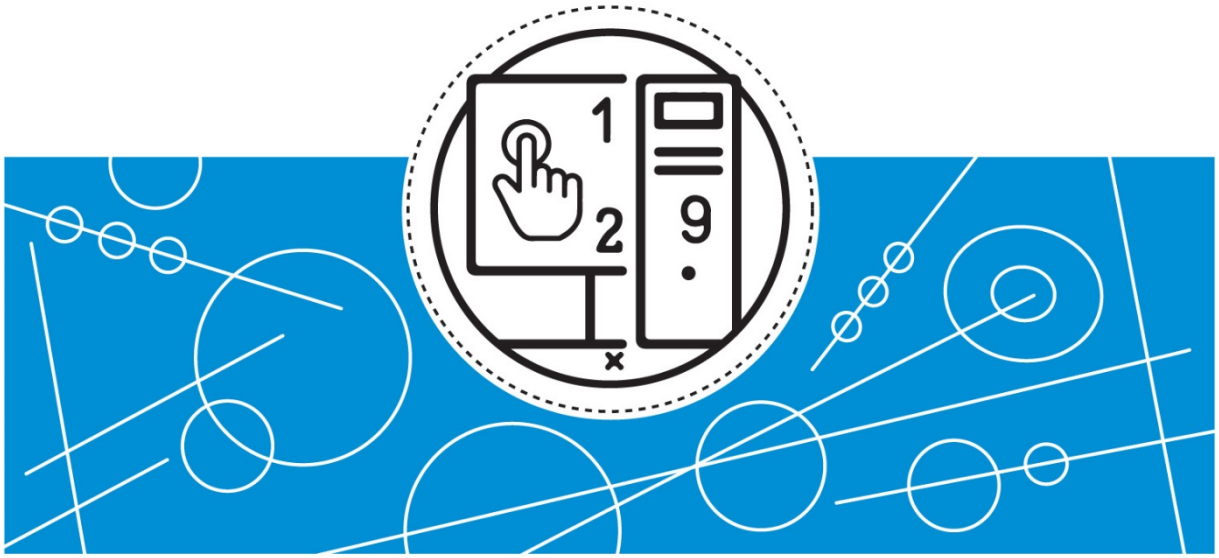
ИТМО

COMPUTER VISION



Санкт-Петербург
2026

MINISTRY OF SCIENCE AND HIGHER EDUCATION
OF THE RUSSIAN FEDERATION

ITMO UNIVERSITY

COMPUTER VISION

STUDY GUIDE

RECOMMENDED FOR USE
AT ITMO UNIVERSITY

within the Master's Program 15.04.06 Robotics and AI

St. Petersburg
2026

С.В. Шаветов, А.Д. Жданов, О.А. Евстафьев, [и др.] Компьютерное зрение .— Санкт-Петербург: Университет ИТМО, 2026. - 157 с.

Рецензент:

В.С. Громов, кандидат технических наук, факультет систем управления и робототехники, Университет ИТМО, Санкт-Петербург, Россия.

Учебное пособие посвящено основам компьютерного зрения. Представлены четыре практических задания возрастающей сложности: от сегментации изображений до обнаружения лиц. Учебное пособие предназначено для магистрантов, обучающихся по направлению 15.04.06 «Робототехника и искусственный интеллект».

ИТМО

Университет ИТМО (Санкт-Петербург) — национальный исследовательский университет и научно-образовательная корпорация. Альма-матер победителей международных соревнований по программированию. Приоритетные направления: IT и искусственный интеллект, фотоника, робототехника, квантовые коммуникации, трансляционная медицина, науки о жизни, Art&Science и научная коммуникация. Лидер федеральной программы “Приоритет 2030”, в рамках которой реализуется инициатива “Открытый код университета”. С 2022 года ИТМО работает по новой модели развития — научно-образовательная корпорация — академическая свобода, поддержка инициатив студентов и сотрудников, распределённая система управления, приверженность открытому исходному коду и бизнес-подходы к организации работы. Обучение в университете строится вокруг индивидуальной образовательной траектории каждого студента. Пять лет подряд ИТМО входит в топ-100 мировых университетов в области Automation & Control (кибернетика) согласно Shanghai Ranking. По данным SuperJob, ИТМО занимает первое место в Санкт-Петербурге и второе в России по уровню зарплат выпускников в IT-сфере. Университет стабильно входит в число лучших российских вузов в международных рейтингах, находится в первой пятёрке вузов России по качеству принятых студентов и удерживает рекорд в Санкт-Петербурге по числу зачисленных победителей олимпиад. С 2019 года ИТМО самостоятельно присуждает учёные степени кандидата и доктора наук.

Contents

Introduction

Computer vision, the science of enabling machines to interpret and understand visual data has evolved from theoretical foundations into a transformative technology underpinning autonomous systems, biometrics, augmented reality, and industrial automation. This practical course explores the core algorithmic pillars of classical computer vision, emphasizing mathematical principles, robust implementation in Python using OpenCV and scikit-image, and critical evaluation of performance trade-offs.

The journey begins with image segmentation, the process of partitioning an image into semantically coherent regions. Students implement thresholding strategies ranging from global (e.g., Otsu's method [1]) to adaptive techniques that handle non-uniform illumination. Color-based segmentation leverages perceptually uniform spaces like CIE Lab [2], which better aligns with human vision than RGB, while k-means clustering [3] provides an unsupervised approach to dominant color extraction. Texture segmentation further extends this paradigm using statistical descriptors such as entropy and local variance, concepts rooted in image statistics and information theory [4].

Next, the course introduces the Hough Transform, a voting-based framework for detecting parametric shapes. Originally proposed by Paul Hough for particle trajectory detection [5], it was later adapted by Duda and Hart for line detection using the (ρ, θ) parameterization [6]. The method is extended to circles and implemented via both OpenCV and scikit-image, allowing students to compare accumulator space access and probabilistic variants that improve computational efficiency.

The third module focuses on local feature detection and description, a cornerstone of geometric computer vision. Two landmark algorithms are studied in depth:

- **SIFT (Scale-Invariant Feature Transform)** [7], which achieves invariance to scale, rotation, and affine distortion through Difference-of-Gaussians extrema detection, orientation assignment, and 128-dimensional gradient histograms;
- **ORB (Oriented FAST and Rotated BRIEF)** [8], a computationally efficient binary alternative combining the FAST corner detector with rotation-aware BRIEF descriptors [9].

Feature matching is refined using cross-check validation and Lowe's ratio test [7] followed by geometric verification via RANSAC (Random Sample Consensus) [10] to estimate homography and reject outliers — a pipeline essential for object recognition and image alignment.

Finally, the course culminates with the Viola–Jones object detection framework [11], a real-time face detector that revolutionized embedded vision. Its four innovations remain influential:

1. **Haar-like features** as simple yet discriminative intensity contrasts;
2. The **integral image** for constant-time rectangular sum computation;
3. **AdaBoost** [12] for selecting a sparse set of critical features;
4. A **cascade classifier** architecture that enables early rejection of background regions, achieving high speed without sacrificing accuracy.

Although deep learning now dominates many vision tasks, these classical methods retain critical relevance: they require no training data, operate efficiently on resource-constrained devices, offer interpretability, and often serve as preprocessing or fallback mechanisms in hybrid pipelines. By mastering them, students gain not only practical coding proficiency but also a deep appreciation for the geometric, statistical, and algorithmic ingenuity that laid the groundwork for modern computer vision.

1 Practical Assignment No 1. Image Segmentation

Objectives

- **Master basic and automatic image thresholding methods**
  Implement single, double, and automatically calculated thresholding (Otsu, mean value, gradient weighting).
  Understand the difference between global and adaptive thresholding.
- **Study color-based segmentation principles**
  Implement segmentation based on Weber principle.
  Apply skin color segmentation models in RGB space under various lighting conditions.
- **Learn segmentation in uniform color spaces**
  Work with the CIE Lab color space to separate chromatic components from lightness.
  Implement nearest neighbor segmentation with interactive seed point selection.
  Apply k-means clustering for automatic pixel classification.
- **Acquire skills in texture segmentation**
  Use entropy and statistical moments as texture descriptors.
  Build a complete pipeline: texture feature extraction → thresholding → morphological filtering → contour detection.

Preface

To successfully complete this practical assignment, students will need the following:

- **Python 3.x** with core scientific libraries:
  - `opencv` (`opencv-python` pip library) — for image processing operations;
  - `numpy` (<=1.26.4) — for efficient array manipulations;
  - `scikit-image` — for entropy filtering and morphological tools;

  - `matplotlib` — for visualization.

  Example installation command: `pip install opencv-python, numpy==1.26.4, scikit-image, matplotlib`

  To execute Python programs, you, ust install a development environment, for example, Visual Studio Code with the Python development extension and additional Python packages to run programs – `ipykernel`.

- **Development environment:** It is recommended to use **Jupyter Notebook** or **Jupyter Lab** (locally or via Google Colab). Alternatively, any Python IDE with support for interactive execution (e.g., VS Code with Python extension) is acceptable.

- **Utility module:** The file `pa_utils.py` is provided alongside the assignment. This module contains auxiliary functions essential for the work:

  - `ShowImages()` — displays multiple images side-by-side with titles;
  - `bwareaopen()` — removes small connected components (MATLAB-style);
  - `imfillholes()` — fills holes in binary images;
  - `ipython_exit_cell()` — allows clean cell termination within Jupyter without stopping the kernel.

  The module is imported at the very beginning of the notebook and is used throughout all tasks.

1.1 Task 1. Binarization

The simplest way to segment an image into two classes (background and object pixels) is by **binarization**. In this case we will convert our image to black-and-white color model by applying threshold to image pixel intensity values. There are several common approaches to how this can be done, so let us check them out.

The general image binarization algorithm would be as follows:

1. load an image;

2. convert it to grayscale;

3. threshold the grayscale image.

1.1.1 Simple Thresholding

We will try the thresholding algorithms on the classic Lena image. Let us read it in BGR and then convert to a GRAYSCALE mode.

```
# Read an image from a file in BGR
fn = "images/lena_color.png"
I1 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I1, np.ndarray) or I1.data
    == None:
     print("Error reading file \"{}\"".format(
    fn))
     exit()

# Convert loaded BGR image to grayscale
I1gray = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
# Display it
ShowImages([("Source image", I1),
            ("Grayscale", I1gray)], 2)
```

Figure 1.1. Source Image

Figure 1.2. Grayscale image

1.1.2 Single Thresholding

The first and the most simple binarization is the single thresholding method, which is described by the following formula:

$$I_{st}(x,y) = \begin{cases} 0 & \text{if } I(x,y) \leq t, \\ 1 & \text{if } I(x,y) > t, \end{cases}$$

where

- I is the source image;
- I_{st} is the single threshold binarized image;
- t is the binarization threshold.

In OpenCV, the thresholding operation is executed with `cv2.threshold(image, t, maxval, method)` function. It takes an `image` to process, the threshold parameter `t`, the value to set for thresholded image pixels `maxval`, and the thresholding `method` as its parameters. It returns a tuple with the threshold used during thresholding and the thresholded image.

The following single thresholding methods are supported in OpenCV:

- `cv2.THRESH_BINARY` — set everything greater than t value to the `maxval` and everything else to 0:

$$I_{\text{binary}}(x,y) = \begin{cases} 0 & \text{if } I(x,y) \leq t, \\ \text{maxval} & \text{if } I(x,y) > t, \end{cases}$$

- `cv2.THRESH_BINARY_INV` — set everything greater than t value to 0 and everything else to `maxval`:

$$I_{\text{binary\_inv}}(x,y) = \begin{cases} \text{maxval} & \text{if } I(x,y) \leq t, \\ 0 & \text{if } I(x,y) > t, \end{cases}$$

- `cv2.THRESH_TRUNC` — truncate everything greater than t to `maxval` while keeping other intact:

$$I_{\text{trunc}}(x,y) = \begin{cases} I(x,y) & \text{if } I(x,y) \leq t, \\ \text{maxval} & \text{if } I(x,y) > t, \end{cases}$$

- `cv2.THRESH_TOZERO` — threshold everything less or equal than t to 0 while keeping other intact:

$$I_{\text{tozero}}(x,y) = \begin{cases} 0 & \text{if } I(x,y) \leq t, \\ I(x,y) & \text{if } I(x,y) > t, \end{cases}$$

- `cv2.THRESH_TOZERO_INV` — threshold everything greater than t to 0 while keeping other intact:

$$I_{\text{tozero\_inv}}(x,y) = \begin{cases} I(x,y) & \text{if } I(x,y) \leq t, \\ 0 & \text{if } I(x,y) > t. \end{cases}$$

The `cv2.THRESH_BINARY` method is used for simple threshold binarization according to the formula above.

Let us try and execute single thresholding with a threshold value of 127 to segment every pixel above the threshold to 1 and every pixel below the threshold to 0.

```
# Single thresholding
t = 127
ret, I1st = cv2.threshold(I1gray, t, 1, cv2.
    THRESH_BINARY)
ShowImages([("Single thresholding", I1st)])
```

Figure 1.3. Single Thresholding

By varying the thresholding parameter $\boldsymbol{t}$, we can shift the binarization boundary.

1.1.3 Double Thresholding

In some cases, the single thresholding is not enough, for example, if we need to select a region. In this case, the double (range) thresholding is used. The double thresholding method is described by the following formula:

$$I_{dt}(x,y) = \begin{cases} 0, & I(x,y) \leq t_1, \\ 1, & t_1 < I(x,y) \leq t_2, \\ 0, & I(x,y) > t_2, \end{cases}$$

where

- I is the source image;
- I_{dt} is the double threshold binarized image;
- t_1 is the lower binarization threshold;
- t_2 is the upper binarization threshold.

OpenCV does not provide a special function for double thresholding, however, it can be done by the subsequent call of two threshold functions with different methods:

1. At first, should set all values above t_2 to zeros while keeping other intact (`THRESH_TOZERO_INV` method);
2. Secondly, set all values below t_1 to zeros (`THRESH_BINARY` method).

Let us try executing the double thresholding with $(127, 200)$ range to segment every pixel within this range to 1 and every pixel outside of this range to 0.

```
# Double thresholding
t1 = 127
t2 = 200
ret, I1dt = cv2.threshold(I1gray, t2, 1, cv2.
    THRESH_TOZERO_INV)
ret, I1dt = cv2.threshold(I1dt, t1, 1, cv2.
    THRESH_BINARY)
ShowImages([("Double thresholding", I1dt)])
```

By varying the thresholding parameters `t1` and `t2` we can shift the binarization range borders.

Figure 1.4. Double Thresholding

1.1.4 Automatic Thresholding

Binarization thresholds t, t_1, and t_2 can either be set manually or calculated using special algorithms. In the case of automatic threshold calculation, the following algorithms can be used.

1.1.5 Mean value

Find the maximum I_{max} and minimum I_{min} intensity values of the original grayscale image and find their arithmetic mean. The arithmetic mean will be the global binarization threshold t:

$$t = \frac{I_{max} - I_{min}}{2}.$$

1.1.6 Intensity gradient weighting

Find the optimal threshold t based on the modulus of each pixel intensity gradient. For this, it is required to first calculate the modulus of the gradient at each image point (x,y):

$$G(x,y) = \max\left\{|I(x+1,y) - I(x-1,y)|, |I(x,y+1) - I(x,y-1)|\right\},$$

then calculate the optimal threshold value t:

$$t = \frac{\sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} I(x,y)G(x,y)}{\sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} G(x,y)}.$$

1.1.7 Otsu method

The optimal threshold t can be calculated by the statistical Otsu method [1] which splits all pixels into two classes 1 and 2. This method minimizes the variance within each class $\sigma_1^2(t)$ and $\sigma_2^2(t)$, and maximizes the variance between classes.

The algorithm for calculating the threshold by the Otsu method consists of the following steps:

1. Compute an image histogram of intensities, and probabilities $p_i = \frac{n_i}{N}$ for each intensity level, where n_i is the number of pixels with intensity level i, and N is the number of pixels in the image.

2. Set the initial threshold $t = 0$ and threshold $k \in (0, L)$, which divides all pixels into two classes, where L is the maximum value of the image intensity.

3. In the loop for each value of threshold from $k = 1$ to $k = L-1$: 1. Compute probabilities of two classes $\omega_j(0)$, and arithmetic mean $\mu_j(0)$, where $j = \overline{1,2}$:

$$\omega_1(k) = \sum_{s=0}^{k} p_s,$$

$$\omega_2(k) = \sum_{s=k+1}^{L} p_s = 1 - \omega_1(k),$$

$$\mu_1(k) = \sum_{s=0}^{k} \frac{s \cdot p_s}{\omega_1},$$

$$\mu_2(k) = \sum_{s=k+1}^{L} \frac{s \cdot p_s}{\omega_2}.$$

2. Calculate the interclass variance $\sigma_b^2(k)$:

$$\sigma_b^2(k) = \omega_1(k)\omega_2(k)(\mu_1(k) - \mu_2(k))^2.$$

3. If the calculated value $\sigma_b^2(k)$ is greater than the current value t, assign the value of the interclass variance to the threshold $t = \sigma_b^2(k)$.

4. The optimal threshold t corresponds to the maximum $\sigma_b^2(k)$.

In OpenCV, thresholding with the threshold calculated by the Otsu method is executed with the same `cv2.threshold(...)` function. To use the Otsu threshold method, you have to use the `cv2.THRESH_OTSU` parameter as a thresholding method. This function can be used to calculate the Otsu threshold without thresholding an image; for this, we have to run it in a dry run mode:

```
tr, ret = cv2.threshold(I, 0, 1, cv2.THRESH\
    _OTSU + cv2.THRESH\_DRYRUN)
print("The Otsu threshold value is \{\}".
    format(tr))
```

Let us try executing the thresholding with the threshold calculated by the Otsu method.

```
# Otsu thresholding
tr, I1otsu = cv2.threshold(I1gray, 0, 1, cv2.
    THRESH_OTSU)
ShowImages([("Otsu thresholding", I1otsu)])
```

1.1.8 Adaptive thresholding

Adaptive methods do not work with the entire image, but only with its fragments. Such approaches are often used when working with what represent non-uniformly illuminated objects.

In OpenCV, adaptive thresholding is performed with the `cv2.adaptiveThreshold(image, maxval, algorithm, method, blockSize, c)` function. It supports two adaptive thresholding algorithms:

- `ADAPTIVE_THRESH_MEAN_C` uses simple rectangular kernel mean;

Figure 1.5. Otsu Thresholding

- `ADAPTIVE_THRESH_GAUSSIAN_C` uses a kernel with Gauss weights.

These algorithms support two thresholding methods:

- `cv2.THRESH_BINARY` to set everything greater than the threshold value to the *maxval* and everything else to 0;
- `cv2.THRESH_BINARY_INV` to set everything greater than the threshold value to 0 and everything else to *maxval*.

Kernel size is defined by the `blockSize` parameter, and the parameter value `c` is subtracted from the mean value calculated within a window.

```
I1ta = cv2.adaptiveThreshold(I1gray, 255, cv2.
    ADAPTIVE_THRESH_GAUSSIAN_C,
                                 cv2.THRESH_BINARY,
     121, 5)
ShowImages([("Adaptive thresholding", I1ta)])
```

1.2 Task 2. Color Segmentation

Take an arbitrary image containing face(s). Perform the image segmentation according to the Weber principle. Perform image segmenta-

Figure 1.6. Adaptive Thresholding

tion based on the skin color and try different formulas on the photos with various photo illumination conditions.

1.2.1 Weber Principle

The Weber principle supposes that there is a distance that is not noticeable for human eye, and depending on the intensity value, this range can be calculated using the formula:

$$W(I) = \begin{cases} 20 - \dfrac{12I}{88}, & \text{if } 0 \leq I \leq 88, \\ 0.002\,(I-88)^2, & \text{if } 88 < I \leq 138, \\ \dfrac{7(I-138)}{117} + 13, & \text{if } 138 < I \leq 255. \end{cases}$$

Hence, the Weber principle assumes that the human eye does not perceive well the difference in the gray levels between $I(n)$ and $I(n) + W(I(n))$, where $W(I(n))$ is the Weber function, n is the class number, I is the piecewise non-linear grayscale function. Based on this principle, the segmentation algorithm is designed for the segmentation of grayscale images.

The Weber principle segmentation algorithm includes of the following steps:

1. Initialize initial conditions: first class number $n = 1$, all image pixels are not segmented.

2. Find the minimum value of the not-segmented image part and store it to $I(n)$.

3. Calculate the $W(I(n))$ value according to the Weber formula and assign the segment n and the value $I(n)$ to all pixels whose intensities are in the range $[I(n), I(n) + W(I(n))]$.

4. If there are any not-segmented pixels in the image, then increment n ($n = n + 1$) and go to step 2.

5. If there are no non-segmented pixels then the segmentation is finished.

The segmentation data can be displayed with artificial colors and a JET color scheme:

```
Ijet = cv2.applyColorMap(((I - I.min()).astype
    (np.float32) * 255 /
         (I.max() - I.min())).astype(np.uint8),
     cv2.COLORMAP_JET)
```

Example of implementation of the Weber function

```
## The Weber function
# @param[in] i An intensity value in [0, 255]
    range
def W(i):
  if i < 0:
    return 0
  if i > 255:
    return 255

  if i <= 88:
    return int(20 - 12 * i / 88)

  if i <= 138:
    return int(0.002 * (i - 88) * (i - 88))

```

```
  return int(7 * (i - 138) / 117 + 13)
  # End of W()

I1segments = np.zeros_like(I1gray)
I1weber = np.zeros_like(I1gray)
n = 1
Wn = [i for i in range(256)]

# While there are pixels without segment
while (I1segments == 0).any():
  # Find the new minimum
  In = I1gray[I1segments == 0].min()
  # Calculate Weber distance
  IWn = W(In)
  Wn[n] = In
  # Find all pixels with intensity in a given
    range
  mask = np.logical_and(I1gray >= In, I1gray
    <= In + IWn)
  # Segment them
  I1segments[mask] = n
  # And plot
  I1weber[mask] = In
  # Go to the next segment
  n = n + 1

Wn = Wn[:n]
print("Weber function of {} values for each
    segment: {}".format(len(Wn), Wn))

ShowImages([("Test", np.vectorize(lambda n: Wn
    [n], otypes = [np.uint8])(I1segments))])
cv.imwrite("results/weber_example.png",
    I1segments)
cv.imwrite("results/weber_segments.png", np.
    vectorize(lambda n: Wn[n], otypes = [np.
    uint8])(I1segments))
```

```
cv.imwrite("results/weber_jet.png", cv.
    applyColorMap(((I1segments - I1segments.min
    ()).astype(np.float32)  * 255 /
       (I1segments.max() - I1segments.min())).
    astype(np.uint8), cv.COLORMAP_JET))

ShowImages([("Weber transformation", I1weber),
            ("Weber segments", cv.
    applyColorMap(((I1segments - I1segments.min
    ()).astype(np.float32)  * 255 /
       (I1segments.max() - I1segments.min())).
    astype(np.uint8), cv.COLORMAP_JET))], 2)
```

Table 1.1. Weber function values (44 data points)

0	25	42	57	70	81	90	91	92	93	94
95	96	97	98	99	100	101	102	103	104	105
106	107	108	109	110	111	113	115	117	119	121
124	127	131	135	140	154	168	183	199	216	234

Figure 1.7. Weber Transformation

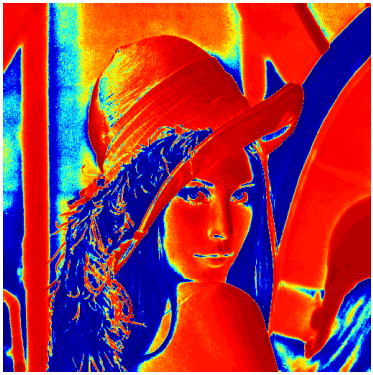


Figure 1.8. Weber Segments

1.2.2 Skin Color Segmentation

The general principle of skin color segmentation is determining the criterion for the proximity of the pixel intensity to the skin tone. It is difficult to describe the **skin tone** analytically since its description is based on the human perception of color. Moreover, it changes with lighting, differs among nationalities, etc.

There are several analytical descriptions for images in the RGB color space that allow a pixel to be assigned to the *skin* class. For example, at uniform daylight illumination, the following conditions may be used to segment a part of an image as skin:

$$\begin{cases} R > 95, \\ G > 40, \\ B > 20, \\ \max\{R, G, B\} - \min\{R, G, B\} > 15, \\ |R - G| > 15, \\ R > G, \\ R > B, \end{cases}$$

In the case of the flashlight or daylight lateral illumination:

$$\begin{cases} R > 220, \\ G > 210, \\ B > 170, \\ |R - G| \leq 15, \\ G > B, \\ R > B, \end{cases}$$

Also, there is a formula that is based on normalized RGB values (r, g, b) which are normalized by the sum of R, G, B:

$$\begin{cases} r = \dfrac{R}{R+G+B}, \\ g = \dfrac{G}{R+G+B}, \\ b = \dfrac{B}{R+G+B}, \\ \dfrac{r}{g} > 1.185, \\ \dfrac{rb}{(r+g+b)^2} > 0.107, \\ \dfrac{rg}{(r+g+b)^2} > 0.112. \end{cases}$$

To implement these segmentation methods with OpenCV we use NumPy matrix operations and create a mask to segment skin. Let us implement the skin color segmentation for the first of these three formulas.

```
# We will load a random image with a face on it
# (image source: https://thispersondoesnotexist.com/)
import requests

url = "https://thispersondoesnotexist.com/"
try:
    response = requests.get(url)
    I2 = cv2.imdecode(np.asarray(bytearray(response.content), dtype=np.uint8), cv2.IMREAD_COLOR)
    if not isinstance(I2, np.ndarray) or I2.data == None:
        print("Error loading URL \"{}\"".format(url))
except BaseException:
    # If something got wrong then load a local failsafe file
```

```
13      print("Something went wrong while loading
    URL \"{}\"".format(url))
14      print("Loading a local failsafe file...")
15
16      fn = "images/people.jpg"
17      I2 = cv2.imread(fn, cv2.IMREAD_COLOR)
18      if not isinstance(I2, np.ndarray) or I2.
    data == None:
19          print("Error reading file \"{}\"".
    format(fn))
20          exit()
21
22  # Split into layers
23  I2BGR = cv2.split(I2)
24
25  # Build up the mask
26  I2skin = np.logical_and.reduce(
27      [I2BGR[2] > 95,    # R > 95
28       I2BGR[1] > 40,    # G > 40
29       I2BGR[0] > 20,    # B > 20
30       np.maximum.reduce(I2BGR) - np.minimum.
    reduce(I2BGR) > 15,  # Max(R,G,B) - Min(R,G
    ,B) > 15
31       np.abs(I2BGR[2] - I2BGR[1]) > 15,  # |R -
     G| > 15
32       I2BGR[2] > I2BGR[0],  # R > B
33       I2BGR[2] > I2BGR[1]]  # G > B
34  )
35
36  # Create a copy of the source image and remove
      everything except for the face from it
37  I2copy = I2.copy()
38  I2copy[~I2skin] = 0
39
40  # Display the result
41  ShowImages([("Source image", I2),
42              ("Skin segment", I2skin),
43              ("Skin only", I2copy)], 3)
```

```
44
45 cv.imwrite("results/skin_source.png", I2)
46 cv.imwrite("results/skin_segment_1.png",
       I2skin.astype(np.uint8) * 255)
47 cv.imwrite("results/skin_only_1.png", I2copy)
```

(a) Source Image (b) Skin Segment (c) Skin

Figure 1.9. Skin Color Segmentation

1.3 Task 3. Chromatic Segmentation

Take an arbitrary image containing a limited number of colored objects. Perform image segmentation in the uniform color space by the nearest neighbors method. Perform image segmentation in the CIE Lab color space by the k-means method.

1.3.1 Segmentation in CIE Lab color space

The algorithm's general idea is to divide a color image into segments of dominant colors. For this, we will let the user select a set of dominant colors, then search for a distance to each of these colors and segment image pixels by the shortest distance.

First, we have to let the user select points on the image. To do it we will use the OpenCV GUI library and create a window with the mouse callback function `MouseHandler()`. This function will register each left mouse button click and add the corresponding pixel coordinates to an array with the selected points. This will continue until the window is shown or until the user presses an ESC button.

Thus, the general workflow is as follows:

1. Create a GUI window with `cv2.imshow(window_name, I)` function.
2. Register a mouse callback with `cv2.setMouseCallback(window_name, MouseHandler, params)`.
3. Run the infinite loop and break it in case if:
   - Window is closed (checked by `cv2.getWindowProperty(window_name, cv2.WND_PROP_VISIBLE) < 1`);
   - Or the ESC button is pressed (checked by `cv2.waitKey(100) == 27`).

In the mouse handler (`MouseHandler()`) function we check the event type, append click coordinates to the selected points list, and update the shown image by making a new image and calling `cv2.imshow(window_name, I)` function again.

```
# Read an image from file
fn = "images/people.jpg"
I3 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I3, np.ndarray) or I3.data == None:
    print("Error reading file \"{}\"".format(fn))
    exit()

# Set window name and other parameters
window_name = "Select points"
points = []
radius = 10

# Define the mouse handler
def MouseHandler(event, x, y, flags, param):
    # Only the left button click event is processed
```

```
16      # All others are ignored
17      if event != cv2.EVENT_LBUTTONDOWN:
18          return
19      # Append new point coordinates
20      points.append((x, y))
21      # Create a new image
22      I3copy = I3.copy()
23      # And draw circles for each registered
        mouse click
24      for p in points:
25          cv2.circle(I3copy, p, radius, (0, 0,
        255), 2)
26      # Then update the image shown in the
        window
27      cv2.imshow(window_name, I3copy)
28
29  # Create a window, show it, and register the
        callback
30  cv2.namedWindow(window_name)
31  cv2.imshow(window_name, I3)
32  cv2.setMouseCallback(window_name, MouseHandler
        )
33
34  # Wait for an ESC key press or window is
        closed
35  while True:
36      if cv2.waitKey(100) == 27:
37          break
38      if cv2.getWindowProperty(window_name, cv2.
        WND_PROP_VISIBLE) < 1:
39          break
40
41  # Destroy all windows data
42  cv2.destroyAllWindows()
43
44  # Draw circles for each registered mouse click
45  I3copy = I3.copy()
46  for p in points:
```

```
47      cv2.circle(I3copy, p, radius, (0, 0, 255),
        2)
48  # And display it
49  ShowImages([("Labels", I3copy)])
```

Figure 1.10. Chosen Labels

Now that we have the list of the selected points we may proceed with the general segmentation algorithm:

1. Convert an image to CIE Lab color space.

2. Take an area around selected pixels and calculate the mean value in this area to get the reference ab color for each segment.

3. Calculate the Euclidean distance in ab layers $d = \sqrt{(a_2 - a_1)^2 + (b_2 - b_1)^2}$ for each image pixel from the selected points.

4. For each image pixel select the nearest color in ab coordinate space. This will be the desired segmentation.

```
1  # 1. Convert to CIE Lab
2  I3lab = cv2.cvtColor(I3, cv2.COLOR_BGR2LAB)
3  # Split it into layers
4  I3lab = cv2.split(I3lab)
```

```
1  # 2. Instead of taking the selected pixel
       value
```

```
# We will calculate the mean color value in an area around the pixel
points_ab = []
points_bgr = []
for p in points:
    mask = np.zeros_like(I3lab[0])
    cv2.circle(mask, p, radius, 255, -1)
    a = I3lab[1].mean(where=mask > 0)
    b = I3lab[2].mean(where=mask > 0)
    points_ab.append((a, b))
    points_bgr.append(I3[mask > 0, :].mean(axis=(0,)))
```

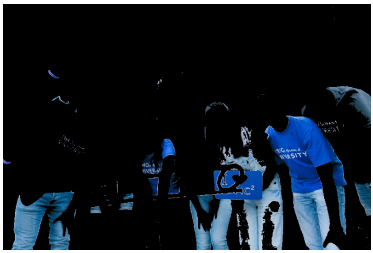
```
# 3. Calculate distance
distances = []
for ab in points_ab:
    distances.append(np.sqrt(
        np.power(I3lab[1] - ab[0], 2) +
        np.power(I3lab[2] - ab[1], 2)
    ))

# Calculate the minimum distance among them all
distance_min = np.minimum.reduce(distances)
```

```
# 4. Create segmented areas
I3labels = np.zeros_like(I3lab[0], dtype=np.uint8)

I3segments = []
# Fill them for each selected color
for i in range(len(points_ab)):
    Itmp = np.zeros_like(I3)
    # Segmentation mask is created by finding all pixels with a matching distance
    mask = distance_min == distances[i]
    # Add labels to the segmentation plot
    I3labels[mask] = i
```

```
    # Segmented area of source image
    Itmp[mask] = I3[mask]
    # Add the segmented area to a list of segments
    I3segments.append("Segment {}".format(i), Itmp)

# Display it
ShowImages([("Labels", cv2.applyColorMap(
    ((I3labels - I3labels.min()).astype(np.float32) * 255 /
     (I3labels.max() - I3labels.min())).astype(np.uint8), cv2.COLORMAP_JET)
), ("Segments", I3segments)], 2)
```

Figure 1.11. Color Plot

(a) First Mask



(b) Second Mask

(c) Third Mask

(d) Fourth Mask

Figure 1.12. All Segments

Finally, we can place the distribution of image pixel colors to a plot in the (a, b) coordinate system. The plot pixel color is defined by the mean BGR which was calculated along with calculating the mean ab value and stored in the `points_bgr` array.

```
# The color distribution plot
I3plot = np.full((256, 256, 3), 255, dtype=np.
    uint8)
for i in range(len(points_ab)):
    mask = I3labels == i
    I3plot[I3lab[1][mask], I3lab[2][mask], :]
    = points_bgr[i]

# Display it
ShowImages([("Color plot", I3plot)])
```

Figure 1.13. Color Plot

1.3.2 k-means Clustering

The idea of the method is to determine the centers of k-clusters and assign to each cluster the pixels closest to these centers. All pixels are considered as vectors $x_i, i = \overline{1,p}$. The segmentation algorithm consists of the following steps:

1. Randomly determine k vectors $m_j, j = \overline{1,k}$, which are referred to as initial centers of clusters.

2. Update the mean values of the vectors m_j by calculating the distances from each vector x_i to each m_j and their classification according to the criterion of minimal distance from the vector to the cluster, recalculation of the average values m_j across all clusters.

3. Repeat step 2 until the cluster centers stop changing.

The implementation of the method is very similar to the previous approach and contains a number of similar actions. We will work in the CIE Lab color space, so the first step is transformation from the BGR space to the Lab and splitting into layers:

```
Ilab = cv2.cvtColor(I, cv2.COLOR_BGR2LAB)
Ilab = cv2.split(Ilab)
```

Then we have to merge a and b layers of the Lab representation of our source image and use the NumPy reshaping function to create the two-dimensional array of the image pixel colors:

```
ab = cv2.merge([Ilab[1], Ilab[2]])
ab = ab.reshape(-1, 2).astype(np.float32)
```

This two-dimensional array can be then passed to the `cv2.kmeans()` function that performs the k-means clustering. Parameters allow us to define stop criteria and the number of attempts to select a set of starting points. In the following code the stop criterion is defined as not more than 10 iterations of difference between the steps less than 1. The starting points are selected randomly (due to the `cv2.KMEANS_RANDOM_CENTERS` flag being used) and selection is done 10 times. After algorithm execution is finished, the returned `labels` parameter is reshaped back to an original image shape:

```
k = 3
criteria = (cv2.TERM_CRITERIA_EPS + cv2.
    TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret, labels, centers = cv2.kmeans(ab, k, None,
     criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
labels = labels.reshape(Ilab[0].shape)
```

Then, it is possible to use the generated labels to segment the image into a set of images or masks:

```
segmentedFrames = []
for i in range(k):
    Itmp = np.zeros_like(I)
    mask = labels == i
    Itmp[mask] = I[mask, :]
    segmentedFrames.append(Itmp)
```

Example

```
# TODO Place your solution here

Ilab = cv2.cvtColor(I3, cv2.COLOR_BGR2LAB)
Ilab = cv2.split(Ilab)

ab = cv2.merge([Ilab[1], Ilab[2]])
```

```
ab = ab.reshape(-1, 2).astype(np.float32)

k = 4
criteria = (cv2.TERM_CRITERIA_EPS + cv.
    TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret, labels, centers = cv2.kmeans(ab, k, None,
     criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
labels = labels.reshape((Ilab[0].shape))

segmentedFrames = []
for i in range(k):
  Itmp = np.zeros_like(I3)
  mask = labels == i
  Itmp[mask] = I3[mask, :]
  segmentedFrames.append((f"Segment {i}", Itmp
    ))

# Display it
ShowImages([("Labels", cv.applyColorMap(((
    labels - labels.min()).astype(np.float32)
    * 255 /
                                            (
    labels.max() - labels.min()))).astype(np.
    uint8), cv.COLORMAP_JET))])
ShowImages(segmentedFrames, 2)

cv.imwrite("results/k_means_segments.png", cv.
    applyColorMap(((labels - labels.min()).
    astype(np.float32)  * 255 /

       (labels.max() - labels.min()))).astype(np
    .uint8), cv.COLORMAP_JET))
for i in range(len(segmentedFrames)):
  cv.imwrite("results/k_means_segment_{}.png".
    format(i), segmentedFrames[i][1])
```

Figure 1.14. k-means Labels

(a) k-means Segment One

(b) k-means Segment Two

(c) k-means Segment Three

(d) k-means Segment Four

Figure 1.15. Result k-means Method

1.4 Task 4. Texture Segmentation

Take an arbitrary image containing two heterogeneous textures. Perform the texture segmentation of the image.

In texture segmentation, three main approaches are used to describe texture: statistical, structural, and spectral. In the practical assignment, we will consider the statistical approach that describes the segment texture as smooth, rough, or grainy.

We will consider the image intensity I as a random variable z, which corresponds to the distribution probability $p(z)$ calculated from the image histogram. The *Central moment* of order n of a random variable z is the parameter $\mu_n(z)$ calculated by the formula:

$$\mu_n(z) = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i),$$

where

- L is the number of intensity levels;
- m is the mean value of a random variable z:

$$m = \sum_{i=0}^{L-1} z_i p(z_i).$$

The expression above implies that $\mu_0 = 1$ and $\mu_1 = 0$. To describe the texture, the *variance* of a random variable is important, which is equal to the second moment $\sigma^2(z) = \mu_2(z)$ and is a measure of the brightness contrast. It can be used to calculate the features of *smoothness*.

Let us introduce a measure of relative smoothness R:

$$R = 1 - \frac{1}{1 + \sigma^2(z)},$$

The relative smoothness is zero for the areas with constant intensity (zero variance) and approaches unity for large variances $\sigma^2(z)$. For grayscale images with an intensity range $[0, 255]$, it is necessary to normalize the variance to the range $[0, 1]$, since the values of the variances

will be too large for the initial range. Normalization is carried out by dividing the variance $\sigma^2(z)$ by $(L-1)^2$.

The *standard deviation* is also often used as a texture characteristic:

$$s = \sigma(z).$$

The third point is the *histogram symmetry* characteristic. To estimate the texture features, the *entropy* E function is used, which determines the spread of neighboring pixels intensities:

$$E = -\sum_{i=0}^{L-1} p(z_i) \log_2 p(z_i).$$

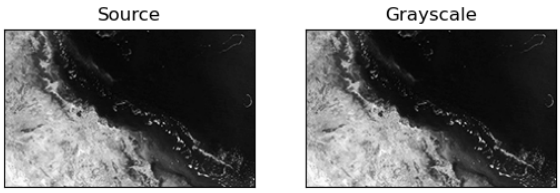
Another important characteristic that describes the texture is the *uniformity measure* U, which evaluates the uniformity of the histogram:

$$U = \sum_{i=0}^{L-1} p^2(z_i).$$

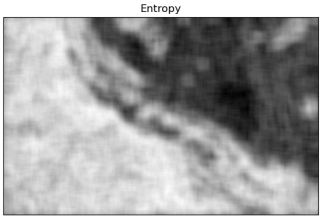
Let us try and do the segmentation using *Entropy* of a pixel neighborhood as a texture characteristic. For this we have to calculate the image Entropy first, then threshold it to get the starting morphology. Unfortunately, OpenCV does not provide the corresponding function for Entropy calculation; however, in the case of Python it can be found in the `scikit-image` library named `skimage.filters.rank.entropy()`. To define the neighboring area we will use the rectangular 9×9 kernel created by `skimage.morphology.square(9)` function. Since scikit-image converts an image to `float64` type we have to do a backward conversion along with normalization to a $[0, 1]$ range.

```
1 # Read an image from file
2 fn = "images/texture.jpg"
3 I4 = cv2.imread(fn, cv2.IMREAD_COLOR)
4 if not isinstance(I4, np.ndarray) or I4.data
      == None:
5     print("Error reading file \"{}\"".format(
      fn))
6     exit()
```

```
7
8  # Convert to grayscale
9  I4gray = cv2.cvtColor(I4, cv2.COLOR_BGR2GRAY)
10 # And display
11 ShowImages([("Source", I4), ("Grayscale",
       I4gray)], 2)
12
13 # Calculate entropy
14 I4e = skimage.filters.rank.entropy(I4gray,
       skimage.morphology.square(9)).astype(np.
       float32)
15 # Normalize it to [0, 1] range
16 I4en = (I4e - I4e.min()) / (I4e.max() - I4e.
       min())
17 # And display
18 ShowImages([("Entropy", I4en)], 2)
```



(a) Result of the Grayscaling



(b) Entropy Method

Figure 1.16. Entropy

Next, we threshold the entropy image and filter the resulting morphology. For thresholding, we will use the Otsu thresholding, while the morphological filtering will be performed in three steps:

1. First, remove the connected regions (equivalent to MATLAB's `bwareaopen()` function);

2. Secondly, remove the internal defects with the closing operation (executed by `cv2.morphologyEx()` function with `cv2.MORPH_CLOSE` parameter) and rectangular structure element of size 9×9 (created by `cv2.getStructuringElement()` function with shape parameter `cv2.MORPH_RECT`);

3. And, thirdly, fill the remaining holes (equivalent to MATLAB's `imfill('holes')` function).

Even if OpenCV lacks MATLAB's `bwareaopen(A, dim)` and `imfill(I, 'holes')` functions, they can be easily implemented using OpenCV's `connectedComponentsWithStats()`. You may check the `pa_utils.py` file for the implementation of these functions with OpenCV. However, now we will simply import and use them.

```
from pa_utils import bwareaopen, imfillholes

# Threshold
ret, I4t = cv2.threshold(np.uint8(I4en * 255),
    0, 255, cv2.THRESH_OTSU)

# Remove connected regions
I4bwao = bwareaopen(I4t, 2000)

# Remove internal defects with closing
nhood = cv2.getStructuringElement(cv2.
   MORPH_RECT, (9, 9))
I4close = cv2.morphologyEx(I4bwao, cv2.
   MORPH_CLOSE, nhood)

# Fill holes to get the final mask
I4mask = imfillholes(I4close)

```

```
ShowImages([("Threshold", I4t),
            ("bwareaopen", I4bwao),
            ("After closing", I4close),
            ("Fill holes", I4mask)], 2)
```

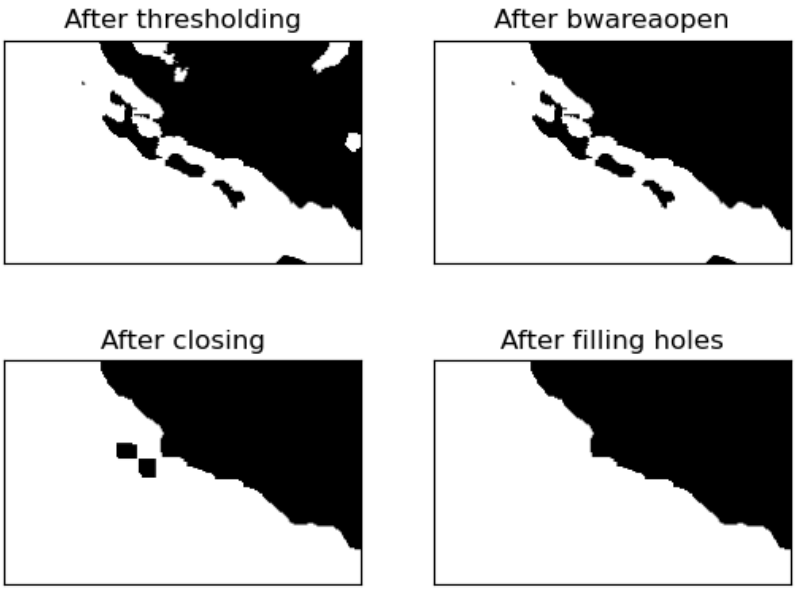


Figure 1.17. Final Mask

Next, using the OpenCV `cv2.findContours()` function it is possible to find the shape contours and then draw them oagainst a black background with the `cv2.drawContours()` function to define the contour mask. Then we can use this mask to apply the border to the source image.

```
# Find boundary
contours, h = cv2.findContours(I4mask, cv2.
    RETR_TREE, cv2.CHAIN_APPROX_NONE)
I4boundary = np.zeros_like(I4mask)
```

```
cv2.drawContours(I4boundary, contours, -1,
    255, 1)

# Outline the whole image as well
cv2.rectangle(I4boundary, (0, 0), (I4.shape[1]
     - 1, I4.shape[0] - 1), 255)

# Separate segments
I4seg1 = I4.copy()
I4seg1[I4mask == 0] = 0
I4seg2 = I4.copy()
I4seg2[I4mask != 0] = 0
I4split = I4.copy()
I4split[I4boundary != 0] = [0, 0, 255]

ShowImages([("Boundary", I4boundary),
            ("Split image", I4split),
            ("Segment 1", I4seg1),
            ("Segment 2", I4seg2)], 2)
```

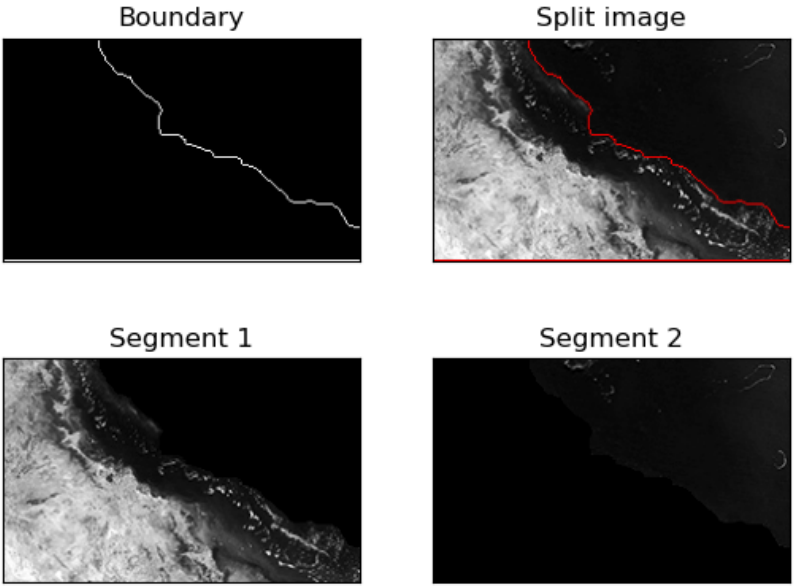


Figure 1.18. Boundary and Segmentation

We also suggest another way of implementing this method. We will use a little trick here. We will search for the contours of the image white segment with a border where the border pixels are replicated. Then, when displaying this contour on top of the source image, the image edge will be outside the visuazaliation area and will not be visible. This will make it look as if we only highlighted the border between the two segments without outlining the whole image.

```
1 # Find boundary
2 #
3 I4mask_with_border = cv.copyMakeBorder(I4mask,
      1, 1, 1, 1, cv.BORDER_REPLICATE)
4 contours, h = cv.findContours(
     I4mask_with_border, cv.RETR_TREE, cv.
     CHAIN_APPROX_NONE, None, None, (-1, -1))
5 I4boundary = np.zeros_like(I4mask)
```

```
#cv.polylines(I4boundary, contours, True, 255, 1)
cv.drawContours(I4boundary, contours, -1, 255, 1)
# Outline the whole image as well
#cv.rectangle(I4boundary, (0, 0), (I4.shape[1] - 1, I4.shape[0] - 1), 255)

# Separate segments
I4seg1 = I4.copy()
I4seg1[I4mask == 0] = 0
I4seg2 = I4.copy()
I4seg2[I4mask != 0] = 0
I4split = I4.copy()
I4split[I4boundary != 0] = [0, 0, 255]

ShowImages([("Boundary", I4boundary),
            ("Split image", I4split),
            ("Segment 1", I4seg1),
            ("Segment 2", I4seg2)], 2)
```

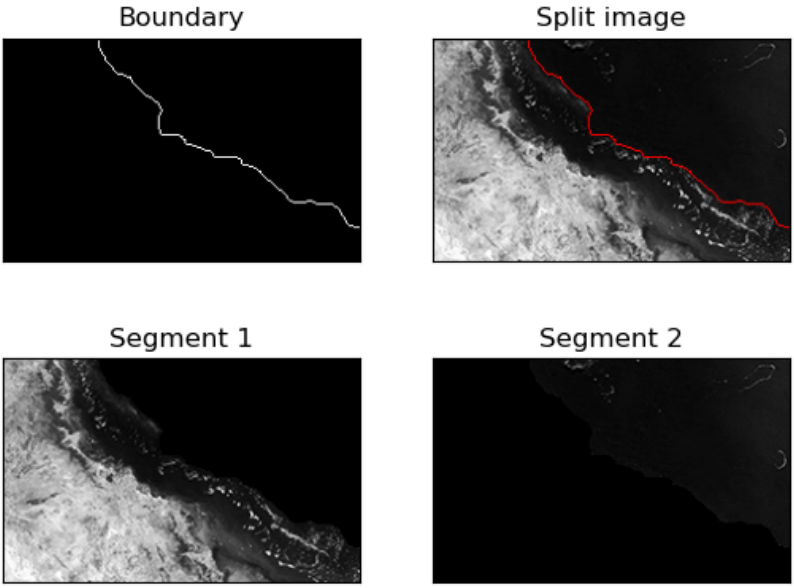


Figure 1.19. Boundary and Segmentation Second Method

1.5 Procedure of Performing Practical Assignment

All tasks are based on the material presented in the corresponding sections of the assignment. Students are expected to write code for each task using Python, OpenCV, and NumPy.

Task 1. Binarization

1. **Simple and double thresholding**
   Take an arbitrary image and threshold it using:

   - single thresholding method;
   - double thresholding method.

   Display the original image and both thresholding results.

2. **Automatic thresholding**
   Take an arbitrary image and threshold it using three automatic threshold calculation methods:

   - mean value threshold;
   - intensity gradient weighted threshold;
   - Otsu threshold.

   Hints for implementation:

   - *Mean value threshold:* Use `np.min()`, `np.max()`, and simple arithmetic.
   - *Gradient weighted threshold:*
     - Use `cv2.copyMakeBorder(I, 1, 1, 1, 1, cv2.BORDER_REPLICATE)` to create an image with a 1-pixel border.
     - Calculate horizontal and vertical differences using NumPy slicing: `dx = np.abs(I_border[1:-1, 2:] - I_border[1:-1, :-2])` `dy = np.abs(I_border[2:, 1:-1] - I_border[:-2, 1:-1])`
     - Gradient magnitude = `np.maximum(dx, dy)`
     - Apply the formula: `t = np.sum(I * G) / np.sum(G)`
   - *Otsu threshold:* Use `cv2.threshold(I, 0, 1, cv2.THRESH_OTSU)` — the threshold value is returned as the first parameter.

Task 2. Color Segmentation

1. **Weber principle segmentation**
   Use the Weber principle to implement the image segmentation algorithm . Display the resulting images using artificial colors (e.g., JET colormap).

   Hints for implementation:

   - Convert the image to grayscale first.

- Use the Weber formula: `W(I) = 10 + 240 * (I / 255) ** 2` (example approximation — adjust as needed).
- Iterate while there are unsegmented pixels:
  (a) Find the minimum intensity among unsegmented pixels.
  (b) Calculate the upper bound: `min_val + W(min_val)`.
  (c) Assign the current segment index to all pixels in the range [min_val, upper_bound].
- For visualization, use `cv2.applyColorMap()` with `cv2.COLORMAP_JET`.

2. **Skin color segmentation**
   Implement the two remaining skin segmentation algorithms (formulas 2 and 3 from the theoretical section). Display the resulting images.

   Hints for implementation:

   - *Formula 2 (flashlight illumination):*
     - Conditions: `R > 10 && G > 10 && B > 10` and `R - G <= 16` and `R - B >= -16` and `G - B >= -16` and `G - B <= 16`.
     - Works best on `images/flashlight.jpg`.
   - *Formula 3 (normalized RGB):*
     - Normalize RGB values: `r = R / (R+G+B)`, `g = G / (R+G+B)`, `b = B / (R+G+B)`.
     - Typical skin conditions: `r > 0.36 && r < 0.47`, `g > 0.28 && g < 0.36`, `b > 0.19 && b < 0.28`.
     - Use `np.logical_and.reduce()` to combine multiple conditions efficiently.
   - Create a binary mask and apply it to the original image to show only skin regions.

Task 3. Chromatic Segmentation

1. **Segmentation in CIE Lab color space with user-selected points**

Modify the provided algorithm to use the **HSV color model** instead of CIE Lab.

Hints for implementation:

- Convert BGR to HSV: `cv2.cvtColor(I, cv2.COLOR_BGR2HSV)`.
- Split into H, S, V channels.
- For segmentation, use only H (hue) and S (saturation) channels — they contain color information.
- Calculate Euclidean distance in 2D space (H and S) instead of (a, b).
- *Note:* H channel values are in the range $[0, 179]$ in OpenCV — be careful with the circular nature of the hue.

2. **k-means clustering**
Implement the k-means clustering algorithm for image segmentation in the CIE Lab color space. Display the resulting segmented image.

Hints for implementation:

- Convert the image to Lab color space: `cv2.cvtColor(I, cv2.COLOR_BGR2LAB)`.
- Extract a and b channels only (ignore L for color-based segmentation).
- Reshape ab channels: `ab = ab.reshape((-1, 2)).astype(np.float32)`.
- Use `cv2.kmeans()` with parameters:
  - `K = 3` (or try different values);
  - `criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)`;
  - `flags = cv2.KMEANS_RANDOM_CENTERS`.
- Reshape the labels back to the original image dimensions.
- Create separate images for each segment using masking.

Task 4. Texture Segmentation

1. **Entropy-based texture segmentation**
   Take an arbitrary image containing two heterogeneous textures. Implement texture segmentation using entropy as a texture descriptor.

   Hints for implementation:

   - Convert image to grayscale.
   - Calculate entropy using: `skimage.filters.rank.entropy (I_gray, skimage.morphology.square(9))`.
   - Normalize entropy to [0, 1] range.
   - Threshold using Otsu method: `cv2.threshold(normalized_entropy * 255, 0, 255, cv2.THRESH_OTSU)`.
   - Apply morphological operations:
     (a) Remove small objects: use `bwareaopen()` from `pa_utils.py`.
     (b) Close the gaps: `cv2.morphologyEx()` with `cv2.MORPH_CLOSE`.
     (c) Fill the holes: `imfillholes()` from `pa_utils.py`.
   - Find the contours with `cv2.findContours()` and draw boundaries.
   - Separate segments by masking.

2. **Optional**
   Use another statistical texture parameter instead of entropy when implementing texture segmentation. Display and compare the results.

   Hints for implementation:

   - Possible texture parameters:
     – *Variance (smoothness):* `R = 1 - 1/(1 + variance_normalized)`.
     – *Standard deviation:* `s = std_dev`.
     – *Uniformity:* `U = sum(p(z)^2)`.

  - *Third moment (skewness):*
    `mu3 = sum((z - m)^3 * p(z))`.
- Calculate these locally using sliding window (e.g., `cv2.filter2D()` or manual loops with padding).
- Normalize and threshold similarly to entropy approach.

General Requirements

- All results must be visualized using the `ShowImages()` function from the `pa_utils.py` module.
- Code must be well-structured and commented.
- Each task must be implemented in a separate Jupyter notebook cell or section.

Questions for Practical Assignment Report Defense

- When is it appropriate to use Weber segmentation?
- What are the a and b color coordinates values for a grayscale image in the CIE Lab color space?
- What is the reason for performing an image segmentation in the CIE Lab color space instead of the original RGB one?

2 Practical Assignment No 2. Hough Transform

Objectives

- **Study the principles of the Hough transform**
  To understand the fundamental concept of voting and accumulation (accumulator) as a method for identifying geometric primitives in binary images, and to analyze the relationship between the image space and the parameter space.

- **Master computer vision tools**
  To gain practical skills in using OpenCV and scikit-image library functions to implement classical and probabilistic Hough transforms for detecting lines and circles.

- **Conduct experimental research on detection**
  To perform line and circle detection on real images, learn to adjust algorithm parameters (thresholds, resolution, radius range), and evaluate the results (number of objects, line lengths, circle parameters).

- **Compare implementations and deepen understanding**
  To compare the results of classical and probabilistic methods, as well as implementations from different libraries (OpenCV vs. scikit-image). Optionally, to implement a basic classical Hough transform algorithm from scratch to solidify understanding of its internal workings.

Preface

To successfully complete this practical assignment, students will need the following:

- **Python 3.x** with core scientific libraries:
  - `opencv` (`opencv-python` pip library) — for image I/O and Hough transform implementations;
  - `numpy` (<=1.26.4) — for efficient array and matrix manipulations;

  - `scikit-image` — for additional Hough transform functions and accumulator visualization;
  - `math` — for trigonometric calculations and distance computations.

  Example installation command: `pip install opencv-python, numpy==1.26.4, scikit-image`

- **Development environment:** It is recommended to use `Jupyter Notebook` or `Jupyter Lab` (locally or via Google Colab). All tasks are designed to be completed interactively in a `.ipynb` format. Alternatively, any Python IDE with support for interactive execution (e.g., VS Code with Python extension) is acceptable.

  To execute Python programs, you have to install a development environment, for example, Visual Studio Code with the Python development extension and additional Python packages to run programs – `ipykernel`.

- **Utility module:** The file `pa_utils.py` is provided alongside the assignment. This module contains auxiliary functions essential for the work:

  - `ShowImages()` — displays multiple images side-by-side with titles, used throughout all tasks for result visualization;
  - `exit()` — provides clean cell termination within Jupyter without stopping the kernel.

  The module is imported at the very beginning of the notebook and is used throughout all tasks to display intermediate and final results.

2.1 Preliminary

2.1.1 Common Locus of Points

The main principle of the Hough transform [5] is to find a common *locus of points*. For example, this approach is used when designing a triangle along three given sides. First, one side of the triangle is laid

off, after that the ends of the segment are considered as the centers of circles with the radii equal to the lengths of the second and the third segments. The intersection of the two circles is the common locus of points, from where the segments are drawn to the ends of the first segment. In other words, a *voting* of two points was held in favor of the probable location of the third vertex of the triangle. As a result of voting the *winner* was the point that got two votes (the points on the circles got one vote each, and outside them, they got zero), see 2.1.

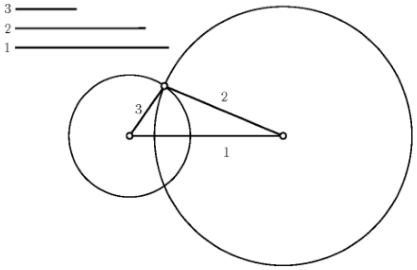


Figure 2.1. Triangle by its Sides

Let us generalize this idea for working with real data when the image has a large number of special feature points participating in the vote. Let us assume that it is necessary to search for a circle of the known radius R in a binary point set, and in this set, there may also be false points that do not lie on the desired circle. The set of possible circle centers for the desired radius around each characteristic point forms a circle of radius R, see 2.2 below. Thus, the point corresponding to the maximum intersection of the number of circles will be the center of the required radius circle.

2.1.2 Classic Hough Transform

The classic Hough transform is based on the point voting idea considered above. It was originally designed to select lines on binary images. The Hough transform uses the parameter space to search for geometric primitives. The most well-known parametric equation of lines is:

$$y = kx + b,$$

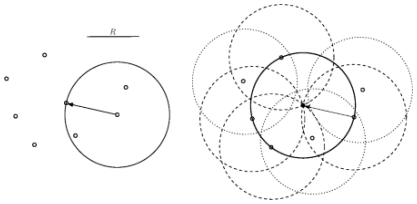


Figure 2.2. Searching Circle of known radius

This equation has an issue as it cannot be used to define the lines that are parallel to Ox axis, so the Hough transform uses another common parametric equation of a line by a point and an inclination angle:

$$x \cos \Theta + y \sin \Theta = \rho,$$

where

- ρ is the radius vector drawn from the origin to the line;
- Θ is the inclination angle of the radius vector.

Let the straight line in the Cartesian coordinate system be given by the above equation, from which it is easy to calculate the radius vector ρ and angle Θ. Then we can define the Hough parameter space with the coordinated ρ and Θ. As a result, each line of the source Cartesian space will be represented by a single point with the coordinates (ρ_0, Θ_0) in the Hough parameter space, see Fig. 2.3.

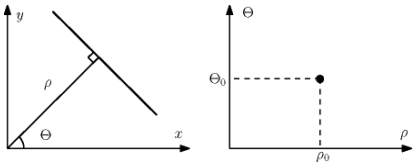


Figure 2.3. Representation of a straight line in Hough space

During the Hough transform all lines that pass via the points in the source Cartesian space are converted to the points in the Hough parameter space and vote for the corresponding lines. As a result,

the parameter space sums up the upvotes given for various lines. In general, the parameter space is continuous, however, in a digital form, it is stored as a matrix. Therefore, in the Hough transform the Hough space is called *accumulator* and is a matrix $A(\rho, \Theta)$ that stores voting information.

An infinite number of straight lines can be drawn passing through any of the points in the source Cartesian coordinate space, and they will generate a sinusoidal response function in the parameter space. Thus, any two sinusoidal response functions in the parameter space will intersect at the point (ρ, Θ) only if the points originating from them in the source Cartesian space lie on a single straight line, see Fig. 2.4. Based on this, we can conclude that to find straight lines in the source space, it is necessary to find all the local maxima of the accumulator matrix.

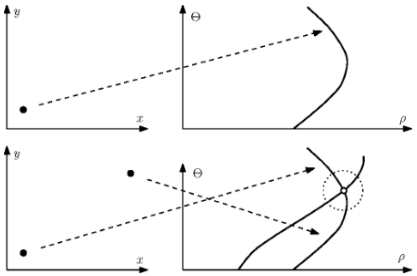


Figure 2.4. Voting in the Hough parameter space

The line search algorithm considered can be used in the same way to search for any other curve described in space by some function with a certain number of parameters $F = (a_1, a_2, \ldots, a_n, x, y)$, which will only affect the dimension of the parameter space.

Let us use the Hough transform to search for the circles of a given radius R. It is known that a circle on a plane is described by the formula:

$$(x - x_0)^2 + (y - y_0)^2 = R^2.$$

The set of the centers of all possible circles of radius R passing through a feature point forms a circle of radius R around that point. Hence, the response function in the Hough transform for finding circles

is a circle of the same size centered at the voting point, see Fig. 2. Then, similarly to the previous case, it is necessary to find the local maxima of the accumulator function $A(x, y)$ in the space of parameters (x, y), which will be the centers of the required circles.

The Hough transform is invariant to shift, scaling, and rotation. Considering that under projective transformations of a three-dimensional space, straight lines always go only to straight lines (in the degenerate case, to points), the Hough transform makes it possible to detect lines invariably not only to affine transformations of the plane but also to the group of projective transformations in space.

2.2 Task 1. Search for Lines

2.2.1 Hough Transform for Lines with OpenCV

OpenCV library provides two implementations for the Hough transform algorithm for search of the straight lines:

- `cv2.HoughLines(image, rho, theta, threshold)` $\rightarrow$ `lines` function performs the classic Hough line transform of an `image` and searches for straight lines. The `rho` and `theta` parameters define the subdivision of Hough parameter space for the corresponding axis. The `threshold` parameter defines the threshold, i.e., the number of votes that a line should get to be added to the returned `lines` array. Each line in returned `lines` array is an infinite line which is defined by `rho` (ρ) and `theta` (Θ) parameters.

- `cv2.HoughLinesP(image, rho, theta, threshold, minLineLength, maxLineGap)` $\rightarrow$ `lines` function performs the probabilistic Hough line transform of an `image` and searches for straight lines. The main function parameters are the same, however, two extra are added: these are `minLineLength` for a minimum line length, so no line shorter than this won't be selected, and `maxLineGap` for a maximum gap between two segments of the line to consider them to be the same line instead of two separate ones. The returned 2-dimensional `lines` array contains the start and end points of each of the found line segments as arrays of four: $[x_1, y_1, x_2, y_2]$.

2.2.2 Classic Hough Transform for Lines with OpenCV

To execute the Hough line, the transform in OpenCV first must preprocess an image to get the edges by executing the Canny algorithm.

```
# Read an image from a file in BGR
fn = "images/barcode.png"
I1 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I1, np.ndarray) or I1.data
    == None:
     print("Error reading file \"{}\"".format(
    fn))
     exit()

# Preprocess with Canny algorithm
I1edge = cv2.Canny(I1, 50, 200, None, 3)
# Display it
ShowImages([("Source image", I1),
            ("Edges", I1edge)], 2)
```

The result you can see 2.5 and 2.6

Figure 2.5. Source Image

Figure 2.6. Edges

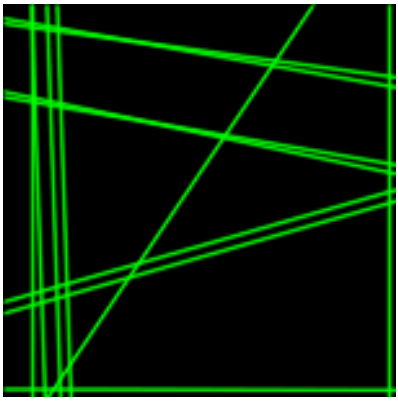
Then run the Hough line transform to get line parameters.

```
# Find lines with Hough transform
```

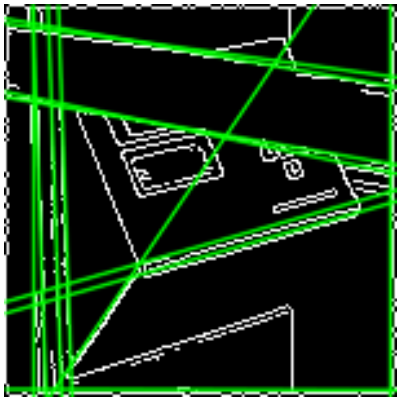
```
# Distance resolution: 1 pix
# Angle resolution: 1 deg in rad
# Accumulator threshold: 100
I1classic = cv2.HoughLines(I1edge, 1, np.pi /
    180, 100)
```

Finally, we can draw infinite lines by returned parameters with the help of the `cv2.line()` function.

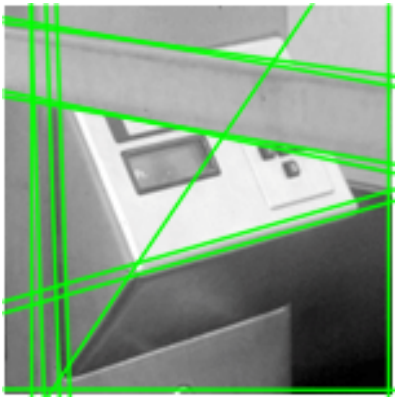
```
# Create output images
I1classic_out = I1.copy()
I1classic_lines = np.zeros_like(I1)
h, w = I1.shape[:2]
l = math.sqrt(h ** 2 + w ** 2)

# Go through all found lines and display them
if I1classic is not None:
    for i in range(0, len(I1classic)):
        rho = I1classic[i][0][0]
        theta = I1classic[i][0][1]
        a, b = math.cos(theta), math.sin(theta
    )
        x0, y0 = a * rho, b * rho
        pt1 = np.int32((x0 - l * b, y0 + l * a
    ))
        pt2 = np.int32((x0 + l * b, y0 - l * a
    ))
        cv2.line(I1classic_out, pt1, pt2, (0,
    255, 0), 1, cv2.LINE_AA)
        cv2.line(I1classic_lines, pt1, pt2,
    (0, 255, 0), 1, cv2.LINE_AA)

# Display it
print("Found {} lines".format(len(I1classic)))
ShowImages([("OpenCV lines", I1classic_lines),
            ("OpenCV lines highlighted",
    I1classic_out)], 2)
```



(a) Open CV Lines



(b) Open CV Lines on Edges



(c) Open CV Lines Highlighted

Figure 2.7. Classic CV Lines

2.2.3 Probabilistic Hough Transform for Lines with OpenCV

The probabilistic Hough line transform is executed similarly:

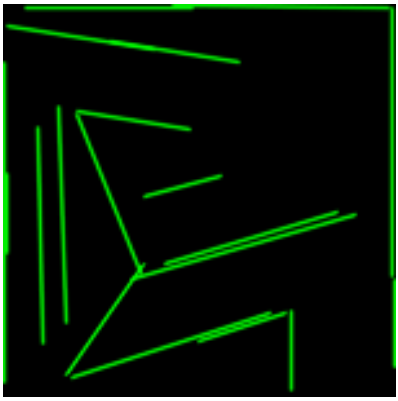
```
# Find lines with Probabilistic Hough
    transform
# Distance resolution: 1 pix
```

```
# Angle resolution: 1 deg in rad
# Accumulator threshold: 50
# Minimum line length: 50
# Maximum line gap: 4
I1prob = cv2.HoughLinesP(I1edge, 1, np.pi /
    180, 50, None, 50, 4)
```

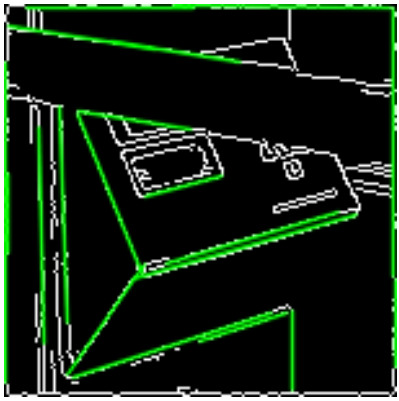
The probabilistic Hough transform will return an array of pairs of the endpoints of the found line segments, which we can use to draw them on top of the source image:

```
# Create and output image
I1prob_out = I1.copy()
I1prob_lines = np.zeros_like(I1)

# Go through all found lines and display them
if I1prob is not None:
    min_line = max_line = I1prob[0][0]
    min_line_len = max_line_len = math.sqrt((
    min_line[0] - min_line[2])**2 + (min_line
    [1] - min_line[3])**2)
    for i in range(1, len(I1prob)):
        l = I1prob[i][0]
        line_len = math.sqrt((l[0] - l[2])**2
    + (l[1] - l[3])**2)
        if line_len < min_line_len:
            min_line_len = line_len
            min_line = l
        if line_len > max_line_len:
            max_line_len = line_len
            max_line = l
        cv2.line(I1prob_out, (l[0], l[1]), (l
    [2], l[3]), (0, 255, 0), 1, cv2.LINE_AA)
        cv2.line(I1prob_lines, (l[0], l[1]), (
    l[2], l[3]), (0, 255, 0), 1, cv2.LINE_AA)

# Draw min and max lines (highlighted)
```

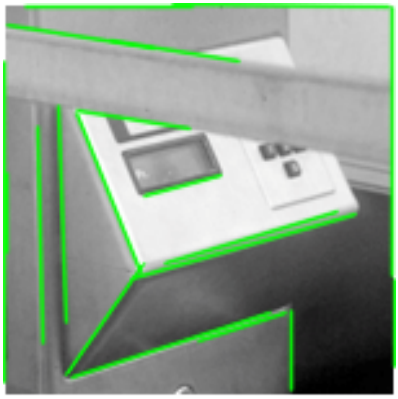
```
cv2.line(I1prob_out, (min_line[0], min_line[1]), (min_line[2], min_line[3]), (0, 0, 255), 1, cv2.LINE_AA)
cv2.line(I1prob_lines, (min_line[0], min_line[1]), (min_line[2], min_line[3]), (0, 0, 255), 1, cv2.LINE_AA)
cv2.line(I1prob_out, (max_line[0], max_line[1]), (max_line[2], max_line[3]), (255, 0, 0), 1, cv2.LINE_AA)
cv2.line(I1prob_lines, (max_line[0], max_line[1]), (max_line[2], max_line[3]), (255, 0, 0), 1, cv2.LINE_AA)

# Display it
print("Found {} lines".format(len(I1prob)))
print("The shortest found line length is {}".format(min_line_len))
print("The shortest found line is ({}, {}) - ({}, {})".format(min_line[0], min_line[1], min_line[2], min_line[3]))
print("The longest found line length is {}".format(max_line_len))
print("The longest found line is ({}, {}) - ({}, {})".format(max_line[0], max_line[1], max_line[2], max_line[3]))
ShowImages([("OpenCV lines", I1prob_lines),
            ("OpenCV lines highlighted", I1prob_out)], 2)
```



(a) Probabilistic Lines



(b) Probabilistic Lines on Edges



(c) Probabilistic Lines Highlighted

Figure 2.8. Probabilistic Open CV Lines

2.2.4 Hough Transform for Lines with scikit-image

Unfortunately, OpenCV does not provide the functionality to obtain the Hough accumulator matrix. However, the `scikit-image` library also provides functions that we can use to perform the Hough transform,

and unlike to OpenCV, `scikit-image` executes it in several steps, thus allowing us to process and display the Hough parameter space.

Same as OpenCV, `scikit-image` provides two implementations for the Hough transform algorithm for search of the straight lines. These are classic and probabilistic approaches. The main difference between OpenCV and `scikit-image` libraries is that in the classic approach the Hough transform contains two steps:

1. Calculation of the Hough parameter space;
2. Search for peaks in the Hough parameter space to find lines.

The following functions for Hough transform for lines are provided in `scikit-image`:

- `skimage.transform.hough_line(image, thetas)` → `hspace, angles, distances` function performs the classic Hough line transform of an `image` and calculates the Hough accumulator matrix (parameter space). The `thetas` parameter defines an array of Θ angles in radians to subdivide the Hough parameter space (defaults to 180 values in the range $[-\pi/2, \pi/2]$). It returns a tuple with three items: `hspace` with the Hough transform accumulator matrix; `angles` with the array of Θ angles in radians which define the Hough parameter space; and `distances` with the array of ρ distances which define the Hough parameter space.

- `skimage.transform.hough_line_peaks(hspace, angles, distances, min_distance, min_angle, threshold, num_peaks)` → `accum, theta, rho` function performs the search for peak values in the Hough parameter space acquired at the previous step. The input parameters list contains the data acquired from `skimage.transform.hough_line()` function with additional parameters for minimum distance and angle between two lines to consider them separate lines (`min_distance` and `min_angle` parameters, defined in the accumulator matrix dimensions), the minimum peak value `threshold` to choose (defaults to the half of the maximum `hspace` value) and the maximum number of peaks to search for (`num_peaks` parameter). It returns a tuple of three items (`accum`, `theta`, `rho`) corresponding to peak values in the Hough accumulator matrix. Each line

in the returned arrays is an infinite line which is defined by a pair of `rho` (ρ) and `theta` (Θ) parameters with the same array indices that scored the corresponding `accum` votes.

- `skimage.transform.probabilistic_hough_line(image, threshold, line_length, line_gap, theta)` → `lines` function performs the probabilistic Hough line transform of an `image` and searches for straight lines. The main function parameters are similar to the ones used in Opencv2. They are the `threshold` for the minimum required peak value, the minimum line length to consider (`line_length` parameter), the maximum gap between two segments of the same line (`line_gap` parameter), and an array `theta` of Θ angles to use during transformation. The returned 3-dimensional `lines` array contains the start and end points of each of the found line segments as 2D arrays with four elements: $[[x_1, y_1], [x_2, y_2]]$.

2.2.5 Classic Hough Transform for Lines with scikit-image

Similar to the OpenCV library, to execute the Hough line transform in scikit-image first we have to preprocess an image to get edges by executing the Canny algorithm.

```
# Read an image from file in BGR
fn = "images/barcode.png"
I1 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I1, np.ndarray) or I1.data
    == None:
     print("Error reading file \"{}\"".format(
    fn))
     exit()

# Preprocess with Canny algorithm
I1edge = cv2.Canny(I1, 50, 200, None, 3)
# Display it
ShowImages([("Source image", I1),
            ("Edges", I1edge)], 2)
```

Figure 2.9. Source Image

Figure 2.10. Edges

Then run the Hough line transform to get the Hough parameter space accumulator matrix. Since now we have the accumulator matrix we can display it as well to see how the parameter space is formed.

```
# Do Hough transform
# We will use 360 angles instead of the default 180
thetas = np.linspace(-np.pi / 2, np.pi / 2, 360, endpoint=False)
I1h, angles, distances = skimage.transform.hough_line(I1edge, theta=thetas)

# Show Hough parameter space
# We will resize it to make square for a better display
ShowImages([("Parameter space",
             cv2.resize(I1h.astype(np.float32) / np.max(I1h),
                        (I1h.shape[1], I1h.shape[1])))])
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [0.0..1.6271064].

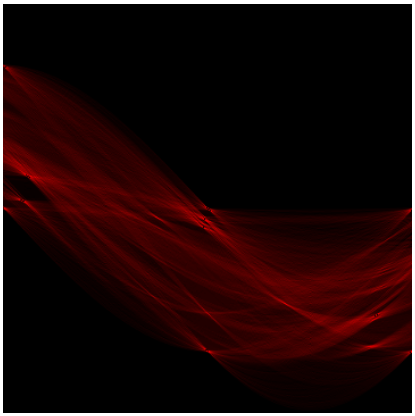


Figure 2.11. Voting in the Hough parameter space

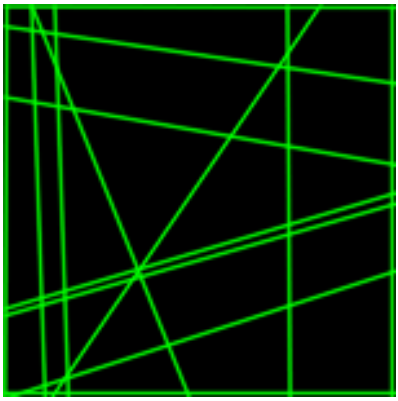
Now we can search for the maxima values in the Hough parameter space.

```
# Find lines with Hough transform
# Minimum distance between lines: 0
# Minimum angle between lines: 0
I1accum, I1theta, I1rho = skimage.transform.
    hough_line_peaks(
     I1h, angles, distances, min_distance=0,
    min_angle=0
)
```

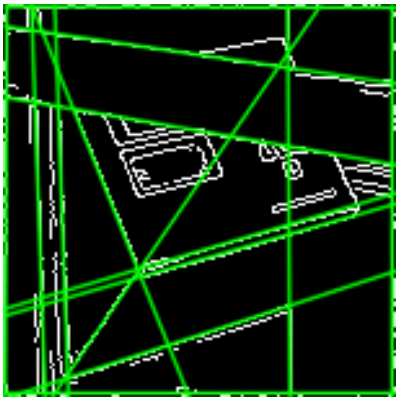
Finally, we can draw the infinite lines found by the scikit-image Hough transform implementation with subsequent calls to the `cv2.line()` function.

```
# Create output images
I1classic_out = I1.copy()
I1classic_lines = np.zeros_like(I1)
h, w = I1.shape[:2]
l = math.sqrt(h ** 2 + w ** 2)

# Go through all found lines and display them
if I1theta is not None and I1rho is not None:
```

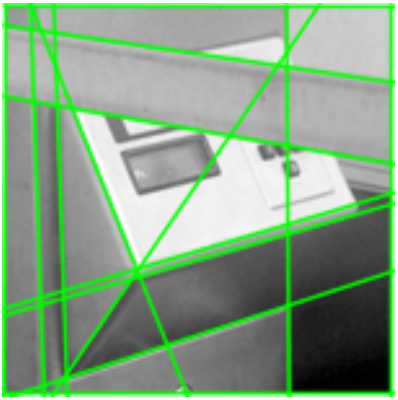
```
    for i in range(0, len(I1theta)):
        a, b = math.cos(I1theta[i]), math.sin(I1theta[i])
        x0, y0 = a * I1rho[i], b * I1rho[i]
        pt1 = np.int32((x0 - l * b, y0 + l * a))
        pt2 = np.int32((x0 + l * b, y0 - l * a))
        cv2.line(I1classic_out, pt1, pt2, (0, 255, 0), 1, cv2.LINE_AA)
        cv2.line(I1classic_lines, pt1, pt2, (0, 255, 0), 1, cv2.LINE_AA)

# Display it
print("Found {} lines".format(len(I1theta)))
ShowImages([("scikit-image Lines", I1classic_lines),
            ("SKImage Lines on the source", I1classic_out)], 2)
```



(a) Lines



(b) Lines on Edges



(c) Lines on the Source

Figure 2.12. Scikit-image Lines

2.2.6 Probabilistic Hough Transform for Lines with scikit-image

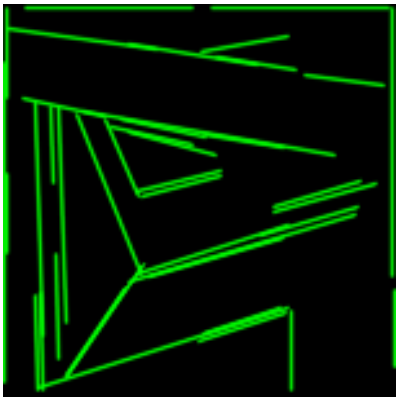
The probabilistic Hough line transform with scikit-image is executed similarly to Opencv2. The only two differences are that now you must use the

`skimage.transform.probabilistic_hough_line(image, threshold, line_length, line_gap, theta)` → `lines` function and returned lines parameters are in 3D array instead of 2D array in Opencv2. So, the line length calculation will be as follows:

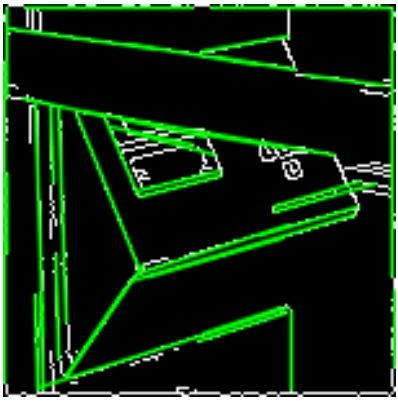
```
line = lines[i]
line_len = math.sqrt((line[0][0] - line[1][0])**2 + (line[0][1] - line[1][1])**2)
```

And the line drawing will be as follows:

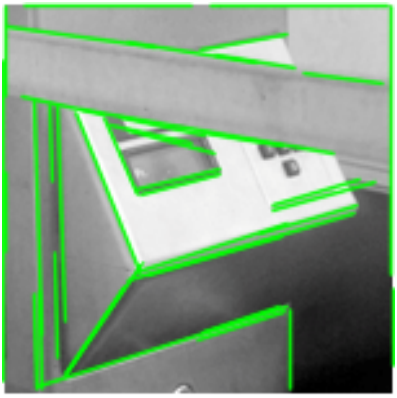
```
color = (0, 0, 255)  # Color in OpenCV is defined in BGR, so this will be the Red color
cv2.line(Iout, lines[i][0], lines[i][1], color, 1, cv2.LINE_AA)
```



(a) Probabilistic Lines



(b) Probabilistic Lines on Edges



(c) Probabilistic Lines on the Source

Figure 2.13. Probabilistic Scikit-image Lines

2.3 Task 2. Search for Circles

When searching for circles, the Hough parameter space becomes 3-dimensional. The third dimension here is added for the circle radius. Since the circle radius can be of an infinite range, it is required to

define this range when executing the Hough transform operations to create the correct accumulator space 3D matrix.

2.3.1 Hough Transform for Circles with OpenCV

The search for circles with OpenCV is rather straightforward. You don't need to do any image preprocessing as the Canny algorithm is built into a Hough transform implementation. To run Hough transform for circles with OpenCV you have to use the following function:

- `cv2.HoughCircles(image, method, dp, minDist, param1, param2, minRadius, maxRadius)` → `circles` function is used to execute the Hough transform for circles on a `grayscale` image. It returns a 2-dimensional array of circles where each circle has three parameters: (x_0, y_0, R) — for the center of the circle and its radius.

The Hough transform for circles is parameterized in the following way:

- `method` parameter defines one of two possible methods for this transformation; it can be either `cv2.HOUGH_GRADIENT` or `cv2.HOUGH_GRADIENT_ALT`. These methods are almost similar, however the second one should give a bit better accuracy.
- `dp` parameter defines the scale ratio of the accumulator matrix. The resolution of the accumulator matrix is `dp` times smaller than the resolution of the source image. In most cases, 1.5 would fit well.
- `minDist` parameter defines the minimum distance between circle centers to be detected.
- `param1` is a high threshold that is passed to the Canny edge detector, while the low threshold would be equal to half of `param1`.
- `param2` parameter defines the threshold for the circle detection. In the case of the `cv2.HOUGH_GRADIENT` method, it is the accumulator matrix value, while for the `cv2.HOUGH_GRADIENT_ALT` method it is the circle quality measure defined in [0,1] range where 1 is a perfect circle.

- `minRadius` and `maxRadius` parameters define the circle radius search range. This will define the third dimension of the Hough parameter space accumulator matrix range.

To display the found circles the `cv2.circle(image, center, radius, color, thickness, lineType)` function may be used to draw a circle on the `image` with the given `center` point, `radius`, and `thickness`. The `lineType` parameter is used to define the type of the line, which can be set to `cv2.LINE_AA` for an antialiased line.

For example, drawing a circle on an image `Iout` with center $(50, 50)$ and radius 10 and thickness 1 with blue color would be written as follows:

```
1 cv2.circle(Iout, (50, 50), 10, (255, 0, 0), 1)
```

2.3.2 Hough Transform for Circles with scikit-image

When implementing the Hough transform for circles with `scikit-image` library you must do the same two steps as you did for lines. These are:

1. calculation of the Hough parameter space;
2. search for peaks in the Hough parameter space to find circles.

Since the Hough transform is split into two steps it means that with `scikit-image` we can check the Hough parameter space as well.

The following functions for Hough transform for circles are provided in `scikit-image`:

- `skimage.transform.hough_circle(image, radius, normalize, full_output)` → `hspaces` function performs the Hough circle transform of an `image` and calculates the 3-dimensional Hough accumulator matrix. The `radius` parameter is either a scalar to search for a single radius or an array of radii. Two optional parameters allow to `normalize` the output accumulator spaces by the number of pixels required to draw a circle to make different radii circles have the same weight in the accumulator (True by default) and to increase the size of the accumulator by two maximum radii to be able to get

`full_output` and find circles with centers at outside of the image (False by default). It returns a 3D accumulator Hough parameter space matrix (an array of 2D Hough parameter spaces for each of the radii from the `radius` list).

- `skimage.transform.hough_circle_peaks(hspaces, radii, min_xdistance, min_ydistance, threshold, num_peaks, total_num_peaks, normalize)` → `accum, cx, cy, rad` function performs the search for peak values in the 3D Hough parameter space `hspaces` acquired at the previous step with an array of `radii`. Optional parameters allow specifying the minimum distances between picked circle centers (`min_xdistance` along Ox and `min_ydistance` along Oy), the `threshold` parameter for peaks (half of the maximum `hspaces` value by default), the maximum number of peaks selected in a single layer Hough parameter space and all spaces total (`num_peaks` and `total_num_peaks` parameters correspondingly). Also, it's possible to `normalize` peaks by the radius as otherwise the peaks of the higher radius would be preferred over the smaller radius (default behavior).

It should be noted that `scikit-image` also provides a function for the Hough transform for ellipses which is out of the current practical assignment scope. If you are interested, you may try it as well. The arguments are similar to the one used for circles; however, it also allows searching for ellipses with the specified parameters range and returns an array of tuples with found ellipses:

```
skimage.transform.hough_ellipse(image,
    threshold=4, accuracy=1, min_size=4,
    max_size=None) -> accum, yc, xc, a, b,
    orientation)
```

Let us take an image with coins and try finding them by the Hough transform. Since `scikit-image` libraries work with black-and-white images, we have to preprocess it with Canny algorithm to find edges first.

```
# Read an image from file in BGR
fn = "images/coins.jpg"
```

```
I2 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I2, np.ndarray) or I2.data == None:
    print("Error reading file \"{}\"".format(fn))
    exit()

# Preprocess with Canny algorithm
I2edge = cv2.Canny(I2, 200, 250)

# Display it
ShowImages([("Source image", I2),
            ("Edges", I2edge)], 2)
```

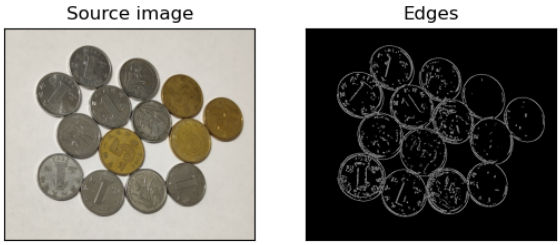


Figure 2.14. Source Image and its Edges

Now we can run the Hough circles transform. Since we know that we are searching for circles in the range from 40 to 60 pixels, so we will define it as our `radii` range with step 5.

```
# Find circles with Hough transform
# Detect circles with radii in range from 40 to 60 with step 5
radii = np.arange(40, 61, 5)
I2h = skimage.transform.hough_circle(I2edge, radii)

# Show Hough parameter space
```

```
# We will resize it to make square for a better display
spaces = []
for i in range(len(radii)):
    spaces.append(("Radius {}".format(radii[i]),
                   cv2.resize(I2h[i].astype(np.float32) / np.max(I2h[i]),
                              (I2h[i].shape[1], I2h[i].shape[1]))))
ShowImages(spaces, 3)
```

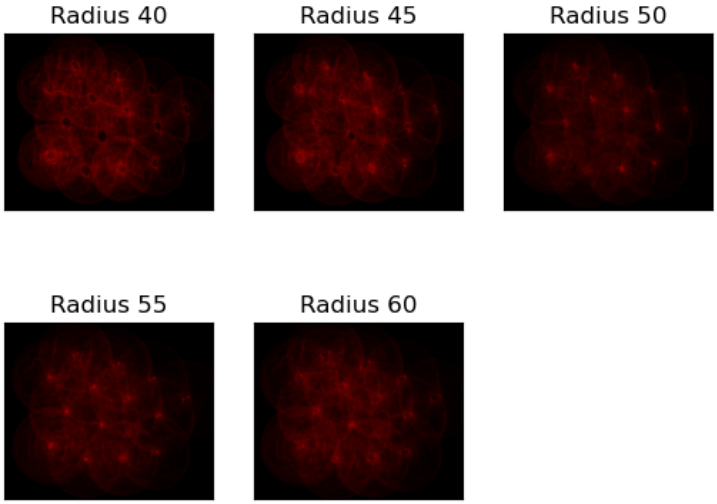


Figure 2.15. Detecting circles with different radii

Now we can search for the maxima values in the Hough parameter spaces and draw the found circles with subsequent calls to the `cv2.circle()` function.

```
# Select the most prominent 10 circles
# Minimum distance between centers along X: 30
# Minimum distance between centers along Y: 30
```

```
I2accum, I2cx, I2cy, I2radii = skimage.transform.hough_circle_peaks(
    I2h, radii, min_distance=45, min_distance=45, total_num_peaks=14
)

# Create output images
I2out = I2.copy()
I2circles = np.zeros_like(I2)

# Go through all found circles and display them
if I2cx is not None and I2cy is not None and I2radii is not None:
    for i in range(0, len(I2cx)):
        center = np.int32((I2cx[i], I2cy[i]))
        radius = int(I2radii[i])
        cv2.circle(I2out, center, 1, (0, 255, 0), 1)           # center point
        cv2.circle(I2out, center, radius, (0, 255, 0), 1)      # circle outline
        cv2.circle(I2circles, center, 1, (0, 255, 0), 1)
        cv2.circle(I2circles, center, radius, (0, 255, 0), 1)

# Display it
print("Found {} circles".format(len(I2cx)))
ShowImages([("scikit-image Circles", I2circles),
            ("scikit-image Circles on the source", I2out)], 2)
```

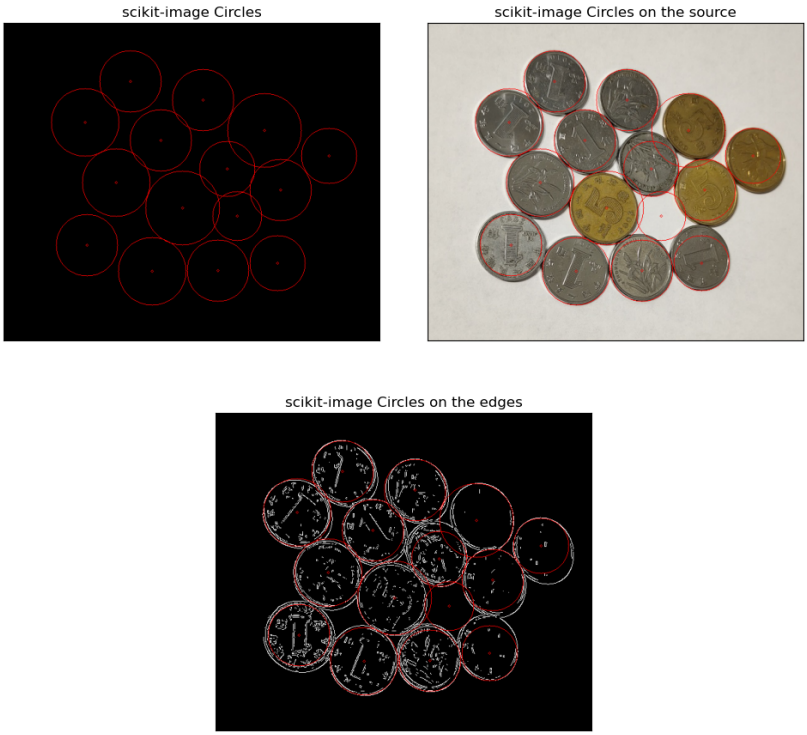


Figure 2.16. Scikit-image Cicrles matching with Source Image and Edges

2.4 Procedure of Practical Assignment Performing

All tasks are based on the material presented in the corresponding sections of the assignment. Students are required to write code for each task using Python, OpenCV, and NumPy.

Task 1. Search for Lines

1. **Classic Hough Transform for Lines with OpenCV**
   Take three arbitrary images containing lines and execute the clas-

sic Hough transform with OpenCV. Display the lines found and printtheir number.

Hints for implementation:

- Preprocess images with Canny edge detector: `cv2.Canny(image, 50, 200, None, 3)`.
- Apply classic Hough transform: `cv2.HoughLines(edge_image, rho, theta, threshold)`.
  - `rho = 1` (distance resolution in pixels);
  - `theta = np.pi / 180` (angle resolution in radians, equivalent to 1 degree);
  - `threshold = 100` (minimum votes required to detect a line).
- Draw infinite lines using the returned (ρ, θ) parameters:
  - Calculate `a = math.cos(theta)`, `b = math.sin(theta)`;
  - Find point `(x0, y0) = (a * rho, b * rho)`;
  - Calculate two points for drawing: `(x0 ± l * b, y0 ∓ l * a)`, where `l = sqrt(h^2 + w^2)`;
  - Use `cv2.line()` to draw the line.
- **Note:** You may have to change the Hough transform parameters (threshold, rho, theta) to match your specific images.

2. **Probabilistic Hough Transform for Lines with OpenCV** Take three arbitrary images and execute the probabilistic Hough transform with OpenCV. Display the found lines, print their number, and print the shortest and longest line parameters.

   Hints for implementation:

   - Apply probabilistic Hough transform: `cv2.HoughLinesP(edge_image, rho, theta, threshold, minLineLength, maxLineGap)`.
     - `rho = 1` (distance resolution);
     - `theta = np.pi / 180` (angle resolution);
     - `threshold = 50` (accumulator threshold);

  - `minLineLength = 50` (minimum line length);
  - `maxLineGap = 4` (maximum gap between line segments).
- The function returns an array of line segments: each line is represented as `[x1, y1, x2, y2]`.
- Calculate line length: `math.sqrt((x2 - x1)^2 + (y2 - y1)^2)`.
- Iterate through all found lines to find the shortest and longest ones.
- Draw all lines on the output image using `cv2.line()`.
- **Note:** Adjust parameters (threshold, minLineLength, maxLineGap) based on your images.

3. **Classic Hough Transform for Lines with scikit-image**
Take three arbitrary images and execute the classic Hough transform with the scikit-image library. Display the Hough parameter space and found lines. Print the found lines and their number.

Hints for implementation:

- Preprocess with Canny edge detector: `cv2.Canny(I, 50, 200, None, 3)`.
- Define theta angles: `thetas = np.linspace(-np.pi/2, np.pi/2, 360, endpoint=False)`.
- Calculate Hough parameter space: `hspace, angles, distances = skimage.transform.hough_line(edge_image, thetas)`.
- Display the parameter space: use `cv2.resize()` to make it square for better visualization.
- Find peaks (lines): `accum, theta, rho = skimage.transform.hough_line_peaks(hspace, angles, distances, min_distance=0, min_angle=0)`.
- Draw infinite lines similarly to the OpenCV classic approach using the returned (ρ, θ) pairs.
- **Note:** The scikit-image implementation separates accumulator calculation from peak detection, allowing visualization of the Hough space.

4. **Probabilistic Hough Transform for Lines with scikit-image**
   Take three arbitrary images and execute the probabilistic Hough transform with the scikit-image library. Display the lines found, type their number, and print the shortest and longest line parameters.

   Hints for implementation:

   - Apply probabilistic Hough transform: `lines = skimage.transform.probabilistic_hough_line( edge_image, threshold, line_length, line_gap, theta)`.
     - `threshold` - minimum peak value;
     - `line_length` - minimum line length;
     - `line_gap` - maximum gap between segments;
     - `theta` - array of angles to use.
   - The returned `lines` array is 3-dimensional: each line is represented as `[[x1, y1], [x2, y2]]`.
   - Calculate line length: `math.sqrt((line[0][0] - line[1][0])^2 + (line[0][1] - line[1][1])^2)`.
   - Draw lines using `cv2.line(Iout, lines[i][0], lines[i][1], color, 1, cv2.LINE_AA)`.
   - Find and display the shortest and longest lines.
   - **Note:** Unlike OpenCV, scikit-image returns lines in a 3D array format — pay attention to indexing.

Task 2. Search for Circles

1. **Hough Transform for Circles with OpenCV**
   Take three arbitrary images containing circles. Search for circles of a known radius range using the Hough transform with OpenCV. Plot the found circles on the original image and print the number of found circles.

   Hints for implementation:

   - No explicit preprocessing needed — Canny algorithm is built into the HoughCircles function.

- Use grayscale image as input.
- Apply Hough circle transform: `cv2.HoughCircles(image, method, dp, minDist, param1, param2, minRadius, maxRadius)`.
  - `method = cv2.HOUGH_GRADIENT` or `cv2.HOUGH_GRADIENT_ALT`;
  - `dp = 1.5` (inverse ratio of accumulator resolution to image resolution);
  - `minDist` - minimum distance between circle centers;
  - `param1 = 100` (high threshold for Canny edge detector);
  - `param2 = 30` (accumulator threshold for circle detection);
  - `minRadius, maxRadius` - radius search range.
- The function returns circles as `(x0, y0, R)` for each detected circle.
- Draw circles using `cv2.circle(image, center, radius, color, thickness, cv2.LINE_AA)`.
- Also draw the center point (radius = 1) for better visualization.
- **Note:** The `param2` parameter interpretation differs between `HOUGH_GRADIENT` (accumulator value) and `HOUGH_GRADIENT_ALT` (circle quality in [0,1] range).

2. **Hough Transform for Circles with scikit-image**
   Take three arbitrary images and execute the Hough transform for circles with the scikit-image library. Display the circles found and print their number.

   Hints for implementation:

   - Preprocess with Canny edge detector: `cv2.Canny(I, 200, 250)`.
   - Define radius range: `radii = np.arange(min_radius, max_radius + 1, step)`.
   - Calculate Hough parameter space: `hspaces = skimage.transform.hough_circle(edge_image, radii)`.

  - Returns a 3D accumulator matrix (array of 2D Hough spaces for each radius).
- Display individual Hough spaces for each radius (resize for better visualization).
- Find peaks (circles): `accum, cx, cy, rad = skimage.transform.hough_circle_peaks(hspaces, radii, min_{xdistance}, min_{ydistance}, threshold, num_peaks, total_num_peaks, normalize=True)`.
- Draw circles using `cv2.circle()` for both center points and circles.
- **Note:** The normalize parameter ensures circles of different radii have equal weight in peak selection.

Task 3. Optional: Self-implementation of Classic Hough Transform

1. **Manual implementation of classic Hough transform for lines**
   Implement the classic Hough transform algorithms for lines yourself. Compare your implementation results with those obtained in the first parts of the assignment. Highlight the selected points in the Hough parameter space.

   Hints for implementation:

   - Preprocess with Canny edge detector to filter edge points.
   - Allocate accumulator matrix manually: `accumulator = np.zeros((num_rho, num_theta), dtype=np.int32)`.
   - Implement the classic algorithm:

```
For each point (x, y) in edge image:
  For theta = 0 to 180:
    rho = x * cos(theta) + y * sin(theta)
    H[theta, rho] = H[theta, rho] + 1
  end
end
```

- Define theta range: `thetas = np.deg2rad(np.arange(0, 180))`.
- Calculate rho range: `rho_max = int(np.sqrt(w^2 + h^2))`; `rho_range = np.arange(-rho_max, rho_max + 1)`.
- For each edge point, compute rho for all theta and increment corresponding accumulator cell.
- Select maxima using thresholding method (from previous classes).
- Draw the detected lines on the original image.
- Highlight the corresponding peak points in the Hough parameter space visualization.
- **Note:** Python implementation of nested loops will be slow — consider using low-resolution input images or optimize with vectorized operations.

General Requirements

- All results must be visualized using the `ShowImages()` function from the `pa_utils.py` module.
- Code must be well-structured and commented.
- Each task must be implemented in a separate Jupyter notebook cell or section.
- Answers the questions and write a conclusion.

Questions to Practical Assignment Report Defense

- Can the Hough transform be used to find arbitrary contours that cannot be described analytically?
- What are the recurrent and generalized Hough transforms?
- What are the other ways of line parametrization in the Hough transform?
- What is the main principle of the Hough transform?

3 Practical Assignment No 3. Feature Detectors

Objectives

- **Study the principles of feature point detection and description**
  To understand the fundamental concepts of what makes a point "feature-worthy" (corners, blobs) and to analyze the mathematical foundations of scale and rotation invariance using SIFT and ORB algorithms as primary examples.

- **Master feature detection tools in OpenCV**
  To gain practical skills in using OpenCV functions for detecting keypoints and computing their descriptors (SIFT, ORB), and to visualize detection results with different levels of detail (position, scale, orientation).

- **Conduct experimental research on feature matching**
  To perform feature point matching between object and scene images using various matchers (brute force, FLANN), apply filtering techniques (cross-checking, Lowe's ratio test), and evaluate matching quality under different conditions.

- **Apply geometric verification for object detection**
  To implement homography estimation using the RANSAC algorithm for outlier removal, accurately localize objects within complex scenes, and optionally extend the acquired information to panoramic image stitching.

Preface

To successfully complete this practical assignment, students will need the following:

- **Python 3.x** with core scientific libraries:
  - `opencv` (`opencv-python` pip library) — for feature point detection (SIFT, ORB), descriptor computation, matching, and homography estimation;

- `numpy` (<=1.26.4) — for efficient array operations, coordinate transformations, and descriptor manipulations;
- `matplotlib` — for additional visualization of feature matches and geometric transformations;
- `math` — for trigonometric calculations and distance computations.

Sample installation command: `pip install opencv-python, numpy==1.26.4, matplotlib`

- **Development environment:** It is recommended to use `Jupyter Notebook` or `Jupyter Lab` (locally or via Google Colab). All tasks are designed to be completed interactively in a `.ipynb` format, allowing step-by-step exploration of feature detection and matching results. Alternatively, any Python IDE with support for interactive execution (e.g., VS Code with Python extension) is acceptable.
- **Test images:** The assignment requires two types of image pairs for feature matching experiments:
  - **Object-scene pairs** — images containing a specific object (e.g., a book, building fragment) and a wider scene containing that object, used for object detection tasks;
  - **Panorama sequences (optional)** — three or more overlapping images of the same scene taken from different viewpoints, used for image stitching experiments.

  Sample images are provided in the `images/` / directory, but students are encouraged to capture their own images to test algorithm performance under different conditions (lighting or viewpoint changes, scale variations).
- **Utility module:** The file `pa_utils.py` is provided alongside the assignment. This module contains auxiliary functions essential for the work:
  - `ShowImages()` — displays multiple images side-by-side with titles, used throughout all tasks for visualizing keypoints, matches, and detection results;

– `exit()` — provides clean cell termination within Jupyter without stopping the kernel.

The module is imported at the very beginning of the notebook and is used throughout all tasks to display intermediate and final results.

3.1 Preliminary

3.1.1 Feature Points

First, we have to understand what are the image feature points. Let us look at Fig. 3.1.

Figure 3.1. Building image

As you can see, it has 6 patches (named by letters from A to F)). If you try searching for these patches on the source image you will find out that it is not possible to locate the position of patches A and B since they are taken somewhere from a repeating pattern of the sky or the building wall. If you look at patches C, D and D you would also face a problem locating them since they are somewhere at the edge of a building. However, if you take patches E or F you would easily locate them since they are corners. Such types of points that are easy to locate on an image are called *feature points*. From a formal point of view, feature points can be defined as points significantly different

from their neighborhood. So, if you move a sliding window by one pixel from a feature point you would get a completely different image, see Fig. 3.2.

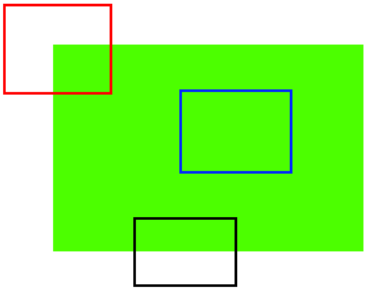


Figure 3.2. Sliding Window

There are a lot of algorithms designed to detect and describe feature points, including:

- Harris corner detector [13];
- Shi-Tomasi corner detector [14];
- Scale-Invariant Feature Transform (SIFT) detector and descriptor [7];
- Speeded-Up Robust Features (SURF) detector and descriptor [15];
- Features from Accelerated Segment Test (FAST) detector [16];
- Binary Robust Independent Elementary Features (BRIEF) descriptor [9];
- Oriented FAST and Rotated BRIEF (ORB) detector and descriptor [8].

In the current practical assignment, we will use SIFT and ORB feature point detectors and descriptors for image matching.

3.1.2 SIFT Detector

SIFT stands for Scale-Invariant Feature Transform [7]. The algorithm was patented; however, in 2020 the patent expired, so now it can be used freely in any application.

One of the serious problems of traditional corner detectors, e.g., the Harris detector, is that they are not scale-invariant. Depending on the scale they may result in different feature points detected, see Fig. 3.3.

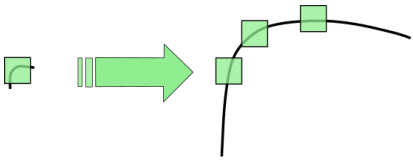


Figure 3.3. Dependence of the corner feature on the scale

To calculate the characteristic scale of feature points, the ideas of the Laplacian of Gaussian (LoG) method are used. It can be calculated as a scale-space maximum response of the Laplacian of Gaussian of an image with varying the σ value, which is calculated by convolution of the variable-scale Gaussian $G(x, y, \sigma)$ with an input image $I(x, y)$:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y),$$

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}},$$

where $*$ is a convolution operation in x and y.

To detect scale-space maxima efficiently the Difference of Gaussian (DoG) method was proposed, which is computed which is computed according to the following formula with a predefined constant multiplier k by simple image subtraction, see Fig. 3.4:

$$\begin{aligned} D(x, y, \sigma) &= \big(G(x, y, k\sigma) - G(x, y, \sigma)\big) * I(x, y) \\ &= L(x, y, k\sigma) - L(x, y, \sigma). \end{aligned}$$

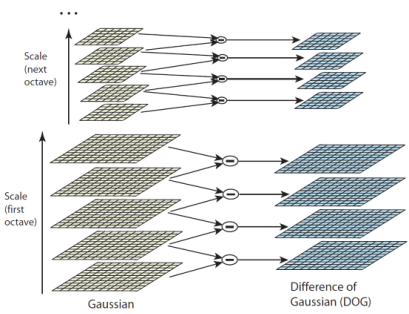


Figure 3.4. Difference of Gaussian calculation

The maxima of the DoG convolution for a pixel can be calculated by comparing a pixel with its 26 neighbors in current and adjacent scales as shown in Fig. 3.5.

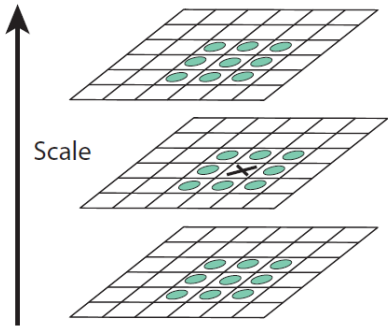


Figure 3.5. Selecting DoG maxima

After a point location and its characteristic scale are found, its location is adjusted according to the nearby image data. Low-contrast or poorly localized points are filtered out since they are highly sensitive to noise.

Next, the characteristic orientation of the neighbor feature point patch is estimated by calculating a histogram of gradients of the patch and selecting a histogram maximum value. In cases where several

strong maxima are detected, the feature point is considered as several points with different orientations. The histogram contains 36 bins and covers 360°, see Fig. 3.6.

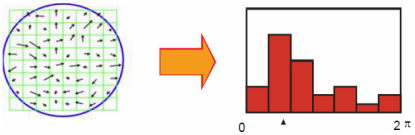


Figure 3.6. Selecting the characteristic orientation

Then the patch is rotated according to the characteristic orientation and a descriptor is built by computing 16 histograms for 4×4 subwindows of a 16×16 pixels window around the feature point, see Fig. 3.7.

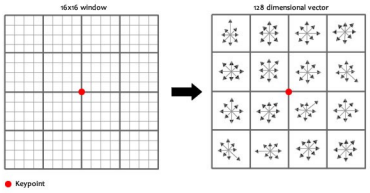


Figure 3.7. SIFT descriptor

Thus, the SIFT descriptor contains 16 histograms, and each histogram contains 8 bins, which gives a total 128-dimensional vector for a feature point descriptor.

An estimation of similarity for two SIFT feature point descriptors is done with the Euclidean distance, which is the standard L2 norm distance between two vector descriptors $\mathbf{a}$ and $\mathbf{b}$:

$$d_{\text{Euclidean}}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2},$$

where

- $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ is the first SIFT descriptor;

- $\mathbf{b} = [b_1, b_2, \ldots, b_n]$ is the second SIFT descriptor;
- a_i and b_i are the i-th components of $\mathbf{a}$ and $\mathbf{b}$, respectively;
- n is the descriptor length and is equal to 128 for the SIFT descriptor.

In practice, the squared Euclidean distance can be used to increase the computation efficiency:

$$d_{\text{SquaredEuclidean}}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{n} (a_i - b_i)^2.$$

3.1.3 ORB Detector

ORB detector [8] is a fusion of FAST feature point detector and BRIEF descriptor with many modifications to enhance the detector performance. First, it uses the FAST detector to find feature points, then applies the Harris corner measure to find the top N-points among them. Secondly, it uses a pyramid to produce multiscale features. Since the FAST feature point detector is not rotation invariant, the following method is used to calculate the characteristic rotation of the point: the intensity-weighted centroid of the patch with the corner located in the center is calculated. The direction of the vector from this corner point to the centroid is considered the orientation of the feature point. To improve the rotation invariance, moments are computed with x and y axes which must be in a circular region of radius r, where r is the size of the patch.

ORB uses BRIEF descriptors for its feature points. The BRIEF descriptor [9] is a bit string description of an image patch constructed from a set of binary intensity tests:

$$\tau(p, x, y) = \begin{cases} 1, & p(x) < p(y), \\ 0, & p(x) \geq p(y), \end{cases}$$

where $p(x)$ denotes the intensity at pixel location x.

Then the BRIEF feature point descriptor defined as a binary can be calculated from the set of simple binary intensity tests as follows:

$$f_n(p) = \sum_{i=1}^{n} 2^{i-1} \tau(p, x_i, y_i).$$

To improve the performance of the BRIEF descriptor for rotated features, the descriptor is rotated according to the orientation of the feature points. For any feature set of n binary tests at location (x_i, y_i), a $2 \times n$ matrix is defined, which contains the coordinates of these pixels:

$$S = \begin{pmatrix} x_1, & \dots, & x_n \\ y_1, & \dots, & y_n \end{pmatrix}.$$

Then using the orientation θ of a patch, its rotation matrix R_θ is calculated and used to rotate the S matrix to get a rotated version S_θ:

$$S_\theta = R_\theta \, S,$$

where

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

ORB quantizes the angle to increments of $2\pi/30 = 12^\circ$, so a lookup table of precomputed BRIEF patterns can be calculated for each possible angle. As long as the keypoint orientation θ is consistent across views, the correct set of points S_θ will be used to compute its descriptor:

$$g_n(p, \theta) = f_n(p) \mid (x_i, y_i) \in S_\theta.$$

To compute the distance between two ORB descriptors $\mathbf{a}$ and $\mathbf{b}$, a Hamming distance is used (the Hamming distance is the number of positions at which the corresponding elements differ):

$$d_{\text{Hamming}}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{n} \mathbf{1}(a_i \neq b_i),$$

where

- $\mathbf{a} = [a_1, a_2, \dots, a_n]$ is the first ORB descriptor;
- $\mathbf{b} = [b_1, b_2, \dots, b_n]$ is the second ORB descriptor;
- a_i and b_i are the i-th components of $\mathbf{a}$ and $\mathbf{b}$, respectively;
- $\mathbf{1}(a_i \neq b_i)$ is an indicator function that equals 1 if $a_i \neq b_i$, and 0 otherwise;

- n is the descriptor length and is equal to 256 bits for the ORB descriptor.

The multi-probe Locality-Sensitive Hashing (LSH) method may be used to accelerate ORB descriptor matching.

3.1.4 Feature Point Descriptors Matching

The simplest way to match two sets of feature points is to use a brute force method. In this case, for each point of the first set, an item of the second set is selected, which has the smallest distance. For SIFT descriptor you can use the Euclidean L_2 distance. As for the ORB descriptor, since it is a binary mask, the Hamming distance between the descriptors should be used instead.

The brute force method works slowly. To speed up the descriptor matching, an accelerating structure must be built on top of the descriptors set; then when matching a descriptor from the search query it is not compared with the whole set, but only with the descriptors from some cluster. The simplest accelerating structure is KD-tree (k-dimensional tree), which is built on the space of training set descriptors. If the descriptor is defined as a binary mask, then it is preferable to use the Locality-Sensitive Hashing (LSH) method.

Using the first best match may result in a lot of descriptor matches; however, many of them are false matches since some feature points are from repeating patterns (e.g., windows, water, clouds, etc.). There are two possible solutions to filter some of the not strong matches.

The first solution is to use *cross-checking*, which requires the descriptor to be matched in two directions: when matching two images it must be the best match in the forward and the backward directions.

The second solution is to use the *k-nearest matching* method. In this case, for each point, several best matches are found, sorted by the distance, and the match is considered to be *good* if it is significantly different from the next nearest match, so the distance from the first nearest match is significantly lower compared to the distance from the second nearest match:

$$D_1 < r \cdot D_2,$$

where

- D_1 is the distance to the first nearest match;

- D_2 is the distance to the second nearest match;
- r is the difference ratio, which is advised to be 0.75 by the SIFT method authors.

Please note that the k-nearest matching method is not compatible with cross-checking since the cross-check does not allow to find more than one match.

Using accelerating structures and k-nearest filtering we can get a set of strong matches between images. When matching the descriptors, we did not take the feature point positions into account, so the next step would be to calculate the geometric transformation between the images considering that there may still be a lot of outliers or false matches. The most commonly used solution is the Random Sequence Consensus (RANSAC) method. The general idea of the method is to estimate not all data, but only a small sample, then build a hypothesis based on this sample and check whether this hypothesis is correct. After checking a number of such hypotheses, we choose one that best fits most of the data.

1. On the input we have a set of pairs of matched feature point coordinates on two images:

$$S = \{(x, y) \mid x \in X, y \in Y\},$$

where X is the first image, and Y is the second image.

2. For each i from 1 to N build a hypothesis and check it:

   (a) We build a hypothesis θ_i by selecting random pairs

$$S_i = \{(x_i, y_i) \mid (x_i, y_i) \in S\}.$$

   In our case, it is enough to select 4 points from each of X and Y sets to build a matrix M for the perspective transformation hypothesis.

   (b) Evaluate the hypothesis θ_i by applying the perspective transformation matrix M to all points of the first X set and checking their matches with the points of the second

Y set with some threshold. The number of matches is the hypothesis evaluation score $R(\theta_i)$:

$$R(\theta) = \sum_{x \in X p(\theta, x, Y)},$$

$$p(\theta, x, Y) = \begin{cases} 1, & |\varepsilon(\theta, x, Y)| \leq T, \\ 0, & |\varepsilon(\theta, x, Y)| > T, \end{cases}$$

where $\varepsilon(\theta, x)$ is the minimum distance from point x to the points of the set Y with hypothesis θ.

(c) If this is the first hypothesis, then store it as a current best hypothesis θ_0. Otherwise, check if the current hypothesis θ_i is better than the best one found before θ_0, and if so, then it is stored as the new best hypothesis:

$$(i = 0) \vee \big(R(\theta_i) > R(\theta_0)\big) \Rightarrow \theta_0 = \theta_i.$$

3. After finishing N iterations, the θ_0 stores the best hypothesis. In our case, it is a perspective transformation matrix that transforms the first image into the coordinate system of the second image.

The probability of choosing at least one sample without outliers with the RANSAC method can be estimated as follows:

$$p = 1 - \big(1 - N(1 - e)^s\big)^N,$$

where

- p is the probability of getting a good sample in N iterations;
- N is the number of samples (iterations);
- s is the number of points in the sample;
- e is the ratio of outliers.

Since after estimating at least one hypothesis we can estimate the ratio of outliers, we can estimate the required number of iterations based on the currently best hypothesis:

$$N = \frac{\log(1 - p)}{\log\big(1 - (1 - e)^s\big)},$$

where:

- p is the desired probability of success (e.g., 0.99),
- e is the estimated outlier ratio,
- s is the minimal number of points required to estimate the model (e.g., 4 for homography).

The modification of the RANSAC method that uses M-estimator to evaluate the hypothesis is called M-SAC. In this case, each point score $p(\theta, x, Y)$ depends on the minimum distance from point x to the points of the set Y with hypothesis θ:

$$R(\theta) = \sum_{x \in X} p(\theta, x, Y),$$

$$p(\theta, x, Y) = \begin{cases} \varepsilon^2(\theta, x, Y), & |\varepsilon(\theta, x, Y)| \leq T, \\ T^2, & |\varepsilon(\theta, x, Y)| > T, \end{cases}$$

where $\varepsilon(\theta, x, Y)$ is the residual (distance) between the transformed point x and its nearest match in Y under hypothesis θ, and T is a threshold.

3.2 Task 1. Feature Points Detection

3.2.1 SIFT Feature Points Detector with OpenCV

OpenCV provides a `cv2.SIFT` class to work with the SIFT feature point detector [7]. An instance of this class can be created by the `cv2.SIFT_create()` function. The constructor specifies additional detector parameters, e.g., the first parameter named *nfeatures* limits the number of detected features to a specified number of strong feature points. This class implements the `cv2.Feature2D` interface with the following functions used to detect feature points and compute their descriptors:

- `cv2.SIFT.detect(image, mask) -> keypoints` function detects feature points of the *image* with the region of interest (ROI) defined by *mask* and returns the *keypoints* list of detected feature points.

- `cv2.SIFT.compute(image, keypoints) -> fp, descriptors` function computes descriptors for a list feature points *fp* of the *image* and returns a tuple with feature points *fp* and *descriptors*. In case some descriptor cannot be calculated, it is removed from the returned array. If two dominant orientations are found, then the feature point is duplicated in the returned array with two separate descriptors.

- `cv2.SIFT.detectAndCompute(image, mask) -> keypoints, descriptors` function unites functions `detect()` and `compute()`. It detects the feature points of the *image* with the region of interest defined by *mask*, computes their descriptors, and stores points to the list *keypoints* with the corresponding *descriptors* list.

After detecting, feature points can be displayed using `cv2.drawKeypoints(image, keypoints, outImage, color, flags) -> Iout` function. This function draws the feature points defined by *keypoints* with a given *color* on the image *image* and stores them in the *outImage* image. If the output image is not specified, then it is automatically created and returned. By default, the *color* of each feature point is different, and only the feature point position is displayed. The optional *flags* parameter value of `cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS` allows drawing the feature point size and orientation.

Let us try opening an image and displaying all feature points detect with OpenCV and SIFT detector in automatic mode.

```
# Read an image from a file in BGR
fn = "images/tower.jpg"
I1 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I1, np.ndarray) or I1.data
    == None:
     print("Error reading file \"{}\"".format(
    fn))
     exit()

# Convert to grayscale
I1gray = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
```

```
10
11 # Instantiate SIFT class object
12 sift = cv2.SIFT_create()
13
14 # Detect SIFT feature points
15 I1fp = sift.detect(I1gray)
16
17 # Draw feature points (key points) on the
     image
18 I1sift_all = cv2.drawKeypoints(I1, I1fp, None)
19
20 # Display it
21 ShowImages([("Source image", I1),
22             ("SIFT features", I1sift_all)], 2)
```

Figure 3.8. Source Image

Figure 3.9. SIFT features

As you can see, multiple feature points were detected, but we can not understand the size and orientation of each feature. Let us try and draw them in rich format to show the characteristic scale and orientation of each feature point as well.

```
# Draw feature points (key points) on the
    image with a rich description
I1sift_all_rich = cv2.drawKeypoints(
    I1, I1fp, None,
```

```
    flags=cv2.
    DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
)

# Display it
ShowImages([("Source image", I1),
            ("Rich SIFT features",
    I1sift_all_rich)], 2)
```

Figure 3.10. Source Image

Figure 3.11. Rich SIFT Features

Now the image is overloaded with data, so let us limit the number of feature points to the top 100 strongest ones and display them in a rich format. In addition, we will specify the green feature points color to make them more noticeable on the image.

```
# Instantiate SIFT class object for 100 strongest features
sift = cv2.SIFT_create(nfeatures=100)

# Detect SIFT feature points
I1fp = sift.detect(I1gray)

# Draw feature points (key points) on the image and red color (to be used later in comparison)
I1sift_top = cv2.drawKeypoints(I1, I1fp, None, color=(0, 0, 255))

# Draw feature points (key points) on the image with a rich description
I1sift_top_rich = cv2.drawKeypoints(
    I1, I1fp, None, color=(0, 255, 0),
    flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
)

# Display it
ShowImages([("Source image", I1),
            ("Top 100 SIFT features", I1sift_top_rich)], 2)
```

Figure 3.12. Source Image

Figure 3.13. Top 100 SIFT Features

As you can see, the resulting strongest feature points are exactly the corner points with the corresponding scales. Hence, if we try and search for the features on the same object with a bit different view angle or scale, the same points would be detected.

3.2.2 ORB Feature Points Detector with OpenCV

Now let us try and do the same with the ORB detector. OpenCV provides a class named `cv2.ORB` for the detection of ORB feature points and the calculation of corresponding descriptors [8]. A class instance is created with `cv2.ORB_create()`. Additional constructor parameters allow modify the descriptor parameters, e.g., the first *nfeatures* parameter specifies the number of features to extract from an image. TThe class interface is the same as the SIFT detector and detects feature points and computes their descriptors.

This class implements the same `cv2.Feature2D` interface with the `cv2.SIFT` class, so it implements the same set of functions to compute feature points and their descriptors.

Let us do the same as we did for the SIFT detector but now for the ORB:

- Display all feature points
- Display 100 strongest features with rich descriptions.

```
# Instantiate ORB class object
orb = cv2.ORB_create()

# Detect ORB feature points
I1fp = orb.detect(I1gray)

# Draw feature points (key points) on the
    image
I1orb_all = cv2.drawKeypoints(I1, I1fp, None)

# Instantiate ORB class object for 100
    strongest features
orb = cv2.ORB_create(nfeatures=100)

# Detect ORB feature points
I1fp = orb.detect(I1gray)

# Draw feature points (key points) on the
    image and red color (to be used later in
    comparison)
```

```
I1orb_top = cv2.drawKeypoints(I1, I1fp, None,
    color=(0, 0, 255))

# Draw feature points (key points) on the
    image with a rich description
I1orb_top_rich = cv2.drawKeypoints(
    I1, I1fp, None, color=(0, 255, 0),
    flags=cv2.
    DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
)

# Display it
ShowImages([("Source image", I1),
            ("ORB features", I1orb_all),
            ("Top 100 ORB features",
    I1orb_top_rich)], 3)
```

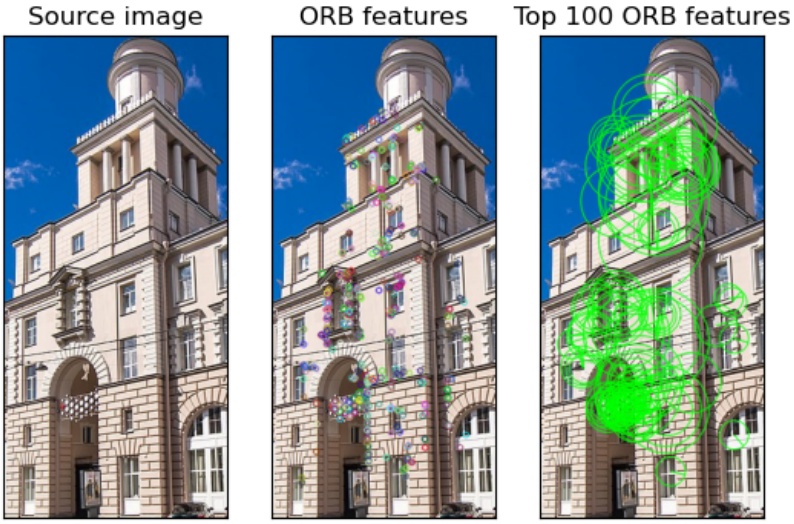


Figure 3.14. Orb Features

Now we can compare features detected by SIFT and ORB.

```
# Display it
ShowImages([("Top 100 SIFT", I1sift_top),
            ("Top 100 ORB", I1orb_top)], 3)
```

Figure 3.15. Comparing Top 100 SIFT and ORB Features

As you can see, the top feature points are very similar, no matter which detector is used. However, later you will see that feature point descriptors make some difference.

3.3 Task 2. Feature Points Matching

Take two pairs of images: the first image of each pair must have an object (e.g., some book) and the second image must be a scene containing this object. Extract the feature points of an object and match them with the feature points of a scene containing this object. Calculate

the transformation matrix using the RANSAC method and highlight the object position in the scene. Show the inlier matches. Compare feature point descriptors for the task of image matching.

Let us try and search for the objects with the OpenCV library. We will take a pair of images of an object and the scene with this object. In our case, this will be:

- the tower of the main building of ITMO University (object),
- a photo of the whole building (scene).

They are shot from a bit different angles, but the same feature points can be found on both of the images. Since the two views have different affine geometric transformations, our goal is to match the object with the whole scene and find this transformation.

This task is done in the following steps:

1. Detect the feature points on two images and compute the descriptors for them.
2. Match the detected feature points.
3. Find the transformation between the two images by searching for a transformation matrix that best fits these matches.

3.3.1 Detect the feature points on two images and compute the descriptors

First, we have to load images and detect the feature points. It's worth using the `detectAndCompute()` function to calculate the feature point descriptors along with their detection. We will use the SIFT descriptor for the feature points search.

```
# Read the first image from a file in BGR
fn1 = "images/tower.jpg"
I1 = cv2.imread(fn1, cv2.IMREAD_COLOR)
if not isinstance(I1, np.ndarray) or I1.data
    == None:
     print("Error reading file \"{}\"".format(
    fn1))
     exit()
```

```
7
8  # Read the second image from file in BGR
9  fn2 = "images/building.jpg"
10 I2 = cv2.imread(fn2, cv2.IMREAD_COLOR)
11 if not isinstance(I2, np.ndarray) or I2.data
       == None:
12     print("Error reading file \"{}\"".format(
       fn2))
13     exit()
14
15 # Convert to grayscale
16 I1gray = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
17 I2gray = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)
18
19 # Instantiate SIFT class object
20 sift = cv2.SIFT_create()
21
22 # Detect and compute SIFT feature points
23 I1fp, I1des = sift.detectAndCompute(I1gray,
       None)
24 I2fp, I2des = sift.detectAndCompute(I2gray,
       None)
25
26 # Draw feature points (key points) on the
       image with a rich description
27 I1sift_all_rich = cv2.drawKeypoints(
28     I1, I1fp, None,
29     flags=cv2.
       DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
30 )
31 I2sift_all_rich = cv2.drawKeypoints(
32     I2, I2fp, None,
33     flags=cv2.
       DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
34 )
35
36 # Display it
37 ShowImages([("Object", I1sift_all_rich),
```

```
38                ("Target image", I2sift_all_rich)
    ], 2)
```

Figure 3.16. Feature Object

Figure 3.17. The Target

3.3.2 Match the detected feature points

After the feature points are detected and their descriptors are computed, the next step is to match the feature point descriptors on the object image with the descriptors on the target image. OpenCV provides two feature point descriptor matchers:

- `cv2.BFMatcher` class implements the brute force matcher. For each feature point descriptor of the first set of points, it finds the best match in the second set by iterating through all its feature point descriptors. The class constructor has two parameters:
  - *normType* sets the matcher distance calculation method. The default value is Euclidean L2 distance specified by `cv2.NORM_L2`, which fits SIFT descriptors. In the case of ORB descriptors, it's better to use the Hamming distance, which can be specified by `cv2.NORM_HAMMING`.
  - *crossCheck* sets if the cross-checks filter must be used. By default, it is `Off`.
- `cv2.FlannBasedMatcher` class is the implementation of Fast Library for Approximate Nearest Neighbors matcher. It uses various algorithms for accelerated feature point descriptors matching (KD-trees, k-means, LSH, etc.).

All OpenCV feature point matchers implement the `cv2.DescriptorMatcher` interface. They have two main matching functions:

- `match(queryDescriptors, trainDescriptors) -> matches` class method matches descriptors *queryDescriptors* and *trainDescriptors*. For each descriptor in *queryDescriptors*, a single best match is found in the *trainDescriptors*. This match is returned in the *matches* array.
- `match(queryDescriptors, trainDescriptors, k) -> matches` class method matches k-nearest descriptors *queryDescriptors* and *trainDescriptors*. For each descriptor in *queryDescriptors*, the k best matches are found in the *trainDescriptors* and are returned in the *matches* array. Each item of

the *matches* array is an array containing the strongest k matches for each feature point. Please note that the k-nearest matching method is not compatible with the cross-check in the brute force matcher since the cross-check does not allow to find more than one match.

Each match in the returned *matches* array is an instance of the `cv2.DMatch` class containing the following fields:

- *queryIdx* is an index of the feature point in the query feature points set (*queryDescriptors*);
- *trainIdx* is an index of the feature point in the train feature points set (*trainDescriptors*);
- *imgIdx* is an index of the image in the train feature points images set;
- *distance* is a distance measure between these two descriptors.

The `cv2.DescriptorMatcher` interface also has the functions that can be used to train a matcher on a set of images with the corresponding descriptors to match a single query (*queryDescriptors* in our case) with a set of image descriptors (instead of single *trainDescriptors*). This is the reason for the index returned in the *imgIdx* parameter of the `cv2.DMatch` class. However, since we are matching only two images we don't need this extra information.

Let us create these two matchers and match the feature points without cross-checking.

```
# Create a brute force matcher
bf_matcher = cv2.BFMatcher(normType=cv2.
    NORM_L2, crossCheck=False)
# And match descriptors
bf_matches = bf_matcher.match(I1des, I2des)

# Create a FLANN matcher
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=
    FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
```

```
flann_matcher = cv2.FlannBasedMatcher(
    index_params, search_params)
# And match descriptors
flann_matches = flann_matcher.match(I1des,
    I2des)
```

When using the ORB descriptor and FLANN matcher, the LSH algorithm must be used. It can be done as follows:

```
# Set the FLANN parameters
FLANN_INDEX_LSH = 6
index_params = dict(
    algorithm=FLANN_INDEX_LSH,
    table_number=6,
    key_size=12,
    multi_probe_level=1
)
search_params = dict(checks=50)

# Create matcher
flann_lsh_matcher = cv2.FlannBasedMatcher(
    index_params, search_params)
```

Now we can use `cv2.drawMatches(img1, keypoints1, img2, keypoints2, matches1to2, outImg, matchColor, singlePointColor, matchesMask, flags)` function to display the found matches. It combines images *img1* and *img2* into a single image, draws feature points *keypoints1* and *keypoints2* on them, and connects the matched ones with the lines according to the *matches1to2* list of matches. Optional parameters allow specifying the match connection line color, non-matched feature points color, mask for match visualization, and extra visualization *flags* (we will use `cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS` flag to exclude not matched points from visualization).

To display only top matches, the *matches* array acquired after descriptor matching can be sorted by the *distance* value and its top part visualized either by slicing the matches array or by specifying a mask for match drawing.

Let us do it.

```
# Define a parameter for a count of matches to visualize
num_matches = 50

# Create an image with brute force matches
matches = sorted(bf_matches, key=lambda x: x.distance)
I1bf_match = cv2.drawMatches(
    I1, I1fp, I2, I2fp, matches, None,
    matchesMask=[1 if i < num_matches else 0 for i in range(len(matches))],
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS,
    matchColor=(0, 255, 0)
)

# Create an image with FLANN matches
matches = sorted(flann_matches, key=lambda x: x.distance)
I1flann_match = cv2.drawMatches(
    I1, I1fp, I2, I2fp, matches, None,
    matchesMask=[1 if i < num_matches else 0 for i in range(len(matches))],
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS,
    matchColor=(0, 255, 0)
)

# Display it
ShowImages([("Brute force matches", I1bf_match),
            ("FL matches", I1flann_match)], 2)
```

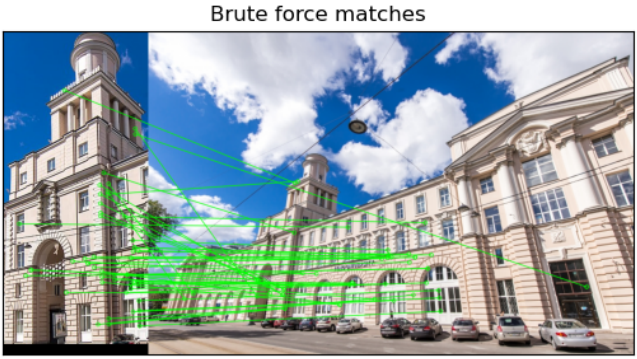


Figure 3.18. Direct Matches

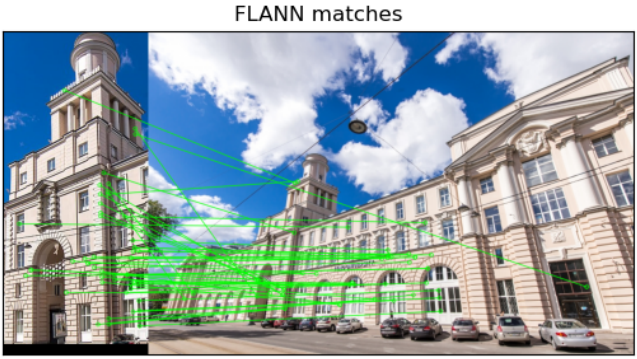


Figure 3.19. FLANN Matches

As can be seen from feature points matching results, using the first best match may result in a lot of descriptor matches and a lot of false matches among them.

When using the brute force matcher we can add the cross-check operation as well to increase the match quality. Let us try it.

```
# Create a brute force matcher with cross-
    checking
```

```
bf_cc_matcher = cv2.BFMatcher(crossCheck=True)
# And match descriptors
bf_cc_matches = bf_cc_matcher.match(I1des, I2des)

# Create an image with brute force cross-checked matches
matches = sorted(bf_cc_matches, key=lambda x: x.distance)
I1bf_cc_match = cv2.drawMatches(
    I1, I1fp, I2, I2fp, matches, None,
    matchesMask=[1 if i < num_matches else 0 for i in range(len(matches))],
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS,
    matchColor=(0, 255, 0)
)

# Display it
ShowImages([("Brute force cross-checked matches", I1bf_cc_match)], 2)
```

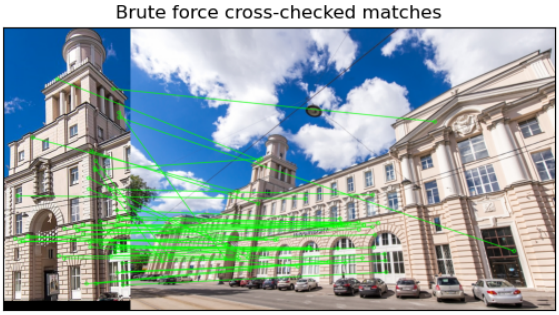


Figure 3.20. Brute Cross-checked Matching

Another solution is to use the k-nearest matching to filter some of the detected matches. In this case, the match is considered to be *good* if

it is significantly different from the next nearest match, so the distance between the first nearest match is significantly lower compared to the distance with the second nearest match.

Let us try it as well.

```
# We will use FLANN matcher for this
flann_knn_matches = flann_matcher.knnMatch(I1des, I2des, k=2)

# This is the ratio between the first and second match to select the first one
knn_ratio = 0.75

# Select good matches only
good = []
for m in flann_knn_matches:
    if len(m) > 1:
        if m[0].distance < knn_ratio * m[1].distance:
            good.append(m[0])
flann_knn_matches = good

# Create an image with k-nn filtered matches
matches = sorted(flann_knn_matches, key=lambda x: x.distance)
I1flann_knn_match = cv2.drawMatches(
    I1, I1fp, I2, I2fp, matches, None,
    matchesMask=[1 if i < num_matches else 0 for i in range(len(matches))],
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS,
    matchColor=(0, 255, 0)
)

# Display it
ShowImages([("k-nn filtered FLANN matches", I1flann_knn_match)], 2)
```

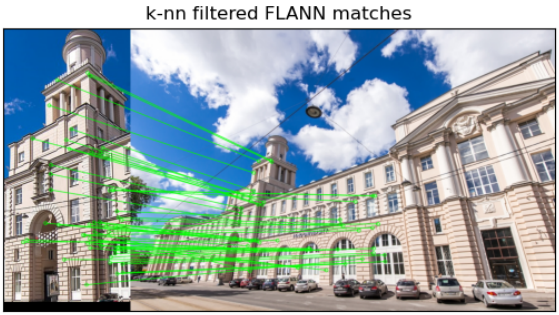


Figure 3.21. k-nn FLANN Matching

3.3.3 Find the transformation matrix

Now we have a set of matching feature points on two images, but they don't form any transformation as they still have outliers. Hence, we have to filter out the outliers and calculate the transformation matrix between the two sets of matched feature points. This can be done by the RANSAC (RANdom SAmple Consensus) algorithm implemented in OpenCV:

```
1    cv2.findHomography(srcPoints, dstPoints,
     method, ransacReprojThreshold, mask,
     maxIters, confidence)} \to H
```

This function calculates the homography transformation matrix H for the perspective transformation from the coordinate system of the first image `srcPoints` to the second image `dstPoints`. The inliers and outliers of the best RANSAC hypothesis are marked in the returned `mask`. Optional parameters specify:

- `ransacReprojThreshold` — reprojection threshold (defaults to 3),
- `maxIters` — maximum number of iterations (defaults to 2000),
- `confidence` — confidence level (defaults to 0.995).

The OpenCV implementation of the RANSAC method requires at least 10 matches to work, so this must be checked before the call.

Since we already have found matches, now we can execute the RANSAC method to find the homography matrix.

Let us do it for matches found with the SIFT descriptor and the brute force matcher with cross-checking.

```
# First we have to check that we have the
    required minimum of 10 matches
MIN_MATCH_COUNT = 10
if len(bf_cc_matches) < MIN_MATCH_COUNT:
    print("Not enough matches to calculate the
    homography matrix.")
else:
    # Create arrays of coordinates in two
    images
    matches = sorted(bf_cc_matches, key=lambda
    x: x.distance)
    I1pts = np.float32([I1fp[m.queryIdx].pt
    for m in matches]).reshape(-1, 1, 2)
    I2pts = np.float32([I2fp[m.trainIdx].pt
    for m in matches]).reshape(-1, 1, 2)

    # Run the RANSAC method to find the
    transformation
    H, mask = cv2.findHomography(I1pts, I2pts,
    cv2.RANSAC, 5)

    if H is None:
        print("Homography matrix was not found
    .")
    else:
        mask = mask.ravel().tolist()

        # Calculate a rectangle of the first
    image in the second CS
        # For this we define four corner
    points and transform them with the
    homography matrix
        h, w = I1.shape[:2]
```

```
22          I1box = np.float32([[0, 0], [0, h-1],
    [w-1, h-1], [w-1, 0]]).reshape(-1, 1, 2)
23          I1to2box = cv2.perspectiveTransform(
    I1box, H)
24
25          # Now when we have the coordinates of
    the first image corners in the coordinate
    system
26          # of the second image, we can
    highlight the object on the second image
27          I2res = I2.copy()
28          cv2.polylines(I2res, [np.int32(
    I1to2box)], True, (0, 0, 255), 2, cv2.
    LINE_AA)
29
30          # Draw found good matches with found
    transformation
31          I2trans = cv2.drawMatches(
32              I1, I1fp, I2, I2fp, matches, None,
33              matchesMask=mask,
34              flags=cv2.
    DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS,
35              matchColor=(0, 255, 0)
36          )
37
38          # Display an image
39          ShowImages([("Query on Source", I2res)
    ,
40                      ("Matches with brute force
     and cross-checking", I2trans)], 2)
```

Figure 3.22. Query

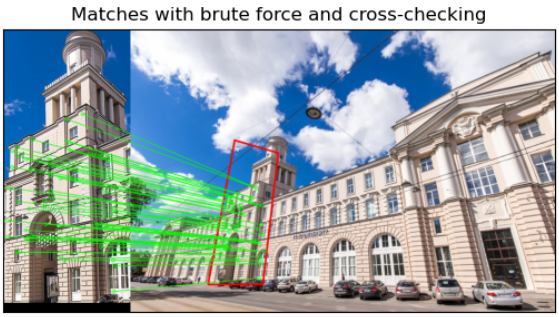


Figure 3.23. Final Matching

3.4 Procedure of Performing Practical Assignment

All tasks are based on the material presented in the corresponding sections of the assignment. Students are expected to write a code for each task using Python, OpenCV, and NumPy.

Task 1. Feature Points Detection

1. **SIFT Feature Points Detector with OpenCV**
   Take three arbitrary images and search for the feature points with the SIFT detector. Display the detected points on the images.

   Hints for implementation:

   - Use `cv2.SIFT_create()` to instantiate the SIFT detector.
   - Use `sift.detect(image, mask)` to detect feature points.
   - Use `cv2.drawKeypoints(image, keypoints, outImage, color, flags)` to display feature points.
   - Use `cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS` flag to show feature point size and orientation.
   - You can limit the number of features using `cv2.SIFT_create(nfeatures = N)` parameter.

2. **ORB Feature Points Detector with OpenCV**
   Take the same three images that you used with the SIFT detector and search for the feature points with the ORB detector. Display the detected points on the images. Compare top SIFT and ORB feature points.

   Hints for implementation:

   - Use `cv2.ORB_create()` to instantiate the ORB detector.
   - Use `orb.detect(image, mask)` to detect feature points.
   - Use `cv2.drawKeypoints()` with rich flags to display feature points with orientation and scale.
   - Compare the results visually with SIFT detector results.
   - Note that ORB is faster but may detect slightly different feature points compared to SIFT.

Task 2. Feature Points Matching

1. **Find Homography Matrices for Different Matching Methods**
   Take the three remaining match sets found in the example and find the homography matrices:

(a) `bf_matches` for brute force matches filtered without cross-checking;

(b) `flann_matches` for FLANN matches;

(c) `knn_matches` for FLANN matches filtered with tolerancing k-nn matches.

If the object is found then highlight the object position in the scene and show the inlier matches. Was the object found correctly?

Hints for implementation:

- Check that you have at least 10 matches (`MIN_MATCH_COUNT = 10`) before calculating homography.
- Create arrays of coordinates: extract `queryIdx` and `trainIdx` from the matches.
- Use `cv2.findHomography(srcPoints, dstPoints, cv2.RANSAC, ransacReprojThreshold)` to find the transformation matrix.
- Use `cv2.perspectiveTransform()` to transform the corner points of the first image.
- Draw the transformed rectangle on the second image using `cv2.polylines()`.
- Use the returned `mask` from `findHomography` to show only the inlier matches.
- Compare the results from different matching methods - which one gives the best result?

2. **Object Detection with SIFT and ORB**
Take two arbitrary pairs of images: the first image of each pair must have an object (e.g., some book) and the second image must be a scene containing this object. Extract the feature points of the object and match them with the feature points of the scene containing this object. Calculate the transformation matrix using the RANSAC method and highlight the object position in the scene. Show the inlier matches. Display the results.

The task must be completed with both SIFT and ORB detectors. You can select any feature points matching algorithm.

Hints for implementation:

- **For SIFT detector:**
  - Use `cv2.BFMatcher(normType = cv2.NORM_L2)` for brute force matching.
  - Or use `cv2.FlannBasedMatcher` with KD-tree algorithm.
  - Consider using cross-checking or k-nn matching with ratio test ($r = 0.75$).
- **For ORB detector:**
  - When using the brute force matcher for ORB descriptors, it must be created with the Hamming distance norm: `cv2.BFMatcher(normType = cv2.NORM_HAMMING, crossCheck = True)`.
  - When using the FLANN matcher for ORB descriptors, it must be created with the LSH algorithm:

```
FLANN_INDEX_LSH = 6
index_params = dict(algorithm =
                    FLANN_INDEX_LSH,
                    table_number = 6,
                    key_size = 12,
                    multi_probe_level = 1)
search_params = dict(checks = 50)
flann_lsh_matcher =
    cv2.FlannBasedMatcher(index_params,
    search_params)
```

- Follow the same pipeline as in the previous task: detect, compute descriptors, match, find homography, transform and draw.
- Compare the results between SIFT and ORB - which detector performs better for your images?

Task 3. Optional: Automatic Image Stitching

1. **Simple Automatic Image Stitching**
   Implement the simple automatic image stitching for three images. Display the result.

Algorithm description:

(a) Create an empty canvas and place the *middle* image in the center. This will be the canvas where you add other images.

(b) Calculate the homography matrix for transformation from the *left* image to the canvas.

(c) Transform the *left* image with this transformation matrix and place it on areas with black color on the canvas.

(d) Do the same for the *right* image.

Hints for implementation:

- You can assume that the order of the images is known (e.g., all the three images are shot while moving the camera from left to right), so reordering is not required.
- Detect the feature points and compute the descriptors for all the three images (use SIFT or ORB).
- Match the features between *left* and *middle* images, and between *middle* and *right* images.
- Use RANSAC to find homography matrices for both transformations.
- Create a canvas large enough to fit all the three images (consider the transformed coordinates).
- Use `cv2.warpPerspective()` to transform the images according to homography matrices.
- Place the transformed images on the canvas, being careful with the overlapping regions.
- **Note:** This is the simplest approach, so the corners may be cut when stitching.
- **Note:** Since parts of the panoramic images are shot with different exposition parameters, they may slightly differ in lightness. In the scope of this task, this is not taken into account.

General Requirements

- All the results must be visualized using the `ShowImages()` function from the `pa_utils.py` module.
- Code must be well-structured and commented.
- Each task must be implemented in a separate Jupyter notebook cell or section.
- Answers the questions and write a conclusion.

Questions for the Practical Assignment Report Defense

- How can the characteristic orientation (rotation) of the feature point be estimated?
- How are the not-strong feature point descriptor matches on a repeating texture (e.g., windows, water, etc.) filtered out?
- What is the minimum required sample size (the number of matched pairs of feature points) to build an affine transformation hypothesis with the RANSAC method? What is the minimum required sample size to build a perspective transformation hypothesis with the RANSAC method?
- How are feature points used for stitching a panoramic image?

4 Practical Assignment No 4. Face Detection. Viola-Jones Approach

Objectives

- **Study the Viola-Jones face detection framework**
  To understand the four key concepts of the method: Haar-like features as weak classifiers, integral image representation for fast feature computation, AdaBoost training algorithm for combining weak classifiers into a strong one, and cascade structure for efficient processing.
- **Master practical face detection using OpenCV**
  To gain hands-on experience with OpenCV's built-in cascade classifiers for detecting faces in static images, learn how to work with various pre-trained cascades, and understand the impact of detection parameters (scaleFactor, minNeighbors) on the results.
- **Develop skills in body part detection using ROI**
  To learn how to improve detection accuracy by applying Regions of Interest (ROI) based on previously detected faces, implement detection of facial features (eyes, mouth) in the upper or lower parts of the face, and properly transform coordinates between ROI and original image spaces.
- **Analyze detection quality and explore real-time applications**
  To evaluate the performance of the Viola-Jones method by counting detected objects, identifying false positives and missed detections, and optionally implement face detection in video streams, optimizing parameters for real-time processing.

Preface

To successfully complete this practical assignment, students will need the following:

- **Python 3.x** with core scientific libraries:
  - `opencv` (`opencv-python` pip library) — for image I/O, built-in cascade classifiers, and video stream processing;

- `numpy` (<=1.26.4) — for efficient array and matrix manipulations, image slicing for ROI extraction;
- `matplotlib` — for additional visualization of detection results (optional);
- `IPython.display` — for displaying within Jupyter notebooks (optional).

Example installation command: `pip install opencv-python, numpy==1.26.4, matplotlib, ipython`

Note: OpenCV installation includes pre-trained cascade files (`.xml`) for face, eye, smile, and body part detection. These files are essential for this assignment and are typically located in the `data/haarcascades/` folder of the OpenCV distribution. Students have to make sure they can access these cascade files or download them from the OpenCV GitHub repository.

- **Development environment:** It is recommended to use `Jupyter Notebook` or `Jupyter Lab` (locally or via Google Colab). All tasks are designed to be completed interactively in a `.ipynb` format. Alternatively, any Python IDE with the support for interactive execution (e.g., VS Code with Python extension) is acceptable.

  To execute Python programs, you need to install a development environment, for example, Visual Studio Code with the Python development extension and additional Python packages to run programs – `ipykernel`.

- **Utility module:** The file `pa_utils.py` is provided alongside the assignment. This module contains auxiliary functions essential for the work:

  - `ShowImages()` — displays multiple images side-by-side with titles, used throughout all tasks for result visualization of detected faces, eyes, and other body parts;
  - `exit()` — provides clean cell termination within Jupyter without stopping the kernel.

The module is imported at the very beginning of the notebook and is used throughout all tasks to display intermediate and final results.

- **Test images and video:** Students prepare or download test images containing multiple faces at different scales and orientations. For the optional task, a video file with faces is required. Example images and videos may be provided in the `images/` folder alongside the assignment.

4.1 Preliminary

The Viola-Jones face detector method [17] is based on the following four concepts:

1. Haar-like features as weak classifiers;
2. Integral image representation for fast calculation of Haar-like features;
3. AdaBoost training method to combine weak classifiers into a strong classifier;
4. Combining strong classifiers into a cascade classifier.

4.1.1 Haar-like Features

Haar-like feature is a kind of weak classifier. It can be defined as the difference of the sum of pixels of areas inside the rectangle, which can be at any position and scale within the original image. Traditional Viola-Jones face detector algorithm uses four types of Haar-like features that are shown in Fig. 1.

First of all, we have to understand what these features are. Let us look at Fig. 4.1.

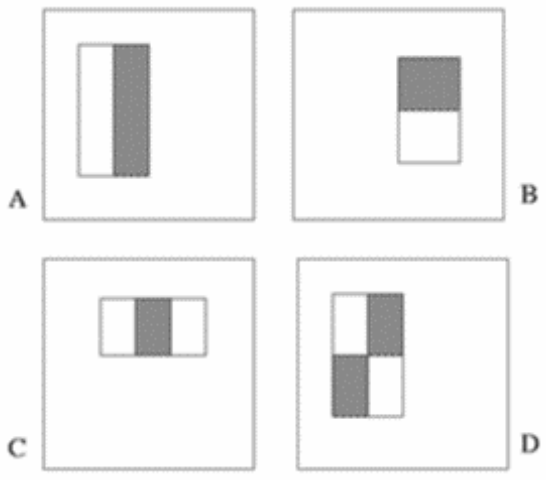


Figure 4.1. Haar-like features used in Viola-Jones face detector

To calculate the value of the Haar-like feature we calculate sums of pixels inside rectangular areas of the image and do it as fast as possible.

$$\text{value} = \sum(\text{pixels in black area}) - \sum(\text{pixels in white area}).$$

The straightforward calculation of the sum of pixel values in a rectangle would require the number of addition operations equal to the number of pixels minus one. To speed up the feature calculation, an integral image representation is used. In this representation, each pixel stores the sum of all pixel values positioned to the left and above of the current pixel, see Fig. 4.2.

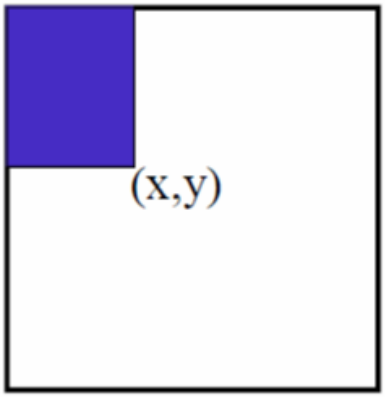


Figure 4.2. Integral image

After the integral image is calculated, to calculate the sum of pixel intensity values in an arbitrary rectangle we need to access only four pixels of an integral image located at the corners of the rectangle, see Fig. 4.3.

$$\text{sum} = D - B - C + A,$$

where

- D is the bottom right corner of the rectangle,
- B is the pixel one pixel above the top right corner of the rectangle,
- C is the pixel one pixel to the left of the bottom left corner of the rectangle,
- A is the pixel one pixel above and to the left of the top left corner of the rectangle.

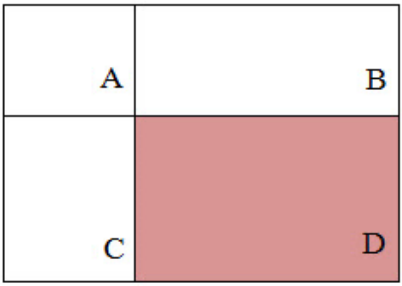


Figure 4.3. Rectangular sum calculation with an integral image

The set of Haar-like features (which are weak classifiers) can be combined with a weighted sum of their values to form a more complex strong classifier, see Fig. 4.4. The training algorithm is called AdaBoost. It consists of several boosting rounds, and each boosting round is a selection of the best weak Haar-like feature to classify the training set taking into account the classification errors of the previous rounds.

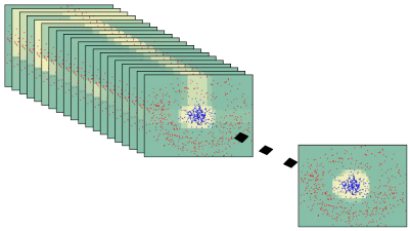


Figure 4.4. Combining weak classifiers into a strong classifier

Formally, the AdaBoost training scheme algorithm can be described with the following steps:

1. On an input we have a training set $T = \{(x_i, y_i) \mid x_i \in X, y_i \in \{-1, 1\}\}$ and a set of all possible weak classifiers $\{h\}$.
2. Initialize the weights for training set items to be equal and sum up to 1: $D_1(i) = 1/m$, where m is the number of training set items.

3. Do K iterations:

   (a) Choose h_k from a set of weak classifiers H, so that the weighted classification error probability is minimal (the probability of the wrong classification with taking weights into account):

$$\epsilon_k = \Pr_{i \sim D_k} \big[h_k(x_i) \neq y_i \big].$$

   (b) Calculate the weight of the currently selected weak classifier based on its classification error probability:

$$\alpha_k = \frac{1}{2} \ln \left(\frac{1 - \epsilon_k}{\epsilon_k} \right).$$

   (c) Reweight the training set with new weights:

$$D_{k+1}(i) = \frac{D_k(i)}{Z_k} \cdot \begin{cases} e^{-\alpha_k}, & h_k(x_i) = y_i, \\ e^{\alpha_k}, & h_k(x_i) \neq y_i. \end{cases}$$

   where Z_k is a normalization factor ensuring $\sum_i D_{k+1}(i) = 1$.

4. After completing K iterations build a strong classifier as a weighted sum of weak classifiers that were selected during boosting rounds:

$$H(x) = \operatorname{sign} \left(\sum_{k=1}^{K} \alpha_k h_k(x) \right).$$

4.1.2 Cascade Classifiers

A strong classifier that has a required accuracy may require the calculation of too many weak classifiers that would slow down the detection speed. Considering that most scanned windows do not contain faces, to speed up the detection rate a set of classifiers with increasing complexity is organized in a cascade of classifiers. The cascade contains a set of classifiers with an increasing complexity and detection rate, see Fig. 4.5.

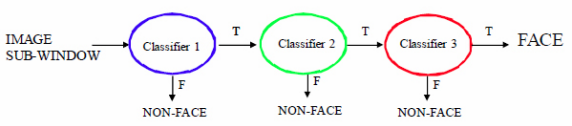


Figure 4.5. Cascade classifier

To be classified positively, a sliding window must pass all cascade stages. In case any classifier rejects the window, it is immediately rejected, and the detector proceeds to the next window. As a result, most of the negative windows are quickly rejected with the first fast classifiers in the cascade.

The detection rate (true positive rate, TP) of a cascade classifier is a multiplication of detection rates of all classifiers in a cascade:

$$TP = \prod_i TP_i.$$

The false positive rate (FP) is also a multiplication of false positives of cascade classifiers:

$$FP = \prod_i FP_i.$$

As a result, to build a classifier with 0.9 true positive rate and 10^{-6} false negative, each classifier in a cascade must meet the requirement of 0.99 true positive and just 0.3 false positive.

Each classifier of the cascade is trained using the AdaBoost training scheme with the requirement to maximize the true positive detection rate while keeping false positives within a given range. The training set is modified between the boosting rounds to increase the complexity of each cascade step.

4.2 Task 1. Faces detection

OpenCV provides a class named `cv2.CascadeClassifier` for cascade classifier execution. Cascade is defined in an XML file and can be loaded from a file named `filename` to a classifier object with the `cv2.CascadeClassifier.load(filename)` function. It returns `True` or `False` depending on whether the cascade was loaded successfully.

Notes.

1. The OpenCV library provides some built-in cascade descriptors for faces, eyes, mouths, cat faces, and Russian license plates which can be found in the `data/haarcascades` folder of OpenCV distribution. See [1] for a complete list of built-in cascades.

2. Other cascade classifiers can be trained using the cascade training tool which is beyond of the scope of the current practical assignment. You can refer to the `traincascade` OpenCV tool documentation, see [2] for additional information.

OpenCV provides several built-in pre-trained cascades for face detection:

- `haarcascade_frontalface_default.xml` is the default 24×24 window size pre-trained AdaBoost cascade for face detection. It has a higher detection rate but at the same time it has more false-positives.

- `haarcascade_frontalface_alt.xml` is an alternative 20×20 window size pre-trained AdaBoost cascade for face detection. It has a lower detection rate but also fewer false-positives.

- `haarcascade_frontalface_alt2.xml` and `haarcascade_frontalface_alt_tree.xml` are two other alternative 20×20 window size pre-trained AdaBoost cascade with tree-based decision tree.

- `haarcascade_profileface.xml` is a 20×20 window size pre-trained AdaBoost cascade for profile face detections.

Let us try and create a cascade classifier and load a default built-in cascade for frontal face detection from a file `haarcascade_frontalface_default.xml`.

```
# Load a cascade for face
cascade_face_fn = "haarcascades/
    haarcascade_frontalface_default.xml"
cascade_face = cv2.CascadeClassifier()
if cascade_face.load(cv2.samples.findFile(
    cascade_face_fn)):
```

```
    print("Face cascade successfully loaded
    from \"{}\"".format(cascade_face_fn))
else:
    print("Error loading face cascade from
    \"{}\"".format(cascade_face_fn))
    exit()
```

After a cascade is loaded it can be applied to an image with `cv2.detectMultiScale(image, scaleFactor, minNeighbors, flags, minSize, maxSize) -> objects` function. This function takes an *image* and applies a previously loaded cascade to detect *objects*. Additional arguments allow specifying optional parameters for object detection:

- *scaleFactor* defines a scale factor between two window sizes when scanning an image with window sizes from *minSize* to *maxSize* (default scale factor is 1.1);
- *minNeighbors* defines a threshold of how many objects should be found in the same region to consider the detection to be positive; it allows filtering false-positive detections (default threshold is 3);
- *flags* are not used for Haar cascades, so could be ignored;
- *minSize* and *maxSize* are tuples defining the minimum and maximum size of the window.

There are also two additional functions for multiscale detections that allow to get additional classification parameters, these are:

- `cv2.detectMultiScale2(...)` → *objects*, *numDetections* — allows getting the number of different windows joined together to form the detected objects' location;
- `cv2.detectMultiScale3(..., outputRejectLevels=True)` → *objects*, *rejectLevels*, *levelWeights* — returns the detection weights which are the certainty of classification at the last stage.

Let us load an image with people and run face detection with the cascade classifier we loaded.

```
# Read an image from a file in BGR
fn = "images/faces.jpg"
I1 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I1, np.ndarray) or I1.data
    == None:
     print("Error reading file \"{}\"".format(
    fn))
     exit()

# Convert to grayscale
I1gray = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
# Equalize histogram for a better processing
    quality
I1gray = cv2.equalizeHist(I1gray)

# Display it
ShowImages([("Source image", I1)], 2)

# And detect faces
I1objects = cascade_face.detectMultiScale(
    I1gray, minNeighbors=3)

print("{} faces were detected".format(len(
    I1objects)))
```

Figure 4.6. Source Faces

The returned `objects` array stores the list of rectangles for the found objects (in our case they are faces). So, we can iterate them all and highlight them on the source image. Also, we can extract faces and display them separately.

```
# Highlight faces
I1out = I1.copy()
I1faces = []
for (x, y, w, h) in I1objects:
    # Draw a rectangle on an image to highlight the detected face with a yellow color
    cv2.rectangle(I1out, (x, y, w, h), (0, 255, 255), thickness=1)

    # Extract the face from an image by matrix slicing
    I1face = I1[y:y+h, x:x+w]
    # And add it to the array for displaying
    I1faces.append("Face {}".format(len(I1faces) + 1), I1face)

# Display it
ShowImages([("Detected faces", I1out)], 2)
```

```
15 ShowImages(I1faces, 5)
```

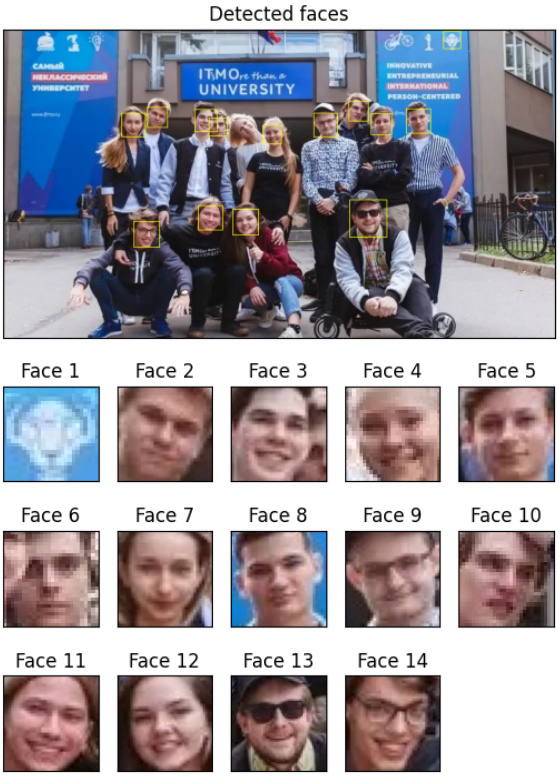


Figure 4.7. Detected Faces

As can be seen from the resulting image, almost all faces were detected, except the one that is rotated and could not be classified by a cascade which was not trained to find this type of rotated faces. Also, a false-positive face-like object was detected; however, it's really a face-like one and completely meets the cascade requirements.

Let us also get additional information about the detected faces, e.g., a detection counter and a certainty for each of them. Since we have

to use different functions to get this additional information, we may get different orders of face detections or even different detections. So, we will process detections with counter information and search for a corresponding weight from the second function output.

```
# Detect faces with extra info
# Detect with hits counter
I1objects, I1nums = cascade_face.
    detectMultiScale2(I1gray, minNeighbors=3)

# And detect with weights
I1objects_, I1rejects, I1weights =
    cascade_face.detectMultiScale3(I1gray,
    minNeighbors=3, outputRejectLevels=True)

print("{} faces were detected".format(len(
    I1objects)))

# Process it the same way as before and print
    the results
# Highlight faces
I1out = I1.copy()
I1faces = []
I1info = []

for i in range(len(I1objects)):
    # Take next face box
    (x, y, w, h) = I1objects[i]
    # And find it in the second array of
    objects to match them
    # We need this to match hits and weights
    j = np.where((I1objects_ == I1objects[i]).
    all(axis=1))[0]
    # If we can't match faces from two
    detections -- ignore this face
    if len(j) != 1:
        print("Something went wrong with face
    {}, ignoring".format(i))
```

```
        continue
    j = j[0][0]

    # Draw a rectangle on an image to highlight the detected face with a yellow color
    cv2.rectangle(I1out, (x, y, w, h), (0, 255, 255), thickness=1)

    # Extract the face from an image by slicing
    I1face = I1[y:y+h, x:x+w]
    # And add it to the array for displaying
    I1faces.append("Face {}".format(i+1), I1face)
    I1info.append("Face {:2d} detected with {:3d} windows joined and weight {:7.4f}".format(
        i, I1nums[i], I1weights[j]))

# Display it
ShowImages([("Detected faces", I1out)], 2)
ShowImages(I1faces, 5)
print("\n".join(I1info))
```

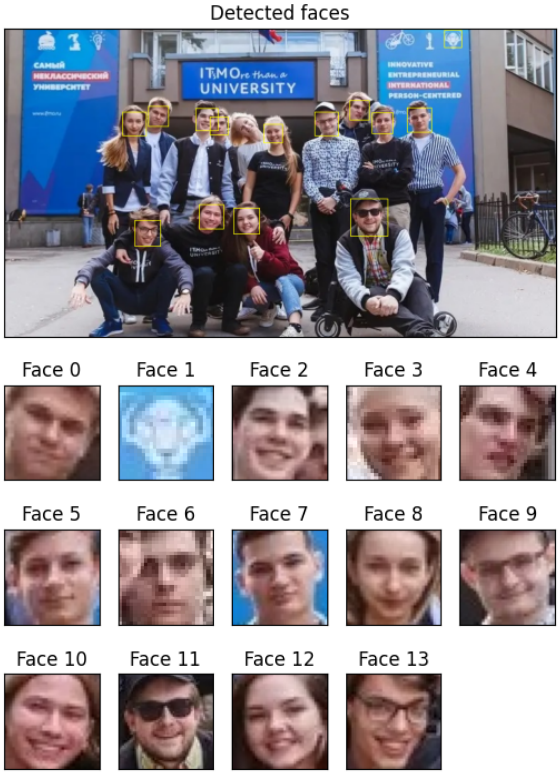


Figure 4.8. Detected Faces with Extra Information

4.3 Task 2. Body Parts Detection

Now let us try to detect eyes on the found face. For this need, we will take a higher-resolution image of a face.

It's obvious that we do not need to scan the whole image for eyes as there would be a lot of false-positive results, so we will define a Region-Of-Interest (ROI) object as a face that was already found, and then search for the eyes by taking into account the ROI. With Python,

we can define the ROI by simply slicing an array — as we already did when displaying the faces found. So, to detect eyes we must:

1. Load a cascade for faces;
2. Use the loaded face cascade to detect faces on the image;
3. Load a cascade for eyes;
4. Go through all the detected faces and use the loaded eyes cascade to detect eyes on the face.

Please note that:

1. When detecting eyes on the face we use ROI, the returned coordinates are in the face coordinate system and must be recalculated to the image coordinate system to display them on the face as well.
2. The zero point of the image coordinate system is the top left corner of the image.
3. When addressing the image array with `numpy`, the image coordinate system (x, y) is inverted according to matrix rules, so if you want to access the point of an `image` with coordinates (x, y) you must write `image[y, x]`. So, slicing an ROI window with the given size would be written as:

```
# Take the rectangle of the next face from the faces array
(x, y, w, h) = faces[i]
# Define an ROI for this rectangle with an array slicing
roi = image[y : y + h, x : x + w]
```

OpenCV provides several cascades which may be used for body parts detections, these are:

- `haarcascade_eye.xml` for eyes detection;
- `haarcascade_eye_tree_eyeglasses.xml` — an alternative eyes detection cascade with a higher accuracy and allowing detection of eyes with eyeglasses;

- `haarcascade_lefteye_2splits.xml` for detection of the left eye only;
- `haarcascade_righteye_2splits.xml` for detection of the right eye only;
- `haarcascade_smile.xml` for a smile (or mouth) detection.

And of course there are cascades for a whole body detection:

- `haarcascade_fullbody.xml` for the full body;
- `haarcascade_upperbody.xml` for the upper body;
- `haarcascade_lowerbody.xml` for the lower body.

These cascades can be added to the first stage of detection if you have to detect the faces of pedestrians. In this case you first have to detect the body and only then use ROI to search for a face.

Let us try this method with the default OpenCV cascade for eyes. Also, now we will use another image with a higher resolution to find theeyes on it.

First, we will run almost the same code as we did before and collect all face information. As a result, we must have:

1. `I2` with source image;
2. `I2out` with source image with faces highlighted;
3. `I2objects` with an array of rectangles for each face on the source image;
4. `I2faces` with an array of tuples with face names and ROI images with the detected faces.

```
# Read an image from a file in BGR
fn = "images/face.png"
I2 = cv2.imread(fn, cv2.IMREAD_COLOR)
if not isinstance(I2, np.ndarray) or I2.data
    == None:
     print("Error reading file \"{}\"".format(
    fn))
```

```
    exit()

# Convert to grayscale
I2gray = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)
# Equalize histogram for a better processing quality
I2gray = cv2.equalizeHist(I2gray)

# And detect faces
# Since our image has high resolution, we set a threshold to 10
I2objects = cascade_face.detectMultiScale(I2gray, minNeighbors=10)
print("{} faces were detected".format(len(I2objects)))

# Highlight faces and search for eyes on each of them
I2out = I2.copy()
I2faces = []
for (x, y, w, h) in I2objects:
    # Draw a rectangle on an image to highlight the detected face with a red color
    cv2.rectangle(I2out, (x, y, w, h), (0, 0, 255), thickness=3)

    # Extract face ROI from an image with slicing
    I2face = I2[y:y+h, x:x+w]
    # And add it to the array for displaying
    I2faces.append("Face {}".format(len(I2faces) + 1), I2face)

# Display it
ShowImages([("Source image", I2),
            ("Detected faces", I2out)], 2)
ShowImages(I2faces, 5)
```

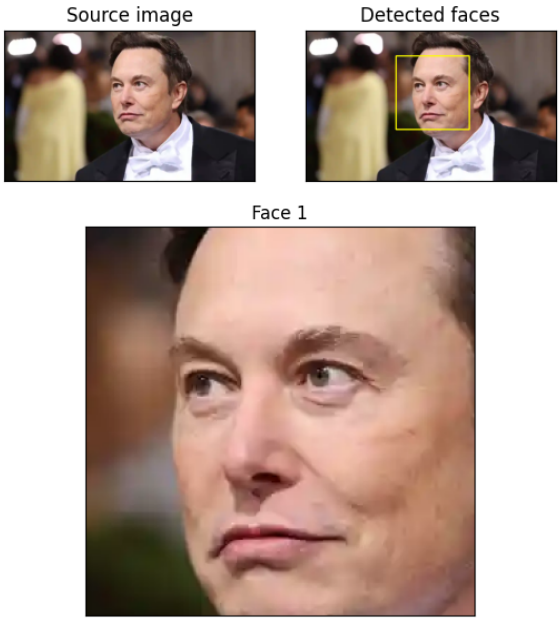


Figure 4.9. Detected Face

Now we can load the eyes cascade and go through all the detected faces to find the eyes and highlight them.

```
# Load cascade for eyes
cascade_eyes_fn = "haarcascades/
    haarcascade_eye.xml"
cascade_eyes = cv2.CascadeClassifier()
if cascade_eyes.load(cv2.samples.findFile(
    cascade_eyes_fn)):
     print("Face cascade successfully loaded
    from \"{}\"".format(cascade_eyes_fn))
else:
     print("Error loading face cascade from
    \"{}\"".format(cascade_eyes_fn))
     exit()
```

```
9
10 I2out_eyes = I2out.copy()
11 I2faces_eyes = []
12
13 # Go through all faces
14 for i in range(len(I2objects)):
15     # Take face and convert it to grayscale
   and normalize
16     face = I2faces[i][1]
17     face = cv2.cvtColor(face, cv2.
   COLOR_BGR2GRAY)
18     face = cv2.equalizeHist(face)
19
20     # Then search for eyes
21     I2eyes = cascade_eyes.detectMultiScale(
   face, minNeighbors=10)
22     print("{} eyes were detected on face {}".
   format(len(I2eyes), i + 1))
23
24     # Now go through all eyes
25     I2face_out = I2faces[i][1].copy()
26     for (x, y, w, h) in I2eyes:
27         # Highlight them on the face
28         cv2.rectangle(I2face_out, (x, y, w, h)
   , (0, 255, 255), thickness=1)
29         # And on the source image
30         # For this we have to shift eyes by
   face box position
31         # This is (I2objects[i][0], I2objects[
   i][1])
32         cv2.rectangle(I2out_eyes,
33                       (x + I2objects[i][0], y +
    I2objects[i][1], w, h),
34                       (255, 0, 0), thickness=2)
35
36     I2faces_eyes.append((I2faces[i][0],
   I2face_out))
37
```

```
38 # Display it
39 ShowImages([("Detected faces and eyes",
       I2out_eyes)], 2)
40 ShowImages(I2faces_eyes, 5)
```

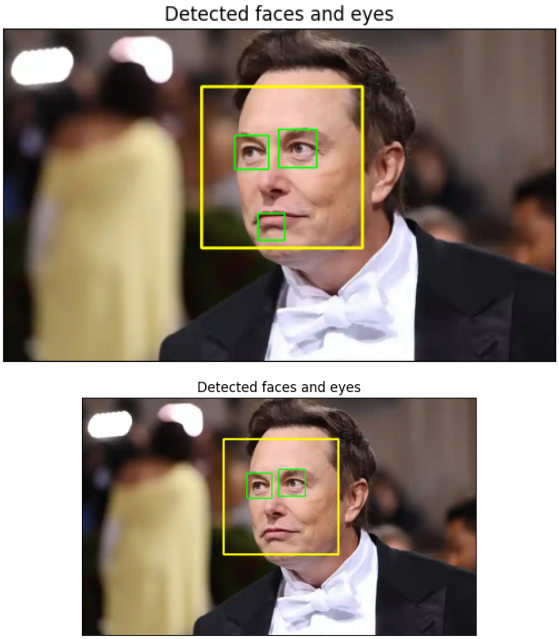


Figure 4.10. Parts of the Face with false-positive detection

As can be seen, some false-positive eye positions were detected. To address this issue we can modify the parameters of the cascade (the minimum required number of matches or the scale factor). However, it can be noticed that all false-positives are located at the bottom part of the face, whereas as we all know eyes are always located at the top 2/3 of the face, so we can modify the ROI definition taking this into account and search for eyes only in the top 2/3 part.

Let us do it as well.

```
I2out_eyes = I2out.copy()
I2faces_eyes = []

# Go through all faces
for i in range(len(I2faces)):
    # Take face and convert it to grayscale and normalize
    face = I2faces[i][1]
    face = cv2.cvtColor(face, cv2.COLOR_BGR2GRAY)
    face = cv2.equalizeHist(face)

    # Slice the face to take the top 2/3 of a face
    face_roi = face[0 : face.shape[0] * 2 // 3, :]

    # Then search for eyes
    I2eyes = cascade_eyes.detectMultiScale(face_roi, minNeighbors=10)
    print("{} eyes were detected on face {}".format(len(I2eyes), i + 1))

    # Now go through all eyes
    I2face_out = I2faces[i][1].copy()
    for (x, y, w, h) in I2eyes:
        # Highlight them on the face
        cv2.rectangle(I2face_out, (x, y, w, h), (0, 255, 255), thickness=1)
        # And on the source image
        # For this we have to shift eyes by face box position
        # Since we use the top part of an image, the ROI zero point and face zero points are the same
        cv2.rectangle(I2out_eyes,
                      (x + I2objects[i][0], y + I2objects[i][1], w, h),
```

```
                        (255, 0, 0), thickness=2)

    I2faces_eyes.append((I2faces[i][0],
    I2face_out))

# Display it
ShowImages([("Detected faces and eyes",
    I2out_eyes)], 2)
ShowImages(I2faces_eyes, 5)
```

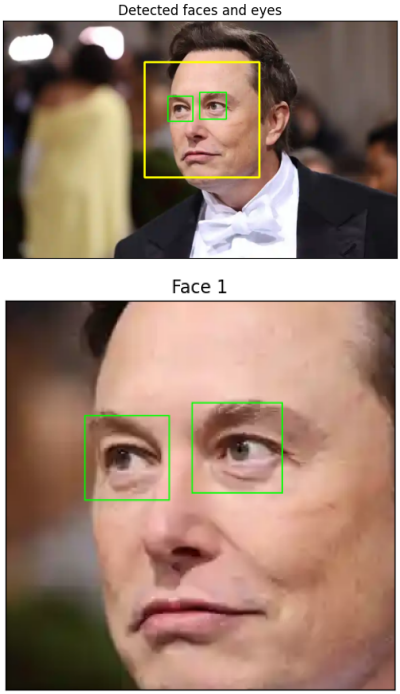


Figure 4.11. Parts of the Face without false-positive detection

The same approach can be used when detecting other parts of the face, for example, the mouth is located at the bottom 1/3 of the face, so you can define the ROI for mouth detection by slicing the bottom part of the face image matrix.

4.4 Procedure of Practical Assignment Performing

All tasks are based on the material presented in the corresponding sections of the assignment. Students are expected to write code for each task using Python, OpenCV, and NumPy.

Task 1. Faces Detection

1. **Viola-Jones Face Detection**

   Take three arbitrary images containing several faces. Try to use images with a different number of faces and different scales. Perform a search for faces using the Viola-Jones approach. Calculate the number of faces found on each image. Display the detected faces on the images.

   Hints for implementation:

   - Use `cv2.CascadeClassifier` for cascade classifier execution.
   - Load the cascade from an XML file: `cascade_face.load(cv.samples.findFile(filename))`.
   - Use the default built-in cascade: `haarcascade_frontalface_default.xml`.
   - Apply detection using `detectMultiScale(image, scaleFactor, minNeighbors, flags, minSize, maxSize)`.
     - `scaleFactor = 1.1` (default scale factor);
     - `minNeighbors = 3` (default threshold for detection);
     - `minSize, maxSize` (optional window size limits).
   - Iterate through the returned objects array (list of rectangles) to highlight faces using `cv2.rectangle()`.

- You may try using alternative cascades (e.g., `haarcascade_frontalface_alt.xml`, `haarcascade_profileface.xml`).
- **Note:** Convert images to grayscale and equalize histogram (`cv2.equalizeHist()`) for better processing quality.

Task 2. Body Parts Detection

1. **Multi-part Body Detection with ROI**

   Take three arbitrary images containing several faces. Try to use images with a different number of faces and different scales. Perform a search of at least two parts of bodies in one image (e.g., eyes, mouths, noses). To increase the accuracy use ROI (e.g., upper part of bodies or faces). Calculate the found elements in each category.

   Hints for implementation:

   - First detect faces using the Viola-Jones approach (as in Task 1).
   - Define Region-Of-Interest (ROI) for each detected face by slicing the image array: `roi = image[y : y + h, x : x + w]`.
   - Load cascades for body parts:
     - `haarcascade_eye.xml` (eyes detection);
     - `haarcascade_smile.xml` (mouth/smile detection);
     - `haarcascade_nose_default.xml` (if available).
   - Search for parts within the face ROI to reduce false positives.
   - **For eyes:** Search in the top 2/3 of the face ROI (`face_roi = face[0: face.shape[0] * 2 // 3, :]`).
   - **For mouth:** Search in the bottom part of the face ROI.
   - When highlighting parts on the source image, shift coordinates by the face box position (`x + face_x, y + face_y`).
   - **Note:** Remember that numpy array indexing is `image[y, x]`, so slicing is `image[y:y+h, x:x+w]`.

Task 3. Optional: Face Detection in Video

Implement face detection in a video stream using a pre-recorded video with faces.

OpenCV provides methods to work with video streams as a set of images. For this, you first have to create an instance of the `cv2.VideoCapture` class.

The following class methods can be used:

- `cv2.VideoCapture.open()` → `retval` — method is called in the constructor; it allows opening a new video stream (returns `True` or `False` — if the stream was opened);
- `cv2.VideoCapture.isOpened()` → `retval` — method allows checking if the video was successfully loaded (returns `True` or `False` — if the stream was opened);
- `cv2.VideoCapture.grab()` → `retval` — method allows grabbing the next frame for processing to retrieve it (returns `True` or `False` — if grabbing succeeds);
- `cv2.VideoCapture.retrieve()` → `retval, image` — method allows retrieving the next frame after it finished processing (returns a tuple with success `True` or `False` flag and the next frame `image`);
- `cv2.VideoCapture.read()` → `retval, image` — method grabs and retrieves the next frame in a single call (returns a tuple with success `True` or `False` flag and the next frame `image`);
- `cv2.VideoCapture.release()` — method releases the video capture device. This function is called automatically in the object destructor or a subsequent opening of a new stream.

In terms of the current task, you may read video frames with the `read()` function. However, to get a slightly higher performance you can use the combination of `grab()` and `retrieve()` functions.

To display the live video frames, we will use the same OpenCV GUI library we used in the first practical assignment. It will open a separate window and Jupyter cell will keep running until the video window is closed or an ESC button is pressed.

```
# To open a video from a file just pass the video file name to
# the VideoCapture class constructor
video = cv2.VideoCapture("images/video.mp4")

# Check if a video was opened
if not video.isOpened():
    print("Error opening video capture")
    exit()

# If a video was opened then grab the next frame to process
video.grab()

# Create a window
window_name = "Video stream"
cv2.namedWindow(window_name)

# Wait for an ESC key press or window to be closed
while True:
    # Unlike the cv2.waitKey, the cv2.pollKey() function checks
    # if a key has been pressed without introducing a delay
    if cv2.waitKey(1) == 27:
        break
    if cv2.getWindowProperty(window_name, cv2.WND_PROP_VISIBLE) < 1:
        break

    # While the video is still shown update the window contents
    ret, I = video.retrieve()
    if ret:
        last_frame = I
        cv2.imshow(window_name, I)
        video.grab()
```

```
    else:
        break

# And destroy all information about the window
    we used
cv2.destroyAllWindows()

# Let us show the last frame here
ShowImages([("Video frame", last_frame)])
```

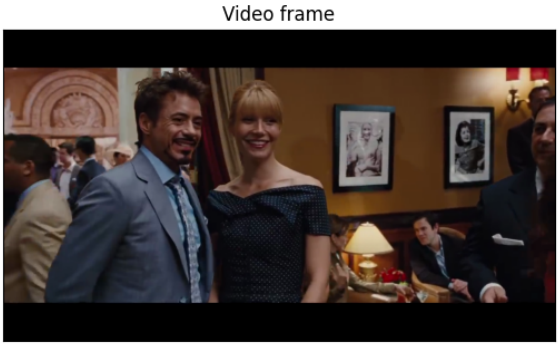


Figure 4.12. Frame

Another option to view the video through base64 encoding, for easy viewing.

```
from IPython.display import display, HTML
import base64

def show_video(video_path):
    video_file = open(video_path, "rb").read()
    video_url = f"data:video/mp4;base64,{
    base64.b64encode(video_file).decode()}"
    return HTML(f"""<video width=500 controls
    ><source src="{video_url}"></video>""")

```

```
show_video("images/video_students.mp4")
```

Figure 4.13. Video Frame

If you want to get your video via webcam, you can do it inthis way (if you wish). To connect to a camera and process the live video stream you must open the device by its ID. The first video capture device has ID 0, the second one's ID is 2, and so on. For example, connecting to the first camera would be written as follows:

```
# Connect to a camera
video = cv2.VideoCapture(0)
```

Note that in the aboveexample, the video was not processed. In the framework of this task, you must add video processing frame-by-frame and highlight found faces. The Viola-Jones face detection algorithm works quite fast and can be implemented even on low-performance embedded systems hardware, so try and tune the classifier parameters to achieve real-time performance. The `scaleFactor` and window size range are the parameters that affect the algorithm performance most.

General Requirements

- All results must be visualized using the `ShowImages()` function from the `pa_utils.py` module.
- Code must be well-structured and commented.

- Each task must be implemented in a separate Jupyter notebook cell or section.
- Answers the questions and write a conclusion.

Questions to Practical Assignment Report Defense

- What is the special image representation used in the Viola-Jones approach?
- What is the main advantage of Haar-like features for classifier training?
- Can you use Viola-Jones approach for detecting arbitrary objects and if so, why?

List of references

[1] Otsu Nobuyuki. A threshold selection method from gray-level histograms // IEEE Transactions on Systems, Man, and Cybernetics. — 1979. — Vol. 9, no. 1. — P. 62–66.

[2] CIE. Colorimetry. — 3rd ed. — Vienna : CIE Central Bureau, 2004.

[3] MacQueen James. Some methods for classification and analysis of multivariate observations // Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability. — 1967. — P. 281–297.

[4] Haralick Robert M., Shanmugam K., Dinstein Its'hak. Textural features for image classification // IEEE Transactions on Systems, Man, and Cybernetics. — 1973. — Vol. SMC-3, no. 6. — P. 610–621.

[5] Method and means for recognizing complex patterns Hough Paul V. C. ; General Motors Corporation ; — no. 460936.

[6] Duda Richard O., Hart Peter E. Use of the Hough transformation to detect lines and curves in pictures // Communications of the ACM. — 1972. — Vol. 15, no. 1. — P. 11–15.

[7] Lowe David G. Distinctive image features from scale-invariant keypoints // International Journal of Computer Vision. — 2004. — Vol. 60, no. 2. — P. 91–110.

[8] ORB: An efficient alternative to SIFT or SURF / Rublee Ethan, Rabaud Vincent, Konolige Kurt, and Bradski Gary // Proceedings of the IEEE International Conference on Computer Vision (ICCV). — 2011. — P. 2564–2571.

[9] BRIEF: Binary robust independent elementary features / Calonder Michael, Lepetit Vincent, Strecha Christoph, and Fua Pascal // European Conference on Computer Vision (ECCV). — 2010. — P. 778–792.

[10] Fischler Martin A., Bolles Robert C. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography // Communications of the ACM. — 1981. — Vol. 24, no. 6. — P. 381–395.

[11] Viola Paul, Jones Michael. Robust real-time face detection // International Journal of Computer Vision. — 2004. — Vol. 57, no. 2. — P. 137–154.

[12] Freund Yoav, Schapire Robert E. A decision-theoretic generalization of on-line learning and an application to boosting // Journal of Computer and System Sciences. — 1997. — Vol. 55, no. 1. — P. 119–139.

[13] Harris Chris, Stephens Mike. A combined corner and edge detector // Alvey Vision Conference / Citeseer. — 1988. — Vol. 15. — P. 10–5244.

[14] Shi Jianbo. Good features to track // 1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition. — IEEE. — 1994. — P. 593–600.

[15] Bay Herbert, Tuytelaars Tinne, Van Gool Luc. SURF: Speeded up robust features // European Conference on Computer Vision. — Springer. — 2006. — P. 404–417.

[16] Viswanathan Deepak Geetha. Features from accelerated segment test (FAST) // Proceedings of the 10th Workshop on Image Analysis for Multimedia Interactive Services. — London, UK. — 2009. — P. 6–8.

[17] Viola Paul, Jones Michael. Robust real-time object detection // International Journal of Computer Vision. — 2001. — Earlier version presented at CVPR 2001.

Сергей Васильевич Шаветов
Андрей Дмитриевич Жданов
Олег Александрович Евстафьев
Владимир Владимирович Беспалов

Компьютерное зрение на Python

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49