



С.В. Шаветов, А.Д. Жданов

ОСНОВЫ ОБРАБОТКИ  
ИЗОБРАЖЕНИЙ: ЛАБОРАТОРНЫЙ  
ПРАКТИКУМ, ЧАСТЬ 2



Санкт-Петербург

2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

С.В. Шаветов, А.Д. Жданов

**ОСНОВЫ ОБРАБОТКИ  
ИЗОБРАЖЕНИЙ: ЛАБОРАТОРНЫЙ  
ПРАКТИКУМ, ЧАСТЬ 2**

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ  
В УНИВЕРСИТЕТЕ ИТМО

по направлениям подготовки

15.03.06 Мехатроника и робототехника

27.03.04 Управление в технических системах

в качестве учебно-методического пособия для реализации  
образовательных программ высшего образования бакалавриата

**ИТМО**

Санкт-Петербург

2022

С.В. Шаветов, А.Д. Жданов. Основы обработки изображений: лабораторный практикум, часть 2. — СПб: Университет ИТМО, 2022. — 84 с.

**Рецензенты:**

И.С. Потемин, канд. техн. наук, доцент факультета ПИ и КТ, Университет ИТМО.

Учебно-методическое пособие посвящено основам обработки цифровых изображений. Представлено четыре лабораторных работы с последовательно увеличивающейся сложностью. В пособии представлены примеры реализации алгоритмов на трех языках программирования: MATLAB, Python и C++. Пособие предназначено для студентов, изучающих дисциплину «Техническое зрение» по образовательной программе бакалавриата «Робототехника и искусственный интеллект» и направлениям подготовки 15.03.06 «Мехатроника и робототехника» и 27.03.04 «Управление в технических системах».



**Университет ИТМО** — национальный исследовательский университет, ведущий вуз России в области информационных, фотонных и биохимических технологий. Альма-матер победителей международных соревнований по программированию — ICPC (единственный в мире семикратный чемпион), Google Code Jam, Facebook Hacker Cup, Яндекс.Алгоритм, Russian Code Cup, Topcoder Open и др. Приоритетные направления: ИТ, фотоника, робототехника, квантовые коммуникации, трансляционная медицина, Life Sciences, Art&Science, Science Communication. Входит в ТОП-100 по направлению «Автоматизация и управление» Шанхайского предметного рейтинга (ARWU) и занимает 74 место в мире в британском предметном рейтинге QS по компьютерным наукам (Computer Science and Information Systems). С 2013 по 2020 гг. — лидер Проекта 5–100.

© Университет ИТМО, 2022

© С.В. Шаветов, 2022

© А.Д. Жданов, 2022

# Содержание

<b>2 Введение</b>	<b>5</b>
<b>Лабораторная работа №1. Сегментация изображений</b> 6	
Цель работы . . . . .	6
Методические рекомендации . . . . .	6
Теоретические сведения . . . . .	6
Бинаризация изображений . . . . .	6
Сегментация изображений . . . . .	9
Порядок выполнения работы . . . . .	22
Содержание отчета . . . . .	23
Вопросы к защите лабораторной работы . . . . .	23
<b>Лабораторная работа №2. Преобразование Хафа</b> 24	
Цель работы . . . . .	24
Методические рекомендации . . . . .	24
Теоретические сведения . . . . .	24
Порядок выполнения работы . . . . .	28
Содержание отчета . . . . .	29
Вопросы к защите лабораторной работы . . . . .	30
<b>Practical Assignment №3. Features Detectors</b> 31	
Objective . . . . .	31
Guidelines . . . . .	31
Brief Theory . . . . .	31
SIFT detector . . . . .	33
ORB detector . . . . .	41
Feature point descriptors matching . . . . .	44
Procedure of Practical Assignment Performing . . . . .	63
Content of the Report . . . . .	64
Questions to Practical Assignment Report Defense . . . . .	64
<b>Practical Assignment №4. Face Detection using Viola-Jones Approach</b> 66	
Objective . . . . .	66
Guidelines . . . . .	66

Brief Theory . . . . .	66
Procedure of Practical Assignment Performing . . . . .	80
Content of the Report . . . . .	80
Questions to Practical Assignment Report Defense . . . . .	81

## 2 Введение

Задача технического зрения является одной из ключевых задач, которые необходимо решить для осуществления автоматизации и роботизации производственных процессов. В процессе выполнения данного лабораторного практикума учащиеся получат практические навыки и умения по обработке изображений, включающие алгоритмы анализа изображений.

Для выполнения заданий учащимся предлагается использовать один из трех языков программирования: MATLAB, C++ или Python. В пособии рассматриваются особенности и методики решения задач обработки изображений с использованием каждого из перечисленных языков программирования, что позволит учащимся понять, какой язык лучше подойдет для решения задач технического зрения, с которыми они столкнутся после завершения курса обучения.

По итогам выполнения каждой из лабораторных работ учащиеся должны предоставить отчет в формате PDF, содержащий описание используемых методов в приложении к выбранному языку программирования, полученные результаты обработки собственных изображений и исходные тексты программ. Также они должны ответить на дополнительные теоретические вопросы для проверки уровня освоения практических навыков.

Учебное пособие рекомендовано к использованию студентам, изучающим дисциплину «Техническое зрение» по образовательной программе «Робототехника и искусственный интеллект», реализуемой в рамках направлений подготовки бакалавров 15.03.06 «Мехатроника и робототехника» и 27.03.04 «Управление в технических системах» для лучшего усвоения излагаемого лекционного материала, предусмотренного данной дисциплиной, подготовке к практическим занятиям, а также для получения практических знаний и умений в области обработки изображений. По завершению практикума учащиеся получат знания и разовьют умения и навыки использования методов интеллектуальной обработки данных для решения прикладных задач обработки цифровых изображений.

# Лабораторная работа №1

## Сегментация изображений

### Цель работы

Освоение основных способов сегментации изображений на семантические области.

### Методические рекомендации

До начала работы студенты должны ознакомиться с основными функциями среды MATLAB по преобразованию цветовых пространств изображений и способами определения порогов. Лабораторная работа рассчитана на 5 часов.

### Теоретические сведения

#### Бинаризация изображений

Простейшим способом сегментации изображения на два класса (фоновые пиксели и пиксели объекта) является *бинаризация*. Бинаризацию можно выполнить по порогу или по двойному порогу. В первом случае:

$$I_{new}(x,y) = \begin{cases} 0, & I(x,y) \leq t, \\ 1, & I(x,y) > t, \end{cases} \quad (3.1)$$

где  $I$  — исходное изображение,  $I_{new}$  — бинаризованное изображение,  $t$  — порог бинаризации. Бинаризация данным методом в среде MATLAB может быть выполнена с использованием функций `im2bw()` (устаревшая) или `imbinarize()`.

**Листинг 3.1.** Бинаризация.

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

Бинаризация по двойному порогу (диапазонная бинаризация):

$$I_{new}(x,y) = \begin{cases} 0, & I(x,y) \leq t_1, \\ 1, & t_1 < I(x,y) \leq t_2, \\ 0, & I(x,y) > t_2, \end{cases} \quad (3.2)$$

где  $I$  — исходное изображение,  $I_{new}$  — бинаризованное изображение,  $t_1$  и  $t_2$  — верхний и нижний пороги бинаризации. Бинаризация данным методом в среде MATLAB может быть выполнена с использованием функции `roicolor()`. Для преобразования полноцветного изображения в полутонаовое можно предварительно воспользоваться функцией `rgb2gray()`.

**Листинг 3.2.** Бинаризация по двойному порогу.

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

Пороги бинаризации  $t$ ,  $t_1$  и  $t_2$  могут быть либо заданы вручную, либо вычислены с помощью специальных алгоритмов. В случае автоматического вычисления порога можно воспользоваться следующими алгоритмами.

1. Поиск максимального  $I_{max}$  и минимального  $I_{min}$  значений интенсивности исходного полутонаового изображения и нахождение их среднего арифметического. Среднее арифметическое будет являться глобальным порогом бинаризации  $t$ :

$$t = \frac{I_{max} - I_{min}}{2}. \quad (3.3)$$

2. Поиск оптимального порога  $t$  на основе модуля градиента яркости каждого пикселя. Для этого сначала вычисляется модуль градиента в каждой точке  $(x,y)$ :

$$G(x,y) = \max \{|I(x+1,y) - I(x-1,y)|, |I(x,y+1) - I(x,y-1)|\}, \quad (3.4)$$

затем вычисляется оптимальный порог  $t$ :

$$t = \frac{\sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} I(x,y) G(x,y)}{\sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} G(x,y)}. \quad (3.5)$$

3. Вычисление оптимального порога  $t$  статистическим методом Отсу (Оцу, англ. Otsu), разделяющим все пиксели на два класса 1 и 2, минимизируя дисперсию внутри каждого класса  $\sigma_1^2(t)$  и  $\sigma_2^2(t)$  и максимизируя дисперсию между классами.

Алгоритм вычисления порога методом Отсу:

1. Вычисление гистограммы интенсивностей изображения и вероятности  $p_i = \frac{n_i}{N}$  для каждого уровня интенсивности, где  $n_i$  — число пикселей с уровнем интенсивности  $i$ ,  $N$  — число пикселей в изображении.
2. Задание начального порога  $t = 0$  и порога  $k \in (0, L)$ , разделяющего все пиксели на два класса, где  $L$  — максимальное значение интенсивности изображения. В цикле для каждого значения порога от  $k = 1$  до  $k = L - 1$ :
  - (a) Вычисление вероятностей двух классов  $\omega_j(0)$  и средних арифметических  $\mu_j(0)$ , где  $j = \overline{1, 2}$ :

$$\omega_1(k) = \sum_{s=0}^k p_s, \quad (3.6)$$

$$\omega_2(k) = \sum_{s=k+1}^L p_s = 1 - \omega_1(k), \quad (3.7)$$

$$\mu_1(k) = \sum_{s=0}^k \frac{s \cdot p_s}{\omega_1}, \quad (3.8)$$

$$\mu_2(k) = \sum_{s=k+1}^L \frac{s \cdot p_s}{\omega_2}. \quad (3.9)$$

- (b) Вычисление межклассовой дисперсии  $\sigma_b^2(k)$ :

$$\sigma_b^2(k) = \omega_1(k)\omega_2(k)(\mu_1(k) - \mu_2(k))^2. \quad (3.10)$$

- (c) Если вычисленное значение  $\sigma_b^2(k)$  больше текущего значения  $t$ , то присвоить порогу значение межклассовой дисперсии  $t = \sigma_b^2(k)$ .

3. Оптимальный порог  $t$  соответствует максимуму  $\sigma_b^2(k)$ .

В среде MATLAB порог  $t$  методом Отсю может быть вычислен с использованием функции `graythresh()`:

**Листинг 3.3.** Бинаризация методом Отсю.

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

либо с использованием функции `otsuthresh()` на основе гистограммы изображения:

**Листинг 3.4.** Бинаризация методом Отсю на основе гистограммы.

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

4. Адаптивные методы, работающие не со всем изображением, а лишь с его фрагментами. Такие подходы зачастую используются при работе с изображениями, на которых представлены неоднородно освещенные объекты. В среде MATLAB порог  $t$  адаптивным методом может быть вычислен при помощи функции `adaptthresh()`:

**Листинг 3.5.** Бинаризация адаптивным методом.

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

Помимо рассмотренных методов существуют и многие другие, например методы Бернсена, Эйквела, Ниблэка, Яновица и Брукштейна и др.

## Сегментация изображений

Рассмотрим несколько основных методов сегментации изображений.

## На основе принципа Вебера

Алгоритм предназначен для сегментации полутоновых изображений. *Принцип Вебера* подразумевает, что человеческий глаз плохо воспринимает разницу уровней серого между  $I(n)$  и  $I(n) + W(I(n))$ , где  $W(I(n))$  — функция Вебера,  $n$  — номер класса,  $I$  — кусочно-нелинейная функция градаций серого. Функция Вебера может быть вычислена по формуле:

$$W(I) = \begin{cases} 20 - \frac{12I}{88}, & 0 \leq I \leq 88, \\ 0,002(I - 88)^2, & 88 < I \leq 138, \\ \frac{7(I - 138)}{117} + 13, & 138 < I \leq 255. \end{cases} \quad (3.11)$$

Можно объединить уровни серого из диапазона  $[I(n), I(n) + W(I(n))]$  заменив их одним значением интенсивности.

Алгоритм сегментации состоит из следующих шагов:

1. Инициализация начальных условий: номер первого класса  $n = 1$ , уровень серого  $I(n) = 0$ .
2. Вычисление значения  $W(I(n))$  по формуле Вебера и присваивание значения  $I(n)$  всем пикселям, интенсивность которых находится в диапазоне  $[I(n), I(n) + W(I(n))]$ .
3. Поиск пикселей с интенсивностью выше  $G = I(n) + W(I(n)) + 1$ . Если найдены, то увеличение номера класса  $n = n + 1$ , присваивание  $I(n) = G$  и переход на второй шаг. В противном случае закончить работу. Изображение будет сегментировано на  $n$  классов интенсивностью  $W(I(n))$ .

## Сегментация RGB-изображений по цвету кожи

Общим принципом данного подхода является определение критерия близости интенсивности пикселей к оттенку кожи. Аналитически описать *оттенок кожи* довольно затруднительно, поскольку его описание базируется на человеческом восприятии цвета, меняется при изменении освещения, отличается у разных народностей, и т.д.

Существует несколько аналитических описаний для изображений в цветовом пространстве RGB, позволяющих отнести пиксель к классу «кожа» при выполнении условий:

$$\begin{cases} R > 95, \\ G > 40, \\ B < 20, \\ \max R, G, B - \min R, G, B > 15, \\ |R - G| > 15, \\ R > G, \\ R > B, \end{cases} \quad (3.12)$$

или

$$\begin{cases} R > 220, \\ G > 210, \\ B > 170, \\ |R - G| \leq 15, \\ G > B, \\ R > B, \end{cases} \quad (3.13)$$

или

$$\begin{cases} r = \frac{R}{R+G+B}, \\ g = \frac{G}{R+G+B}, \\ b = \frac{B}{R+G+B}, \\ \frac{r}{g} > 1,185, \\ \frac{rb}{(r+g+b)^2} > 0,107, \\ \frac{rg}{(r+g+b)^2} > 0,112. \end{cases} \quad (3.14)$$

### На основе цветового пространства CIE Lab

В цветовом пространстве **Lab** значение светлоты отделено от значения хроматической составляющей цвета (тон, насыщенность). Светлота задается координатой **L**, которая может находиться в диапазоне от 0 (темный) до 100 (светлый). Хроматическая составляющая цвета задается двумя декартовыми координатами **a** (означает положение цвета в диапазоне от зеленого (-128) до красного (127)) и **b** (означает положение цвета в диапазоне от синего (-128) до желтого (127)). Бинарное изображение получается при нулевых

значениях координат  $a$  и  $b$ . Идея алгоритма состоит в разбиении цветного изображения на сегменты доминирующих цветов.

В качестве исходных данных выберем следующее цветное изображение:



Рис. 3.1 — Исходное цветное изображение.

В первую очередь, чтобы уменьшить влияние освещенности на результат сегментации, преобразуем полноцветное изображение из цветового пространства **RGB** в пространство **Lab**. Для этого в среде MATLAB используется функция `rgb2lab()`.

**Листинг 3.6.** Сегментация на основе цветового пространства **Lab**.

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

На следующем шаге необходимо определить количество цветов, на которые будет сегментировано изображение, и задать области, содержащие пиксели примерно одного цвета. Области можно задать для каждого цвета интерактивно в виде многоугольников при помощи функции `roipoly()`:

```
6 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
```

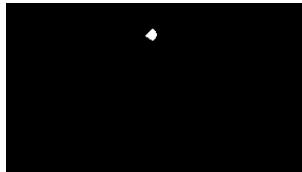


Рис. 3.2 — Пример выделенной красной области из четырех точек.

После этого требуется определить цветовые метки для каждого из сегментов путем расчета среднего значения цветности в каждой выделенной области. Средние значения можно вычислить при помощи функции `mean2()`:

```
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
```

Затем используем принцип ближайшей окрестности для классификации пикселей путем вычисления евклидовых метрик между пикселями и метками: чем меньше расстояние до метки, тем лучше пикセル соответствует данному сегменту. Евклидова метрика по двум цветовым координатам расчитывается по формуле:  $\sqrt{(a(x,y) - a_{mark})^2 + (b(x,y) - b_{mark})^2}$ . Для поиска минимального расстояния будем использовать функцию `min()`. Приведем листинг для поиска меток сегментов `label` для каждого пикселя:

```
20 distance = zeros([size(a), numColors]);
21 for i=1:1:numColors
22     distance(:,:,i) = ...
23         ((a-colorMarks(i,1)).^2 + ...
24             (b-colorMarks(i,2)).^2).^.0.5;
25     colorLabels = i;
26 end
27 [~, label] = min(distance, [], 3);
28 label = colorLabels(label);
```

Таким образом, матрица `label` размерности равной исходному изображению будет содержать идентификаторы классов для каждого пикселя. Для сегментации изображения на фрагменты `segmentedFrames` используем следующий листинг:

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



a)



б)

Рис. 3.3 — Сегментированные области: а) красные, б) желтые.

Отметим распределение сегментированных пикселей на координатной плоскости  $(a,b)$ :

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

### На основе кластеризации методом $k$ -средних

Идея метода заключается в определении центров  $k$ -кластеров и отнесении к каждому кластеру пикселей, наиболее близко относя-

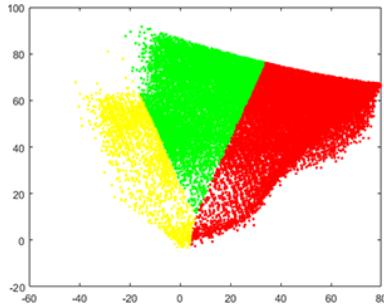


Рис. 3.4 — Распределение сегментированных пикселей на координатной плоскости  $(a,b)$ .

щихся к этим центрам. Все пиксели рассматриваются как векторы  $x_i, i = \overline{1,p}$ . Алгоритм сегментации состоит из следующих шагов:

1. Определение случайным образом  $k$  векторов  $m_j, j = \overline{1,k}$ , которые объявляются начальными центрами кластеров.
2. Обновление значений средних векторов  $m_j$  путем вычисления расстояний от каждого вектора  $x_i$  до каждого  $m_j$  и их классификации по критерию минимальности расстояния от вектора до кластера, пересчет средних значений  $m_j$  по всем кластерам.
3. Повторение шагов 2 и 3 до тех пора, пока центры кластеров не перестанут изменяться.

Реализация метода очень похожа на предыдущий подход и содержит ряд аналогичных действий (используем исходное изображение рис. 3.1). Будем работать в цветовом пространстве **Lab**, поэтому первым шагом перейдем из пространства **RGB** в **Lab**:

**Листинг 3.7.** Сегментация на основе кластеризации методом  $k$ -средних.

```

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

Рассмотрим координатную плоскость  $(a,b)$ . Для этого сформируем трехмерный массив **ab**, а затем функцией **reshape()** превратим его в двумерный вектор, содержащий все пиксели изображения:

```
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);
```

Кластеризация методом  $k$ -средних в среде MATLAB осуществляется функцией **kmeans()**. Аналогично предыдущему способу разобьем изображение на три области соответствующих цветов. Используем евклидову метрику (параметр ‘**distance**’ со значением ‘**sqEuclidean**’ и для повышения точности повторим процедуру кластеризации три раза (параметр ‘**Replicates**’ со значением 3)):

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

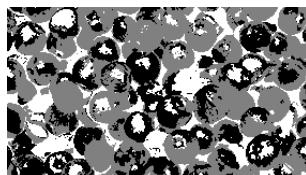


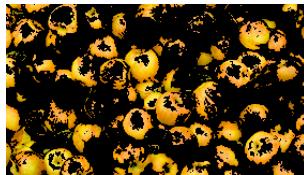
Рис. 3.5 — Метки классов.

Матрица **label** размера равном исходному изображению будет содержать идентификаторы классов для каждого пикселя. Для сегментации изображения на фрагменты **segmentedFrames** используем следующий листинг:

```
15 segmentedFrames = cell(1, 3);
16 rgblLabel = repmat(label, [1 1 3]);
17 for i = 1:k
18     color = I;
19     color(rgblLabel ~= i) = 0;
```



a)



б)

Рис. 3.6 — Сегментированные области: а) красные, б) желтые.

```
20     segmentedFrames{i} = color;
21     figure, imshow(segmentedFrames{i});
22 end
```

Массив L содержит значение о светлоте изображения. Используя эти данные можно, например, сегментированные красные области разделить на светло-красные и темно-красные сегменты.

### Текстурная сегментация

При текстурной сегментации для описания текстуры применяются три основных метода: статистический, структурный и спектральный. В лабораторной работе будем рассматривать статистический подход, который описывает текстуру сегмента как гладкую, грубую или зернистую. Характеристики соответствующих текстурам параметров приведены в табл. 3.1. Рассмотрим пример изображения, представленного на рис. 3.7, на котором имеется два типа текстур. Их разделение в общем случае невозможно выполнить с использованием только лишь простой бинаризации.



Рис. 3.7 — Исходное полутоновое изображение.

Будем рассматривать интенсивность изображения  $I$  как случайную величину  $z$ , которой соответствует вероятность распределе-

ния  $p(z)$ , вычисляемая из гистограммы изображения. Центральным моментом порядка  $n$  случайной величины  $z$  называется параметр  $\mu_n(z)$ , вычисляемый по формуле:

$$\mu_n(z) = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i), \quad (3.15)$$

где  $L$  — число уровней интенсивностей,  $m$  — среднее значение случайной величины  $z$ :

$$m = \sum_{i=0}^{L-1} z_i p(z_i). \quad (3.16)$$

Из 3.15 следует, что  $\mu_0 = 1$  и  $\mu_1 = 0$ . Для описания текстуры важна дисперсия случайной величины, равная второму моменту  $\sigma^2(z) = \mu_2(z)$  и являющаяся мерой яркостного контраста, которую можно использовать для вычисления признаков гладкости. Введем меру относительной гладкости  $R$ :

$$R = 1 - \frac{1}{1 + \sigma^2(z)}, \quad (3.17)$$

которая равна нулю для областей с постоянной интенсивностью (нулевой дисперсией) и приближается к единице для больших значений дисперсии  $\sigma^2(z)$ . Для полуточновых изображений с интервалом интенсивностей  $[0, 255]$  необходимо нормировать дисперсию до интервала  $[0, 1]$ , поскольку для исходного диапазона значения дисперсий будут слишком велики. Нормирование осуществляется делением дисперсии  $\sigma^2(z)$  на  $(L - 1)^2$ . В качестве характеристики текстуры также зачастую используется *стандартное отклонение*:

$$s = \sigma(z). \quad (3.18)$$

Третий момент является *характеристикой симметрии гистограммы*. Для оценки текстурных особенностей используется функция энтропии  $E$ , определяющая разброс интенсивностей соседних пикселей:

$$E = - \sum_{i=0}^{L-1} p(z) \log_2 p(z_i). \quad (3.19)$$

Таблица 3.1 — Значения параметров текстур.

Текстура	$m$	$s$	$R \in [0,1]$
Гладкая	82,64	11,79	0,0020
Грубая	143,56	74,63	0,0079
Периодическая	99,72	33,73	0,0170

Текстура	$\mu_3(z)$	$U$	$E$
Гладкая	-0,105	0,026	5,434
Грубая	-0,151	0,005	7,783
Периодическая	0,750	0,013	6,674

Еще одной важной характеристикой, описывающей текстуру, является *мера однородности*  $U$ , оценивающая равномерность гистограммы:

$$U = \sum_{i=0}^{L-1} p^2(z_i). \quad (3.20)$$

После вычисления какого-либо признака или набора признаков необходимо построить бинарную маску, на основе которой и будет производится сегментация изображения. Например, можно использовать энтропию  $E$  в окрестности каждого пикселя  $(x,y)$ . Для этого в среде MATLAB используется функция `entropyfilt()`, по умолчанию у которой используется окрестность размера  $9 \times 9$ . Для нормирования функции энтропии в диапазоне от 0 до 1 используем функцию `mat2gray()`, а для построения маски бинаризуем полученный нормированный массив `Eim` методом Отсю.

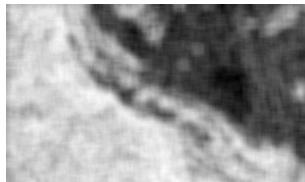
**Листинг 3.8.** Текстурная сегментация.

```

1 I = imread('pic3.jpg');
2 E = entropyfilt(I);
3 Eim = mat2gray(E);
4 BW1 = imbinarize(Eim, graythresh(Eim));

```

После этого используем морфологические фильтры (будут рассмотрены подробнее в лабораторной работе №6) сначала для удаления связных областей, содержащих менее заданного количества



а)



б)

Рис. 3.8 — а) Энтропия исходного изображения,  
б) бинаризованное изображение.

пикселей (функция `bwareaopen()`), а затем для удаления внутренних *дефектов формы* или «дырок» (функция `imclose()` со структурным элементом размера  $9 \times 9$ ). Оставшиеся крупные «дырки» заполним при помощи функции `imfill()`. Таким образом, получим маску:

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



а)

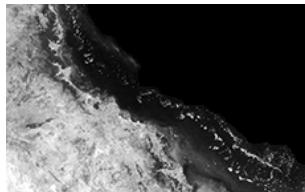


б)

Рис. 3.9 — а) Результат выполнения функции `bwareaopen()`;  
б) результат выполнения функции `imclose()`.

Применив полученную маску к исходному изображению выделим сегменты воды и сушки.

Границу между текстурами рассчитаем с использованием функции определения периметра `bwperim()`:



a)



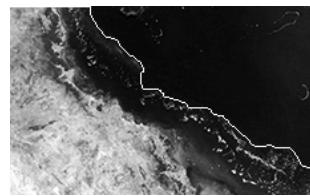
б)

Рис. 3.10 — а) Текстура сушки, б) текстура воды.

```
9 boundary = bwperim(Mask1);
10 segmentResults = I;
11 segmentResults(boundary) = 255;
```



а)



б)

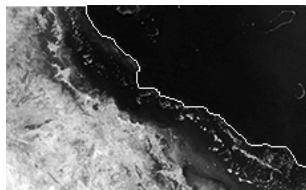
Рис. 3.11 — а) Результат выполнения функции imfill(),
б) выделенная граница функцией bwperim().

Аналогичный подход можно применить для построения маски относительно суши:

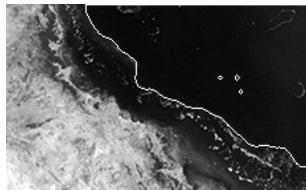
```
12 I2 = I;
13 I2(Mask1) = 0;
14 E2 = entropyfilt(I2);
15 E2im = mat2gray(E2);
16 BW2 = imbinarize(E2im, graythresh(E2im));
17 Mask2 = bwareaopen(BW2, 2000);
18 boundary = bwperim(Mask2);
19 segmentResults = I;
20 segmentResults(boundary) = 255;
```

Найдем текстуры сушки и воды:

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



а)



б)

Рис. 3.12 — а) Результат сегментации относительно воды,  
б) результат сегментации относительно сушки.

## Порядок выполнения работы

1. *Бинаризация.* Выбрать произвольное изображение. Выполнить бинаризацию изображения при помощи рассмотренных методов. В зависимости от изображения использовать бинаризацию по верхнему или нижнему порогу.
2. *Сегментация 1.* Выбрать произвольное изображение, содержащее лицо(-а). Выполнить сегментацию изображения либо по принципу Вебера, либо на основе цвета кожи (на выбор).
3. *Сегментация 2.* Выбрать произвольное изображение, содержащее ограниченное количество цветных объектов. Выполнить сегментацию изображения в пространстве **CIE Lab** либо по методу ближайших соседей, либо по методу  $k$ -средних (на выбор).
4. *Сегментация 3.* Выбрать произвольное изображение, содержащее две разнородные текстуры. Выполнить текстурную сегментацию изображения, оценить не менее трех параметров

выделенных текстур, определить к какому классу относятся текстуры.

## Содержание отчета

1. Цель работы.
2. Теоретическое обоснование применяемых методов и функций сегментации изображений.
3. Ход выполнения работы:
  - (a) Исходные изображения;
  - (b) Листинги программных реализаций;
  - (c) Комментарии;
  - (d) Результирующие изображения.
4. Выводы о проделанной работе.

## Вопросы к защите лабораторной работы

1. В каких случаях целесообразно использовать сегментацию по принципу Вебера?
2. Какие значения имеют цветовые координаты  $a$  и  $b$  цветового пространства **CIE Lab** в полутновом изображении?
3. Зачем производить сегментацию в цветовом пространстве **CIE Lab**, а не в исходном **RGB**?
4. Что такое *цветовое пространство* и *цветовой охват*?

# Лабораторная работа №2

## Преобразование Хафа

### Цель работы

Освоение преобразования для поиска геометрических примитивов.

### Методические рекомендации

До начала работы студенты должны ознакомиться с функциями среды MATLAB для работы с преобразованием Хафа. Знать о подходе «голосования» точек. Лабораторная работа рассчитана на 5 часов.

### Теоретические сведения

Идея преобразования Хафа (англ. Hough, возможные варианты перевода Хох, Хо) заключается в поиске общих *геометрических мест точек* (ГМТ). Например, данный подход используется при построении треугольника по трем заданным сторонам, когда сначала откладывается одна сторона треугольника, после этого концы отрезка рассматриваются как центры окружностей радиусами равными длинам второго и третьего отрезков. Место пересечения двух окружностей является общим ГМТ, откуда и проводятся отрезки до концов первого отрезка. Иными словами можно сказать, что было проведено *голосование* двух точек в пользу вероятного расположения третьей вершины треугольника. В результате «голосования» «победила» точка, набравшая два «голоса» (точки на окружностях набрали по одному голосу, а вне их — по нулю).

Обобщим данную идею для работы с реальными данными, когда на изображении имеется большое количество особых характеристических точек, участвующих в голосовании. Допустим, необходимо найти в бинарном точечном множестве окружность известного радиуса  $R$ , причем в данном множестве могут присутствовать и ложные точки, не лежащие на искомой окружности. Набор центров возможных окружностей искомого радиуса вокруг каждой характеристической точки образует окружность радиуса  $R$ , см. рис. 4.2.

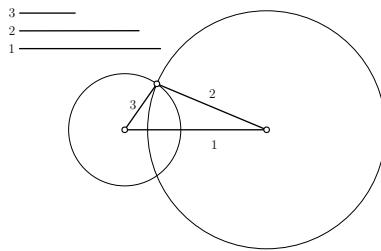


Рис. 4.1 — Построение треугольника по трем заданным сторонам.

Таким образом, точка, соответствующая максимальному пересечению числа окружностей, и будет являться центром окружности искомого радиуса.

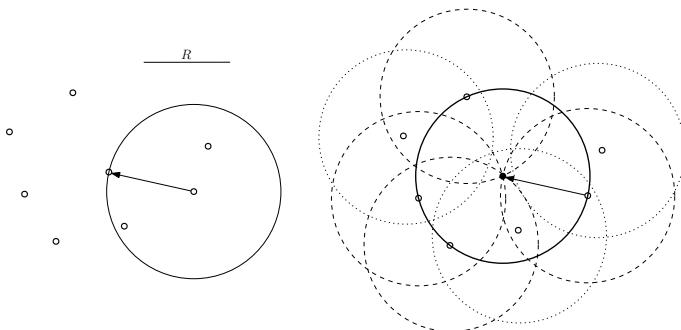


Рис. 4.2 — Обнаружение окружности известного радиуса в точечном множестве.

Классическое преобразование Хафа, базирующееся на рассмотренной идеи голосования точек, изначально было предназначено для выделения прямых на бинарных изображениях. В преобразовании Хафа для поиска геометрических примитивов используется пространство параметров. Самым распространенным параметрическим уравнением прямых является:

$$y = kx + b, \quad (4.1)$$

$$x \cos \Theta + y \sin \Theta = \rho, \quad (4.2)$$

где  $\rho$  — радиус-вектор, проведенный из начала координат до прямой;  $\Theta$  — угол наклона радиус-вектора.

Пусть в декартовой системе координат прямая задана уравнением (4.1), из которого легко вычислить радиус-вектор  $\rho$  и угол  $\Theta$  (4.2). Тогда в пространстве параметров Хафа прямая будет представлена точкой с координатами  $(\rho_0, \Theta_0)$ , см. рис. 4.3.

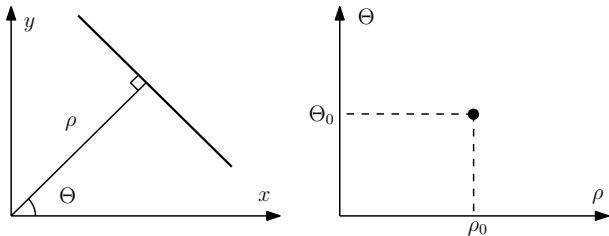


Рис. 4.3 — Представление прямой в пространстве Хафа.

Подход преобразования Хафа заключается в том, что для каждой точки пространства параметров суммируется количество голосов, поданных за нее, поэтому в дискретном виде пространство Хафа называется *аккумулятором* и представляет собой некоторую матрицу  $A(\rho, \Theta)$ , хранящую информацию о голосовании. Через каждую точку в декартовой системе координат можно провести бесконечное число прямых, совокупность которых породит в пространстве параметров синусоидальную функцию отклика. Таким образом, любые две синусоидальные функции отклика в пространстве параметров пересекутся в точке  $(\rho, \Theta)$  только в том случае, если порождающие их точки в исходном пространстве лежат на прямой, см. рис. 4.4. Исходя из этого можно сделать вывод, что для того, чтобы найти прямые в исходном пространстве, необходимо найти все локальные максимумы аккумулятора.

Рассмотренный алгоритм поиска прямых может быть таким же образом использован для поиска любой другой кривой, описываемой в пространстве некоторой функцией с определенным числом параметров  $F = (a_1, a_2, \dots, a_n, x, y)$ , что повлияет лишь на размерность пространства параметров. Воспользуемся преобразованием

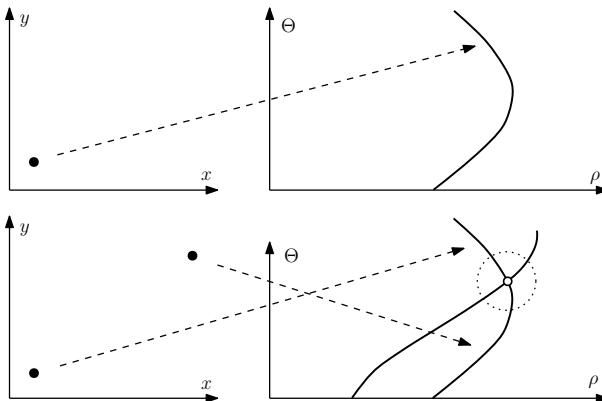


Рис. 4.4 — Процедура голосования.

Хафа для поиска окружностей заданного радиуса  $R$ . Известно, что окружность на плоскости описывается формулой  $(x - x_0)^2 + (y - y_0)^2 = R^2$ . Набор центров всех возможных окружностей радиуса  $R$ , проходящих через характеристическую точку, образует окружность радиуса  $R$  вокруг этой точки, поэтому функция отклика в преобразовании Хафа для поиска окружностей представляет окружность такого же размера с центром в голосующей точке. Тогда аналогично предыдущему случаю необходимо найти локальные максимумы аккумуляторной функции  $A(x,y)$  в пространстве параметров  $(x,y)$ , которые и будут являться центрами искомых окружностей.

Преобразование Хафа инвариантно к сдвигу, масштабированию и повороту. Учитывая, что при проективных преобразованиях трехмерного пространства прямые линии всегда переходят только в прямые линии (в вырожденном случае — в точки), преобразование Хафа позволяет обнаруживать линии инвариантно не только к аффинным преобразованиям плоскости, но и к группе проективных преобразований в пространстве.

Пусть задано некоторое изображение. Выделим контуры алгоритмом Кэнни и выполним преобразование Хафа функцией `hough()`.

**Листинг 4.1.** Поиск прямых преобразованием Хафа.

```

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

```

Вычислим пики функцией `houghpeaks()` в пространстве Хафа и нанесем их на полученное изображение функций откликов:

```

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');

```

Определим на основе пиков прямые функцией `houghlines()` и нанесем их на исходное изображение:

```

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

```

Для поиска окружностей преобразованием Хафа можно воспользоваться функцией `imfindcircles(I,R)`.

## Порядок выполнения работы

- Поиск прямых.* Выбрать три произвольных изображения, содержащие прямые. Осуществить поиск прямых с помощью преобразования Хафа как для исходного изображения, так и для изображения, полученного с помощью использования какого-либо дифференциального оператора. Отразить найденные линии на исходном изображении. Отметить точки на-

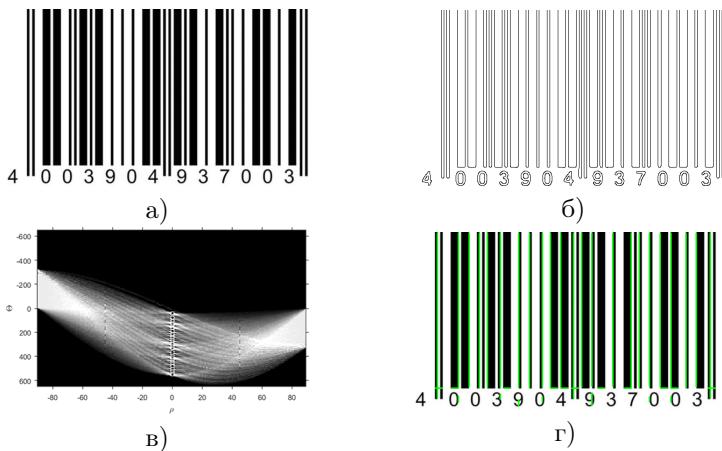


Рис. 4.5 — а) Исходное изображение, б) обработанное алгоритмом Кэнни, в) пространство параметров, г) выделенные линии.

чала и окончания линий. Определить длины самой короткой и самой длинной прямых, вычислить количество найденных прямых.

2. *Поиск окружностей.* Выбрать три произвольных изображения, содержащие окружности. Осуществить поиск окружностей как определенного радиуса, так и из заданного диапазона с помощью преобразования Хафа как для исходного изображения, так и для изображения, полученного с помощью использования какого-либо дифференциального оператора. Отразить найденные окружности на исходном изображении.

## Содержание отчета

1. Цель работы.
2. Теоретическое обоснование применяемого преобразования для поиска геометрических примитивов.
3. Ход выполнения работы:

- (a) Исходные изображения;
  - (b) Листинги программных реализаций;
  - (c) Комментарии;
  - (d) Результирующие изображения.
4. Выводы о проделанной работе.

## Вопросы к защите лабораторной работы

1. Какая идея лежит в основе преобразования Хафа?
2. Можно ли использовать преобразование Хафа для поиска произвольных контуров, которые невозможно описать аналитически?
3. Что такое *рекуррентное* и *обобщенное* преобразования Хафа?
4. Какие бывают способы параметризации в преобразовании Хафа?

# 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. 5.1



Рис. 5.1 — Building image

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. 5.2.

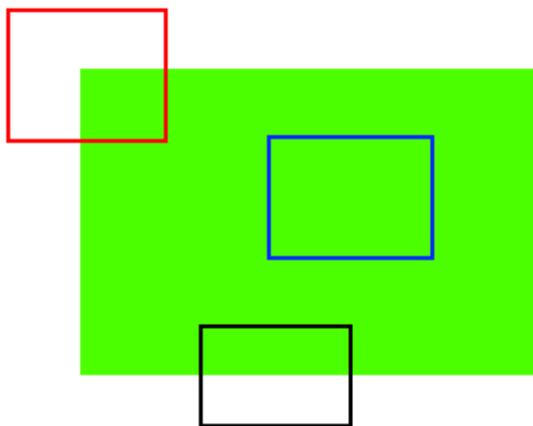


Рис. 5.2 — Feature points

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

- Harris corner detector [?].
- Shi-Tomasi corner detector [?].

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

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 [?]. 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. 5.3.

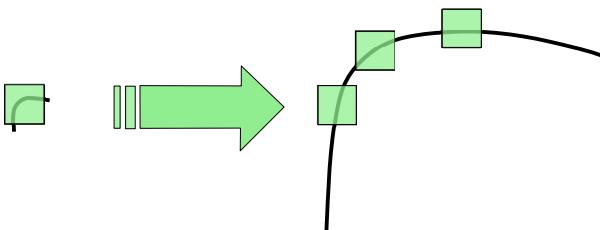


Рис. 5.3 — Dependence of the corner feature on the scale

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  $\sigma$  value, which is calculated by convolution of the variable-scale Gaussian  $G(x, y, \sigma)$  with an input image  $I(x, y)$ :

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

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (5.1)$$

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. 5.4:

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) =$$

$$= L(x, y, k\sigma) - L(x, y, \sigma) \quad (5.2)$$

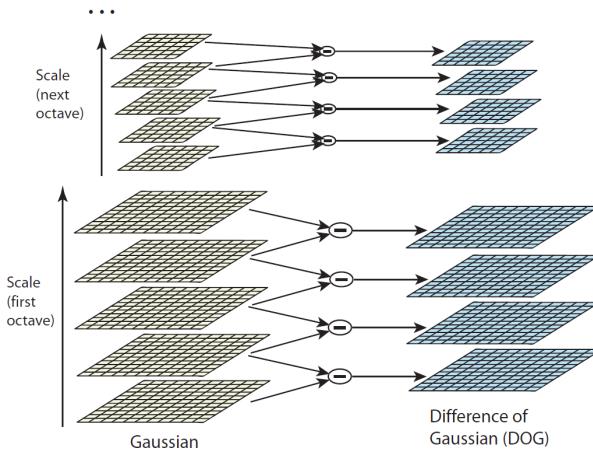


Рис. 5.4 — Difference of Gaussian calculation

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. 5.5

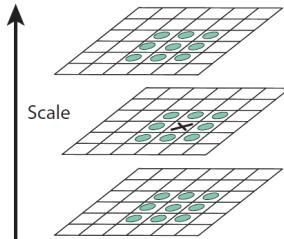


Рис. 5.5 — Selecting DoG maxima

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  $360^\circ$ .

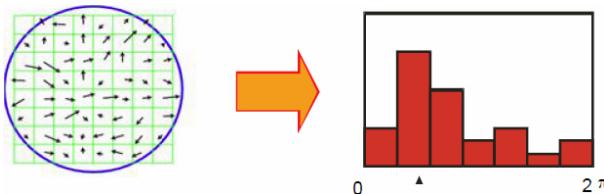


Рис. 5.6 — Selecting characteristic orientation

Then the patch is rotated according to the characteristic orientation and a descriptor is built by computing 16 histograms for  $4 \times 4$  subwindows of a  $16 \times 16$  pixels window around the feature point, see Fig 5.7.

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

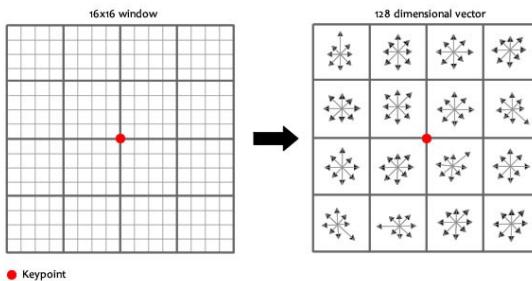


Рис. 5.7 — SIFT descriptor

## SIFT detector with MATLAB

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

**Listing 5.1.** 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 5.2.** 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 5.3.** 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. 5.8.

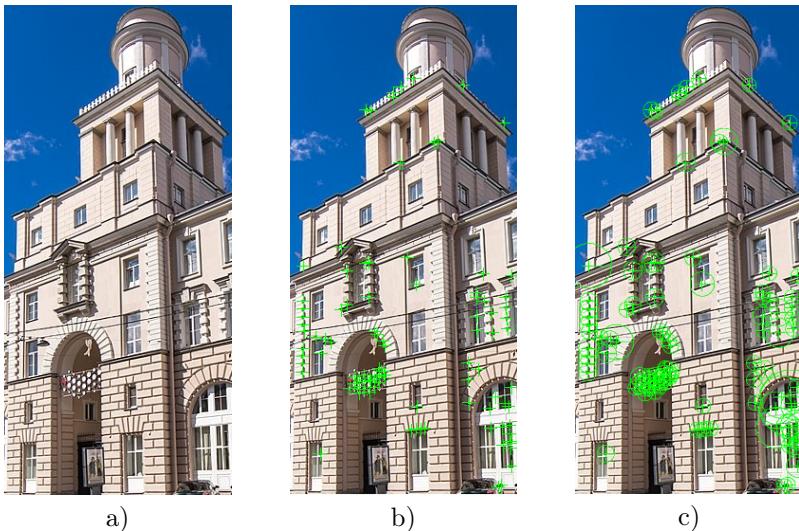


Рис. 5.8 — SIFT feature detector with MATLAB: a) Source image, b) 100 strongest SIFT feature points, c) 100 strongest SIFT feature points with the scale and orientation.

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

**Listing 5.4.** 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. 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 5.5.** 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 5.6.** Detecting SIFT 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 sift = cv2.SIFT_create()
6 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 5.7.** Detecting 100 strongest SIFT feature points with OpenCV and C++.

```

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

```

**Listing 5.8.** Detecting 100 strongest SIFT feature points with OpenCV and Python.

```

1 sift = cv2.SIFT_create(100)
2 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 5.9.** Displaying SIFT feature points with OpenCV and C++.

```

1 cv::Mat Iout;
2 cv::drawKeypoints(I, Ifp, Iout);
3 cv::imshow("SIFTdetector", Iout);

```

**Listing 5.10.** Displaying SIFT feature points with OpenCV and Python.

```

1 Iout = cv2.drawKeypoints(I, Ifp, None)
2 cv2.imshow("SIFTdetector", 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 5.11.** Displaying SIFT feature points in green color with scale and orientation with OpenCV and C++.

```
1  cv::Mat Iout;
2  cv::drawKeypoints(I, Ifp, Iout, color,
3    cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
4  cv::imshow("SIFTdetector", Iout);
```

**Listing 5.12.** Displaying SIFT feature points in green color with scale and orientation with OpenCV and Python.

```
1  Iout = cv2.drawKeypoints(I, Ifp, None,
2    color = (0, 255, 0), flags =
3    cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
4  cv2.imshow("SIFTdetector", Iout)
```

An example of 100 strongest SIFT feature points detected with OpenCV are shown in fig. 5.9.



Рис. 5.9 — 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 [?] 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 uses 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 [?] is a bit string description of an image patch constructed from a set of binary intensity tests:

$$\tau(p, x, y) = \begin{cases} 1, & p(x) < p(y), \\ 0, & p(x) \geq p(y). \end{cases} \quad (5.3)$$

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

$$f_n(p) = \sum_{i=1}^n 2^{i-1} \tau(p, x_i, y_i). \quad (5.4)$$

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.

$$S = \begin{pmatrix} x_1, \dots, x_n \\ y_1, \dots, y_n \end{pmatrix}. \quad (5.5)$$

Then using the orientation  $\theta$  of a patch its rotation matrix  $R_\theta$  is calculated and used to rotate the  $S$  matrix to get a rotated version  $S_\theta$ .

$$S_\theta = R_\theta S. \quad (5.6)$$

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

$$g_n(p, \theta) = f_n(p)|(x_i, y_y) \in S_\theta. \quad (5.7)$$

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 [?]. This function works with grayscale images also. The rest of listings are the same to the listings 5.3–5.4. An example of 100 strongest ORB feature points are shown in fig. 5.10.

## ORB detector with OpenCV

OpenCV provides a class for detection of ORB feature points and calculation of corresponding descriptors. 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 5.13.** 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;
```

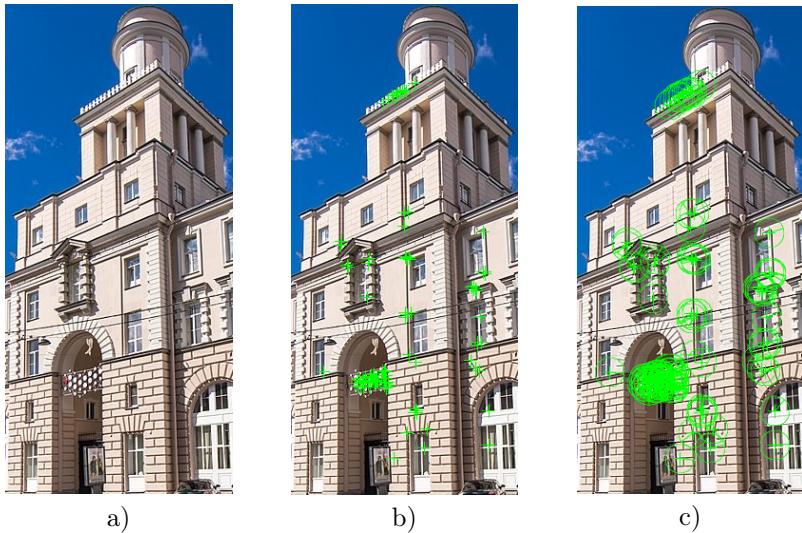


Рис. 5.10 — ORB feature detector with MATLAB: a) Source image, b) 100 strongest ORB feature points, c) 100 strongest ORB feature points with the scale and orientation.

```

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("ORBdetector", Iout);

```

**Listing 5.14.** 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("ORBdetector", Iout)

```

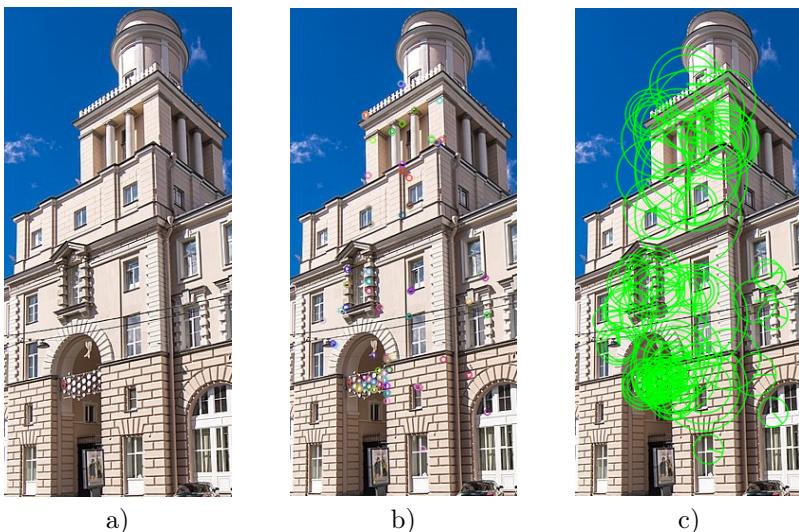


Рис. 5.11 — ORB feature detector with OpenCV: a) Source image, b) 100 strongest ORB feature points in default colors, c) 100 strongest ORB feature points with the scale and orientation in green color.

### 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:

$$D_1 < r \cdot D_2, \quad (5.8)$$

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  $\theta_i$  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  $\theta_i$  by applying the perspective transformation matrix  $M$  to all points of the first  $X$  set and checking their matches with the points of the second  $Y$  set with some threshold. The number of matches is the hypothesis evaluation score  $R(\theta_I)$ :

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

$$p(\theta, x, Y) = \begin{cases} 1, & |\varepsilon(\theta, x, Y)| \leq T, \\ 0, & |\varepsilon(\theta, x, Y)| > T, \end{cases} \quad (5.9)$$

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

- (c) If this is the first hypothesis, then store it as a current best hypothesis  $\theta_0$ . Else, check if the current hypothesis  $\theta_i$  is better than the best one found before  $\theta_0$ , and, if so, then it is stored as the new best hypothesis.

$$(i = 0) \vee (R(\theta_i) > R(\theta_0)) \Rightarrow \theta_0 = \theta_i. \quad (5.10)$$

3. After finishing  $N$  iterations, the  $\theta_0$  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:

$$p = 1 - (1 - N(1 - e)^s)^N, \quad (5.11)$$

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:

$$N = \frac{\log(1-p)}{\log(1 - (1-e)^s)}. \quad (5.12)$$

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  $\theta$ :

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

$$p(\theta, x, Y) = \begin{cases} \varepsilon^2(\theta, x, Y), |\varepsilon(\theta, x, Y)| \leq T, \\ T^2, |\varepsilon(\theta, x, Y)| > T. \end{cases} \quad (5.13)$$

### Feature point descriptors matching with MATLAB

Let's consider an example, see fig. 5.12. We have two figures: an object and the scene with this object. Scene has some affine geometric transformations.



Рис. 5.12 — a) Figure 1 — object, b) Figure 2 — scene.



Рис. 5.13 — a) An object with 30 strongest features, b) Scene with 100 strongest features.

At first step of matching, we should calculate descriptors of our figures. See fig. 5.13

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. 5.12). After that, you should take into account only putatively matched points (including outliers) from these figures using array `pairs`.

**Listing 5.15.** 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 listing 5.16 and the result in fig. 5.14.

**Listing 5.16.** 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.

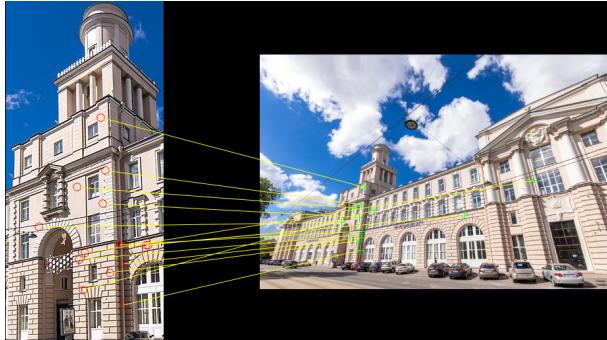


Рис. 5.14 — Putatively matched points (including outliers)

**Listing 5.17.** Estimation of geometric transformation

```

7   [tform, inliersIds] = ...
8     estimateGeometricTransform2D(... 
9       matchedPoints1, ...
10      matchedPoints2, 'affine');
11    inlierPoints1 = ...
12      matchedPoints1(inliersIds, :);
13    inlierPoints2 = ...
14      matchedPoints2(inliersIds, :);

```

where `tform` — transformation matrix.

The result of using only inliers is shown in fig. 5.15.

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 5.18.** Bounding polygon for the object

```

15  boxPolygon = [1, 1;... % top-left
16    size(I1, 2), 1;... % top-right
17    size(I1, 2), size(I1, 1);... % bottom-right
18    1, size(I1, 1);... % bottom-left
19    1, 1]; % top-left
20                      % again to close the polygon

```



Рис. 5.15 — 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.



Рис. 5.16 — Detected object

**Listing 5.19.** 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. 5.12, 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 5.20.** 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 5.21.** Detecting and computing SIFT feature points with OpenCV and Python.

```
1  I1 = cv2.imread("pic1.jpg",
2      cv2.IMREAD_COLOR)
3  I2 = cv2.imread("pic2.jpg",
4      cv2.IMREAD_COLOR)
5  I1gray = cv2.cvtColor(I1,
6      cv2.COLOR_BGR2GRAY)
7  I2gray = cv2.cvtColor(I2,
8      cv2.COLOR_BGR2GRAY)
9  sift = cv2.SIFT_create()
10 I1fp, I1des = \
11     sift.detectAndCompute(I1gray, None)
12 I2fp, I2des = \
13     sift.detectAndCompute(I2gray, None)
```



a)



b)

Рис. 5.17 — 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 matcher 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 5.22.** 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 5.23.** 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 5.24.** 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 5.25.** 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 5.26.** Creating FLANN 5 KD-trees descriptor matcher for SIFT descriptors with OpenCV and C++.

```

1   cv::Ptr<cv::DescriptorMatcher> matcher;
2   matcher =
3       cv::makePtr<cv::FlannBasedMatcher>(
4           cv::makePtr<cv::flann::KDTreeIndexParams>(
5               5));

```

**Listing 5.27.** Creating FLANN 5 KD-trees descriptor matcher for SIFT descriptors with OpenCV and Python.

```

1   FLANN_INDEX_KDTREE = 1
2   index_params = \
3       dict(algorithm = FLANN_INDEX_KDTREE,
4             trees = 5)
5   matcher = \
6       cv2.FlannBasedMatcher(index_params,
7             dict())

```

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

**Listing 5.28.** Creating FLANN LSH descriptor matcher for ORB descriptors with OpenCV and C++.

```

1   cv::Ptr<cv::DescriptorMatcher> matcher;
2   matcher =
3       cv::makePtr<cv::FlannBasedMatcher>(
4           cv::makePtr<cv::flann::LshIndexParams>(
5               6, 12, 1));

```

**Listing 5.29.** Creating FLANN LSH descriptor matcher for ORB descriptors with OpenCV and Python.

```

1   FLANN_INDEX_LSH = 6
2   index_params = \
3       dict(algorithm = FLANN_INDEX_LSH,
4             table_number = 6, key_size = 12,
5             multi_probe_level = 1)
6   matcher =
7   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 5.30.** 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 5.31.** 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 5.18 5.19 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 distance between first nearest match is significantly lower comparing to distance

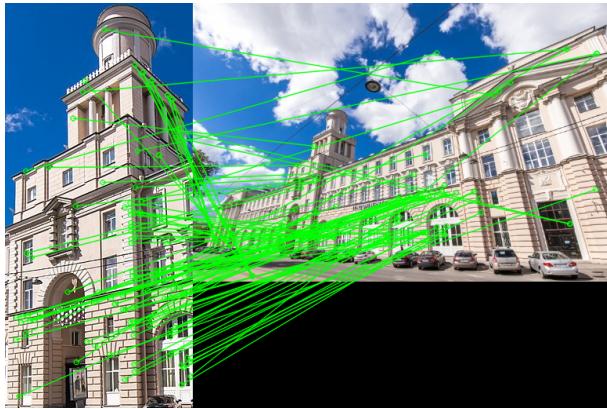


Рис. 5.18 — 100 strongest matches with brute force method and SIFT descriptors

with the second nearest match. With OpenCV this method can be implemented as following:

**Listing 5.32.** Finding k-nearest best match for two sets of descriptors and filtering them with OpenCV and C++.

```
1 std::vector<cv::DMatch> matches;
2 std::vector<std::vector<cv::DMatch>>
3     knn_matches;
4 // Find KNN matches with k = 2
5 matcher->knnMatch(I1des, I2des,
6     knn_matches, 2);
7 // Select good matches
8 double knn_ratio = 0.75;
9 for (int m = 0; m < knn_matches.size(); m++)
10    if (knn_matches[m].size() > 1)
11        if (knn_matches[m][0].distance <
12            knn_ratio *
13                knn_matches[m][1].distance)
14            matches.push_back(knn_matches[m][0]);
```

**Listing 5.33.** Finding k-nearest best match for two sets of descriptors and filtering them with OpenCV and Python.

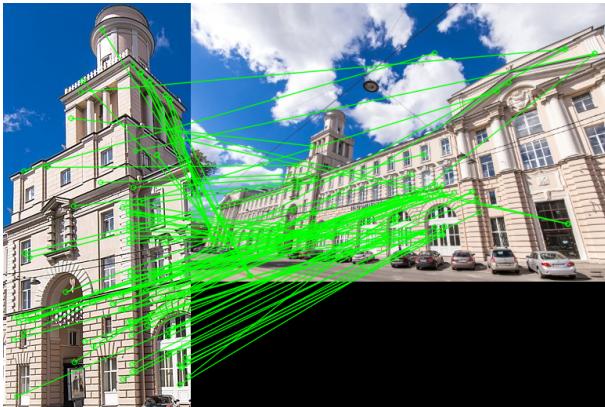


Рис. 5.19 — 100 strongest matches with FLANN method and SIFT descriptors

```

1 # Find KNN matches with k = 2
2 matches = \
3     matcher.knnMatch(I1des, I2des, k = 2)
4 # Select good matches
5 knn_ratio = 0.75
6 good = []
7 for m in matches:
8     if len(m) > 1:
9         if m[0].distance < \
10             knn_ratio * m[1].distance:
11                 good.append(m[0])
12 matches = good

```

Figures 5.20–5.21 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 `imgIdx` parameter of the `DMatch`. However since we are matching only two images we don't need this extra data.



Рис. 5.20 — All matches with brute force method, K-nearest ratio filter and SIFT descriptors

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 5.34.** 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),
13                           std::vector<char>(0),
```



Рис. 5.21 — All matches with FLANN method, K-nearest ratio filter and SIFT descriptors

```
14     cv::DrawMatchesFlags::
15         NOT_DRAW_SINGLE_POINTS);
```

**Listing 5.35.** 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 5.36.** Executing RANSAC to calculate the transformation matrix with OpenCV and C++.

```
1 const int MIN_MATCH_COUNT = 10;
2 if (matches.size() < MIN_MATCH_COUNT)
3 {
4     std::cout << "Not enough matches.\n";
5     return;
6 }
7 // Create arrays of point coordinates
8 std::vector<cv::Point2f> I1pts, I2pts;
9 for (int m = 0; m < matches.size(); m++)
10 {
11     I1pts.push_back(
12         I1fp[matches[m].queryIdx].pt);
13     I2pts.push_back(
14         I2fp[matches[m].trainIdx].pt);
15 }
16 // Run RANSAC method
17 std::vector<char> mask;
18 cv::Mat M = cv::findHomography(I1pts, I2pts,
19                                cv::RANSAC, 5, mask);
```

**Listing 5.37.** Executing RANSAC to calculate the transformation matrix with OpenCV and Python.

```
1 MIN_MATCH_COUNT = 10
2 if len(matches) < MIN_MATCH_COUNT:
3     print("Not enough matches.")
4     return
5 # Create arrays of point coordinates
6 I1pts = np.float32([I1fp[m.queryIdx].pt
7                     for m in matches]).reshape(-1, 1, 2)
8 I2pts = np.float32([I2fp[m.trainIdx].pt
9                     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 5.38.** 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 5.39.** Displaying the location of the first image on the second one with OpenCV and Python.

```

1 # Image corners
2 h, w = I1.shape[:2]
3 I1box = np.float32([[0, 0], [0, h - 1],
4                     [w - 1, h - 1], [w - 1, 0]]). \

```

```

5      reshape(-1, 1, 2)
6  I1to2box = \
7      cv2.perspectiveTransform(I1box, M)
8  # Draw a red box on the second image
9  I2res = cv2.polyline(I2,
10     [np.int32(I1to2box)], True, (0, 0, 255),
11     1, cv2.LINE_AA)
12  cv2.imshow("Search result", I2res)

```



Рис. 5.22 — 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 5.40.** Displaying inlier matches with OpenCV and C++.

```

1  cv::Mat Itrans;
2  cv::drawMatches(I1, I1fp, I2res, I2fp,
3      matches, Itrans, cv::Scalar(0, 255, 0),
4      cv::Scalar(-1), mask,
5      cv::DrawMatchesFlags::
6      NOT_DRAW_SINGLE_POINTS);
7  cv::imshow("Transformation", Itrans);

```

**Listing 5.41.** Displaying inlier matches with OpenCV and Python.

```

1 Itrans = \
2     cv2.drawMatches(I1, I1fp, I2res, I2fp,
3         matches, None, matchesMask = mask, flags =
4         cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS,
5         matchColor = (0,255,0))
6 cv2.imshow("Transformation", Itrans)

```

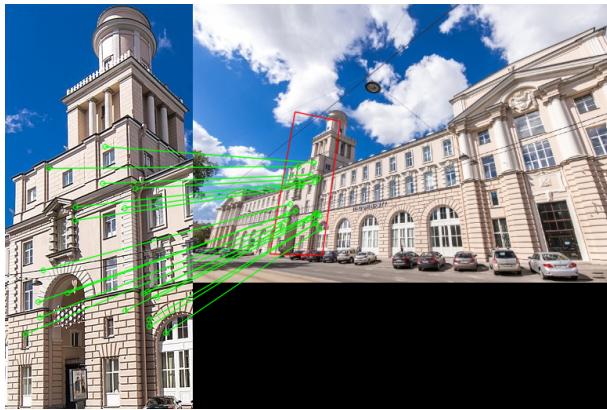


Рис. 5.23 — Inliers found by the RANSAC method

## 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 [?] 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. 6.1 are used. To calculate the value of the

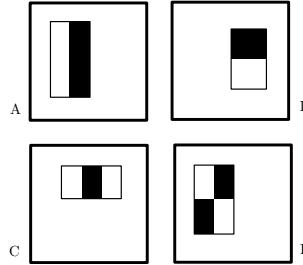


Рис. 6.1 — Haar-like features used in Viola-Jones face detector

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

$$value = \sum (pixels \text{ in black area}) - \sum (pixels \text{ in white area}). \quad (6.1)$$

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

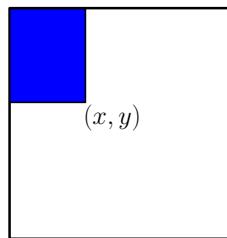


Рис. 6.2 — Integral image

arbitrary rectangle we need to access four pixels of an integral image which are located at the corners of the rectangle, see Fig. 6.3.

$$sum = D - B - C + A, \quad (6.2)$$

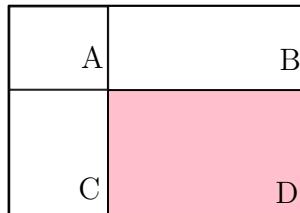


Рис. 6.3 — Rectangular sum calculation with an integral image

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.

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.

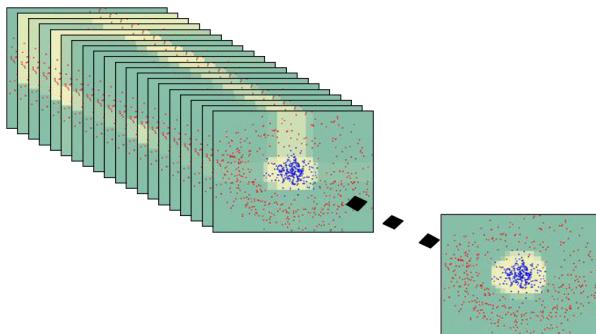


Рис. 6.4 — Combining weak classifiers into a strong classifier

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):
 
$$\epsilon_k = \Pr_{i \sim D_k}[h_k(x_i) \neq y_i]. \quad (6.3)$$
  - (b) Calculate the weight of the currently selected weak classifier basing on its classification error probability:
 
$$\alpha_k = \frac{1}{2} \ln \left( \frac{1 - \epsilon_k}{\epsilon_k} \right). \quad (6.4)$$
  - (c) Reweigh the training set with new weights:

$$D_{k+1}(i) = \frac{D_k(i)}{Z_k} \cdot \begin{cases} e^{-\alpha_k}, & h_k(x_i) = y_i, \\ e^{\alpha_k}, & h_k(x_i) \neq y_i. \end{cases} \quad (6.5)$$

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

$$H(x) = \text{sign} \left( \sum_{k=1}^K \alpha_k h_k(x) \right). \quad (6.6)$$

## 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

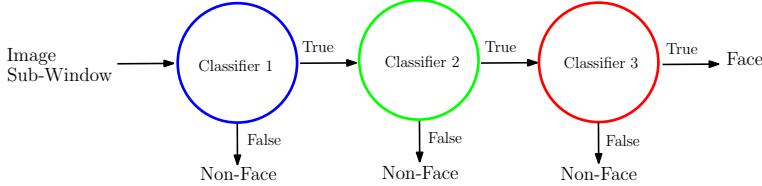


Рис. 6.5 — Cascade classifier

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. 6.5.

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 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:

$$TP = \prod_i TP_i. \quad (6.7)$$

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

$$FP = \prod_i FP_i. \quad (6.8)$$

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 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 [?]. 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 6.1.** Create an object using `CascadeObjectDetector`.

```
1 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 6.2.** Applying detector to the image with MATLAB.

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



Рис. 6.6 — 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. 6.7.

**Listing 6.3.** Applying detector to the image with MATLAB.

```

4 IFaces = insertObjectAnnotation(I, ...
5     'rectangle',bboxes,'Face');
6 imshow(IFaces);

```



Рис. 6.7 — Image with detected faces

MATLAB provides several pretrained classifiers, e.g. `UpperBody`, `EyePairSmall`, `Mouth`, or `Nose`. The complete list you can find in [?]. Specify the region of interest (ROI) in the detector `detector(I,roi)` 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 6.4.** Create an object with `EyePairSmall` classification model and ROI.

```

7 eyesDetector = ...
8 vision.CascadeObjectDetector(... ...
9 'ClassificationModel','EyePairSmall',...
10 '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 6.5.** Eyes detector using ROI in faces.

```

11 counter = 1;
12 for i = 1:length(bboxes)

```

```

13     temp = eyesDetector(I, bboxes(i,:));
14     if ~isempty(temp)
15         bboxes2(counter,:) = temp;
16         counter = counter + 1;
17     end
18 end

```

After that we can put labels of eyes to the previous image with faces' annotations, see Fig. 6.8.

**Listing 6.6.** Display eyes in the image with faces.

```

19 IEyes = insertObjectAnnotation(IFaces, ...
20     'rectangle',bboxes2,'Eyes',...
21     'Color','magenta');
22 imshow(IEyes)

```



Рис. 6.8 — 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 6.7.** Create and load a cascade classifier faces with OpenCV and C++.

```
1 cv::CascadeClassifier detector;
2 cv::String cascade_fn =
3     cv::samples::findFile(
4         "haarcascade_frontalface_default.xml");
5 detector.load(cascade_fn);
```

**Listing 6.8.** 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 [?] 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 [?] 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 6.9.** 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 6.10.** 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)
```



Рис. 6.9 — 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 6.11.** 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 6.12.** 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. 6.10).



Рис. 6.10 — Image with detected faces highlighted

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. 6.11

It's obvious that we don't need to scan the whole image for eyes as there would be a lot of false-positive results, so we will define a Region-Of-Interest (ROI) object as a face 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 its constructor that takes an 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 6.13.** 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(
```



Рис. 6.11 — Close up image of a face

```
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 {
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 6.14.** 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. 6.12.

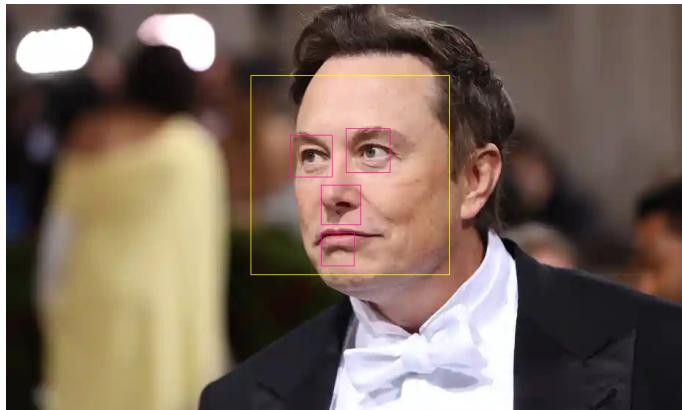


Рис. 6.12 — Eyes detection result

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

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

**Listing 6.16.** Define ROI and top 2/3 of the face image with OpenCV and Python.

```
1  Iface_top = \  
2  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. 6.13

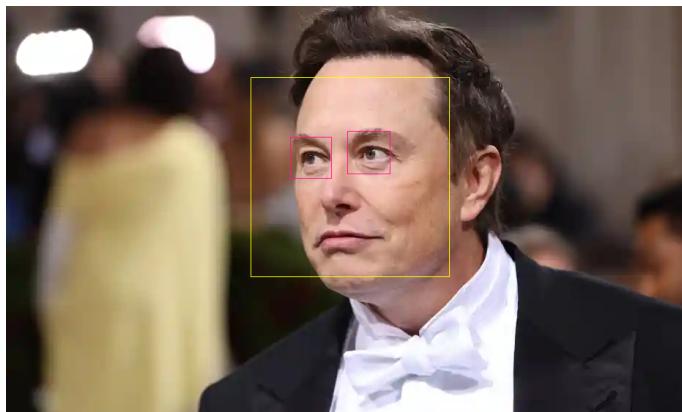


Рис. 6.13 — Eyes detection result in top 2/3 of the face

The same approach can be used when detecting other parts of the face, for example mouth is located at the bottom 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?

## Список литературы

- [1] *Журавель И.М.* Краткий курс теории обработки изображений: [Электронный ресурс]. URL: <https://hub.exponenta.ru/post/kratkiy-kurs-teorii-obrabotki-izobrazheniy734>. (Дата обращения: 22.03.2022).
- [2] *MATLAB Documentation.* Computer Vision System Toolbox: [Электронный ресурс]. URL: <https://www.mathworks.com/help/vision/index.html>. (Дата обращения: 22.03.2022).
- [3] *Шаветов С.В.* Основы технического зрения: учебное пособие / С.В. Шаветов. — Санкт-Петербург: НИУ ИТМО, 2017. — 86 с. — Текст: электронный // Лань: электронно-библиотечная система. — URL: <https://e.lanbook.com/book/110455> (дата обращения: 22.03.2022). — Режим доступа: для авториз. пользователей.
- [4] *Старовойтов В.В.* Цифровые изображения: от получения до обработки / Старовойтов В.В., Голуб Ю.И. — Минск: ОИПИ НАН Беларуси, 2014. — 202 с. — ISBN 978-985-6744-80-1.
- [5] *Визильтер Ю.В.* Обработка и анализ изображений в задачах машинного зрения: Курс лекций и практических занятий / Визильтер Ю.В., Желтов С.Ю., Бондаренко А.В., Ососков М.В., Моржин А.В. — М.: Физматкнига, 2010. — 672 с. — ISBN 978-5-89155-201-2.
- [6] *Визильтер Ю.В.* Обработка и анализ цифровых изображений с примерами на LabVIEW IMAQ Vision / Визильтер Ю.В., Желтов С.Ю., Князь В.А., Ходарев А.Н., Моржин А.В. — М.: ДМК Пресс, 2007. — 464 с. — ISBN 5-94074-404-4.

Шаветов Сергей Васильевич  
Жданов Андрей Дмитриевич

**Основы обработки изображений: лабораторный практикум, часть 2**

**Учебно-методическое пособие**

В авторской редакции

Редакционно-издательский отдел Университета ИТМО

Зав. РИО

Н.Ф. Гусарова

Подписано к печати

Заказ №

Тираж

Отпечатано на ризографе

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