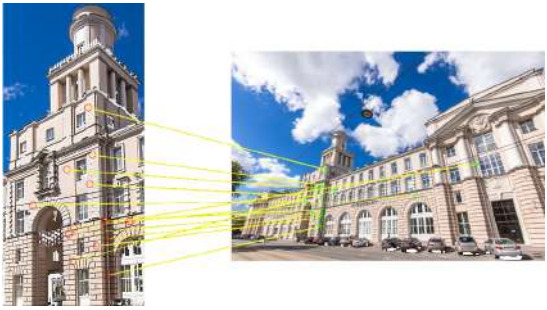
ITMO

Sergei Shavetov & Andrei Zhdanov

COMPUTER VISION



St. Petersburg
2022

MINISTRY OF SCIENCE AND HIGHER EDUCATION
OF THE RUSSIAN FEDERATION

ITMO UNIVERSITY

S.V. Shavetov, A.D. Zhdanov

COMPUTER VISION

STUDY GUIDE

RECOMMENDED FOR USE
AT ITMO UNIVERSITY

within the Master's Program 15.04.06 Robtics and AI

St. Petersburg
2022

S.V. Shavetov, A.D. Zhdanov. Computer Vision. — St. Petersburg: ITMO University, 2022. — 115 p.

Reviewer:
V.S. Gromov, PhD, Control Systems and Robotics Faculty, ITMO University, St. Petersburg, Russia.

The study guide is devoted to the basics of computer vision. Four practical assignments of increasing complexity are presented, ranging from images segmentation to face detection. The study guide is intended for master students studying in 15.04.06 Mechatronics and Robotics.

Contents

Practical Assignment №1
Images Segmentation

Objective

Study of the basic methods for images segmentation into semantic areas.

Guidelines

Before getting started, students should be familiar with the basic functions of the MATLAB [1] or OpenCV [2] for working on of color spaces transformation of images, and methods for determining thresholds. Practical assignment is designed for 4 hours.

Brief Theory

Images Binarization

The simplest way to segment an image into two classes (background and object pixels) is *binarization*. Binarization can be performed by thresholding or by double thresholding. In the first case:

$$I_{new}(x,y) = \begin{cases} 0, I(x,y) \leqslant t, \\ 1, I(x,y) > t, \end{cases} \tag{1.1}$$

where I — source image, I_{new} — binarized image, t — binarization threshold. Binarization by this method in the MATLAB can be done using functions `im2bw()` (old) or `imbinarize()`.

Listing 1.1. Binarization with MATLAB.

```
1   I = imread('pic.jpg');
2   L = 255;
3   t = 127 / L; %norm to 0...1
4   Inew = im2bw(I, t);
```

In OpenCV thresholding is performed with `cv::threshold()` function in C++ and `cv2.threshold()` function in Python. It takes an image to process, threshold parameter, the value to set for thresholded image pixels and thresholding method as its parameters. Simple

threshold binarization according to formula (1.1) is performed by using `THRESH_BINARY` method.

Listing 1.2. Binarization with OpenCV and C++.

```
cv::Mat I;
I = cv::imread("pic.jpg",
  cv::IMREAD_GRAYSCALE);
double t = 127;
cv::Mat Inew;
cv::threshold(I, Inew, t, 255,
  cv::THRESH_BINARY);
```

Listing 1.3. Binarization with OpenCV and Python.

```
I = cv2.imread("pic.jpg",
  cv2.IMREAD_GRAYSCALE)
t = 127
ret, Inew = cv2.threshold(I, t, 255,
  cv2.THRESH_BINARY)
```

Double threshold binarization (range binarization):

$$I_{new}(x,y) = \begin{cases} 0, I(x,y) \leqslant t_1, \\ 1, t_1 < I(x,y) \leqslant t_2, \\ 0, I(x,y) > t_2, \end{cases} \tag{1.2}$$

where I — source image, I_{new} — binarized image, t_1 and t_2 — upper and lower binarization thresholds. Binarization by this method in the MATLAB can be done using function `roicolor()`. To convert a color image to grayscale one, you can previously use the function `rgb2gray()`.

Listing 1.4. Double threshold binarization with MATLAB.

```
I = imread('pic.jpg');
t1 = 110;
t2 = 200;
Igray = rgb2gray(I);
Inew = roicolor(Igray, t1, t2);
```

OpenCV does not provide special function for double thresholding, however it can be done by subsequent call of two threshold functions with different methods. First would set all values above `t2` to zeros

(THRESH_TOZERO_INV method), and second would set all values below t1 to zeros as well (THRESH_BINARY method).

Listing 1.5. Double threshold binarization with OpenCV and C++.

```
cv::Mat I;
I = cv::imread("pic.jpg", cv::IMREAD_COLOR);
double t1 = 127;
double t2 = 200;
cv::Mat Igray, Inew;
cv::cvtColor(I, Igray, cv::COLOR_BGR2GRAY);
cv::threshold(Igray, Inew, t2, 255,
  cv::THRESH_TOZERO_INV);
cv::threshold(Inew, Inew, t1, 255,
  cv::THRESH_BINARY);
```

Listing 1.6. Double threshold binarization with OpenCV and Python.

```
I = cv2.imread("pic.jpg", cv2.IMREAD_COLOR)
t1 = 127
t2 = 200
Igray = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)
ret, Inew = cv2.threshold(Igray, t2, 255,
  cv2.THRESH_TOZERO_INV)
ret, Inew = cv2.threshold(Inew, t1, 255,
  cv2.THRESH_BINARY)
```

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

$$t = \frac{I_{max} - I_{min}}{2}. \tag{1.3}$$

2. Finding the optimal threshold t based on the modulus of the each pixel brightness gradient. For this, the modulus of the gradient is

first calculated at each point (x,y):

$$G(x,y) = \max\left\{|I(x+1,y) - I(x-1,y)|, |I(x,y+1) - I(x,y-1)|\right\}, \tag{1.4}$$

then the optimal threshold t is calculated:

$$t = \frac{\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} I(x,y)G(x,y)}{\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} G(x,y)}. \tag{1.5}$$

3. Calculation of the optimal threshold t by statistical method Otsu [3] which divides all pixels into two classes 1 and 2. This method minimizing the variance within each class $\sigma_1^2(t)$ and $\sigma_2^2(t)$, and maximizing the variance between classes.

The algorithm for calculating the threshold by the Otsu method:

1. Computing an image histogram of intensities, and probability $p_i = \dfrac{n_i}{N}$ for each intensity level, where n_i — number of pixels with intensity level i, N — the number of pixels in the image.

2. Setting 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. In the loop for each value of threshold from $k = 1$ to $k = L - 1$:

   (a) Computing probabilities of two classes $\omega_j(0)$, and arithmetic mean $\mu_j(0)$, where $j = \overline{1,2}$:

$$\omega_1(k) = \sum_{s=0}^{k} p_s, \tag{1.6}$$

$$\omega_2(k) = \sum_{s=k+1}^{L} p_s = 1 - \omega_1(k), \tag{1.7}$$

$$\mu_1(k) = \sum_{s=0}^{k} \frac{s \cdot p_s}{\omega_1}, \tag{1.8}$$

$$\mu_2(k) = \sum_{s=k+1}^{L} \frac{s \cdot p_s}{\omega_2}. \tag{1.9}$$

(b) Interclass variance calculation $\sigma_b^2(k)$:

$$\sigma_b^2(k) = \omega_1(k)\omega_2(k)(\mu_1(k) - \mu_2(k))^2. \tag{1.10}$$

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

3. Optimal threshold t corresponds to the maximum $\sigma_b^2(k)$.

Threshold t by Otsu method in the MATLAB can be done using function `graythresh()`:

Listing 1.7. Binarization by the Otsu method with MATLAB.

```
I = imread('pic.jpg');
t = graythresh(I);
Inew = im2bw(I, t);
```

or using the `otsuthresh()` function based on the image histogram:

Listing 1.8. Binarization by the Otsu method based on the histogram with MATLAB.

```
I = imread('pic.jpg')
Igray = rgb2gray(I);
[counts,x] = imhist(Igray);
t = otsuthresh(counts);
Inew = imbinarize(Igray, t);
```

In OpenCV thresholding by Otsu is performed with the same `cv::threshold()` function in C++ and `cv2.threshold()` function in Python. To use the Otsu thresholding method you have to use the `THRESH_OTSU` parameter as a thresholding method.

Listing 1.9. Binarization by the Otsu method based on the histogram with OpenCV and C++.

```
cv::Mat I;
I = cv::imread("pic.jpg", cv::IMREAD_COLOR);
cv::Mat Igray, Inew;
cv::cvtColor(I, Igray, cv::COLOR_BGR2GRAY);
cv::threshold(Igray, Inew, 0, 255,
  cv::THRESH_OTSU);
```

Listing 1.10. Binarization by the Otsu method based on the histogram with OpenCV and Python.

```
I = cv2.imread("pic.jpg", cv2.IMREAD_COLOR)
Igray = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)
ret, Inew = cv2.threshold(Igray, 0, 255,
  cv2.THRESH_OTSU)
```

4. Adaptive methods that do not work with the entire image, but only with its fragments. Such approaches are often used when working with images that represent non-uniformly illuminated objects. Threshold t by the adaptive method in the MATLAB can be calculated using the `adaptthresh()` function:

Listing 1.11. Binarization by adaptive method with MATLAB.

```
I = imread('pic.jpg');
Igray = rgb2gray(I);
t = adaptthresh(Igray);
Inew = imbinarize(Igray, t);
```

In OpenCV adaptive thresholding is performed with the `cv::adaptiveThreshold()` function in C++ and `cv2.adaptiveThreshold()` function in Python. It supports two adaptive thresholding algorithms: `ADAPTIVE_THRESH_MEAN_C` for simple rectangular kernel mean and `ADAPTIVE_THRESH_GAUSSIAN_C` for a kernel with Gauss weights. Kernel size is defined by `blockSize` parameter (is set to `11` in the following example).

Listing 1.12. Binarization by adaptive method with OpenCV and C++.

```
cv::Mat I;
I = cv::imread("pic.jpg", cv::IMREAD_COLOR);
cv::Mat Igray, Inew;
cv::cvtColor(I, Igray, cv::COLOR_BGR2GRAY);
cv::adaptiveThreshold(Igray, Inew, 255,
  cv::ADAPTIVE_THRESH_GAUSSIAN_C,
  cv::THRESH_BINARY, 11, 2);
```

Listing 1.13. Binarization by adaptive method with OpenCV and Python.

```
I = cv2.imread("pic.jpg", cv2.IMREAD_COLOR)
```

```
2   Igray = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)
3   Inew = cv2.adaptiveThreshold(Igray, 255,
4     cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
5     cv2.THRESH_BINARY, 11, 2)
```

In addition to the considered methods, there are many others, for example, the methods of Bernsen, Eikwell, Niblack, Yanowitz and Brookstein, etc.

Images Segmentation

Let's consider several basic methods of image segmentation.

Algorithm based on the Weber principle

The algorithm is designed for segmentation of grayscale images. *Weber principle* assumes that the human eye does not perceive well the difference in gray levels between $I(n)$ and $I(n) + W(I(n))$, where $W(I(n))$ — Weber function, n — class number, I — piecewise nonlinear grayscale function. The Weber function can be calculated using the formula:

$$W(I) = \begin{cases} 20 - \dfrac{12I}{88}, 0 \leqslant I \leqslant 88, \\ 0{,}002(I - 88)^2, 88 < I \leqslant 138, \\ \dfrac{7(I - 138)}{117} + 13, 138 < I \leqslant 255. \end{cases} \tag{1.11}$$

You can merge grayscale levels from a range $[I(n),I(n) + W(I(n))]$ replacing them with a single intensity value.

The segmentation algorithm consists of the following steps:

1. Initialization initial conditions: first class number $n = 1$, grayscale level $I(n) = 0$.

2. Calculation value $W(I(n))$ according to the Weber formula and value assign $I(n)$ to all pixels whose intensity is in the range $[I(n),I(n) + W(I(n))]$.

3. Search for pixels with intensity higher $G = I(n)+W(I(n))+1$. If found, then increase the class number $n = n+1$, assign $I(n) = G$ and go to the second step. Otherwise, finish the algorithm. The image will be segmented into n classes with the intensity $W(I(n))$.

Segmentation of RGB images by skin color

The general principle of this approach is to determine the criterion for the proximity of the pixels intensity to the skin tone. It is quite difficult to describe *skin tone* analytically, since its description is based on human perception of color, changes with lighting, differs among different 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 if the following conditions are met at uniform day light illumination:

$$\begin{cases} R > 95, \\ G > 40, \\ B > 20, \\ \max R,G,B - \min R,G,B > 15, \\ |R - G| > 15, \\ R > G, \\ R > B, \end{cases} \tag{1.12}$$

or under flash light or daylight lateral illumination:

$$\begin{cases} R > 220, \\ G > 210, \\ B > 170, \\ |R - G| \leqslant 15, \\ G > B, \\ R > B, \end{cases} \tag{1.13}$$

or using normalized RGB values:

$$\begin{cases} r = \frac{R}{R+G+B}, \\ g = \frac{G}{R+G+B}, \\ b = \frac{B}{R+G+B}, \\ \frac{r}{g} > 1{,}185, \\ \frac{rb}{(r+g+b)^2} > 0{,}107, \\ \frac{rg}{(r+g+b)^2} > 0{,}112. \end{cases} \tag{1.14}$$

Algorithm based on CIE Lab color space

In the **Lab** color space [4], the value of lightness is separated from the value of the chromatic components of the color (hue, saturation). The lightness is given by the $\boldsymbol{L}$ coordinate, which can range from 0 (dark) to 100 (light). The chromatic component of a color is given by two Cartesian coordinates $\boldsymbol{a}$ (means the color position in the range from *green* (−128) to *red* (127)) and $\boldsymbol{b}$ (means the color position in the range *blue* (−128) to *yellow* (127)). A binary image is obtained with zero coordinates $\boldsymbol{a}$ and $\boldsymbol{b}$. The algorithm general idea is to divide a color image into segments of dominant colors.

Let's choose the following color image as the initial data:

Figure 1.1. Original color image

First of all, in order to reduce the illumination effect on the segmentation result, we transform a color image from the **RGB** color space to the **Lab** space. This transformation in MATLAB can be performed by the `rgb2lab()` function.

Listing 1.14. Segmentation based on **Lab** color space with MATLAB.

```
I = imread('pic2.jpg');
Ilab = rgb2lab(I);
L = Ilab(:,:,1);
a = Ilab(:,:,2);
b = Ilab(:,:,3);
```

The next step is to determine the number of colors into which the image will be segmented, and define areas containing pixels of approximately the same color. Regions can be interactively defined for each color as polygons using the `roipoly()` function:

```
numColors = 3;
```

```
7   sampleAreas = false([size(I, 1)
8     size(I, 2) numColors]);
9   for i=1:1:numColors
10    [BW, xi, yi] = roipoly(I);
11    sampleAreas(:,:,i) = roipoly(I, xi, yi);
12  end
```

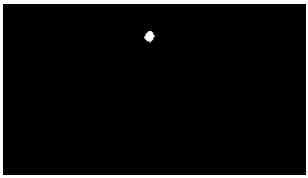


Figure 1.2. An example of a highlighted red area of four points

After that, it is required to determine the color marks for each segment by calculating the average color value in each selected area. Mean values can be calculated using the `mean2()` function:

```
13  colorMarks = zeros([numColors, 2]);
14  for i=1:1:numColors
15    colorMarks(i,1) = ...
16      mean2(a(sampleAreas(:,:,i)));
17    colorMarks(i,2) = ...
18      mean2(b(sampleAreas(:,:,i)));
19  end
```

Then we use the principle of nearest neighborhood to classify pixels by calculating Euclidean metrics between pixels and marks: the smaller the distance to the mark, the better the pixel fits this segment. The Euclidean metric for two color coordinates is calculated by the formula: $\sqrt{(a(x,y) - a_{mark})^2 + (b(x,y) - b_{mark})^2}$. To find the minimum distance, we will use the `min()` function. Here is a listing for finding segment labels `label` for each pixel:

```
20  distance = zeros([size(a), numColors]);
21  colorLabels = zeros([1,numColors]);
22  for i = 1:1:numColors
23    distance(:, :, i) = ...
```

```
24        ((a - colorMarks(i, 1)).^2 + ...
25        (b - colorMarks(i, 2)).^2).^0.5;
26      colorLabels(:, i) = i;
27    end
28    [~, label] = min(distance, [], 3);
29    label = colorLabels(label);
```

Thus, the matrix `label` of dimension equal to the original image will contain class identifiers for each pixel. To segment an image into `segmentedFrames` fragments, use the following listing:

```
30    rgbLabel = repmat(label, [1 1 3]);
31    segmentedFrames = ...
32    zeros([size(I), numColors], 'uint8');
33    for i=1:1:numColors
34      color = I;
35      color(rgbLabel ~= colorLabels(i)) = 0;
36      segmentedFrames(:,:,:,i) = color;
37    end
```

a) b)

Figure 1.3. Segmented areas: a) red, b) yellow

Place the distribution of segmented pixels on the coordinate plane (a,b):

```
38    plotColors = {'red', 'green', 'yellow'};
39    figure
40    for i=1:1:numColors
41      plot(a(label == i), b(label==i), ...
42        '.', 'MarkerEdgeColor', ...
43        plotColors{i}, ...
44        'MarkerFaceColor', plotColors{i});
```

```
45      axis on, axis normal, hold on
46    end
47    title('Segmented Pixels Distribution');
```

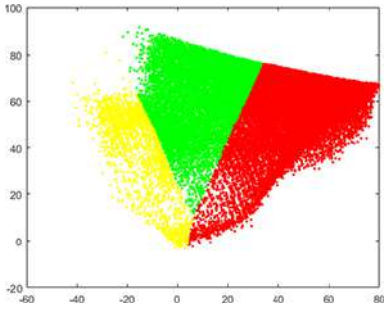


Figure 1.4. Distribution of segmented pixels on a coordinate plane (a,b)

The same algorithm can be implemented with OpenCV libraries and C++ programming language. First to select points in the image we have to define the mouse callback function [5]. The `cv::EVENT_LBUTTONDBLCLK` is processed to take only double clicks into an account.

Listing 1.15. Mouse callback function definition with OpenCV and C++.

```
1   void MouseHandler(int event, int x, int y,
2        int flags, void *param)
3   {
4     if (event != cv::EVENT_LBUTTONDBLCLK)
5       return;
6     if (param == NULL)
7       return;
8     ((std::vector<cv::Vec2i> *)param)->
9       push_back(cv::Vec2i(x, y));
10  }
```

Conversion from **BGR** to **Lab** color space can be performed by `cv::cvtColor()` function with parameter `cv::COLOR_BGR2LAB`. The resulting images after each algorithm step are the same with MATLAB

implementation, so they are omitted in the following examples with OpenCV.

Please note that opposite ot MATLAB OpenCV uses **BGR** to store color images. So we should use **BGR2LAB** converter.

Listing 1.16. Segmentation based on **Lab** color space with OpenCV and C++.

```
cv::Mat I;
I = cv::imread("pic2.jpg",
  cv::IMREAD_COLOR);
// Convert to CIE Lab
cv::Mat Ilab0;
cv::cvtColor(I, Ilab0, cv::COLOR_BGR2Lab);
// Split an image into layers
std::vector<cv::Mat> Ibgr, Ilab;
cv::split(I, Ibgr);
cv::split(Ilab0, Ilab);
// L = Ilab[0]
// a = Ilab[1]
// b = Ilab[2]
```

To define target segmentation colors we can redefine the OpenCV image viewer mouse callback function with the `cv::setMouseCallback()` function. Then we have to call the events processing for 20 milliseconds by using the `cv::waitKey(20)` function until the desired number of points is accumulated in an array. After the desired number of points is collected the callback can be removed by passing an empty callback function to the `cv::setMouseCallback()` function. By defining the different callback behavior we can define different type of regions. Alternatively the `cv::selectROI()` function can be used to select a single rectangular region of interest and `cv::selectROIs()` function to select multiple rectangular regions of interest.

```
cv::imshow("Image", I);
std::vector<cv::Vec2i> sampleAreas;
cv::setMouseCallback("Image", MouseHandler,
  &sampleAreas);
while (sampleAreas.size() < 3)
  cv::waitKey(20);
```

```
20   cv::setMouseCallback("Image", NULL);
```

To calculate the mean color for each of selected image points we can use OpenCV `cv::mean()` function and circular mask which can be created by drawing a white circle with `cv::circle()` function on a black background. Means are calculated both for **Lab** and **BGR** color spaces. **BGR** one would be needed to create a pixel color distribution plot later on.

```
21   std::vector<cv::Vec2d> colorMarks;
22   std::vector<cv::Vec3b> colorMarksBGR;
23   for (int i = 0; i < sampleAreas.size(); i++)
24     {
25     cv::Mat mask =
26       cv::Mat::zeros(Ilab[0].rows,
27       Ilab[0].cols, CV_8U);
28     cv::circle(mask, sampleAreas[i], 10,
29       cv::Scalar(255), -1);
30     cv::Scalar a = cv::mean(Ilab[1], mask);
31     cv::Scalar b = cv::mean(Ilab[2], mask);
32     colorMarks.push_back(
33       cv::Vec2d(a[0], b[0]));
34     cv::Scalar B = cv::mean(Ibgr[0], mask);
35     cv::Scalar G = cv::mean(Ibgr[1], mask);
36     cv::Scalar R = cv::mean(Ibgr[2], mask);
37     colorMarksBGR.push_back(
38       cv::Vec3b((uchar)B[0], (uchar)G[0],
39       (uchar)R[0]));
40     }
```

Next, for each pixel an Euclidean distance between the pixel color in (a, b) color space and previously selected averaged colors is calculated.

```
41   std::vector<cv::Mat> distance;
42   for (int i = 0; i < colorMarks.size(); i++)
43     {
44     cv::Mat tmp, tmp2;
45     cv::subtract(Ilab[1], colorMarks[i][0],
46       tmp, cv::noArray(), CV_64F);
47     cv::multiply(tmp, tmp, tmp);
```

```
48      cv::subtract(Ilab[2], colorMarks[i][1],
49        tmp2, cv::noArray(), CV_64F);
50      cv::multiply(tmp2, tmp2, tmp2);
51      cv::sqrt(tmp + tmp2, tmp);
52      distance.push_back(tmp);
53      }
```

After that the minimum distance is estimated for each image pixel.

```
54    cv::Mat distance_min = distance[0].clone();
55    for (int i = 1; i < distance.size(); i++)
56      cv::min(distance_min, distance[i],
57        distance_min);
```

When all distances are calculated, it became possible to calculate labels by finding the distance image id which is equal to a minimum for each image pixel.

```
58    cv::Mat labels =
59      cv::Mat::zeros(Ilab[0].rows, Ilab[0].cols,
60      CV_8U);
61    for (int i = 0; i < colorMarks.size(); i++)
62      {
63      cv::Mat mask = distance[i] == distance_min;
64      labels.setTo(cv::Scalar(i), mask);
65      }
```

After the `labels` for each of the image pixel are calculated it became possible to segment the source image to a set of segmented images stored in `segmentedFrames`.

```
66    std::vector<cv::Mat> > segmentedFrames;
67    for (int i = 0; i < colorMarks.size(); i++)
68      {
69      cv::Mat Itmp = cv::Mat::zeros(I.rows,
70        I.cols, I.type());
71      I.copyTo(Itmp, labels == i);
72      segmentedFrames.push_back(Itmp);
73      }
```

Finally, we can place distribution of image pixel colors to a plot in the (a, b) coordinate system. The plot pixel color is defined by a mean **BGR** which was calculated before and stored to `colorMarksBGR` array.

```
74   cv::Mat Iplot(256, 256, CV_8UC3,
75     cv::Scalar(255, 255, 255));
76   for (int i = 0; i < colorMarks.size(); i++)
77     {
78     cv::Mat Itmp =
79       cv::Mat::zeros(I.rows,
80       I.cols, I.type());
81     Mat mask = labels == i;
82     for (int x = 0; x < mask.cols; x++)
83       for (int y = 0; y < mask.rows; y++)
84         if (mask.at<uchar>(y, x) != 0)
85           Iplot.at<cv::Vec3b>(
86             Ilab[1].at<uchar>(y, x),
87             Ilab[2].at<uchar>(y, x)) =
88             colorMarksBGR[i];
89     }
```

The same algorithm can be implemented with OpenCV libraries and Python programming language. Conversion from **BGR** to **Lab** color space can be performed by `cv2.cvtColor()` function with parameter `cv2.COLOR_BGR2LAB`.

Please note that opposite ot MATLAB OpenCV uses **BGR** to store color images. So we should use **BGR2LAB** converter.

Listing 1.17. Segmentation based on **Lab** color space with OpenCV and Python.

```
1   I = cv2.imread("pic2.jpg", cv2.IMREAD_COLOR)
2   Ilab = cv2.cvtColor(I, cv2.COLOR_BGR2LAB)
3   Ilab = cv2.split(Ilab)
4   # L = Ilab[0]
5   # a = Ilab[1]
6   # b = Ilab[2]
```

To define target segmentation colors we can redefine the OpenCV image viewer mouse callback function with the `cv2.setMouseCallback()` function. Then we have to call the

events processing for 20 milliseconds by using the `cv2.waitKey(20)` function until the desired number of points is accumulated in an array. The `cv2.EVENT_LBUTTONDBLCLK` is processed to take only double clicks into an account. After the desired number of points is collected the callback can be removed by passing an empty callback function to the `cv2.setMouseCallback()` function. By defining the different callback behavior we can define different type of regions. Alternatively the `cv2.selectROI()` function can be used to select a single rectangular region of interest and `cv2.selectROIs()` function to select multiple rectangular regions of interest.

```
 7   sampleAreas = []
 8   def MouseHandler(event, x, y, flags, param):
 9     if event != cv2.EVENT_LBUTTONDBLCLK:
10       return
11     sampleAreas.append((x, y))
12
13   cv2.imshow("Image", I)
14   cv2.setMouseCallback('Image', MouseHandler)
15   while len(sampleAreas) < 3:
16     cv2.waitKey(20)
17   cv2.setMouseCallback('Image',
18     lambda *args : None)
```

To calculate the mean color for each of selected image points we can use NumPy `mean()` function and circular region of interest which can be created by drawing a white circle with `cv2.circle()` function on a black background. Means are calculated both for **Lab** and **BGR** color spaces. **BGR** one would be needed to create a pixel color distribution plot later on.

```
19   colorMarks = []
20   colorMarksBGR = []
21   for pix in sampleAreas:
22     mask = np.zeros_like(Ilab[0])
23     cv2.circle(mask, pix, 10, 255, -1)
24     a = Ilab[1].mean(where = mask > 0)
25     b = Ilab[2].mean(where = mask > 0)
26     colorMarks.append((a, b))
27     colorMarksBGR.append(
```

```
28        I[mask > 0, :].mean(axis=(0)))
```

Next, for each pixel an Euclidean distance between the pixel color in (a, b) color space and previously selected averaged colors is calculated. After that the minimum distance is estimated for each image pixel.

```
29    distance = []
30    for color in colorMarks:
31      distance.append(np.sqrt(
32        np.power(Ilab[1] - color[0], 2) +
33        np.power(Ilab[2] - color[1], 2)))
34    distance_min = np.minimum.reduce(distance)
```

When all distances are calculated, it became possible to calculate labels by finding the distance image id which is equal to a minimum for each image pixel.

```
35    labels = np.zeros_like(Ilab[0],
36      dtype = np.uint8)
37    for i in range(len(colorMarks)):
38      mask = distance_min == distance[i]
39      labels[mask] = i
```

After the `labels` for each of the image pixel are calculated it became possible to segment the source image to a set of segmented images stored in `segmentedFrames`.

```
40    segmentedFrames = []
41    for i in range(len(colorMarks)):
42      Itmp = np.zeros_like(I)
43      mask = labels == i
44      Itmp[mask] = I[mask]
45      segmentedFrames.append(Itmp)
```

Finally, we can place distribution of image pixel colors to a plot in the (a, b) coordinate system. The plot pixel color is defined by a mean **BGR** which was calculated before and stored to `colorMarksBGR` array.

```
46    Iplot = np.full((256, 256, 3), 255,
47      dtype = np.uint8)
48    for i in range(len(colorMarks)):
49      Itmp = np.zeros_like(I)
```

```
50      mask = labels == i
51      Iplot[Ilab[1][mask], Ilab[2][mask], :] = \
52        colorMarksBGR[i]
```

Algorithm based on 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 determining k vectors $m_j, j = \overline{1,k}$, which are declared as initial centers of clusters.
2. Updating mean values of the vectors m_j by calculating 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 average values m_j across all clusters.
3. Repetition of steps 2 and 3 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 (using original image fig. 1.1). We will work in the **Lab** color space, so the first step is transformation from the **RGB** space to **Lab**:

Listing 1.18. Segmentation based k-means clustering method with MATLAB.

```
1   I = imread('pic2.jpg');
2   Ilab = rgb2lab(I);
3   L = Ilab(:,:,1);
4   a = Ilab(:,:,2);
5   b = Ilab(:,:,3);
```

Consider the coordinate plane (a,b). Let's form a three-dimensional array `ab`, and then use the `reshape()` function to turn it into a two-dimensional vector containing all the pixels of the image:

```
6   ab(:,:,1) = a;
7   ab(:,:,2) = b;
8   nrows = size(I, 1);
9   ncols = size(I, 2);
10  ab = reshape(ab, nrows * ncols, 2);
```

Clustering by the k-means method in the MATLAB can by realized by the `kmeans()` function. Similarly to the previous method, we divide the image into three areas of the corresponding colors. We use the Euclidean metric (parameter `'distance'` with value `'sqEuclidean'` and repeat the clustering procedure three times (parameter `'Replicates'` with value `3`) to improve accuracy):

```
11   k = 3;
12   [ids, centers] = kmeans(ab, k, 'distance',...
13     'sqEuclidean', 'Replicates', 3);
14   label = reshape(ids, nrows, ncols);
```

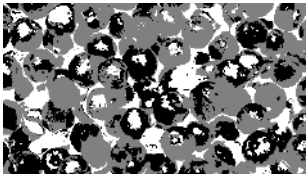


Figure 1.5. Classes labels

The `label` matrix of size equal to the original image will contain class identifiers for each pixel. To segment an image into `segmentedFrames` fragments, use the following listing:

```
15   segmentedFrames = cell(1, 3);
16   rgbLabel = repmat(label, [1 1 3]);
17   for i = 1:1:k
18     color = I;
19     color(rgbLabel ~= i) = 0;
20     segmentedFrames{i} = color;
21     figure, imshow(segmentedFrames{i});
22   end
```

The `L` array contains the lightness value of the image. Using this data, for example, segmented red areas can be divided into light red and dark red segments.

The same algorithm can be implemented with OpenCV libraries and C++ programming language. To convert from **BGR** to **Lab** color space the `cv::cvtColor()` function can be used. Then image is split

a) b)

Figure 1.6. Segmented areas: a) red, b) yellow

into layers using `cv::split()` function. The resulting images after each algorithm step are the same with MATLAB implementation, so they are omitted here.

Listing 1.19. Segmentation based k-means clustering method with OpenCV and C++.

```
1   cv::Mat I;
2   I = cv::imread("pic2.jpg",
3     cv::IMREAD_COLOR);
4   cv::Mat Ilab0;
5   cv::cvtColor(I, Ilab0, cv::COLOR_BGR2Lab);
6   std::vector<cv::Mat> Ilab;
7   cv::split(Ilab0, Ilab);
8   // L = Ilab[0]
9   // a = Ilab[1]
10  // b = Ilab[2]
```

By merging a and b layers of Lab representation of our source image with `cv::merge()` function and reshaping by using the `cv::Mat::reshape()` function we create the two-dimensional array of the image pixel colors. Function `cv::convertTo()` is used to convert reshaped array to float values (parameter `CV_32F`.

```
11  cv::Mat ab;
12  cv::merge(&(Ilab[1]), 2, ab);
13  ab = ab.reshape(0, 1);
14  ab.convertTo(ab, CV_32F);
```

This two-dimensional array can be then passed to `cv::kmeans()` function that performs the k-means clustering. Parameter allows us

to define stop criteria (object of `cv::TermCriteria` type) and number of attempts to select a set of starting points. In the following code stop criteria is defined as not more than 10 iterations of difference between steps less than 1. Starting points are selected randomly (due to `cv::KMEANS_RANDOM_CENTERS` is used) and selection is done 10 times. After algorithm execution is finished, the returned `labels` parameter is reshaped back to an original image shape.

```
15   int k = 3;
16   cv::Mat labels;
17   cv::kmeans(ab, k, labels,
18     cv::TermCriteria(cv::TermCriteria::EPS +
19     cv::TermCriteria::COUNT, 10, 1.0),
20     10, cv::KMEANS_RANDOM_CENTERS);
21   labels = labels.reshape(0, Ilab[0].rows);
```

Then, using the generated labels it is possible to segment our image to a set of images or masks.

```
22   std::vector<cv::Mat> segmentedFrames;
23   for (int i = 0; i < k; i++)
24     {
25     cv::Mat Itmp =
26       cv::Mat::zeros(I.rows, I.cols,
27       I.type());
28     I.copyTo(Itmp, labels == i);
29     segmentedFrames.push_back(Itmp);
30     }
```

The same algorithm can be implemented with OpenCV libraries and Python programming language. To convert from **BGR** to **Lab** color space the `cv2.cvtColor()` function can be used.

Listing 1.20. Segmentation based k-means clustering method with OpenCV and Python.

```
1   I = cv2.imread("pic2.jpg", cv2.IMREAD_COLOR)
2   Ilab = cv2.cvtColor(I, cv2.COLOR_BGR2LAB)
3   Ilab = cv2.split(Ilab)
4   # L = Ilab[0]
5   # a = Ilab[1]
6   # b = Ilab[2]
```

By merging a and b layers of Lab representation of our source image and using the NumPy `reshape()` function we 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 `cv2.kmeans()` function that performs the k-means clustering. Parameter allows us to define stop criteria and number of attempts to select a set of starting points. In the following code stop criteria is defined as not more than 10 iterations of difference between steps less than 1. Starting points are selected randomly (due to `cv2.KMEANS_RANDOM_CENTERS` is 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 +
  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))
```

Then, using the generated labels it is possible to segment our image to 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)
```

Texture segmentation

In texture segmentation, three main methods are used to describe texture: statistical, structural, and spectral. In the practical assignment, we will consider a statistical approach that describes the segment texture as smooth, rough, or grainy. Characteristics of parameters corresponding to textures are given in Table 1.1. Let's consider an example

of an image shown in fig. 1.7, which has two types of textures. Their separation in the general case cannot be performed using only simple binarization.

Figure 1.7. Original grayscale image

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:

$$\mu_n(z) = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i), \tag{1.15}$$

where L — number of intensity levels, m — mean value of a random variable z:

$$m = \sum_{i=0}^{L-1} z_i p(z_i). \tag{1.16}$$

The expression (1.15) 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 introduce a measure of relative smoothness R:

$$R = 1 - \frac{1}{1 + \sigma^2(z)}, \tag{1.17}$$

which is zero for 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

Table 1.1. Texture parameter values

Texture	m	s	$R \in [0,1]$
Smooth	82,64	11,79	0,0020
Rough	143,56	74,63	0,0079
Periodical	99,72	33,73	0,0170

Texture	$\mu_3(z)$	U	E
Smooth	−0,105	0,026	5,434
Rough	−0,151	0,005	7,783
Periodical	0,750	0,013	6,674

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 $(L-1)^2$. *Standard deviation* is also often used as a texture characteristic:

$$s = \sigma(z). \tag{1.18}$$

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

$$E = -\sum_{i=0}^{L-1} p(z_i) \log_2 p(z_i). \tag{1.19}$$

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

$$U = \sum_{i=0}^{L-1} p^2(z_i). \tag{1.20}$$

After calculating any feature or set of features, it is necessary to design a binary mask. This mask is the basis for the image segmentation. For example, you can use the entropy E in the neighborhood of each pixel (x,y). In MATLAB you can use the `entropyfilt()` function, which by default uses a neighborhood of size 9×9. To normalize the entropy function in the range $[0, 1]$, we use the `mat2gray()` function. For mask creation we should binarize the resulting normalized array `Eim` using the Otsu method.

Listing 1.21. Texture segmentation with MATLAB.

```
I = imread('pic3.jpg');
E = entropyfilt(I);
Eim = mat2gray(E);
BW1 = imbinarize(Eim,graythresh(Eim));
```

a) b)

Figure 1.8. a) Entropy of the original image, b) binarized image

After that, we use morphological filters. First, to remove connected areas containing less than a given number of pixels (`bwareaopen()` function). Then to remove internal *form defects* or «holes» (function `imclose()` with a structural element of size 9×9). The remaining large «holes» will be filled using the `imfill()` function. Thus, we get a mask:

```
BWao = bwareaopen(BW1,2000);
nhood = true(9);
closeBWao = imclose(BWao,nhood);
Mask1 = imfill(closeBWao,'holes');
```

Applying the resulting mask to the original image, select the segments of water and land.

We calculate the border between textures using the perimeter function `bwperim()`:

```
boundary = bwperim(Mask1);
segmentResults = I;
segmentResults(boundary) = 255;
```

A similar approach can be applied to create a mask relative to land:

```
I2 = I;
```

a) b)

Figure 1.9. a) Result image after applying `bwareaopen()` function; b) result image after applying `imclose()` function

a) b)

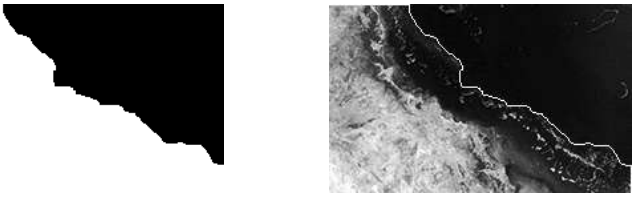
Figure 1.10. a) Texture of land, b) texture of water

```
13   I2(Mask1) = 0;
14   E2 = entropyfilt(I2);
15   E2im = mat2gray(E2);
16   BW2 = imbinarize(E2im, graythresh(E2im));
17   Mask2 = bwareaopen(BW2,2000);
18   boundary = bwperim(Mask2);
19   segmentResults = I;
20   segmentResults(boundary) = 255;
```

Let's find the textures of land and water:

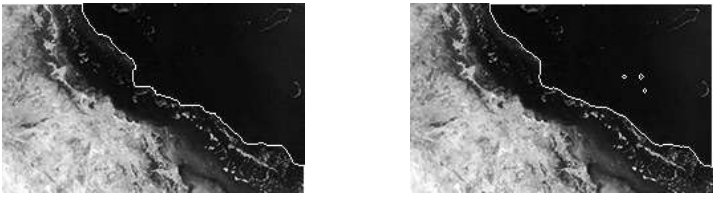
```
21   texture1 = I;
22   texture1(~Mask2) = 0;
23   texture2 = I;
24   texture2(Mask2) = 0;
```

Let us do the same segmentation with OpenCV and C++ programming language. First we have to calculate entropy of the image.



a) b)

Figure 1.11. a) Result image after applying `imfill()` function, b) highlighted border by function `bwperim()`



a) b)

Figure 1.12. a) Result of segmentation relative to water, b) Result of segmentation relative to land

OpenCV does not provide the corresponding function, so have to implement this method by oneself. Please refer to Appendix 1.3 for an example source codes with implementation of MATLAB's `entropyfilt()` function with OpenCV and C++ programming language. The resulting images after each algorithm step are the same with MATLAB implementation, so they are omitted here.

Listing 1.22. Texture segmentation with OpenCV and C++.

```
1   cv::Mat I;
2   I = cv::imread("pic3.jpg",
3     cv::IMREAD_GRAYSCALE);
4   cv::Mat E, Eim;
5   cv::Mat el = cv::getStructuringElement(
6     cv::MORPH_RECT, cv::Size(9, 9));
7   entropy(I, E, el);
8   double Emin, Emax;
```

```
 9   cv::minMaxLoc(E, &Emin, &Emax);
10   Eim = (E - Emin) / (Emax - Emin);
11   Eim.convertTo(Eim, CV_8U, 255);
12   cv::Mat BW1;
13   cv::threshold(Eim, BW1, 0, 255,
14     cv::THRESH_OTSU);
```

Next, the morphological filtering is performed with three steps: at first, remove connected regions (equivalent to MATLAB's `bwareaopen()` function), secondly, remove internal defects with closing operation (executed by `cv::morphologyEx()` function with `cv2.MORPH_CLOSE` parameter) and rectangular structure element of size 9×9 (created by `cv::getStructuringElement()` function with shape parameter `cv::MORPH_RECT`), and, thirdly, fill remaining lagle «holes» (equivalent to MATLAB's `imfill('holes')` function).

MATLAB's `bwareaopen(A, dim)` and `imfill(I, 'holes')` can be easily implemented using OpenCV's `connectedComponentsWithStats()`. Please refer to Appendix 1.1 and Appendix 1.2 for example source codes with OpenCV and C++ or Python programming language implementations of these functions.

```
15   cv::Mat BWao, closeBWao, Mask1;
16   bwareaopen(BW1, BWao, 2000);
17   cv::Mat nhood = cv::getStructuringElement(
18     cv::MORPH_RECT, cv::Size(9, 9));
19   cv::morphologyEx(BWao, closeBWao,
20     cv::MORPH_CLOSE, nhood);
21   imfillholes(closeBWao, Mask1);
```

Next, using OpenCV `cv::findContours()` function it's possible to find shape contours and then draw them on a black background (with `cv::drawContours()` function) to find the contour mask.

```
22   std::vector<std::vector<cv::Point> >
23     contours;
24   cv::findContours(Mask1, contours,
25     cv::RETR_TREE, cv::CHAIN_APPROX_NONE);
26   cv::Mat boundary =
27     cv::Mat::zeros(Mask1.rows, Mask1.cols,
28     CV_8UC1);
```

```
29   cv::drawContours(boundary, contours, -1,
30     255, 1);
```

Then it's possible to apply the found border to the source image.

```
31   cv::Mat segmentResults = I.clone();
32   segmentResults.setTo(cv::Scalar(255),
33     boundary != 0);
```

So, the texture of water was found, and after excluding the water area from the source image we can do the same steps to select the remaining texture of the land.

```
34   cv::Mat I2 = I.clone();
35   I2.setTo(0, Mask1 != 0);
36   // Entropy and binarization
37   cv::Mat E2, Eim2;
38   entropy(I2, E2, element);
39   double Emin2, Emax2;
40   cv::minMaxLoc(E2, &Emin2, &Emax2);
41   Eim2 = (E2 - Emin2) / (Emax2 - Emin2);
42   Eim2.convertTo(Eim2, CV_8U, 255);
43   cv::Mat BW2;
44   cv::threshold(Eim2, BW2, 0, 255,
45     cv::THRESH_OTSU);
46   // Filter
47   cv::Mat BW2ao, closeBW2ao, Mask2;
48   bwareaopen(BW2, BW2ao, 2000);
49   cv::morphologyEx(BW2ao, closeBW2ao,
50     cv::MORPH_CLOSE, nhood);
51   imfillholes(closeBW2ao, Mask2);
52   // Select boundary
53   std::vector<std::vector<cv::Point> >
54     contours2;
55   cv::findContours(Mask2, contours2,
56     cv::RETR_TREE, cv::CHAIN_APPROX_NONE);
57   cv::Mat boundary2 =
58     cv::Mat::zeros(Mask2.rows, Mask2.cols,
59     CV_8UC1);
60   cv::drawContours(boundary2, contours2, -1,
```

```
61      255, 1);
62    cv::Mat segmentResults2 = I2.clone();
63    segmentResults2.setTo(255, boundary2 != 0);
```

And select textures of water and land basing on either of masks.

```
64    cv::Mat texture1 = I.clone();
65    texture1.setTo(0, Mask2 == 0);
66    cv::Mat texture2 = I.clone();
67    texture2.setTo(0, Mask2 != 0);
```

Now let us do the same segmentation with Python programming language. First we have to calculate entropy of the image. OpenCV does not provide the corresponding function, however in case of Python it can be found in SciPy library and is named `skimage.filters.rank.entropy()`. The rectangular with 9×9 kernel created by `skimage.morphology.square()` function will be used. Since SkImage converts image to `float64` we have to do a backward conversion along with normalization to a $[0, 1]$ range.

Listing 1.23. Texture segmentation with OpenCV and Python.

```
1   I = cv2.imread("pic3.jpg",
2     cv2.IMREAD_GRAYSCALE)
3   E = skimage.filters.rank.entropy(I,
4     skimage.morphology.square(9)).astype(
5     np.float32)
6   Eim = (E - E.min()) / (E.max() - E.min())
7   ret, BW1 = cv2.threshold(
8     np.uint8(Eim * 255), 0, 255,
9     cv2.THRESH_OTSU)
```

Next, the morphological filtering is performed with three steps: at first, remove connected regions (equivalent to MATLAB's `bwareaopen()` function), secondly, remove internal defects with 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`), and, thirdly, fill remaining lagle "holes" (equivalent to MATLAB's `imfill('holes')` function).

MATLAB's `bwareaopen(A, dim)` and `imfill(I, 'holes')` can be easily implemented using OpenCV's

connectedComponentsWithStats(). Please refer to Appendix 1.1 and Appendix 1.2 for example source codes with OpenCV and Python programming language implementation for these functions.

```
10   BWao = bwareaopen(BW, 2000)
11   nhood = cv2.getStructuringElement(
12     cv2.MORPH_RECT, (9, 9))
13   closeBWao = cv2.morphologyEx(BWao,
14     cv2.MORPH_CLOSE, nhood)
15   Mask1 = imfillholes(closeBWao)
```

Next, using OpenCV cv2.findContours() function it's possible to find shape contours and then draw them on a black background (with cv2.drawContours() function) to find the contour mask.

```
16   contours, h = cv2.findContours(Mask1,
17     cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
18   boundary = np.zeros_like(Mask1)
19   cv2.drawContours(boundary, contours, -1,
20     255, 1)
```

Then it's possible to apply the found border to the source image.

```
21   segmentResults = I.copy()
22   segmentResults[boundary != 0] = 255
```

So, the texture of water was found, and after excluding the water area from the source image we can do the same steps to select the remaining texture of the land.

```
23   I2 = I.copy()
24   I2[Mask1 != 0] = 0
25   # Entropy and binarization
26   E2 = skimage.filters.rank.entropy(I2,
27     skimage.morphology.square(9)).astype(
28     np.float32)
29   Eim2 = (E2 - E2.min()) / (E2.max() -
30     E2.min())
31   ret, BW2 = cv2.threshold(
32     np.uint8(Eim2 * 255), 0, 255,
33     cv.THRESH_OTSU)
34   # Filter
```

```
35  BW2ao = bwareaopen(BW2, 2000)
36  nhood = cv2.getStructuringElement(
37    cv2.MORPH_RECT, (9, 9))
38  closeBW2ao = cv2.morphologyEx(BW2ao,
39    cv2.MORPH_CLOSE, nhood)
40  Mask2 = imfillholes(closeBW2ao)
41  # Select boundary
42  contours2, h = cv.findContours(Mask2,
43    cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
44  boundary2 = np.zeros_like(Mask2)
45  cv.drawContours(boundary2, contours2, -1,
46    255, 1)
47  segmentResults2 = I2.copy()
48  segmentResults2[boundary2 != 0] = 255
```

And select textures of water and land basing on either of masks.

```
49  texture1 = I.copy()
50  texture1[Mask2 == 0] = 0
51  texture2 = I.copy()
52  texture2[Mask2 != 0] = 0
```

Procedure of Practical Assignment Performing

1. *Binarization.* Choose an arbitrary image. Perform the image binarization using the considered methods. Depending on the image, use upper or lower threshold binarization.

2. *Segmentation 1.* Select an arbitrary image containing the face(s). Perform the image segmentation according to the Weber principle (obligatory). Perform the image segmentation based on the skin color and try different formulas with different photo illumination conditions (optional).

3. *Segmentation 2.* Select an arbitrary image containing a limited number of colored objects. Perform image segmentation in the **CIE Lab** color space by the nearest neighbors method (obligatory). Perform image segmentation in the **CIE Lab** color space by the k-means method (optional).

4. *Segmentation 3.* Select an arbitrary image containing two heterogeneous textures. Perform texture segmentation of the image (obligatory). Evaluate at least three parameters of the selected textures, determine which class the textures belong to (optional).

Note. Please note that when doing the practical assignment you are not allowed to use the *"Lenna"* image or any other image that was used either in this book or during the presentation.

Content of the Report

1. Title page.
2. Objective.
3. Theoretical substantiation of the applied methods and functions.
4. Assignment steps:
   (a) Original images;
   (b) Code of the scripts;
   (c) Comments;
   (d) Resulting images.
5. Conclusion.
6. Answers to questions for the defense.

Questions to Practical Assignment Report Defense

1. When is it appropriate to use Weber segmentation?
2. What are the $\boldsymbol{a}$ and $\boldsymbol{b}$ color coordinates of the **CIE Lab** color space in a grayscale image?
3. What is the reason for performing an image segmentation in the **CIE Lab** color space and not in the original **RGB** one?

Appendix 1.1. MATLAB's `bwareaopen()` implementation with OpenCV

Listing 1.24. Implementing MATLAB's `bwareaopen()` function implementation with OpenCV and C++ programming language.

```
1   // Remove small objects from a binary image
2   // @param[in] A Input image
3   // @param[put] C Output image
4   // @param[in] dim A minimum size of an area
5   //                  to keep
6   // @param[in] conn Pixel connectivity
7   // @return An image with components less
8   //          then dim removed
9   void bwareaopen(const cv::Mat &A,
10    cv::Mat &C, int dim, int conn = 8)
11  {
12    if (A.channels() != 1 ||
13        A.type() != CV_8U)
14      return;
15    // Find all connected components
16    cv::Mat labels, stats, centers;
17    int num =
18      cv::connectedComponentsWithStats(A,
19      labels, stats, centers, conn);
20    // Clone image
21    C = A.clone();
22    // Check size of all connected components
23    std::vector<int> td;
24    for (int i = 0; i < num; i++)
25      if (stats.at<int>(i,
26          cv::CC_STAT_AREA) < dim)
27        td.push_back(i);
28    // Remove small areas
29    if (todel.size() > 0)
30      if (img.type() == CV_8U)
31      {
32        for (int i = 0; i < C.rows; i++)
33          for (int j = 0; j < C.cols; j++)
```

```
34              for (int k = 0; k < td.size();
35                   k++)
36                if (labels.at<int>(i, j) ==
37                    td[k])
38                {
39                  C.at<uchar>(i, j) = 0;
40                  continue;
41                }
42        }
43      else
44        {
45          for (int i = 0; i < C.rows; i++)
46            for (int j = 0; j < C.cols; j++)
47              for (int k = 0; k < td.size();
48                   k++)
49                if (labels.at<int>(i, j) ==
50                    td[k])
51                {
52                  C.at<float>(i, j) = 0;
53                  continue;
54                }
55        }
56    }
```

Listing 1.25. Implementing MATLAB's `bwareaopen()` function implementation with OpenCV and Python programming language.

```
1   # Remove small objects from binary image
2   # @param[in] A Input image
3   # @param[in] dim A mininum size of an area
4   #                  to keep
5   # @param[int] conn Pixel connectivity
6   # @return An image with components less then
7   #          dim removed
8   def bwareaopen(A, dim, conn = 8):
9     if A.ndim != 2 or A.dtype != np.uint8:
10      return None
11    # Find all connected components
12    num, labels, stats, centers = \
```

```
13        cv2.connectedComponentsWithStats(A,
14        connectivity = conn)
15      # Check size of all connected components
16      for i in range(num):
17        if stats[i, cv2.CC_STAT_AREA] < dim:
18          A[labels == i] = 0
19      return A
```

Appendix 1.2. MATLAB's `imfill('holes')` implementation with OpenCV

In the following implementation we assume that the binary image is defined in *uint*8 space using two colors: 0 and 255.

Listing 1.26. Implementing MATLAB's `imfill('holes')` function with OpenCV and C++ programming language.

```
1   // Implementation of MATLAB's
2   // imfill(I, 'holes') function
3   // @param[in] I Image to process
4   // @param[out] Iout Output image
5   void imfillholes(cv::Mat &I, cv::Mat &Iout)
6     {
7     // Check input image data
8     if (I.channels() != 1 ||
9         I.type() != CV_8U)
10      return;
11    cv::Mat mask = I.clone();
12    // Fill mask from all horizontal borders
13    for (int i = 0; i < I.cols; i++)
14      {
15      if (mask.at<uchar>(0, i) == 0)
16        cv::floodFill(mask, cv::Point(i, 0),
17          cv::Scalar(255), NULL,
18          cv::Scalar(10), cv::Scalar(10));
19      if (mask.at<uchar>(I.rows - 1, i) == 0)
20        cv::floodFill(mask,
21          cv::Point(i, I.rows - 1),
22          cv::Scalar(255), NULL,
```

```
23            cv::Scalar(10), cv::Scalar(10));
24        }
25      // Fill mask from all vertical borders
26      for (int i = 0; i < I.rows; i++)
27        {
28        if (mask.at<uchar>(i, 0) == 0)
29          cv::floodFill(mask, cv::Point(0, i),
30            cv::Scalar(255), NULL, cv::Scalar(10),
31            cv::Scalar(10));
32        if (mask.at<uchar>(i, I.cols - 1) == 0)
33          cv::floodFill(mask,
34            cv::Point(I.cols - 1, i), cv::Scalar(255),
35            NULL, cv::Scalar(10), cv::Scalar(10));
36        }
37      // Use the mask to create an  image
38      Iout = I.clone();
39      Iout.setTo(cv::Scalar(255), mask == 0);
40      }
```

Listing 1.27. Implementing MATLAB's `imfill('holes')` function with OpenCV and Python programming language.

```
1   # Implementation of MATLAB's
2   # imfill(I, 'holes') function
3   # @param[in] I Image to process
4   # @return An image with holes removed
5   def imfillholes(I):
6     if I.ndim != 2 or I.dtype != np.uint8:
7       return None
8     rows, cols = I.shape[0:2]
9     mask = I.copy()
10    # Fill mask from all horizontal borders
11    for i in range(cols):
12      if mask[0, i] == 0:
13        cv.floodFill(mask, None,
14          (i, 0), 255, 10, 10)
15      if mask[rows - 1, i] == 0:
16        cv.floodFill(mask, None,
17          (i, rows - 1), 255, 10, 10)
```

```
18      # Fill mask from all vertical borders
19      for i in range(rows):
20        if mask[i, 0] == 0:
21          cv.floodFill(mask, None,
22            (0, i), 255, 10, 10)
23        if mask[i, cols - 1] == 0:
24          cv.floodFill(mask, None,
25            (cols - 1, i), 255, 10, 10)
26      # Use the mask to create a resulting image
27      res = I.copy()
28      res[mask == 0] = 255
29      return res
```

Appendix 1.3. MATLAB's `entropyfilt()` implementation with OpenCV

In the following implementation we assume that the binary image is defined in *uint*8 space using single layer.

Listing 1.28. Implementing MATLAB's `entropyfilt()` function with OpenCV and C++ programming language.

```
1   // Entropy filter
2   // @param[in] I Input image
3   // @param[out] Iout Output image
4   // @param[in] el Structuring element
5   void entropy(cv::Mat &I, cv::Mat &Iout,
6       cv::Mat &el)
7     {
8     // Check input image data
9     if (I.channels() != 1 ||
10        I.type() != CV_8U)
11      return;
12
13    // Convert to image with border
14    cv::Mat Icopy;
15    cv::copyMakeBorder(I, Icopy,
16      int((el.rows - 1) / 2),
17      int(el.rows / 2),
```

```
18        int((el.cols - 1) / 2),
19        int(el.cols / 2), cv::BORDER_REPLICATE);
20
21      // Initialize output image
22      Iout = cv::Mat::zeros(I.rows, I.cols,
23        CV_32F);
24
25      // Initialize local histogram
26      double hist[256];
27      for (int i = 0; i < 256; i++)
28        hist[i] = 0;
29
30      // Calculate element size
31      int count = 0;
32      for (int i = 0; i < el.rows; i++)
33        for (int j = 0; j < el.cols; j++)
34          if (el.at<uchar>(i, j))
35            count++;
36
37      // For each image pixel
38      for (int y = 0; y < I.rows; y++)
39        for (int x = 0; x < I.cols; x++)
40          {
41          // Calculate local histogram
42          for (int i = 0; i < el.rows; i++)
43            for (int j = 0; j < el.cols; j++)
44              if (el.at<uchar>(i, j))
45                hist[Icopy.at<uchar>(
46                  y + i, x + j)] += 1;
47
48          // Calculate entropy
49          double val = 0;
50          for (int i = 0; i < 256; i++)
51            if (hist[i] > 0)
52              {
53              val -= hist[i] / count *
54                log2(hist[i] / count);
55              hist[i] = 0;
```

```
56              }
57          Iout.at<float>(y, x) = float(val);
58          }
59      }
```

Practical Assignment №2
Hough Transform

Objective

Study of the Hough transformation to find of geometric primitives.

Guidelines

Before getting started, students should be familiar with the functions of the MATLAB or OpenCV for working with the Hough transform and know the «voting» points approach. Practical assignment is designed for 4 hours.

Brief Theory

The main principle of the Hough transform [6] is to find common *locus of points*. For example, this approach is used when designing a triangle along three given sides. At first one side of the triangle is first laid off, after that the ends of the segment are considered as the centers of circles with radii equal to the lengths of the second and 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» «winning» was the point that got two «votes» (the points on the circles got one vote each, and outside them — zero).

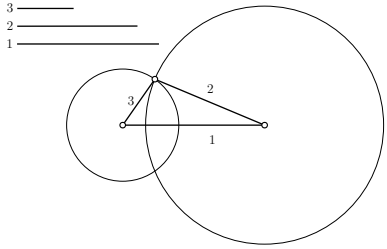


Figure 2.1. Design of a triangle by given three sides

Let's generalize this idea for working with real data, when the image has a large number of special feature points participating in the vote. Let assume that it is necessary to search a circle of 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 circles centers for the desired radius around each characteristic point forms a circle of radius R, see Fig. 2.2. Thus, the point corresponding to the maximum intersection of the number of circles will be the center of the required radius circle.

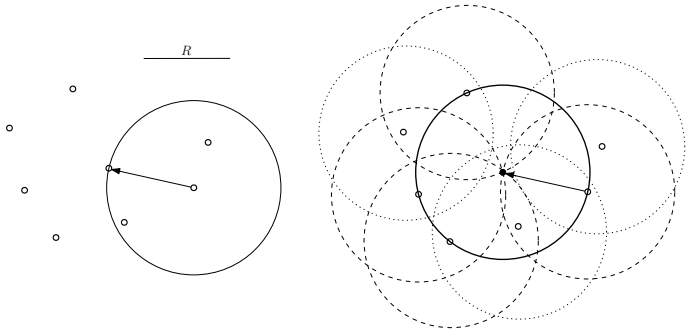


Figure 2.2. Search a circle of known radius in a point set

The classical Hough transform, based on the considered point voting idea, was originally designed to select lines on binary images. The Hough transform uses the parameter space to search for geometric primitives. The most common parametric equation of lines is:

$$y = kx + b, \tag{2.1}$$

$$x \cos \Theta + y \sin \Theta = \rho, \tag{2.2}$$

where ρ — radius vector drawn from the origin to the line; Θ — inclination angle of the radius vector.

Let the straight line in the Cartesian coordinate system be given by the equation (2.1), from which it is easy to calculate the radius vector ρ and angle Θ (2.2). Then in the Hough parameter space the line will be represented by a point with coordinates (ρ_0,Θ_0), see Fig. 2.3.

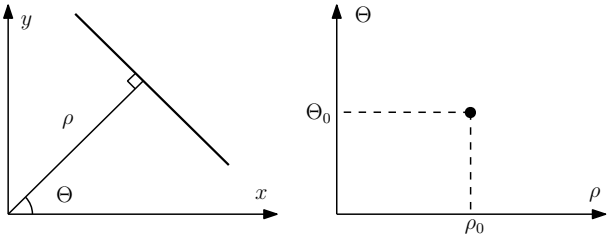


Figure 2.3. Representation of a straight line in Hough space

The Hough transform approach is for each point in the parameter space the number of votes given for it is summed up. Therefore, in discrete form, the Hough space is called *accumulator* and is a matrix $A(\rho,\Theta)$ that stores voting information. Through each point in the Cartesian coordinate system, an infinite number of straight lines can be drawn, the totality of which 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 generating them in the initial space lie on a straight line, see fig. 2.4. Based on this, we can conclude that in order to find straight lines in the original space, it is necessary to find all the local maxima of the accumulator.

The considered line search algorithm 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 $F = (a_1,a_2,...,a_n,x,y)$, which will only affect the dimension of the parameter space. Let us use the Hough transform to search for 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 centers of all possible circles of radius R passing through a feature point forms a circle of radius R around that point. Due to this the response function in the Hough transform for finding circles is a circle of the same size centered at the voting point. 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. Taking into account that under projective transformations of three-

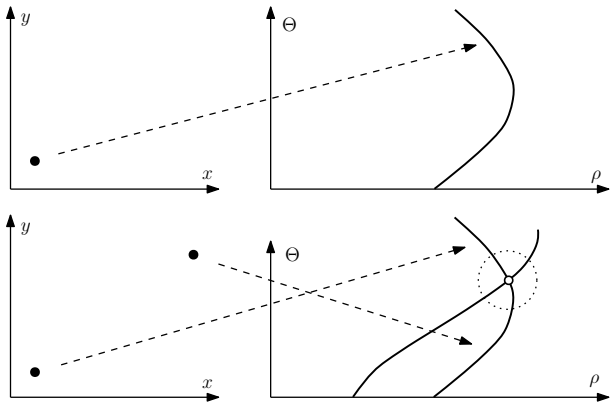


Figure 2.4. Voting procedure

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 invariantly not only to affine transformations of the plane, but also to the group of projective transformations in space.

Let some image be given. Let select the contours using the Canny algorithm and apply the Hough transform using the `hough()` MATLAB function.

Listing 2.1. Search for straight lines by the Hough transform with MATLAB.

```
1   I = imread('pic.png');
2   Iedge = edge(I, 'Canny');
3   [H,Theta,rho] = hough(Iedge);
4   figure, imshow(imadjust(mat2gray(H)),[],...
5     'YData',rho,'XData',Theta,...
6     'InitialMagnification','fit');
7   xlabel('\rho'), ylabel('\Theta')
8   axis on, axis normal, hold on
```

Let's calculate the peaks using the `houghpeaks()` function in the Hough space and plot them on the resulting image of the response functions:

```
9   peaks = houghpeaks(H,100,'threshold',...
```

```
10      ceil(0.5*max(H(:))));
11    x = Theta(peaks(:,2));
12    y = rho(peaks(:,1));
13    plot(x,y,'s','color','white');
```

Based on the peaks, we determine the straight lines using the `houghlines()` MATLAB function and plot them on the original image:

```
14    lines = houghlines(Iedge,Theta,rho,peaks,...
15      'FillGap',5,'MinLength',10);
16    figure, imshow(I), hold on
17    for k = 1:length(lines)
18      xy = [lines(k).point1; lines(k).point2];
19      plot(xy(:,1),xy(:,2),'LineWidth',2,...
20        'Color','green');
21    end
```

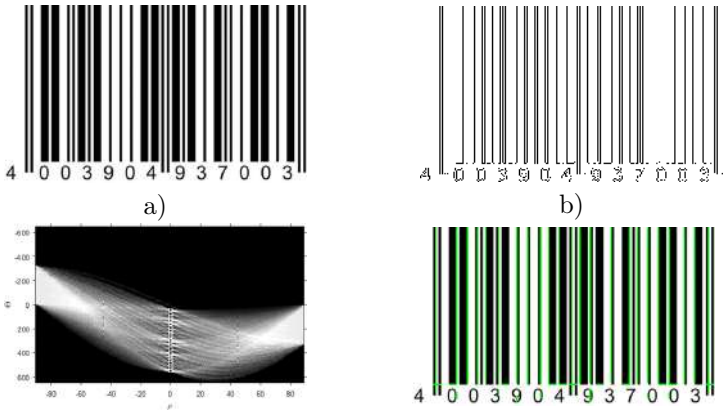


Figure 2.5. a) Source image, b) processed by the Canny algorithm, c) parameter space, d) found lines

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

- `cv::HoughLines(I, lines, rho, theta, threshold)` C++ function (`cv2.HoughLines(I, rho, theta, threshold)` →

`lines` in Python) — perform the classic Hough line transform of an image `I` and find straight lines. `rho` and `theta` parameters define the subdivision of Hough parameter space for corresponding axis. The `threshold` parameter defined 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` and `theta` parameters.

- `cv::HoughLinesP(I, lines, rho, theta, threshold, minLineLength, maxLineGap)` C++ function (`cv2.HoughLinesP(I, rho, theta, threshold, minLineLength, maxLineGap)` → `lines` in Python) — perform the probabilistic Hough line transform of an image `I` and find straight lines. The main function parameters are the same, however two extra are added: these are `minLineLength` — minimum line length, so no line shorter than this won't be selected, and `maxLineGap` — maximum gap between two points of the line to consider them the same line but not two separate ones. The returned `lines` array contains start and end points of each of the found line segments.

To execute the Hough line transform in OpenCV first have to preprocess an image to get edges by executing the Canny algorithm:

Listing 2.2. Search for straight lines by the classic Hough transform with OpenCV and C++ programming language.

```
1   cv::Mat I;
2   I = cv::imread(fn, cv::IMREAD_COLOR);
3   cv::Mat Iedge;
4   cv::Canny(I, Iedge, 50, 200);
```

Then run the Hough line transform to get lines parameters:

```
5   vector<cv::Vec2f> lines;
6   cv::HoughLines(Iedge, lines, 1, M_PI / 180,
7     100);
```

Finally, we can draw infinite lines by returned parameters by calling `cv::line()` function:

```
8   cv::Mat Iout = I.clone();
```

```
 9   for (int i = 0; i < lines.size(); i++)
10   {
11     double rho = lines[i][0];
12     double theta = lines[i][1];
13     double a = cos(theta), b = sin(theta);
14     double x0 = a * rho, y0 = b * rho;
15     cv::Point pt1, pt2;
16     pt1.x = int(x0 - 1000 * b);
17     pt1.y = int(y0 + 1000 * a);
18     pt2.x = int(x0 + 1000 * b);
19     pt2.y = int(y0 - 1000 * a);
20     cv::line(Iout, pt1, pt2,
21       cv::Scalar(0, 255, 0), 1, cv::LINE_AA);
22   }
23   cv::imshow("Classic", Iout);
```

The probabilistic Hough line transform is executed in a similar way:

Listing 2.3. Search for straight lines by the probabilistic Hough transform with OpenCV and C++ programming language.

```
1   vector<cv::Vec4i> linesP;
2   cv::HoughLinesP(Iedge, linesP, 1,
3     M_PI / 180, 50, 50, 4);
```

Then the `linesP` array will hold two points for each of the found line segments which we can use to draw them on top of the source image:

```
4   cv::Mat IoutP = I.clone();
5   for (int i = 0; i < linesP.size(); i++)
6   {
7     cv::Vec4i l = linesP[i];
8     cv::line(IoutP, cv::Point(l[0], l[1]),
9       cv::Point(l[2], l[3]),
10      cv::Scalar(0, 255, 0), 1, cv::LINE_AA);
11  }
12  cv::imshow("Probabilistic", IoutP);
```

Opposite to MATLAB, OpenCV does not allows you to see the Hough parameter space, however the algorithm is rather simple and

can be implemented in the straightforward way as it is shown in the Appendix 2.1.

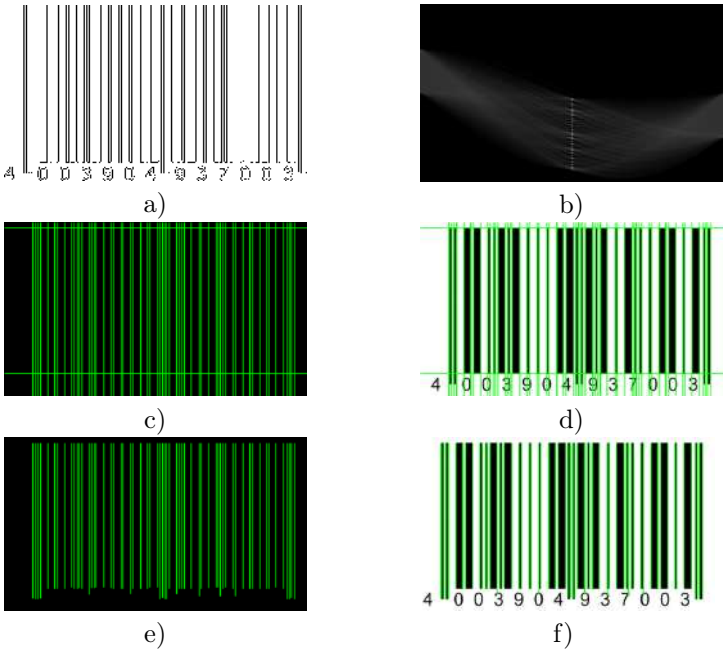


Figure 2.6. a) Source image processed by the Canny algorithm, b) parameter space, c) found lines, d) found lines on top of the source image, e) lines found with probabilistic method, f) lines found with probabilistic method on top of the source image

With Python programming language the Hough line transform can be done in the same way. First have to preprocess an image with Canny algorithm:

Listing 2.4. Search for straight lines by the classic Hough transform with OpenCV and Python programming language.

```
1   I = cv2.imread(fn, cv.IMREAD_COLOR)
2   Iedge = cv2.Canny(I, 50, 200, None, 3)
```

Then execute the Hough line transform:

```
3   lines = cv2.HoughLines(Iedge, 1, np.pi / 180,
4     100)
```

And, finally, display lines:

```
5   Iout = I.copy()
6   if lines is not None:
7     for i in range(0, len(lines)):
8       rho = lines[i][0][0]
9       theta = lines[i][0][1]
10      a, b = math.cos(theta), math.sin(theta)
11      x0, y0 = a * rho, b * rho
12      pt1 = np.int32((x0 - 1000 * b,
13        y0 + 1000 * a))
14      pt2 = np.int32((x0 + 1000 * b,
15        y0 - 1000 * a))
16      cv2.line(Iout, pt1, pt2, (0, 255, 0),
17        1, cv.LINE_AA)
18  cv2.imshow("Classic", Iout)
```

The probabilistic Hough line transform is executed the same way, but instead of infinite line parameters, the function returns line endpoints:

Listing 2.5. Search for straight lines by the probabilistic Hough transform with OpenCV and Python programming language.

```
1   linesP = cv2.HoughLinesP(Iedge, 1,
2     np.pi / 180, 50, None, 50, 4)
3   IoutP = I.copy()
4   if linesP is not None:
5     for i in range(0, len(linesP)):
6       l = linesP[i][0]
7       cv2.line(IoutP, (l[0], l[1]),
8         (l[2], l[3]), (0, 255, 0), 1,
9         cv2.LINE_AA)
10  cv2.imshow("Probabilistic", IoutP)
```

The Hough paramete space can be displayed by using the `skimage.transform.hough_line()` SciKit function:

Listing 2.6. Displaying the Hough transform parameter space with SciKit and Python programming language.

```
angles = np.linspace(-np.pi / 2, np.pi / 2,
  360, endpoint=False)
Ih, theta, rho = \
  skimage.transform.hough_line(Iedge,
  theta = angles)
cv.imshow("Parameter␣space", cv.resize(
  Ih.astype(np.float32) / np.max(Ih),
  (Ih.shape[1], 400)))
```

Then the `skimage.transform.hough_line_peaks()` function can be used to calculate peaks in the `Ih` Hough parameter space to find lines. Please refer to an Appendix 2.2 for the full example of the classic Hough line transform with SciKit library.

To search for circles by the Hough transform, you can use the MATLAB function `[centers, rad] = imfindcircles(I,R)` and function `viscircles(centers, rad)` for plotting circles. If using OpenCV library, you can use the `cv::HoughCircles()` function in C++ and `cv2.HoughCircles()` function in Python. SciKit also provides function for Hough cirlces transform. These are `skimage.transform.hough_circle()` function, which is used to calculate the Hough parameter space, and `skimage.transform.hough_circle_peaks()` function, which searches for peak values in the Hough circles parameter space.

Procedure of Practical Assignment Performing

1. *Search for lines.* Select three arbitrary images containing lines. Perform to search for straight lines using the Hough transform both for the original image and for the image obtained using any differential operator. Plot the found lines on the original image. Mark the start and end points of the lines. Determine the lengths of the shortest and longest lines, calculate the number of lines found.

2. *Search for circles.* Select three arbitrary images containing circles. Search for circles of both a certain radius and from a given range using the Hough transform, both for the original image and for the image obtained using any differential operator. Plot the found circles on the original image.

3. *Optional.* Implement the classic Hough transform algorithms for lines and circles. Compare your implementation results with ones obtained in the first two points of the assignment. Highlight the selected points in the Hough parameter space.

Note. Please note that when doing the practical assignment you are not allowed to use the *"Lenna"* image or any other image that was used either in this book or during the presentation.

Content of the Report

1. Title page.
2. Objective.
3. Theoretical substantiation of the applied methods and functions.
4. Assignment steps:
   (a) Original images;
   (b) Code of the scripts;
   (c) Comments;
   (d) Resulting images.
5. Conclusion.
6. Answers to questions for the defense.

Questions to Practical Assignment Report Defense

1. What is the main principle of the Hough transform?
2. May the Hough transform be used to find arbitrary contours that cannot be described analytically?
3. What are the *recurrent* and *generalized* Hough transforms?
4. What are the ways of parametrization in the Hough transform?

Appendix 2.1. Classic Hough line transform with OpenCV and C++

Listing 2.7. Implementation of Hough transform parameter space calculation for lines with OpenCV and C++ programming language.

```
1   // Transform an image to Hough parameter
2   // space for lines
3   // @param[in]  I The image to transform
4   // @param[out] Ih The image in Hough
5   //             parameter space with 32 bit
6   //             signed integers
7   // @param[in]  thetas The number of theta
8   //             angles to use
9   // @param[in]  rhos The number of rho vlaues
10  //             to use
11  void HoughTransformSpace(cv::Mat &I,
12      cv::Mat &Ih, int thetas = 180,
13      int rhos = 400)
14  {
15    // Check input image data
16    if (I.channels() != 1 ||
17        I.type() != CV_8U)
18      return;
19
20    // Create matrix for Hough parameter space
21    Ih = cv::Mat::zeros(rhos, thetas, CV_32S);
22
23    // Do voting for each image pixel
24    double theta_step = M_PI / thetas;
25    double rho_step =
26      2 * sqrt(I.rows * I.rows +
27      I.cols * I.cols) / rhos;
28    for (int i = 0; i < I.rows; i++)
29    {
30      for (int j = 0; j < I.cols; j++)
31      {
32        if (I.at<uchar>(i, j) == 255)
33        {
```

```
34            // Theta counter is changing
35            // from 0 to thetas
36            // At the same time the angle
37            // changes from -PI/2 to PI/2
38            for (int theta = 0; theta < thetas;
39                 theta++)
40            {
41              // Calculate rho for given theta
42              // We add rows / 2 to get rid of
43              // negative numbers
44              int rho =
45                int((j * cos(theta *
46                     theta_step - M_PI_2) +
47                     i * sin(theta *
48                     theta_step - M_PI_2)) /
49                    rho_step) + Ih.rows / 2;
50              Ih.at<int32_t>(rho, theta)++;
51            }
52          }
53        }
54      }
55    }
```

Listing 2.8. Usage example of the `HoughTransformSpace()` function.

```
1   cv::Mat I;
2   I = cv::imread(fn, cv::IMREAD_COLOR);
3   cv::Mat Iedge;
4   cv::Canny(I, Iedge, 50, 200);
5   cv::Mat Ih;
6   HoughTransformSpace(Iedge, Ih, 360);
7   double Ih_min, Ih_max;
8   cv::minMaxLoc(Ih, &Ih_min, &Ih_max);
9   Ih.convertTo(Ih, CV_32F, 1 / Ih_max);
10  cv::resize(Ih, Ih, cv::Size(I.cols, 400));
11  cv::imshow("Parameter␣space", Ih);
```

Appendix 2.2. Classic Hough line transform with SciKit and Python

Listing 2.9. Implementation of classic Hough line transform with calculation of parameter space using SciKit library and Python programming language.

```
# Read an image from file and preprocess
I = cv.imread(fn, cv.IMREAD_COLOR)
Iedge = cv.Canny(I, 50, 200, None, 3)
Igray = cv.cvtColor(I, cv.COLOR_BGR2GRAY)
# Do Hough transform
angles = np.linspace(-np.pi / 2, np.pi / 2,
  360, endpoint=False)
Ih, theta, rho = \
  skimage.transform.hough_line(Iedge,
  theta = angles)
cv.imshow("Parameter␣space", cv.resize(
  Ih.astype(np.float32) / np.max(Ih),
  (Ih.shape[1], 400)))
# Find lines with Hough transform
Ih, theta, rho = \
  skimage.transform.hough_line_peaks(Ih,
  theta, rho, 0, 0)
# Create and output image
Iout = I.copy()
if theta is not None:
  for i in range(0, len(theta)):
    a = math.cos(theta[i])
    b = math.sin(theta[i])
    x0, y0 = a * rho[i], b * rho[i]
    pt1 = np.int32((x0 - 1000 * b,
      y0 + 1000 * a))
    pt2 = np.int32((x0 + 1000 * b,
      y0 - 1000 * a))
    cv.line(Iout, pt1, pt2, (0, 255, 0), 1,
      cv.LINE_AA)
# And with source colors
cv.imshow("Classic", Iout)
```

Practical Assignment №3
Features Detectors

Objective

Study of feature point detectors and descriptors.

Guidelines

Before getting started, students should be familiar with the functions of the MATLAB or OpenCV for working with the feature points detectors and descriptors. Practical assignment is designed for 4 hours.

Brief Theory

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

Figure 3.1. Building image [7]

As you can see it have 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's not possible to locate 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 CD and D you would also face a problem of locating them since they are located 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 type of points which are easily to locate on an image are called feature points. From formal point of view, feature points can be defined as points that are significantly different from their neighborhood. So, if you move a sliding windows by one pixel from a feature point you would get a completely different image, see Fig. 3.2.

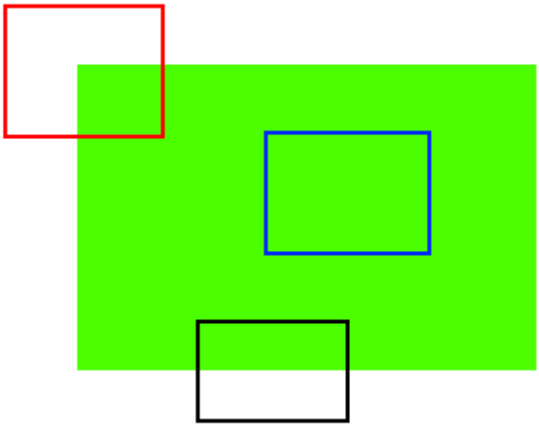


Figure 3.2. Feature points

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

- Harris corner detector [8].
- Shi-Tomasi corner detector [9].

- Scale-Invariant Feature Transform (SIFT) detector and descriptor [10].
- Speeded-Up Robust Features (SURF) detector and descriptor [11].
- Features from Accelerated Segment Test (FAST) detector [12].
- Binary Robust Independent Elementary Features (BRIEF) descriptor [13].
- Oriented FAST and Rotated BRIEF (ORB) detector and descriptor [14].

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

SIFT detector

SIFT stands for Scale-Invariant Feature Transform [10]. The algorithm was patented, however in 2020 the patent has expired, so now it can be used freely in any applications.

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

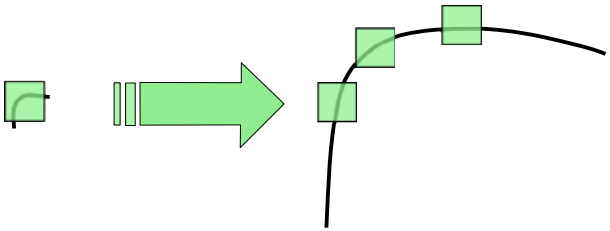


Figure 3.3. Dependence of the corner feature on the scale

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 of the variable-scale Gaussian $G(x, y, \sigma)$ with an input image $I(x, y)$:

$$\begin{aligned} L(x, y, \sigma) &= G(x, y, \sigma) * I(x, y) \\ G(x, y, \sigma) &= \frac{1}{2\pi\sigma^2} e^{\frac{-x^2+y^2}{2\sigma^2}} \end{aligned} \tag{3.1}$$

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 with the following formula with a predefined constant multiplier k by simple image subtraction, see Fig. 3.4:

$$\begin{aligned} D(x, y, \sigma) &= (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) = \\ &= L(x, y, k\sigma) - L(x, y, \sigma) \end{aligned} \tag{3.2}$$

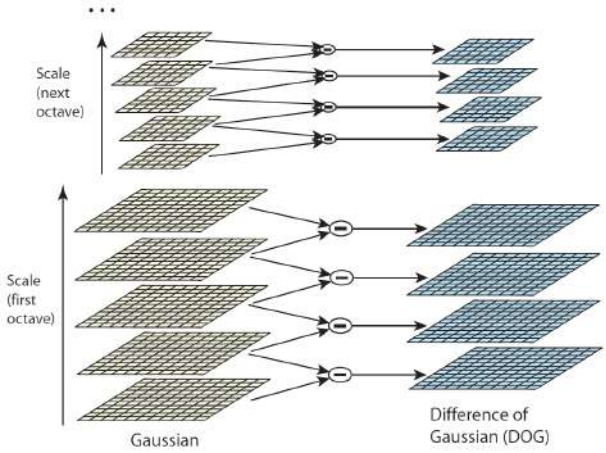


Figure 3.4. Difference of Gaussian calculation

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 it is shown on Fig. 3.5

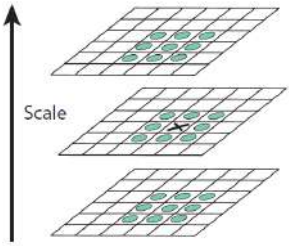


Figure 3.5. Selecting DoG maxima

After an point location and its characteristic scale is found it's location is adjusted according to 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 if several strong maxima are detected, the feature point is considered as several points with different orientations. Histogram contains 36 bins and covers 360°.

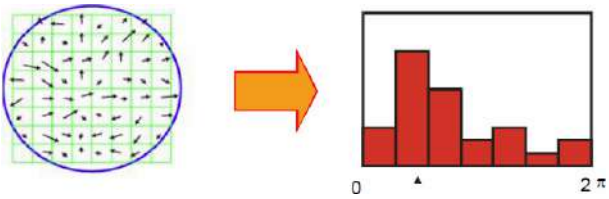


Figure 3.6. Selecting characteristic orientation

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.

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

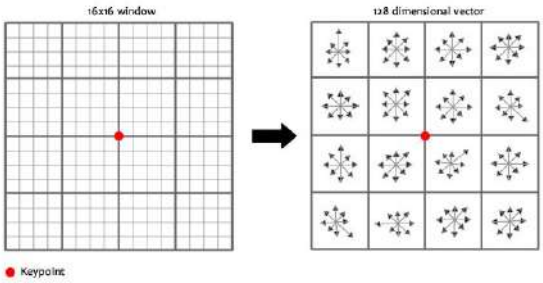


Figure 3.7. SIFT descriptor

SIFT detector with MATLAB

MATLAB provides `detectSIFTFeatures()` function [15] to detect images' features using SIFT algorithm. Typically, this function works with grayscale images.

Listing 3.1. Detecting SIFT feature points with MATLAB.

```
1   I = imread('figure.jpg');
2   Igray = rgb2gray(I);
3   points = detectSIFTFeatures(Igray);
```

Next detected strongest feature points can be displayed using `selectStrongest(points, points_number)` MATLAB function.

Listing 3.2. Displaying 100 strongest SIFT feature points with MATLAB.

```
4   imshow(I);
5   hold on;
6   plot(selectStrongest(points, 100));
```

You can add some extra parameters to display, for example, `scale` and `orientation` of the feature.

Listing 3.3. Displaying 10 strongest SIFT feature points with the scale and orientation with MATLAB.

```
7   figure;
8   imshow(I);
9   hold on;
```

```
10   plot(selectStrongest(points, 10, ...
11     'ShowScale',true, 'showOrientation',true));
```

An example of 100 strongest SIFT feature points are shown in fig. 3.8.

a) b) c)

Figure 3.8. SIFT feature detector with MATLAB: a) Source image, b) 100 strongest SIFT feature points, c) 100 strongest SIFT feature points with the scale and orientation

To obtain the descriptors of feature points you can use `[features, points] = extractFeatures(image, points)` MATLAB function.

Listing 3.4. Obtaining descriptors of feature points.

```
12   [fp,points] = extractFeatures(Igray,points);
```

where variable `fp` consists of descriptors.

SIFT detector with OpenCV

OpenCV provides a C++ class `cv::SIFT` (`cv2.SIFT` in Python) to work with the SIFT feature point detector [16]. An instance of this class can be created by `cv::SIFT::create()` function with C++ and

`cv2.SIFT_create()` function with Python. Constructor allows specifying additional detector parameters, e.g., the first parameter named `nFeatures` allows limiting the number of detected features to a specified number of the most strong feature points. After a class instance is created it provides following functions to work with SIFT detector:

- `cv::SIFT::detect(I, fp, mask)` C++ function (`cv2.SIFT.detect(I, mask)` → `fp` in Python) — detect feature points of the image `I` with region of interest defined by `mask` and store them to the list `fp`.
- `cv::SIFT::compute(I, fp, des)` C++ function (`cv2.SIFT.compute(I, mask)` → `fp, des` in Python) — computes descriptors for a list feature points `fp` of the image `I` and stores them to the list `des`. In case if descriptor can not be calculated it is being removed. If two dominant orientations are found then the feature point is duplicated with two separate descriptors.
- `cv::SIFT::detectAndCompute(I, mask, fp, des)` C++ function (`cv2.SIFT.detectAndCompute(I, mask)` → `fp, des` in Python) — unites function `detect()` and `compute()`. It detect feature points of the image `I` with region of interest defined by `mask`, computes their descriptors and store points to the list `fp` and corresponding descriptors to the list `des`.

With OpenCV the SIFT detector is executed as following:

Listing 3.5. Detecting SIFT feature points with OpenCV and C++.

```
1  cv::Mat I;
2  I = cv::imread("pic.jpg", cv::IMREAD_COLOR);
3  cv::Mat Igray;
4  cv::cvtColor(I, Igray, cv::COLOR_BGR2GRAY);
5  cv::Ptr<cv::SIFT> sift = cv::SIFT::create();
6  std::vector<cv::KeyPoint> Ifp;
7  sift->detect(Igray, Ifp);
```

Listing 3.6. Detecting SIFT feature points with OpenCV and Python.

```
I = cv2.imread("pic.jpg",
  cv2.IMREAD_COLOR)
Igray = cv2.cvtColor(I,
  cv2.COLOR_BGR2GRAY)
sift = cv2.SIFT_create()
Ifp = sift.detect(Igray)
```

To limit the detector to detect only first 100 strongest features it should be specified in the SIFT descriptor constructor:

Listing 3.7. Detecting 100 strongest SIFT feature points with OpenCV and C++.

```
sift = cv::SIFT::create(100);
sift->detect(Igray, Ifp);
```

Listing 3.8. Detecting 100 strongest SIFT feature points with OpenCV and Python.

```
sift = cv2.SIFT_create(100)
Ifp = sift.detect(Igray)
```

Next detected feature points can be displayed using `cv::drawKeypoints(I, fp, Iout, color, flags)` C++ function (`cv2.drawKeypoints(I, fp, Iout, color, flags)` → `Iout` in Python). By default `color` of each feature point is different and only feature point position is displayed.

Listing 3.9. Displaying SIFT feature points with OpenCV and C++.

```
cv::Mat Iout;
cv::drawKeypoints(I, Ifp, Iout);
cv::imshow("SIFT␣detector", Iout);
```

Listing 3.10. Displaying SIFT feature points with OpenCV and Python.

```
Iout = cv2.drawKeypoints(I, Ifp, None)
cv2.imshow("SIFT␣detector", Iout)
```

The optional `flags` parameter value of `cv::DRAW_RICH_KEYPOINTS` (`cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS` in Python) allows drawing feature point size and orientation as well.

Listing 3.11. Displaying SIFT feature points in green color with scale and orientation with OpenCV and C++.

```
cv::Mat Iout;
cv::drawKeypoints(I, Ifp, Iout, color,
 cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
cv::imshow("SIFT␣detector", Iout);
```

Listing 3.12. Displaying SIFT feature points in green color with scale and orientation with OpenCV and Python.

```
Iout = cv2.drawKeypoints(I, Ifp, None,
 color = (0, 255, 0), flags =
 cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow("SIFT␣detector", Iout)
```

An example of 100 strongest SIFT feature points detected with OpenCV are shown in fig. 3.9.

a) b) c)

Figure 3.9. SIFT feature detector with OpenCV: a) Source image, b) 100 strongest SIFT feature points in default colors, c) 100 strongest SIFT feature points with the scale and orientation in green color

ORB detector

ORB detector [14] is a fusion of FAST feature point detector and BRIEF descriptor with many modifications to enhance the detector performance. First it uses FAST detector to find feature points, then applies Harris corner measure to find top N-points among them. It also use 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 located corner at center is calculated. The direction of the vector from this corner point to centroid is considered as the orientation of the feature point. To improve the rotation invariance, moments are computed with x and y axis which should 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 [13] is a bit string description of an image patch constructed from a set of binary intensity tests:

$$\tau(p, x, y) = \begin{cases} 1, p(x) < p(y), \\ 0, p(x) \geqslant p(y). \end{cases} \tag{3.3}$$

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

$$f_n(p) = \sum_{i=1}^{n} 2^{i-1} \tau(p, x_i, y_i). \tag{3.4}$$

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

$$S = \begin{pmatrix} x_1, ..., x_n \\ y_1, ..., y_1 \end{pmatrix}. \tag{3.5}$$

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.

$$S_\theta = R_\theta S. \tag{3.6}$$

ORB quantize the angle to increments of $2\pi/30 = 12\,\text{deg}$, 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.

$$g_n(p, \theta) = f_n(p) | (x_i, y_y) \in S_\theta. \tag{3.7}$$

To compute a distance between two ORB descriptors a Hamming distance can be used. The multi-probe Locality-sensitive hashing (LSH) method is used for ORB descriptor matching.

ORB detector with MATLAB

MATLAB provides the similar to SIFT `detectORBFeatures(image)` function [17]. This function works with grayscale images also. The rest of listings are the same to the listings 3.3–3.4. An example of 100 strongest ORB feature points are shown in fig. 3.10.

ORB detector with OpenCV

OpenCV provides a class for detection of ORB feature points and calculation of corresponding descriptors [18]. The class is named `cv::ORB` in C++ and `cv2.ORB` in Python. A class instance is created with `cv::ORB::create()` method in C++ (`cv2.ORB_create()` in Python). Additional constructor parameters allows modifying descriptor parameters, e.g., the first *nFeatures* parameter specified the number of features to extract from an image. The class interface is the same with SIFT detector and allows detecting feature points and computing their descriptors. With OpenCV the ORB detector for 100 strongest points is executed as following:

Listing 3.13. Detecting and displaying ORB feature points with OpenCV and C++.

```
1   cv::Mat I;
2   I = cv::imread(fn, cv::IMREAD_COLOR);
3   cv::Mat Igray;
4   cv::cvtColor(I, Igray, cv::COLOR_BGR2GRAY);
5   cv::Ptr<cv::ORB> orb;
```

a) b) c)

Figure 3.10. ORB feature detector with MATLAB: a) Source image, b) 100 strongest ORB feature points, c) 100 strongest ORB feature points with the scale and orientation

```
6   orb = cv::ORB::create(100);
7   orb->detect(Igray, Ifp);
8   cv::Mat Iout;
9   cv::drawKeypoints(I, Ifp, Iout, color,
10    cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
11  cv::imshow("ORB␣detector", Iout);
```

Listing 3.14. Detecting and displaying 100 strongest ORB feature points with OpenCV and Python.

```
1   I = cv2.imread("pic.jpg",
2     cv2.IMREAD_COLOR)
3   Igray = cv2.cvtColor(I,
4     cv2.COLOR_BGR2GRAY)
5   orb = cv2.ORB_create(100)
6   Ifp = orb.detect(Igray)
7   Iout = \
```

```
 8    cv2.drawKeypoints(I, Ifp, None,
 9    color = (0, 255, 0), flags =
10    cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
11   cv2.imshow("ORB␣detector", Iout)
```

a) b) c)

Figure 3.11. ORB feature detector with OpenCV: a) Source image, b) 100 strongest ORB feature points in default colors, c) 100 strongest ORB feature points with the scale and orientation in green color

Feature point descriptors matching

The simplest way to match two sets of feature points is by using a brute force method. In this case for each point of the first set an item of the second set is selected which have 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, so the Hamming distance between descriptors should be used instead.

Obviously, the brute force method works slow. To speed up the descriptor matching an accelerating structure should be build on top of the descriptors set, then when matching a descriptor from the search

query it is compared not with whole set, but only with descriptors from some cluster. The simplest accelerating structure is KD-tree (k-dimensional tree) that is built on the space of training set descriptors. In case if descriptor is defined as a binary mask, then it is preferable to use Locality-Sensitive Hashing (LSH) method.

Using the first best match may result in a lot descriptor matches, however a lot of them are false matches due to the fact that some feature points are from repeating pattern (e.g., windows, water, clouds, etc.). There are two possible solution to filter some of the not strong matches.

First solution is to use the cross checking, that requires the descriptor to be matched in two directions: when matching two images it should be the best match in forward and in 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 between first nearest match is significantly lower comparing to distance with the second nearest match:

$$D_1 < r \cdot D_2, \tag{3.8}$$

where D_1 is a distance to the first nearest match, D_2 is a distance to the second nearest match, and r is 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 the cross-checking since the cross-check does not allow more than one match to be found

Using accelerating structures and k-nearest filtering we could get a set of strong matches between images. Since when matching descriptors we did not took the feature point positions into an account, so the next step would be to calculate the geometric transformation between images taking into an account that there may still be lot of outliers or false matches. The most commonly used solution is to use 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 basing on this sample and check how correct this hypothesis is. After checking number of such hypothesis, we choose one that best fits with most of the data.

1. On input we have a set of pairs of matched feature point coordinates on two images: $S = \{(x, y)\}|x \in X, y \in Y$, where X is a first image, and Y is a 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 $S_i = \{(x_i, y_i)\}|(x_i, y_i) \in S$. In our case it is enough to select 4 points from each of X and Y sets to build a matrix M for 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 $R(\theta_I)$:

$$\begin{aligned} R(\theta) &= \sum_{x \in X} p(\theta, x, Y), \\ p(\theta, x, Y) &= \begin{cases} 1, |\varepsilon(\theta, x, Y)| \leqslant T, \\ 0, |\varepsilon(\theta, x, Y)| > T, \end{cases} \end{aligned} \tag{3.9}$$

   where $\varepsilon(\theta, x)$ is the minimum distance from point x to 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. Else, 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.

$$(i = 0) \vee (R(\theta_i) > R(\theta_0)) \Rightarrow \theta_0 = \theta_i. \tag{3.10}$$

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 to 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 following:

$$p = 1 - (1 - N(1 - e)^s)^N, \tag{3.11}$$

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, and e is the ratio of outliers.

Since after estimating at least one hypothesis we can estimate the ratio of outliers, this allows us to estimate required number of iterations basing on the currently best hypothesis:

$$N = \frac{\log(1-p)}{\log(1-(1-e)^s)}. \tag{3.12}$$

The modification of 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 points of the set Y with hypothesis θ:

$$\begin{aligned} R(\theta) &= \sum_{x \in X} p(\theta, x, Y), \\ p(\theta, x, Y) &= \begin{cases} \varepsilon^2(\theta, x, Y), |\varepsilon(\theta, x, Y)| \leqslant T, \\ T^2, |\varepsilon(\theta, x, Y)| > T. \end{cases} \end{aligned} \tag{3.13}$$

Feature point descriptors matching with MATLAB

Let's consider an example, see fig. 3.12. We have two figures: an object and the scene with this object. Scene has some affine geometric transformations.

a)

b)

Figure 3.12. a) Figure 1 — object, b) Figure 2 — scene

At first step of matching, we should calculate descriptors of our figures. See fig. 3.13

a) b)

Figure 3.13. a) An object with 30 strongest features, b) Scene with 100 strongest features

In MATLAB you can use the function `matchFeatures(features1, features2)` for searching pairs of matched features, where `features1` and `features2` are arrays consists of descriptors for corresponding figures (`figure1` and `figure2`, see fig. 3.12). After that, you should take into account only putatively matched points (including outliers) from these figures using array `pairs`.

Listing 3.15. Feature point descriptors matching

```
1   pairs = matchFeatures(features1, features2);
2   matchedPoints1 = points1(pairs(:, 1), :);
3   matchedPoints2 = points2(pairs(:, 2), :);
```

To display matched features in MATLAB you can use built-in function `showMatchedFeatures()` with several parameters, see the listing 3.16 and the result in fig. 3.14.

Listing 3.16. Feature point descriptors matching

```
4   showMatchedFeatures(figure1, figure2, ...
5     matchedPoints1, matchedPoints2, ...
6     'montage');
```

To filter outliers we can estimate the geometric transformation of one figure with respect to the second one. In this step we should use the function `estimateGeometricTransform2D()` with our putatively matched points and specify the type of transformation.

Listing 3.17. Estimation of geometric transformation

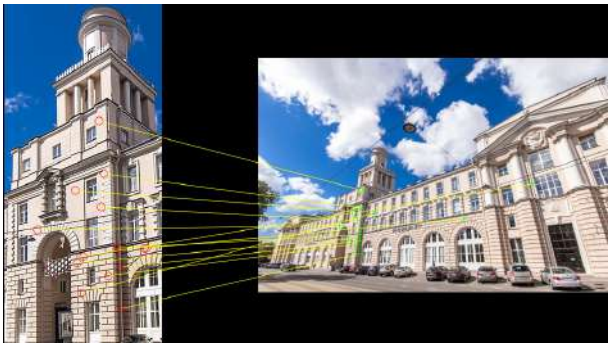


Figure 3.14. Putatively matched points (including outliers)

```
[tform, inliersIds] = ...
  estimateGeometricTransform2D(...
    matchedPoints1, ...
    matchedPoints2, 'affine');
  inlierPoints1 = ...
    matchedPoints1(inliersIds, :);
  inlierPoints2 = ...
    matchedPoints2(inliersIds, :);
```

where `tform` — transformation matrix.

The result of using only inliers is shown in fig. 3.15.

At the final step we can try to find the object (tower from the Figure 1) in the Figure 2 (main building of ITMO University). To perform it let's get the bounding polygon `boxPolygon` of the reference image (Figure 1).

Listing 3.18. Bounding polygon for the object

```
boxPolygon = [1, 1;...                % top-left
  size(I1, 2), 1;...                  % top-right
  size(I1, 2), size(I1, 1);...        % bottom-right
  1, size(I1, 1);...                  % bottom-left
  1, 1];                              % top-left
                    %again to close the polygon
```

Figure 3.15. Matched points (inliers only)

At the next step we should transform the polygon to `newBoxPolygon` using transformation matrix `tform` and function `transformPointsForward()` into the coordinate system of the target image (scene). The transformed polygon indicates the location of the object in the scene.

Figure 3.16. Detected object

Listing 3.19. Display the detected object

```
21    newBoxPolygon = ...
```

```
22      transformPointsForward(tform, boxPolygon);
23    figure;
24    imshow(I2);
25    hold on;
26    line(newBoxPolygon(:, 1), ...
27      newBoxPolygon(:, 2), 'Color', 'r');
```

Feature point descriptors matching with OpenCV

Let's do the same with OpenCV library. So, consider an example from Fig. 3.12, which has two figures: an object (a tower) and the scene with this object (the whole building). Scene has some affine geometric transformations, so our goal is to match the object with the whole scene and find this transformation.

At first have to load images and detect feature points. It's worth using `detectAndCompute()` function to calculate feature point descriptors along with detection. The following code uses SIFT detector to detect feature points and compute descriptors:

Listing 3.20. Detecting and computing SIFT feature points with OpenCV and C++.

```
 1  cv::Mat I1, I2;
 2  I1 = cv::imread("pic1.jpg",
 3    cv::IMREAD_COLOR);
 4  I2 = cv::imread("pic2.jpg",
 5    cv::IMREAD_COLOR);
 6  cv::Mat I1gray, I2gray;
 7  cv::cvtColor(I1, I1gray,
 8    cv::COLOR_BGR2GRAY);
 9  cv::cvtColor(I2, I2gray,
10    cv::COLOR_BGR2GRAY);
11  std::vector<cv::KeyPoint> I1fp, I2fp;
12  cv::Mat I1des, I2des;
13  cv::Ptr<cv::SIFT> sift = cv::SIFT::create();
14  sift->detectAndCompute(I1gray,
15    cv::noArray(), I1fp, I1des);
16  sift->detectAndCompute(I2gray,
17    cv::noArray(), I2fp, I2des);
```

Listing 3.21. Detecting and computing SIFT feature points with OpenCV and Python.

```
I1 = cv2.imread("pic1.jpg",
  cv2.IMREAD_COLOR)
I2 = cv2.imread("pic2.jpg",
  cv2.IMREAD_COLOR)
I1gray = cv2.cvtColor(I1,
  cv2.COLOR_BGR2GRAY)
I2gray = cv2.cvtColor(I2,
  cv2.COLOR_BGR2GRAY)
sift = cv2.SIFT_create()
I1fp, I1des = \
  sift.detectAndCompute(I1gray, None)
I2fp, I2des = \
  sift.detectAndCompute(I2gray, None)
```

a)

b)

Figure 3.17. Detected SIFT feature points. a) An object with default strongest features, b) Scene with default strongest features

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

- `cv::BFMatcher` C++ class (`cv2.BFMatcher` in Python) — brute force matcher. For each feature point descriptor of the first set of points it find the best match in the second set by iterating though all its feature point descriptors.

- `cv::FlannBasedMatcher` C++ class (`cv2.FlannBasedMatcher` in Python) — Fast Library for Approximate Nearest Neighbors matcher. It uses different algorithms for accelerated feature point descriptors matching (KD-trees, k-means, LSH, etc.).

The brute force mathcer can be created either with a cross-check filer or without. This is controlled by the second argument of the class constructor named `crossCheck`. In the following examples it is turned off. The brute force descriptor can be created as following:

Listing 3.22. Creating brute force descriptor matcher with OpenCV and C++.

```
1  cv::Ptr<cv::DescriptorMatcher> matcher;
2  bool crossCheck = false;
3  matcher = cv::BFMatcher::create(NORM_L2,
4    crossCheck);
```

Listing 3.23. Creating brute force descriptor matcher with OpenCV and Python.

```
1  matcher = cv2.BFMatcher(crossCheck = False)
```

This descriptor matcher works well for SIFT descriptors, however for ORB descriptors need to use the Hamming distance measure.

Listing 3.24. Creating brute force descriptor matcher with Hamming distance with OpenCV and C++.

```
1  cv::Ptr<cv::DescriptorMatcher> matcher;
2  bool crossCheck = false;
3  matcher = cv::BFMatcher::create(
4    cv::NormTypes::NORM_HAMMING, crossCheck);
```

Listing 3.25. Creating brute force descriptor matcher with Hamming distance with OpenCV and Python.

```
1  matcher = cv2.BFMatcher(
2    cv2.NORM_HAMMING, crossCheck = False)
```

The FLANN matcher with kd-tree algorithm for SIFT descriptors is created as following:

Listing 3.26. Creating FLANN 5 KD-trees descriptor matcher for SIFT descriptors with OpenCV and C++.

```
cv::Ptr<cv::DescriptorMatcher> matcher;
matcher =
  cv::makePtr<cv::FlannBasedMatcher>(
  cv::makePtr<cv::flann::KDTreeIndexParams>(
  5));
```

Listing 3.27. Creating FLANN 5 KD-trees descriptor matcher for SIFT descriptors with OpenCV and Python.

```
FLANN_INDEX_KDTREE = 1
index_params = \
  dict(algorithm = FLANN_INDEX_KDTREE,
  trees = 5)
matcher = \
  cv2.FlannBasedMatcher(index_params,
  dict())
```

In case of ORB descriptors it is recommended to use LSH algorithm:

Listing 3.28. Creating FLANN LSH descriptor matcher for ORB descriptors with OpenCV and C++.

```
cv::Ptr<cv::DescriptorMatcher> matcher;
matcher =
  cv::makePtr<cv::FlannBasedMatcher>(
  cv::makePtr<cv::flann::LshIndexParams>(
  6, 12, 1));
```

Listing 3.29. Creating FLANN LSH descriptor matcher for ORB descriptors with OpenCV and Python.

```
FLANN_INDEX_LSH = 6
index_params = \
  dict(algorithm = FLANN_INDEX_LSH,
  table_number = 6,  key_size = 12,
  multi_probe_level = 1)
matcher =
cv2.FlannBasedMatcher(index_params, dict())
```

All OpenCV feature point matchers implement the `DescriptorMatcher` interface. It has two main matching functions:

- `match(des1, des2, matches)` C++ method (`match(des1, des2)` → `matches` in Python) — match descriptors `des1` and `des2`. For each descriptor in `des1` a one best match is found in the `des2` set and is returned in `matches` array.

- `knnMatch(des1, des2, matches, k)` C++ method (`match(des1, des2, k)` → `matches` in Python) — match k-nearest descriptors `des1` and `des2`. For each descriptor in `des1` `k` best matches are found in the `des2` set and are returned in `matches` array. Each item of `matches` array is an array containing at maximum `k` matches. 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 more than one match to be found.

Straightforward descriptor matching can be executed as following:

Listing 3.30. Finding single best match for two sets of descriptors with OpenCV and C++.

```
1  std::vector<cv::DMatch> matches;
2  matcher->match(I1des, I2des, matches);
```

Listing 3.31. Finding single best match for two sets of descriptors with OpenCV and Python.

```
1  matches = matcher.match(I1des, I2des)
```

Each match is an instance of `cv::DMatch` class containing following data:

- `queryIdx` — an index of feature point in the first set (`des1`);

- `trainIdx` — an index of feature point in the second set (`des2`);

- `imgIdx` — an index of of image in the second set;

- `distance` — a distance measure between these two descriptors.

As it can be seen from Figures 3.18 — 3.19 using the first best match may result in a lot descriptor matches and a lot of false matches among them. So, it is worth using 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

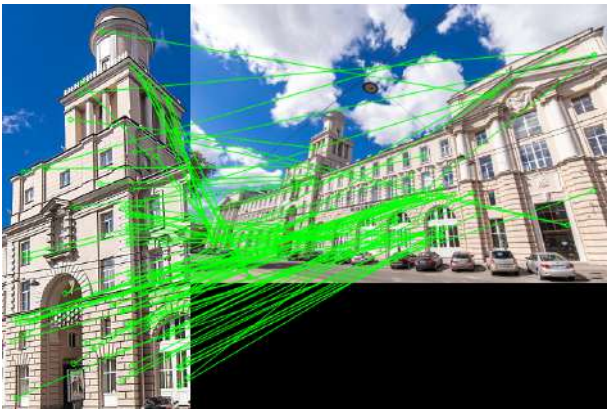


Figure 3.18. 100 strongest matches with brute force method and SIFT descriptors

distance between first nearest match is significantly lower comparing to distance with the second nearest match. With OpenCV this method can be implemented as following:

Listing 3.32. Finding k-nearest best match for two sets of descriptors and filtering them with OpenCV and C++.

```
std::vector<cv::DMatch> matches;
std::vector<std::vector<cv::DMatch> >
  knn_matches;
// Find KNN matches with k = 2
matcher->knnMatch(I1des, I2des,
  knn_matches, 2);
// Select good matches
double knn_ratio = 0.75;
for (int m = 0; m < knn_matches.size(); m++)
  if (knn_matches[m].size() > 1)
    if (knn_matches[m][0].distance <
        knn_ratio *
        knn_matches[m][1].distance)
      matches.push_back(knn_matches[m][0]);
```

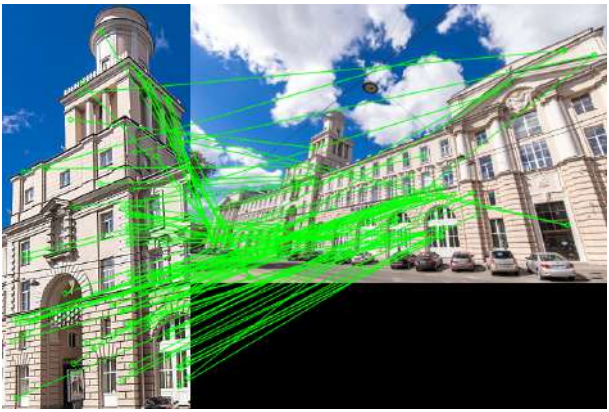


Figure 3.19. 100 strongest matches with FLANN method and SIFT descriptors

Listing 3.33. Finding k-nearest best match for two sets of descriptors and filtering them with OpenCV and Python.

```
# Find KNN matches with k = 2
matches = \
  matcher.knnMatch(I1des, I2des, k = 2)
# Select good matches
knn_ratio = 0.75
good = []
for m in matches:
  if len(m) > 1:
    if m[0].distance < \
       knn_ratio * m[1].distance:
      good.append(m[0])
matches = good
```

Figures 3.20 — 3.21 shows the result of using the k-nearest method in combination with brute force and FLANN descriptor matchers.

The `DescriptorMatcher` interface also has functions which can be used to train a matcher on set of images with corresponding descriptors to match a single query (`des1` in our case) with a set of image descriptors (instead of single `des2`). This is the reason for index returned in

Figure 3.20. All matches with brute force method, k-nearest ratio filter and SIFT descriptors

`imgIdx` parameter of the `DMatch`. However since we are matching only two images we don't need this extra data.

To display found matches a `cv::drawMatches()` C++ function can be used (`cv2.drawMatches()` in Python). It combines two images into one, draws feature points and match them with lines. To display only top matches a `matches` array which we acquired after descriptor matching can be sorted by `distance` value and its top part can be visualized:

Listing 3.34. Displaying top 10 matches with OpenCV and C++.

```
1   sort(matches.begin(), matches.end(),
2     [](const cv::DMatch &a,
3        const cv::DMatch &b)
4       {
5       return a.distance < b.distance;
6       });
7   int num_matches =
8     std::min(10, (int)matches.size());
9   cv::drawMatches(I1, I1fp, I2, I2fp,
10    std::vector<cv::DMatch>(matches.begin(),
11      matches.begin() + num_matches), Imatch,
12    cv::Scalar(0, 255, 0), cv::Scalar(-1),
```

Figure 3.21. All matches with FLANN method, k-nearest ratio filter and SIFT descriptors

```
13      std::vector<char>(0),
14      cv::DrawMatchesFlags::
15        NOT_DRAW_SINGLE_POINTS);
```

Listing 3.35. Displaying top 10 matches with OpenCV and Python.

```
1   num_matches = 10
2   matches = sorted(matches,
3     key = lambda x:x.distance)
4   Imatch = cv2.drawMatches(I1, I1fp, I2, I2fp,
5     matches[:num_matches], None, flags =
6    cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS,
7    matchColor = (0,255,0))
```

Next, we need to filter the outliers and calculate the transformation matrix between two sets of matched feature points. This can be done by the RANSAC (RANdom SAmples Consensus) algorithm implemented in OpenCV `cv::findHomography(I1pts, I2pts, method, threshold, mask)` → `M` C++ function (`cv2.findHomography(I1pts, I2pts, method, threshold)` → `M, mask`). This function calculates the transformation matrix M from

the points of the first image `I1pts` to the second image `I2pts` with given `threshold`. The inliers and outliers of best RANSAC hypothesis are marked in `mask`. The `cv::RANSAC` (`cv2.RANSAC` in Python) method is used. Its OpenCV implementation requires at least 10 pairs to work, so this should be checked:

Listing 3.36. Executing RANSAC to calculate the transformation matrix with OpenCV and C++.

```
const int MIN_MATCH_COUNT = 10;
if (matches.size() < MIN_MATCH_COUNT)
{
  std::cout << "Not␣enough␣matches.\n";
  return;
}
// Create arrays of point coordinates
std::vector<cv::Point2f> I1pts, I2pts;
for (int m = 0; m < matches.size(); m++)
{
  I1pts.push_back(
    I1fp[matches[m].queryIdx].pt);
  I2pts.push_back(
    I2fp[matches[m].trainIdx].pt);
}
// Run RANSAC method
std::vector<char> mask;
cv::Mat M = cv::findHomography(I1pts, I2pts,
  cv::RANSAC, 5, mask);
```

Listing 3.37. Executing RANSAC to calculate the transformation matrix with OpenCV and Python.

```
MIN_MATCH_COUNT = 10
if len(matches) < MIN_MATCH_COUNT:
  print("Not␣enough␣matches.")
  return
# Create arrays of point coordinates
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)
```

```
10  # Run RANSAC method
11  M, mask = cv2.findHomography(I1pts, I2pts,
12    cv2.RANSAC, 5)
13  mask = mask.ravel().tolist()
```

Now we can use the calculated transformation matrix `M` to display a location of the tower on top of the building. To do this we have to calculate the transformation of four corners of the first image and transform them with perspective transformation matrix `M`:

Listing 3.38. Displaying the location of the first image on the second one with OpenCV and C++.

```
1   std::vector<cv::Point2f> I1box, I1to2box;
2   // Image corners
3   I1box.push_back(cv::Point2f(0, 0));
4   I1box.push_back(
5     cv::Point2f(0, (float)I1.rows - 1));
6   I1box.push_back
7     (cv::Point2f((float)I1.cols - 1,
8                  (float)I1.rows - 1));
9   I1box.push_back(
10    cv::Point2f((float)I1.cols - 1, 0));
11  cv::perspectiveTransform(I1box, I1to2box, M);
12  // Convert to integers
13  std::vector<cv::Point2i> I1to2box_i;
14  for (int i = 0; i < I1to2box.size(); i++)
15    I1to2box_i.push_back(
16      cv::Point2i(I1to2box[i]));
17  // Draw a red box on the second image
18  cv::Mat I2res = I2.clone();
19  cv::polylines(I2res, I1to2box_i, true,
20    cv::Scalar(0, 0, 255), 1,
21    cv::LineTypes::LINE_AA);
22  cv::imshow("Search␣result", I2res);
```

Listing 3.39. Displaying the location of the first image on the second one with OpenCV and Python.

```
1   # Image corners
2   h, w = I1.shape[:2]
```

```
I1box = np.float32([[0, 0], [0, h - 1],
  [w - 1, h - 1], [w - 1, 0]]). \
  reshape(-1, 1, 2)
I1to2box = \
  cv2.perspectiveTransform(I1box, M)
# Draw a red box on the second image
I2res = cv2.polylines(I2,
  [np.int32(I1to2box)], True, (0, 0, 255),
  1, cv2.LINE_AA)
cv2.imshow("Search result", I2res)
```

Figure 3.22. Location of the object on the scene

Finally, after we have found a transformation and filtered outliers, we can display inlier matches along with found transformation:

Listing 3.40. Displaying inlier matches with OpenCV and C++.

```
cv::Mat Itrans;
cv::drawMatches(I1, I1fp, I2res, I2fp,
  matches, Itrans, cv::Scalar(0, 255, 0),
  cv::Scalar(-1), mask,
  cv::DrawMatchesFlags::
  NOT_DRAW_SINGLE_POINTS);
cv::imshow("Transformation", Itrans);
```

Listing 3.41. Displaying inlier matches with OpenCV and Python.

```
Itrans = \
 cv2.drawMatches(I1, I1fp, I2res, I2fp,
 matches, None, matchesMask = mask, flags =
 cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS,
 matchColor = (0,255,0))
cv2.imshow("Transformation", Itrans)
```

Figure 3.23. Inliers found by the RANSAC method

Procedure of Practical Assignment Performing

1. *Feature points detection.* Select three arbitrary images. Perform to search for feature points using the SIFT and ORB feature point descriptors.

2. *Feature points matching.* Select two pairs of images: first image of each pair should have an object (e.g., some book) and the second image should be a scene containing this object. Extract feature points of an object and match them with feature points of a scene containing this object. Calculate the transformation matrix using RANSAC method and highlight the object position in the scene.

Show the inlier matches. Compare feature point descriptors for the task of image matching.

3. *Optional.* Implement the simple automatic image stitching. Use learned methods to calculate the transformation matrix between two images and stitch them into a single panoramic image. Use it to stitch three images into a single panoramic image. You may assume that the order of the images is known (e.g., all three images are shot with moving camera from left to right), so reordering is not required.

Note. Please note that when doing the practical assignment you are not allowed to use the *"Lenna"* image or any other image that was used either in this book or during the presentation.

Content of the Report

1. Title page.
2. Objective.
3. Theoretical substantiation of the applied methods and functions.
4. Assignment steps:
   (a) Original images;
   (b) Code of the scripts;
   (c) Comments;
   (d) Resulting images.
5. Conclusion.
6. Answers to questions for the defense.

Questions to Practical Assignment Report Defense

1. How the characteristic orientation (rotation) of the feature point can be estimated?
2. How to filter the not-strong feature point descriptor matches on a repeating texture (e.g., windows, water, etc.)?

3. What is the minimum required sample size (the number of matched pairs of feature points) to build an affine transformation hypothesis with RANSAC method? What is the minimum required sample size to build a perspective transformation hypothesis with RANSAC method?

4. How to use feature points for stitching a panoramic image?

Practical Assignment №4
Face Detection using Viola-Jones Approach

Objective

Study of Viola-Jones approach for detection of faces and part of bodies in the images.

Guidelines

Before getting started, students should be familiar with the functions of the MATLAB or OpenCV for working with the cascade object detectors and Viola-Jones approach. Practical assignment is designed for 4 hours.

Brief Theory

The Viola-Jones face detector method [19] is based on the following concepts:

1. Haar-like feaures 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 of strong classifiers into a cascade classifier.

Haar-like features

Haar-like feature is a kind of a 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. In a traditional Viola-Jones face detector algorithm 4 types of Haar-like features that are shown in Fig. 4.1 are used. To calculate the value of the Haar-like feature we need to calculate sums of pixels inside rectangular areas of the image and do it as fast as possible.

$$value = \sum (pixels\ in\ black\ area) - \sum (pixels\ in\ white\ area). \quad (4.1)$$

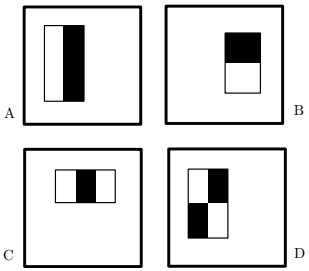


Figure 4.1. Haar-like features used in Viola-Jones face detector

Obviously, the straightforward calculation of the sum of pixel values in a rectangle would require number of sums that is equal to 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 that are positioned to the left and above of the current pixel. To calculate the sum of pixel intensity values in an

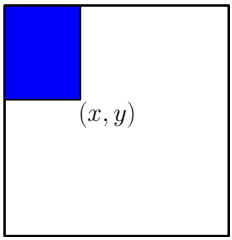


Figure 4.2. Integral image

arbitrary rectangle we need to access four pixels of an integral image which are located at the corners of the rectangle, see Fig. 4.3.

$$sum = D - B - C + A, \tag{4.2}$$

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

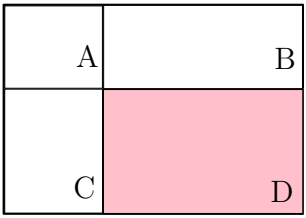


Figure 4.3. Rectangular sum calculation with an integral image

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. The training algorithm is called AdaBoost. It consists of several boosting rounds, and each boosting round is a selection of a best weak Haar-like feature to classify the training set with taking the classification errors of the previous rounds into an account.

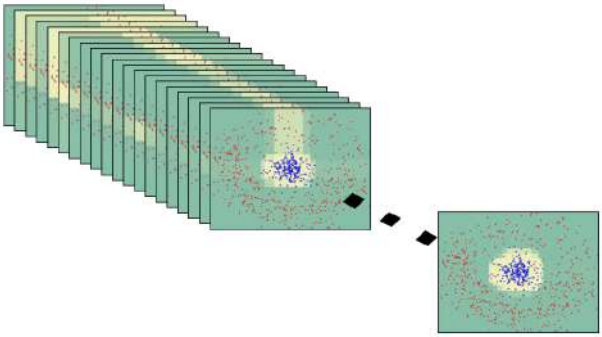


Figure 4.4. Combining weak classifiers into a strong classifier

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

1. On an input we have a training set $T = \{(x_i, y_i)|x_i \in X, y_i \in \{-1,1\}\}$ and a set of all possible weak classifiers $\{h\}$.

2. Initialize the weights for a training set items to be equal and sum up to 1. $D_1(i) = 1/m$, where m is a 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 an account):

   $$\epsilon_k = Pr_{i\ D_k}[h_k(x_i) \neq y_i]. \tag{4.3}$$

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

   $$\alpha_k = \frac{1}{2} \ln \left(\frac{1 - \epsilon_k}{\epsilon_k} \right). \tag{4.4}$$

   (c) Reweigh the training set with new weights:

   $$D_{k+1}(i) = \frac{D_k(i)}{Z_k} \cdot \begin{cases} e^{-\alpha_k}, h_k(x_i) = y_i, \\ e^{\alpha_k}, h_k(x_i) \neq y_i. \end{cases} \tag{4.5}$$

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

   $$H(x) = \text{sign} \left(\sum_{k=1}^{K} \alpha_k h_k(x) \right). \tag{4.6}$$

Cascade classifiers

A strong classifier that have a required accuracy may require calculation of too much weak classifiers that would slow down the detection speed taking into an account that most of scanned windows do not contain faces. To speed up the detection rate a set of classifiers with increasing complexity are organized in a cascade of classifiers. The cascade contains a set of classifiers with an increasing complexity and detection rate, see Fig. 4.5.

To be classified positively, a sliding window should pass all cascade stages. In case if any classifier rejects the window, it is immediately rejected and detector proceeds to the next window. As a result, that

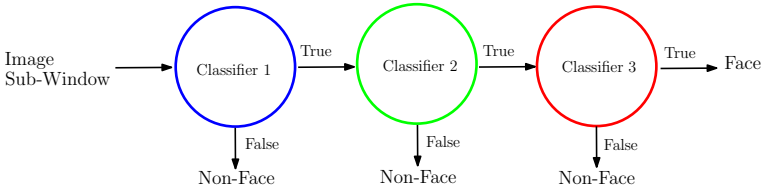


Figure 4.5. Cascade classifier

most of negative windows are rejected fast with 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:

$$TP = \prod_i TP_i. \tag{4.7}$$

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

$$FP = \prod_i FP_i. \tag{4.8}$$

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

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

Viola-Jones approach with MATLAB

MATLAB provides `vision.CascadeObjectDetector` object [20]. The cascade object detector uses the Viola-Jones algorithm to detect people's faces, noses, eyes, mouth, or upper body using special parameters of the object. Of course, you can train the custom detector on your own objects using `Image Labeler`.

Listing 4.1. Create an object using `CascadeObjectDetector`.

```
faceDetector = vision.CascadeObjectDetector;
```

After that you should apply your detector `faceDetector` to the image `I`. As a result, you will obtain the coordinates of the rectangular bounding boxes around the faces in a format `[x y width height]`, that specifies in pixels the upper-left corner and size of the bounding boxes.

Listing 4.2. Applying detector to the image with MATLAB.

```
I = imread('photo_faces.jpg');
bboxes = faceDetector(I);
```

Figure 4.6. Original image with faces

In the last step you should place bounding boxes to the image. In MATLAB you can use the function `insertObjectAnnotation(I, shape, position, label[, optional parameters])`. Let's place rectangular annotations to the faces and display it using `imshow(I)` function. The result is shown in Fig. 4.7.

Listing 4.3. Applying detector to the image with MATLAB.

```
IFaces = insertObjectAnnotation(I, ...
  'rectangle',bboxes,'Face');
imshow(IFaces);
```

MATLAB provides several pretrained classifiers, e.g. `UpperBody`, `EyePairSmall`, `Mouth`, or `Nose`. The complete list you can find in [20]. Specify the region of interest (ROI) in the detector `detector(I,roi)`

Figure 4.7. Image with detected faces

you can find, for examples, eyes only in the regions of faces. To do that you should use the optional parameter `UseROI` with value `true` in the cascade object detector. For example, let's detect eyes on the detected faces.

Listing 4.4. Create an object with `EyePairSmall` classification model and ROI.

```
eyesDetector = ...
  vision.CascadeObjectDetector(...
  'ClassificationModel','EyePairSmall',...
  'UseROI',true);
```

Unfortunately, MATLAB doesn't allow to apply the whole array of bounding boxes from one detector to another one as ROI. So, let's do it step-by-step.

Listing 4.5. Eyes detector using ROI in faces.

```
counter = 1;
for i = 1:1:length(bboxes)
  temp = eyesDetector(I, bboxes(i,:));
  if ~isempty(temp)
    bboxes2(counter,:) = temp;
    counter = counter + 1;
  end
end
```

After that we can put labels of eyes to the previous image with faces' annotations, see Fig. 4.8.

Listing 4.6. Display eyes in the image with faces.

```
IEyes = insertObjectAnnotation(IFaces,...
  'rectangle',bboxes2,'Eyes',...
  'Color','magenta');
imshow(IEyes)
```

Figure 4.8. Image with detected faces and eyes

Viola-Jones approach with OpenCV

To execute a cascade classifier OpenCV provides a class named `cv::CascadeClassifier` in C++ and `cv2.CascadeClassifier` in Python. Cascade is defined in an `XML` file and can be loaded to a classifier object with the `cv::CascadeClassifier::load(cv::String file_name)` C++ function (`cv2.CascadeClassifier.load()` in Python). The creation of cascade classifier for faces is as following:

Listing 4.7. Create and load a cascade classifier faces with OpenCV and C++.

```
cv::CascadeClassifier detector;
cv::String cascade_fn =
  cv::samples::findFile(
    "haarcascade_frontalface_default.xml");
detector.load(cascade_fn);
```

Listing 4.8. Create and load a cascade classifier faces with OpenCV and Python.

```
1  detector = cv2.CascadeClassifier()
2  cascade_fn = cv2.samples.findFile(
3    "haarcascade_frontalface_default.xml")
4  detector.load(cascade_fn)
```

Note. OpenCV library provides some built-in cascade descriptors for faces, eyes, mouth, cat faces and Russian license plates which can be found in `data\haarcascades\` folder. See [21] for a complete list of built-in cascades. Other cascade classifiers can be trained using the cascade training tool which is out of the scope of the current assignment You can refer to the *traincascade* OpenCV tool documentation [22] for more information.

After a cascade is loaded it can be applied to an image with `cv::detectMultiScale(I, objs, scale_f, min_neighb)` C++ function (`cv2.detectMultiScale(I, scale_f, min_neighb)` → `objs` in Python). This function takes an argument of the I image to process. The returned list of rectangular areas that satisfied the cascade classifier condition is stored in the `objs` list which is passed as a second argument in C++ (or is returned by the `detectMultiScale()` function in Python). Optional arguments allow specifying the additional cascade parameters which are the scale factor that is used when increasing the windows size, minimum number of rectangles to be classified in the area when filtering for the false positive results and minimum and maximum sizes of the detector area. The cascade classification is executed as following:

Listing 4.9. Execute a cascade classifier for faces using scale factor 1.07 and 3 minimum required number of matches with OpenCV and C++.

```
1  std::vector<cv::Rect> faces;
2  detector.detectMultiScale(Igray, faces,
3    1.07, 3);
```

Listing 4.10. Execute a cascade classifier for faces using scale factor 1.07 and 3 minimum required number of matches with OpenCV and Python.

```
1  faces = detector.detectMultiScale(Igray,
```

```
2      scaleFactor = 1.07, minNeighbors = 3)
```

Figure 4.9. Original image with faces

As result the `faces` array will store the list of rectangles for found objects (faces). Then we can iterate over them all and display them on the source image.

Listing 4.11. Display found faces with OpenCV and C++.

```
1   cv::Mat Iout = I.clone();
2   for (int i = 0; i < faces.size(); i++)
3     cv::rectangle(Iout, faces[i],
4       cv::Scalar(0, 255, 255), 1);
```

Listing 4.12. Display found faces with OpenCV and Python.

```
1   Iout = I.copy()
2   for (x, y, w, h) in faces:
3     Iout = cv.rectangle(Iout, (x, y, w, h),
4       (0, 255, 255), 1)
```

As it can be seen from the resulting image, almost all faces were detected, excepting the one which is rotated and could not be classified by a cascade which was not trained to find this type of rotated faces (see Fig. 4.10).

Now let us try to detect eyes on the found face. For this need we will take a higher resolution image of face with eyes, shown on the Fig. 4.11

Figure 4.10. Image with detected faces highlighted

It's obvious that we don't need to scan the whole image for eyes as there would be a lot of false-popsitive results, so we will define a Region-Of-Interest (ROI) object as a face which was already found and then search for eyes with taking the ROI into an account. In C++ it can be implemented by creating a special `cv::Mat` object with it's constructor that takes and image and rectangular ROI defined by `cv::Rect` object. This is exactly the object that is returned by the `cv::detectMultiScale()` detector function. So, to find faces, first we need to load a classifier for faces, then for each found face execute the eyes detector with ROI.

Listing 4.13. Detect and display eyes with taking face areas into an account with OpenCV and C++.

```
1   // Load eyes cascade
2   cv::CascadeClassifier eye_detector;
3   cv::String eye_cascade_fn =
4     cv::samples::findFile(
5       "haarcascade_eye.xml");
6   eye_detector.load(eye_cascade_fn);
7   // For each face use it as a ROI
8   // and detect eyes
9   for (int i = 0; i < faces.size(); i++)
10  {
```

Figure 4.11. Close up image of a face

```
11      Rect f = faces[i];
12      cv::Mat Iface = cv::Mat(Igray, f);
13      std::vector<cv::Rect> eyes;
14      eye_detector.detectMultiScale(Iface,
15        eyes, 1.05);
16      for (int j = 0; j < eyes.size(); j++)
17      {
18        eyes[j].x += f.x;
19        eyes[j].y += f.y;
20        cv::rectangle(Iout, eyes[j],
21          cv::Scalar(147, 20, 255), 1);
22      }
23    }
```

When using the Python programming language it's even easier to define ROI. For this need a slice of the Numpy array should be done, then this subarray is used as a normal image with all OpenCV functions.

Listing 4.14. Detect and display eyes with taking face areas into an account with OpenCV and Python.

```
1   # Load eyes cascade
2   eye_detector = cv2.CascadeClassifier()
3   cascade_fn = cv2.samples.findFile(
```

```
4       "haarcascade_eye.xml")
5     eye_detector.load(cascade_fn)
6     # For each face use it as a ROI
7     # and detect eyes
8     for (x, y, w, h) in faces:
9       Iface = I[y : y + h, x : x + w]
10      eyes = eye_detector.detectMultiScale(
11        Iface, scaleFactor = 1.05)
12      for (x2, y2, w2, h2) in eyes:
13        Iout = cv.rectangle(Iout,
14          (x + x2, y + y2, w2, h2),
15          color = (147, 20, 255))
```

The eyes detection result is shown in the Fig. 4.12.

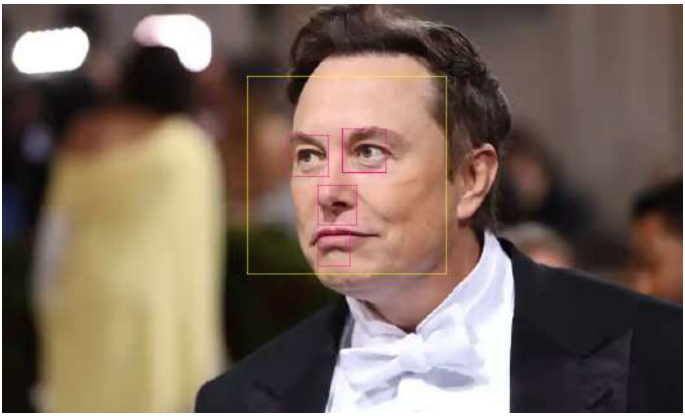


Figure 4.12. Eyes detection result

As it can be seen, some false-positive eye positions are detected. To overcome this problem we can modify the parameters of the cascade (the minimum required number of matches or scale factor). However you can notice that all false-positives are located at the bottom part of the face, however eyes are always located at the top 2/3 of the face, so we can modify the ROI definition taking this into an account.

Listing 4.15. Define ROI and top 2/3 of the face image with OpenCV and C++.

```
cv::Mat Iface_top = cv::Mat(Igray,
  cv::Rect(f.x, f.y, f.width,
  f.height * 2 / 3));
```

Listing 4.16. Define ROI and top 2/3 of the face image with OpenCV and Python.

```
Iface_top = \
  Igray[y : y + h * 2 // 3, x : x + w]
```

Then, this `Iface_top` image can be used in `detectMultiScale()` function of the cascade detector to get a better result shown in Fig. 4.13

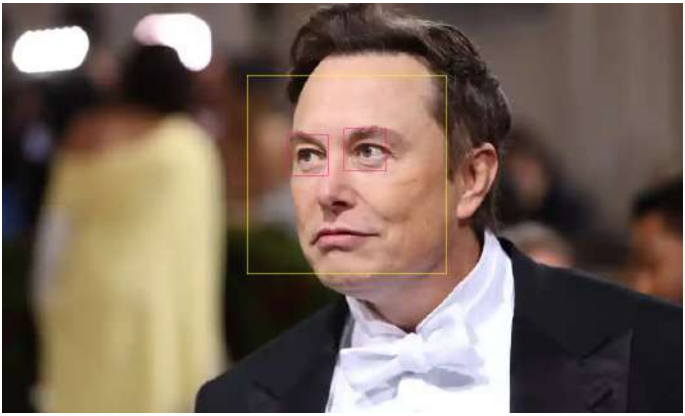


Figure 4.13. Eyes detection result in top 2/3 of the face

The same approach can be used when detecting other parts of the face, for example mouth is located at the botom 1/3 of the face and so on.

Procedure of Practical Assignment Performing

1. *Faces detection.* Select three arbitrary images contains several faces. Try to use images with a different number of faces and different scales. Perform to search faces using Viola-Jones approach. Calculate the number of found faces on each image.

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

3. *Optional 1.* Implement the face detection in videostream using pre-recorded video with faces.

4. *Optional 2.* Implement the face detection in live videostream using web-camera.

Note. Please note that when doing the practical assignment you are not allowed to use the *"Lenna"* image or any other image that was used either in this book or during the presentation.

Content of the Report

1. Title page.

2. Objective.

3. Theoretical substantiation of the applied methods and functions.

4. Assignment steps:

   (a) Original images;

   (b) Code of the scripts;

   (c) Comments;

   (d) Resulting images.

5. Conclusion.

6. Answers to questions for the defense.

Questions to Practical Assignment Report Defense

1. What is the special image representation used in the Viola-Jones approach?
2. What is the main advantage of Haar-like features for classifier training?
3. Could you use Viola-Jones approach for detecting arbitrary objects and why?

List of references

[1] MATLAB - MathWorks. — https://www.mathworks.com/products/matlab.html. — 2022. — [Online; accessed 01-Dec-2022].

[2] Home - OpenCV. — https://opencv.org/html. — 2022. — [Online; accessed 01-Dec-2022].

[3] 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.

[4] Colorimetry — Part 4: CIE 1976 L*a*b* colour space | CIE. — https://cie.co.at/publications/colorimetry-part-4-cie-1976-lab-colour-space-1. — 2022. — [Online; accessed 01-Dec-2022].

[5] Mouse as a Paint-Brush. — https://docs.opencv.org/4.6.0/db/d5b/tutorial_py_mouse_handling.html. — 2022. — [Online; accessed 01-Dec-2022].

[6] Hough Paul VC. Method and means for recognizing complex patterns. — 1962. — Dec. 18. — US Patent 3,069,654.

[7] Wagoner Amy R., Schrader Daniel K., Matson Eric T. Survey on Detection and Tracking of UAVs Using Computer Vision // 2017 First IEEE International Conference on Robotic Computing (IRC). — 2017. — P. 320–325.

[8] Harris Chris, Stephens Mike et al. A combined corner and edge detector // Alvey vision conference / Citeseer. — 1988. — 15 no. 50. — P. 10–5244.

[9] Shi Jianbo et al. Good features to track // 1994 Proceedings of IEEE conference on computer vision and pattern recognition / IEEE. — 1994. — P. 593–600.

[10] Lowe David G. Distinctive image features from scale-invariant keypoints // International journal of computer vision. — 2004. — Vol. 60, no. 2. — P. 91–110.

[11] Bay Herbert, Tuytelaars Tinne, Gool Luc Van. Surf: Speeded up robust features // European conference on computer vision / Springer. — 2006. — P. 404–417.

[12] 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.

[13] Brief: Binary robust independent elementary features / Calonder Michael, Lepetit Vincent, Strecha Christoph, and Fua Pascal // European conference on computer vision / Springer. — 2010. — P. 778–792.

[14] ORB: An efficient alternative to SIFT or SURF / Rublee Ethan, Rabaud Vincent, Konolige Kurt, and Bradski Gary // 2011 International conference on computer vision / IEEE. — 2011. — P. 2564–2571.

[15] Detect SIFT Features. — `https://www.mathworks.com/help/vision/ref/detectsiftfeatures.html`. — 2021. — [Online; accessed 20-May-2022].

[16] Introduction to SIFT (Scale-Invariant Feature Transform). — `https://docs.opencv.org/4.6.0/da/df5/tutorial_py_sift_intro.html`. — 2022. — [Online; accessed 01-Dec-2022].

[17] Detect ORB Features. — `https://www.mathworks.com/help/vision/ref/detectorbfeatures.html`. — 2019. — [Online; accessed 20-May-2022].

[18] ORB (Oriented FAST and Rotated BRIEF). — `https://docs.opencv.org/4.6.0/d1/d89/tutorial_py_orb.html`. — 2022. — [Online; accessed 01-Dec-2022].

[19] Viola Paul, Jones Michael. Robust Real-time Object Detection // International Journal of Computer Vision. — 2001.

[20] Cascade Object Detector. — https://www.mathworks.com/help/vision/ref/vision.cascadeobjectdetector-system-object.html. — 2012. — [Online; accessed 20-May-2022].

[21] OpenCV Haar cascades repository. — https://github.com/opencv/opencv/tree/4.x/data/haarcascades. — 2020. — [Online; accessed 20-May-2022].

[22] Cascade Classifier Training. — https://docs.opencv.org/4.5.5/dc/d88/tutorial_traincascade.html. — 2021. — [Online; accessed 20-May-2022].

Sergei Shavetov
Andrei Zhdanov

Computer Vision

Study guide

Original version
Editorial-Publishing Department of ITMO University
EP Department Head N. Gusarova
Signed to print
Order No
Printed circulation
Risograph printing

Editorial-Publishing Department of
ITMO University
197101, St. Petersburg, Kronverkskiy pr., 49