ИТМО

С.В. Шаветов, А.Д. Жданов

ОСНОВЫ ОБРАБОТКИ ИЗОБРАЖЕНИЙ: ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Санкт-Петербург

2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

С.В. Шаветов, А.Д. Жданов

ОСНОВЫ ОБРАБОТКИ ИЗОБРАЖЕНИЙ: ЛАБОРАТОРНЫЙ ПРАКТИКУМ

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. — СПб: Университет ИТМО, 2022. — 147 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, начиная от представления изображений как массива чисел до морфологического анализа изображений. 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».

ИТМО

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: IT, 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.

Содержание

Введение

Задача технического зрения является одной из ключевых задач, которые необходимо решить для осуществления автоматизации и роботизации производственных процессов. Важным подразделом задач технического зрения является обработка и фильтрация изображений, необходимые для предварительной обработки цифровых изображений и видеоданных, поступающих с наружных камер роботизированной системы. В процессе выполнения данного лабораторного практикума учащиеся получат практические навыки и умения по обработке изображений, включающие в себя анализ и корректировку основных цвето-световых характеристик изображений, геометрические преобразования изображений, фильтрацию изображений, морфологический анализ изображений, использование алгоритмов цветовой сегментации и другие.

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
Гистограммы, профили и проекции

Цель работы

Освоение основных яркостных и геометрических характеристик изображений и их использование для анализа изображений.

Методические рекомендации

До начала работы студенты должны ознакомиться с основными функциями среды MATLAB или библиотеки OpenCV по работе с гистограммами, профилями и проекциями изображений. Лабораторная работа рассчитана на 5 часов.

Теоретические сведения

Пиксель цифрового изображения характеризуется тремя параметрами: (x,y,I), где пара целочисленных значений (x,y) описывает геометрическое положение пикселя в плоскости изображения, а значение I характеризует его яркость (интенсивность) в точке плоскости. Таким образом, в изображении можно выделить яркостную и геометрическую составляющие. В общем случае данные составляющие не связаны между собой (например, изменение в освещенности сцены не изменит геометрических параметров объектов на сцене). Из-за этого проще исследовать отдельно яркостные свойства изображения и отдельно — геометрические. Такой подход понижает порядок исследуемого изображения в случае геометрических свойств с $n = 3$ (x,y,I) до $n = 2$ (x,y), а в случае яркостных свойств — до $n = 1$ (I). Яркостная составляющая изображения характеризуется одномерным массивом гистограммы, из которого можно вычислить контраст.

Гистограмма — это распределение частоты встречаемости пикселей одинаковой яркости на изображении.

Яркость — это среднее значение интенсивности сигнала.

Контраст — это интервал значений между минимальной и максимальной яркостями изображения.

Для сведéния геометрических составляющих изображения к одномерному массиву данных $n = 1$ используются такие характеристики, как *«профили»* и *«проекции»* изображения.

Профиль вдоль линии — это функция интенсивности изображения, распределенного вдоль данной линии *(прорезки)*.

Проекция на ось — это сумма интенсивностей пикселей изображения взятая в направлении перпендикулярном данной оси.

Гистограмма изображения

Для 8-битного полутонового изображения гистограмма яркости представляет собой одномерный целочисленный массив Hist из 256 элементов $[0 \dots 255]$. Элементом гистограммы $\mathrm{Hist}[i]$ является сумма пикселей изображения с яркостью i. По визуальному отображению гистограммы можно оценить необходимость изменения яркости и контрастности изображения, оценить площадь, занимаемую светлыми и темными элементами, определить местоположение на плоскости изображения отдельных объектов, соответствующих некоторым диапазонам яркости. Для цветного RGB-изображения необходимо построить три гистограммы по каждому цвету.

Листинг 1.1. Построение гистограмм изображения в MATLAB.

```
1 [numRows numCols Layers] = size(I);
2 imhist(I(:,:,1)); %red
3 imhist(I(:,:,2)); %green
4 imhist(I(:,:,3)); %blue
```

Listing 1.2. Построение гистограмм изображения с использованием OpenCV и языка программирования C++

```
1   // I is an RGB-image
2   // Number of histogram bins
3   int histSize = 256;
4   // Histogram range
5   // The upper boundary is exclusive
6   float range[] = { 0, 256 };
7   const float* histRange[] = { range };
8   // Split an image into color layers
9   // OpenCV stores RGB image as BGR
10  vector<Mat> I_BGR;
```

```
split(I, I_BGR);
// Calculate a histogram for each layer
Mat bHist, gHist, rHist;
calcHist(&I_BGR[0], 1, 0, Mat(), bHist, 1,
         &histSize, histRange);
calcHist(&I_BGR[1], 1, 0, Mat(), gHist, 1,
         &histSize, histRange);
calcHist(&I_BGR[2], 1, 0, Mat(), rHist, 1,
         &histSize, histRange);
```

Listing 1.3. Построение гистограмм изображения с использованием OpenCV и языка программирования Python

```
# I is an RGB-image
# Number of histogram bins
histSize = 256
# Histogram range
# The upper boundary is exclusive
histRange = (0, 256)
# Split an image into color layers
# OpenCV stores RGB image as BGR
I_BGR = cv2.split(I)
# Calculate a histogram for each layer
bHist = cv2.calcHist(I_BGR, [0], None,
                     [histSize], histRange)
gHist = cv2.calcHist(I_BGR, [1], None,
                     [histSize], histRange)
rHist = cv2.calcHist(I_BGR, [2], None,
                     [histSize], histRange)
```

Если гистограмма неравномерна, то для улучшения изображения можно ее выровнять, причем выравнивание гистограммы в зависимости от решаемой задачи можно выполнять различным образом.

Арифметические операции

Простейшими способами выравнивания гистограммы являются *арифметические операции* с изображениями. Например, в случае, если большинство значений гистограммы находятся слева, то изображение является темным. Для увеличения детализации тем-

ных областей можно сдвинуть гистограмму правее, в более светлую область, например, на 50 градаций для каждого цвета. В среде MATLAB программная реализация имеет вид:

```
Inew(i,j) = I(i,j) + 50 / 255;
```

Гистограмма исходного изображения сдвигается в среднюю часть диапазона, которая является более приемлемой с точки зрения визуального восприятия. Данный подход обладает следующим недостатком: повышение интенсивностей темных областей приводит к сдвигу светлых к максимуму, что может привести к потере информации в светлых областях.

Растяжение динамического диапазона

Если интенсивности пикселей областей интереса находятся в узком динамическом диапазоне, то можно растянуть этот диапазон. Подобные преобразования выполняются согласно следующему выражению:

$$I_{new} = \left(\frac{I - I_{min}}{I_{max} - I_{min}} \right)^{\alpha}, \tag{1.1}$$

где I и I_{new} — массивы значений интенсивностей исходного и нового изображений соответственно; I_{min} и I_{max} — минимальное и максимальное значения интенсивностей исходного изображения соответственно; α — коэффициент нелинейности.

Данное выражение является нелинейным из-за коэффициента α. В случае, если $\alpha = 1$, применение формулы (1.1) к исходному изображению не даст желаемого эффекта, поскольку гистограммы цветовых компонент изображения занимают весь возможный диапазон. Нелинейные преобразования проводятся для каждой цветовой составляющей.

Листинг 1.4. Нелинейное растяжение динамического диапазона при $\alpha = 0{,}5$ в MATLAB.

```
1 [numRows numCols Layers] = size(I);
2 for k=1:1:Layers;
3     Imin = min(min(I(:, :, k)));
4     Imax = max(max(I(:, :, k)));
5     for i=1:1:numRows;
```

```
6           for j=1:1:numCols;
7               Inew(i,j,k) = ...
8                   (((I(i,j,k) - Imin) / ...
9                       (Imax - Imin)))^0.5;
10              if Inew(i,j,k) > 1;
11                  Inew(i,j,k) = 1;
12              end
13              if Inew(i,j,k) < 0;
14                  Inew(i,j,k) = 0;
15              end
16          end
17      end
18  end
```

Библиотека OpenCV не производит автоматическое преобразование типов при выполнении деления, поэтому при делении двух целых чисел, находящихся в диапазоне от 0 до 255, будет получено также целое число 0 или 1. Поэтому при обработке изображений с использованием библиотеки OpenCV следует предварительно конвертировать изображение в вещественные числа с помощью функции `convertTo`. После завершения обработки изображения его следует сконвертировать к изначальной глубине цвета.

Listing 1.5. Нелинейное растяжение динамического диапазона при $\alpha = 0{,}5$ с использованием OpenCV и языка программирования C++.

```
1   alfa = 0.5;
2   Mat Inew;
3   // Convert to floating points
4   if (I.depth() == CV_8U)
5     I.convertTo(Inew, CV_32F, 1.0 / 255);
6   else
7     Inew = I;
8   // We need to process layers separately
9   vector<Mat> I_BGR;
10  cv::split(Inew, I_BGR);
11  for (int k = 0; k < I_BGR.size(); k++)
12  {
13    // Create a new image with same size and
```

```
14      // one floating point layer
15      Inew = Mat(I_BGR[k].rows, I_BGR[k].cols,
16        I_BGR[k].type());
17      // Calculate min and max values
18      double Imin, Imax;
19      minMaxLoc(I_BGR[k], &Imin, &Imax);
20      // Change intensity of each image pixel
21      for (int i = 0; i < I_BGR[k].rows; i++)
22      {
23        for (int j = 0; j < I_BGR[k].cols; j++)
24          Inew.at<float>(i, j) =
25            float(pow((I_BGR[k].at<uchar>(i, j)
26            - Imin) / (Imax - Imin), alfa));
27      }
28      I_BGR[k] = Inew;
29    }
30    // Merge back
31    cv::merge(I_BGR, Inew);
32    // Convert back to uint if needed
33    if (I.depth() == CV_8U)
34      Inew.convertTo(Inew, CV_8U, 255);
```

Listing 1.5. Нелинейное растяжение динамического диапазона при $\alpha = 0{,}5$ с использованием OpenCV и языка программирования Python.

```
1   alfa = 0.5
2   # Convert to floating point
3   if I.dtype == np.uint8:
4     Inew = I.astype(np.float32) / 255
5   else:
6     Inew = I
7   # We need to process layers separately
8   I_BGR = cv2.split(Inew)
9   Inew_BGR = []
10  for layer in I_BGR:
11    Imin = layer.min()
12    Imax = layer.max()
13    Inew = np.clip((((layer - Imin) /
```

```
14        (Imax - Imin)) ** alfa), 0, 1)
15      Inew_BGR.append(Inew)
16    # Merge back
17    Inew = cv2.merge(Inew_BGR)
18    # Convert back to uint if needed
19    if (I.dtype == np.uint8):
20      Inew = (255 * Inew).clip(0, 255). \
21        astype(np.uint8)
```

Равномерное преобразование

Осуществляется по следующей формуле:

$$I_{new} = (I_{max} - I_{min}) \cdot P(I) + I_{min}, \tag{1.2}$$

где I_{min}, I_{max} — минимальное и максимальное значения интенсивностей исходного изображения I; $P(I)$ — функция распределения вероятностей исходного изображения, которая аппроксимируется *кумулятивной гистограммой*:

$$P(I) \approx \sum_{m=0}^{i} \text{Hist}(m). \tag{1.3}$$

Кумулятивная гистограмма исходного изображения в среде MATLAB может быть вычислена с помощью функции `cumsum()`:

```
CH = cumsum(H) ./ (numRows * numCols);
```

где `H` — гистограмма исходного изображения, `numRows` и `numCols` — число строк и столбцов исходного изображения соответственно.

Согласно формуле (1.2) можно вычислить значения интенсивностей пикселей результирующего изображения. В среде MATLAB программная реализация имеет вид:

```
Inew(i,j) = (Imax - Imin) * CH(ceil(I(i,j) + eps)) + ...
  Imin / 255.0;
```

Параметр `eps` необходим для защиты программы от присваивания индексам кумулятивной гистограммы нулевых значений.

При использовании библиотеки OpenCV и языка программирования C++ вычисление кумулятивной гистограммы выполняется следующим способом:

```
Mat CH = H.clone();
for (int i = 1; i < CH.size[0]; i++)
  CH.at<float>(i) += CH.at<float>(i - 1);
CH /= numRows * numCols;
```

Далее, согласно формуле (1.2) можно вычислить значения интенсивностей пикселей результирующего изображения обращением к соответствующим элементам массива:

```
Inew<float>(i,j) = (Imax - Imin) *
  CH.at<float>(I.at<uchar>(i, j)) + Imin;
```

Поскольку в данном примере исходное изображение задано целыми числами в диапазоне от 0 до 255, то дополнительная коррекция на `eps` не требуется. Однако, если исходное изображение определено значениями с плавающей запятой, коррекцию следует учитывать.

При использовании языка программирования Python можно использовать встроенную функцию `cumsum()` для вычисления кумулятивной гистограммы:

```
CH = np.cumsum(H) / (numRows * numCols)
```

Затем, для вычисления значения интенсивности пикселей изображения, можно обратиться к соответствующему элементу этого массива для одного пикселя:

```
Inew[i, j] = (Imax - Imin) * CH[I[i, j]] + Imin
```

Или для слоя изображения:

```
Inew[:, :, k] = (Imax - Imin) * CH[I[:, :, k]] + Imin
```

Экспоненциальное преобразование

Осуществляется по следующей формуле:

$$I_{new} = I_{min} - \frac{1}{\alpha} \cdot \ln(1 - P(I)), \tag{1.4}$$

где α — постоянная, характеризующая крутизну преобразования. Согласно формуле (1.4) можно вычислить значения интенсивностей пикселей результирующего изображения. В среде MATLAB программная реализация имеет вид:

```
Inew(i,j) = double(Imin) - ...
  (1 / alfa) * log(1 - CH(index));
```

При использовании библиотеки OpenCV и языка программирования C++ программная реализация примет вид:

```
Inew<float>(i,j) = Imin - 1 / alfa *
  log(1 - CH.at<float>(I.at<uchar>(i, j)));
```

При использовании библиотеки OpenCV и языка программирования Python программная реализация примет вид:

```
Inew[i, j] = Imin - 1 / alfa * math.log(1 - CH[I[i, j]])
```

Преобразование по закону Рэлея

Осуществляется по следующей формуле:

$$I_{new} = I_{min} + \left(2\alpha^2 \ln\left(\frac{1}{1 - P(I)}\right)\right)^{1/2}, \qquad (1.5)$$

где α — постоянная, характеризующая гистограмму распределения интенсивностей элементов результирующего изображения. В среде MATLAB программная реализация имеет вид:

```
Inew(i,j) = double(Imin) + sqrt(2 * alfa^2 * ...
 log(1 / (1 - CH(ceil(I(i,j) + eps)))));
```

При использовании библиотеки OpenCV и языка программирования C++ программная реализация примет вид:

```
Inew<float>(i,j) = Imin + sqrt(2 * alfa * alfa *
  log(1 / (1 - CH.at<float>(I.at<uchar>(i, j)))));
```

При использовании библиотеки OpenCV и языка программирования Python программная реализация примет вид:

```
Inew[i, j] = Imin + sqrt(2 * alfa ** 2 *
  math.log(1 / 1 - CH[I[i, j]]))
```

Преобразование по закону степени $2/3$

Осуществляется по следующей формуле:

$$I_{new} = P(I)^{2/3}. \tag{1.6}$$

В среде MATLAB программная реализация имеет вид:

```
Inew(i,j) = (CH(ceil(I(i,j) + eps)))^(2/3);
```

При использовании библиотеки OpenCV и языка программирования C++ программная реализация примет вид:

```
Inew<float>(i,j) =
  pow(CH.at<float>(I.at<uchar>(i, j)), 2/3);
```

При использовании библиотеки OpenCV и языка программирования Python программная реализация примет вид:

```
Inew[i, j] = CH[I[i, j]] ** (2 / 3)
```

Гиперболическое преобразование

Осуществляется по следующей формуле:

$$I_{new} = \alpha^{P(I)}, \tag{1.7}$$

где α — постоянная, относительно которой осуществляется преобразование и, как правило, равная минимальному значению интенсивности элементов исходного изображения $\alpha = I_{min}$.

В среде MATLAB программная реализация имеет вид:

```
alfa = 0.04;
Inew(i,j) = alfa^CH(ceil(I(i,j) + eps));
```

При использовании библиотеки OpenCV и языка программирования C++ программная реализация примет вид:

```
alfa = 0.04;
Inew<float>(i,j) =
  pow(alfa, CH.at<float>(I.at<uchar>(i, j)));
```

При использовании библиотеки OpenCV и языка программирования Python программная реализация примет вид:

```
alfa = 0.04
Inew[i, j] = alfa ** CH[I[i, j]]
```

Рассмотренные методы преобразования гистограмм могут применяться для устранения искажений при передаче уровней квантования, которым были подвергнуты изображения на этапе формирования, передачи или обработки данных. Кроме того, данные методы могут применяться не только ко всему изображению, но использоваться локально в *скользящем окне*, что позволит повысить детализированность отдельных областей.

В среде MATLAB реализовано несколько функций, автоматически выравнивающих гистограммы полутонового изображения:

1. `imadjust()` — повышает контрастность изображения, изменяя диапазон интенсивностей исходного изображения;
2. `histeq()` — эквализирует (выравнивает) гистограмму методом распределения значений интенсивностей элементов исходного изображения;
3. `adapthisteq()` — выполняет контрастно-ограниченное адаптивное выравнивание гистограммы методом анализа и эквализации гистограмм локальных окрестностей изображения.

Для обработки цветного изображения данные функции можно применять поочередно к каждому цвету.

В библиотеке OpenCV также реализовано несколько функций для автоматического выравнивания гистограммы полутонового изображения :

1. `equalizeHist()` — эквализирует (выравнивает) гистограмму методом распределения значений интенсивностей элементов исходного изображения;
2. `createCLAHE()` и `CLAHE.apply()` — выполняет контрастно-ограниченное адаптивное выравнивание гистограммы методом анализа и эквализации гистограмм локальных окрестностей изображения.

Аналогично с MATLAB, данные функции можно применять только к одному слою изображения. Поэтому для обработки цветного изображения данные функции можно применять поочередно к каждому цвету.

Таблица поиска

Таблица поиска или *LUT (Lookup table)* — удобный инструмент для преобразования интенсивностей точек всего изображения. Поскольку большинство изображений определяется с использованием ограниченного набора возможных дискретных значений цвета (чаще всего цвет задается с использованием целых чисел, находящихся в диапазоне [0, 255]), то можно заранее вычислить преобразование цвета для каждого потенциально-возможного значения цвета, сохранить их в таблице поиска, а затем преобразовать цвет исходного изображения, используя формулу:

$$I_{new} = LUT[I_{old}] \tag{1.8}$$

В библиотеке OpenCV есть встроенная функция для применения преобразования, заданного с использованием таблицы поиска, к изображению. Таблица поиска может быть определена либо одинаковой для всех слоев изображения, либо для каждого слоя отдельно, в зависимости от количества слоев таблицы. В языке программирования C++ программная реализация растяжения динамического диапазона с использованием таблиц поиска имеет вид:

```
alfa = 0.5;
Mat lut(1, 256, CV_8U);
uchar *lut_ptr = lut.ptr();
for (int i = 0; i < 256; i++)
{
  double var = (i - Imin) / (Imax - Imin);
  if (var < 0)
    lut_ptr[i] = 0;
  else
    lut_ptr[i] = saturate_cast<uchar>(
      255 * pow(var, alfa));
}
LUT(I, lut, Inew);
```

Аналогичным образом реализуется использование таблиц поиска в языке программирования Python:

```
alfa = 0.5
```

```
lut = np.arange(256, dtype = np.uint8)
lut = (lut - Imin) / (Imax - Imin)
lut = np.where(lut > 0, lut, 0)
lut = np.clip(255 * np.power(lut, alfa), 0, 255)
Inew = cv2.LUT(I, lut)
```

Профиль изображения

Профилем изображения вдоль некоторой линии называется функция интенсивности изображения, распределенного вдоль данной линии *(прорезки)*. Простейшим случаем профиля изображения является профиль строки:

$$\text{Profile}\, i(x) = I(x,i), \tag{1.9}$$

где i — номер строки изображения I.

Профиль столбца изображения:

$$\text{Profile}\, j(y) = I(j,y), \tag{1.10}$$

где j — номер столбца изображения I.

В общем случае профиль можно рассматривать вдоль любой прямой, ломаной или кривой линии, пересекающей изображение. После формирования массива профиля изображения проводится его анализ стандартными средствами. Анализ позволяет автоматически выделять особые точки функции профиля, соответствующие контурам изображения, пересекаемым данной линией. Например, на рис. 1.1 представлен профиль изображения штрих-кода, взятого вдоль оси Ox. Данный профиль содержит всю необходимую информацию для считывания штрих-кода, поскольку позволяет определить последовательность чередования «толстых» и «тонких» штрихов и пробелов различной ширины. В среде MATLAB профиль изображения можно найти при помощи фунции `improfile()`.

Листинг 1.6. Поиск профиля штрих-кода вдоль оси Ox в среде MATLAB.

```
1 I = imread('code.jpg');
2 [numRows, numCols, Layers] = size(I);
3 x = [1 numCols];
4 y = [ceil(numRows/2) ceil(numRows/2)];
```

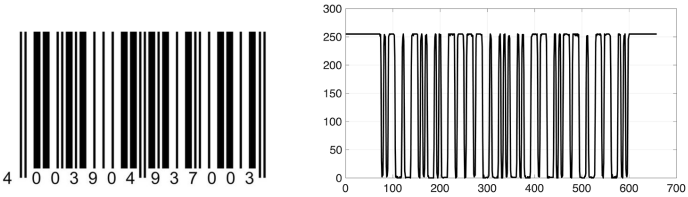


Рис. 1.1 — Слева — штрих-код,
справа — его профиль вдоль оси Ox.

```
5 figure
6 improfile(I,x,y),grid on;
```

Для интерактивного указания линии (ломаной), вдоль которой следует построить профиль, необходимо использовать функцию `improfile` без параметров.

Листинг 1.7. Интерактивное задание линии профиля в среде MATLAB.

```
1 I = imread('code.jpg');
2 imshow(I);
3 improfile
```

По сравнению с MATLAB в библиотеке OpenCV нет такого удобного инструмента для вычисления профиля изображения вдоль произвольной оси. Однако можно получить горизонтальный или вертикальный профиль с использованием функций библиотеки для быстрого доступа к отделенной строке или столбцу изображения.

Listing 1.8. Поиск профиля штрих-кода вдоль оси Ox с использованием библиотеки OpenCV и языка программирования C++.

```
1   Mat I = imread("code.jpg", IMREAD_COLOR);
2   Mat profile = I.row(I.rows / 2);
```

Listing 1.9. Поиск профиля штрих-кода вдоль оси Ox с использованием библиотеки OpenCV и языка программирования Python.

```
1   I = cv2.imread("code.jpg", cv2.IMREAD_COLOR)
2   profile = I[round(I.shape[0] / 2), :]
```

Проекция изображения

Проекцией изображения на некоторую ось называется сумма интенсивностей пикселей изображения в направлении, перпендикулярном данной оси. Простейшим случаем проекции двумерного изображения являются вертикальная проекция на ось Ox, представляющая собой сумму интенсивностей пикселей *по столбцам* изображения:

$$\mathrm{Proj}\, X(y) = \sum_{y=0}^{\dim Y - 1} I(x,y), \tag{1.11}$$

и горизонтальная проекция на ось Oy, представляющая собой сумму интенсивностей пикселей *по строкам* изображения:

$$\mathrm{Proj}\, Y(x) = \sum_{x=0}^{\dim X - 1} I(x,y). \tag{1.12}$$

Запишем выражение для проекции на произвольную ось. Допустим, что направление оси задано единичным вектором с координатами (e_x,e_y). Тогда проекция изображения на ось Oe определяется следующим выражением:

$$\mathrm{Proj}\, E(t) = \sum_{xe_x+ye_y=t} I(x,y). \tag{1.13}$$

Анализ массива проекции позволяет выделять характерные точки функции проекции, которые соответствуют контурам объектов на изображении. Например, если на изображении имеются контрастные объекты, то в проекции будут видны перепады или экстремумы функции, соответствующие положению каждого из объектов.

На рис. 1.2 показан пример проекции на ось Oy машиночитаемого документа. Видно, что две машиночитаемые текстовые строки порождают два существенных экстремума функции проекции. Подобные проекции могут быть использованы в алгоритмах обнаружения и сегментации текстовых строк в системах распознавания текста.

Листинг 1.10. Определение положения текста в среде MATLAB.

```
for i=1:1:numRows
    Proj(i,1) = (round(sum(I(i,:,1))) + ...
    round(sum(I(i,:,2))) + ...
    round(sum(I(i,:,3)))) / (256 * 3);
end
```

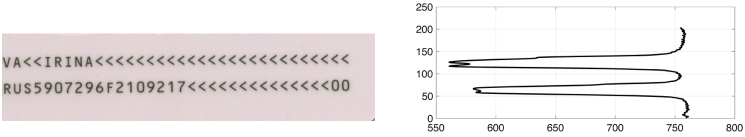


Рис. 1.2 — Слева — фрагмент заграничного паспорта, справа — его проекция на ось Oy.

Listing 1.11. Определение положения текста с использованием библиотеки OpenCV и языка программирования C++.

```
// An array to store projection
Mat Proj(I.rows, 1, CV_32F);
bool bw = (I.channels() == 1);
for (int i = 0; i < I.rows; i++)
{
  double sum = 0;
  for (int j = 0; j < I.cols; j++)
  {
    if (bw)
      sum += I.at<uchar>(i, j);
    else
    {
      const Vec3b &pix = I.at<Vec3b>(i, j);
      sum += pix[0] + pix[1] + pix[2];
    }
  }
  Proj.at<float>(i) = float(sum);
}
Proj /= 256 * I.channels();
// Calculate min and max values
```

```
21  double ProjMin, ProjMax;
22  minMaxLoc(Proj, &ProjMin, &ProjMax);
23  // Create graph image
24  Mat ProjI(256, img.rows, CV_8UC3,
25    Scalar(255, 255, 255));
26  DrawGraph<float>(ProjI, Proj,
27    Scalar(0, 0, 0), ProjMax);
28  transpose(ProjI, ProjI);
29  flip(ProjI, ProjI, 1);
30  // And display it
31  imshow("Projection␣to␣Oy", ProjI);
32  waitKey();
```

Listing 1.12. Определение положения текста с использованием библиотеки OpenCV и языка программирования Python.

```
1   # Calculate projection to Oy
2   if I.ndim == 2:
3     ProjI = np.sum(I, 1) / 255
4   else:
5     ProjI = np.sum(I, (1, 2)) / 255 / \
6       I.shape[2]
7   # Create graph image
8   ProjI = np.full((256, Proj.shape[0], 3),
9                   255, dtype = np.uint8)
10  DrawGraph(ProjI, Proj, (0, 0, 0),
11            Proj.max())
12  ProjI = cv2.transpose(ProjI)
13  ProjI = cv2.flip(ProjI, 1)
14  # And display it
15  cv2.imshow('Projection␣to␣Oy', ProjI)
16  cv2.waitKey()
```

Порядок выполнения работы

1. *Гистограммы.* Выбрать произвольное слабоконтрастное изображение. Выполнить выравнивание гистограммы и растяжение контраста, использовать рассмотренные преобразования

и встроенные функции пакета MATLAB. Сравнить полученные результаты.

2. *Профили.* Выбрать произвольное изображение, содержащие штрих-код. Выполнить построение профиля изображения вдоль штрих-кода.

3. *Проекции.* Выбрать произвольное изображение, содержащее монотонные области и выделяющиеся объекты. Произвести построение проекций изображения на вертикальную и горизонтальную оси. Определить границы областей объектов.

Содержание отчета

1. Титульный лист.

2. Цель работы.

3. Теоретическое обоснование применяемых методов и функций геометрических преобразований.

4. Ход выполнения работы:

   (a) Исходные изображения;

   (b) Листинги программных реализаций;

   (c) Комментарии;

   (d) Результирующие изображения.

5. Выводы о проделанной работе.

Вопросы к защите лабораторной работы

1. Что такое контрастность изображения и как её можно изменить?

2. Чем эффективно использование профилей и проекций изображения?

3. Каким образом можно найти объект на равномерном фоне?

Лабораторная работа №2
Геометрические преобразования изображений

Цель работы

Освоение основных видов отображений и использование геометрических преобразований для решения задач пространственной коррекции изображений.

Методические рекомендации

До начала работы студенты должны ознакомиться с основными функциями среды MATLAB или библиотеки OpenCV по работе с геометрическими преобразованиями изображений. Лабораторная работа рассчитана на 5 часов.

Теоретические сведения

Геометрические преобразования изображений подразумевают пространственное изменение местоположения множества пикселей с целочисленными координатами (x,y) в другое множество с координатами (x',y'), причем интенсивность пикселей сохраняется. В двумерных плоских геометрических преобразованиях, как правило, используется евклидово пространство $\mathbf{P}^2$ с ортонормированной декартовой системой координат. В этом случае пикселю изображения соответствует пара декартовых координат, которые интерпретируются в виде двумерного вектора, представленного отрезком из точки (0,0) до точки $X_i = (x_i,y_i)$. Двумерные преобразования на плоскости можно представить в виде движения точек, соответствующих множеству пикселей.

Для общности с дальнейшими преобразованиями будем использовать *однородные координаты*, обладающие тем свойством, что определяемый ими объект не меняется при умножении всех координат на одно и то же ненулевое число. Из-за этого свойства необходимое количество координат для представления точек всегда на одну

больше, чем размерность пространства $\mathbf{P}^n$, в котором эти координаты используются. Например, для представления точки $X = (x,y)$ на плоскости в двумерном пространстве $\mathbf{P}^2$ необходимо три координаты $X = (x,y,w)$. Проиллюстрируем это следующим примером:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = w \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Leftrightarrow \begin{bmatrix} x' & y' & w \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} w, \tag{2.1}$$

где w — скалярный произвольный множитель, $x = \dfrac{x'}{w}$, $y = \dfrac{y'}{w}$.

При помощи троек однородных координат и матриц третьего порядка можно описать любое линейное преобразование плоскости. Таким образом, геометрические преобразования являются матричными преобразованиями, и множества координат пикселей преобразованного и исходного изображений связаны следующим матричным соотношением либо в строчном виде $X' = XT$, либо в столбцовом $X' = T^{\mathrm{T}}X$. Распишем эти матричные соотношения:

$$\begin{bmatrix} x' & y' & w' \end{bmatrix} = \begin{bmatrix} x & y & w \end{bmatrix} \cdot \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} \tag{2.2}$$

$$\Leftrightarrow \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}. \tag{2.3}$$

Точка с декартовым координатами (x,y) в однородных координатах запишется как $(x,y,1)$:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} A & D & 0 \\ B & E & 0 \\ C & F & 1 \end{bmatrix} \tag{2.4}$$

$$\Leftrightarrow \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{2.5}$$

Формулу (2.4) можно представить в виде следующей системы уравнений:

$$\begin{cases} x' = Ax + By + C, \\ y' = Dx + Ey + F. \end{cases} \tag{2.6}$$

Линейные преобразования

Эвклидово отображение — это отображение, при котором сохраняется форма бесконечно малых фигур и углы между кривыми в точках их пересечения. К таким преобразованиям относятся сдвиг, отражение, однородное масштабирование и поворот. Эвклидовы преобразования являются подмножеством аффинных преобразований.

Сдвиг изображения

Система уравнений (2.6) и матрица преобразования координат T в случае *сдвига* изображения $A = E = 1$, $B = D = 0$ примут вид:

$$\begin{cases} x' = x + C, \\ y' = y + D, \end{cases} \tag{2.7}$$

$$\Leftrightarrow \begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} T \Rightarrow T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ C & F & 1 \end{bmatrix}, \tag{2.8}$$

где C и F — сдвиг по осям Ox и Oy соответственно.

Листинг 2.1. Сдвиг изображения в среде MATLAB на 50 и 100 пикселей по осям Ox и Oy соответственно. Размер исходного рисунка `roadSign.jpg` 300×300 пикселей.

```
1  I = imread('roadSign.jpg');
2  T = [1 0 0; 0 1 0; 50 100 1];
3  tform = affine2d(T);
4  I_shift = imwarp(I, tform, ...
5    'Interp', 'nearest', ...
6    'OutputView', imref2d(size(I), ...
7    [1 size(I,2)], [1 size(I,1)]) ...
8  );
```

Функция `affine2d()` создает матрицу преобразования, которая применяется к изображению `I` при помощи функции `imwarp()`. Дополнительные параметры задают одинаковые координаты для исходного и преобразованного изображений, а также метод интерполяции для неопределенных пикселей.

В отличие от MATLAB, в библиотеке OpenCV матрицы хранятся в виде строк, поэтому следует использовать формулу (2.5) для задания матрицы преобразования:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow T = \begin{bmatrix} 1 & 0 & C \\ 0 & 1 & F \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.9}$$

Кроме того, последний ряд матрицы не хранится, поэтому матрица будет задаваться следующим образом:

$$T = \begin{bmatrix} 1 & 0 & C \\ 0 & 1 & F \end{bmatrix}, \tag{2.10}$$

Листинг 2.2.Сдвиг изображения с использованием библиотеки OpenCV и языка программирования C++ на 50 и 100 пикселей по осям Ox и Oy соответственно. Размер исходного рисунка `roadSign.jpg` 300×300 пикселей.

```
1  Mat I =
2    imread("roadSign.jpg", IMREAD_COLOR);
3  Mat T = (Mat_<double>(2, 3) <<
4           1, 0, 50 ,
5           0, 1, 100);
6  Mat I_shift;
7  warpAffine(I, I_shift, T,
8    Size(I.cols, I.rows));
```

В случае использования библиотеки OpenCV и языка программирования C++ все матрицы (в том числе и матрица изображения) хранятся с использованием объекта `cv::Mat`, представляющего собой матрицу произвольного размера. Матрицу преобразования можно создать вручную с помощью операции `Mat_<double>(2, 3) <<`, где аргументы `(2, 3)` задают размеры матрицы (количество строк и количество столбцов матрицы соответственно), а

аргументы, передаваемые после оператора <<, определяют элементы матрицы построчно, начиная с первой строки. Функция `cv::warpAffine()` используется для выполнения любого вида аффинного преобразования изображения. Первый аргумент — это изображение для преобразования, второй — место для сохранения результирующего изображения, третий — матрица преобразования, а четвертый — разрешение, которое будет использоваться при формировании результирующего изображения.

Листинг 2.3.Сдвиг изображения с использованием библиотеки OpenCV и языка программирования Python на 50 и 100 пикселей по осям Ox и Oy соответственно. Размер исходного рисунка `roadSign.jpg` 300×300 пикселей.

```
1   rows, cols = I.shape[0:2]
2   T = np.float32([[1, 0, 50], [0, 1, 100]])
3   I_shift = cv2.warpAffine(I, T, (cols, rows))
```

В случае использования языка программирования Python матрица преобразования представляет собой простой массив библиотеки NumPy размерами 3×2, поэтому его можно создать с помощью встроенной функции библиотеки NumPy `numpy.float32()`. Как и в C++, в Python функция `cv2.warpAffine()` используется для выполнения любого вида аффинного преобразования, однако, в отличие от реализации библиотеки OpenCV для C++, результирующее изображение не является обязательным аргументом, а может быть и возвращаемым значением функции. Все остальные аргументы те же. В реализации библиотеки OpenCV для языка программирования Python массивы NumPy используются повсеместно для хранения изображений, поэтому все методы, применимые к массивам NumPy, применимы и для изображений. Следует обратить внимание, что в параметре формы изображения `I.shape` разрешение изображения хранится в виде количества строк и столбцов, а в функцию преобразования оно должно передаваться в виде количества столбцов и строк. Это связано с тем, что матрицы традиционно задаются указанием количества строк и столбцов, а изображения — указанием ширины и высоты.

Отражение изображения

Система уравнений (2.6) и матрица преобразования координат T в случае *отражения* изображения вдоль оси Ox при $A = 1$, $E = -1$, $B = C = D = F = 0$ примут вид:

$$\begin{cases} x' = x \\ y' = -y \end{cases} \Rightarrow T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.11}$$

Листинг 2.4. Отражение относительно оси Ox в среде MATLAB.

```
1   T = [1 0 0; 0 -1 0; 0 0 1];
2   tform = affine2d(T);
3   I_reflect = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для отражения изображения определяется следующим образом:

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & I.rows - 1 \end{bmatrix}, \tag{2.12}$$

Дополнительный сдвиг изображения необходим для устранения отрицательных координат пикселей изображения, которые будут получены после преобразования.

Листинг 2.5. Отражение относительно оси Ox с использованием библиотеки OpenCV и языка программирования C++.

```
1   Mat T = (Mat_<double>(2, 3) <<
2            1, 0, 0,
3            0, -1, I.rows - 1);
4   Mat I_reflect;
5   warpAffine(I, I_reflect, T,
6     Size(I.cols, I.rows));
```

Листинг 2.6. Отражение относительно оси Ox с использованием библиотеки OpenCV и языка программирования Python.

```
1   T = np.float32([[1, 0, 0],
2                   [0, -1, rows - 1]])
3   I_reflect = \
4     cv.warpAffine(I, T, (cols, rows))
```

В библиотеке OpenCV также присутствует функция `flip()`, предназначенная для выполнения отражения изображения. Данная функция требует передачи одного параметра в дополнение к изображению. Значение параметра 0 означает отражение относительно оси Ox, 1 — отражение относительно оси Oy, а -1 — отражение относительно обеих осей. Например, отражение изображения относительно оси Ox с использованием языка программирования C++ будет выглядеть следующим образом:

```
cv::flip(I, I_reflect, 0);
```

А с использованием языка программирования Python:

```
I_reflect = cv2.flip(I, 0)
```

Однородное масштабирование изображения

Система уравнений (2.6) и матрица преобразования координат T в случае *масштабирования* изображения $A = \alpha$, $E = \beta$, $B = C = D = F = 0$ примут вид:

$$\begin{cases} x' = \alpha x, \alpha > 0 \\ y' = \beta y, \beta > 0 \end{cases} \Rightarrow T = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.13}$$

если $\alpha < 1$ и $\beta < 1$, то изображение уменьшается, если $\alpha > 1$ и $\beta > 1$ — увеличивается. Если $\alpha \neq \beta$, то пропорции будут не одинаковыми по ширине и высоте. В общем случае данное отображение будет являться аффинным, а не эвклидовыми.

Листинг 2.7. Увеличение исходного изображения в два раза в среде MATLAB.

```
1   T = [2 0 0; 0 2 0; 0 0 1];
2   tform = affine2d(T);
3   I_scale = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для масштабирования изображения определяется следующим образом:

$$T = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \end{bmatrix}, \tag{2.14}$$

Листинг 2.8. Увеличение исходного изображения в два раза с использованием библиотеки OpenCV и языка программирования C++.

```
1   Mat T = (Mat_<double>(2, 3) <<
2            2, 0, 0,
3            0, 2, 0);
4   Mat I_scale;
5   warpAffine(I, I_scale, T,
6     Size(int(I.cols * 2), int(I.rows * 2)));
```

Листинг 2.9. Увеличение исходного изображения в два раза с использованием библиотеки OpenCV и языка программирования Python.

```
1   T = np.float32([[scale_x, 0, 0],
2                   [0, scale_y, 0]])
3   I_scale = cv2.warpAffine(I, T,
4     (int(cols * scale_x),
5     int(rows * scale_y)))
```

Чтобы учесть изменение разрешения изображения при масштабировании, размер результирующего изображения был вычислен с учетом коэффициента масштабирования.

В библиотеке OpenCV также присутствует функция `resize()`, предназначенная для изменения размера изображения. Например, увеличение размера изображения в два раза с использованием языка программирования C++ будет выглядеть следующим образом:

```
cv::resize(I, I_scale,
  Size(int(I.cols * 2), int(I.rows * 2)));
```

А с использованием языка программирования Python:

```
I_scale = cv2.resize(I, None, fx = 2, fy = 2,
  interpolation = cv2.INTER_CUBIC)
```

Поворот изображения

Система уравнений (2.6) и матрица преобразования координат T в случае *поворота* изображения по часовой стрелке $A = \cos\varphi$, $B = -\sin\varphi$, $D = \sin\varphi$, $E = \cos\varphi$, $C = F = 0$ примут вид:

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi \\ y' = x\sin\varphi + y\cos\varphi \end{cases} \Rightarrow T = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (2.15)$$

Если $\varphi = 90°$, то $\cos\varphi = 0$ и $\sin\varphi = 1$ и выражение (2.15) примет вид:

$$\begin{cases} x' = -y \\ y' = x \end{cases} \Rightarrow T = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (2.16)$$

Листинг 2.10. Поворот изображения на $\varphi = 17°$ в среде MATLAB.

```
phi = 17*pi/180;
T = [cos(phi) sin(phi) 0;
  -sin(phi) cos(phi) 0;
  0 0 1];
tform = affine2d(T);
I_rotate = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для поворота изображения определяется следующим образом:

$$T = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \end{bmatrix}, \quad (2.17)$$

Листинг 2.11. Поворот изображения на $\varphi = 17°$ с использованием библиотеки OpenCV и языка программирования C++.

```
double phi = 17.0 * M_PI / 180;
Mat T = (Mat_<double>(2, 3) <<
          cos(phi), -sin(phi), 0,
          sin(phi), cos(phi), 0);
Mat I_rotate;
warpAffine(I, I_rotate, T,
  Size(I.cols, I.rows));
```

Листинг 2.12. Поворот изображения на $\varphi = 17°$ с использованием библиотеки OpenCV и языка программирования Python.

```
1   phi = 17.0 * math.pi / 180
2   T = np.float32(
3     [[math.cos(phi), -math.sin(phi), 0],
4     [math.sin(phi), math.cos(phi), 0]])
5   I_rotate = \
6     cv2.warpAffine(I, T, (cols, rows))
```

Так как при использовании матрицы преобразования (2.17) поворот осуществляется вокруг верхнего левого угла изображения с координатами $(0, 0)$, то поворот вокруг произвольной точки следует определять как сдвиг изображения так, чтобы желаемая точка вращения находилась в верхнем левом углу, затем поворот на заданный угол и обратный сдвиг. Это преобразование можно вычислить в матричной форме, а затем выполнить как одно аффинное преобразование. Например для поворота изображения вокруг его центра матрица поворота будет вычисляться следующим образом:

$$T1 = \begin{bmatrix} 1 & 0 & -(I.cols - 1)/2 \\ 0 & 1 & -(I.rows - 1)/2 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.18}$$

$$T2 = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.19}$$

$$T3 = \begin{bmatrix} 1 & 0 & (I.cols - 1)/2 \\ 0 & 1 & (I.rows - 1)/2 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.20}$$

где матрица `T1` — это прямой сдвиг для совмещения точки $(0, 0)$ с центром изображения, `T2` — матрица преобразования вращения на угол φ, а `T3` — обратный сдвиг. Полная матрица преобразования `T` вычисляется путем матричного умножения этих трех матриц преобразования.

$$T = T3 \cdot T2 \cdot T1 \tag{2.21}$$

Учитывая то, что в библиотеке OpenCV используются только первые две строки матрицы для аффинного преобразования, требуется убрать последнюю строку после умножения промежуточных квадратных матриц преобразования.

Листинг 2.13. Вычисление матрицы преобразования для поворота изображения на угол $\varphi = 17°$ вокруг центра изображения с использованием библиотеки OpenCV и языка программирования C++.

```
double phi = 17.0 * M_PI / 180;
Mat T1 = (Mat_<double>(3, 3) <<
          1, 0, -(I.cols - 1) / 2.0,
          0, 1, -(I.rows - 1) / 2.0,
          0, 0, 1);
Mat T2 = (Mat_<double>(3, 3) <<
          cos(phi), -sin(phi), 0,
          sin(phi), cos(phi), 0,
          0, 0, 1);
Mat T3 = (Mat_<double>(3, 3) <<
          1, 0, (img.cols - 1) / 2.0,
          0, 1, (img.rows - 1) / 2.0,
          0, 0, 1);
Mat T = Mat(T3 * T2 * T1, Rect(0, 0, 3, 2));
```

Листинг 2.14. Вычисление матрицы преобразования для поворота изображения на угол $\varphi = 17°$ вокруг центра изображения с использованием библиотеки OpenCV и языка программирования Python.

```
phi = 17.0 * math.pi / 180
T1 = np.float32(
  [[1, 0, -(cols - 1) / 2.0],
  [0, 1, -(rows - 1) / 2.0],
  [0, 0, 1]])
T2 = np.float32(
  [[math.cos(phi), -math.sin(phi), 0],
  [math.sin(phi), math.cos(phi), 0],
  [0, 0, 1]])
T3 = np.float32(
  [[1, 0, (cols - 1) / 2.0],
  [0, 1, (rows - 1) / 2.0],
  [0, 0, 1]])
T = np.matmul(T3, np.matmul(T2, T1))[0:2, :]
```

В библиотеке OpenCV также присутствует функция `getRotationMatrix2D()`, предназначенная для вычисления матрицы преобразования для поворота изображения против часовой стрелки на произвольный угол вокруг произвольной точки изображения одновременно с масштабированием изображения. Например, вычисление матрицы преобразования для поворота изображения на угол $\varphi = 17°$ вокруг центра изображения с использованием языка программирования C++ будет выглядеть следующим образом:

```
double phi = 17.0;
T = cv::getRotationMatrix2D(
  Point2f(float((I.cols - 1) / 2.0),
          float((I.rows - 1) / 2.0)), -phi, 1);
```

А с использованием языка программирования Python:

```
phi = 17.0
T = cv2.getRotationMatrix2D(
  ((cols - 1) / 2.0, (rows - 1) / 2.0), -phi, 1)
```

Рис. 2.1 — Эвклидовы преобразования, верхний ряд слева направо: исходное изображение, сдвиг, отражение, вращение; нижний ряд: однородное масштабирование.

Аффинное отображение — это отображение, при котором параллельные прямые переходят в параллельные прямые, пересекающиеся в пересекающиеся, скрещивающиеся в скрещивающиеся; сохраняются отношения длин отрезков, лежащих на одной прямой (или на параллельных прямых), и отношения площадей фигур. Базовыми преобразованиями являются эвклидовы преобразования, скос и неоднородное масштабирование. Произвольное аффинное преобразование можно получить при помощи последовательного произведения матриц базовых преобразований. В непрерывной геометрии любое аффинное преобразование имеет обратное аффинное преобразование, а произведение прямого и обратного дает единичное преобразование, которое оставляет все точки на месте. Аффинные преобразования являются подмножеством проекционных преобразований.

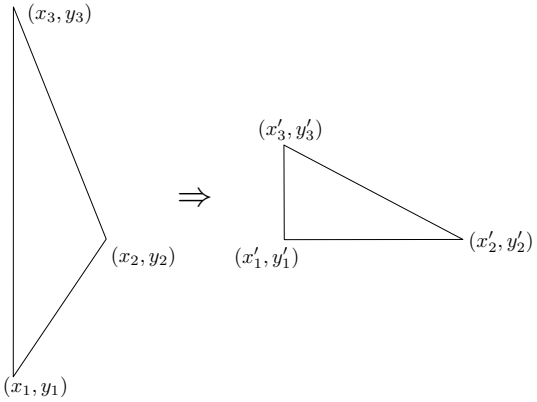


Рис. 2.2 — Аффинное отображение.

В библиотеке OpenCV присутствует функция `getAffineTransform()`, предназначенная для вычисления матрицы произвольного аффинного преобразования, заданного набором координат исходных точек изображения $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ и их координат после преобразования $(x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3)$. Программная реализация вычисления матрицы преобразования для произвольного аффинного преобразования с использованием

языка программирования C++ будет выглядеть следующим образом:

```
vector <Point2f> pts_src =
  { {50, 300}, {150, 200}, {50, 50} };
vector <Point2f> pts_dst =
  { {50, 200}, {250, 200}, {50, 10} };
T = cv::getAffineTransform(pts_src, pts_dst);
```

А с использованием языка программирования Python:

```
pts_src = np.float32([[50, 300], [150, 200], [50, 50]])
pts_dst = np.float32([[50, 200], [250, 200], [50, 100]])
T = cv2.getAffineTransform(pts_src, pts_dst)
```

Далее, вычисленная матрица может быть применена с использованием функции `warpAffine()`.

Скос изображения

Система уравнений (2.6) и матрица преобразования координат T в случае *скоса* изображения $A = E = 1$, $B = s$, $C = D = F = 0$ примут вид:

$$\begin{cases} x' = x + sy, \\ y' = y, \end{cases} \Rightarrow T = \begin{bmatrix} 1 & 0 & 0 \\ s & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.22}$$

Листинг 2.15. Скос изображения в среде MATLAB, $s = 0.3$.

```
1   T = [1 0 0; 0.3 1 0; 0 0 1];
2   tform = affine2d(T);
3   I_bevel = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для скосов изображения определяется следующим образом:

$$T = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \end{bmatrix}, \tag{2.23}$$

Листинг 2.16. Скос изображения с использованием библиотеки OpenCV и языка программирования C++, $s = 0.3$.

```
1   double s = 0.3;
2   Mat T = (Mat_<double>(2, 3) <<
3            1, s, 0,
4            0, 1, 0);
5   Mat I_bevel;
6   warpAffine(I, I_bevel, T,
7     Size(I.cols, I.rows));
```

Листинг 2.17. Скос изображения с использованием библиотеки OpenCV и языка программирования Python, $s = 0.3$.

```
1   s = 0.3
2   T = np.float32([[1, s, 0], [0, 1, 0]])
3   I_bevel = cv.warpAffine(I, T, (cols, rows))
```

Кусочно-линейные преобразования

Кусочно-линейное отображение — это отображение, при котором изображение разбивается на части, а затем к каждой из этих частей применяются различные линейные преобразования.

Листинг 2.18. Пример кусочно-линейного отображения в среде MATLAB — левая половина изображения остается без изменений, а правая растягивается в два раза вдоль оси Ox.

```
1   imid = round(size(I,2) / 2);
2   I_left = I(:, 1:imid, :);
3   stretch = 2;
4   I_right = I(:, (imid + 1:end), :);
5   T = [stretch 0 0; 0 1 0; 0 0 1];
6   tform = affine2d(T);
7   I_scale = imwarp(I_right, tform);
8   I_piecewiselinear = [I_left I_scale];
```

При использовании библиотеки OpenCV кусочно-линейные преобразования выполняются с использованием специального типа изображений, называемых ROI — Region of Interest, которые создаются ограничением области исходного изображения. Этот тип изображений имеет общие данные с исходным, но при этом может использоваться аналогично с обычным изображением. В C++ такой объект создается с помощью операции `()` и передачи области в

качестве аргумента. Например, для создания фрагмента изображения прямоугольной формы необходимо передать результат работы функции `cv::Rect()`, как показано в следующем примере.

Листинг 2.19. Пример кусочно-линейного отображения с использованием библиотеки OpenCV и языка программирования C++ — левая половина изображения остается без изменений, а правая растягивается в два раза вдоль оси Ox.

```
1  double stretch = 2;
2  Mat T = (Mat_<double>(2, 3) <<
3           stretch, 0, 0,
4           0, 1, 0);
5  Mat I_piecewiselinear = I.clone();
6  Mat I_right = I_piecewiselinear(
7    Rect(int(I.cols / 2), 0,
8    I.cols - int(I.cols / 2), I.rows));
9  warpAffine(I_right, I_right, T,
10   Size(I.cols - int(I.cols / 2), I.rows));
```

При использовании языка программирования Python аналогичные объекты создаются средствами индексации массивов NumPy.

Листинг 2.20. Пример кусочно-линейного отображения с использованием библиотеки OpenCV и языка программирования Python — левая половина изображения остается без изменений, а правая растягивается в два раза вдоль оси Ox.

```
1  stretch = 2
2  T = np.float32([[stretch, 0, 0], [0, 1, 0]])
3  I_piecewiselinear = I.copy()
4  I_piecewiselinear[:, int(cols / 2):, :] = \
5    cv.warpAffine(
6    I_piecewiselinear[:, int(cols / 2):, :],
7    T, (cols - int(cols / 2), rows))
```

Нелинейные преобразования

При рассмотрении геометрических преобразований предполагается, что изображения получены при помощи идеальной модели камеры. В действительности формирование изображений сопровож-

Рис. 2.3 — Слева — скос, справа — кусочно-линейное растяжение.

дается различными нелинейными искажениями, см. рис. 2.4. Для их коррекции используются различные нелинейные функции.

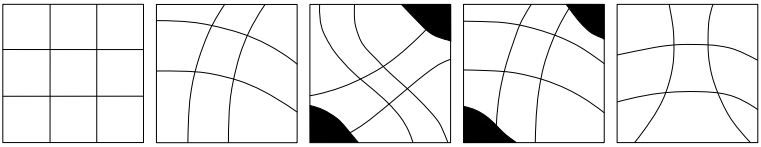


Рис. 2.4 — Примеры нелинейных искажений.

Проекционное преобразование

Проекционное отображение — это отображение, при котором прямые линии остаются прямыми линиями, однако геометрия фигуры может быть нарушена, т.к. данное отображение в общем случае не сохраняет параллельности линий. Свойством, сохраняющимся при проективном преобразовании, является *коллинеарность* точек: три точки, лежащие на одной прямой (коллинеарные), после преобразования остаются на одной прямой. Проекционное отображение может быть как *параллельным* (изменяется масштаб), так и *проективным* (изменяется геометрия фигуры).

В случае проективного отображения точки трехмерной сцены $\mathbf{P}^3$ проецируются на двумерную плоскость $\mathbf{P}^2$ изображения. Такое преобразование $\mathbf{P}^3 \to \mathbf{P}^2$ отображает евклидову точку сцены $P = (x,y,z)$ (в однородных координатах (x',y',z',w)) в точку изображения $X = (x,y)$ (в однородных координатах (x',y',w)). Для нахождения декартовых координат точек из однородных координат воспользуемся следующими соотношениями: $P = (\frac{x'}{w}, \frac{y'}{w}, \frac{z'}{w})$ — координаты вектора $\mathbf{P}$ и $X = (\frac{x'}{w}, \frac{y'}{w})$ — координаты вектора $\mathbf{X}$. Подставляя в (2.2) $w = 1$, для вектора $\mathbf{X}$ получим систему уравнений:

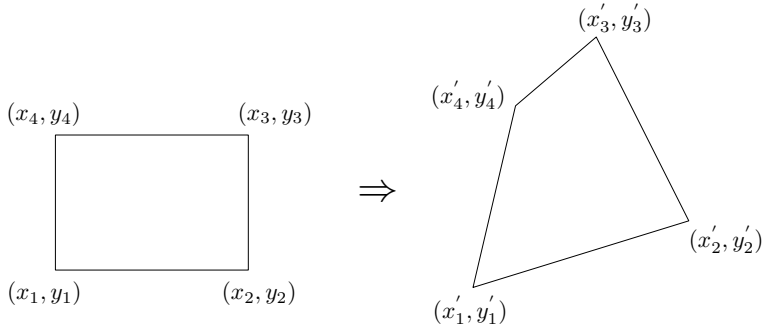


Рис. 2.5 — Проекционное проективное отображение: прямые остались прямыми, но параллельные отобразились в скрещивающиеся.

$$\begin{cases} x' = \dfrac{Ax + By + C}{Gx + Hy + I}, \\ y' = \dfrac{Dx + Ey + F}{Gx + Hy + I}, \end{cases} \Rightarrow T = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & 1 \end{bmatrix}. \tag{2.24}$$

Из-за нормирования координат на w' в общем случае проекционное отображение является нелинейным.

Листинг 2.21. Пример проективного отображения в среде MATLAB при $A = 1.1$, $B = 0.35$, $C = F = 0$, $D = 0.2$, $E = 1.1$, $G = 0.00075$, $H = 0.0005$, $I = 1$.

```
1   T = [1.1 0.35 0; 0.2 1.1 0; 0.00075 0.0005 1];
2   tform = projective2d(T);
3   I_projective = imwarp(I, tform);
```

Листинг 2.22. Пример проективного отображения с использованием библиотеки OpenCV и языка программирования C++ при $A = 1.1$, $B = 0.35$, $C = F = 0$, $D = 0.2$, $E = 1.1$, $G = 0.00075$, $H = 0.0005$, $I = 1$.

```
1   Mat T = (Mat_<double>(3, 3) <<
2            1.1, 0.2, 0.00075,
3            0.35, 1.1, 0.0005,
4            0, 0, 1);
```

```
5   Mat I_projective;
6   cv::warpPerspective(I, I_projective, T,
7     Size(I.cols, I.rows));
```

Листинг 2.23. Пример проективного отображения с использованием библиотеки OpenCV и языка программирования Python при $A = 1.1$, $B = 0.35$, $C = F = 0$, $D = 0.2$, $E = 1.1$, $G = 0.00075$, $H = 0.0005$, $I = 1$.

```
1   T = np.float32(
2     [[1.1, 0.2, 0.00075],
3     [0.35, 1.1, 0.0005],
4     0, 0, 1]])
5   I_projective = cv2.warpPerspective(I, T,
6     (cols, rows))
```

В библиотеке OpenCV присутствует функция `getPerspectiveTransform()`, предназначенная для вычисления матрицы произвольного проекционного отображения, заданного набором координат исходных точек изображения $(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$ и их координат после преобразования $(x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3), (x'_4, y'_4)$. Программная реализация вычисления матрицы преобразования для произвольного проекционного отображения с использованием языка программирования C++ будет выглядеть следующим образом:

```
vector <Point2f> pts_src =
  { {50, 461}, {461, 461}, {461, 50}, {50, 50} };
vector <Point2f> pts_dst =
  { {50, 461}, {461, 440}, {450, 10}, {100, 50} };
T = cv::getPerspectiveTransform(pts_src, pts_dst);
```

А с использованием языка программирования Python:

```
pts_src = np.float32(
  [[50, 461], [461, 461], [461, 50], [50, 50]])
pts_dst = np.float32(
  [[50, 461], [461, 440], [450, 10], [100, 50]])
T = cv2.getPerspectiveTransform(pts_src, pts_dst)
```

Полиномиальное преобразование

Полиномиальное отображение — это отображение исходного изображения с помощью полиномов. В данном случае матрица преобразования координат T будет содержать коэффициенты полиномов соответствующих порядков для координат x и y. Например, в случае полиномиального преобразования *второго порядка* система уравнений 2.6 примет вид:

$$\begin{cases} x' = a_1 + a_2 x + a_3 y + a_4 x^2 + a_5 xy + a_6 y^2, \\ y' = b_1 + b_2 x + b_3 y + b_4 x^2 + b_5 xy + b_6 y^2, \end{cases} \tag{2.25}$$

где x, y — координаты точек в одной системе координат; x', y' — координаты этих точек в другой системе координат; $a_1 \dots a_6$, $b_1 \dots b_6$ — коэффициенты преобразования.

Листинг 2.24. Пример полиномиального отображения с заданной матрицей преобразования T и в котором не используются предопределенные функции MATLAB. Результирующее преобразованное изображение не будет содержать интерполированных пикселей.

```
[numRows, numCols, Layers] = size(I);
T = [0 0; 1 0; 0 1; 0.00001 0;
    0.002 0; 0.001 0];
for k=1:1:Layers
    for y=1:1:numCols
        for x=1:1:numRows
            xnew = round(T(1,1)+T(2,1)*x+...
                T(3,1)*y+T(4,1)*x^2+...
                T(5,1)*x*y+T(6,1)*y^2);
            ynew = round(T(1,2)+T(2,2)*x+...
                T(3,2)*y+T(4,2)*x^2+...
                T(5,2)*x*y+T(6,2)*y^2);
            I_polynomial(xnew,ynew,k) = ...
                I(x,y,k);
        end
    end
end
```

Листинг 2.25. Пример полиномиального отображения с заданной матрицей преобразования T с использованием библиотеки

OpenCV и языка программирования C++. Результирующее преобразованное изображение не будет содержать интерполированных пикселей.

```
const double T[2][6] =
  { { 0, 1, 0, 0.00001, 0.002, 0.002 },
  { 0, 0, 1, 0, 0, 0 } };
Mat I_polynomial;
// Convert to floating points
if (I.depth() == CV_8U)
  I.convertTo(I_polynomial, CV_32F,
    1.0 / 255);
else
  I_polynomial = I;
// We need to process layers separately
vector<Mat> I_BGR;
cv::split(I_polynomial, I_BGR);
for (int k = 0; k < I_BGR.size(); k++)
{
  I_polynomial =
  Mat::zeros(I_BGR[k].rows, I_BGR[k].cols,
    I_BGR[k].type());
  for (int x = 0; x < I_BGR[k].cols; x++)
    for (int y = 0; y < I_BGR[k].rows; y++)
    {
      int xnew = int(round(T[0][0] +
        x * T[0][1] + y * T[0][2] +
        x * x * T[0][3] + x * y * T[0][4] +
        y * y * T[0][5]));
      int ynew = int(round(T[1][0] +
        x * T[1][1] + y * T[1][2] +
        x * x * T[1][3] + x * y * T[1][4] +
        y * y * T[1][5]));
      if (xnew >= 0 && xnew < I_BGR[k].cols
       && ynew >= 0 && ynew < I_BGR[k].rows)
        I_polynomial.at<float>(ynew, xnew) =
          I_BGR[k].at<float>(y, x);
      }
```

```
35          I_BGR[k] = I_polynomial;
36      }
37      // Merge back
38      cv::merge(I_BGR, I_polynomial);
39      // Convert back to uint if needed
40      if (I.depth() == CV_8U)
41        I_polynomial.convertTo(I_polynomial,
42          CV_8U, 255);
```

Листинг 2.26. Пример полиномиального отображения с заданной матрицей преобразования T с использованием библиотеки OpenCV и языка программирования Python. Результирующее преобразованное изображение не будет содержать интерполированных пикселей.

```
1   T = np.array([[0, 0], [1, 0], [0, 1],
2     [0.00001, 0], [0.002, 0], [0.001, 0]])
3   I_polynomial = np.zeros(I.shape, I.dtype)
4   x, y = np.meshgrid(np.arange(cols),
5                      np.arange(rows))
6   # Calculate all new X and Y coordinates
7   xnew = np.round(T[0, 0] + x * T[1, 0] +
8     y * T[2, 0] + x * x * T[3, 0] +
9     x * y * T[4, 0] +
10    y * y * T[5, 0]).astype(np.float32)
11  ynew = np.round(T[0, 1] + x * T[1, 1] +
12    y * T[2, 1] + x * x * T[3, 1] +
13    x * y * T[4, 1] +
14    y * y * T[5, 1]).astype(np.float32)
15  # Calculate mask of valid indexes
16  mask = np.logical_and(
17    np.logical_and(xnew >= 0, xnew < cols),
18    np.logical_and(ynew >= 0, ynew < rows))
19  # Apply reindexing
20  if img.ndim == 2:
21    I_polynomial[ynew[mask].astype(int),
22      xnew[mask].astype(int)] = \
23      img[y[mask], x[mask]]
24  else:
```

```
25      I_polynomial[ynew[mask].astype(int),
26        xnew[mask].astype(int), :] = \
27        img[y[mask], x[mask], :]
```

Следует отметить, что при использовании языка программирования Python попиксельная работа с изображением крайне неэффективна. Вместо этого необходимо вычислить новые координаты для каждого пикселя исходного изображения и применить их с использованием индексации NumPy. Для начального формирования массивов переиндексации используется специальный конструктор `numpy.meshgrid()`. Он создает два массива координат X и Y для каждого пикселя изображения. Затем эти координаты преобразуются и используются для сопоставления координат новых пикселей изображения со старыми. Матрица `mask` используется для исключения пикселей, вышедших за пределы допустимого диапазона после преобразования.

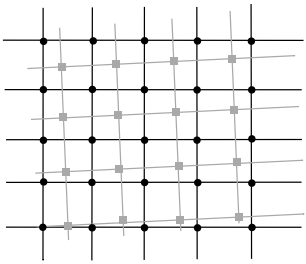


Рис. 2.6 — Смещение координат пикселей в преобразованном изображении.

Как правило, при нелинейном искажении коэффициенты $a_1 \dots a_6$, $b_1 \dots b_6$ и матрица преобразования координат T *неизвестны*. В случае, когда помимо искаженного изображения доступно исходное, можно вычислить коэффициенты преобразования, найдя пары соответствующих точек на исходном и преобразованном изображениях. Число минимально необходимых пар точек вычисляется по формуле:

$$t_{min} = \frac{(n+1)(n+2)}{2}, \tag{2.26}$$

где n — порядок преобразования.

Если имеет место полиномиальное искажение *второго порядка*, то согласно (2.26) необходимо знать координаты хотя бы *шести пар* соответствующих точек до и после трансформации: $(x_1, y_1) \dots (x_6, y_6)$ и $(x'_1, y'_1) \dots (x'_6, y'_6)$. Согласно (2.25) запишем уравнения для вычисления коэффициентов в матричном виде:

$$\begin{bmatrix} a_1 \\ \dots \\ a_6 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_6 & y_6 & x_6^2 & x_6 y_6 & y_6^2 \end{bmatrix}^{-1} \begin{bmatrix} x'_1 \\ \dots \\ x'_6 \end{bmatrix}, \tag{2.27}$$

$$\begin{bmatrix} b_1 \\ \dots \\ b_6 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_6 & y_6 & x_6^2 & x_6 y_6 & y_6^2 \end{bmatrix}^{-1} \begin{bmatrix} y'_1 \\ \dots \\ y'_6 \end{bmatrix}. \tag{2.28}$$

Для интерактивного указания одинаковых точек на исходном и преобразованном изображениях можно воспользоваться функцией MATLAB `cpselect('source.jpg','transformed.jpg')`, которая возвращает два массива `movingPoints` и `fixedPoints` с координатами преобразованного и исходного изображений соответственно.

Синусоидальное искажение

В качестве еще одного из примеров нелинейного преобразования можно рассмотреть гармоническое искажение изображения.

Листинг 2.27. Пример синусоидального искажения изображения в среде MATLAB.

```
[xi,yi] = meshgrid(1:ncols,1:nrows);
imid = round(size(I,2)/2);
u = xi + 20*sin(2*pi*yi/90);
v = yi;
tmap_B = cat(3,u,v);
resamp = makeresampler('linear','fill');
I_sinusoid = tformarray(I,[],resamp,...
    [2 1],[1 2],[],tmap_B,.3);
```

Листинг 2.28. Пример синусоидального искажения изображения с использованием библиотеки OpenCV и языка программирования C++.

```
1   Mat u = Mat::zeros(I.rows, I.cols, CV_32F);
2   Mat v = Mat::zeros(I.rows, I.cols, CV_32F);
3   for (int x = 0; x < I.cols; x++)
4     for (int y = 0; y < I.rows; y++)
5     {
6       u.at<float>(y, x) = float(x +
7         20 * sin(2 * M_PI * y / 90));
8       v.at<float>(y, x) = float(y);
9     }
10  Mat I_sinusoid;
11  remap(I, I_sinusoid, u, v, INTER_LINEAR);
```

Листинг 2.29. Пример синусоидального искажения изображения с использованием библиотеки OpenCV и языка программирования Python.

```
1   u, v = np.meshgrid(np.arange(cols),
2     np.arange(rows))
3   u = u + 20 * np.sin(2 * math.pi * v / 90)
4   I_sinusoid = \
5     cv2.remap(I, u.astype(np.float32),
6     v.astype(np.float32), cv2.INTER_LINEAR)
```

Функция библиотеки OpenCV `remap()` преобразует изображение в соответствии с новыми координатами пикселей, заданными для каждого пикселя исходного изображения.

Рис. 2.7 — Слева — полиномиальное искажение, в центре — проективное, справа — синусоидальное.

Коррекция дисторсии

При формировании изображения оптической системой на нем может возникнуть *дисторсия*. *Дисторсия* — это оптическое искажение, выражающееся в искривлении прямых линий. Световые лучи, проходящие через центр линзы, сходятся в точке, расположенной дальше от линзы, чем лучи, проходящие через ее края. Прямые линии искривляются за исключением тех, которые лежат в одной плоскости с оптической осью. Например, изображение квадрата, центр которого пересекает оптическая ось, имеет вид подушки (*подушкообразная дисторсия*) при положительной дисторсии и вид бочки (*бочкообразная дисторсия*) при отрицательной дисторсии (рис. 2.8).

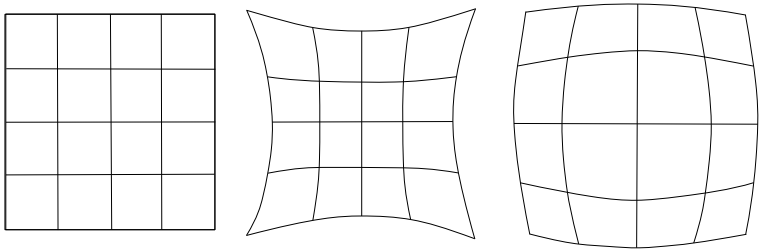


Рис. 2.8 — Примеры дисторсий. Слева — исходное изображение, по центру — подушкообразная дисторсия, справа — бочкообразная.

Пусть $\mathbf{r} = (x,y)$ — вектор, задающий две координаты в плоскости, расположенной перпендикулярно оптической оси. Для идеального изображения все лучи, вышедшие из этой точки и прошедшие через оптическую систему, попадут в точку изображения с координатами $\mathbf{R}$, которые определяются по формуле:

$$\mathbf{R} = b_0\mathbf{r}, \tag{2.29}$$

где b_0 — коэффициент линейного увеличения.

Если присутствует дисторсия более высокого порядка (для осесимметричных оптических систем дисторсия может быть только нечетных порядков: третьего, пятого, седьмого и т.д.), то в выражение (2.29) необходимо добавить соответствующие слагаемые:

$$\mathbf{R} = b_0\mathbf{r} + F_3 r^2\mathbf{r} + F_5 r^4\mathbf{r} + \ldots, \tag{2.30}$$

где r — длина вектора $\mathbf{r}$; $F_i, i = 3{,}5{,}\ldots{,}n$ — коэффициенты дисторсии n-го порядка, которые вносят наибольший вклад в искажение формы изображения. При дисторсии третьего порядка, если коэффициент F_3 имеет тот же знак, что и b_0: $\mathrm{sign}(F_3) = \mathrm{sign}(b_0)$, возникает подушкообразное искажение, в противном случае — бочкообразное.

Для коррекции дисторсии используется подход, описанный выше для проективного отображения. Используется изображение регулярной сетки и его искаженное изображение, находятся пары точек на этих изображениях и вычисляются коэффициенты корректирующего преобразования.

Бочкообразная дисторсия

Листинг 2.30. Пример наложения бочкообразной дисторсии пятого порядка на исходное изображение в среде MATLAB.

```
 1 [xi,yi] = meshgrid(1:ncols,1:nrows);
 2 imid = round(size(I,2)/2);
 3 xt = xi(:) - imid;
 4 yt = yi(:) - imid;
 5 [theta,r] = cart2pol(xt,yt);
 6 F3 = .000001;
 7 F5 = .0000012;
 8 R = r + F3 * r.^3 + F5 * r.^5;
 9 [ut,vt] = pol2cart(theta, R);
10 u = reshape(ut, size(xi)) + imid;
11 v = reshape(vt, size(yi)) + imid;
12 tmap_B = cat(3, u, v);
13 resamp = makeresampler('linear','fill');
14 I_barrel = tformarray(I,[],resamp,...
15     [2 1],[1 2],[],tmap_B,.3);
```

Для формирования сетки изображения используется функция `meshgrid()`. Затем на строках 2-4 происходит получение отцентрированного набора координат всех пикселей. После этого все координаты функцией `cart2pol()` переводятся из декартовой системы координат в полярную, для более удобного применения фор-

мулы (2.30) через радиус-векторы каждой точки. Затем координаты пикселей преобразуются при помощи функций `pol2cart()` и `reshape()` обратно в декартовы координаты. Из итоговой сетки формируется трехмерная матрица преобразования `tmap_B` для функции `tformarray`, которая производит само преобразование исходного изображения по заданной сетке.

Листинг 2.31. Пример наложения бочкообразной дисторсии пятого порядка на исходное изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Do the meshgrid() routine
2   Mat xi, yi;
3   std::vector<float> t_x, t_y;
4   for (int i = 0; i < img.cols; i++)
5     t_x.push_back(float(i));
6   for (int i = 0; i < img.rows; i++)
7     t_y.push_back(float(i));
8   cv::repeat(Mat(t_x).reshape(1, 1),
9     I.rows, 1, xi);
10  cv::repeat(Mat(t_y).reshape(1, 1).t(),
11    1, I.cols, yi);
12  // Shift and normalize grid
13  double xmid = xi.cols / 2.0;
14  double ymid = xi.rows / 2.0;
15  xi -= xmid;
16  xi /= xmid;
17  yi -= ymid;
18  yi /= ymid;
19  // Convert to polar and do transformation
20  Mat r, theta;
21  cartToPolar(xi, yi, r, theta);
22  double F3(0.1), F5(0.12);
23  Mat r3, r5;
24  pow(r, 3, r3); // r3 = r^3
25  pow(r, 5, r5); // r5 = r^5
26  r += r3 * F3;
27  r += r5 * F5;
28  // Undo conversion, normalization and shift
```

```
29  Mat u, v;
30  polarToCart(r, theta, u, v);
31  u *= xmid;
32  u += xmid;
33  v *= ymid;
34  v += ymid;
35  // Do remapping
36  Mat I_barrel;
37  remap(I, I_barrel, u, v, INTER_LINEAR);
```

Листинг 2.32. Пример наложения бочкообразной дисторсии пятого порядка на исходное изображение с использованием библиотеки OpenCV и языка программирования Python.

```
1   # Create mesh grid for X, Y
2   xi, yi = np.meshgrid(np.arange(cols),
3     np.arange(rows))
4   # Shift and normalize grid
5   xmid = cols / 2.0
6   ymid = rows / 2.0
7   xi = xi - xmid
8   yi = yi - ymid
9   # Convert to polar and do transformation
10  r, theta = cv.cartToPolar(xi / xmid, yi / ymid)
11  F3 = 0.1
12  F5 = 0.12
13  r = r + F3 * r ** 3 + F5 * r ** 5
14  # Undo conversion, normalization and shift
15  u, v = cv.polarToCart(r, theta)
16  u = u * xmid + xmid;
17  v = v * ymid + ymid;
18  # Do remapping
19  I_barrel = \
20    cv2.remap(I, u.astype(np.float32),
21    v.astype(np.float32), cv.INTER_LINEAR)
```

Подушкообразная дисторсия

Листинг 2.33. Пример наложения подушкообразной дисторсии третьего порядка на исходное изображение.

```
1 F3 = -0.003;
2 R = r + F3 * r.^2;
```

Рис. 2.9 — Слева — исходное изображение, в центре — бочкообразная дисторсия, справа — подушкообразная дисторсия.

«Сшивка» изображений

Геометрические преобразования можно использовать, например, для построения мозаики из нескольких изображений. Мозаика («сшивка», «склейка») — это объединение двух или более изображений в единое целое, причем системы координат склеиваемых изображений могут отличаться из-за разного ракурса съемки, изменения положения камеры или движения самого объекта. Однако необходимо, чтобы оба изображения имели области перекрытия, т.е. на них присутствовали одинаковые объекты.

Основной задачей обработки таких изображений является приведение их в общую систему координат. В качестве общей системы координат можно использовать систему первого изображения, тогда требуется найти преобразование координат всех пикселей второго изображения (x,y) в общую систему координат (x',y'). Если имеет место полиномиальное искажение, то для пересчета координат можно воспользоваться системой уравнений (2.25). В случае аффиного отображения система (2.25) примет вид:

$$\begin{cases} x' = a_1 + a_2 x + a_3 y, \\ y' = b_1 + b_2 x + b_3. \end{cases} \tag{2.31}$$

Поэтому необходимо найти лишь по 3 коэффициента преобразования по каждой координате:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix}, \tag{2.32}$$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} \begin{bmatrix} y'_1 \\ y'_2 \\ y'_3 \end{bmatrix}. \tag{2.33}$$

Для этого на обоих изображениях следует выбрать соответствующие пары точек (три пары в случае аффинного искажения и не менее шести пар в случае полиномиального искажения). Интерактивно выполнить эту операцию можно при помощи функции MATLAB `cpselect()`, либо воспользоваться специальными алгоритмами. Например, можно определить эти точки на основе коэффициента корреляции, который в MATLAB вычисляется с помощью функции `corr2()`.

Рассмотрим простой случай «склейки» двух неискаженных изображений, имеющих одинаковую ширину. Необходимо склеить их по вертикали, т.е. добавить к первому второе снизу. Однако, граница склейки неизвестна. Эту задачу можно реализовать при помощи корреляционного подхода.

Листинг 2.34. Пример «склейки» изображений в среде MATLAB. Считывание верхней и нижней картинок соответственно:

```
1 topPart = imread('itmoTop.jpg');
2 botPart = imread('itmoBot.jpg');
```

Для простоты определения коэффициента корреляции преобразуем цветные изображения в полутоновые:

```
3 topPartHT = im2double(rgb2gray(topPart));
4 botPartHT = im2double(rgb2gray(botPart));
```

Определим размеры полутоновых изображений и инициализируем промежуточные массивы для вычислений:

```
5 [numRows, numCols, Layers] = size(topPart);
6 [numRowsBot, numColsBot] = size(botPartHT);
7 botPartCorrHT = zeros(intersecPart,numCols);
8 topPartCorrHT = zeros(intersecPart,numCols);
9 correlationArray = [];
```

Определим верхнюю строку нижнего изображения для вычисления коффициента корреляции со строками верхнего:

```
intersecPart = 5;
for j = 1:1:numCols
    for i = 1:1:intersecPart
        botPartCorrHT(i,j) = botPartHT(i,j);
    end
end
```

Сравним полученную строку со строками верхнего изображения и вычислим коэффициент корреляции:

```
for j = 0:1:numRows-intersecPart
    for i = 1:1:intersecPart
        topPartCorrHT(i,:) = topPartHT(i+j,:);
    end
    correlationCoefficient = ...
        corr2(topPartCorrHT, botPartCorrHT);
    correlationArray = ...
        [correlationArray
        correlationCoefficient];
    correlationCoefficient = 0;
end
[M, I] = max(correlationArray);
```

Построим цветное «склеенное» изображение `result_img` с границей склейки на основе полученного индекса, соответствующего номеру строки с максимальным коэффициентом корреляции:

```
numRowsBotCorr = numRowsBot + I - 1;
for k = 1:1:Layers
    for j = 1:1:numCols
        for i = 1:I-1
            result_img(i,j,k) = ..
                topPart(i,j,k);
        end
        for i = I:1:numRowsBotCorr
            result_img(i,j,k) = ...
                botPart(i-I+1,j,k);
        end
```

```
39      end
40  end
```

Листинг 2.35. Пример «склейки» изображений с использованием библиотеки OpenCV и языка программирования C++. Считывание верхней и нижней картинок соответственно:

```
1   Mat topPart =
2     imread("itmoTop.jpg", IMREAD_COLOR);
3   Mat botPart =
4     imread("itmoBot.jpg", IMREAD_COLOR);
```

Создадим шаблон для вычисления коэффициента корреляции как нижние 10 строк от верхнего изображения:

```
5   int templ_size = 10;
6   Mat templ = topPart(
7     Rect(0, topPart.rows - templ_size - 1,
8          topPart.cols, templ_size));
```

Вычислим коэффициенты корреляции шаблона с нижним изображением:

```
9   Mat res;
10  cv::matchTemplate(botPart, templ, res,
11    TM_CCOEFF);
```

В качестве точки «склейки» выберем точку с максимальным значением коэффициента корреляции:

```
12  Mat res;  double min_val, max_val;
13  Point2i min_loc, max_loc;
14  minMaxLoc(res, &min_val, &max_val,
15    &min_loc, &max_loc);
```

Построим «склеенное» изображение `result_img` с границей склейки на основе полученного индекса:

```
16  Mat result_img =
17    Mat::zeros(topPart.rows + botPart.rows -
18                max_loc.y - templ_size,
19                topPart.cols, topPart.type());
```

Скопируем данные из верхней части изображения:

```
20   topPart.copyTo(result_img(Rect(0, 0,
21     topPart.cols, topPart.rows)));
```

И из нижней части:

```
22   botPart(Rect(0, max_loc.y + templ_size,
23     botPart.cols, botPart.rows -
24     max_loc.y - templ_size)).
25     copyTo(result_img(Rect(0, topPart.rows,
26     botPart.cols,
27     botPart.rows - max_loc.y - templ_size)));
```

Листинг 2.36. Пример «склейки» изображений с использованием библиотеки OpenCV и языка программирования Python. Считывание верхней и нижней картинок соответственно:

```
1   topPart = cv2.imread("itmoTop.jpg", \
2                        cv.IMREAD_COLOR)
3   botPart = cv2.imread("itmoBot.jpg", \
4                        cv.IMREAD_COLOR)
```

Создадим шаблон для вычисления коэффициента корреляции как нижние 10 строк от верхнего изображения и вычислим коэффициенты корреляции шаблона с нижним изображением:

```
5   # Match template
6   templ_size = 10
7   templ = topPart[-templ_size:, :, :]
8   res = cv2.matchTemplate(botPart, templ,
9     cv2.TM_CCOEFF)
```

В качестве точки «склейки» выберем точку с максимальным значением коэффициента корреляции:

```
10  min_val, max_val, min_loc,
11    max_loc = cv2.minMaxLoc(res)
```

Построим «склеенное» изображение `result_img` с границей склейки на основе полученного индекса:

```
12  result_img = np.zeros((topPart.shape[0] +
13    botPart.shape[0] - max_loc[1] -
14    templ_size, topPart.shape[1],
```

```
15      topPart.shape[2]), dtype = np.uint8)
16    result_img[0:topPart.shape[0], :, :] = \
17      topPart
18    result_img[topPart.shape[0]:, :, :] = \
19      botPart[max_loc[1] + templ_size:, :, :]
```

Рис. 2.10 — Слева — `itmoTop.jpg`, в центре — `itmoBot.jpg`, справа — `result_img`.

Автоматическая склейка изображений в библиотеке OpenCV

Библиотека OpenCV предоставляет специальный класс для автоматического «склеивания» изображений, который называется `Stitcher`. Он предназначен для автоматической «склейки» панорамных фотографий (параметр `cv::Stitcher::PANORAMA` в C++ и `cv2.Stitcher_PANORAMA` в Python), и для автоматической «склейки» отсканированных изображений (параметр `cv::Stitcher::SCANS` в C++ и `cv2.Stitcher_SCANS` в Python). Работать с объектом `Stitcher` очень просто, что продемонстрировано в следующем примере программной реализации склейки трех фотографий в одно панорамное изображение.

Листинг 2.37. Пример автоматический «склейки» изображений с использованием библиотеки OpenCV и языка программирования C++. Создание объекта `Stitcher`:

```
1   Ptr<Stitcher> stitcher =
2     cv::Stitcher::create(Stitcher::PANORAMA);
```

Далее необходимо создать массив исходных изображений для «склейки»:

```
3   vector<Mat> imgs;
4   imgs.push_back(I_1);
5   imgs.push_back(I_2);
```

```
6   imgs.push_back(I_3);
```

И передать этот массив в метод `stitch()`:

```
7   Mat I_stitch;
8   Stitcher::Status status =
9     stitcher->stitch(imgs, I_stitch);
```

Листинг 2.38. Пример автоматический «склейки» изображений с использованием библиотеки OpenCV и языка программирования Python. Создание объекта `Stitcher`:

```
1   stitcher = \
2     cv2.Stitcher.create(cv.Stitcher_PANORAMA)
```

Далее необходимо создать массив исходных изображений для «склейки» и передать его в метод `stitch()`:

```
3   status, I_stitch = \
4     stitcher.stitch([I_1, I_2, I_3])
```

Значение `status`, возвращаемое методом `stitch()`, показывает результат «склейки». В случае успеха он равен `cv::Stitcher::OK` для C++ и `cv2.Stitcher_OK` для Python, а в `I_stitch` сохраняется «склеенное» изображение.

Порядок выполнения работы

1. *Простейшие геометрические преобразования.* Выбрать произвольное изображение. Выполнить над ним линейные и нелинейные преобразования (эвклидовы, аффинные и проективные отображения).

2. *Коррекция дисторсии.* Выбрать два произвольных изображения: с подушкообразной и с бочкообразной дисторсией. Выполнить коррекцию изображений.

3. *«Склейка» изображений.* Выбрать два изображения (снимки с фотокамеры, фрагменты сканированного изображения и пр.), на которых имеется область пересечения. Выполнить коррекцию второго изображения для его перевода в систему координат первого; затем выполнить автоматическую «склейку» из двух изображений в одно.

Содержание отчета

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

Вопросы к защите лабораторной работы

1. Каким образом можно выполнить поворот изображения, не используя матрицу поворота?
2. Какое минимальное количество соответствующих пар точек необходимо задать на исходном и искаженном изображениях, если порядок преобразования $n = 4$?
3. После геометрического преобразования изображения могут появиться пиксели с неопределенными значениями интенсивности. С чем это связано и как решается данная проблема?

Лабораторная работа №3
Фильтрация и выделение контуров

Цель работы

Освоение основных способов фильтрации изображений от шумов и выделения контуров.

Методические рекомендации

До начала работы студенты должны ознакомиться с основными функциями среды MATLAB или библиотеки OpenCV для работы фильтрацией изображений и методами низкочастной и высокочастотной фильтрации. Лабораторная работа рассчитана на 5 часов.

Теоретические сведения

Типы шумов

Цифровые изображения, полученные различными оптико-электронными приборами, могут содержать в себе разнообразные искажения, обусловленные разного рода помехами, которые принято называть *шумом*. Шум на изображении затрудняет его обработку автоматическими средствами, и, поскольку шум может иметь различную природу, для его успешного подавления необходимо определить адекватную математическую модель. Рассмотрим наиболее распространенные модели шумов. В среде MATLAB шум может быть наложен на изображение с помощью функции `imnoise()`. К сожалению, в библиотеке OpenCV отсутствуют функции для наложения шумов на изображения, но они могут быть реализованы используя возможности данной библиотеки. С другой стороны, в библиотеке `SciPy` для языка порграммирования Python есть функция `skimage.util.random_noise()`, предназначенная для наложения шумов на изображение и аналогичная функции `imnoise()` из среды MATLAB.

Импульсный шум

При импульсном шуме сигнал искажается выбросами с очень большими отрицательными или положительными значениями ма-

Рис. 3.1 — Исходное полутоновое изображение.

лой длительностью и может возникать, например, из-за ошибок декодирования. Такой шум приводит к появлению на изображении белых («соль») или черных («перец») точек, поэтому зачастую называется *точечным* шумом. Для его описания следует принять во внимание тот факт, что появление шумового выброса в каждом пикселе $I(x,y)$ не зависит ни от качества исходного изображения, ни от наличия шума в других точках и имеет вероятность появления p, причем значение интенсивности пикселя $I(x,y)$ будет изменено на значение $d \in [0,255]$:

$$I(x,y) = \begin{cases} d, \text{ с вероятностью } p, \\ s_{x,y}, \text{ с вероятностью } (1-p), \end{cases} \tag{3.1}$$

где $s_{x,y}$ — интенсивность пикселя исходного изображения, I — зашумленное изображение, если $d = 0$ — шум типа «перец», если $d = 255$ — шум типа «соль».

В среде MATLAB импульсный шум задается параметром `'salt & pepper'` функции `imnoise()`: `imnoise(I,'salt & pepper')`. При использовании языка программирования Python наложение данного шума выполняется функцией `skimage.util.random_noise(I, 's&p')` библиотеки обработки изображений `SciPy` . Необязательный параметр `amount` используется для изменения вероятности добавления шума к пикселю. Следующий необязательный параметр `salt_vs_pepper` используется для задания соотношения вероятностей шумов типа «соль» и «перец». Функция `skimage.util.random_noise(I, 'salt')` накладывает только шум типа «соль», а функция `skimage.util.random` — только шум ти-

па «перец». В библиотеке OpenCV отсутствует функция для наложения импульсного шума, однако она может быть реализована на основе матричных операций над изображениями (используя класс `Mat` в C++ и массивы NumPy в Python). В Приложении 3.1 представлен пример программной реализации наложения импульсного шума с использованием библиотеки OpenCV.

Аддитивный шум

Аддитивный шум описывается следующим выражением:

$$I_{new}(x,y) = I(x,y) + \eta(x,y), \tag{3.2}$$

где I_{new} — зашумленное изображение, I — исходное изображение, η — не зависящий от сигнала аддитивный шум с гауссовым или любым другим распределением функции плотности вероятности.

Мультипликативный шум

Мультипликативный шум описывается следующим выражением:

$$I_{new}(x,y) = I(x,y) \cdot \eta(x,y), \tag{3.3}$$

где I_{new} — зашумленное изображение, I — исходное изображение, η — не зависящий от сигнала мультипликативный шум, умножающий зарегистрированный сигнал. В качестве примера можно привести зернистость фотопленки, ультразвуковые изображения и т.д. Частным случаем мультипликативного шума является *спекл*-шум, который появляется на изображениях, полученных устройствами с когерентным формированием изображений, например, медицинскими сканерами или радарами. На таких изображениях можно отчетливо наблюдать светлые пятна, крапинки (спеклы), которые разделены темными участками изображения.

В среде MATLAB спекл-шум накладывается на изображение `I` функцией `imnoise(I,'speckle')`. В языке программирования Python спекл-шум можно наложить на изображение функцией `skimage.util.random_noise(I, 'speckle')` библиотеки `SciPy`. Необязательные параметры `mean` и `var` используются для задания параметров нормального распределения, используемого при наложении шума. В языке программирования C++ необходимо реализовывать данный тип шума самостоятельно. В приложении 3.1

представлен пример программной реализации наложения шумов с использованием библиотеки OpenCV и языков программирования C++ и Python.

Гауссов (нормальный) шум

Гауссов шум на изображении может возникать в следствие недостатка освещенности сцены, высокой температуры и т.д. Модель шума широко распространена в задачах низкочастотной фильтрации изображений. Функция распределения плотности вероятности $p(z)$ случайной величины z описывается следующим выражением:

$$p(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(z-\mu)^2}{2\sigma^2}}, \tag{3.4}$$

где z — интенсивность изображения (например, для полутонового изображения $z \in [0{,}255]$), μ — среднее (математическое ожидание) случайной величины z, σ — среднеквадратичное отклонение, дисперсия σ^2 определяет мощность вносимого шума. Примерно 67% значений случайной величины z сосредоточено в диапазоне $[(\mu - \sigma),(\mu + \sigma)]$ и около 96% в диапазоне $[(\mu - 2\sigma),(\mu + 2\sigma)]$.

В среде MATLAB шум может быть задан с помощью функции `imnoise(I,'gaussian')` или `imnoise(I,'localvar')` в случае нулевого математического ожидания. В языке программирования Python шум Гаусса можно наложить на изображение функцией `skimage.util.random_noise(I, 'gaussian')` библиотеки `SciPy`. Необязательные параметры `mean` и `var` используются для задания параметров нормального распределения, используемого при наложении шума. Функция `skimage.util.random_noise(I, 'localvar')` позволяет задать локальную дисперсию шума для каждой точки изображения дополнительным параметром `local_vars`. В языке программирования C++ необходимо реализовывать данный тип шума самостоятельно. В приложении 3.1 представлен пример программной реализации наложения шумов с использованием библиотеки OpenCV и языков программирования C++ и Python.

Шум квантования

Зависит от выбранного шага квантования и самого сигнала. Шум квантования может приводить, например, к появлению лож-

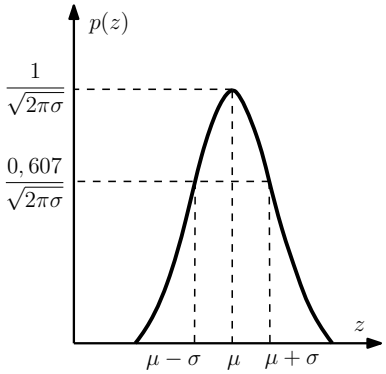


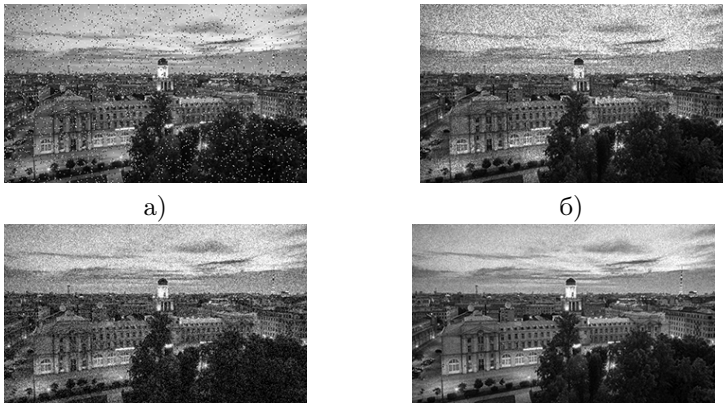
Рис. 3.2 — Функция распределения плотности вероятности $p(z)$.

ных контуров вокруг объектов или убирать слабо контрастные детали на изображении. Такой шум не устраняется.

Приближенно шум квантования можно описать распределением Пуассона и задать функцией `imnoise(I,'poisson')` в среде MATLAB. В языке программирования Python шум квантования можно наложить, используя функцию `skimage.util.random_noise(I, 'poisson')` библиотеки `SciPy`. В языке программирования C++ необходимо реализовывать данный тип шума самостоятельно. В приложении 3.1 представлен пример программной реализации наложения шумов с использованием библиотеки OpenCV и языков программирования C++ и Python.

Фильтрация изображений

Рассмотрим основные методы фильтрации изображений. Если для вычисления значения интенсивности каждого пикселя учитываются значения соседних пикселей в некоторой окрестности, то такое преобразование называется *локальным*, а сама окрестность — *окном*, представляющим собой некоторую матрицу, называемую *маской*, *фильтром*, *ядром фильтра*, а сами значения элементов матрицы называются *коэффициентами*. Центр маски совмещается с анализируемым пикселем, а коэффициенты маски умножаются на значения интенсивностей пикселей, накрытых маской. Как



а) б) в) г)

Рис. 3.3 — Результат наложения: а) шума типа «соль» и «перец», б) спекл-шума, в) нормального шума, г) шума Пуассона.

правило, маска имеет квадратную форму размера 3×3, 5×5 и т.п. Фильтрация изображения I, имеющего размеры $M \times N$, с помощью маски размера $m \times n$ описывается формулой:

$$I_{new}(x,y) = \sum_s \sum_t w(s,t) I(x+s, y+t), \tag{3.5}$$

где s и t — координаты элементов маски относительно ее центра (в центре $s = t = 0$). Такого рода преобразования называются *линейными*. После вычисления нового значения интенсивности пикселя $I_{new}(x,y)$ окно w, в котором описана маска фильтра, сдвигается и вычисляется интенсивность следующего пикселя, поэтому подобное преобразование называется *фильтрацией в скользящем окне*.

В среде MATLAB фильтрация изображения может быть осуществлена при помощи функции `filter2(mask,I)`, где матрица `mask` задает маску фильтра. Маска может задаваться либо вручную, либо с помощью функции `fspecial()`. В библиотеке OpenCV фильтрация изображений выполняется функцией `cv::filter2D(src, dst, ddepth, kernel)` в C++ и `dst = cv.filter2D(src, ddest, kernel)` в Python. Параметр `src` — входное изображение, `dst` — выходное изображение, `kernel` — мас-

ка фильтра, а параметр `ddepth` задает глубину цвета выходного изображения. Значение глубины цвета -1 сохраняет глубину цвета исходного изображения неизменной. Маску фильтра можно задать вручную, создав новый объект типа `Mat` в C++:

```
mask = (Mat_<double>(3, 3) << 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

Или NumPy массив в Python:

```
mask = np.float64([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
```

Низкочастотная фильтрация

Низкочастотные пространственные фильтры ослабляют высокочастотные компоненты (области с сильным изменением интенсивностей) и оставляют низкочастотные компоненты изображения без изменений. Используются для снижения уровня шума и удаления высокочастотных компонент, что позволяет повысить точность исследования содержания низкочастотных компонент. В результате применения низкочастотных фильтров получим сглаженное или размытое изображение. Главными отличительными особенностями ядра низкочастотного фильтра являются:

1. неотрицательные коэффициенты маски;
2. сумма всех коэффициентов равна единице.

Примеры ядер низкочастотных фильтров:

$$w = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}, w = \frac{1}{10}\begin{bmatrix}1 & 1 & 1\\ 1 & 2 & 1\\ 1 & 1 & 1\end{bmatrix}. \tag{3.6}$$

Рассмотрим основные виды низкочастнотных сглаживающих фильтров.

Арифметический усредняющий фильтр

Данный фильтр усредняет значение интенсивности пикселя по окрестности с использованием маски с одинаковыми коэффициентами, например, для маски размером 3×3 коэффициенты равны

1/9, при 5×5 — 1/25. Благодаря такому нормированию значение результата фильтрации будет приведено к диапазону интенсивностей исходного изображения. Графически двумерная функция, описывающая маску фильтра, похожа на параллелепипед, поэтому в англоязычной литературе используется название *box*-фильтр. Арифметическое усреднение достигается при использовании следующей формулы:

$$I_{new}(x,y) = \frac{1}{m \cdot n} \sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j), \tag{3.7}$$

где $I_{new}(x,y)$ — значение интенсивности пикселя отфильтрованного изображения, $I(i,j)$ — значение интенсивностей пикселей исходного изображения в маске, m и n — ширина и высота маски фильтра соответственно. Данный алгоритм эффективен для слабо зашумленных изображений.

В среде MATLAB фильтрация изображения `I` с маской размера 3×3 может быть осуществлена при помощи функции `filter2(fspecial('average',3),I)`. При использовании библиотеки OpenCV аналогичная фильтрация может быть осуществлена при помощи функции `cv::blur(I, I_out, Size(3, 3))` в C++ или `cv2.blur(I, (3, 3))` в Python.

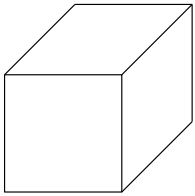


Рис. 3.4 — Графическое представление *box*-фильтра.

Геометрический усредняющий фильтр

Геометрическое усреднение рассчитывается при помощи формулы:

$$I_{new}(x,y) = \left[\prod_{i=0}^{m} \prod_{j=0}^{n} I(i,j) \right]^{\frac{1}{m \cdot n}}. \tag{3.8}$$

Эффект от применения данного фильтра аналогичен предыдущему методу, однако отдельные объекты исходного изображения искажаются меньше. Фильтр может использоваться для подавления высокочастотного аддитивного шума с лучшими статистическими характеристиками по сравнению с арифметическим усредняющим фильтром.

Геометрический усредняющий фильтр может быть задан через арифметический усредняющий фильтр и функции логарифма и экспоненты, что делает упрощает реализацию в матричном виде:

$$I_{new}(x,y) = e^{\frac{1}{m \cdot n} \sum_{i=0}^{m} \sum_{j=0}^{n} \ln(I(i,j))}. \tag{3.9}$$

Гармонический усредняющий фильтр

Фильтр базируется на выражении:

$$I_{new}(x,y) = \frac{m \cdot n}{\sum_{i=0}^{m} \sum_{j=0}^{n} \frac{1}{I(i,j)}}. \tag{3.10}$$

Фильтр хорошо подавляет шумы типа «соль» и не работает с шумами типа «перец».

Контргармонический усредняющий фильтр

Фильтр базируется на выражении:

$$I_{new}(x,y) = \frac{\sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j)^{Q+1}}{\sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j)^{Q}}, \tag{3.11}$$

где Q — порядок фильтра. Контргармонический фильтр является обобщением усредняющих фильтров и при $Q > 0$ подавляет шумы типа «перец», а при $Q < 0$ — шумы типа «соль», однако одновременное удаление белых и черных точек невозможно. При $Q = 0$ фильтр превращается в арифметический, а при $Q = -1$ — в гармонический.

Фильтр Гаусса

Пиксели в скользящем окне, расположенные ближе к анализируемому пикселю, должны оказывать большее влияние на результат фильтрации, чем крайние. Поэтому коэффициенты весов маски можно описать колоколообразной функцией Гаусса (3.4). При фильтрации изображений используется двумерный фильтр Гаусса:

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-x^2}{2\sigma^2}} \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-y^2}{2\sigma^2}}. \tag{3.12}$$

Чем больше параметр σ, тем сильнее размывается изображение. Как правило, радиус фильтра $r = 3\sigma$. В таком случае размер маски $2r + 1 \times 2r + 1$ и размер матрицы $6\sigma + 1 \times 6\sigma + 1$. За пределами данной окрестности значения функции Гаусса будут пренебрежимо малы. В среде MATLAB фильтрация изображения фильтром Гаусса может быть осуществлена при помощи функции `imgaussfilt(I)`. При использовании OpenCV фильтрация изображения фильтром Гаусса может быть осуществлена при помощи функции `cv::GaussianBlur(I, I_out)` в C++ и функции `cv2.cv::GaussianBlur(I)` в Python.

Нелинейная фильтрация

Низкочастотные фильтры линейны и оптимальны в случае, когда имеет место нормальное распределение помех на цифровом изображении. Линейные фильтры локально усредняют импульсные помехи, сглаживая изображения. Для устранения импульсных помех лучше использовать нелинейные, например, *медианные* фильтры.

Медианная фильтрация

В классическом медианном фильтре используется маска с единичными коэффициентами. Произвольная форма окна может задаваться при помощи нулевых коэффициентов. Значения интенсивностей пикселей в окне представляются в виде вектора-столбца и сортируются по возрастанию. Отфильтрованному пикселю присвается медианное (среднее) в ряду значение интенсивности. Номер медианного элемента после сортировки может быть вычислен по формуле $n = \frac{N+1}{2}$, где N — число пикселей, участвующих в

сортировке. В среде MATLAB медианный фильтр может быть реализован с использованием функции `medfilt2(I)`. При использовании OpenCV, медианная фильтрация изображения может быть выполнена при помощи функции `cv::medianBlur(I, I_out, ksize)` в C++ и `cv2.medianBlur(I, ksize)` в Python.

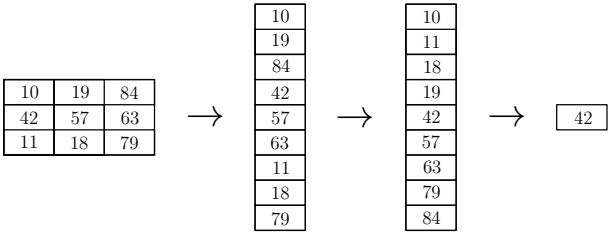


Рис. 3.5 — Иллюстрация работы медианной фильтрации в окне 3×3.

Взвешенная медианная фильтрация

В данной модификации медианной фильтрации в маске используются весовые коэффициенты (числа $2, 3$ и т.д.), чтобы отразить бóльшее влияние на результат фильтрации пикселей, расположенных ближе к фильтруемому элементу. Медианная фильтрация качественно удаляет имульсные шумы, а также на полутоновых изображениях не вносит новых значений интенсивностей. При увеличении размера окна увеличивается шумоподавляющая способность фильтра, но начинают искажаться очертания объектов. В MATLAB может быть реализован с использованием функции `medfilt2(I, [m n])`, где второй аргумент `[m n]` функции `medfilt2()` задает маску фильтра размера $m \times n$. В библиотеке OpenCV отсутствует функция для взвешенной медианной фильтрации, но она может быть реализована с использованием возможностей библиотеки. В Приложении 3.2 представлен пример программной реализации взвешенного медианного фильтра с использованием библиотеки OpenCV и языков программирования C++ и Python.

Адаптивная медианная фильтрация

В данной модификации фильтра скользящее окно размера $s \times s$ адаптивно увеличивается в зависимости от результата фильтрации.

Обозначим через z_{min}, z_{max}, z_{med} минимальное, максимальное и медианное значения интенсивностей в окне, $z_{i,j}$ — значение интенсивности пикселя с координатами (i,j), s_{max} — максимально допустимый размер окна. Алгоритм адаптивной медианной фильтрации состоит из следующих шагов:

1. Вычисление значений z_{min}, z_{max}, z_{med}, $A_1 = z_{med} - z_{min}$, $A_2 = z_{med} - z_{max}$ пикселя (i,j) в заданном окне.
   (a) Если $A_1 > 0$ и $A_2 < 0$, перейти на шаг 2. В противном случае увеличить размер окна.
   (b) Если текущий размер окна $s \leqslant s_{max}$, повторить шаг 1. В противном случае результат фильтрации равен $z_{i,j}$.
2. Вычисление значений $B_1 = z_{i,j} - z_{min}$, $B_2 = z_{i,j} - z_{max}$.
   (a) Если $B_1 > 0$ и $B_2 < 0$, результат фильтрации равен $z_{i,j}$. В противном случае результат фильтрации равен z_{med}.
3. Изменение координат (i,j).
   (a) Если не достигнут предел изображения, перейти на шаг 1. В противном случае фильтрация окончена.

Основной идеей является увеличение размера окна до тех пор, пока алгоритм не найдет медианное значение, не являющееся импульсным шумом, или пока не достигнет максимального размера окна. В последнем случае алгоритм вернет величину $z_{i,j}$.

Ранговая фильтрация

Обобщением медианной фильтрации является *ранговый фильтр* порядка r, выбирающий из полученного вектора-столбца элементов маски пиксель с номером $r \in [1,N]$, который и будет являться результатом фильтрации.

1. Если число пикселей в окне N нечетное и $r = \frac{N+1}{2}$, тогда ранговый фильтр является *медианным*. В случае окна 3×3 в MATLAB можно воспользоваться функцией `ordfilt2(I,5,ones(3,3))`.

2. Если $r = 1$, фильтр выбирает наименьшее значение интенсивности и называется *min-фильтром*. В MATLAB задается функцией `ordfilt2(I,1,ones(3,3))`.

3. Если $r = N$, фильтр выбирает максимальное значение интенсивности и называется *max-фильтром*. В MATLAB задается функцией `ordfilt2(I,9,ones(3,3))`.

Ранг иногда записывается в процентах, например для *min-фильтра* ранг равен 0%, медианного — 50%, *max-фильтра* — 100%.

В библиотеке OpenCV отсутствует функция для ранговой фильтрации изображения, но она может быть реализована с использованием возможностей библиотеки. В Приложении 3.2 представлен пример программной реализации данного фильтра с использованием библиотеки OpenCV и языков программирования C++ и Python.

Винеровская фильтрация

Использует пиксельно-адаптивный метод Винера, основанный на статистических данных, оцененных из локальной окрестности каждого пикселя.

$$\mu = \frac{1}{n \cdot m} = \sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j), \tag{3.13}$$

$$\sigma^2 = \frac{1}{n \cdot m} = \sum_{i=0}^{m} \sum_{j=0}^{n} I^2(i,j) - \mu^2, \tag{3.14}$$

$$I_{new}(x,y) = \mu + \frac{\sigma^2 - v^2}{\sigma^2} \left(I(x,y) - \mu \right), \tag{3.15}$$

где μ — среднее в окрестности, σ^2 — дисперсия, v^2 — дисперсия шума. В MATLAB реализуется при помощи функции `wiener2(I,[m n])`.

В библиотеке OpenCV отсутствует функция для фильтрации изображения методом Вейнера, но она может быть реализована с использованием возможностей библиотеки. В Приложении 3.3 представлен пример программной реализации данного фильтра с использованием библиотеки OpenCV и языков программирования C++ и Python.

Высокочастотная фильтрация

Высокочастотные пространственные фильтры усиливают высокочастотные компоненты (области с сильным изменением интенсивностей) и ослабляют низкочастотные составляющие изображения. Используются для выделения перепадов интенсивностей и определения границ (контуров) на изображениях. Для этого в MATLAB может быть использована функция `edge()`. В результате применения высокочастотных фильтров повышается резкость изображения.

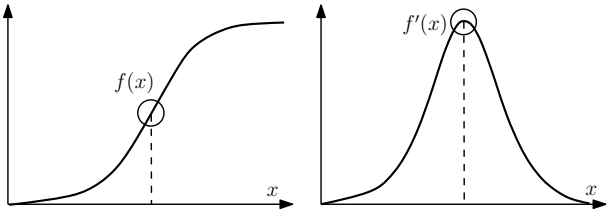


Рис. 3.6 — Функция интенсивности и ее первая производная, максимум производной соответствует краю.

Высокочастотные фильтры аппроксимируют вычисление производных по направлению, при этом приращение аргумента Δx принимается равным 1 или 2. Главными отличительными особенностями являются:

1. коэффициенты маски фильтра могут принимать отрицательные значения;

2. сумма всех коэффициентов равна нулю.

Фильтр Робертса

Фильтр Робертса работает с минимально допустимой для вычисления производной маской размерности 2×2, поэтому является быстрым и довольно эффективным. Возможные варианты масок для нахождения градиента по осям Ox и Oy:

$$G_x = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix}, \tag{3.16}$$

либо

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}. \tag{3.17}$$

В результате применения дифференциального оператора Робертса получим оценку градиента по направлениям G_x и G_y. Модуль градиента всех краевых детекторов может быть вычислен по формуле $G = \sqrt{G_x^2 + G_y^2} = |G_x| + |G_y|$, а направление градиента — по формуле $\arctan\left(\frac{G_y}{G_x}\right)$.

В MATLAB с использованием дифференциального оператора Робертса границы можно выделить при помощи функции `edge(I, 'Roberts')`. При использовании библиотеки OpenCV он реализуется двумя вызовами функции `cv::filter2D` с масками G_x and G_y и вычислением среднеквадратичного значения полученных изображений с помощью функции `cv::magnitude`. На языке программирования C++ программная реализация примет вид:

```
Mat G_x = (Mat_<double>(2, 2) << -1, 1, 0, 0);
Mat G_y = (Mat_<double>(2, 2) << 1, 0, -1, 0);
Mat I_x, I_y, I_out;
cv::filter2D(I, I_x, -1, G_x);
cv::filter2D(I, I_y, -1, G_y);
cv::magnitude(I_x, I_y, I_out);
```

На языке программирования Python программная реализация будет аналогичной:

```
G_x = np.array([[1, -1], [0, 0]])
G_y = np.array([[1, 0], [-1, 0]])
I_x = cv.filter2D(I, -1, G_x)
I_y = cv.filter2D(I, -1, G_y)
I_out = cv.magnitude(I_x, I_y)
```

Фильтр Превитта

В данном подходе используются две ортогональные маски размером 3×3, позволяющие более точно вычислить производные по осям Ox и Oy:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}. \tag{3.18}$$

В MATLAB границы фильтром Превитта можно выделить при помощи функции `edge(I, 'Prewitt')`. При использовании библиотеки OpenCV фильтр Превитта реализуется аналогично с фильтром Робертса, как было описано выше.

Фильтр Собела

Данный подход аналогичен фильтру Робертса, однако используются разные веса в масках. Типичный пример фильтра Собела:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \tag{3.19}$$

В MATLAB границы фильтром Собела можно выделить при помощи функции `edge(I, 'Sobel')`. При использовании библиотеки OpenCV он может быть реализован аналогично с фильтром Робертса, как было описано выше, или же при помощи функции `cv::Sobel(I, I_out, ddepth, dx, dy)` в C++ или `cv2.Sobel(I, ddepth, dx, dy)` в Python.

Фильтр Лапласа

Фильтр Лапласа использует аппроксимацию вторых производных по осям Ox и Oy, в отличие от предыдущих подходов, использующих первую производную:

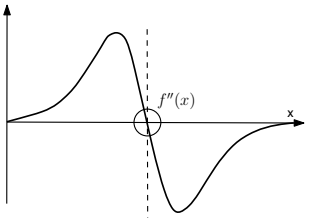


Рис. 3.7 — Вторая производная функции яркости меняет знак (проходит через ноль в точке, соответствующей краю).

$$L(I(x,y)) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}. \tag{3.20}$$

Формула 3.20 может быть аппроксимирована следующей маской:

$$w = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}. \tag{3.21}$$

Алгоритм Кэнни

Одним из самых распространенных и эффективных алгоритмов выделения контуров на изображении является *алгоритм Кэнни*. Данный алгоритм позволяет не только определять краевые пиксели, но и связные граничные линии. Алгоритм состоит из следующих шагов:

1. Устранение мелких деталей путем сглаживания исходного изображения с помощью фильтра Гаусса.

2. Использование дифференциального оператора Собела для определения значений модуля градиента всех пикселей изображения, причем результат вычисления округляется с шагом $45°$.

3. Анализ значений модулей градиента пикселей, расположенных ортогонально исследуемому. Если значение модуля градиента исследуемого пикселя больше, чем у ортогональных соседних пикселей, то он является *краевым*, в противном случае — *немаксимумом*.

4. Выполнение двойной пороговой фильтрации краевых пикселей, отобранных на предыдущем шаге:

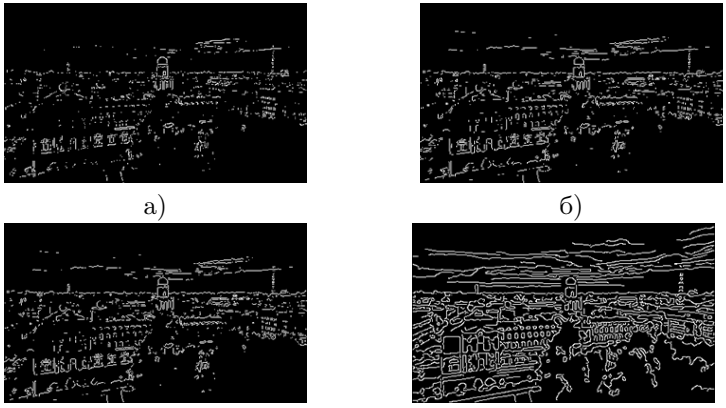
   (a) Если значение модуля градиента выше порога t_2, то наличие края в пикселе является достоверным.

   (b) Если значение модуля градиента ниже порога t_1, то пиксель однозначно не является краевым.

   (c) Если значение модуля градиента лежит в интервале $[t_1, t_2]$, то такой пиксель считается *неоднозначным*.

5. Подавление всех неоднозначных пикселей, не связанных с достоверными пикселями по 8-связности.

В MATLAB алгоритмом Кэнни можно выделить границы при помощи функции `edge(I, 'Canny')`. В библиотеке OpenCV алгоритм Canny реализуется функцией `cv::Canny(I, I_out, t1, t2)` в C++ и `cv2.Canny(I, t1, t2)` в Python.



а) б) в) г)

Рис. 3.8 — Результат выделения границ: а) фильтром Робертса, б) фильтром Превитта, в) фильтром Собела, г) алгоритмом Кэнни.

Порядок выполнения работы

1. *Типы шумов.* Выбрать произвольное изображение. Получить искаженные различными шумами изображения с помощью функции `imnoise()` с отличными от значений по умолчанию параметрами.

2. *Низкочастотная фильтрация.* Обработать полученные в предыдущем пункте искаженные изображения фильтром Гаусса и контргармоническим усредняющим фильтром с различными значениями параметра Q.

3. *Нелинейная фильтрация.* Обработать полученные в первом пункте искаженные изображения медианной, взвешенной медианной, ранговой и винеровской фильтрациями при различных размерах маски и ее коэффициентов. Реализовать адаптивную медианную фильтрацию.

4. *Высокочастотная фильтрация.* Выбрать исходное изображение. Выделить границы фильтрами Робертса, Превитта, Собела, Лапласа, алгоритмом Кэнни.

Содержание отчета

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

Вопросы к защите лабораторной работы

1. В чем заключаются основные недостатки адаптивных методов фильтрации изображений?
2. При каких значениях параметра Q контргармонический фильтр будет работать как арифметический, а при каких — как гармонический?
3. Какими операторами можно выделить границы на изображении?
4. Для чего на первом шаге выделения контуров, как правило, выполняется низкочастотная фильтрация?

Приложение 3.1. Реализация наложения шумов на изображение с использованием библиотеки OpenCV

Наиболее эффективным способом наложения шумов при использовании языка программирования C++ будет попиксельная обработка изображения и вычисление шума методами, описанными в параграфе «Теоретические сведения». С другой стороны, при наличии дополнительных ресурсов ОЗУ можно реализовать наложение шумов с помощью матричных операций OpenCV.

Листинг 3.1. Наложение импульсного шума на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Noise parameter
2   double d = 0.05;
3   // Salat vs pepper distribution
4   double s_vs_p = 0.5;
5   // Split an image into layers
6   vector<Mat> I_out_bgr;
7   split(I, I_out_bgr);
8   // Process layers
9   for (int i = 0; i < I_out_bgr.size(); i++)
10  {
11    Mat vals(I_out_bgr[i].size(), CV_32F);
12    randu(vals, Scalar(0), Scalar(1));
13    if (I_out_bgr[i].depth() == CV_8U)
14      I_out_bgr[i].setTo(Scalar(255),
15        vals < d * s_vs_p);
16    else
17      I_out_bgr[i].setTo(Scalar(1),
18        vals < d * s_vs_p);
19    I_out_bgr[i].setTo(Scalar(0),
20      (vals >= d * s_vs_p) & (vals < d));
21  }
22  // Merge layers to create an output image
23  merge(I_out_bgr, I_out);
```

При наложении мультипликативного шума использование целочисленного представления цвета может привести к потере точности, поэтому входное изображение необходимо сконвертировать в представление с использованием вещественных чисел.

Листинг 3.2. Наложение мультипликативного шума на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Variance parameter
2   double var = 0.05;
3   // Split an image into layers
4   vector<Mat> I_out_bgr;
5   split(I, I_out_bgr);
6   // Process layers
7   for (int i = 0; i < I_out_bgr.size(); i++)
8   {
9     Mat gauss(I_out_bgr[i].size(), CV_32F);
10    randn(gauss, Scalar(0),
11          Scalar(sqrt(var)));
12    if (I_out_bgr[i].depth() == CV_8U)
13    {
14      Mat I_out_bgr_f;
15      I_out_bgr[i].convertTo(I_out_bgr_f,
16        CV_32F);
17      I_out_bgr_f += out_bgr_f.mul(gauss);
18      I_out_bgr_f.convertTo(I_out_bgr[i],
19        I_out_bgr[i].type());
20    }
21    else
22      I_out_bgr[i] += I_out_bgr[i].mul(gauss);
23  }
24  // Merge layers to create an output image
25  merge(I_out_bgr, I_out);
```

Листинг 3.3. Наложение гауссова шума на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Mean parameter
2   double mean = 0;
```

```
3   // Variance parameter
4   double var = 0.05;
5   // Split an image into layers
6   vector<Mat> I_out_bgr;
7   split(I, I_out_bgr);
8   // Process layers
9   for (int i = 0; i < I_out_bgr.size(); i++)
10  {
11    Mat gauss(I_out_bgr[i].size(), CV_32F);
12    randn(gauss, Scalar(mean),
13          Scalar(sqrt(var)));
14    if (I_out_bgr[i].depth() == CV_8U)
15    {
16      Mat I_out_bgr_f;
17      I_out_bgr[i].convertTo(I_out_bgr_f,
18        CV_32F);
19      I_out_bgr_f += gauss * 255;
20      I_out_bgr_f.convertTo(I_out_bgr[i],
21        I_out_bgr[i].type());
22    }
23    else
24      I_out_bgr[i] += gauss;
25  }
26  // Merge layers to create an output image
27  merge(I_out_bgr, I_out);
```

Поскольку в библиотеке OpenCV отсутствуют функции для генерации матрицы случайных чисел Пуассона, то для генерации шума квантования изображение должно быть обработано попиксельно.

Листинг 3.4. Наложение шума квантования на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Convert image to floats
2   if (I.depth() == CV_8U)
3     I.convertTo(out, CV_32F, 1.0 / 255);
4   else
5     out = img.clone();
```

```
6   // Calculate quantization parameter
7   size_t vals = unique(out).size();
8   vals = (size_t)pow(2, ceil(log2(vals)));
9   int rows = out.rows;
10  int cols = out.cols * out.channels();
11  // If the image is continuous then
12  // we can process it as a single row
13  if (out.isContinuous())
14  {
15    cols *= rows;
16    rows = 1;
17  }
18  // Create poisson generator
19  using param_t =
20    std::poisson_distribution<int>::
21    param_type;
22  std::default_random_engine engine;
23  std::poisson_distribution<> poisson;
24  // Process image pixel-by-pixel
25  for (int i = 0; i < rows; i++)
26  {
27    float *ptr = out.ptr<float>(i);
28    for (int j = 0; j < cols; j++)
29      ptr[j] = float(poisson(engine,
30        param_t({ ptr[j] * vals }))) / vals;
31  }
32  // Convert back to uchar if needed
33  if (img.depth() == CV_8U)
34    out.convertTo(out, CV_8U, 255);
```

В представленной реализации функции наложения шума квантования используется функция `unique`, которая формирует массив из всех уникальных цветов пикселей изображения. Эта функция отсутствует в библиотеке OpenCV, но она может быть легко реализована.

Листинг 3.5. Поиск уникальных значений элементов матрицы с использованием библиотеки OpenCV и языка программирования C++.

```
1  vector<float> unique(const cv::Mat& I,
2    bool sort = false)
3  {
4    if (I.depth() != CV_32F)
5    {
6    std::cerr <<
7      "unique()␣Only␣works␣with␣CV_32F␣Mat" <<
8      std::endl;
9    return std::vector<float>();
10   }
11 std::vector<float> out;
12 // Acquire matrix size
13 int rows = img.rows;
14 int cols = img.cols * img.channels();
15 if (img.isContinuous())
16 {
17   cols *= rows;
18   rows = 1;
19 }
20 // Process each pixel
21 for (int y = 0; y < rows; y++)
22 {
23   const float* row_ptr = img.ptr<float>(y);
24   for (int x = 0; x < cols; x++)
25   {
26     float value = row_ptr[x];
27     if (std::find(out.begin(), out.end(),
28         value) == out.end())
29       out.push_back(value);
30   }
31 }
32 // Sort if needed
33 if (sort)
34   std::sort(out.begin(), out.end());
35 return out;
36 }
```

Реализация функций наложения шума на языке Python заметно проще, так как все необходимые дополнительные функции уже присутствуют в библиотеке NumPy.

Листинг 3.6. Наложение импульсного шума на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
# Noise parameter
d = 0.05
# Salat vs pepper distribution
s_vs_p = 0.5
# Generate radnom numbers
rng = np.random.default_rng()
vals = rng.random(I.shape)
# Salt
I_out = np.copy(I)
if out.dtype == np.uint8:
  I_out[vals < d * s_vs_p] = 255
else:
  I_out[vals < d * s_vs_p] = 1.0
# Pepper
I_out[np.logical_and(vals >= d * s_vs_p,
      vals < d)] = 0
```

Листинг 3.7. Наложение мультипликативного шума на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
# Variance parameter
var = 0.05
# Generate radnom numbers
rng = np.random.default_rng()
gauss = rng.normal(0, var ** 0.5, I.shape)
# Process uchar and float images separately
if I.dtype == np.uint8:
  I_f = I.astype(np.float32)
  I_out = (I_f + I_f * gauss). \
        clip(0, 255).astype(np.uint8)
else:
  I_out = I + I * gauss
```

Листинг 3.8. Наложение гауссова шума на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
# Mean parameter
mean = 0
# Variance parameter
var = 0.01
# Generate radnom numbers
rng = np.random.default_rng()
gauss = \
  rng.normal(mean, var ** 0.5, I.shape)
gauss = gauss.reshape(I.shape)
# Process uchar and float images separately
if I.dtype == np.uint8:
  I_out = (I.astype(np.float32) +
    gauss * 255).clip(0, 255). \
    astype(np.uint8)
else:
  I_out = (I + gauss).astype(np.float32)
```

Листинг 3.9. Наложение шума квантования на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
rng = np.random.default_rng()
if I.dtype == np.uint8:
  I_f = I.astype(np.float32) / 255
  vals = len(np.unique(I_f))
  vals = 2 ** np.ceil(np.log2(vals))
  I_out = (255 * \
    (rng.poisson(I_f * vals) / \
    float(vals)).clip(0, 1)). \
    astype(np.uint8)
else:
  vals = len(np.unique(I))
  vals = 2 ** np.ceil(np.log2(vals))
  I_out = \
    rng.poisson(I * vals) / float(vals)
```

Приложение 3.2. Реализация взвешенной ранговой фильтрации изображений с использованием библиотеки OpenCV

Реализации взвешенного медианного и рангового фильтров отличаются только одной строкой, то есть выбором элемента в отфильтрованном массиве на последнем шаге фильтрации. По этой причине их реализация может быть объединена в одну функцию следующим образом:

Листинг 3.10. Взвешенная ранговая фильтрация изображения с использованием библиотеки OpenCV и языка программирования C++. .

```
1   // Filter parameters
2   int k_size[] = { 3, 3 };
3   Mat kernel =
4     Mat::ones(k_size[0], k_size[1], CV_64F);
5   int rank = 4;
6   // Convert to float
7   // and make image with border
8   Mat I_copy;
9   if (I.depth() == CV_8U)
10    I.convertTo(I_copy, CV_32F, 1.0 / 255);
11  else
12    I_copy = I;
13  cv::copyMakeBorder(I_copy, I_copy,
14    int((k_size[0] - 1) / 2),
15    int(k_size[0] / 2),
16    int((k_size[1] - 1) / 2),
17    int(k_size[1] / 2), cv::BORDER_REPLICATE);
18  // Split into layers
19  vector<Mat> bgr_planes;
20  cv::split(I_copy, bgr_planes);
21  // Process all layers
22  for (int k = 0; k < bgr_planes.size(); k++)
23  {
24    Mat I_tmp = Mat::zeros(I.size(),
25      bgr_planes[k].type());
26    // Allocate memory for arrays
```

```
27      vector<double> c;
28      c.reserve(k_size[0] * k_size[1]);
29      // For each image pixel
30      for (int i = 0; i < I.rows; i++)
31        for (int j = 0; j < I.cols; j++)
32        {
33          // Empty array
34          c.clear();
35          // Fill array
36          for (int a = 0; a < k_size[0]; a++)
37            for (int b = 0; b < k_size[1]; b++)
38              c.push_back(bgr_planes[k].
39                at<float>(i + a, j + b) *
40                kernel.at<double>(a, b));
41          // Sort array
42          std::sort(c.begin(), c.end());
43          // Select id with given rank
44          I_tmp.at<float>(i, j) =
45            float(c[rank]);
46        }
47      bgr_planes[k] = I_tmp;
48    }
49    // Merge back
50    cv::merge(bgr_planes, I_out);
51    // Convert back to uint if needed
52    if (I.depth() == CV_8U)
53      I_out.convertTo(I_out, CV_8U, 255);
```

При реализации взвешенного рангового фильтра на языке программирования Python можно значительно ускорить выполнение кода за счет использования операций с массивами NumPy, однако для этого решения потребуется хранить несколько экземпляров исходного изображения, по одному для каждого элемента окна фильтра.

Листинг 3.11. Взвешенная ранговая фильтрация изображения с использованием библиотеки OpenCV и языка программирования Pyhton.

```
1   # Filter parameters
```

```
2   k_size = (3, 3)
3   rank = 4
4   kernel = np.ones(k_size, dtype = np.float32)
5   rows, cols = I.shape[0:2]
6   # Convert to float
7   # and make image with border
8   if I.dtype == np.uint8:
9     I_copy = I.astype(np.float32) / 255
10  else:
11    I_copy = I
12  I_copy = cv.copyMakeBorder(I_copy,
13    int((k_size[0] - 1) / 2),
14    int(k_size[0] / 2),
15    int((k_size[1] - 1) / 2),
16    int(k_size[1] / 2), cv.BORDER_REPLICATE)
17  # Fill arrays for each kernel item
18  I_layers = np.zeros(I.shape +
19    (k_size[0] * k_size[1], ),
20    dtype = np.float32)
21  if I.ndim == 2:
22    for i in range(k_size[0]):
23      for j in range(k_size[1]):
24        I_layers[:, :, i * k_size[1] + j] = \
25          kernel[i, j] * \
26          I_copy[i:i + rows, j:j + cols]
27  else:
28    for i in range(k_size[0]):
29      for j in range(k_size[1]):
30        I_layers[:, :, :, i * k_size[1] + \
31          j] = kernel[i, j] * \
32          I_copy[i:i + rows, j:j + cols, :]
33  # Sort arrays
34  I_layers.sort()
35  # Choose layer with rank
36  if I.ndim == 2:
37    I_out = I_layers[:, :, rank]
38  else:
39    I_out = I_layers[:, :, :, rank]
```

```
40  # Convert back to uint if needed
41  if (I.dtype == np.uint8):
42    I_out = (255 * I_out).clip(0, 255). \
43      astype(np.uint8)
```

Приложение 3.3. Реализация фильтра Винера с использованием библиотеки OpenCV

Реализаций фильтра Винера требует двухпроходной обработки входного изображения: первый проход для оценки значения ошибки изображения, а второй проход непосредственно для фильтрации изображения.

Листинг 3.12. Винеровская фильтрация изображения с использованием библиотеки OpenCV и языка программирования C++. .

```
1   // Define parameters
2   int k_size[] = { 5, 5 };
3   Mat kernel = Mat::ones(k_size[0],
4     k_size[1], CV_64F);
5   double k_sum = cv::sum(kernel)[0];
6   // Convert to float
7   // and make image with border
8   Mat I_copy;
9   if (I.depth() == CV_8U)
10    I.convertTo(I_copy, CV_32F, 1.0 / 255);
11  else
12    I_copy = I;
13  cv::copyMakeBorder(I_copy, I_copy,
14    int((k_size[0] - 1) / 2),
15    int(k_size[0] / 2),
16    int((k_size[1] - 1) / 2),
17    int(k_size[1] / 2),
18    cv::BORDER_REPLICATE);
19  // Split into layers
20  vector<Mat> bgr_planes;
21  cv::split(I_copy, bgr_planes);
22  // Process all layers
```

```
23  for (int k = 0; k < bgr_planes.size(); k++)
24  {
25    Mat I_tmp = Mat::zeros(I.size(),
26      bgr_planes[k].type());
27    double v(0);
28    // Calculate the average of all
29    // the local estimated variances
30    for (int i = 0; i < I.rows; i++)
31      for (int j = 0; j < I.cols; j++)
32      {
33        // Calculate variance
34        double m(0), q(0);
35        for (int a = 0; a < k_size[0]; a++)
36          for (int b = 0; b < k_size[1]; b++)
37          {
38            double t = bgr_planes[k].
39              at<float>(i + a, j + b) *
40              kernel.at<double>(a, b);
41            m += t;
42            q += t * t;
43          }
44        m /= k_sum;
45        q /= k_sum;
46        q -= m * m;
47        v += q;
48      }
49    v /= I.cols * I.rows;
50    // For each image pixel
51    for (int i = 0; i < I.rows; i++)
52      for (int j = 0; j < I.cols; j++)
53      {
54        // Calculate variance
55        double m(0), q(0);
56        for (int a = 0; a < k_size[0]; a++)
57          for (int b = 0; b < k_size[1]; b++)
58          {
59            double t = bgr_planes[k].
60              at<float>(i + a, j + b) *
```

```
61                kernel.at<double>(a, b);
62              m += t;
63              q += t * t;
64            }
65          m /= k_sum;
66          q /= k_sum;
67          q -= m * m;
68          // Calculate pixel value
69          double im = bgr_planes[k].
70            at<float>(i + (k_size[0] - 1) / 2,
71            j + (k_size[1] - 1) / 2);
72          if (q < v)
73            I_tmp.at<float>(i, j) = float(m);
74          else
75            I_tmp.at<float>(i, j) =
76              float((im - m) * (1 - v / q) + m);
77      }
78      bgr_planes[k] = I_tmp;
79    }
80    // Merge back
81    cv::merge(bgr_planes, I_out);
82    // Convert back to uint if needed
83    if (I.depth() == CV_8U)
84      I_out.convertTo(I_out, CV_8U, 255);
```

Листинг 3.13. Винеровская фильтрация изображения с использованием библиотеки OpenCV и языка программирования Python.

```
1   # Define parameters
2   k_size = (7, 7)
3   kernel = np.ones((k_size[0], k_size[1]))
4   # Convert to float
5   # and make image with border
6   if I.dtype == np.uint8:
7     img_copy = I.astype(np.float32) / 255
8   else:
9     img_copy_nb = I
10  img_copy = cv.copyMakeBorder(img_copy,
```

```
    int((k_size[0] - 1) / 2),
    int(k_size[0] / 2),
    int((k_size[1] - 1) / 2),
    int(k_size[1] / 2),
    cv.BORDER_REPLICATE)
  # Split into layers
  bgr_planes = cv.split(img_copy)
  bgr_planes_2 = []
  k_power = np.power(kernel, 2)
  # For all layers
  for plane in bgr_planes:
    # Calculate temporary matrices for I ** 2
    plane_power = np.power(plane, 2)
    m = np.zeros(I.shape[0:2], np.float32)
    q = np.zeros(I.shape[0:2], np.float32)
    # Calculate variance values
    for i in range(k_size[0]):
      for j in range(k_size[1]):
        m = m + kernel[i, j] * \
          plane[i:i + rows, j:j + cols]
        q = q + k_power[i, j] * \
          plane_power[i:i + rows, j:j + cols]
    m = m / np.sum(kernel)
    q = q / np.sum(kernel)
    q = q - m * m
    # Calculate noise as an average variance
    v = np.sum(q) / I.size
    # Do filter
    plane_2 = plane[(k_size[0] - 1) // 2: \
      (k_size[0] - 1) // 2 + rows, \
      (k_size[1] - 1) // 2: \
      (k_size[1] - 1) // 2 + cols]
    plane_2 = np.where(q < v, m,
      (plane_2 - m) * (1 - v / q) + m)
    bgr_planes_2.append(plane_2)
  # Merge image back
  I_out = cv.merge(bgr_planes_2)
  # Convert back to uint if needed
```

```
49   if (I.dtype == np.uint8):
50     I_out = (255 * I_out).clip(0, 255). \
51       astype(np.uint8)
```

Лабораторная работа №4
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:

$$I_{new}(x,y) = \begin{cases} 0, I(x,y) \leqslant t, \\ 1, I(x,y) > t, \end{cases} \tag{4.1}$$

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 4.1. Бинаризация.

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

Бинаризация по двойному порогу (диапазонная бинаризация):

$$I_{new}(x,y) = \begin{cases} 0, I(x,y) \leqslant t_1, \\ 1, t_1 < I(x,y) \leqslant t_2, \\ 0, I(x,y) > t_2, \end{cases} \tag{4.2}$$

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 использование функции `roicolor()`. Для преобразования полноцветного изображения в полутоновое можно предварительно воспользоваться функцией `rgb2gray()`.

Листинг 4.2. Бинаризация по двойному порогу.

```
I = imread('pic.jpg');
t1 = 110;
t2 = 200;
Igray = rgb2gray(I);
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 полутонового изображения и нахождение их среднего арифметического. Среднее арифметическое будет являться глобальным порогом бинаризации t:

$$t = \frac{I_{max} - I_{min}}{2}. \tag{4.3}$$

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

$$G(x,y) = \max\left\{|I(x+1,y) - I(x-1,y)|, |I(x,y+1) - I(x,y-1)|\right\}, \tag{4.4}$$

затем вычисляется оптимальный порог t:

$$t = \frac{\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} I(x,y)G(x,y)}{\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} G(x,y)}. \tag{4.5}$$

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 $p_i = \dfrac{n_i}{N}$ для каждого уровня интенсивности, где n_i — число пикселей с уровнем интенсивности i, N — число пискселей 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 $j = \overline{1,2}$:

$$\omega_1(k) = \sum_{s=0}^{k} p_s, \tag{4.6}$$

$$\omega_2(k) = \sum_{s=k+1}^{L} p_s = 1 - \omega_1(k), \tag{4.7}$$

$$\mu_1(k) = \sum_{s=0}^{k} \frac{s \cdot p_s}{\omega_1}, \tag{4.8}$$

$$\mu_2(k) = \sum_{s=k+1}^{L} \frac{s \cdot p_s}{\omega_2}. \tag{4.9}$$

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

$$\sigma_b^2(k) = \omega_1(k)\omega_2(k)(\mu_1(k) - \mu_2(k))^2. \tag{4.10}$$

   (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 Отсу может быть вычислен с использованием функции `graythresh()`:

Листинг 4.3. Бинаризация методом Отсу.

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

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

Листинг 4.4. Бинаризация методом Отсу на основе гистограммы.

```
I = imread('pic.jpg');
Igray = rgb2gray(I);
[counts,x] = imhist(Igray);
t = otsuthresh(counts);
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 4.5. Бинаризация адаптивным методом.

```
I = imread('pic.jpg');
Igray = rgb2gray(I);
t = adaptthresh(Igray);
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:

$$W(I) = \begin{cases} 20 - \dfrac{12I}{88}, 0 \leqslant I \leqslant 88, \\ 0{,}002(I - 88)^2, 88 < I \leqslant 138, \\ \dfrac{7(I - 138)}{117} + 13, 138 < I \leqslant 255. \end{cases} \tag{4.11}$$

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 приницпом 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:

$$\begin{cases} R > 95, \\ G > 40, \\ B < 20, \\ \max R,G,B - \min R,G,B > 15, \\ |R - G| > 15, \\ R > G, \\ R > B, \end{cases} \tag{4.12}$$

или

$$\begin{cases} R > 220, \\ G > 210, \\ B > 170, \\ |R - G| \leqslant 15, \\ G > B, \\ R > B, \end{cases} \tag{4.13}$$

или

$$\begin{cases} r = \frac{R}{R+G+B}, \\ g = \frac{G}{R+G+B}, \\ b = \frac{B}{R+G+B}, \\ \frac{r}{g} > 1{,}185, \\ \frac{rb}{(r+g+b)^2} > 0{,}107, \\ \frac{rg}{(r+g+b)^2} > 0{,}112. \end{cases} \tag{4.14}$$

На основе цветового пространства CIE Lab

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

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

В качестве исходных данных выберем следующее цветное изображение:

Рис. 4.1 — 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 4.6. Сегментация на основе цветового пространства **Lab**.

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

```
numColors = 3;
sampleAreas = false([size(I, 1)
    size(I, 2) numColors]);
for i=1:1:numColors
    [BW, xi, yi] = roipoly(I);
    sampleAreas(:,:,i) = roipoly(I, xi, yi);
end
```

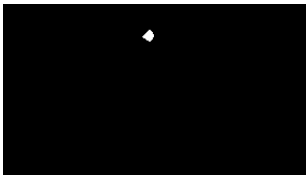


Рис. 4.2 — Пример выделенной красной области из четырех точек.

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()`:

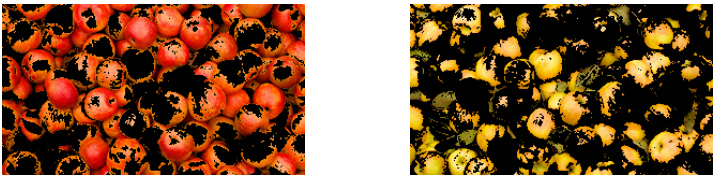
```
colorMarks = zeros([numColors, 2]);
for i=1:1:numColors
    colorMarks(i,1) = ...
        mean2(a(sampleAreas(:,:,i)));
    colorMarks(i,2) = ...
        mean2(b(sampleAreas(:,:,i)));
end
```

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

```
distance = zeros([size(a), numColors]);
for i=1:1:numColors
    distance(:,:,i) = ...
        ((a-colorMarks(i,1)).^2 + ...
        (b-colorMarks(i,2)).^2).^0.5;
    colorLabels = i;
end
[~, label] = min(distance, [], 3);
label = colorLabels(label);
```

Таким образом, матрица `label` рамерности равном 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
```



а) б)

Рис. 4.3 — Сегментированные области: а) красные, б) желтые.

Отметим распредление 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-

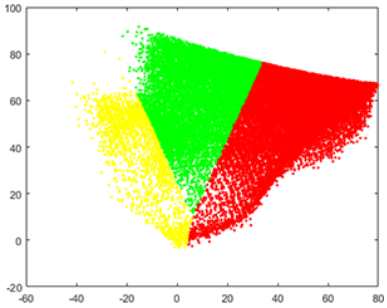


Рис. 4.4 — Распределение сегментированных пикселей на координатной плоскости (a,b).

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. 4.1). Будем работать в цветовом пространстве **Lab**, поэтому первым шагом перейдем из пространства **RGB** в **Lab**:

Листинг 4.7. Сегментация на основе кластеризации методом k-средних.

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

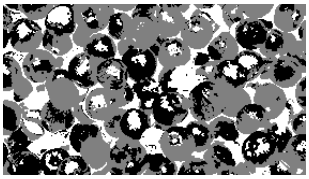
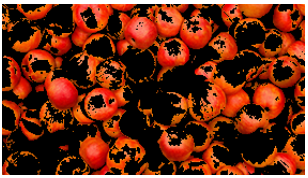


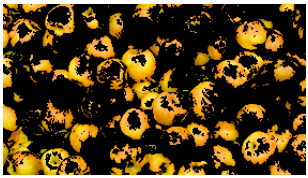
Рис. 4.5 — Метки классов.

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 rgbLabel = repmat(label, [1 1 3]);
17 for i = 1:1:k
18     color = I;
19     color(rgbLabel ~= i) = 0;
```



а)



б)

Рис. 4.6 — 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. 4.1. Рассмотрим пример изображения, представленного на рис. 4.7, на котором имеется два типа текстур. Их разделение в общем случае невозможно выполнить с использованием только лишь простой бинаризации.

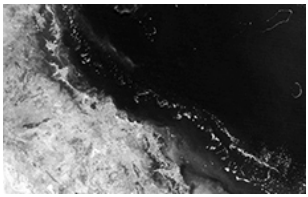


Рис. 4.7 — Исходное полутоновое изображение.

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:

$$\mu_n(z) = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i), \tag{4.15}$$

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

$$m = \sum_{i=0}^{L-1} z_i p(z_i). \tag{4.16}$$

Из 4.15 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:

$$R = 1 - \frac{1}{1 + \sigma^2(z)}, \tag{4.17}$$

которая равна нулю для областей с постоянной интенсивностью (нулевой дисперсией) и приближается к единице для больших значений дисперсий $\sigma^2(z)$. Для полутоновых 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*:

$$s = \sigma(z). \tag{4.18}$$

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

$$E = -\sum_{i=0}^{L-1} p(z) \log_2 p(z_i). \tag{4.19}$$

Таблица 4.1 — 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:

$$U = \sum_{i=0}^{L-1} p^2(z_i). \tag{4.20}$$

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 9×9. Для нормирования функции энтропии в диапазоне от 0 до 1 используем функцию `mat2gray()`, а для построения маски бинаризуем полученный нормированный массив `Eim` методом Отсу.

Листинг 4.8. Текстурная сегментация.

```
I = imread('pic3.jpg');
E = entropyfilt(I);
Eim = mat2gray(E);
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. 4.8 — 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 9×9). 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. 4.9 — а) Результат выполнения функции `bwareaopen()`;
б) результат выполнения функции `imclose()`.

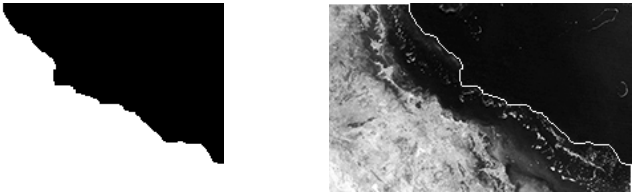
Применив полученную маску к исходному изображению выделим сегменты воды и суши.

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

а) б)

Рис. 4.10 — а) Текстура суши, б) текстура воды.

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



а) б)

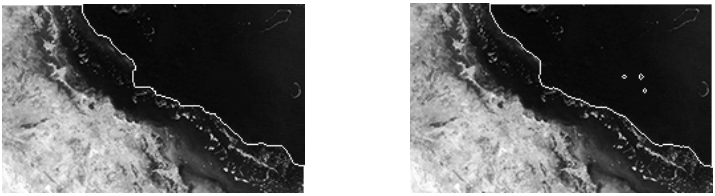
Рис. 4.11 — а) Результат выполнения функции `imfill()`, б) выделенная граница функцией `bwperim()`.

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 суши и воды:

```
texture1 = I;
texture1(~Mask2) = 0;
texture2 = I;
texture2(Mask2) = 0;
```



а) б)

Рис. 4.12 — а) Результат сегментации относительно воды, б) результат сегментации относительно суши.

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 $\boldsymbol{a}$ и $\boldsymbol{b}$ 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 №5
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. 5.2.

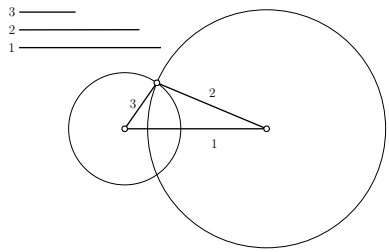


Рис. 5.1 — Построение треугольника по трем заданным сторонам.

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

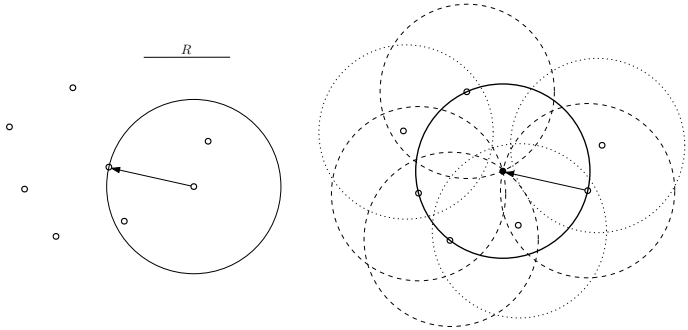


Рис. 5.2 — Обнаружение окружности известного радиуса в точечном множестве.

Классическое преобразование Хафа, базирующееся на рассмотренной идее 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:

$$y = kx + b, \tag{5.1}$$

$$x\cos\Theta + y\sin\Theta = \rho, \tag{5.2}$$

где ρ — радиус-вектор, проведенный из начала координат до прямой; Θ — угол наклона радиус-вектора.

Пусть в декартовой системе координат прямая задана уравнением (5.1), из которого легко вычислить радиус-вектор ρ и угол Θ (5.2). Тогда в пространстве параметров Хафа прямая будет представлена точкой с координатами (ρ_0,Θ_0), см. рис. 5.3.

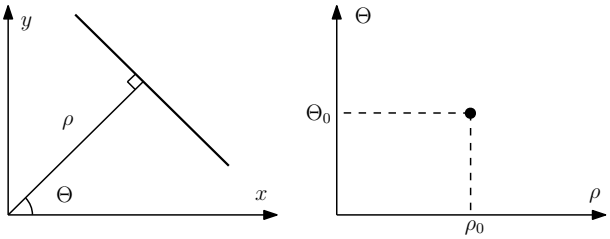


Рис. 5.3 — Представление прямой в пространстве Хафа.

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 (ρ,Θ) только в том случае, если порождающие их точки в исходном пространстве лежат на прямой, см. рис. 5.4. 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 $F = (a_1,a_2,...,a_n,x,y)$, что повлияет лишь на размерность пространства параметров. Воспользуемся преобразованием

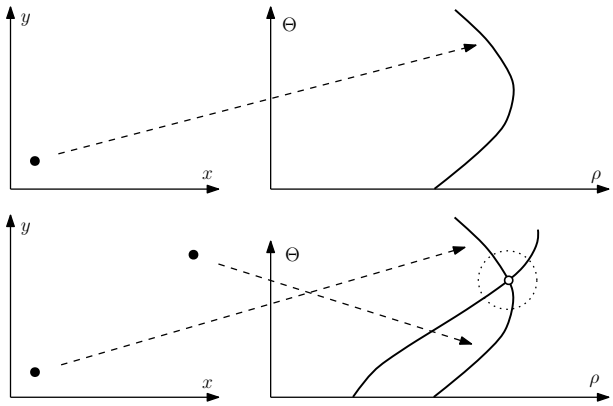


Рис. 5.4 — Процедура голосования.

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 парамаетров (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 5.1. Поиск прямых преобразованием Хафа.

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

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

```
peaks = houghpeaks(H,100,'threshold',...
    ceil(0.5*max(H(:))));
x = Theta(peaks(:,2));
y = rho(peaks(:,1));
plot(x,y,'s','color','white');
```

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

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

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

Порядок выполнения работы

1. *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-

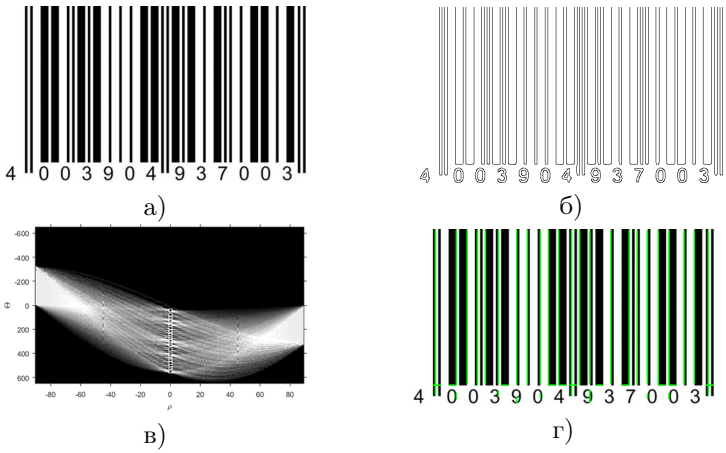


Рис. 5.5 — а) Исходное изображение, б) обработанное алгоритмом Кэнни, в) пространство параметров, г) выделенные линии.

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?

Лабораторная работа №6
Морфологический анализ изображений

Цель работы

Освоение принципов математической морфологии в области обработки и анализа изображений.

Методические рекомендации

До начала работы студенты должны ознакомиться с функциями среды MATLAB или библиотеки OpenCV для работы с бинарной морфологией и с основными операциями и положениями бинарной морфологии. Лабораторная работа рассчитана на 5 часов.

Теоретические сведения

Термин *морфология* дословно переводится как «наука о форме». Морфологический анализ используется во многих областях знаний, в т.ч. и в обработке изображений. Морфологический анализ является относительно универсальным подходом, поэтому стоит обособленно от всех рассмотренных ранее методов. С использованием математической морфологии при обработке изображений можно осуществлять фильтрацию шумов, сегментацию объектов, выделение контуров, реализовать поиск заданного объекта на изображении, вычислить «скелет» образа и т.д. Рассмотрим основные операции бинарной морфологии над изображением A структурным элементом B:

1. Дилатация (расширение, наращивание): $A \oplus B$, в MATLAB выполняется функцией `imdilate(A,B)`, расширяет бинарный образ A структурным элементом B;

2. Эрозия (сжатие, сужение): $A \ominus B$, в MATLAB выполняется функцией `imerode(A,B)`, сужает бинарный образ A структурным элементом B;

3. Открытие (отмыкание, размыкание, раскрытие): $(A \ominus B) \oplus B$, в MATLAB выполняется функцией `imopen(A,B)`, удаляет внешние дефекты бинарного образа A структурным элементом B;

4. Закрытие (замыкание): $(A \oplus B) \ominus B$, в MATLAB выполняется функцией `imclose(A,B)`, удаляет внутренние дефекты бинарного образа A структурным элементом B,

где $\oplus$ и $\ominus$ — сложение и вычитание Минковского, соответственно.

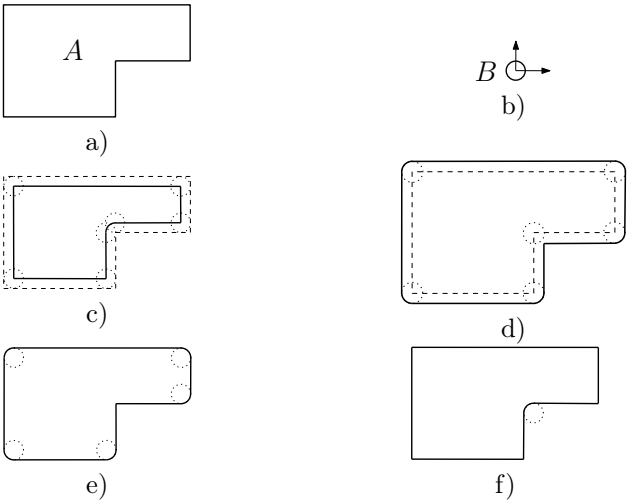


Рис. 6.1 — Результат выполения операций бинарной морфологии: а) бинарный образ A, б) дисковый структурный элемент B, в) операция дилатации, г) операция эрозии, д) операция открытия, е) операция закрытия.

В среде MATLAB имеются следующие полезные функции, часто используемые совместно с морфологическими операциями:

1. `strel()` — задает структурный элемент;

2. `bwmorph(A,operation,n)` — применяет морфологическую фильтрацию `operation` к образу `A` `n` раз;

3. `bwhitmiss(A,B,C)` — выполняет операцию «hit & miss», определяющую пиксели, окрестности которых совпадают по форме со структурным элементом `B` и не совпадают со структурным элементом `C`;

4. `bwareaopen(A,dim)` — удаляет объекты на изображении `A`, содержащие менее `dim` пикселей;

5. `bwselect()` — выбирает определенные объекты на изображении;

6. `bwarea()` — вычисляет площадь объектов;

7. `bwlabel()` — выполняет маркировку связных объектов бинарного изорбражения;

8. `bweuler()` — вычисляет число Эйлера бинарного изображения.

В библиотеке OpenCV имеются следующие функции, предназначенные для работы с морфологией и морфологическими операциями (перечисленные далее имена функций представлены для языков программирования C++ и Python соответственно):

1. `cv::getStructuringElement(shape, size, anchor)`, `cv2.getStructuringElement(shape, size, anchor)` — задает структурный элемент заданной формы *shape* и размера *size*. Если параметр центральной точки *anchor* не задан, то центром структурного элемента становится его геометрический центр. Форма структурного элемента может быть одной из следующих:

   (a) `cv::MORPH_RECT`, `cv2.MORPH_RECT` — структурный элемент прямоугольной формы размера *size*.

   (b) `cv::MORPH_CROSS`, `cv2.MORPH_CROSS` — структурный элемент в форме креста размера *size*.

   (c) `cv::MORPH_ELLIPSE`, `cv2.MORPH_ELLIPSE` — структурный элемент эллиптической формы, вписанный в прямоугольник размера *size*.

2. `cv::morphologyEx(A, C, op, B, anchor, iters, borderType, borderValue)`, `cv::morphologyEx(A, op, B, anchor, iters, borderType, borderValue)` — применяет морфологическую операцию *op* к изображению A используя структурный элемент B *iters* раз. Если параметр *iters* не задан, то морфологическая операция выполняется 1 раз. Также можно задать центральную точку структурного элемента *anchor* и метод обработки граничных точек изображения. В реализации на языке C++ результирующее изображение C передается вторым аргументом функции, тогда как в реализации на Python оно является возвращаемым значением. Используя данную функцию, можно выполнить следующие типы морфологических операций:

   (a) `cv::MORPH_ERODE`, `cv2.MORPH_ERODE` — эрозия. Для выполнения эрозии можно также использовать функцию `cv::erode()` в C++ и `cv2.erode()` в Python.

   (b) `cv::MORPH_DILATE`, `cv2.MORPH_DILATE` — дилатация. Для выполнения дилатации можно также использовать функцию `cv::dilate()` в C++ и `cv2.dilate()` в Python.

   (c) `cv::MORPH_OPEN`, `cv2.MORPH_OPEN` — открытие. $open(A, B) = erode(dilate(A, B), B)$

   (d) `cv::MORPH_CLOSE`, `cv2.MORPH_CLOSE` — закрытие. $close(A, B) = erode(dilate(A, B), B)$.

   (e) `cv::MORPH_GRADIENT`, `cv2.MORPH_GRADIENT` — морфологический градиент. Результатом данной операции является разница между дилатацией и эрозией. $gradient(A, B) = dilate(A, B) - erode(A, B)$.

   (f) `cv::MORPH_TOPHAT`, `cv2.MORPH_TOPHAT` — операция «top hat». Результатом данной операции является разница между изображением и его открытием. $tophat(A, B) = A - open(A, B)$.

   (g) `cv::MORPH_BLACKHAT`, `cv2.MORPH_BLACKHAT` — операция «black hat». Результатом данной операции является разница между закрытием изображения и им самим. $blackhat(A, B) = close(A, B) - A$.

(h) `cv::MORPH_HITMISS`, `cv2.MORPH_HITMISS` — операция «hit & miss».

3. `cv::connectedComponents(A, labels, connectivity)`, `cv2.connectedComponents(A)`
— формирует список всех связных компонент изображения A в матрице *labels*, которая хранит индекс компоненты для каждого пикселя исходного изображения. Возвращает количество связных компонент изображения в C++ и кортеж объектов, хранящий количество связных компонент и матрицу индексов компонент *labels*, в Python.

4. `cv::connectedComponentsWithStats(A, labels, stats, centers)`,
`cv2.connectedComponentsWithStats(A)` — то же самое, что и *connectedComponents*(), но кроме индексов компонент она также считает статистику для каждой компоненты. Возвращает количество связных компонент изображения в C++ и кортеж объектов, хранящий количество связных компонент, матрицу индексов компонент *labels*, статистику *stats* и центры *centers*, в Python. Статистика *stats* представляет из себя двумерную матрицу, где первое измерение — индекс компоненты (*label*), а второе — индекс параметра. Тип данных матрицы — 32 битные целые числа со знаком (*int*32). Центры *centers* — двумерная матрица, где первое измерение — это индекс компоненты (*label*), а второе 0 для оси X и 1 для оси Y. Статистика включает в себя следующие данные:

   (a) `cv::CC_STAT_LEFT`, `cv2.CC_STAT_LEFT` — координата самого левого пикселя компоненты (X);

   (b) `cv::CC_STAT_TOP`, `cv2.CC_STAT_TOP` — координата самого верхнего пикселя компоненты (Y);

   (c) `cv::CC_STAT_WIDTH`, `cv2.CC_STAT_WIDTH` — горизонтальный размер компоненты (ширина);

   (d) `cv::CC_STAT_HEIGHT`, `cv2.CC_STAT_HEIGHT` — вертикальный размер компоненты (высота);

   (e) `cv::CC_STAT_AREA`, `cv2.CC_STAT_AREA` — размер компоненты в пикселях.

Функция `bwareaopen(A, dim)` из MATLAB реализуется с использованием функции `connectedComponentsWithStats()` библиотеки OpenCV. В Приложении 6.1 представлен пример ее программной реализации на языках программирования C++ и Python.

Примеры использования морфологических операций

Выделение границ объектов

Границы объектов могут быть выделены с использованием следующего подхода:

1. $C = A - (A \ominus B)$ — формирование внутреннего контура;
2. $C = (A \oplus B) - A$ — формирование внешнего контура.

Разделение «склеенных» объектов

Одним из примеров использования морфологических операций над изображением может быть задача разделения склеившихся на изображении объектов. Задача может быть решена с достаточной степенью точности при помощи последовательного выполнения нескольких раз фильтра сжатия, а затем максимально возможного расширения полученного результата. Пересечение исходного изображения с обработанным позволит разделить склеенные объекты.

Листинг 6.1. Разделение «склеенных» объектов в среде MATLAB.

```
I = imread('pic.jpg');
t = graythresh(I);
Inew = im2bw(I,t);
Inew = ~Inew;
BW2 = bwmorph(Inew,'erode',7);
BW2 = bwmorph(BW2,'thicken',Inf);
Inew = ~(Inew & BW2);
```

Листинг 6.2. Разделение «склеенных» объектов с использованием библиотеки OpenCV и языка программирования C++.

```
1  Mat I = imread("pic.jpg",
2    cv::IMREAD_GRAYSCALE);
3  Mat Inew;
4  cv::threshold(I, Inew, 160, 255,
5    cv::THRESH_BINARY_INV);
6  Mat B = cv::getStructuringElement(
7    cv::MORPH_ELLIPSE, cv::Size(5, 5));
8  // Erosion
9  Mat BW2;
10 cv::morphologyEx(Inew, BW2, cv::MORPH_ERODE,
11   B, cv::Point(-1, -1), 14,
12   cv::BORDER_CONSTANT, cv::Scalar(0));
13 // Dilation
14 Mat D, C, S;
15 Mat T = Mat::zeros(Inew.rows, Inew.cols,
16   Inew.type());
17 int pix_num = Inew.rows * Inew.cols;
18 while (cv::countNonZero(BW2) < pix_num)
19 {
20   cv::morphologyEx(BW2, D, cv::MORPH_DILATE,
21     B, cv::Point(-1, -1), 1,
22     cv::BORDER_CONSTANT, cv::Scalar(0));
23   cv::morphologyEx(D, C, cv::MORPH_CLOSE,
24     B, cv::Point(-1, -1), 1,
25     cv::BORDER_CONSTANT, cv::Scalar(0));
26   S = C - D;
27   cv::bitwise_or(S, T, T);
28   BW2 = D;
29 }
30 // Closing for borders
31 cv::morphologyEx(T, T, cv::MORPH_CLOSE, B,
32   cv::Point(-1, -1), 14,
33   cv::BORDER_CONSTANT, cv::Scalar(255));
34 // Remove borders from an image
35 cv::bitwise_and(~T, Inew, Inew);
```

Листинг 6.3. Разделение «склеенных» объектов с использованием библиотеки OpenCV и языка программирования Python.

```
I = cv2.imread("pic.jpg",
  cv2.IMREAD_GRAYSCALE);
ret, Inew = cv2.threshold(I, 160, 255,
  cv.THRESH_BINARY_INV)
B = cv2.getStructuringElement(
  cv2.MORPH_ELLIPSE, (5, 5))
# Erosion
BW2 = cv2.morphologyEx(Inew,
  cv2.MORPH_ERODE, B, iterations = 14,
  borderType = cv2.BORDER_CONSTANT,
  borderValue = (0))
# Dilation
T = np.zeros_like(Inew)
while cv2.countNonZero(BW2) < BW2.size:
  D = cv.dilate(BW2, B,
    borderType = cv2.BORDER_CONSTANT,
    borderValue = (0))
  C = cv.morphologyEx(D, cv2.MORPH_CLOSE, B,
    borderType = cv2.BORDER_CONSTANT,
    borderValue = (0))
  S = C - D
  T = cv.bitwise_or(S, T)
  BW2 = D
# Closing for borders
T = cv2.morphologyEx(T, cv2.MORPH_CLOSE, B,
  iterations = 14,
  borderType = cv2.BORDER_CONSTANT,
  borderValue = (255))
# Remove borders from an image
Inew = cv2.bitwise_and(~T, Inew)
```

Сегментация методом управляемого водораздела

Рассмотрим еще один из примеров применения математической морфологии к задаче сегментации изображения. В подходе сегментации по водоразделам изображение рассматривается как карта высот, на котором интенсивности пикселей описывают высоты относительно некоторого уровня. На такую «высотную местность» «льет

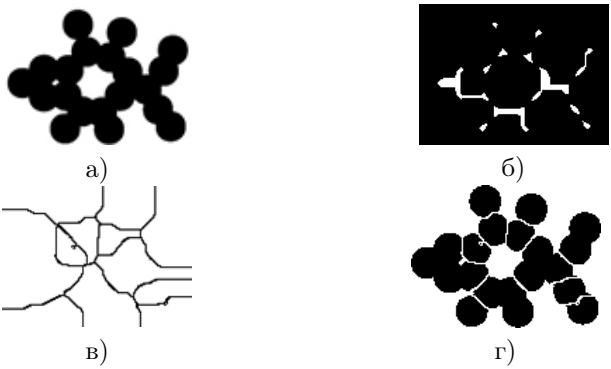


Рис. 6.2 — Разделение «склеенных» объектов: а) исходное изображение, б) эрозия бинарного изображения, в) расширение объектов, г) результат разделения.

дождь», образуя множество *водосборных бассейнов*. Постепенно вода из переполненных бассейнов переливается, и бассейны объединяются в более крупные (см. рис. 6.3). Места объединения бассейнов отмечаются как линии водораздела. Если «дождь» остановить рано, тогда изображение будет сегментировано на мелкие области, а если поздно — на крупные. В таком подходе все пиксели подразделяются на три типа:

1. *локальные минимумы*;
2. находящиеся *на склоне* (с которых вода скатывается в один и тот же локальный минимум);
3. *локальные максимумы* (с которых вода скатывается более чем в один минимум).

При реализации данного метода необходимо определить водосборные бассены и линии водораздела путем обработки локальных областей и вычисления их характеристик. Алгоритм сегментации состоит из следующих шагов:

1. Вычисление функции сегментации. Как правило, для этого используется градиентное представление изображения.

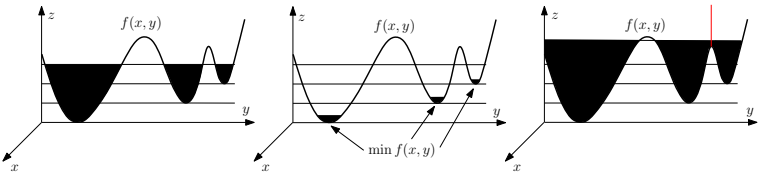


Рис. 6.3 — Водораздел областей.

2. Вычисление маркеров переднего плана на основании связности пикселей каждого объекта.

3. Вычисление маркеров фона, представляющих пиксели, не являющиеся объектами.

4. Модифицирование функции сегментации с учетом взаиморасположения маркеров переднего плана и фона.

В результате работы алгоритма будет получена маска, где пиксели одинаковых сегментов будут помечены одинаковыми метками и будут образовывать связную область. Предположим, задано изображение, представленное на рис. 6.4.

Рис. 6.4 — Исходное изображение.

Выполним морфологическую фильтрацию изображения с использованием базовых морфологических операций и морфологической реконструкции `imreconstruct()`. Функция `imcomplement()` вычисляет изображение-дополнение к изображению-аргументу и представляет собой инвертированное изображение.

Листинг 6.4. Сегментации методом водораздела в среде MATLAB.

```
1 rgb = imread('pic.jpg');
2 A = rgb2gray(rgb);
3 B = strel('disk',6);
4 C = imerode(A,B);
5 Cr = imreconstruct(C,A);
6 Crd = imdilate(Cr,B);
7 Crdr = imreconstruct(imcomplement(Crd), ...
8     imcomplement(Cr));
9 Crdr = imcomplement(Crdr);
```

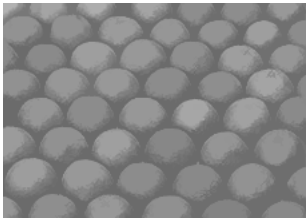


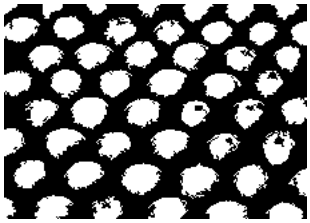
Рис. 6.5 — Отфильтрованное изображение.

После этого определим локальные максимумы функцией `imregionalmax()` для определения маркеров переднего плана (*foreground markers* или `fgm`) и, для наглядности, наложим маркеры на исходное изображение:

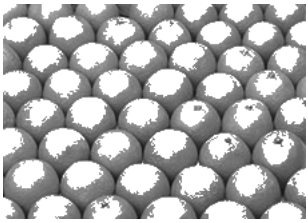
```
10 fgm = imregionalmax(Crdr);
11 A2 = A;
12 A2(fgm) = 255;
```

Как видно из представленных изображений, маркеры сильно изрезаны. Для сглаживания маркеров переднего плана воспользуемся следующей последовательностью морфологических операций:

```
13 B2 = strel(ones(5,5));
14 fgm = imclose(fgm,B2);
15 fgm = imerode(fgm,B2);
16 fgm = bwareaopen(fgm,20);
17 A3 = A;
18 A3(fgm) = 255;
```



а)



б)

Рис. 6.6 — Маркеры переднего плана: а) вычисленные, б) наложенные на исходное изображение.

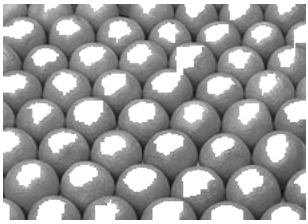
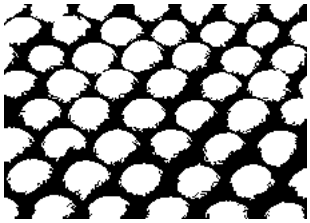


Рис. 6.7 — Отфильтрованные маркеры переднего плана.

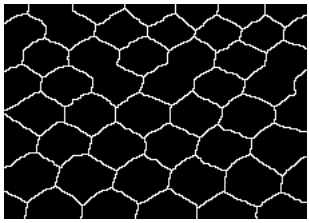
На следующем шаге требуется определить маркеры фона (*background markers* или `bgm`). Для этого бинаризуем отфильтрованное изображение `Crdr`, найдем евклидово расстояние от каждого черного до ближайшего белого пикселя функцией `bwdist()` и вычислим матрицу `L`, содержащую метки сегментов, полученных методом водораздела с использованием функции `watershed()`:

```
19 bw = imbinarize(Crdr);
20 D = bwdist(bw);
21 L = watershed(D);
22 bgm = L == 0;
```

Затем требуется модифицировать функцию сегментации. Для этого можно воспользоваться функцией `imimposemin()`, которая точно определяет локальные минимумы изображения. Благодаря этому функция корректирует значения градиентов на изображении и уточняет расположение маркеров переднего плана и фона. Перед



а)

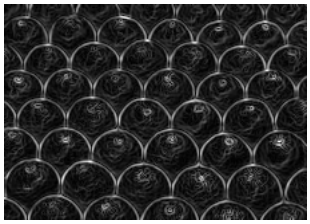


б)

Рис. 6.8 — а) бинаризованное изображение, б) маркеры фона.

этим нужно определить градиентное представление изображения, которое и будет скорректировано:

```
23 hy = fspecial('sobel');
24 hx = hy';
25 Ay = imfilter(double(A), hy, 'replicate');
26 Ax = imfilter(double(A), hx, 'replicate');
27 grad = sqrt(Ax.^2 + Ay.^2);
28 grad = imimposemin(grad, bgm | fgm);
```



а)

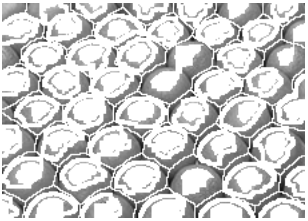
б)

Рис. 6.9 — Градиентное представление изображения: а) исходное, б) модифицированное.

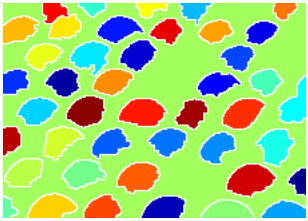
После этого выполняется операция сегментации уточненного градиентного представления изображения на основе водораздела и визуализируется результат работы алгоритма:

```
29 L = watershed(grad);
30 A4 = A;
```

```
31 A4(imdilate(L == 0, ...
32     ones(3,3)) | bgm | fgm) = 255;
33 Lrgb = label2rgb(L, 'jet', 'w', 'shuffle');
```



а)



б)

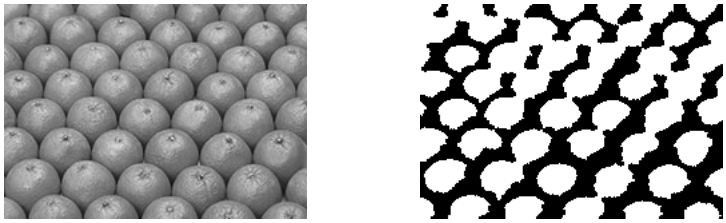
Рис. 6.10 — Маркеры и границы, наложенные на: а) исходное изображение, б) представленное в цветах rgb.

В наборе библиотеке OpenCV отсутствуют некоторые из функций MATLAB, которые использовались при реализации сегментации методом управляемого водораздела, однако можно реализовать аналогичный алгоритм. Прежде всего следует прочитать входное изображение и преобразовать его в черно-белое представление для поиска маркеров.

Листинг 6.5. Сегментация методом водораздела с использованием библиотеки OpenCV и языка программирования C++.

```
1   Mat I, I_gray, I_bw;
2   I = imread("pic.jpg", cv::IMREAD_COLOR);
3   cv::cvtColor(I, I_gray, cv::COLOR_BGR2GRAY);
4   cv::threshold(I_gray, I_bw, 0, 255,
5     cv::THRESH_BINARY + cv::THRESH_OTSU);
6   bwareaopen(I_bw, I_bw, 20, 4);
7   Mat B = cv::getStructuringElement(
8     cv::MORPH_ELLIPSE, cv::Size(5, 5));
9   cv::morphologyEx(I_bw, I_bw,
10    cv::MORPH_CLOSE, B);
```

Для определения маркеров переднего плана можно использовать преобразование евклидова расстояния, которое для каждого пикселя исходного изображения вычисляет расстояние от него



а) б)

Рис. 6.11 — а) полутоновое изображение, б) отфильтрованное черно-белое изображение.

до ближайшего черного пикселя. Это преобразование выполняется путем вызова функции `cv::distanceTransform()` библиотеки OpenCV:

```
11   Mat I_fg;
12   double I_fg_min, I_fg_max;
13   cv::distanceTransform(I_bw, I_fg,
14     cv::DIST_L2, 5);
```

Теперь становится возможно определить маркеры переднего плана как связные компоненты изображения, полученного дополнительной фильтрации, используя функцию `cv::connectedComponents()`:

```
15   cv::minMaxLoc(I_fg, &I_fg_min, &I_fg_max);
16   cv::threshold(I_fg, I_fg, 0.6 * I_fg_max,
17     255, 0);
18   img_fg.convertTo(img_fg, CV_8U,
19     255.0 / I_fg_max);
20   Mat markers;
21   int num = cv::connectedComponents(I_fg,
22     markers);
```

Область фонового маркера можно найти, выполнив алгоритм водораздела, используя полученные ранее маркеры переднего плана, и выбрав граничную область в качестве фона. Это решение хорошо работает в случае, если объекты расположены плотно, и область фона трудно выделить.

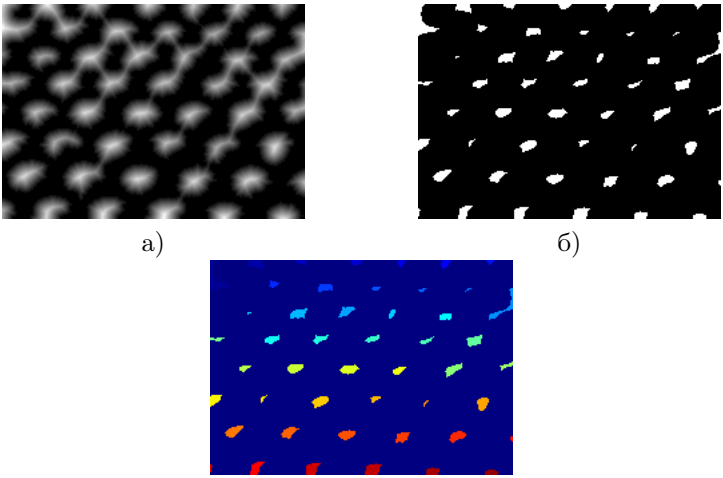


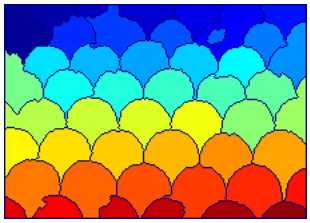
Рис. 6.12 — Определение маркеров переднего плана: а) преобразование евклидова расстояния, б) область маркеров переднего плана, в) маркеры переднего плана.

```
23   Mat I_bg = Mat::zeros(I_bw.rows, I_bw.cols,
24     I_bw.type());
25   Mat markers_bg = markers.clone();
26   cv::watershed(I, markers_bg);
27   I_bg.setTo(Scalar(255), markers_bg == -1);
```

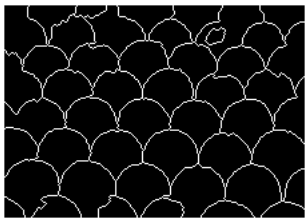
Далее необходимо определить «неопределенную» область, которая должна быть заполнена алгоритмом водораздела. Эту область можно найти путем вычитания области маркеров переднего плана из области, обратной области маркера заднего плана:

```
28   Mat I_unk;
29   cv::bitwise_not(I_bg, I_unk);
30   cv::subtract(I_unk, I_fg, I_unk);
```

Теперь получены все данные, необходимые для работы алгоритма водораздела. Осталось объединить все полученные маркеры в единую матрицу:



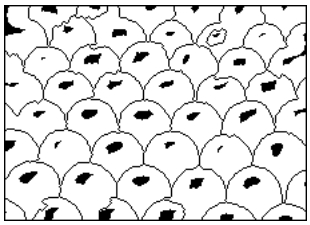
а)



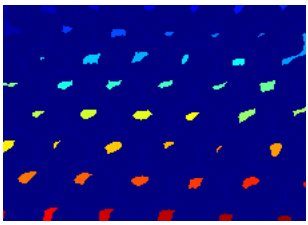
б)

Рис. 6.13 — Определение маркера фона: а) результат сегментации по маркерам переднего плана, б) маркер фона.

```
31    markers += 1;
32    markers.setTo(Scalar(0), I_unk == 255);
```



а)



б)

Рис. 6.14 — Маркеры: а) неопределенная область, б) объединение всех маркеров.

И запустить алгоритм, используя функцию `cv::watershed()`:

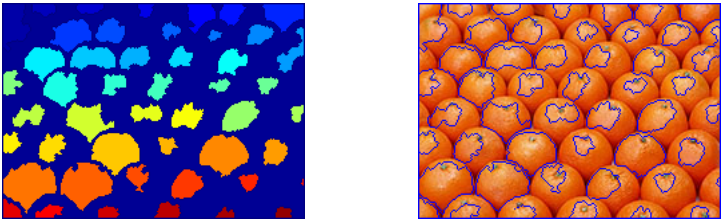
```
33    cv::watershed(I, markers);
```

Для визуализации результатов сегментации можно использовать цветовую карту JET:

```
34    Mat markers_jet;
35    markers.convertTo(markers_jet, CV_8U,
36      255.0 / (num + 1));
37    cv::applyColorMap(markers_jet,
38      markers_jet, cv::COLORMAP_JET);
```

И выделить сегментированные области на исходном изображении синим цветом.

```
39    I.setTo(Scalar(255, 0, 0), markers == -1);
```



а) б)

Рис. 6.15 — Маркеры и границы, наложенные на: а) изображение с использованием цветовой карты JET, б) исходное изображение.

В Приложении 6.2 представлена программная реализация данного метода на языке программирования Python.

Другим решением для определения маркера фона может быть последовательная дилатация отфильтрованного черно-белого изображения. Инверсия результата расширения должна дать нам область, которая обязательно будет являться фоном и может быть использована в качестве маркера фона. Это решение хорошо работает в случае, если область фона достаточно велика и не станет пустой после операции дилатации.

Листинг 6.6. Альтернативное решения для определения маркера фона с использование библиотеки OpenCV.

```
1    Mat I_bg;
2    cv::dilate(I_bw, I_bg, B,
3    cv::Point(-1, -1), 3);
4    cv::bitwise_not(I_bg, I_bg);
```

Порядок выполнения работы

1. *Базовые морфологические операции.* Выбрать произвольное изображение, содержащее дефекты формы (внутренние «дырки» или внешние «выступы») объектов. Используя базовые

морфологические операции, полностью убрать или минимизировать дефекты.

2. *Разделение объектов.* Выбрать произвольное бинарное изображение, содержащее перекрывающиеся объекты. Использовать операции бинарной морфологии для разделения объектов. Выделить контуры объектов.

3. *Сегментация.* Выбрать произвольное изображение, содержащее небольшое число локальных минимумов. Выполнить сегментацию изображения по водоразделам.

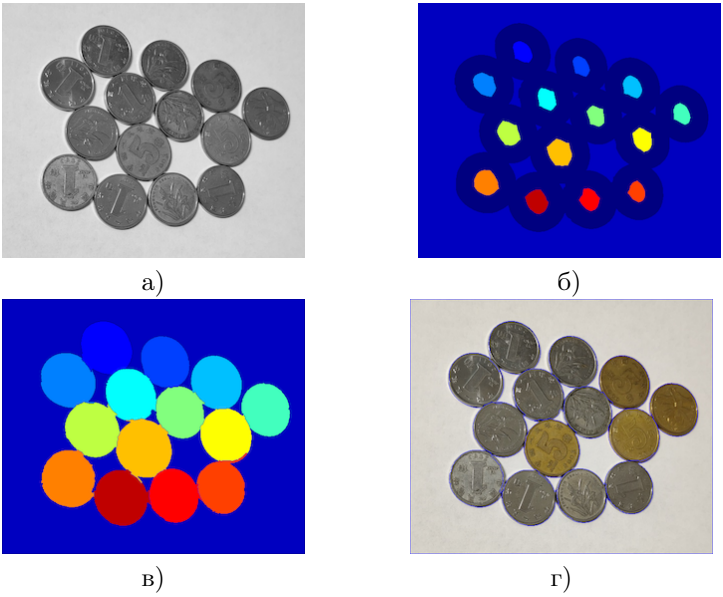


Рис. 6.16 — Использование альтернативного решения для определения маркера фона : а) черно-белое изображение, б) маркеры, в) результат сегментации, г) результат сегментации, наложенный на исходное изображение.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Теоретическое обоснование математической морфологии для анализа изображений.
4. Ход выполнения работы:
   (a) Исходные изображения;
   (b) Листинги программных реализаций;
   (c) Комментарии;
   (d) Результирующие изображения.
5. Выводы о проделанной работе.

Вопросы к защите лабораторной работы

1. Включает ли результат открытия в себя результат закрытия?
2. Какой морфологический фильтр необходимо применить, чтобы убрать у объекта выступы?
3. Каким образом с помощью морфологических операций можно найти контур объекта?
4. Что такое *морфология*?

Приложение 6.1. Реализация функции `bwareaopen()` из среды MATLAB средствами библиотеки OpenCV

Листинг 6.7. Реализация функции `bwareaopen()` из среды MATLAB с использованием библиотеки OpenCV и языка программирования C++.

```
1   void bwareaopen(const Mat &A, Mat &C,
2     int dim, int conn = 8)
3   {
4     if (A.channels() != 1 &&
5         A.type() != CV_8U &&
6         A.type() != CV_32F)
7       return;
8     // Find all connected components
9     Mat labels, stats, centers;
10    int num =
11      cv::connectedComponentsWithStats(A,
12      labels, stats, centers, conn);
13    // Clone image
14    C = A.clone();
15    // Check size of all connected components
16    vector<int> td;
17    for (int i = 0; i < num; i++)
18      if (stats.at<int>(i,
19          cv::CC_STAT_AREA) < dim)
20        td.push_back(i);
21    // Remove small areas
22    if (todel.size() > 0)
23      if (img.type() == CV_8U)
24      {
25        for (int i = 0; i < C.rows; i++)
26          for (int j = 0; j < C.cols; j++)
27            for (int k = 0; k < td.size();
28                 k++)
29              if (labels.at<int>(i, j) ==
30                  td[k])
31              {
32                C.at<uchar>(i, j) = 0;
33                continue;
34              }
35      }
36    else
37      {
38        for (int i = 0; i < C.rows; i++)
```

```
39            for (int j = 0; j < C.cols; j++)
40              for (int k = 0; k < td.size();
41                   k++)
42                if (labels.at<int>(i, j) ==
43                    td[k])
44                {
45                  C.at<float>(i, j) = 0;
46                  continue;
47                }
48        }
49    }
```

Листинг 6.8. Реализация функции `bwareaopen()` из среды MATLAB с использованием библиотеки OpenCV и языка программирования Python.

```
1   def bwareaopen(A, dim, conn = 8):
2     if img.ndim > 2:
3       return None
4     # Find all connected components
5     num, labels, stats, centers = \
6       cv2.connectedComponentsWithStats(A,
7       connectivity = conn)
8     # Check size of all connected components
9     for i in range(num):
10      if stats[i, cv2.CC_STAT_AREA] < dim:
11        A[labels == i] = 0
12    return A
```

Приложение 6.2. Сегментация методом управляемого водораздела средствами OpenCV

Листинг 6.9. Сегментация методом управляемого водораздела с использованием библиотеки OpenCV и языка программирования Python.

```
1   # Read an image
2   # Convert to grayscale and to BW
3   # Filter
```

```
4  I = cv2.imread("pic.jpg", cv.IMREAD_COLOR)
5  I_gray = cv.cvtColor(I, cv2.COLOR_BGR2GRAY)
6  ret, I_bw = cv.threshold(I_gray, 0, 255,
7    cv2.THRESH_BINARY + cv2.THRESH_OTSU)
8  I_bw = bwareaopen(I_bw, 20, 4)
9  B = cv2.getStructuringElement( \
10   cv2.MORPH_ELLIPSE, (5, 5))
11 I_bw = cv2.morphologyEx(I_bw, \
12   cv2.MORPH_CLOSE, B)
13
14 # Do distance transformation
15 # Find foreground location
16 # Define foreground markers
17 I_fg = cv2.distanceTransform(I_bw,
18   cv2.DIST_L2, 5)
19 ret, I_fg = cv.threshold(I_fg,
20   0.6 * I_fg.max(), 255, 0)
21 ret, markers = cv.connectedComponents(I_fg)
22
23 # Find background location
24 I_bg = np.zeros_like(I_bw)
25 markers_bg = markers.copy()
26 markers_bg = cv2.watershed(I, markers_bg)
27 I_bg[markers_bg == -1] = 255
28
29 # Define undefined area
30 I_unk = cv2.subtract(~I_bg, I_fg)
31 # Define all markers
32 markers = markers + 1
33 markers[I_unk == 255] = 0
34 # Do watershed
35 # Prepare for visualization
36 markers = cv2.watershed(I, markers)
37 markers_jet = cv2.applyColorMap(
38   (markers.astype(np.float32) * 255 /
39   (ret + 1)).astype(np.uint8),
40   cv2.COLORMAP_JET)
41 I[markers == -1] = (0, 0, 255)
```

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

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