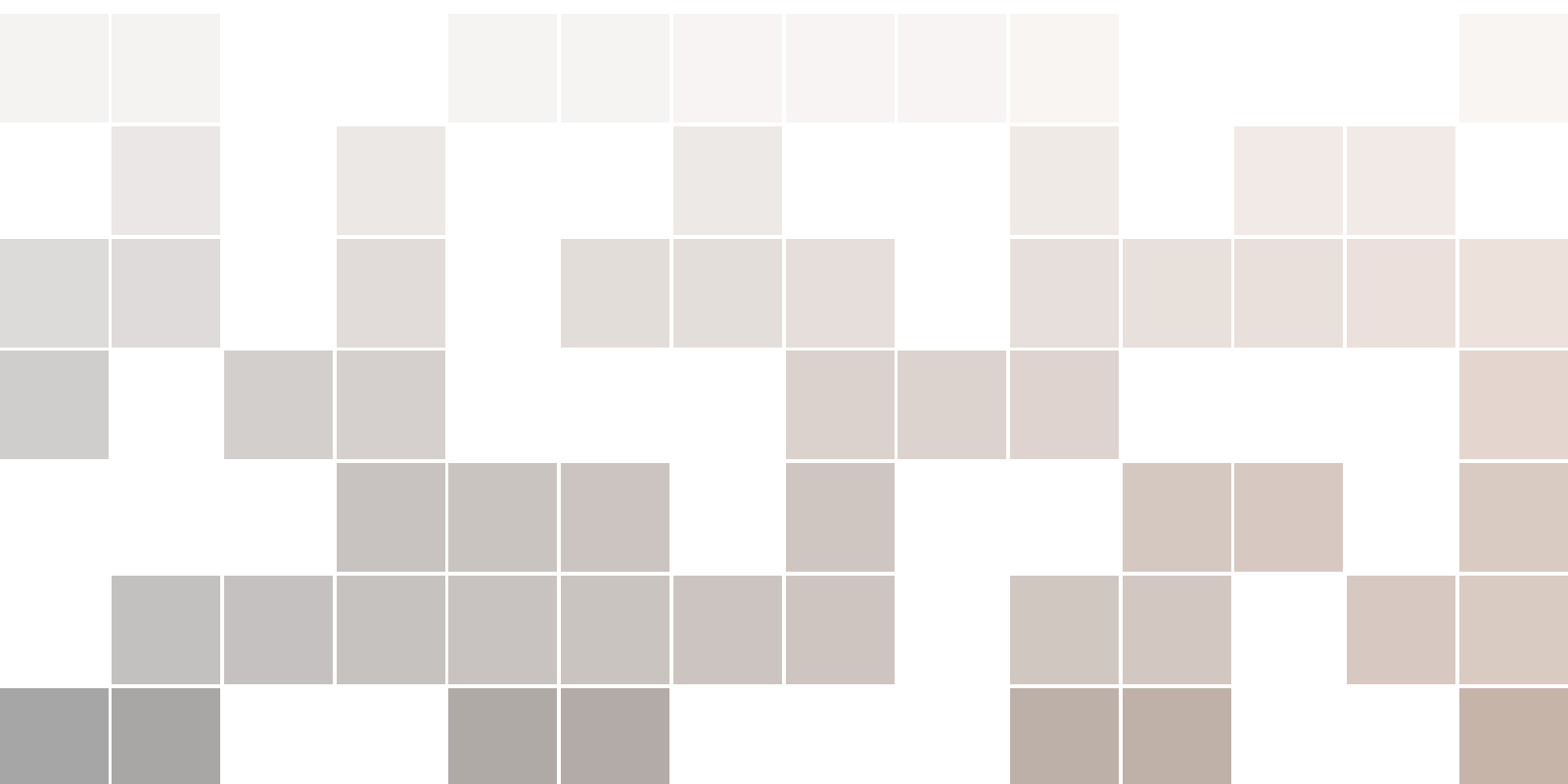




# Zaawansowane algorytmy wizyjne

Skrypt do ćwiczeń laboratoryjnych

Piotr Pawlik, Tomasz Kryjak, Mateusz Wąsala



Copyright © 2023

Piotr Pawlik

Tomasz Kryjak

Mateusz Wąsala

ZESPÓŁ WBUDOWANYCH SYSTEMÓW WIZYJNYCH

LABORATORIUM SYSTEMÓW WIZYJNYCH

WEAiIB, AGH w KRAKOWIE

*Wydanie trzecie, zmienione i poprawione*

*Kraków, luty 2024*

## Spis treści

I	Laboratorium 1	
1	Algorytmy wizyjne w Python 3.X – wstęp .....	9
1.1	Wykorzystywane oprogramowanie	9
1.2	Moduły Pythona wykorzystywane w przetwarzaniu obrazów	9
1.3	Operacje wejścia/wyjścia	10
1.3.1	Wczytywanie, wyświetlanie i zapisywanie obrazu z wykorzystaniem OpenCV .	10
1.3.2	Wczytywanie, wyświetlanie i zapisywanie obrazu z wykorzystaniem modułu <i>Matplotlib</i> .....	11
1.4	Konwersje przestrzeni barw	13
1.4.1	OpenCV .....	13
1.4.2	Matplotlib .....	13
1.5	Skalowanie, zmiana rozdzielczości przy użyciu OpenCV	14
1.6	Operacje arytmetyczne: dodawanie, odejmowanie, mnożenie, moduł z różnicy	14
1.7	Wylczenie histogramu	15
1.8	Wyrównywanie histogramu	15
1.9	Filtracja	16
II	Laboratorium 2	
2	Detekcja pierwszoplanowych obiektów ruchomych .....	19
2.1	Wczytywanie sekwencji obrazów	19

2.2	Odejmowanie ramek i binaryzacja	20
2.3	Operacje morfologiczne	21
2.4	Indeksacja i prosta analiza	21
2.5	Ewaluacja wyników detekcji obiektów pierwszoplanowych	22
2.6	Przykładowe rezultaty algorytmu do detekcji ruchomych obiektów pierwszoplanowych	24

### III

## Laboratorium 3

3	Segmentacja obiektów pierwszoplanowych .....	27
3.1	Cel zajęć	27
3.2	Segmentacja obiektów pierwszoplanowych	27
3.3	Metody oparte o bufor próbek	29
3.4	Aproksymacja średniej i mediany (tzw. sigma-delta)	30
3.5	Polityka aktualizacji	31
3.6	OpenCV – GMM/MOG	31
3.7	OpenCV – KNN	32
3.8	Przykładowe rezultaty algorytmu do segmentacji obiektów pierwszoplanowych	32
3.9	Wykorzystanie sieci neuronowej do segmentacji obiektów pierwszoplanowych	35
3.10	Zadania dodatkowe	35
3.10.1	Inicjalizacja modelu tła w warunkach obecności obiektów pierwszoplanowych na scenie .....	35
3.10.2	Implementacja metod ViBE i PBAS .....	36

### IV

## Laboratorium 4

4	Przepływ optyczny .....	41
4.1	Cel zajęć	41
4.2	Przepływ optyczny (ang. <i>optical flow</i> )	41
4.3	Implementacja metody blokowej	42
4.4	Implementacja wyliczania w wielu skalach	44
4.5	Zadanie dodatkowe 1 – Inne sposoby wyznaczania przepływu optycznego	50
4.6	Zadanie dodatkowe 2 – Detekcja kierunku ruchu i klasyfikacja obiektów z wykorzystaniem przepływu optycznego	51

<b>V</b>	<b>Laboratorium 5</b>	
<b>5</b>	<b>Kalibracja kamery i stereowizja .....</b>	<b>57</b>
5.1	Kalibracja pojedynczej kamery	59
5.2	Kalibracja układu kamer	62
5.3	Obliczenia korespondencji stereo	63
5.4	Wykorzystanie sieci neuronowej do realizacji zadania analizy głębi obrazu	64
5.5	Zadanie dodatkowe – transformata Censusa	65
<b>VI</b>	<b>Laboratorium 6</b>	
<b>6</b>	<b>Punkty Charakterystyczne .....</b>	<b>69</b>
6.1	Cel zajęć	69
6.2	Detekcja narożników metodą Harrisa – teoria	69
6.3	Implementacja metody Harrisa	70
6.4	Prosta deskrypcja punktów charakterystycznych	71
6.5	ORB (FAST+BRIEF)	73
6.6	Wykorzystanie punktów charakterystycznych do łączenia obrazów	75
6.7	Zadanie dodatkowe – SIFT	76





# Laboratorium 1

<b>1</b>	<b>Algorytmy wizyjne w Python 3.X – wstęp</b>	<b>9</b>
1.1	Wykorzystywane oprogramowanie	
1.2	Moduły Pythona wykorzystywane w przetwarzaniu obrazów	
1.3	Operacje wejścia/wyjścia	
1.4	Konwersje przestrzeni barw	
1.5	Skalowanie, zmiana rozdzielczości przy użyciu OpenCV	
1.6	Operacje arytmetyczne: dodawanie, odejmowanie, mnożenie, moduł z różnicy	
1.7	Wyliczenie histogramu	
1.8	Wyrównywanie histogramu	
1.9	Filtracja	





# 1. Algorytmy wizyjne w Python 3.X – wstęp

W ramach ćwiczenia zaprezentowane/przypomniane zostaną podstawowe informacje związane z wykorzystaniem języka Python do realizacji operacji przetwarzania i analizy obrazów, a także sekwencji wideo.

## 1.1 Wykorzystywane oprogramowanie

Przed rozpoczęciem ćwiczeń **proszę utworzyć** własny katalog roboczy w miejscu podanym przez Prowadzącego. Nazwa katalogu powinna być związana z Państwa nazwiskiem – zalecana forma to NazwiskoImie bez polskich znaków.

Do przeprowadzania ćwiczeń można wykorzystać jedno z trzech środowisk programowania w Pythonie – Spyder5, PyCharm lub Visual Studio Code (VSCode). Ich zaletą jest możliwość łatwego podglądu tablic dwuwymiarowych (czyli obrazów). Spyder jest prostszy, ale także ma mniejsze możliwości i więcej niedociągnięć. PyCharm jest oprogramowaniem komercyjnym udostępnianym także w wersji darmowej (wygląd zbliżony do CLion dla C/C++). W PyCharmie pracę należy zacząć od utworzenia projektu, Spyder pozwala na pracę na pojedynczych plikach (bez konieczności tworzenia projektu).

Visual Studio Code jest darmowym, lekkim, intuicyjnym i łatwym w obsłudze edytorem kodu źródłowego. Jest to uniwersalne oprogramowanie do dowolnego języka programistycznego. Konfiguracja programu do ćwiczeń sprowadza się jedynie do wybrania odpowiedniego interpretera, a dokładnie odpowiedniego środowiska wirtualnego w języku Python.

## 1.2 Moduły Pythona wykorzystywane w przetwarzaniu obrazów

Python nie posiada natywnego typu tablicowego pozwalającego na wykonanie operacji pomiędzy elementami dwóch kontenerów (w dogodny i wydajny sposób). Do operacji na tablicach 2D (czyli także na obrazach) powszechnie używa się zewnętrznego modułu *NumPy*. Jest to podstawa dla wszystkich dalej wymienionych modułów wspierających przetwarzanie i wyświetlanie obrazów. Istnieją co najmniej 3 zasadnicze pakiety wspierające przetwarzanie obrazów:

- para modułów: moduł *ndimage* z biblioteki *SciPy* – zawiera funkcje do przetwarzania obrazów (także wielowymiarowych) oraz moduł *pyplot* z biblioteki *Matplotlib* – do wyświetlania obrazów i wykresów,
- moduł *PILLOW* (fork<sup>1</sup> starszego, nierozwijanego modułu *PIL*),
- moduł *cv2* będący nakładką na popularną bibliotekę *OpenCV*.

W naszym kursie oprzemy się na *OpenCV*, aczkolwiek wykorzystywane będzie także *Matplotlib* i sporadycznie *ndimage* – głównie w sytuacjach kiedy funkcje z *OpenCV* nie mają odpowiednich funkcjonalności albo są mniej wygodne w stosowaniu (np. wyświetlanie obrazów).

## 1.3 Operacje wejścia/wyjścia

### 1.3.1 Wczytywanie, wyświetlanie i zapisywanie obrazu z wykorzystaniem OpenCV

**Ćwiczenie 1.1** Wykonaj zadanie, w którym przećwiczysz obsługę plików z wykorzystaniem *OpenCV*.

1. Ze strony kursu pobierz obraz *mandril.jpg* i umieść go we własnym katalogu roboczym.
2. Uruchom program – *spyder*, *pycharm*, *vscode* – z konsoli lub za pomocą ikony. Stwórz nowy plik i zapisz go we własnym katalogu roboczym.
3. W pliku załaduj moduł *cv2* (`import cv2`). Przetestuj wczytywanie i wyświetlanie obrazów za pomocą tej biblioteki – do wczytywania służy funkcja *imread* (przykład użycia: `I = cv2.imread('mandril.jpg')`).
4. Wyświetlenie obrazka wymaga użycia co najmniej dwóch funkcji – *imshow()* tworzy okienko i rozpoczyna wyświetlanie, a *waitKey()* pokazuje obraz przez zadany okres czasu (argument 0 oznacza czekanie na wciśnięcie klawisza). Po zakończeniu pracy okno jest automatycznie zamykane, ale pod warunkiem zakończenia przez naciśnięcie dowolnego klawisza.

**Uwaga!** Zamknięcie okna przez przycisk na belce okna spowoduje zapętlenie programu i konieczność jego przerwania:

- *Spyder* – zamknięcie konsoli *IPython*,
- *PyCharm* – wybranie pozycji 'Stop' w menu lub odpowiadającego jej przycisku,
- *VSCode* – zatrzymanie programu w terminalu poprzez użycie skrótu klawiszowego 'CTRL+Z' lub zamknięcie terminalu.

**R** W przypadku szczególnie obrazów o dużej rozdzielczości warto przed funkcją *cv2.imshow()* użyć funkcję *namedWindow()* z tą samą nazwą co w funkcji *imshow()*.

5. Niepotrzebne już okno można zamknąć za pomocą *destroyWindow* z odpowiednim parametrem lub zamknąć wszystkie otwarte okna za pomocą *destroyAllWindows*.

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

**R** Okno niekoniecznie musi zostać wyświetlone na wierzchu. Dobrą praktyką jest zawsze stosowanie funkcji *cv2.destroyAllWindows()*.

<sup>1</sup> gałąź boczna projektu

6. Zapis do pliku realizuje funkcja `imwrite` (proszę zwrócić uwagę na zmianę formatu z `jpg` na `png`):

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

7. Wyświetlany obraz jest kolorowy, co oznacza, że składa się z 3 składowych. Innymi słowy jest reprezentowany jako tablica 3D. Często istnieje potrzeba podglądu wartości tej tablicy. Można to robić w formie jednowymiarowej, ale lepiej to zrobić za pomocą macierzy dwuwymiarowej.

- Spyder – w zakładce *Variable explorer* (narzędzie podobne do *Workspace* znanego z Matlaba) wystarczy kliknąć na zmienną `I`.
- PyCharm – konieczne jest uruchomienie programu w trybie *Debug* i ustawienie breakpointa. Po zatrzymaniu na nim i kliknięciu w zakładkę *Debugger* należy kliknąć w tekst *View as Array* na końcu linii pokazującej zmienną `I`,
- VSCode – konieczne jest uruchomienie programu w trybie *Debug* i ustawienie breakpointa. W zakładce *Run and Debug* i w sekcji *Variables* znajdziemy zmienną `I`. Aby wyświetlić wartości macierzy `I` należy wybrać opcję *View Value in Data Viewer* pod prawym klawiszem myszki. W sekcji *Watch* możliwe jest odwołanie do konkretnego elementu tablicy `I` lub wykonać odpowiednie operacje na niej.

We wszystkich przypadkach wyświetlana jest macierz 2D będąca “wycinkiem” tablicy 3D, jednakże domyślnie jest to wycinek “w złej osi” – wyświetlana jest pojedyncza linia w 3 składowych. Aby uzyskać wyświetlenie całego obrazu dla jednej składowej należy zmienić oś.

- Spyder – pole *Axis* należy ustawić na 2,
- PyCharm – wycinek `I[0]` należy zmienić na `I[:, :, 0]`,
- VSCode – w sekcji *SLICING* wybrać odpowiednią oś – ustawić na 2 lub wybrać odpowiednie indeksowanie i nacisnąć przycisk *Apply*.

Pozostałe składowe są dostępne:

- Spyder – pole *Index*, wartości 1 i 2,
- PyCharm – Wycinki `I[:, :, 1]` i `I[:, :, 2]`,
- VSCode – pole *Index*.

8. W pracy przydatny może być dostęp do parametrów obrazu:

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

Funkcja `print` to oczywiście wyświetlanie na konsolę.

### **Rozliczenie ćwiczenia**

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu `.py` lub `.ipynb` w odpowiednim miejscu w zasobach kursu na UPeL.

## **1.3.2 Wczytywanie, wyświetlanie i zapisywanie obrazu z wykorzystaniem modułu *Matplotlib***

Alternatywny sposób realizacji operacji wejścia/wyjścia to użycie biblioteki *Matplotlib*. Wtedy obsługa wczytywania/wyświetlania itp. jest zbliżona do tej znanej z pakietu Matlab. Dokumentacja

biblioteki dostępna jest on-line [Matplotlib](#).

**Ćwiczenie 1.2** Wykonaj zadanie, w którym przećwiczysz obsługę plików z wykorzystaniem biblioteki Matplotlib. W celu zachowania pewnego porządku w kodzie, utwórz nowy plik z kodem źródłowym.

1. Załaduj moduł *pyplot*.

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

(as pozwala skrócić nazwę, którą trzeba będzie wykorzystywać w projekcie)

2. Wczytaj ten sam obraz *mandril.jpg*

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

3. Wyświetlanie realizuje się bardzo podobnie jak w pakiecie Matlab:

```
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. Zapisywanie obrazu:

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

5. Podczas laboratorium przydatne może być też wyświetlanie pewnych elementów na obrazku – np. punktów, czy ramek.
6. Dodanie wyświetlania punktów:

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

**Uwaga!** Przy ustalaniu wartości tablicy przecinki są niezbędne (w odróżnieniu od Matlab). W poleceniu `plot` składania podobna jak w Matlabie – 'r' – kolor, '.' – kropka oraz rozmiar markera.

Pełna lista możliwości w dokumentacji.

7. Dodanie wyświetlania prostokąta – rysowanie kształtów tj. prostokątów, elips, kół itp. jest dostępne w *matplotlib.patches*. Proszę zwrócić uwagę na komentarze w poniższym kodzie.

```
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()
```

### Rozliczenie ćwiczenia

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu .py lub .ipynb w odpowiednim miejscu w zasobach kursu na UPeL.



## 1.4 Konwersje przestrzeni barw

### 1.4.1 OpenCV

Do konwersji przestrzeni barw służy funkcja *cvtColor*.

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

**Uwaga!** Proszę zauważyć, że w OpenCV odczyt jest w kolejności BGR, a nie RGB. Może to być istotne w przypadku ręcznego manipulowania pikselami. Pełna lista dostępnych konwersji wraz ze stosownymi wzorami w [dokumentacji OpenCV](#)

**Ćwiczenie 1.3** Wykonaj konwersję przestrzeni barw podanego obrazu.

1. Dokonać konwersji obrazu *mandril.jpg* do odcieni szarości i przestrzeni HSV. Wynik wyświetlić.
2. Wyświetlić składowe H, S, V obrazu po konwersji.

#### Rozliczenie ćwiczenia

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu .py lub .ipynb w odpowiednim miejscu w zasobach kursu na UPeL.



**Uwaga!** Proszę zwrócić uwagę, że w odróżnieniu np. od pakietu Matlab, tu indeksowanie jest od 0.

Przydatna składnia:

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

### 1.4.2 Matplotlib

Tu wybór dostępnych konwersji jest dość ograniczony.

1. RGB do odcieni szarości. Można wykorzystać rozbiecie na poszczególne kanały i wzór:

$$G = 0.299 \cdot R + 0.587 \cdot G + 0.144 \cdot B \quad (1.1)$$

Całość można opakować w funkcję:

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

**Uwaga!** Przy wyświetlaniu należy ustawić mapę kolorów. Inaczej obraz wyświetli się w domyślnej, która nie jest bynajmniej w odcieniach szarości: `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 Skalowanie, zmiana rozdzielczości przy użyciu OpenCV

**Ćwiczenie 1.4** Przeskaluj obraz *mandril*. Do skalowania służy funkcja `resize`.

### Rozliczenie ćwiczenia

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu `.py` lub `.ipynb` w odpowiednim miejscu w zasobach kursu na UPeL. ■

Przykład użycia:

```
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 Operacje arytmetyczne: dodawanie, odejmowanie, mnożenie, moduł z różnicą

Obrazy są macierzami, a zatem operacje arytmetyczne są dość proste – tak jak w pakiecie Matlab. Należy oczywiście pamiętać o konwersji na odpowiedni typ danych. Zwykle dobrym wyborem będzie `double`.

**Ćwiczenie 1.5** Wykonaj operacje arytmetyczne na obrazie *lena*.

1. Pobierz ze strony kursu obraz *lena*, a następnie go wczytaj za pomocą funkcji z OpenCV – dodaj ten fragment kodu do pliku, który zawiera wczytywanie obrazu *mandril*. Wykonaj konwersję do odcieni szarości. Dodaj macierze zawierające mandryla i Leny w skali szarości. Wyświetl wynik.
2. Podobnie wykonaj odjęcie i mnożenie obrazów.
3. Zaimplementuj kombinację liniową obrazów.
4. Ważną operacją jest moduł z różnicy obrazów. Można ją wykonać „ręcznie” – konwersja na odpowiedni typ, odjęcie, moduł (`abs`), konwersja na `uint8`. Alternatywa to wykorzystanie funkcji `absdiff` z OpenCV. Proszę obliczyć moduł z różnicy „szarych” wersji mandryla i Leny.

### Rozliczenie ćwiczenia

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu `.py` lub `.ipynb` w odpowiednim miejscu w zasobach kursu na UPeL. ■

**Uwaga!** Przy wyświetleniu obraz nie będzie poprawny, ponieważ jest on typu *float64 (double)*. Stosowna konwersja:

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

## 1.7 Wyliczenie histogramu

Wyliczanie histogramu można wykonać z wykorzystaniem funkcji `calcHist`. Jednak zanim do tego przejdziemy, przypomnimy sobie podstawowe struktury sterowania w Pythonie – funkcje i podprogramy. Proszę samodzielnie dokończyć poniższą funkcję wyliczającą histogram z obrazu w 256-ciu odcieniach szarości:

```
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
```

**Uwaga!** W Pythonie ważne są wcięcia, gdyż to one wyznaczają blok funkcji, czy pętli!

Histogram można wyświetlić wykorzystując funkcję `plt.hist` lub `plt.plot` z biblioteki *Matplotlib*.

Funkcja `calcHist` może policzyć histogram kilku obrazów (lub składowych), stąd jako parametry otrzymuje tablice (np. obrazów), a nie jeden obraz. Najczęściej jednak wykorzystywana jest postać:

```
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
```

Proszę sprawdzić czy histogramy uzyskane obiema metodami są takie same.

## 1.8 Wyrównywanie histogramu

Wyrównywanie histogramu to popularna i ważna operacja przetwarzania wstępnego.

1. Wyrównywanie "klasyczne" jest metodą globalną, wykonuje się na całym obrazie. W bibliotece OpenCV znajduje się gotowa funkcja do tego typu wyrównania:

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

2. Wyrównywanie CLAHE (ang. *Contrast Limited Adaptive Histogram Equalization*) – jest to metoda adaptacyjna, która poprawia warunki oświetleniowe na obrazie. Wyrównuje histogram w poszczególnych fragmentach obrazu, a nie dla całego obrazu. Metoda działa następująco:

- podział obrazu na rozłączne bloki (kwadratowe),
- wyliczanie histogramu w blokach,

- wykonanie wyrównywania histogramu, przy czym ogranicza się maksymalną „wysokość” histogramu, a nadmiar re-dystrybuuje na sąsiednie przedziały,
- interpolacja wartości pikseli na podstawie wyliczonych histogramów dla danych bloków (uwzględnia się cztery sąsiednie środki kwadratów).

Szczegóły na [Wiki](#) oraz [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)
```

**Ćwiczenie 1.6** Uruchom i porównaj obie metody wyrównywania.

#### **Rozliczenie ćwiczenia**

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu .py lub .ipynb w odpowiednim miejscu w zasobach kursu na UPeL. ■

## 1.9 Filtracja

Filtracja to bardzo ważna grupa operacji na obrazach. W ramach ćwiczenia proszę uruchomić:

- filtrację Gaussa (GaussianBlur)
- filtrację Sobela (Sobel)
- Laplasjan (Laplacian)
- medianę (medianBlur)

**Uwaga!** Pomocna będzie [dokumentacja OpenCV](#).

Proszę zwrócić uwagę również na inne dostępne funkcje:

- filtrację bilateralną,
- filtry Gabora,
- operacje morfologiczne.

**Ćwiczenie 1.7** Uruchom i porównaj wymienione metody.

#### **Rozliczenie ćwiczenia**

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu .py lub .ipynb w odpowiednim miejscu w zasobach kursu na UPeL. ■





# Laboratorium 2

<b>2</b>	<b>Detekcja pierwszoplanowych obiektów ruchomych</b>	<b>19</b>
2.1	Wczytywanie sekwencji obrazów	
2.2	Odejmowanie ramek i binaryzacja	
2.3	Operacje morfologiczne	
2.4	Indeksacja i prosta analiza	
2.5	Ewaluacja wyników detekcji obiektów pierwszoplanowych	
2.6	Przykładowe rezultaty algorytmu do detekcji ruchomych obiektów pierwszoplanowych	



## 2. Detekcja pierwszoplanowych obiektów ruchomych

### Co to są obiekty pierwszoplanowe ?

Takie, które są dla nas (dla rozważanej aplikacji) interesujące. Zwykle: ludzie, zwierzęta, samochody (lub inne pojazdy), bagaże (potencjalne bomby). Zatem definicja jest ściśle związana z docelową aplikacją.

### Czy segmentacja obiektów pierwszoplanowych to segmentacja obiektów ruchomych ?

Nie. Po pierwsze, obiekt, który się zatrzymał, nadal może być dla nas interesujący (np. stojący przed przejściem dla pieszych człowiek). Po drugie, istnieje cały szereg ruchomych elementów sceny, które nie są dla nas interesujące. Przykłady to płynąca woda, fontanna, poruszające się drzewa i krzaki itp. Należy również zauważyć, że zwykle obiekty ruchome są brane pod uwagę podczas przetwarzania obrazu. Zatem detekcję obiektów pierwszoplanowych można i warto wspomagać detekcją obiektów ruchomych.

Najprostsza metodą detekcji obiektów ruchomych to przeprowadzenie odejmowania kolejnych (sąsiednich) ramek. W ramach ćwiczenia zrealizujemy proste odejmowanie dwóch ramek, połączone z binaryzacją, indeksacją i analizą otrzymanych obiektów. Na koniec spróbujemy uznać wynik odejmowania jako rezultat segmentacji obiektów pierwszoplanowych i sprawdzimy jakość tej segmentacji.

### 2.1 Wczytywanie sekwencji obrazów

#### Ćwiczenie 2.1 Wczytywanie sekwencji obrazów

1. Ściągnij odpowiednią sekwencję testową z platformy UPeL. W ćwiczeniu skupimy się na sekwencji *pedestrians*. Wykorzystywane sekwencje pochodzą ze zbioru danych ze strony [changedetection.net](http://changedetection.net). Zbiór zawiera poetykietowane sekwencje, tzn. każda ramka

obrazu posiada maskę obiektów - maskę referencyjną (ang. *ground truth*). Na nich każdy z pikseli został przydzielony do jednej z pięciu kategorii: tło (0), cień (50), poza obszarem zainteresowania (85), nieznany ruch (170) oraz obiekty pierwszoplanowe (255) – w nawiasach podano odpowiadające poziomy szarości.

W ramach ćwiczenia interesować nas będzie tylko podział na obiekty pierwszoplanowe oraz pozostałe kategorie. Dodatkowo w folderze znajduje się maska obszaru zainteresowania (ROI) oraz plik tekstowy z przedziałem czasowym dla którego należy analizować wyniki (temporalROI.txt) – szczegóły w dalszej części ćwiczenia.

2. Wczytanie sekwencji umożliwia następujący, przykładowy kod:

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

- R** Zakłada się, że sekwencja została rozpakowana w tym samym folderze, co plik źródłowy (w innym przypadku trzeba dodać odpowiednią ścieżkę).

Należy użyć funkcji `waitKey`. Inaczej nie nastąpi odświeżenie wyświetlanego obrazka.

3. Do pętli dodaj opcję analizy co i-tej ramki – wykorzystywana funkcja *range* może mieć jako trzeci parametr: *krok*. Poeksperymentuj z jego wartością teraz (przy wyświetlaniu filmu) i później (przy detekcji obiektów ruchomych).



## 2.2 Odejmowanie ramek i binaryzacja

W celu detekcji elementu ruchomych, odejmujemy dwie kolejne ramki. Należy zaznaczyć, że dokładnie chodzi nam o obliczenie modułu z różnicy.

- R** Aby uniknąć problemów z odejmowaniem liczb bez znaku (*uint8*), należy wykonać konwersję do typu *int* – `IG = IG.astype('int')`.

Dla uproszczenia rozważań lepiej wcześniej dokonać konwersji do odcieni szarości. Na początku trzeba „jakoś” obsłużyć pierwszą iterację. Przykładowo – wczytać pierwszą ramkę przed pętlą i uznać ją za poprzednią. Później na końcu pętli należy dodać przypisanie ramka poprzednia = ramka bieżąca. Testowo wyświetlamy wyniki odejmowania – powinny być widoczne głównie krawędzie.

Binaryzację można wykonać wykorzystując następującą składnię:

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

- R** W takim przypadku `1*` oznacza konwersję z typu logicznego na liczbowy. Osobną kwestią jest poprawne wyświetlenie – trzeba zmienić zakres (pomnożyć przez 255) i dokonać konwersji na *uint8* (możemy być potrzebny import biblioteki *numpy*).

Próg należy dobrać, tak aby obiekty były względnie wyraźne. Proszę zwrócić uwagę na artefakty związane z kompresją.

Alternatywa to użycie funkcji wbudowanej w 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** Pierwszy argument zwracanych wartości przez funkcję `cv2.threshold` to próg binaryzacji (jest on użyteczny w przypadku stosowania automatycznego wyznaczenia progu np. metodą Otsu lub trójkątów). Szczegóły w dokumentacji OpenCV.

## 2.3 Operacje morfologiczne

**Ćwiczenie 2.2** Uzyskany obraz jest dość zaszumiony. Proszę wykonać filtrację wykorzystując erozję i dylację (`erode` i `dilate` z OpenCV).

- R** Celem tego etapu jest uzyskanie maksymalnie widocznej sylwetki, przy minimalnych zakłóce- niach. Dla poprawy efektu warto dodać jeszcze etap filtracji medianowej (przed morfologią) oraz ew. skorygować próg binaryzacji.

## 2.4 Indeksacja i prosta analiza

W kolejnym etapie przeprowadzimy filtrację uzyskanego wyniku. Wykorzystamy indeksację (nada- nie etykiet dla grup połączonych pikseli) oraz obliczanie parametrów tychże grup. Służy do tego funkcja `connectedComponentsWithStats`. Wywołanie:

```
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** Przy wyświetlaniu obrazu *labels* trzeba w odpowiedni sposób ustawić format oraz dodać skalowanie. Do skalowania można wykorzystać informację o liczbie znalezionych obiektów.

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

Następnie wyświetl prostokąt otaczający, pole i indeks dla największego obiektu. Poniżej znajduje się przykładowe rozwiązanie zadania. Proszę go uruchomić i ewentualnie spróbować go zoptymalizować.

```
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))

```

Komentarze do przykładowego rozwiązania:

- `stats.shape[0]` to liczba obiektów. Ponieważ funkcja zlicza też obiekt o indeksie 0 (tj. tło), w sprawdzeniu czy są obiekty warunek jest `> 1`
- kolejne dwie linie to obliczenie indeksu maksimum z kolumny numer 4 (pole).
- uzyskany indeks należy inkrementować, ponieważ w analizie pominęliśmy element 0 (tło).
- do rysowania prostokąta otaczającego na obrazie wykorzystujemy funkcję `rectangle` z OpenCV. Składnia `"%f" % stats[pi,4]` pozwala wypisać wartość w odpowiednim formacie (f - float). Kolejne parametry to współrzędne dwóch przeciwległych wierzchołków prostokąta. Następnie kolor w formacie (B, G, R), a na końcu grubość linii. Szczegóły w dokumentacji funkcji.
- do wypisywania tekstu na obrazie służy funkcja `putText`. Do określenia pozycji pola tekstowego podaje się współrzędne lewego dolnego wierzchołka. Następnie czcionka (pełna lista w dokumentacji), rozmiar i kolor.
-  `I_VIS` to obraz wejściowy jeszcze przed konwersją do odcieni szarości.

**Ćwiczenie 2.3** Wykorzystaj funkcję `cv2.connectedComponentsWithStats` do indeksacji oraz oblicz parametry otrzymanych obiektów. Wyświetl prostokąt otaczający, pole oraz indeks dla największego obiektu. Na podstawie wskazówek i przykładów zamieszczonych w tekście powyżej spróbuj zoptymalizować rozwiązanie. ■

## 2.5 Ewaluacja wyników detekcji obiektów pierwszoplanowych

Aby ocenić, i to najlepiej w miarę obiektywnie, algorytm detekcji obiektów pierwszoplanowych należy wyniki przez niego zwracane tj. maskę obiektów porównać z maską referencyjną (ang. *groundtruth*). Porównywanie odbywa się na poziomie poszczególnych pikseli. Jeśli wykluczy się cienie (a tak założyliśmy na wstępie) to możliwe są cztery sytuacje:

- *TP* – wynik prawdziwie dodatni (ang. *true positive*) – piksel należący do obiektu z pierwszego planu jest wykrywany jako piksel należący do obiektu z pierwszego planu,
- *TN* – wynik prawdziwie ujemny (ang. *true negative*) – piksel należący do tła jest wykrywany jako piksel należący do tła,
- *FP* – wynik fałszywie dodatni (ang. *false positive*) – piksel należący do tła jest wykrywany jako piksel należący do obiektu z pierwszego planu,
- *FN* – wynik fałszywie ujemny (ang. *false negative*) – piksel należący do obiektu jest wykrywany jako piksel należący do tła.

Na podstawie wymienionych współczynników można policzyć szereg miar. My wykorzystamy trzy: precyzję (ang. *precision* - *P*), czułość (ang. *recall* - *R*) i tzw. *F1*. Zdefiniowane są one następująco:

$$P = \frac{TP}{TP + FP} \quad (2.1)$$

$$R = \frac{TP}{TP + FN} \quad (2.2)$$

$$F1 = \frac{2PR}{P + R} \quad (2.3)$$

Miara F1 jest z zakresu [0;1], przy czym im jej wartość jest większa, tym lepiej.

#### Ćwiczenie 2.4 Zaimplementuj obliczanie miar $P$ , $R$ i $F1$ .

1. W pierwszym kroku należy zdefiniować globalne liczniki  $TP$ ,  $TN$ ,  $FP$ ,  $FN$ , a po skończonej pętli obliczyć żądane wartości  $P$ ,  $R$  i  $F1$ .
2. Do pętli należy dodać wczytywanie maski referencyjnej – analogicznie jak obrazka. Jest ona dostępna w folderze *groundtruth*.
3. Kolejny krok to porównanie maski obiektów i referencyjnej. Najprostsze rozwiązanie, czyli pętla `for` po całym obrazie, w każdej iteracji sprawdzamy podobieństwo pikseli. Oczywiście nie jest to zbyt wydajne podejście. Można spróbować wykorzystać możliwości Python'a w realizacji operacji na macierzach. Utwórz odpowiednie warunki logiczne np. dla  $TP$ :

```
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
```

Przy czym obliczenia wykonujemy tylko wtedy, gdy dostępna jest poprawna mapa referencyjna. W tym celu należy sprawdzić zależność licznika ramki i wartości z pliku `temporalROI.txt` – musi się on zawierać w zakresie tam opisanym. Przykładowy kod wczytujący zakres (przy okazji proszę zwrócić uwagę na sposób obsługi plików tekstowych):

```
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. Przeprowadź obliczenia dla metody odejmowania kolejnych klatek. Zwróć uwagę na wartość  $F1$ .
5. Wykonaj obliczenia dla pozostałych dwóch sekwencji – *highway* i *office*.

#### Rozliczenie ćwiczenia

W celu rozliczenia ćwiczenia, po wykonaniu zadania zgłoś swoją gotowość prowadzącemu zajęcia. Równocześnie możesz zgłosić swoją gotowość po wykonaniu wszystkich zadań dotyczących poruszanego tematu zajęć. Po akceptacji z jego strony, umieść skrypt o rozszerzeniu `.py` lub `.ipynb` w odpowiednim miejscu w zasobach kursu na UPeL.

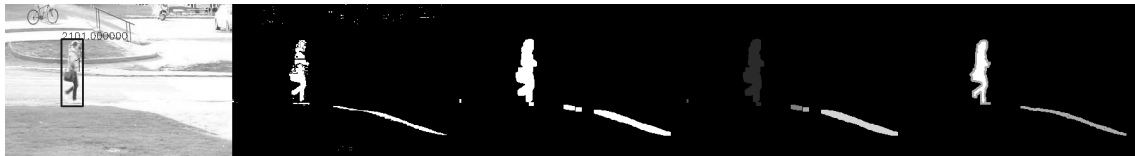
#### Informacje dotyczące kolejnych zajęć.

1. W ramach dalszych ćwiczeń będziemy poznawać kolejne algorytmy i funkcjonalności języka Python. Jednak nie kładziemy szczególnej uwagi na poznawanie języka Python, ale na wykorzystaniu w praktyce kolejnych algorytmów pojawiających się na wykładach. Przedmiot Zaawansowane Algorytmy Wizyjne nie jest kursem Pythona!

2. Przedstawione rozwiązania należy zawsze traktować jako przykładowe. Na pewno problem można rozwiązać inaczej, a czasem lepiej.
3. W kolejnych ćwiczeniach poziom szczegółowości opisu rozwiązania będzie maleć. Zachęcamy zatem do aktywnego korzystania z dokumentacji do Pythona, OpenCV i materiałów/tutoriali znajdujących się w internecie :).

## 2.6 Przykładowe rezultaty algorytmu do detekcji ruchomych obiektów pierwszoplanowych

W celu lepszej weryfikacji poszczególnych etapów zaproponowanej metody do detekcji ruchomych obiektów pierwszoplanowych przedstawiony zostanie przykładowy wynik dla obrazu o indeksie 350.



Rysunek 2.1: Przykładowy wynik poszczególnych etapów algorytmu. Od lewej obraz wejściowy z prostokątem otaczającym, obraz po binaryzacji, obraz po zastosowaniu filtracji medianowej oraz operacji morfologicznych (erode, dilate), obraz reprezentujący etykiety po indeksacji, obraz referencyjny.





# Laboratorium 3

<b>3</b>	<b>Segmentacja obiektów pierwszoplanowych</b>	<b>27</b>
3.1	Cel zajęć	
3.2	Segmentacja obiektów pierwszoplanowych	
3.3	Metody oparte o bufor próbek	
3.4	Aproksymacja średniej i mediany (tzw. sigma-delta)	
3.5	Polityka aktualizacji	
3.6	OpenCV – GMM/MOG	
3.7	OpenCV – KNN	
3.8	Przykładowe rezultaty algorytmu do segmentacji obiektów pierwszoplanowych	
3.9	Wykorzystanie sieci neuronowej do segmentacji obiektów pierwszoplanowych	
3.10	Zadania dodatkowe	



## 3. Segmentacja obiektów pierwszoplanowych

### 3.1 Cel zajęć

- zapoznanie z zagadnieniem segmentacji obiektów pierwszoplanowych oraz problemami z tym związanymi,
- implementacja prostych algorytmów modelowania tła opartych o bufor próbek – analiza ich wad i zalet,
- implementacja algorytmów średniej bieżącej oraz aproksymacji medianowej – analiza ich wad i zalet,
- poznanie metod dostępnych w *OpenCV* – Gaussian Mixture Model (zwanej też Mixture of Gaussians) oraz metody opartej o algorytm KNN (K-Nearest Neighbours, K-najbliższych sąsiadów),
- zapoznanie się z architekturą sieci neuronowej oraz przykładową aplikacją.

### 3.2 Segmentacja obiektów pierwszoplanowych

**Co to jest segmentacja obiektów pierwszoplanowych (ang. *foreground object segmentation*)**

Segmentacja to wyodrębnienie pewnej grupy obiektów obecnych na obrazie. Należy zaznaczyć, że niekoniecznie musi być ona tożsama z klasyfikacją, gdzie wyjściem jest przypisanie obiektu do konkretnej klasy: np. samochód (a nawet typ), pieszy itp. Definicja obiektu pierwszoplanowego nie jest bardzo ścisła – są to wszystkie obiekty istotne dla danej aplikacji. Przykładowo dla monitoringu wizyjnego: ludzie, samochody i może zwierzęta.

**Co to jest model tła ?**

W najprostszym przypadku jest to obraz „pustej sceny”, tj. bez interesujących nas obiektów. Uwaga. W bardziej zaawansowanych algorytmach, model tła nie ma postaci „jawnej” (nie jest to obraz) i nie można go po prostu wyświetlić (przykłady takich metod: GMM, ViBE, PBAS).

### Inicjalizacja modelu tła

W przypadku uruchomienia algorytmu (także ponownego uruchomienia), konieczne jest wykonanie inicjalizacji modelu tła tj. ustalenie wartości wszystkich parametrów (zmiennych), które stanowią ten model. W najprostszym przypadku, za początkowy model tła, przyjmuje się pierwszą ramkę z analizowanej sekwencji (pierwsze N-ramek w przypadku metod z buforem). Podejście to sprawdza się dobrze, przy założeniu, że na tej konkretnej ramce (sekwencji ramek) nie występują obiekty z pierwszego planu. W przeciwnym przypadku początkowy model będzie zawierał błędy, które będą mniej lub bardziej trudne do eliminacji w dalszym działaniu — zależy to od konkretnego algorytmu. W literaturze przedmiotu zagadnienie to określa się mianem *bootstrap*.

Bardziej zaawansowane metody inicjalizacji polegają na analizie czasowej, a nawet przestrzennej pikseli dla pewnej początkowej sekwencji. Zakłada się tutaj, że tło jest widoczne przez większość czasu oraz jest bardziej „spójne przestrzennie” niż obiekty.

### Modelowanie tła, generacja tła

Powstaje pytanie, czy nie wystarczy pobrać obrazu pustej sceny, zapisać go i używać w trakcie całego działania algorytmu? W ogólnym przypadku nie, ale zacznijmy od przypadku szczególnego. Jeśli nasz system nadzoruje pomieszczenie, w którym oświetlenie jest ściśle kontrolowane, a jakiegokolwiek uprzednio nieprzewidziane i zatwierdzone zmiany nie powinny mieć miejsca (np. zabroniona strefa wewnątrz elektrowni, skarbiec banku itp.), to najprostsze podejście ze statycznym tłem się sprawdzi.

Problemy pojawiają się w przypadku, gdy oświetlenie sceny ulega zmianie, czy to z powodu zmiany pory dnia, czy zapalenia dodatkowego światła itp. Wtedy „rzeczywiste tło” ulegnie zmianie i nasz statyczny model tła nie będzie mógł się do tej zmiany dostosować. Drugim, trudniejszym przypadkiem jest zmiana położenia elementów sceny: np. ktoś przestawi ławkę/krzesło/kwiatek. Zmiana ta nie powinna być wykrywana, a przynajmniej po dość krótkim czasie uwzględniona w modelu tła. Inaczej będzie przyczyną generowania fałszywych detekcji (znanych pod nazwą *ghost* – duch), a w konsekwencji fałszywych alarmów.

Wniosek z rozważań jest następujący. Dobry model tła powinien charakteryzować się zdolnością adaptacji do zmian na scenie. Zjawisko aktualizacji modelu określa się w literaturze mianem generacji tła lub modelowania tła.

### Pałapki przy modelowaniu tła

Rozważane zagadnienie nie jest proste. Na przestrzeni lat powstało wiele metod, ale nie jest dostępna jedna uniwersalna i skuteczna w każdych okolicznościach. W literaturze wyróżnia się następujące sytuacje „problematiczne”:

- szum na obrazie (nadal obecny pomimo znacznych postępów w technice produkcji czujników wizyjnych, szczególnie widoczny przy słabym oświetleniu), do tej kategorii warto też zaliczyć artefakty związane z kompresją cyfrowego strumienia wideo (MJPEG, H.264/265, MPEG),
- drżenie kamery – dobrym przykładem są kamery monitorujące ruch drogowy zamontowane na słupach sygnalizacji świetlnej,
- automatyczne nastawy kamery (balans bieli, korekcja ekspozycji) – np. jeśli na scenie pojawi się względnie duży jasny obiekt, to „automatyka” spowoduje zmianę ekspozycji i/lub balansu bieli dla całej sceny, co zwykle kończy się fatalnie dla algorytmu detekcji obiektów,
- pora dnia (ogólnie płynne zmiany oświetlenia),
- nagłe zmiany oświetlenia (włączenie światła w pomieszczeniu, słoneczno-pochmurny dzień),
- obecność obiektów podczas inicjalizacji modelu tła (skomentowane wcześniej),

- ruchome elementy tła (skomentowane wcześniej),
- kamuflaż — duże podobieństwo obiektów do tła pod względem koloru lub tekstury (tu najnowszy trend to wykorzystanie informacji 3D),
- poruszone obiekty w tle (krzesła itp.)

Ostatnie i chyba najtrudniejsze zagadnienia to wtapienie się obiektów z pierwszego planu w tło oraz tak zwane duchy (ang. *ghost*). Oba zjawiska są ze sobą połączone. Rozważmy następującą sytuację. Samochód wjeżdża na parking i zatrzymuje się. Początkowo jest wykrywany jako obiekt, po ale po pewnym czasie (zależnym od metody) stanie się elementem tła. Następnie samochód odjedzie. Puste miejsce po nim będzie wykrywane jako obiekt, gdyż znacząco różni się od bieżącej ramki. Powstaje *ghost*, czyli obiekt, który nie istnieje na bieżącej ramce obrazu, a tylko w modelu tła.

### Polityka aktualizacji

Jak zostało wcześniej stwierdzone, podstawą modelowania (generacji) tła, jest aktualizacja modelu tła. Wyróżnia się dwa skrajne podejścia do zagadnienia: **konserwatywne** i **liberalne**. W pierwszym przypadku aktualizujemy tylko te piksele, które zostały sklasyfikowane jako **tło**. W drugim aktualizacja wykonywana jest **dla wszystkich pikseli**. Oczywiście występuje też cały szereg podejść pośrednich.

Warto zwrócić uwagę na skutki obu polityk. W przypadku konserwatywnej łatwo narażamy się na sytuację, w której błędna klasyfikacja będzie podtrzymywana w nieskończoność, gdyż ten obszar nigdy nie będzie aktualizowany. Z drugiej strony podejście liberalne spowoduje wtapienie obiektów z pierwszego planu do modelu tła i może prowadzić do powstawania *ghost'ów* lub smug ciągnących się za obiektami.

### Cienie

Cienie są bardzo specyficzną kategorią w zagadnieniach detekcji obiektów. Człowiek podświadomie cienie pomija, tj. jak analizujemy scenę to raczej specjalnie nie zwracamy uwagi na to czy np. sylwetka rzuca cień, czy nie. W szczególności dotyczy to cieni słabo widocznych.

W systemie detekcji cień spełnia wszystkie założenia obiektu z pierwszego planu i raczej ciężko go uznać za element tła. Rozważania dotyczą cieni rzucanych przez obiekty pierwszoplanowe. Problem polega na tym, że z punktu widzenia działania dalszych etapów tj. np. klasyfikacji obiektów, stanowi on zakłócenie – mocno utrudnia analizę np. poprzez zmianę kształtu obiektu lub łączenie obiektów.

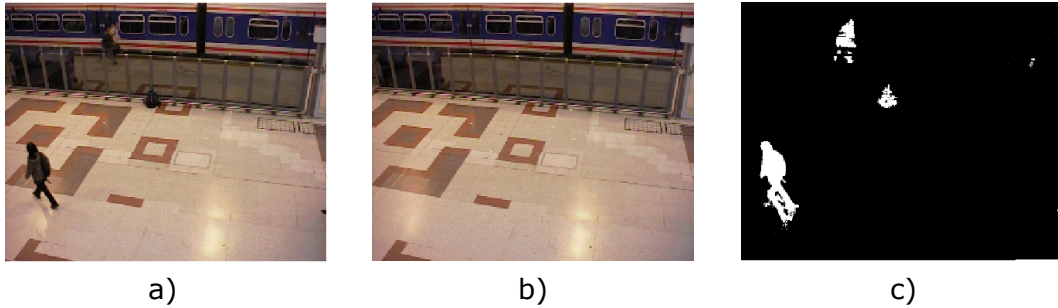
Z wyżej wymienionych powodów prowadzone są badania nad detekcją i eliminacją cieni. Na potrzeby niniejszego ćwiczenia należy zapamiętać, że cień powinien stanowić osobną kategorię - nie jest on ani obiektem pierwszoplanowym, ani elementem tła.

### Przykład

Na rysunku 3.1 przedstawiono przykład segmentacji obiektów z wykorzystaniem modelowania tła. Warto zwrócić na wykryty cień pod nogami osoby po lewej stronie ekranu oraz osobę za barierką (raczej mało widoczną na bieżącej ramce).

## 3.3 Metody oparte o bufor próbek

W ramach ćwiczenia zaprezentowane zostaną dwie metody: średnia z bufora oraz mediana z bufora. Rozmiar bufora przyjmiemy na  $N = 60$  próbek. W każdej iteracji algorytmu należy usunąć ostatnią ramkę z bufora, dodać bieżącą (bufor działa na zasadzie kolejki FIFO) oraz obliczyć średnią lub medianę z bufora.



Rysunek 3.1: Przykład segmentacji obiektów pierwszoplanowych z wykorzystaniem modelowania tła. a) bieżąca ramka, b) model tła, c) maska obiektów pierwszoplanowych. Źródło: sekwencja [PETS 2006](#)

1. Na początku należy zadeklarować bufor o rozmiarze  $N \times YY \times XX$  (polecenie np. `zeros`), gdzie  $YY$  i  $XX$  to odpowiednio wysokość i szerokość ramki z sekwencji *pedestrians*.

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

Należy też zainicjalizować licznik dla bufora (niech się nazywa np. `iN`) wartością 0. Parametr `iN` jest jak wskaźnik, który wskazuje na miejsce w buforze, z którego należy usunąć ostatnią ramkę oraz przypisać obecną ramkę.

2. Obsługa bufora powinna być następująca. W pętli pod adresem `iN` należy zapisać bieżącą ramkę w skali szarości.

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

Następnie licznik `iN` należy inkrementować i sprawdzać czy nie osiąga rozmiaru bufora ( $N$ ). Jeśli tak, to trzeba go ustawić na 0.

3. Obliczenie średniej lub mediany realizuje się funkcjami `mean` lub `median` z biblioteki *numpy*. Jako parametr podaje się wymiar, dla którego ma być liczona wartość (pamiętać o indeksowaniu od zera). Aby wszystko działało poprawnie, należy dokonać konwersji wyniku na `uint8`.
4. Ostatecznie realizujemy odejmowanie tła tj. od bieżącej sceny odejmujemy model, a wynik binaryzujemy – analogicznie jak przy różnicy sąsiednich ramek. Również wykorzystujemy filtrację medianową maski obiektów lub/i operacje morfologiczne.
5. Wykorzystując stworzony w ramach poprzedniego ćwiczenia kod, porównaj działanie metody ze średnią i medianą. Zanotuj wartości wskaźnika  $F1$  dla obu przypadków.

**R** Mediana może liczyć się przez dłuższą chwilę.

Zastanów się dlaczego mediana działa lepiej. Sprawdź działanie na innych sekwencjach – w szczególności dla sekwencji *office*. Zaobserwuj zjawisko smużenia oraz wtapiania się sylwetki do tła oraz powstania *ghosta*.

**Ćwiczenie 3.1** Zaimplementuj algorytm na podstawie powyższego opisu. ■

### 3.4 Aproksymacja średniej i mediany (tzw. sigma-delta)

Użycie bufora na próbki wymaga sporych zasobów pamięciowych. Poza tym, o ile średnią da się policzyć w prosty sposób (proszę się zastanowić jak można zaktualizować średnią z bufora bez

ponownego liczenia dla wszystkich elementów), to już z medianą są problemy i bardzo szybkie algorytmy jej wyznaczania nie istnieją (aczkolwiek nie trzeba sortować wszystkich elementów w każdej iteracji). Dlatego chętniej wykorzystuje się metody, które nie wymagają bufora. W przypadku aproksymacji średniej wykorzystuje się zależność:

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

gdzie:  $I_N$  – bieżąca ramka,  $BG_N$  – model tła,  $\alpha$  – parametr wagowy, zwykle 0.01 – 0.05, choć wartość zależy od konkretnej aplikacji.

Aproksymację mediany uzyskuje się wykorzystując zależność:

$$\begin{aligned} \text{if } BG_{N-1} < I_N & \text{ then } BG_N = BG_{N-1} + 1 \\ \text{if } BG_{N-1} > I_N & \text{ then } BG_N = BG_{N-1} - 1 \\ \text{otherwise } & BG_N = BG_{N-1} \end{aligned} \quad (3.2)$$

### Ćwiczenie 3.2 Zaimplementuj obie metody.

1. Jako pierwszy model tła przyjmij pierwszą ramkę z sekwencji. Zanotuj wartość wskaźnika  $F1$  dla tych dwóch metod (parametr  $\alpha$  ustal na 0.01). Zaobserwuj czas działania.
2. Poeksperymentuj z wartością parametru  $\alpha$ . Zobacz jaki ona ma wpływ na model tła.

**R** Dla metody średniej bieżącej bardzo ważne jest, aby model tła był zmiennoprzecinkowy (*float64*).

W przypadku wzoru 3.1 uniknięcie stosowania pętli po całym obrazie jest oczywiste. Dla zależności 3.2 również jest to możliwe, ale wymaga chwili zastanowienia. Warto korzystać z dokumentacji *numpy*. Dodatkowo proszę zauważyć, że w Pythonie można wykonywać działania arytmetyczne na wartościach boolowskich – `False == 0` natomiast `True == 1`.

## 3.5 Polityka aktualizacji

Jak dotąd stosowaliśmy liberalną politykę aktualizacji – aktualizowaliśmy wszystko. Spróbujemy teraz wykorzystać podejście konserwatywne.

### Ćwiczenie 3.3 Polityka aktualizacji.

1. Dla wybranej metody zaimplementuj konserwatywne podejście do aktualizacji. Sprawdź jego działanie.

**R** Wystarczy zapamiętać poprzednią maskę obiektów i odpowiednio wykorzystać ją w procedurze aktualizacji.

2. Zwróć uwagę na wartość wskaźnika  $F1$  oraz na model tła. Czy pojawiły się w nim jakieś błędy ?

## 3.6 OpenCV – GMM/MOG

W bibliotece OpenCV dostępna jest jedna z najpopularniejszych metod segmentacji obiektów pierwszoplanowych: Gaussian Mixture Models (GMM) lub Mixture of Gaussians (MoG) — obie nazwy występują równolegle w literaturze. W największym skrócie: scena jest modelowana za

pomocą kilku rozkładów Gaussa (średnia jasności (koloru) i odchylenie standardowe). Każdy rozkład opisany jest również przez wagę, która oddaje to jak często był on używany (obserwowany na scenie – prawdopodobieństwo wystąpienia). Rozkłady o największych wagach stanowią tło. Segmentacja polega na obliczaniu odległości pomiędzy bieżącym pikselem, a każdym z rozkładów. Następnie jeśli piksel jest „podobny” do rozkładu uznanego za tło, to jest klasyfikowany jako tło, w innym przypadku uznany zostaje za obiekt. Model tła podlega również aktualizacji w sposób zbliżony do równania 3.1. Zainteresowane osoby odsyłamy do literatury np. [do artykułu](#).

**Ćwiczenie 3.4** Metoda ta jest przykładem algorytmu wielowariantowego – model tła ma kilka możliwych reprezentacji (wariantów).

1. Otwórz dokumentację do [OpenCV](#). Przejdź do *Video Analysis->Motion Analysis*. Interesuje nas klasa `BackgroundSubtractorMOG2`. Aby utworzyć obiekt tej klasy w Python3 należy użyć `createBackgroundSubtractorMOG2`. Korzystając z tego obiektu w pętli dla każdego obrazu wykorzystujemy jego metodę `apply`.
2. Na początek uruchomimy kod z wartościami domyślnymi.
3. Proszę poeksperymentować z parametrami: *history*, *varThreshold* oraz wyłączyć detekcję cieni. W metodzie `apply()` można też „ręcznie” ustalić współczynnik uczenia – *learningRate*

**R** W pakiecie OpenCV dostępna jest wersja GMM nieco inna niż oryginalna – między innymi wyposażona w moduł detekcji cieni oraz dynamicznie zmienianą liczbę rozkładów Gaussa. Odnośniki do stosownych artykułów dostępne są w dokumentacji OpenCV.

4. Wyznacz parametr F1 (dla metody bez detekcji cieni). Zaobserwuj jak działa metoda. Zwróć uwagę, że nie da się „wyświetlić” modelu tła.

### 3.7 OpenCV – KNN

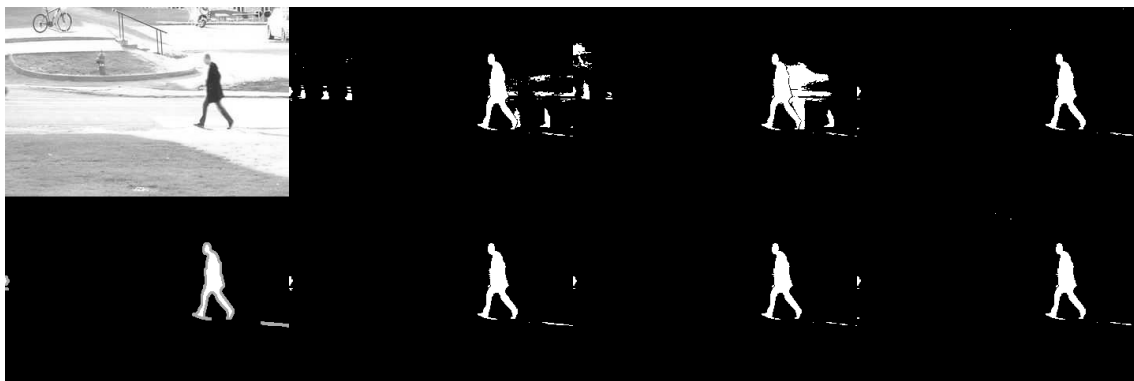
Dругa z metod dostępnych w OpenCV to KNN (`BackgroundSubtractorKNN`).

**Ćwiczenie 3.5** Proszę uruchomić metodę KNN i dokonać jej ewaluacji.

### 3.8 Przykładowe rezultaty algorytmu do segmentacji obiektów pierwszoplanowych

W celu lepszej weryfikacji poszczególnych etapów zaproponowanych metod do segmentacji obiektów pierwszoplanowych przedstawione zostaną przykładowe wyniki dla wybranych ramek obrazu.

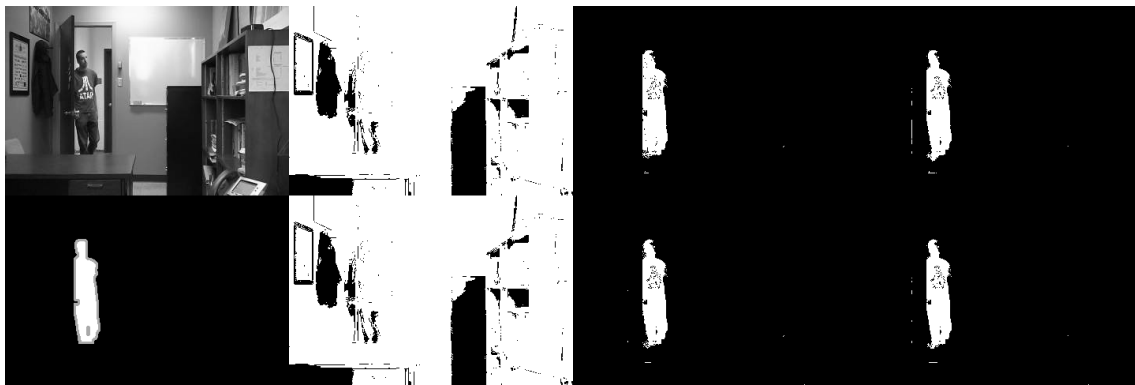




Rysunek 3.2: Przykładowy wynik poszczególnych wersji algorytmu do segmentacji obiektów pierwszoplanowych dla zbioru *pedestrian*. Indeks ramki to 600. Pierwsza od lewej kolumna to obraz wejściowy wraz z obrazem referencyjnym. Pierwszy wiersz dotyczy algorytmów wykorzystujących średnią, drugi wiersz wykorzystuje medianę. Kolejno od drugiej kolumny prezentowany jest wynik algorytmu: liberalna polityka aktualizacji, następnie aproksymacja funkcji z liberalną polityką aktualizacji oraz aproksymacja funkcji z konserwatywną polityką aktualizacji.



Rysunek 3.3: Przykładowy wynik poszczególnych wersji algorytmu do segmentacji obiektów pierwszoplanowych dla zbioru *highway*. Indeks ramki to 1200. Pierwsza od lewej kolumna to obraz wejściowy wraz z obrazem referencyjnym. Pierwszy wiersz dotyczy algorytmów wykorzystujących średnią, drugi wiersz wykorzystuje medianę. Kolejno od drugiej kolumny prezentowany jest wynik algorytmu: liberalna polityka aktualizacji, następnie aproksymacja funkcji z liberalną polityką aktualizacji oraz aproksymacja funkcji z konserwatywną polityką aktualizacji.



Rysunek 3.4: Przykładowy wynik poszczególnych wersji algorytmu do segmentacji obiektów pierwszoplanowych dla zbioru *office*. Indeks ramki to 600. Pierwsza od lewej kolumna to obraz wejściowy wraz z obrazem referencyjnym. Pierwszy wiersz dotyczy algorytmów wykorzystujących średnią, drugi wiersz wykorzystuje medianę. Kolejno od drugiej kolumny prezentowany jest wynik algorytmu: liberalna polityka aktualizacji, następnie aproksymacja funkcji z liberalną polityką aktualizacji oraz aproksymacja funkcji z konserwatywną polityką aktualizacji.

### 3.9 Wykorzystanie sieci neuronowej do segmentacji obiektów pierwszoplanowych

Do segmentacji obiektów pierwszoplanowych można wykorzystać również metody oparte na architekturze sieci neuronowej. Przykładem takiego rozwiązania jest praca<sup>1</sup>. Zaproponowanym podejściem jest wykorzystanie konwolucyjnej sieci neuronowej zwanej BSUV-Net (ang. Background Subtraction Unseen Videos Net). Architektura bazuje na strukturze U-Net z połączeniami resztkowymi (ang. residual connections), która dzieli się na enkoder oraz dekoder. Wejście to konkatenacja kilku obrazów, tj. bieżąca ramka i dwie ramki tła w różnych odstępach czasowych wraz z ich semantycznymi mapami segmentacji. Wyjściem jest maska zawierająca tylko obiekty pierwszoplanowe.

Szczegóły odnośnie tej sieci neuronowej znajdziecie w artykule: <https://arxiv.org/abs/1907.11371>. Architektura z przykładowymi danymi oraz wytrenowanymi modelami sieci jest udostępniona na GitHubie: <https://github.com/ozantezcan/BSUV-Net-inference>.

**Ćwiczenie 3.6** Uruchom konwolucyjną sieć neuronową BSUV-Net.

- Ściągnij z platformy UPEL wszystkie pliki do sieci neuronowej.
- Wykorzystana zostanie baza danych pedestrians, ale odpowiednio przeskalowana do rozmiaru wejścia sieci oraz przekonwertowana na sekwencje wideo,
- Otwórz plik `infer_config_manualBG.py` oraz zmodyfikuj ścieżki do:
  - w klasie `SemanticSegmentation` podaj ścieżkę absolutną do folderu `segmentation`
  - zmienna `root_path`,
  - w klasie `BSUVNet` podaj ścieżkę do modelu sieci `BSUV-Net-2.0.mdl` w folderze `trained_models` – zmienna `model_path`,
  - w klasie `BSUVNet` podaj ścieżkę do obrazu, który zawiera jedynie tło – pierwsze zdjęcie ze zbioru `pedestrians` – zmienna `empty_bg_path`
- W pliku `inference.py` podaj ścieżkę do wejścia sieci, czyli sekwencja wideo `pedestrians` – zmienna `inp_path` oraz ścieżkę do miejsca gdzie ma być zapisane wyjście sieci wraz z nazwą pliku – zmienna `out_path`.

### 3.10 Zadania dodatkowe

#### 3.10.1 Inicjalizacja modelu tła w warunkach obecności obiektów pierwszoplanowych na scenie

**Ćwiczenie 3.7 Zadanie dodatkowe 1**

Jeśli weźmiemy metodę, w której zastosowaliśmy konserwatywne podejście do aktualizacji oraz rozpoczniemy analizę sekwencji nie od ramki nr 1, a od 1000 to zobaczymy w praktyce wady założenia, że pierwsza ramka z sekwencji jest początkowym modelem tła (*ghosty* itp.) Zaproponuj rozwiązanie problemu. W aplikacji należy zademonstrować przewagę własnego podejścia nad inicjalizacją na podstawie pierwszej ramki.

Na rysunku 3.5 przedstawiono przykładowe rozwiązanie problemu.

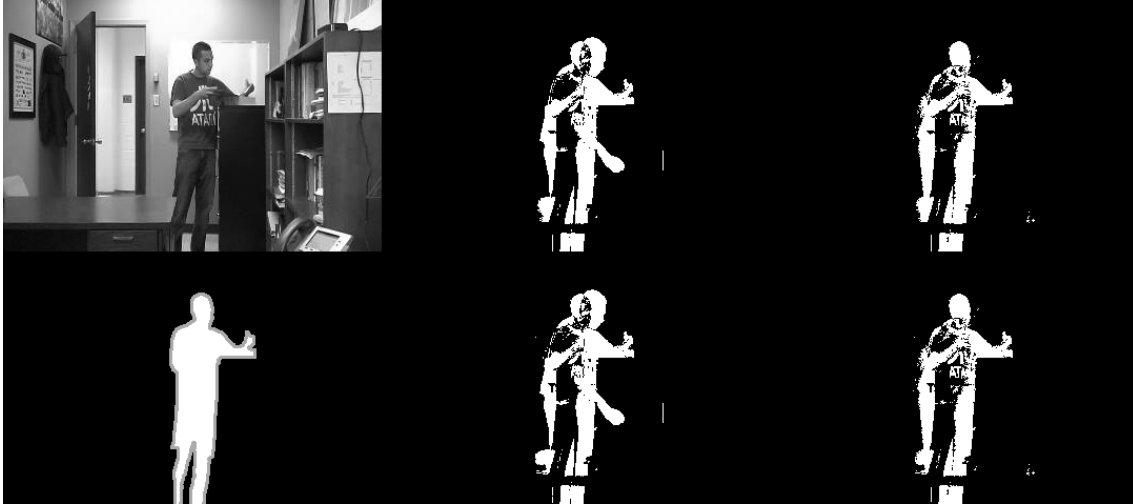


Dobrym pomysłem wydaje się zbuforowanie np. 30 lub więcej ramek oraz przeprowadzenie analizy częstotliwości występowania każdego odcienia jasności w sekwencji – niezależnie dla

<sup>1</sup>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>

każdego piksela. Za tło należy uznać, ten co był obserwowany najczęściej. Inne, skuteczne pomysły, są również mile widziane.

W zadaniu można wykorzystać inną sekwencję.



Rysunek 3.5: Przykładowy wynik proponowanego rozwiązania dla zbioru *office*. Indeks ramki to 800. Pierwsza od lewej kolumna to obraz wejściowy wraz z obrazem referencyjnym. Pierwszy wiersz dotyczy algorytmów wykorzystujących średnią, drugi wiersz wykorzystuje medianę. Kolejno od drugiej kolumny prezentowany jest wynik algorytmu: aproksymacja funkcji z konserwatywną polityką aktualizacji bez buforu inicjalizacyjnego, aproksymacja funkcji z konserwatywną polityką aktualizacji z buforem inicjalizacyjnym.

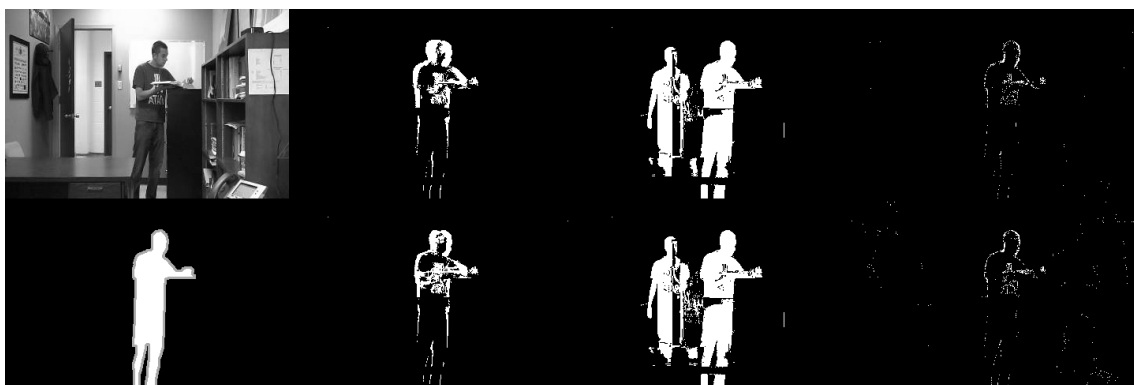
### 3.10.2 Implementacja metod ViBE i PBAS

#### Ćwiczenie 3.8 Zadanie dodatkowe 2

Na stronie kursu dostępne są artykuły, w których opisano metody ViBE i PBAS. Zadanie polega na ich implementacji oraz ewaluacji. Proszę zachować kolejność, gdyż PBAS stanowi rozszerzenie ViBE.

Na rysunku 3.6 przedstawiono przykładowe rozwiązanie problemu.





Rysunek 3.6: Przykładowy wynik proponowanego rozwiązania dla zbioru *office*. Indeks ramki to 1000. Pierwsza od lewej kolumna to obraz wejściowy wraz z obrazem referencyjnym. Następnie aproksymacja średniej oraz mediany z liberalną polityką aktualizacji, aproksymacja średniej oraz mediany z konserwatywną polityką aktualizacji oraz ostatnia kolumna to algorytm ViBe (na górze) i algorytm PBAS (na dole).



# IV

## Laboratorium 4

<b>4</b>	<b>Przepływ optyczny .....</b>	<b>41</b>
4.1	Cel zajęć	
4.2	Przepływ optyczny (ang. <i>optical flow</i> )	
4.3	Implementacja metody blokowej	
4.4	Implementacja wyliczania w wielu skalach	
4.5	Zadanie dodatkowe 1 – Inne sposoby wyznaczania przepływu optycznego	
4.6	Zadanie dodatkowe 2 – Detekcja kierunku ruchu i klasyfikacja obiektów z wykorzystaniem przepływu optycznego	





## 4. Przepływ optyczny

### 4.1 Cel zajęć

- zapoznanie z zagadnieniem przepływu optycznego,
- poznanie zakresu stosowalności przepływu optycznego i jego ograniczeń,
- poznanie wykorzystania wielu skal w wyznaczaniu przepływu optycznego,
- implementacja metody blokowej,
- zapoznanie z metodami dostępnymi w bibliotece OpenCV,
- uruchomienie przykładowych sieci neuronowych,
- wykorzystanie przepływu optycznego do segmentacji obiektów ruchomych,
- klasyfikacja obiektów jako sztywne lub nieszttywne z wykorzystaniem przepływu optycznego.

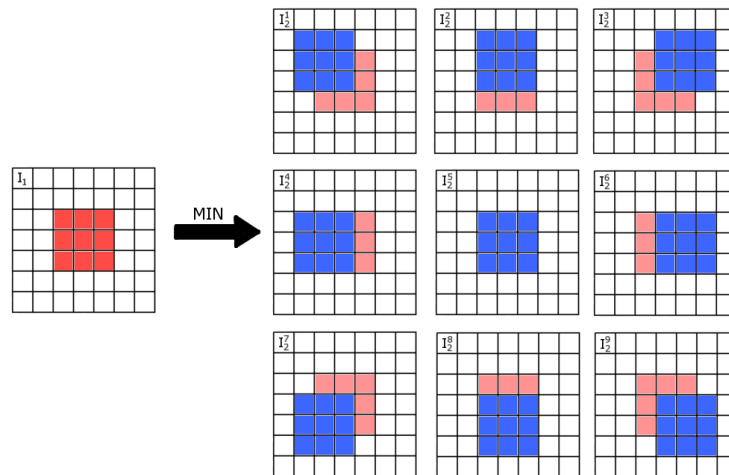
### 4.2 Przepływ optyczny (ang. *optical flow*)

Przepływ optyczny to pole wektorowe opisujące ruch pikseli (ew. tylko wybranych, tj. punkty charakterystyczne) pomiędzy dwoma kolejnymi ramkami z sekwencji wideo. Inaczej mówiąc, dla każdego piksela na ramce  $I_{n-1}$  wyznaczamy wektor przesunięcia, w rezultacie którego otrzymamy obraz  $I_n$ . Warto jednak zaznaczyć, że przepływ można (a niekiedy wręcz jest to pożądane) obliczać dla ramek o odstępach większym niż '1'.

Przepływ optyczny może być gęsty (ang. *dense*) lub rzadki (ang. *sparse*). W pierwszym przypadku przemieszczenie wyznaczone jest dla każdego piksela, a w drugim tylko dla pewnego ich ograniczonego zbioru.

Podstawą działania przepływu optycznego jest „śledzenie” pikseli (lub pewnych lokalnych konfiguracji pikseli tj. otoczeń) pomiędzy ramkami. Oznacza to, że na kolejnych ramkach poszukujemy tych samych pikseli (lub też ich niewielkich otoczeń). Zakłada się przy tym, że jasność piksela jest stała (w skali odcieni szarości, w ogólnym przypadku – stały kolor).

Niestety, jak pokaże ćwiczenie, podejście nie sprawdza się w kilku przypadkach, z których najważniejszy to duże, jednorodne powierzchnie. Dlaczego tak się dzieje? Wyobraźmy sobie dość duże kartonowe pudełko. Ma ono szare, jednorodne ściany bez tekstury oraz oświetlone jest równomiernym światłem. Chcemy wyznaczyć przepływ optyczny dla wszystkich pikseli na obrazie,



Rysunek 4.1: Przykład poszukiwania odpowiadającego bloku

w tym dla piksela ze środka ścianki pudełka. Czy uda nam się znaleźć na kolejnej ramce dokładnie ten punkt? Czy jednoznaczne przypisanie jest możliwe?

Z tej przyczyny czasem korzystniej jest zastosować przepływ rzadki, obliczany w uprzednio określonych lokalizacjach. Tutaj mamy dwie możliwości. Albo arbitralnie wybieramy punkty np. na kwadratowej lub prostokątnej siatce (wtedy celem jest redukcja czasu obliczeń), albo wyszukujemy tzw. punkty charakterystyczne – mogą to być krawędzie, narożniki lub inne „wyraziste” elementy obrazu. Istnieje szereg algorytmów ich detekcji, np. detektor narożników Harris’a, detektory punktów charakterystycznych SIFT lub SURF itp. (będzie to tematem jednego z kolejnych ćwiczeń).

W literaturze opisano bardzo dużo metod – proszę zobaczyć na rankingi dostępne na stronach [Middlebury](#), [KITTI](#), [Sintel](#). Dwie najpopularniejsze („klasyczne”) to Horn-Schunck (HS) i Lucas-Kanade (LK). Druga z nich jest dostępna w bibliotece OpenCV.

### 4.3 Implementacja metody blokowej

Metoda blokowa, jak sama nazwa wskazuje, polega na dopasowywaniu pewnych bloków pikseli między kolejnymi ramkami. Z jednego obrazu wycinamy bloki o określonym rozmiarze (np.  $3 \times 3$ ,  $5 \times 5$  itp.), stosując metodę okna przesuwanego po całym obrazie. Dla drugiego obrazu poszukujemy najbardziej podobnego bloku do tego wyciętego, przy czym poszukiwania ograniczamy do niewielkiego otoczenia współrzędnych wyciętego bloku. Takie postępowanie ma dwa uzasadnienia – zwykle przemieszczenie pikseli między kolejnymi ramkami jest niewielkie, a ponadto znacznie ograniczamy czas wykonywania algorytmu.

**Ćwiczenie 4.1** Zaimplementuj metodę blokową do wyznaczania przepływu optycznego.

1. Utwórz nowy skrypt w języku Python. Wczytaj obrazy *I.jpg* i *J.jpg*. Będą to dwie ramki z sekwencji, oznaczone jako I i J (odpowiednio wcześniejsza i późniejsza ramka). Opcjonalnie możesz zmniejszyć oba obrazy na czas testów, np. czterokrotnie – obliczenia będą się wykonywały szybciej. Wykonaj konwersję obrazów do odcieni szarości – `cvtColor`. Wyświetl zadane obrazy – przydatne może być wyświetlanie przy użyciu `namedWindow` oraz `imshow`. Zwizualizuj różnicę, wykorzystując polecenie: `absdiff`.
2. Wykorzystując poniższe wskazówki, zaimplementuj metodę blokową wyznaczania przepływu optycznego. Przyjmij następujące założenia – porównujemy fragmenty obrazu o rozmiarze  $7 \times 7$ . Wykorzystywane dalej oznaczenia oraz wartości dla okna  $7 \times 7$ :

- $W2 = 3$  – wartość całkowita z połowy rozmiaru okna,
  - $dX = dY = 3$  – rozmiar przeszukiwania (maksymalnego przesuwania okna) w obu kierunkach.
3. Implementację należy zacząć od dwóch pętli `for` po obrazie. Wykorzystaj parametr  $W2$  do warunku uwzględniającego przypadek brzegowy – zakładamy, że na brzegu nie wyliczamy przepływu optycznego.
  4. Wewnątrz pętli wycinamy fragment ramki  $I$ . Dla przypomnienia, niezbędna składnia to:  

$$I0 = \text{np.float32}(I[j-W2:j+W2+1, i-W2:i+W2+1]).$$

**R** W rozważaniach przyjmujemy, że  $j$  – indeks pętli zewnętrznej (po wierszach),  $i$  – indeks pętli wewnętrznej (po kolumnach).

Proszę zwrócić uwagę na składnik „+1” – w Pythonie górny zakres to wartość o 1 większa od największej przyjmowanej wartości – inaczej niż w Matlabie. Konwersja na typ `zmiennoprzecinkowy` potrzebna jest do dalszych obliczeń.

5. Następnie realizujemy kolejne dwie pętle `for` – przeszukiwanie otoczenia piksela  $J(j, i)$ . Mają one zakres od  $-dX$  do  $dX$  i  $-dY$  do  $dY$ . Proszę nie zapomnieć o „+1”. Wewnątrz pętli należy sprawdzić, czy współrzędne aktualnego kontekstu (tj. jego środek) mieszczą się w dopuszczalnym zakresie. Alternatywnie można też zmodyfikować zakres zewnętrznych pętli – tak aby wykluczyć dostęp do pikseli spoza dopuszczalnego zakresu. W tym przypadku dla nieco szerszego brzegu przepływ nie zostanie określony, jednak nie ma to istotnego znaczenia praktycznego.
6. Wycinamy otoczenie  $J0$ , dokonujemy konwersji na `float32` a następnie obliczamy „odległość” między wycinkami  $I0$  a  $J0$ . Można to wykonać instrukcją: `np.sqrt(np.sum((np.square(J0-I0))))`. Spośród wszystkich wycinków z  $J0$  dla danego  $I0$  należy znaleźć najmniejszą „odległość” – lokalizację „najbardziej podobnego” do  $I0$  fragmentu na obrazie  $J$ .
7. Uzyskane współrzędne znalezionych minimów należy zapisać w dwóch macierzach (przykładowo  $u$  i  $v$ ) – należy je wcześniej (przed główną pętlą) utworzyć i zainicjować zerami (funkcja `np.zeros`), wymiary takie jak dla obrazu  $I$ .
8. Wyznaczone pole przepływu optycznego można zwizualizować na dwa sposoby – poprzez wektory (strzałki) – funkcja `plt.quiver` z biblioteki `matplotlib` lub kolory – zrealizujemy drugie z tych podejść. Pomysł opiera się na określeniu kąta i długości wektora wyznaczonego przez dwie składowe przepływu optycznego  $u$  i  $v$  dla każdego piksela. Otrzymamy wówczas reprezentację danych podobną jak w przestrzeni barw HSV. Po wyświetleniu kolor oznacza kierunek, w którym przemieszczają się piksele, natomiast jego nasycenie informuje o względnej szybkości ruchu pikseli – przeanalizuj koło kolorów na rys. 4.2. Dokonaj konwersji wyznaczonego przepływu do układu współrzędnych biegunowych – `cartToPolar`. Utwórz zmienną na obraz w przestrzeni HSV o wymiarach wejściowego obrazu, 3 kanałach i typie `uint8`. Pierwszy kanał to składowa  $H$ , równa  $angle * 90 / \text{np.pi}$  (w Pythonie zakres od 0 do 180). Drugi kanał to składowa  $S$  – dokonaj normalizacji (`normalize`) długości wektora do przedziału 0-255. Trzeci kanał to składowa  $V$ , ustaw ją na 255.

**R** Aby brak ruchu oznaczony był kolorem czarnym, a nie białym, należy zamienić kanały  $S$  i  $V$ .

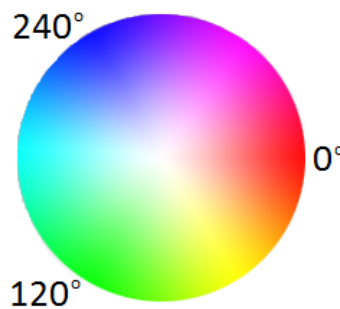
Na koniec wykonaj konwersję obrazu do przestrzeni RGB i wyświetl go.

Proszę przeanalizować uzyskane wyniki oraz poeksperymentować z rozmiarem obrazu wejściowego oraz z parametrami  $W2$  oraz  $dX$  oraz  $dY$  (`np. obraz` dwukrotnie zmniejszony,

parametry równe 5). Czy dla obrazu wejściowego w oryginalnych rozmiarach takie parametry pozwalają poprawnie wyznaczyć przepływ optyczny? Jak duże okno jest potrzebne, aby otrzymać podobne wyniki?

Sprawdź działanie metody dla pary obrazów z innej sekwencji – *cm1.png* i *cm2.png*. *Ground truth* dla tej pary obrazów, używane do ewaluacji algorytmów wyznaczania przepływu optycznego, zamieszczono na rys. 4.3.

- Ⓜ Metoda blokowa z uwagi na swą prostotę jest dość niedokładna – wyliczone przesunięcie w poziomie lub w pionie jest całkowitoliczbowe. W innych metodach (np. HS czy LK) przepływ optyczny jest przeważnie niecałkowity.



Rysunek 4.2: Koło kolorów

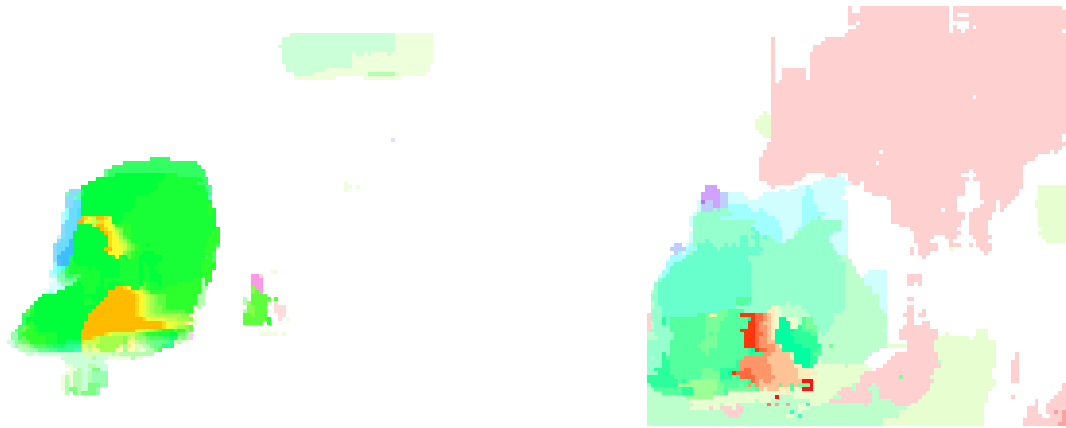
Na rys. 4.4 zamieszczone zostały przykładowe wyniki dla analizowanych sekwencji testowych.



Rysunek 4.3: *Ground truth* dla obrazów *cm1.png* oraz *cm2.png*.

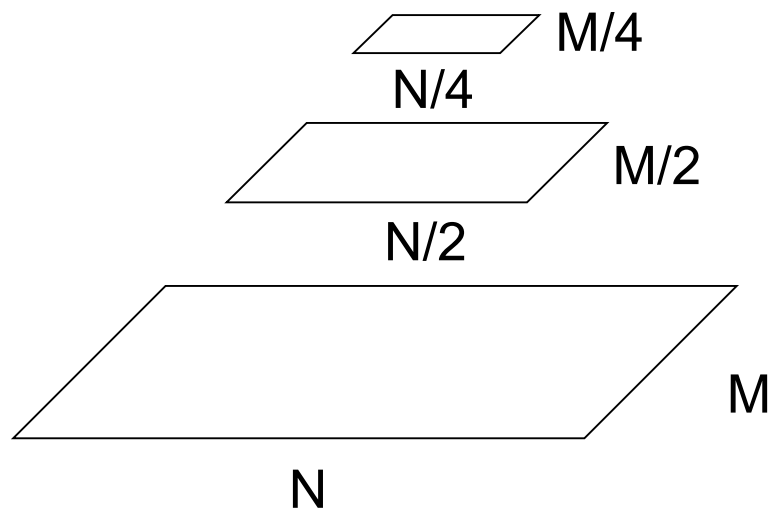
#### 4.4 Implementacja wyliczania w wielu skalach

Wykrywanie dużych przemieszczeń zaimplementowaną metodą jest złożone obliczeniowo. Wynika to z konieczności użycia dużych wartości parametrów  $dX$  i  $dY$ , a zatem dla większości lokalizacji (oprócz brzegowych) konieczne będzie sprawdzenie podobieństwa większej liczby kontekstów (więcej operacji SAD). Takie podejście jest bardzo nieefektywne.



Rysunek 4.4: Przykładowe wyniki przepływu optycznego wyznaczonego metodą blokową: po lewej stronie dla obrazów *I*.jpg oraz *J*.jpg, pomniejszonych 2-krotnie i parametrów  $W2 = dX = dY = 5$ ; po prawej stronie dla obrazów *cm1*.png oraz *cm2*.png, pomniejszonych 2-krotnie i parametrów  $W2 = dX = dY = 5$ .

Lepszym, a także bardzo często stosowanym pomysłem, jest przetwarzanie obrazu w wielu skalach. Idea pokazana jest na rysunku 4.5.



Rysunek 4.5: Przykład konstrukcji piramidy obrazów (wielu skal)

Schemat działania metody:

- wykonujemy przeskalowanie obrazów (*I* oraz *J*) do skal:  $L_0$  – rozdzielczość oryginalna,  $L_1, \dots, L_m$  (najmniejszy obraz) – w literaturze określa się to mianem piramidy obrazów. W tego typu rozwiązaniach najczęściej wykorzystuje się skalowanie ze współczynnikiem 0.5.
- obliczamy przepływ optyczny dla pary obrazów *I* oraz *J* w skali  $L_m$  (np. metodą blokową).
- propagujemy wyniki obliczonego przepływu na skalę  $L_{m-1}$ .
  - w pierwszym kroku poprzednia ramka obrazu zostaje zmodyfikowana zgodnie z przepływem (ang. *warping*). Celem takiego zabiegu jest kompensacja ruchu, czyli przesunięcie pikseli w taki sposób, aby w większej skali  $L_{m-1}$  odpowiadające sobie na dwóch obrazach piksele były bliżej siebie,

- następnie tak zmodyfikowany obraz jest zwiększany do skali  $L_{m-1}$  (czyli zwykle dwukrotnie).
- obliczamy dokładniejsze wartości przepływu na poziomie  $L_{m-1}$  pomiędzy zmodyfikowanym obrazem  $I$  a  $J$  (np. metodą blokową),
- postępowanie realizujemy aż do poziomu  $L_0$ .

**Ćwiczenie 4.2** Zaimplementuj wieloskalową wersję metody blokowej do wyznaczania przepływu optycznego.

1. Na wstępie proszę stworzyć kopię dotychczas stworzonego algorytmu w ramach zadania 1. Następnie fragment algorytmu dotyczący wyliczania OF zamieniamy na **funkcję** dla wybranej skali.

Funkcja powinna mieć następującą postać:

```
def of(I_org, I, J, W2=3, dY=3, dX=3):
```

Poszczególne parametry to  $I\_org$  i  $J$  – obrazy wejściowe,  $I$  – obraz po modyfikacji,  $W2$ ,  $dX$ ,  $dY$  – parametry metody (identyczne jak w 4.3). Tak naprawdę przesyłanie  $I\_org$  do funkcji nie jest potrzebne – sugerowane rozwiązanie ma na celu obliczenie absdiff między obrazami i wyświetlenie wyników oraz obrazów  $I\_org$ ,  $I$  oraz  $J$  dla lepszego zrozumienia działania metody. W rzeczywistości obliczenia prowadzone są dla  $I$  i  $J$ .

2. Przydatne będzie także zamienienie fragmentu algorytmu dotyczącego wizualizacji przepływu optycznego na **funkcję**. Funkcja ta powinna być postaci:

```
def vis_flow(u, v, YX, name):
```

Poszczególne parametry to  $u$  i  $v$  – składowe przepływu optycznego,  $YX$  – wymiary obrazu,  $name$  – wyświetlana nazwa okna, np. 'of scale 2'.

3. Po napisaniu funkcji dobrze jest przetestować jej działanie dla przypadku jednej skali ( $L_0$ ) – wynik powinien być identyczny jak otrzymywany wcześniej.
4. Po utworzeniu dwóch funkcji na podstawie napisanego już algorytmu, przechodzimy do napisania funkcji do generowania piramidy obrazów. Można wykorzystać następującą funkcję:

```
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
```

W tym przypadku zakładamy, że pomniejszenia rozdzielczości zawsze będą dwukrotne. Czyli obraz wejściowy pomniejszamy 2-krotnie, 4-krotnie, 8-krotnie itd. Na potrzeby eksperymentu zakładamy, że ograniczamy się do 3 skal. Piramidę należy wygenerować dla każdego z obrazów wejściowych.

5. Przetwarzanie zaczynamy od skali najmniejszej, zatem do zmiennej  $I$  przypisujemy obraz z piramidy w najmniejszej skali:  $I = IP[-1]$ , gdzie  $IP$  to wygenerowana piramida dla pierwszego obrazu. Proszę zwrócić uwagę na składnię  $[-1]$  – dostęp do ostatniego elementu.
6. Kluczowy komponent algorytmu to pętla po skalach – proszę zastanowić się nad indeksem początkowym i końcowym. Zaczynamy od wyliczenia przepływu i robimy kopię pierwszego obrazu  $I\_new$ . Następny krok to modyfikacja tej kopii zgodnie z przepływem. Tę operację wykonujemy dla wszystkich skal oprócz największej (czyli obrazu o wejściowych wymiarach). Można to zrealizować przy wykorzystaniu dwóch pętli `for` z ew. zabezpieczeniem przed wyjściem poza zakres. Poszczególnym pikselom

z `I_new` przypisujemy odpowiednie piksele z `I`, zgodnie z wyliczonym przepływem. Warto sprawdzić, czy obraz pierwszy po modyfikacji jest zbliżony do obrazu drugiego – jeśli tak jest, to operacja została zrealizowana poprawnie. Na koniec musimy przygotować obraz do kolejnej (większej) skali. W tym celu zwiększamy obraz `I_new` dwukrotnie i przypisujemy go do `I`. Zwiększenie: `I = cv2.resize(I_new, (0,0), fx=2, fy=2, interpolation=cv2.INTER_LINEAR)`.

7. Ostatni element algorytmu to wyliczenie całkowitego przepływu i jego wizualizacja. W tym celu należy przygotować pustą tablicę 2D dla każdej ze składowych `u` i `v` o wymiarach wejściowego obrazu. Następnie w pętli po skalach dodajemy do siebie przepływy z różnych skal.

**R** Krótki przykład dla wyjaśnienia. Mamy piksel, którego przemieszczenie wynosi 9 (*ground truth*), jednak przeszukiwanie mamy ograniczone do 5 pikseli w każdym kierunku. Obraz zmniejszamy dwa razy, teraz przemieszczenie piksela powinno wynosić 4.5. Znajdujemy najbardziej pasujący piksel w mniejszej skali – otrzymujemy przesunięcie równe 5, zatem zgodnie z tym wynikiem modyfikujemy obraz (przesuwamy piksel o 5 w odpowiednim kierunku) i zwiększamy go dwa razy – teraz (w większej skali) analizowany piksel jest przesunięty o 10. Liczymy ponownie przepływ, tym razem w większej skali, i otrzymujemy przesunięcie równe -1 (przepływ rezydualny). Następnie sumujemy wyniki z obu skal i otrzymujemy przesunięcie równe 9.

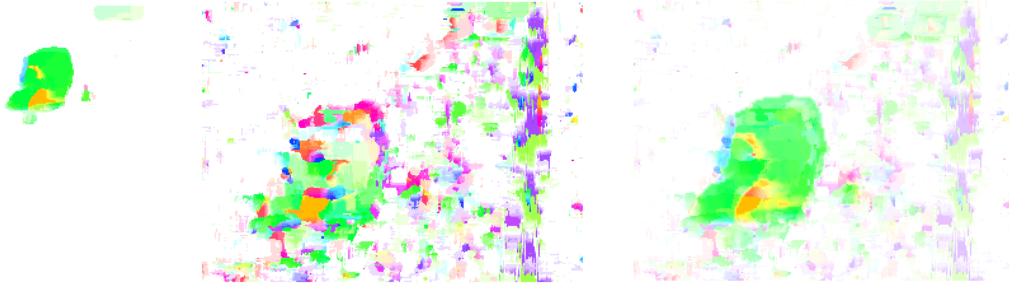
**Wnioski.** Aby otrzymać poprawny wynik końcowy, należy przepływy `u` i `v` z każdej skali zwiększyć na dwa sposoby – wymiary tych „obrazów” muszą być takie, jak dla wejściowego `I`, a wartości przepływów muszą być zwiększone 2, 4, 8 itd. razy, w zależności od skali. Przepływ zsumowany z różnych skal należy zwizualizować przy pomocy funkcji `vis_flow`.

8. Porównaj działanie metody ze skalami i bez. Przeanalizuj wyniki dla większych rozmiarów okien w jednej skali oraz dla mniejszych rozmiarów okien z wieloma skalami, zwróć uwagę na czas wykonania. Czy stosując metodę wieloskalową (np. 3 skale), udało się uzyskać poprawny przepływ dla niewielkiego okna (np.  $dX = dY = 3$ ) dla obrazu o oryginalnym rozmiarze (czyli dla większego przemieszczenia pikseli)? Ponownie sprawdź działanie metody dla pary obrazów `cm1.png` i `cm2.png`.

**R** Metoda wieloskalowa w teorii działa bardzo dobrze – w praktyce dopasowanie pikseli nie zawsze jest dokładne, a skalowanie w dół i w górę zawsze powoduje utratę pewnych informacji, przez co nawet drobne błędy propagowane są na kolejne skale i końcowy wynik może być częściowo nieprawidłowy bądź zaszumiony. Z tego powodu w praktyce używa się 2, 3, maksymalnie 4 skale (ale i tak wyniki różnią się nieco od „oczekiwanych”).

Na rys. 4.6-4.11 zamieszczone zostały przykładowe wyniki dla analizowanych sekwencji testowych.

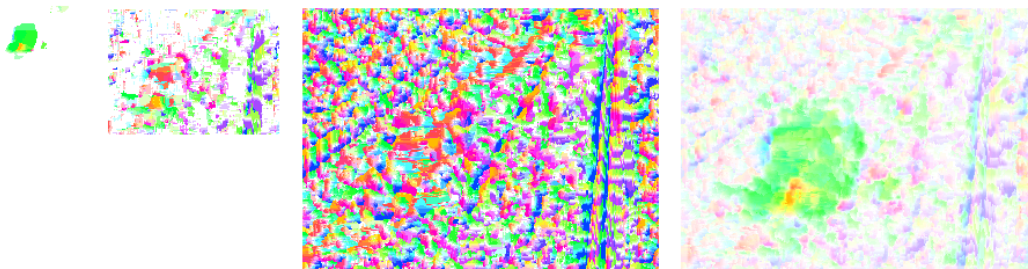




Rysunek 4.6: Przykładowe wyniki przepływu optycznego wyznaczonego metodą blokową w 2 skalach dla obrazów *I.jpg* oraz *J.jpg*, w oryginalnym rozmiarze i parametrów  $W2 = dX = dY = 5$ . Kolejno od lewej: przepływ w mniejszej skali, przepływ rezydualny w skali większej, całkowity przepływ.

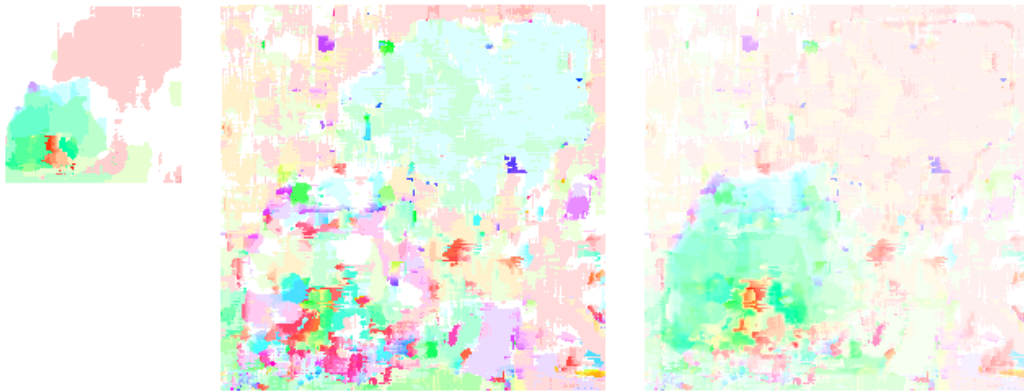


Rysunek 4.7: Przykładowe wyniki *warpingu* dla obrazów *I.jpg* oraz *J.jpg*, w oryginalnym rozmiarze i parametrów  $W2 = dX = dY = 5$ . Kolejno od lewej: ramka poprzednia *I\_org*, ramka zmodyfikowana zgodnie z przepływem w mniejszej skali *I*, ramka następna *J*.



Rysunek 4.8: Przykładowe wyniki przepływu optycznego wyznaczonego metodą blokową w 3 skalach dla obrazów *I.jpg* oraz *J.jpg*, w oryginalnym rozmiarze i parametrów  $W2 = dX = dY = 3$ . Kolejno od lewej: przepływ w najmniejszej skali, przepływ rezydualny w skali środkowej, przepływ rezydualny w skali największej, całkowity przepływ.





Rysunek 4.9: Przykładowe wyniki przepływu optycznego wyznaczonego metodą blokową w 2 skalach dla obrazów *cm1.png* oraz *cm2.png*, w oryginalnym rozmiarze i parametrów  $W2 = dX = dY = 5$ . Kolejno od lewej: przepływ w mniejszej skali, przepływ rezydualny w skali większej, całkowity przepływ.



Rysunek 4.10: Przykładowe wyniki *warpingu* dla obrazów *cm1.png* oraz *cm2.png*, w oryginalnym rozmiarze i parametrów  $W2 = dX = dY = 5$ . Kolejno od lewej: ramka poprzednia  $I_{org}$ , ramka zmodyfikowana zgodnie z przepływem w mniejszej skali  $I$ , ramka następna  $J$ .



Rysunek 4.11: Przykładowe wyniki przepływu optycznego wyznaczonego metodą blokową w 3 skalach dla obrazów *cm1.png* oraz *cm2.png*, w oryginalnym rozmiarze i parametrów  $W2 = dX = dY = 3$ . Kolejno od lewej: przepływ w najmniejszej skali, przepływ rezydualny w skali środkowej, przepływ rezydualny w skali największej, całkowity przepływ.

## 4.5 Zadanie dodatkowe 1 – Inne sposoby wyznaczania przepływu optycznego

Na stronie kursu dostępne są następujące sekwencje wideo:

- *highway* — sekwencja z monitoringu ruchu na autostradzie,
- *pedestrians* — zapis monitoringu kampusu.

Pochodzą one ze zbioru *changedetection.net* (są dostępne na stronie [www](http://www.changedetection.net) o takim adresie).

Wykorzystaj stworzony w ramach wcześniejszych ćwiczeń skrypt do odczytu sekwencji obrazów. Upewnij się, że zawiera on parametr `iStep`, który pozwala ustawić, co która ramka będzie przetwarzana.

**R** Dla uproszczenia analizowane będą sekwencje w odcieniach szarości.

W bibliotece OpenCV dostępnych jest kilka funkcji do obliczania przepływu optycznego:

- Lucas-Kanade (w wersji wieloskalowej, rzadki i gęsty),
- Farneback,
- Dual TV-L1,
- PCA Flow,
- DIS Flow,
- Simple Flow,
- Deep Flow.

**R** Szczegóły odnośnie konkretnych algorytmów są dostępne w [dokumentacji OpenCV](#). Warto też zapoznać się z [tutorialem do przepływu optycznego](#) i go przeanalizować.

### Ćwiczenie 4.3 Uruchom przykładowe rozwiązania.

1. Zadanie 1 – wyznaczyć przepływ optyczny metodami gęstymi (czyli wszystkimi poza rzadkim LK). Kod powinien się składać z kilku elementów:

- wczytywanie obrazów wejściowych i konwersja do odcieni szarości,
- utworzenie zmiennej `flow` o dwóch kanałach (dla  $u$  i  $v$ ),
- utworzenie instancji danej metody wyznaczania przepływu optycznego i wywołanie na niej `calc`,
- wizualizacja wyników jak w rozdziale 4.3.

Zwróć uwagę na wyniki dla różnych metod (m.in. dokładność wyznaczenia krawędzi obiektów), a także na czas wykonywania obliczeń.

**R** Do wizualizacji gęstego LK przydatne może być ograniczenie modułu przepływu do przedziału 0-10, np. poprzez `mag[mag>10]=10`.

2. Zadanie 2 – uruchomić rzadki przepływ optyczny metodą Lucas-Kanade. W [tutoriale](#) pokazano śledzenie punktów charakterystycznych (metody ich detekcji będą elementem jednego z następnych ćwiczeń). My obliczymy przepływ optyczny dla równomiernie rozmieszczonych punktów. Zatem kod należy odpowiednio zaadaptować. Kilka uwag:

- algorytm ma szereg parametrów, które po analizie wykładu powinny być jasne (ew. proszę doczytać w dokumentacji) – rozmiar okna, liczba skal, kryteria zakończenia obliczeń (rozwiązanie układu równań),
- należy wygenerować siatkę równomiernie rozmieszczonych punktów (np. krok 10 pikseli) – tu warto podglądnąć postać `p0` z przykładu,
- druga trudność to inna wizualizacja. Przykładowe rozwiązanie:

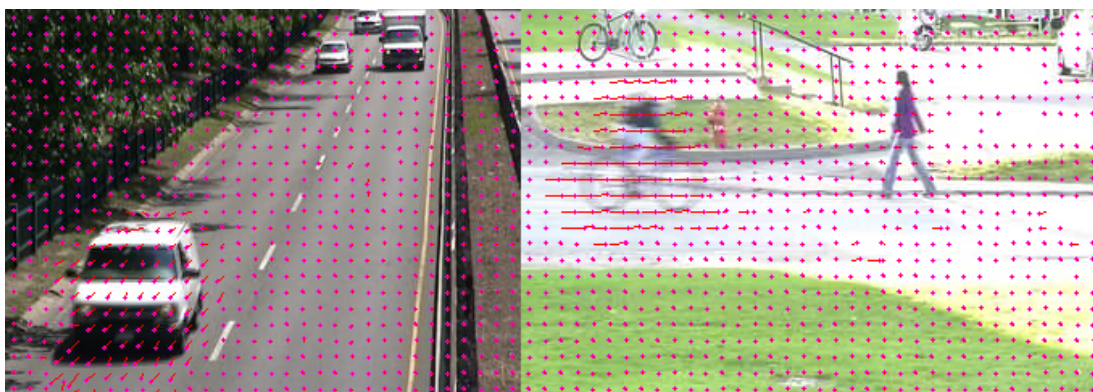
```
img = I
for jj in range(0, y, 1):
    if (st[jj] == 1):
        cv2.line(img, (points[jj,0,0],points[jj,0,1]), (new_points
            [jj,0,0],new_points[jj,0,1]), (0,0,255))
```

- proszę nie zapomnieć o przypisaniu aktualnego obrazu do poprzedniego na końcu pętli.

Proszę zaobserwować jak działa algorytm – czy wyznaczone przepływy są poprawne? Do wyświetlania można dodać wizualizację zwrotu – zaznaczyć początek lub koniec wektora kropką. Można też usunąć wektory o niewielkim module.

Na rys. 4.12 zamieszczone zostały przykładowe wyniki dla analizowanych sekwencji testowych.

3. Zadanie 3 – uruchomić przykładowe sieci neuronowe do wyznaczania przepływu optycznego. Jak łatwo się domyślić, także w tej dziedzinie pojawiły się rozwiązania, które zwracają dużo lepsze wyniki niż metody klasyczne. Na stronie kursu dostępne są implementacje dwóch sieci w PyTorchu – [LiteFlowNet](#) oraz [SPyNet](#), których reimplementacje w PyTorchu zostały pobrane z [GitHuba](#) i dostosowane do prostego uruchomienia. W razie potrzeby zmodyfikuj ścieżki, z których wczytywane są sekwencje. Zmodyfikuj też wartość parametru `iStep` (np. na 5 lub 10) i sprawdź działanie sieci. Szczegóły implementacji do samodzielnej analizy.




Rysunek 4.12: Przykładowe wyniki rzadkiego przepływu optycznego wyznaczonego metodą LK: po lewej stronie dla obrazu o indeksie 158 z sekwencji *highway*; po prawej stronie dla obrazu o indeksie 471 z sekwencji *pedestrians*.

## 4.6 Zadanie dodatkowe 2 – Detekcja kierunku ruchu i klasyfikacja obiektów z wykorzystaniem przepływu optycznego

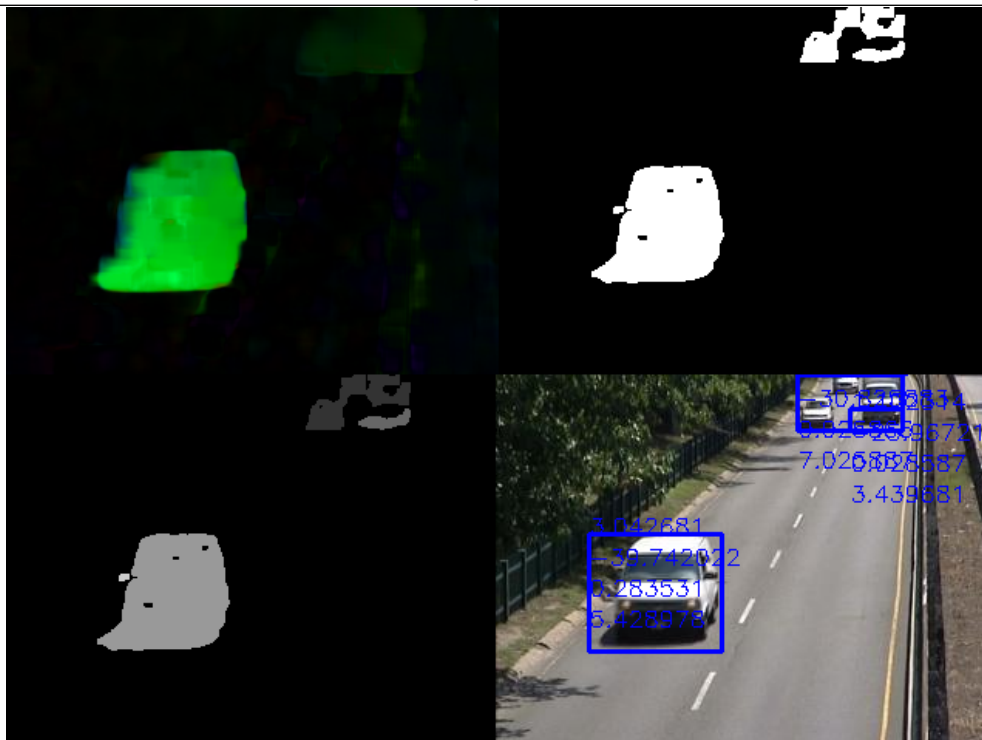
Pomysł jest następujący. Samochód, który jest bryłą sztywną, powinien charakteryzować się spójnym ruchem — każdy jego element powinien mieć podobne przemieszczenie, przynajmniej co do kierunku. Idący człowiek wprost przeciwnie, gdyż specyfika chodu powoduje, że kończyny przemieszczają się w różnych kierunkach (w zależności od fazy kroku).

Spróbujemy zatem przeanalizować wektory ruchu dla każdego z obiektów (w zależności od sekwencji samochodu lub człowieka) i zobaczyć, czy na tej podstawie można coś powiedzieć i ew. dokonać klasyfikacji.

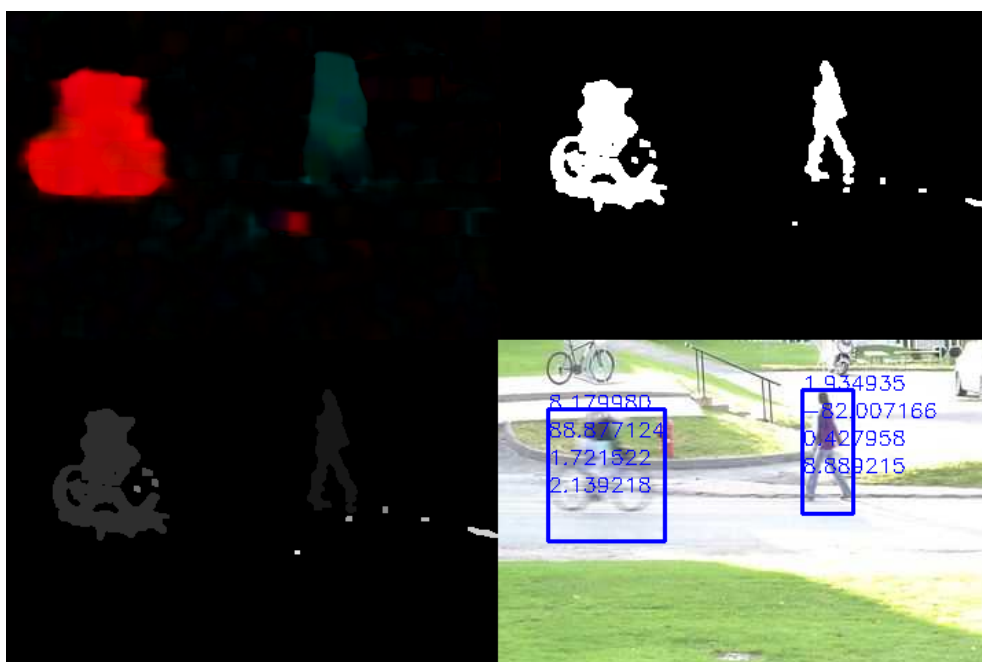
**Ćwiczenie 4.4** Zaimplementuj algorytm, bazując na programie z przepływem optycznym z poprzedniego zadania (np. z algorytmem Farnebacka).

1. Na początku dodamy segmentację obiektów pierwszoplanowych. Wykorzystaj użyty na poprzednich laboratoriach algorytm MOG/GMM.
  2. Bardzo istotna kwestia to filtracja maski. Tu również przydatne będzie doświadczenie zdobyte na poprzednich ćwiczeniach. Dobrym pomysłem wydaje się wyłączenie detekcji cieni – `detectShadows=False` w konstruktorze MOG2.
  3. W kolejnym kroku dodajemy indeksację, której wynik wyświetlamy.
  4. Następnie wykonujemy analizę przepływu optycznego. Sprowadza się ona do obliczenia średniej i odchylenia standardowego dla modułu i kąta wektorów odpowiadających danemu obiektowi. Stopień trudności zadania zależy od biegłości w Pythonie. Kilka odpowiedzi (można, ale nie trzeba skorzystać):
    - obliczenia prowadzimy tylko gdy są obiekty – `retval > 0`,
    - kluczem jest dogodny „kontener” na średnie i odchylenia. Można wykorzystać listy i instrukcję `append`. Wtedy uzyskamy taką funkcjonalność jak w Matlabie. Uwaga. Łatwiej użyć dwóch osobnych list, gdyż prostsze jest późniejsze obliczanie statystyk.
    - zasadnicze obliczenia realizujemy w pętli po obrazku. Jeśli etykieta jest różna od zera to:
      - sprawdzamy, czy amplituda przepływu optycznego w tym punkcie jest większa od zadanego progu (np. 1),
      - jeśli tak, to obliczamy kąt wektora (`atan2` z `math`) i wynik dodajemy do naszej listy pod „odpowiednim adresem”.
    - w kolejnym kroku obliczamy średnią i odchylenie standardowe dla modułu i kąta. Przydatne będą funkcje `mean` i `stdev` z pakietu `statistics`.
-  Przed obliczeniem trzeba sprawdzić, czy istnieją co najmniej dwa elementy na liście dla danego obiektu.
- ostatni etap to wizualizacja. Najprościej kod z ćwiczenia nt. segmentacji obiektów ruchomych przystosować do wyświetlania wyznaczonych parametrów – dwóch średnich i dwóch odchyłeń standardowych. Przy okazji można dodać filtrację małych obiektów.
5. Analiza wyników:
    - Jak zachowuje się średnia i odchylenie standardowe dla obu sekwencji?
    - Czy średni kierunek ruchu odpowiada „mniej więcej” rzeczywistości?
    - Czy da się zauważyć jakąś różnicę pomiędzy obiema sekwencjami – odchylenie standardowe?
    - Czy występują problemy z segmentacją? Jakie?
- Ewentualnie można spróbować wyświetlić dla każdego obiektu średni wektor ruchu. Na rys. 4.13 oraz 4.14 zamieszczone zostały przykładowe wyniki dla analizowanych sekwencji testowych.





Rysunek 4.13: Przykładowe wyniki dla obrazu o indeksie 146 z sekwencji *highway*: po lewej u góry – przepływ optyczny metodą Farnebacka, po prawej u góry – segmentacja obiektów pierwszoplanowych, po lewej na dole – indeksacja, po prawej na dole – wykryte obiekty wraz z wyliczonymi parametrami.



Rysunek 4.14: Przykładowe wyniki dla obrazu o indeksie 471 z sekwencji *pedestrians*: po lewej u góry – przepływ optyczny metodą Farnebacka, po prawej u góry – segmentacja obiektów pierwszoplanowych, po lewej na dole – indeksacja, po prawej na dole – wykryte obiekty wraz z wyliczonymi parametrami.





# Laboratorium 5

<b>5</b>	<b>Kalibracja kamery i stereowizja . . . . .</b>	<b>57</b>
5.1	Kalibracja pojedynczej kamery	
5.2	Kalibracja układu kamer	
5.3	Obliczenia korespondencji stereo	
5.4	Wykorzystanie sieci neuronowej do realizacji zadania analizy głębi obrazu	
5.5	Zadanie dodatkowe – transformata Censusa	





## 5. Kalibracja kamery i stereowizja

Kamera to urządzenie, które przekształca świat trójwymiarowy w obrazy dwuwymiarowe. Zależność tą można opisać następującym równaniem:

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

gdzie:  $x$  oznacza obraz 2D,  $P$  oznacza macierz współczynników projekcji kamery,  $X$  oznacza punkty 3D.

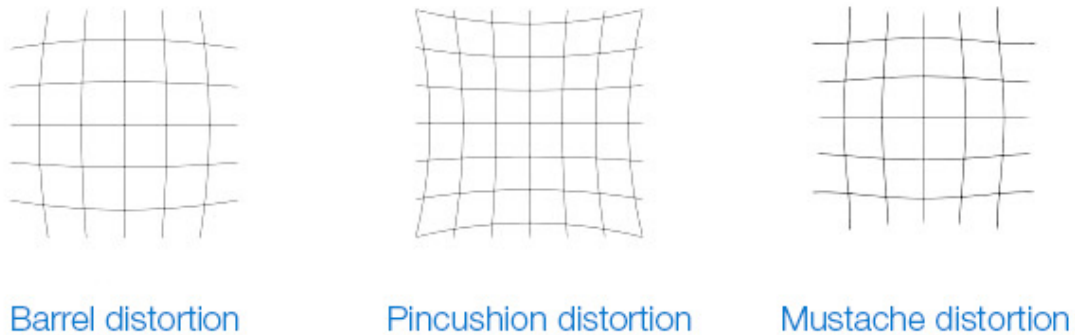
Algorytmy liniowe lub nieliniowe są stosowane do estymacji parametrów wewnętrznych i zewnętrznych z wykorzystaniem znanych punktów w czasie rzeczywistym i ich projekcji na płaszczyźnie obrazu.

Kalibracja kamery ma na celu eliminację zniekształceń (dystorsji) wprowadzanych przez układ optyczny. Zalicza się do nich zniekształcenia: radialne (beczkowe, poduszkowe, faliste – związane z obiektywem) oraz tangensoidalne (związane z nierównoległym montażem soczewki i matrycy – rys. 5.1 oraz rys. 5.2. Szczegółowe omówienie zagadnienia – [Distortion\\_\(optics\)](#) i [Camera Calibration MathWorks](#).

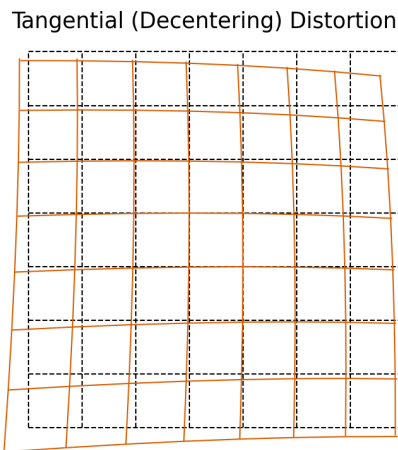
Korekcja zniekształceń wymaga wyliczenia pewnych współczynników. Jest to możliwe w procedurze kalibracji, gdzie rejestruje się obraz wzorca o znanych parametrach (wzajemne odległości między punktami i ich rozmiary “rzeczywiste”) w różnych położeniach względem kamery. Najczęściej stosuje się wzorzec w postaci szachownicy, na którym łatwo wykryć narożniki, nawet z dokładnością sub-pikselową (tak jak na analizowanym obrazie). Praktyka wskazuje, że obrazów powinno być co najmniej 10.

Parametry wyznaczane w procedurze kalibracji kamery dzieli się na dwie grupy:

- parametry zewnętrzne (ang. *Extrinsic*, *External*) – umożliwiają odwzorowanie współrzędnych piksela i współrzędnych kamery w ramce obrazu, np. środek optyczny, ogniskowa i współczynniki zniekształceń radialnych obiektywu.
- parametry wewnętrzne (ang. *Intrinsic*, *Internal*) – opisują orientację i położenie kamery. Odnoszą się do obrotu i przesunięcia kamery w stosunku do pewnego zewnętrznego (względem kamery) układu współrzędnych.



Rysunek 5.1: Rodzaje zniekształceń: (a) beczkowe, (b) poduszkowe (c) faliste.



Rysunek 5.2: Zniekształcenie tangensoidalne. Źródło: <https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i>.

Na rysunku 5.3 przedstawiono równanie modelu kamery, która zawiera współrzędne homogeniczne (jednorodne) oraz macierz projekcji (ang. *camera projection matrix*). Macierz kalibracji to iloczyn macierzy zawierających wewnętrzne oraz zewnętrzne parametry kamery. Do estymacji parametrów wewnętrznych i zewnętrznych wykorzystuje się algorytmy liniowe lub nieliniowe. Wykorzystuje się znajomość parametrów rzeczywistych (odległość pomiędzy punktami, w centymetrach) oraz ich rzutów na płaszczyznę (w pikselach).

Wyróżnia się dwa modele kamery:

- *pinhole camera model* – podstawowy model kamery bez soczewki, szczegóły w dokumentacji OpenCV: **Pinhole model**,
- *fisheye camera model* – wykorzystywany przede wszystkim w sytuacjach, gdzie zniekształcenia są ekstremalne, dobrze modeluje kamery szerokokątne, szczegóły w dokumentacji OpenCV: **Fisheye model**.

Przeprowadzenie prawidłowej korekcji zniekształceń – kalibracji kamery – jest niezbędne, jeśli chcemy dokonać pomiaru rozmiarów obiektów np. w centymetrach, zmierzyć odległość lub określić przemieszczenie kamery.

$$\begin{array}{c}
 \begin{array}{l} \text{Intrinsic Parameters (fundamental} \\ \text{characteristics of the camera)} \end{array} \\
 \begin{array}{c} \left[ \begin{array}{ccc} \frac{1}{\rho_u} & 0 & u_0 \\ 0 & \frac{1}{\rho_v} & v_0 \\ 0 & 0 & 1 \end{array} \right] \left[ \begin{array}{cccc} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array} \right] \end{array} \\
 \text{Camera Matrix} \\
 \begin{array}{c} \text{Extrinsic Parameters (depend on} \\ \text{where camera is)} \end{array} \\
 \begin{array}{c} \left[ \begin{array}{cc} R & t^{-1} \\ 0_{1 \times 3} & 1 \end{array} \right] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \\
 \text{Rotation Matrix} \\
 \text{(orientation of camera)} \\
 \text{Position of camera}
 \end{array}
 \end{array}
 \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} =$$

Rysunek 5.3: Równanie modelu kamery bazujące na ogólnym wzorze 5.1.  $\rho_u, \rho_v$  to rozmiar obrazu  $x, y$  w pikselach,  $u_0, v_0$  to środek optyczny,  $f$  to ogniskowa. Źródło: <https://towardsdatascience.com/understanding-transformations-in-computer-vision-b001f49a9e61>

## 5.1 Kalibracja pojedynczej kamery

Procedura kalibracji kamery jest podzielona na etapy. Niemal identyczne kroki wykonuje się w przypadku kalibracji pojedynczej kamery lub kamer stereowizyjnych. Wykorzystany zostanie zbiór danych, zawierający pary obrazów pochodzących z kamery stereowizyjnej. Do kalibracji pojedynczej kamery należy wybrać zbiór z jednego obiektywu (odpowiednio lewe lub prawe obrazy). Zaproponowany zbiór został zarejestrowany przez kamerę, która wymaga zamodelowania przez model *fish-eye* (tzw. rybie oko).

Ogólnie proces kalibracji kamery można podzielić na:

1. Wybranie odpowiedniego wzoru kalibracyjnego. Do najczęściej stosowanych zalicza się: szachownicę, chArUco (połączenie szachownicy oraz znaczników ArUco), macierz kółek, macierz asymetrycznych kółek.
2. Wykonanie zdjęć planszy kalibracyjnej pod różnymi kątami oraz z różnej odległości. Liczba prawidłowo wykonanych obrazów powinna być większa niż 10.
3. Przygotowanie parametrów planszy kalibracyjnej, np. zmierzenie rozmiaru pól szachownicy w centymetrach.
4. Przygotowanie oraz uruchomienie algorytmu do kalibracji.

Poniżej opisano etapy algorytmu do kalibracji pojedynczej kamery. Wykorzystano do tego wcześniej przygotowany zbiór obrazków (dostępny na platformie UPEL) oraz funkcje z biblioteki OpenCV.

```

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

```

Mając wyliczone współrzędne punktów można przystąpić do kalibracji:

```
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** Opis parametrów otrzymanych w procesie kalibracji:

- `ret` – wartość błędów średniokwadratowego projekcji wstecznej (opisuje jakość dopasowania),
- `K` – macierz parametrów wewnętrznych kamery w postaci:

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

gdzie:  $(f_x, f_y)$  – to długości ogniskowych, a  $(c_x, c_y)$  to współrzędne środka obrazu.

- `D` – współczynniki zniekształceń,
- `rvecs` – wektory rotacji,
- `tvecs` – wektory translacji (wraz z wektorami rotacji umożliwiają przekształcenie położenia wzorca kalibracyjnego z współrzędnych modelu do współrzędnych rzeczywistych)

Po otrzymaniu wszystkich potrzebnych parametrów w procesie kalibracji, należy teraz wybrać dowolny obraz z podanego zbioru oraz usunąć zniekształcenia. Do tego posłuży funkcja `cv2.remap()`:

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

**Ćwiczenie 5.1** Wykorzystując powyższy opis do kalibracji pojedynczej kamery, wykonaj zadanie.

- Wyznacz parametry kamery i wyświetl je.
- Sprawdź korekcję dla jednego z obrazów ze zbioru (test na linię prostą).
- Przeprowadź korekcję zniekształceń dla całego zbioru obrazów. Wyświetl oraz porównaj obraz przed i po przeprowadzeniu kalibracji.

## 5.2 Kalibracja układu kamer

W największym uproszczeniu, kalibracja układu kamer (dla uproszczenia dwóch) ma umożliwić łatwe obliczenie map głębi. Zagadnienie to można rozumieć na różne sposoby: ustawienie identycznych parametrów obu kamer, eliminację zniekształceń, sprowadzenie do wspólnego układu współrzędnych oraz rektyfikację. Szczególnie interesujące jest ostatnie zagadnienie. W ogólnym przypadku korespondujące ze sobą piksele na obu obrazach leżą na tzw. liniach epipolarnych. Rektyfikacja polega na takim przekształceniu obrazów, aby linie te były równoległe to jednej z krawędzi obrazu (zwykle poziomej). Znakomicie ułatwia to wyznaczenie korespondencji (mapy głębi) – trzeba prowadzić poszukiwania tylko w jednej płaszczyźnie (a nie w dwóch wymiarach lub po linii ukośnej (konieczna interpolacja)).

Początkowe operacje przeprowadza się podobnie jak dla pojedynczej kamery. Najlepiej przekopiować stworzony kod do nowego pliku, dodać wczytywanie drugiego obrazu oraz wprowadzić odpowiednie modyfikacje. W efekcie powinniśmy uzyskać parametry kamery dla lewego i prawego obrazu (wyjście z `cv2.fisheye.calibrate`).

Następnie wykonujemy trzy funkcje, które są odpowiedzialne za kalibrację stereo, rektyfikację oraz obliczenie współczynników mapowania dla funkcji `remap`:

```
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,
    disparityToDepthMap) = 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)
```



Opis parametrów wykorzystanych w powyższych metodach:

- `imgpointsLeft`, `imgpointsRight` – lista wykrytych narożników szachownicy dla lewego oraz prawego obrazu, wartości pochodzą z funkcji `cv2.cornerSubPix`,
- `K_left`, `K_right` – macierz parametrów wewnętrznych kamery dla zbioru par obrazów, wartości pochodzą z funkcji `cv2.fisheye.calibrate`,
- `D_left`, `D_right` – współczynniki zniekształceń kamery, wartości pochodzą z funkcji `cv2.fisheye.calibrate`.

Na końcu pozostaje nam wczytać dwa obrazy ze zbioru (dowolne) i wykonać mapowanie:

```
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)
```

Aby sprawdzić działanie rektyfikacji warto zestawić oba obrazy obok siebie i wyrysować poziome linie, co pozwoli sprawdzić, czy faktycznie te same piksele leżą na tych samych liniach. Można to przykładowo zrobić tak:

```
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** Warto wiedzieć, że w OpenCV dostępna jest również funkcja rektyfikacji obrazów nieskalibrowanych `stereoRectifyUncalibrated`. Zachęcamy do jej uruchomienia.

**Ćwiczenie 5.2** Przeprowadź proces kalibracji oraz rektyfikacji dla kamery stereo. Wyświetl obraz z usuniętymi zniekształceniami oraz z poziomymi liniami, które potwierdzą poprawność działania algorytmu. ■

## 5.3 Obliczenia korespondencji stereo

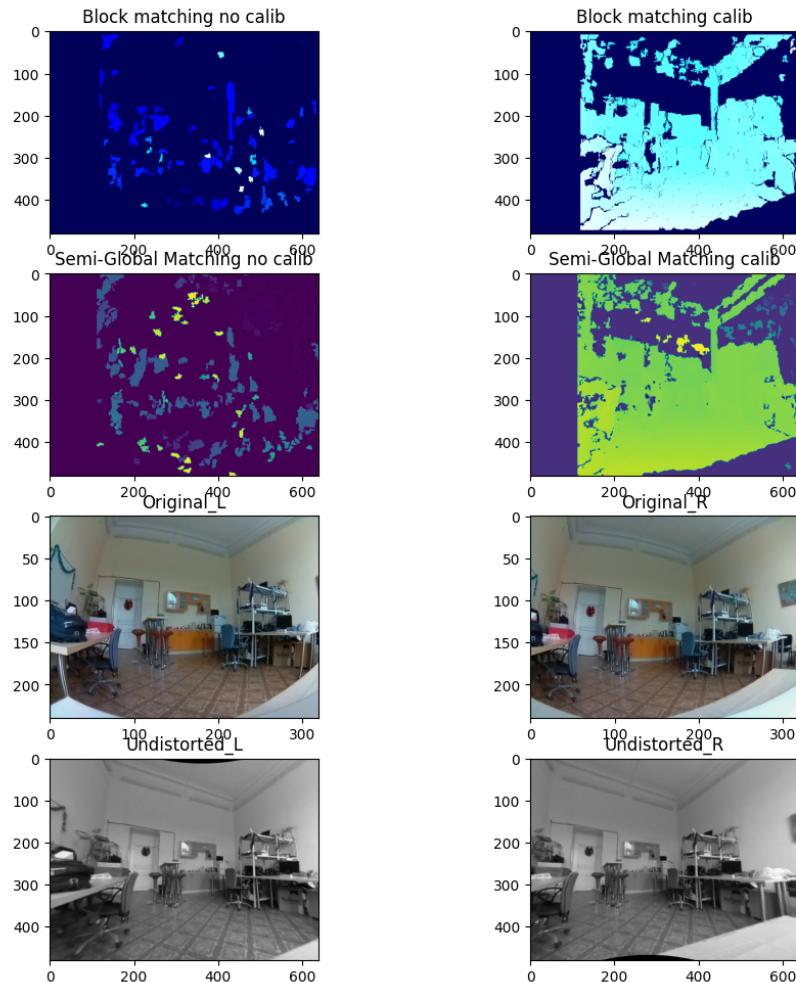
Samo obliczenie korespondencji stereo (mapy głębi, mapy dysparycji) jest stosunkowo proste (przynajmniej w teorii i dla odpowiednich zdjęć). Celem jest znalezienie o ile piksel  $I_L(x, y)$  jest przesunięty ( $d$  – dysparycja) na drugim obrazie  $I_R(x + d, y)$ . Dla przypomnienia – szukamy tylko w liniach poziomych, bo zakładamy, że obrazy poddane zostały rektyfikacji. Można też zauważyć, że zadanie w pewnym sensie jest podobne do przepływu optycznego. Tylko tam mieliśmy dwie kolejne ramki z sekwencji zarejestrowane jedną kamerą, a tu dwa obrazy z dwóch kamer w tym samym czasie. Notabene, warto wiedzieć, że na podstawie sekwencji z pojedynczej kamery też można odtworzyć głębię – o ile kamera ta jest ruchoma – zagadnienie *Structure from Motion*.

W bibliotece OpenCV dostępne są dwie metody obliczenia map głębi: dopasowanie bloków (*Block Matching*) i SGM (*Semi-Global Matching*). Działanie pierwszej jest dość oczywiste, w przypadku drugiej dokonuje się również optymalizacji globalnej mapy dysparycji (w uproszczonej formie), co pozwala poprawić ciągłość mapy.

**Ćwiczenie 5.3** Proszę samodzielnie, na podstawie dokumentacji, wyznaczyć mapę dysparycji dla jednego z obrazów z folderu *example* przed i po procesie kalibracji. Parametry poszczególnych funkcji proszę dobrać empirycznie, stosując się do ograniczeń. Wyświetl obraz oryginalny przed i po kalibracji oraz odpowiednie mapy dysparycji (SGM i BM). ■

Przykładowe rozwiązanie zaprezentowano na rys. 5.4.

- R** Mapę dysparycji przed wyświetleniem należy znormalizować do przedziału 0 – 255. Można również zastosować funkcję do konwersji dysparycji do heatmapy: `cv2.applyColorMap(map, cv2.COLORMAP_HOT)`.



Rysunek 5.4: Przykładowe rozwiązanie.

## 5.4 Wykorzystanie sieci neuronowej do realizacji zadania analizy głębi obrazu

W ramach ćwiczenia proponujemy przeanalizować głęboką konwolucyjną sieć neuronową, która pozwala uzyskać mapy głębi na podstawie **pojedynczych** obrazków (metoda na *single-view depth*). Dodatkowo sieć uczona jest w sposób nienadzorowany (tj. bez nauczyciela – w tym przypadku bez referencyjnych map głębi). Ponadto sieć realizuje estymację pozycji kamery. Szczegóły dotyczące zastosowanej metody można przeczytać w artykule [ZHOU, Tinghui, et al., Unsupervised Learning of Depth and Ego-Motion from Video](#). Wykorzystamy udostępniony przez autorów model sieci. Kod jest dostępny na repozytorium GitHub – [SfmLearner-Pytorch](#).

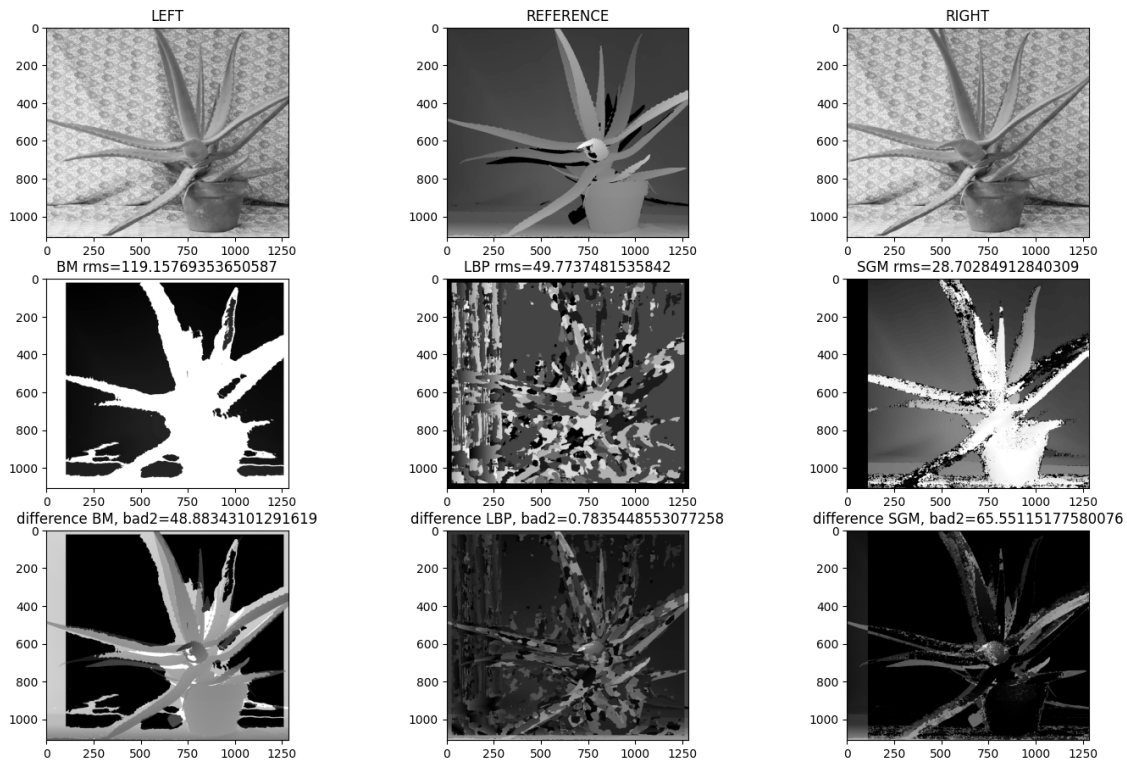


### 5.5 Zadanie dodatkowe – transformata Censusa

Prostą, ale dość dobrą metodą jest tzw. transformata Censusa (znana też pod nazwą *Local Binary Patterns*). Bierzymy blok  $N \times N$  np.  $5 \times 5$ . Binarujemy go z progiem w postaci wartości środkowej z kontekstu. Otrzymujemy ciąg binarny. Na drugim obrazie robimy to samo, ale w  $d$  położeniach (od 0 do  $d_{max} - 1$ ). Porównanie dwóch kontekstów to operacja  $xor(C1, C2)$ . Zliczamy liczbę różnic i szukamy minimum – to będzie nasza wartość  $d$ .

**Ćwiczenie 5.5** Zaimplementuj opisaną metodę i porównaj działanie z BM, SGM oraz DCNN (przynajmniej wizualnie). Wykorzystaj do tego obrazki z folderu *aloes*.

Przykładowe rozwiązanie zaprezentowano na rys. 5.5.



Rysunek 5.5: Przykładowe rozwiązanie.





# Laboratorium 6

<b>6</b>	<b>Punkty Charakterystyczne .....</b>	<b>69</b>
6.1	Cel zajęć	
6.2	Detekcja narożników metodą Harrisa – teoria	
6.3	Implementacja metody Harrisa	
6.4	Prosta deskrypcja punktów charakterystycznych	
6.5	ORB (FAST+BRIEF)	
6.6	Wykorzystanie punktów charakterystycznych do łączenia obrazów	
6.7	Zadanie dodatkowe – SIFT	



## 6. Punkty Charakterystyczne

### 6.1 Cel zajęć

- zapoznanie z zagadnieniem wykrywania punktów charakterystycznych,
- implementacja prostego algorytmu detekcji – metoda Harrisa,
- zapoznanie z zagadnieniem opisywania punktów charakterystycznych,
- otoczenie punktu jako deskryptor,
- porównanie działania metody Harrisa, SIFT i ORB (FAST+BRIEF).

**R** W dzisiejszym ćwiczeniu do wyświetlania obrazów proszę używać funkcji z `matplotlib.pyplot`, a nie z `OpenCV`.

### 6.2 Detekcja narożników metodą Harrisa – teoria

Detekcja narożników metodą Harrisa polega na wyszukiwaniu pikseli dla których moduł gradientu pionowego i poziomego ma znaczną wartość.

Metoda opisana jest za pomocą następujących zależności:

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

gdzie:  $\det$  – wyznacznik macierzy,  $\text{trace}$  – ślad macierzy,  $k$  – stała,  $(x,y)$  – lokalizacja na obrazie. Macierz autokorelacji  $M$  to:

$$M(x,y) = \begin{bmatrix} \langle I_x^2(x,y) \rangle & \langle I_{xy}(x,y) \rangle \\ \langle I_{xy}(x,y) \rangle & \langle I_y^2(x,y) \rangle \end{bmatrix} \quad (6.2)$$

gdzie poszczególne symbole oznaczają:

$$\langle I_x^2(x,y) \rangle = I_x^2(x,y) \otimes h(x,y) \quad (6.3)$$

$$\langle I_y^2(x,y) \rangle = I_y^2(x,y) \otimes h(x,y) \quad (6.4)$$

$$\langle I_{xy}(x,y) \rangle = I_x I_y(x,y) \otimes h(x,y) \quad (6.5)$$

gdzie:  $I_x(x, y)$  oraz  $I_y(x, y)$  są pochodnymi cząstkowymi w kierunku  $x$  i  $y$ , wartości wyznaczone za pomocą np. gradientu Sobela, symbol  $\otimes$  oznacza splot (konwolucję), a  $h(x, y)$  to funkcja Gaussa:

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

gdzie:  $\sigma$  – odchylenie standardowe. Filtracja Gaussa jest używana celem redukcji wpływu szumu, który szczególnie mocno wpływa na obliczanie wartości gradientu.

Wyznacznik i ślad macierzy  $2 \times 2$  (dla przypomnienia):

$$\det(M(x, y)) = \langle I_x^2(x, y) \rangle \langle I_y^2(x, y) \rangle - \langle I_{xy}(x, y) \rangle^2 \quad (6.7)$$

$$\text{trace}(M(x, y)) = \langle I_x^2(x, y) \rangle + \langle I_y^2(x, y) \rangle \quad (6.8)$$

Ostatecznie dany piksel jest klasyfikowany jako narożnik jeśli wyliczona wartość  $H(x, y)$  jest większa niż określony próg.

Jako końcową filtrację warto wykorzystać wyszukiwanie lokalnych maksimów. Za kandydata na narożnik uznajemy tylko te piksele, które są lokalnymi maksimami w swoim otoczeniu (np.  $7 \times 7$ ).

### 6.3 Implementacja metody Harrisa

**Ćwiczenie 6.1** Na podstawie powyższej teorii proszę zaimplementować metodę Harrisa.

1. Ze strony kursu pobierz archiwum z danymi do ćwiczenia i rozpakuj je we własnym katalogu roboczym.
2. **Wczytaj obrazy** *fontanna1.jpg* oraz *fontanna2.jpg*.
3. **Zaimplementuj funkcję** wyliczającą wartość  $H$  z macierzy autokorelacji. Funkcja jako parametry powinna otrzymywać obraz w odcieniach szarości oraz rozmiar masek filtrów Sobela i Gaussa przez nią wykorzystywanych (może to być taki sam rozmiar dla obu filtrów). Przetnij obraz pionowym i poziomym filtrem Sobela (funkcja `cv2.Sobel`) – będzie to realizacja pochodnej obrazu po  $x$  i  $y$ . Jako typ wyniku Sobela proszę podać `cv2.CV_32F`. Wylicz odpowiednie iloczyny pochodnych kierunkowych i rozmyj je za pomocą filtra Gaussa (funkcja `cv2.GaussianBlur` – przykładowe wywołanie: `cv2.GaussianBlur(image, (size, size), 0)`, gdzie *size* to rozmiar maski filtra). Rozmycie filtrem Gaussa odpowiada wyliczeniu ważonej sumy elementów z otoczenia każdego punktu macierzy (ważonej krzywą Gaussa). Do obrazu wynikowego wpisz wartości  $H$  (z wzoru wykorzystującego wyliczone wyznaczniki i ślady). Przyjmij wartość współczynnika  $K = 0.05$ . Zwróć obraz wynikowy (warto go znormalizować np. do zakresu 0-1 celem łatwiejszego doboru progu w następnym punkcie).
4. **Wykorzystaj** poniższą funkcję znajdującą maksima lokalne w otrzymanej jako parametr tablicy:

```
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)
```

Maksima są tu wyszukiwane z użyciem filtra maksymalnego. Zastosowana metoda jest

uproszczona (może zwrócić kilka leżących w obrębie maski takich samych maksimów), ale na nasze potrzeby wystarczająca. Funkcja zwraca znalezione maksima w postaci list współrzędnych – jedna lista dla każdego wymiaru. Dla tablicy 2D będą to listy kolejno: współrzędnych  $y$  i współrzędnych  $x$ . Funkcja działa także dla tablic 3D – wtedy zwróci trzy listy, przy czym pierwsza lista to współrzędne  $z$  (czyli np. wysokość w oktawie piramidy).

5. Dla obu wczytanych obrazów zastosuj powyższe funkcje (druga ma wyszukiwać maksima lokalne w wyniku zwracanym przez pierwszą). Jako rozmiar masek w obu funkcjach możesz przyjąć 7. Wyniki z drugiej funkcji wyświetl jako znaki (np. \*) naniesione na obrazy początkowe. Warto w tym celu stworzyć osobną funkcję rysującą, która utworzy `figure()`, wyświetli obraz za pomocą `plt.imshow()`, a następnie naniesi na niego znaki przy użyciu funkcji `plt.plot()`.

**Przykładowo** `plt.plot(5, 10, '*', color='r')` wyrysuje czerwoną gwiazdkę we współrzędnych  $(x,y)$  równych  $(5,10)$ . Za pomocą tej funkcji można jednocześnie wyrysować kilka gwiazdek: `plt.plot([1,2,3], [4,5,6], '*')` wyrysuje trzy niebieskie gwiazdki we współrzędnych  $(x,y)$  równych  $(1,4)$ ,  $(2,5)$  i  $(3,6)$ . Listy `[1,2,3]` i `[4,5,6]` można uzyskać z drugiej funkcji – ale uwaga na kolejność współrzędnych!

6. **Wyświetl i sprawdź** czy wykrywane punkty na obu obrazach znajdują się (mniej więcej) w tych samych położeniach (czyli czy detektor jest powtarzalny).
7. **Powtórz** operacje z powyższych punktów dla obrazów *budynek1.jpg* i *budynek2.jpg*.



## 6.4 Prosta deskrypcja punktów charakterystycznych

**Ćwiczenie 6.2** Niniejsze ćwiczenie jest kontynuacją poprzedniego zadania. Po detekcji punktów charakterystycznych wymagane jest jeszcze ich opisanie (deskrypcję).

1. Do funkcji z poprzedniego zadania **dodaj funkcję** tworzącą opisy punktów charakterystycznych. Opisem będzie wycinek obrazu z otoczenia punktu o rozmiarze podanym przez parametr. Jako parametry funkcja powinna otrzymać obraz, listę współrzędnych punktów charakterystycznych (wynik funkcji z poprzedniego zadania) oraz wielkość otoczenia. z listy punktów należy usunąć wszystkie punkty, których otoczenia nie mieszczą się w obrazie. Można tu wykorzystać funkcję `filter` z anonimową funkcją `lambda` – przykładowo ( $X,Y$  to rozmiar obrazu, `size` to rozmiar otoczenia/wycinka):

```
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])))
```

Następnie należy utworzyć listę opisów punktów po odfiltrowaniu. Przy czym dobrze będzie sprowadzić go do wektora. Jeżeli np. `p` jest wycinkiem, to za pomocą `p.flatten()` uzyskuje się z niego wektor. Jako wynik funkcji należy zwrócić listę otoczeń uzupełnionych o współrzędne ich punktów centralnych.

2. Na tym etapie można zzipować przefiltrowaną listę współrzędnych z listą otoczeń (funkcja `zip`) – przykładowo:

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

3. **Dodaj funkcję** porównującą opisy punktów charakterystycznych z dwóch obrazów i znajdującą opisy najbardziej podobne. Funkcja jako parametry powinna otrzymać dwie

listy opisów punktów charakterystycznych (z funkcji z punktu 1) oraz liczbę  $n$  najbardziej podobnych do siebie opisów. Funkcja może wykorzystać metodę 'każdy z każdym' (brute force) – każdy opis punktu z listy pierwszej jest porównywany ze wszystkimi opisami z listy drugiej i wybierany jest opis najbardziej podobny. Można różnie określić miarę podobieństwa. Może to być ich odległość wektorów, suma wartości bezwzględnych ich różnic lub wynik ich iloczynu skalarnego. Współrzędne pary punktów, odpowiadających parze najbardziej podobnych do siebie opisów/wektorów, mają być umieszczone w liście wynikowej łącznie z miarą podobieństwa. Funkcja powinna zwrócić listę  $n$  najbardziej podobnych opisów. w tym celu można np. listę posortować (metoda sort) i wziąć jej pierwszych  $n$  elementów (można też zrobić to manualnie przez  $n$ -krotne wybieranie najbardziej podobnego dopasowania). Metoda sort działa 'w miejscu', czyli zastępuje listę listą posortowaną. Aby wykorzystać metodę sort do posortowania listy krotek należy przekazać tej metodzie funkcję, która określi po którym elemencie krotki chcemy sortować. Przykładowo sortowanie po drugim elemencie krotki to:

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

Innymi słowy parametr *key* oczekuje funkcji, która będzie zwracała kolejne elementy do sortowania. Zauważmy też, że  $x[1]$  – to odwołanie do **drugiego** elementu krotki, bo numerujemy od 0.

Standardowo metoda sort sortuje rosnąco. Jeżeli zależy nam na sortowaniu w porządku malejącym, to należy parametr *reverse* ustawić na *True*.

4. **Wczytaj obrazy** *fontanna1.jpg* i *fontanna2.jpg*. Znajdź dla nich punkty charakterystyczne metodą Harris'a za pomocą funkcji z poprzedniego zadania. Dla znalezionych punktów utwórz listy opisów za pomocą funkcji z punktu 1. Rozmiar otoczenia ustaw 15 (możesz poeksperymentować z innymi ustawieniami). Wyszukaj najlepsze dopasowania za pomocą funkcji z punktu 3. Parametr  $n$  można ustawić na 20.
5. **Wyświetl rezultaty**. Wśród danych do ćwiczenia znajduje się plik *pm.py*. Można zaimportować go do swojego pliku (*import pm*) i wykorzystać znajdującą się w nim funkcję *pm.plot\_matches*. Jednakże może być konieczne dostosowanie tej funkcji do Państwa listy zwróconej przez funkcję z punktu 3 (dostarczona implementacja zakłada, że każda para jest zapisana w postaci  $([y1, x1], [y2, x2])$ , obrazy powinny być w odcieniach szarości).
6. **Powtórz operacje** z poprzednich punktów dla obrazów *budynek1.jpg* i *budynek2.jpg*. Zauważ, że na tych obrazach budynki są położone pod nieco innym kątem. Czy wpłynęło to na jakość dopasowania?
7. **Powtórz operacje** z poprzednich punktów dla obrazów *fontanna1.jpg* i *fontanna\_pow.jpg*. Obrazy znacznie różnią się skalą. Czy Harris poradził sobie z taką różnicą?
8. **Powtórz operacje** z poprzednich punktów dla obrazów *eiffel1.jpg* i *eiffel2.jpg*. Obrazy różnią się jasnością i nieco skalą. Czy Harris poradził sobie z taką różnicą?
9. **Zmodyfikuj funkcję** wyznaczającą opisy punktów (z punktu 1), tak aby uwzględnić afiniczne zmiany jasności:

$$w_{aff} = \frac{w - \bar{w}}{|w - \bar{w}|} \quad (6.9)$$

Przy czym średnią  $\bar{w}$  można wyznaczyć jako  $\text{pm.mean}(w)$ , a  $|w - \bar{w}|$  jako odchylenie standardowe za pomocą np.  $\text{pm.std}(w)$ .

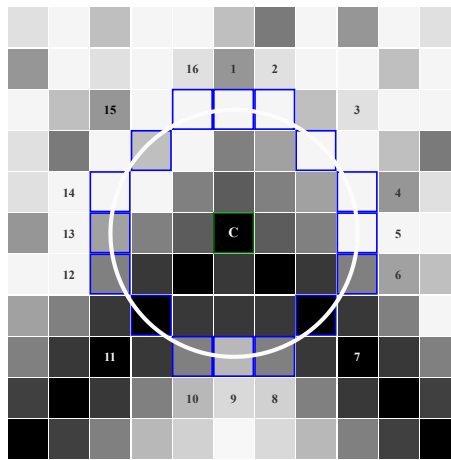
Spróbuj teraz powtórzyć operacje dla wymienionych powyżej obrazów. Czy nastąpiła jakaś poprawa?



## 6.5 ORB (FAST+BRIEF)

Obecnie istnieją trzy najpopularniejsze algorytmy oraz ich modyfikacje do detekcji i deskrypcji punktów charakterystycznych, tj. SIFT, SURF, ORB. Metoda SIFT (ang. *Scale-Invariant Feature Transform*) charakteryzuje się wysoką skutecznością i dokładnością, ale wymaga bardzo dużych nakładów obliczeniowych. Metoda SURF (ang. *Speeded Up Robust Features*) to podejście, które wymaga mniejszej złożoności obliczeniowej, ale jakość uzyskiwanych rezultatów w wybranych przypadkach jest znacząco gorsza niż SIFT. Algorytm ORB (ang. *Oriented FAST and Rotated BRIEF*) stanowi wydajną alternatywę dla SIFT i SURF.

W celu wykrycia punktów charakterystycznych wykorzystywany jest detektor FAST (ang. *Features from Accelerated Segment Test*). Został on zaprojektowany w celu znajdowania narożników, podobnie jak algorytmy Moravca i Harrisa. Realizacja tego zadania odbywa się poprzez porównywanie jasności  $I_c$  danego punktu obrazu z 16 pikselami położonymi na otaczającym go okręgu. Zostało to przedstawione na rysunku 6.1.



Rysunek 6.1: Położenie pikseli, których jasność porównywana jest z punktem centralnym  $c$  w detektorze FAST.

Punkt centralny jest w tym przypadku uznawany za wierzchołek, jeżeli  $n$  kolejnych pikseli na okręgu ma jasność równocześnie niższą niż  $I_c - t$ , bądź równocześnie wyższą od  $I_c + t$ , gdzie  $t$  określa pewien próg — parametr algorytmu. Najlepsze jakościowo rezultaty można uzyskać poprzez przyjęcie  $n = 9$ . Pewien kłopot może natomiast stanowić ryzyko niepożądanego wykrycia krawędzi, nieodłącznie związane z detektorami wierzchołków. Wobec tego należy uszeregować wszystkie otrzymane punkty względem miary Harrisa (6.1), a następnie wybrać  $N$  najlepszych rezultatów. Wszystkie wskazane w ten sposób wierzchołki nie są niestety niezależne od kąta, obrotu obrazu oraz jego skali. W celu rozwiązania pierwszego problemu wykorzystano ideę *centroidu jasności* (ang. *intensity centroid*). Bazuje ona na wyznaczeniu geometrycznych momentów wycinka obrazu wokół danego punktu charakterystycznego, zdefiniowanych poprzez równanie:

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

gdzie:  $x, y$  – lokalne współrzędne względem wykrytego punktu charakterystycznego,  $I(x, y)$  — jasność piksela w punkcie  $(x, y)$ .

Sumowanie przebiega przy tym po wszystkich pikselach znajdujących się w obrębie koła o promieniu  $r$  i środka w danym punkcie charakterystycznym. Na tej podstawie można określić współrzędne centroidu  $C$ :

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

Orientację danego punktu charakterystycznego wyznacza się za pomocą:

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

w celu uodpornienia algorytmu ORB od skali należy zastosować piramidę obrazów.

Po uzyskaniu zbioru punktów charakterystycznych następny etap stanowi ich lokalna deskrypcja. Do tego celu wykorzystywany jest algorytm BRIEF (ang. *Binary Robust Independent Elementary Features*). Polega on na konstruowaniu  $n$ -elementowych łańcuchów bitowych, zgodnie z zależnością:

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

przy czym  $i$ -ty binarny test  $\tau$  jest definiowany poprzez równanie:

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

gdzie:

$p$  – wycinek obrazu wokół danego punktu charakterystycznego,

$u_i, v_i$  – pewne dwa piksele wewnątrz  $\mathbf{p}$  ( $u_i \neq v_i$ ),

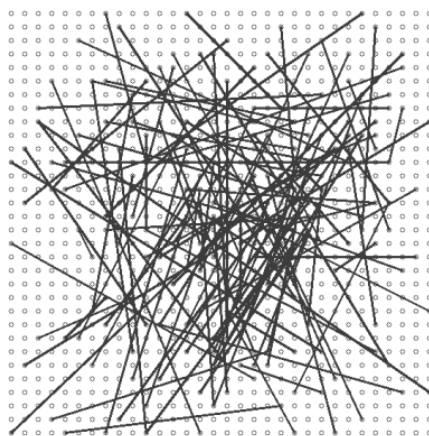
$I(w)$  – jasność obrazu w punkcie  $w = (x, y)$ .

Wycinek  $\mathbf{p}$  powinien zostać przy tym rozmyty w celu niwelacji wpływu zakłóceń. Rozmycie wykonuje się poprzez obrazy całkowite w których dany punkt charakterystyczny reprezentowano poprzez sumaryczną jasność otaczających go pikseli (przyjęto do tego kwadratowe sąsiedztwo  $5 \times 5$  przy rozmiarze wycinka  $31 \times 31$ ). Fundamentalne znaczenie dla działania całego algorytmu ma również odpowiedni wybór par pikseli do poszczególnych testów  $\tau$ . Lokalizacja testowych par jest zdefiniowana losowo – rys. 6.2 lub deterministycznie na podstawie specjalnie opracowanego algorytmu uczącego zaproponowanego przez autorów algorytmu ORB<sup>1</sup>. Wyjściem deskryptora jest 256-elementowy binarny wektor. Do wyznaczenia podobieństwa pomiędzy dwoma binarnymi wektorami stosuje się metrykę Hamminga.

**Ćwiczenie 6.3** Ćwiczenia ma na celu poznanie oraz implementację elementów algorytmu ORB w uproszczonej wersji. Pomocna będzie opisana teoria powyżej. Wyniki należy porównać z rezultatami z poprzedniego zadania.

1. w zadaniu zastosuj parę obrazków używanych w poprzednim zadaniu, np. *fontanna1.jpg* oraz *fontanna2.jpg*.
2. **Zaimplementuj detektor FAST** w celu wykrycia punktów charakterystycznych oraz wyznacz miarę Harrisa dla każdego punktu. Detektor FAST wyznacza punkt charakterystyczny na podstawie wycinka obrazu o rozmiarze  $7 \times 7$  px.
3. Następnie za pomocą metody *non-maximum suppression* wykonujemy pierwszą filtrację. Polega ona na usunięciu punktów charakterystycznych, które znajdują się blisko siebie. Sprawdzamy czy w otoczeniu  $3 \times 3$  px znajduje się wiele wykrytych punktów

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



Rysunek 6.2: Wygenerowane losowo pary punktów dla deskryptora BRIEF.

charakterystycznych, jeśli tak to zostaw ten z największą miarą.

4. Kolejnym krokiem to usunięcie punktów charakterystycznych, które nie mają pełnego otoczenia  $31 \times 31$  wykorzystywanego przez deskryptor.
5. Posortuj otrzymane punkty według miary Harris'a. Wybierz  $N$  najlepszych punktów.
6. Wyznacz współrzędne centroidu  $C$  oraz orientację punktu charakterystycznego na podstawie idei *centroidu jasności* dla każdego punktu.
7. **Zaimplementuj deskryptor BRIEF.** Wycinek o rozmiarze  $31 \times 31$  należy najpierw rozmyć gaussem  $5 \times 5$ . Pary punktów można wyznaczyć losowo z odpowiedniego przedziału lub wczytać gotowe pary punktów z pliku `orb_descriptor_positions.txt`. Jednak są to punkty zdefiniowane dla orientacji  $0^\circ$ . w zależności od kąta  $\alpha$  uzyskanego na podstawie *centroidu jasności* należy obrócić pary punktów, zgodnie z równaniem:

$$x' = \cos \alpha * x - \sin \alpha * y \quad (6.14)$$

$$y' = \sin \alpha * y + \cos \alpha * x \quad (6.15)$$

Wykonujemy binarny test dla wszystkich par punktów, otrzymując 256-bitowy wektor.

8. Aby porównać punkty charakterystyczne z dwóch obrazów, należy wykorzystać metrykę Hamminga. Może być tutaj zastosowana metoda brute force – wyznaczamy podobieństwo dla każdej pary punktów z dwóch obrazów.
9. Następnie sortujemy według najlepszego podobieństwa i wybieramy  $N$  najlepszych punktów.
10. **Wyświetl wyniki** oraz **porównaj** z rezultatami z poprzedniego zadania. Zauważ, że algorytm ORB w porównaniu z poprzednim zadaniem powinien radzić sobie z obrotem.

## 6.6 Wykorzystanie punktów charakterystycznych do łączenia obrazów

**Ćwiczenie 6.4** W zadaniu zostaną wykorzystane gotowe funkcje do detekcji i deskrypcji punktów charakterystycznych z biblioteki OpenCV.

1. Wczytaj obrazy: `left_panorama.jpg` i `right_panorama.jpg`. Przekonwertuj obrazy do skali szarości.
2. Znajdź punkty charakterystyczne na obrazach. Do tego można użyć dowolnego deskryp-

- torą: `cv2.SIFT_create()`, `cv2.xfeatures2d.SURF_create()` lub `cv2.ORB_create()`. Jednak należy pamiętać, że tylko ORB zwraca binarny opis punktu charakterystycznego.
- Wyświetl znalezione punkty za pomocą funkcji: `cv2.drawKeypoints`.
  - Następnie dopasowujemy punkty charakterystyczne z obu obrazów. Do tego służy klasa: `cv2.BFMatcher(NORMA)`. Dla SIFT i SURF norma będzie `cv2.NORM_L2` a dla ORB `cv2.NORM_HAMMING`.
  - Funkcji dopasowujących jest dwie: Brute Force (BF) lub KNN. Jeśli wybierzemy BF to wywołujemy metodę `.match()`, jeśli KNN to metodę `.knnMatch()`.
  - Jeśli BF to otrzymane połączenia sortujemy według dystansu (im mniejszy dystans pomiędzy wektorami deskrypcji tym lepsze podobieństwo) i wybieramy N najlepszych punktów. Jeśli KNN to używamy poniższego kodu:

```
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])
```

albo to samo w wersji z list comprehension:

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

- Dla sprawdzenia można wyświetlić otrzymane połączenia – `cv2.drawMatches`.
- Następnym krokiem jest określenie rotacji i translacji obrazów (punktów) względem siebie, czyli wyznaczamy macierz homografii – funkcja `cv2.findHomography`. Przed podaniem do tej funkcji punktów charakterystycznych należy odpowiednio przekonwertować je. Konwertujemy zbiór punktów do tablicy typu `numpy`:

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

a następnie tworzymy zbiory odpowiednich par już ze sobą dopasowanych:

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

- Ostatnim krokiem jest wywołanie funkcji:

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

gdzie: `(width, height)` to suma wymiarów obu przetwarzanych obrazów. oraz połączenie z drugim obrazem:

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

- Wyświetl wyniki.
- Dodatkowo można wykryć rzeczywisty wymiar połączonych obrazów i usunąć nadmiar czarnego tła.



## 6.7 Zadanie dodatkowe – SIFT

**Ćwiczenie 6.5** Poniżej wykorzystany będzie jeden z najpopularniejszych algorytmów do detekcji oraz deskrypcji punktów charakterystycznych – metoda SIFT (Scale-invariant feature transform).

1. Wykorzystując implementację metody SIFT z OpenCV przeprowadź operacje analogiczne do tych z sekcji 6.4 (znalezienie podobnych do siebie otoczeń punktów charakterystycznych). w tym celu wywołaj funkcję `cv2.xfeatures2d.SIFT_create()` która tworzy obiekt realizujący różne operacje na obrazach z wykorzystaniem metody SIFT. Następnie użyj dwukrotnie metody `detectAndCompute` tego obiektu dla dwóch obrazów: *fontanna1.jpg* i *fontanna2.jpg*. Parametrami tej metody powinny być obraz w odcieniach szarości i None. Metoda zwraca dwie listy – listę punktów charakterystycznych i listę ich opisów (deskrypcji).
2. **Do wyznaczenia podobieństwa** opisów punktów charakterystycznych użyj metody porównującej z OpenCV – `BFMatcher` i jej metody dla  $k$  najbliższych sąsiadów ( $k=2$ , gdyż interesują nas dwa najbardziej podobne otoczenia według miary podobieństwa – będzie to wykorzystane przy rysowaniu wyników). Przykładowo, dla list opisów `desc1` i `desc2` listę pasujących do siebie punktów `matches` uzyskuje się następująco:

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

Tak uzyskaną listę dopasowań można wyświetlić jako obraz zwracany przez funkcję `cv2.drawMatchesKnn` wywołaną z następującymi parametrami: pierwszy obraz, punkty charakterystyczne pierwszego obrazu, drugi obraz, punkty charakterystyczne drugiego obrazu, lista dopasowanych punktów (z `knnMatch`), obraz wyjściowy (ustawiamy na None, obraz wyjściowy jest także zwracany przez tę funkcję), `flags=2` (bez tego argumentu wyświetlone zostaną wszystkie punkty charakterystyczne).

3. Aby pozostawić tylko **najlepsze dopasowania** można z listy `matches` przed wywołaniem `drawMatchesKnn` usunąć te, których drugi sąsiad (z `knn`) ma dość podobne otoczenie. Realizuje to przykładowa pętla:

```
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])
```

albo to samo w wersji z list comprehension:

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

4. **Ponów operacje** z powyższych punktów dla pozostałych obrazów przetwarzanych w sekcji 6.4. Jak SIFT radzi sobie w porównaniu z opisem bazującym na wycinkach (w szczególności dla obrazów *fontanna1.jpg* i *fontanna\_pow.jpg*)?

Jeżeli linii jest za dużo i obraz jest przez to nieczytelny – wystarczy zwiększyć wymagania na to o ile najlepsze dopasowanie jest lepsze od drugiego (czyli **zwiększyć** wartość 0.5).