Ян Эрик Солем

Программирование компьютерного зрения на языке Python

Programming Computer Vision with Python

Jan Erik Solem

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Программирование компьютерного зрения на языке Python

Ян Эрик Солем

ДМК издательство
Москва, 2016

УДК 004.93Python
ББК 32.972.1
С60

С60 Ян Эрик Солем

Программирование компьютерного зрения на языке Python. / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2016. – 312 с.: ил.

ISBN 978-5-97060-200-3

Если вы хотите разобраться в основах теории и алгоритмов компьютерного зрения, то эта книга – как раз то, что вам нужно. Вы узнаете о методах распознавания объектов, трехмерной реконструкции, обработке стереоизображений, дополненной реальности и других приложениях компьютерного зрения. Изложение сопровождается понятными примерами на языке Python. При этом объяснения даются в общих чертах, без погружения в сухую теорию.

Издание идеально подходит для студентов, исследователей и энтузиастов-любителей с базовыми знаниями математики и навыками программирования.

УДК 004.93Python
ББК 32.972.1

ISBN 978-1-44931-654-9 (англ.)
ISBN 978-5-97060-200-3 (рус.)

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В современном мире изображения и видео встречаются повсюду. На сайтах обмена фотографиями и в социальных сетях миллиарды таких файлов. Поисковые системы предлагают изображения едва ли не на каждый запрос. Практически все мобильные телефоны и компьютеры оснащены камерами. В телефонах многих людей хранятся гигабайты фотографий и видео.

Алгоритмы, позволяющие понять, что изображено на этих фотографиях, относятся к дисциплине, называемой «компьютерным зрением». Компьютерное зрение лежит в основе таких приложений, как поиск изображений, ориентация роботов в пространстве, анализ медицинских изображений, управление фотографиями и многих других.

Эта книга задумана как доступное введение в практические вопросы компьютерного зрения с изложением теоретических и алгоритмических основ и ориентирована на студентов, научных работников и энтузиастов-любителей. Для языка программирования Python имеется много отличных и при том бесплатных модулей для обработки изображений, математических вычислений и добычи данных.

Работая над этой книгой, я придерживался следующих принципов.

- Излагать материал так, чтобы у читателя возникало желание выполнять на своих компьютерах примеры по мере чтения текста.
- Использовать по преимуществу бесплатное ПО с открытым исходным кодом, не требующее много времени на начальное изучение. Python казался мне очевидным выбором.
- Полнота и замкнутость. В этой книге рассматривается не весь предмет компьютерного зрения, но она полна в том смысле, что весь используемый в примерах код присутствует и снабжен пояснениями. Читатель сможет повторить все примеры и на их основе писать собственные программы.
- Отдавать предпочтение широте охвата, а не деталям. Способствовать рождению идеи и побуждать к самостоятельным исследованиям, а не излагать сухую теорию.

Короче говоря, я хотел написать книгу, которая станет источником вдохновения для всех, кто интересуется программированием компьютерного зрения.

Требования к читателю и структура книги

В этой книге рассматриваются теоретические основы и алгоритмы решения широкого круга задач. Ниже приведено краткое описание излагаемого материала.

Необходимые знания и навыки

- Начальный опыт программирования. Вы должны знать, как пользоваться редактором и запускать скрипты, уметь структурировать код и иметь представление о базовых типах данных. Знакомство с Python или еще каким-либо скриптовым языком, например Ruby или Matlab, будет дополнительным подспорьем.
- Основы математики. Для понимания примеров следует знать о матрицах, векторах, умножении матриц, стандартных математических функциях и понятиях, в частности, о производной и градиенте. Примеры, в которых используется более сложный математический аппарат, можно пропустить без ущерба для понимания.

Чему вы научитесь

Практическое программирование обработки изображений на Python.

- Методы компьютерного зрения, лежащие в основе разнообразных реальных приложений.
- Многие фундаментальные алгоритмы и способы их реализации и применения в собственных программах.

В примерах демонстрируется решение следующих задач: распознавание объектов, поиск изображений по содержанию, оптическое распознавание символов, оптический поток, трассировка, трехмерная реконструкция, обработка стереоизображений, дополненная реальность, определение положения камеры, создание панорамных

изображений, очистка от шумов, группировка изображений и многое другое.

Структура книги

Глава 1 «Основы обработки изображений»

Знакомство с основными средствами работы с изображениями и используемыми модулями Python. Здесь же рассматриваются многие базовые примеры, которые потребуются в других главах.

Глава 2 «Локальные дескрипторы изображений»

Описываются методы обнаружения особых точек в изображении и использование их для поиска соответственных точек и участков в разных изображениях.

Глава 3 «Преобразования изображений»

Описываются основные преобразования изображений и методы их вычисления. Рассматриваются, в частности, деформирование изображений и создание панорам.

Глава 4 «Модели камер и дополненная реальность»

Дается введение в модели камер, создание проекций трехмерных изображений и оценивание положения камеры.

Глава 5 «Многовидовая геометрия»

Объясняется, как работать с несколькими изображениями одной и той же сцены, дается введение в многовидовую геометрию и вычисление трехмерной реконструкции изображений.

Глава 6 «Кластеризация изображений»

Описано несколько методов кластеризации и показано, как с их помощью сгруппировать изображения по сходству или по содержанию.

Глава 7 «Поиск изображений»

Показано, как хранить изображения, чтобы их можно было эффективно искать по визуальному содержанию.

Глава 8 «Классификация изображений по содержанию»

Описаны алгоритмы классификации изображений по содержанию и их применение для распознавания объектов в изображениях.

Глава 9 «Сегментация изображений»

Описаны различные методы разбиения изображений на значимые участки с применением кластеризации, интерактивного взаимодействия с пользователем или моделей изображений.

Глава 10 «OpenCV»

Показано, как использовать интерфейс из Python к популярной библиотеке компьютерного зрения OpenCV и как работать с данными, полученными от фото- или видеокамеры.

В конце книги приведен список литературы. Библиографические ссылки представлены номером работы в квадратных скобках, например [20].

Введение в компьютерное зрение

Под компьютерным зрением понимается автоматическое извлечение информации из изображений. В роли информации может выступать все, что угодно: 3D-модели, положение камеры, обнаружение и распознавание объектов, группировка изображений и поиск изображений по содержанию. В этой книге принято широкое определение компьютерного зрения, включающее такие вещи, как деформирование, очистка от шумов и дополненная реальность[1].

Иногда требуется, чтобы компьютерное зрение моделировало зрение человека, иногда данные подвергаются статистической обработке, а иногда ключом к решению задачи являются геометрические соображения. Мы попытаемся охватить все эти подходы.

На практике компьютерное зрение представляет собой сплав программирования, моделирования и математики, иногда довольно сложный для усвоения. Я сознательно стремился свести теорию к минимуму – в духе максимы «делай настолько просто, насколько возможно, но не проще». Математические пассажи необходимы для понимания алгоритмов. Некоторые главы, по самой природе материала, насыщены математикой (особенно главы 4 и 5). При желании можно пропустить всю математику и просто пользоваться кодом.

Python и NumPy

Все примеры программ в этой книге написаны на языке Python. Это лаконичный язык с хорошей поддержкой ввода-вывода, численных расчетов, обработки изображений и построения графиков. У языка есть некоторые особенности, например синтаксически значимые отступы и компактный синтаксис, к которым нужно привыкнуть.

[1] В этих примерах порождаются новые изображения, так что они относятся скорее к обработке изображений, чем к выделению информации как таковому.

Все примеры рассчитаны на версию не ниже Python 2.6, поскольку большинство пакетов доступны только для таких версий. В версиях Python 3.x имеется много отличий, не всегда совместимых с Python 2.x и с экосистемой нужных нам пакетов (но это временно).

Знакомство с основами Python облегчит понимание материала. Для начинающих хорошими отправными точками могут стать книга Mark Lutz «Learning Python» [20] и документация на сайте *http://www.python.org/*.

При программировании компьютерного зрения необходимы средства для представления векторов и матриц и операций над ними. Все это есть в модуле `NumPy`, где векторы и матрицы представлены типом `array`. Такое же представление используется и для изображений. Хорошим справочным пособием по модулю `NumPy` является бесплатная книга Travis Oliphant «Guide to NumPy» [24]. Тем, кто только начинает осваивать `NumPy`, поможет также документация на сайте *http://numpy.scipy.org/*. Для визуализации результатов мы пользуемся модулем `Matplotlib`, а для более сложных математических вычислений – модулем `SciPy`. Это основные пакеты, знакомству с ними посвящена глава 1.

Помимо основных, есть много других бесплатных пакетов на Python, применяемых для таких задач, как чтение данных в формате JSON и XML, загрузка и сохранение данных, генерация графиков, графическое программирование, создание демонстраций в вебе, построение классификаторов и многих других. Обычно они нужны только для разработки определенных приложений или демонстраций, в противном случае их можно не устанавливать.

Заслуживает упоминания также интерактивная оболочка IPython, упрощающая отладку и экспериментирование. Скачать ее (вместе с документацией) можно с сайта *http://ipython.org/*.

Обозначения и графические выделения

Код выглядит следующим образом:

```
# точки
x = [100,100,400,400]
y = [200,500,200,500]

# нанести точки на график
plot(x,y)
```

В книге применяются следующие графические выделения.

Курсив
: Определения терминов, имена файлов и переменных.

`Моноширинный`
: Имена функций, модулей Python, примеры кода и вывод на консоль.

Гиперссылка
: URL-адреса

Простой текст
: Все остальное.

Математические формулы приводятся либо в основном тексте, например $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$, либо в отдельной строке:

$$f(\mathbf{x}) = \sum_i w_i x_i + b$$

Формулы нумеруются, только если на них имеются ссылки.

Скалярные величины обозначаются строчными буквами (s, r, λ, θ, ...), матрицы – заглавными буквами (A, V, H, ...) (буквой I обозначается изображение, представленное в виде массива), а векторы – полужирными строчными буквами ($\mathbf{t}$, $\mathbf{c}$, ...). Точки на плоскости (в изображении) и трехмерном пространстве обозначаются соответственно $\mathbf{x} = [x, y]$ и $\mathbf{X} = [X, Y, Z]$.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: Jan Erik Solem «*Programming Computer Vision*

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *http://oreil.ly/comp_vision_w_python*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.
Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.
Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Я выражаю благодарность всем участвовавшим в процессе подготовки и производства книги. Мне помогали многие сотрудники издательства O'Reilly. Отдельное спасибо редактору Энди Ораму (O'Reilly) и Полу Анагностопулосу (Windfall Software), помогавшему в процессе производства.

Многие высказывали замечания по поводу различных черновых редакций книги, которые я выкладывал в сеть. Особо хочу отметить заслуги Класа Джозефсона (Klas Josephson) и Хакана Ардё (Håkan Ardö), приславших подробные замечания и отзывы. Фредрик Кал (Fredrik Kahl) и Пау Гаргалло (Pau Gargallo) помогали в проверке

фактов. Спасибо всем читателям за слова ободрения и за стремление улучшить текст и код. Письма от незнакомых людей, высказывающих свои мысли о черновике рукописи, – мощный источник мотивации.

Напоследок я хочу поблагодарить своих друзей и членов семьи за поддержку и понимание на протяжении всего времени, когда я посвящал вечера и выходные писанию книги. А особенно свою жену, Сару, которая давно уже поддерживает меня во всем.

Об авторе

Ян Эрик Солем, большой энтузиаст языка Python, занимается исследованиями и популяризацией компьютерного зрения. По образованию прикладной математик, работал доцентом, техническим директором стартапа, пишет книги. Иногда рассказывает о компьютерном зрении и Python в своем блоге по адресу *www.janeriksolem.net*. Много лет использует Python для преподавания, исследований и разработки промышленных приложений в области компьютерного зрения. В настоящее время проживает в Сан-Франциско.

ГЛАВА 1. Основы обработки изображений

Эта глава представляет собой введение в обработку изображений. На целом ряде примеров иллюстрируется использование основных пакетов Python, применяемых для работы с изображениями. Здесь мы познакомимся с базовыми средствами чтения изображений, их преобразования и масштабирования, вычисления производных, сохранения результатов, построения графиков и т. д. Все это понадобится в последующих главах.

1.1. PIL – библиотека Python Imaging Library

Библиотека *Python Imaging Library* (*PIL*) содержит общие средства для обработки изображений и разнообразных полезных операций, в том числе: изменение размера, кадрирование, поворот, преобразование цветов и т. д. Библиотека распространяется бесплатно, ее можно скачать с сайта *http://www.pythonware.com/products/pil/*.

PIL позволяет читать изображения, записанные в большинстве существующих форматов, и сохранять в наиболее популярных. Наиболее важен модуль `Image`. Для чтения изображения служит такой код:

```
from PIL import Image

pil_im = Image.open('empire.jpg')
```

В качестве значения метод возвращает объект изображения *pil_im*.

Для преобразования цветов используется метод `convert()`. Чтобы прочитать изображение и сделать его полутоновым, достаточно просто добавить вызов `convert('L')`:

```
pil_im = Image.open('empire.jpg').convert('L')
```

Ниже приведено несколько примеров, заимствованных из документации по PIL на странице *http://www.pythonware.com/library/pil/handbook/index.htm*. Результаты их работы показаны на рис. 1.1.

Рис. 1.1. Результаты обработки изображений с помощью PIL

Преобразование изображения в другой формат

С помощью метода `save()` PIL может сохранять изображения в большинстве графических форматов. В следующем примере мы читаем изображения из файлов, перечисленных в списке *filelist*, и преобразуем их в формат JPEG:

```
from PIL import Image
import os

for infile in filelist:
  outfile = os.path.splitext(infile)[0] + ".jpg"
  if infile != outfile:
     try:
        Image.open(infile).save(outfile)
     except IOError:
        print "не могу преобразовать", infile
```

Функция `open()` создает объект изображения PIL, а метод `save()` сохраняет это изображение в файле с указанным именем. Новое имя файла совпадает с исходным за исключением расширения, которое теперь равно «.jpg». PIL умеет определять формат файла по расширению имени. Проверяется, что исходный формат файла отличен от JPEG, а в случае ошибки преобразования печатается сообщение.

В этой книге мы не раз будем задавать списки файлов, подлежащих обработке. Вот, например, как можно сформировать список имен всех графических файлов в папке. Создайте файл *imtools.py*, в который мы будем помещать полезные функции общего назначения, и добавьте в него такую функцию:

```
import os

def get_imlist(path):
 """ Возвращает список имен всех
   jpg-файлов в каталоге. """

 return [os.path.join(path,f) for f in os.listdir(path)
                              if f.endswith('.jpg')]
```

А теперь вернемся к PIL.

Создание миниатюр

Создавать миниатюры с помощью PIL очень просто. Метод `thumbnail()` принимает кортеж, в котором задается новый размер, и преобразует изображение в миниатюру указанного размера. Вот как создается миниатюра, для которой длина наибольшей стороны равна 128 пикселей:

```
pil_im.thumbnail((128,128))
```

Копирование и вставка областей

Обрезка (кадрирование) изображения производится методом `crop()`:

```
box = (100,100,400,400)
region = pil_im.crop(box)
```

Прямоугольная область определяется 4-кортежем, в котором задаются координаты сторон в порядке (левая, верхняя, правая, нижняя). В PIL используется система координат с началом (0, 0) в левом верхнем углу. С вырезанной областью можно затем производить различные операции, например повернуть и вставить в то же место методом `paste()`:

```
region = region.transpose(Image.ROTATE_180)
pil_im.paste(region,box)
```

Изменение размера и поворот

Для изменения размера изображения служит метод `resize()`, которому передается кортеж, определяющий новый размер:

```
out = pil_im.resize((128,128))
```

Для поворота изображения вызывается метод `rotate()` и задается угол в направлении против часовой стрелки:

```
out = pil_im.rotate(45)
```

Результаты работы некоторых примеров показаны на рис. 1.1. Слева показано исходное изображение, затем его полутоновый вариант, результат вставки вырезанной и повернутой области и, наконец, миниатюра.

1.2. Библиотека Matplotlib

Для построения графиков, а также рисования точек и прямых или кривых линий на изображении применяется графическая библиотека `Matplotlib`, содержащая куда больше средств построения графиков, чем имеется в PIL. `Matplotlib` генерирует высококачественные рисунки, с ее помощью получены многие иллюстрации к этой книге. Интерфейс `PyLab`, включенный в `Matplotlib`, – это набор функций, позволяющих пользователю строить графики. Исходный текст `Matplotlib` открыт, скачать библиотеку можно бесплатно с сайта *http://matplotlib.sourceforge.net/*, где имеется также подробная документация и учебные пособия. Ниже приведено несколько примеров использования функций, которые понадобятся в этой книге.

Рисование точек и прямых линий

Хотя библиотека позволяет создавать красивые столбчатые и секторные диаграммы, диаграммы рассеяния и т. п., для целей компьютерного зрения достаточно всего нескольких команд. Нам нужно выделять особые точки, соответственные области и обнаруженные объекты с помощью точек и прямых линий. Вот пример нанесения на изображение нескольких точек и отрезка прямой:

```
from PIL import Image
```

```
from pylab import *

# прочитать изображение в массив
im = array(Image.open('empire.jpg'))

# поместить на график изображение
imshow(im)

# несколько точек
x = [100,100,400,400]
y = [200,500,200,500]

# нанести точки в виде красных звездочек
plot(x,y,'r*')

# нарисовать отрезок, соединяющий первые две точки
plot(x[:2],y[:2])

# добавить заголовок и показать график
title('Plotting: "empire.jpg"')
show()
```

Здесь на график помещается изображение, затем четыре точки, обозначенные красными звездочками (их координаты задаются в списках x и y), и, наконец, отрезок прямой, соединяющий первые две точки из списка (по умолчанию рисуется синим цветом). Результат показан на рис. 1.2. Функция `show()` открывает графический интерфейс рисунка и создает окна. Цикл обработки сообщений в ГИП блокирует выполнение скриптов на все время, пока не будет закрыто последнее окно рисунка. Вызывать `show()` следует только один раз, обычно в конце скрипта. Обратите внимание, что в `PyLab` начало координат расположено в левом верхнем углу, как принято при работе с изображениями. Оси полезны для отладки, но если хотите получить более красивую картинку, то можете отключить их:

```
axis('off')
```

Тогда график будет выглядеть, как на рис. 1.2 справа.

Существует много параметров для задания цвета и стиля. Самые полезные короткие команды приведены в таблицах 1.1, 1.2 и 1.3. Используются они так:

```
plot(x,y)        # по умолчанию сплошная синияя линия
plot(x,y,'r*')   # красные маркеры в виде звездочек
plot(x,y,'go-') # зеленая линия с маркерами-кружочками
plot(x,y,'ks:') # черная пунктирная линия с маркерами-квадратиками
```

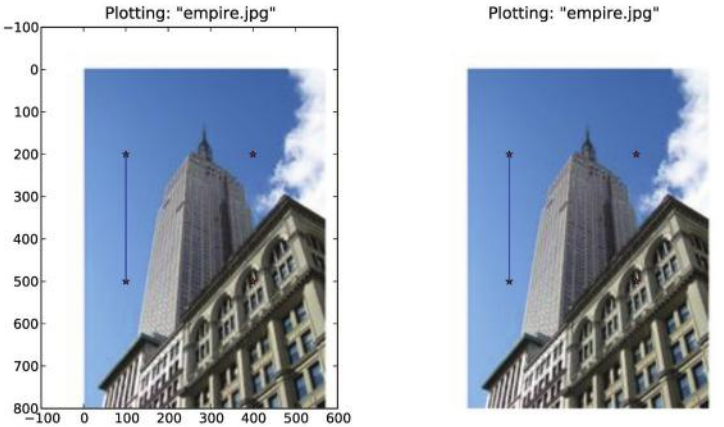


Рис. 1.2. Примеры графиков, построенных с помощью `Matplotlib`. Изображение с точками и отрезком прямой с осями координат и без них

Таблица 1.1. Основные команды задания цветов при построении графиков в `PyLab`

Цвет	
`'b'`	синий
`'g'`	зеленый
`'r'`	красный
`'c'`	голубой
`'m'`	пурпурный
`'y'`	желтый
`'k'`	черный
`'w'`	белый

Таблица 1.2. Основные команды задания цветов при построении графиков в `PyLab`

Стиль линии	
`'-'`	сплошная
`'--'`	штриховая
`':'`	пунктирная

Таблица 1.3. Основные команды задания маркеров при построении графиков в `PyLab`

Маркер	
`'.'`	точка
`'o'`	кружочек
`'s'`	квадратик
`'*'`	звездочка
`'+'`	плюс
`'x'`	x

Изолинии и гистограммы изображений

Рассмотрим два примера специальных графиков: изолинии и гистограммы изображений. Визуализация изолиний изображения (или изолиний других двумерных функций) может оказаться очень полезной. Для этого нужны полутоновые изображения, потому что изолинии строятся по значению какой-нибудь одной величины. Делается это так:

```
from PIL import Image
from pylab import *

# прочитать изображение в массив
im = array(Image.open('empire.jpg').convert('L'))

# создать новый рисунок
figure()

# не использовать цвета
gray()

# показать изолинии относительно левого верхнего угла
contour(im, origin='image')
axis('equal')
axis('off')
```

Как и раньше, метод `convert()` преобразует исходное изображение в полутоновое.

Гистограмма изображения – это график распределения значений пикселей. Область возможных значений разбивается на интервалы, и для каждого интервала определяется количество пикселей, значения которых попадают в этот интервал. Для построения гистограммы полутонового изображения применяется функция `hist()`:

```
figure()
hist(im.flatten(),128)
show()
```

Ее второй аргумент задает количество интервалов. Отметим, что изображение сначала необходимо линеаризовать, потому что `hist()` ожидает получить одномерный массив. Метод `flatten()` преобразует любой массив в одномерный, располагая значения по строкам. На рис. 1.3 показаны изолинии и гистограмма.

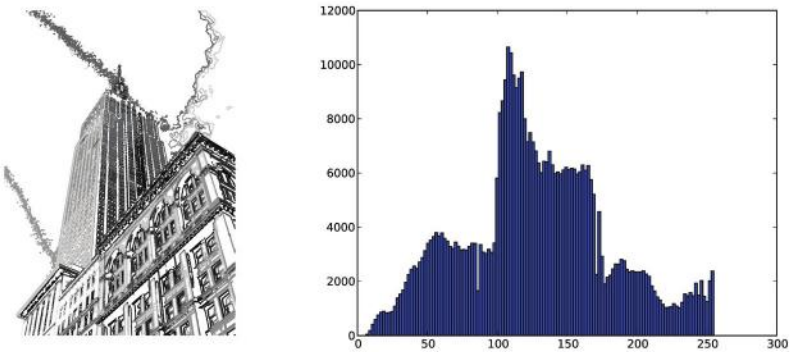


Рис. 1.3. Примеры визуализации изолиний изображения и построения гистограмм с помощью `Matplotlib`

Интерактивное аннотирование

Иногда пользователям нужно взаимодействовать с приложением, например помечать какие-то точки изображения или аннотировать обучающие данные. В `PyLab` для этого имеется простая функция `ginput()`:

```
from PIL import Image
from pylab import *

im = array(Image.open('empire.jpg'))
imshow(im)
print 'Щелкните в 3 точках'
x = ginput(3)
print 'вы щелкнули:',x
show()
```

После вывода графика эта программа будет ждать, пока пользователь три раза щелкнет мышью в области окна рисунка. Координаты $[x, y]$ отмеченных точек сохраняются в списке x.

1.3. Пакет NumPy

Пакет `NumPy` (*http://www.scipy.org/NumPy/*) часто используется для научных расчетов на Python. NumPy включает ряд полезных концепций, в частности объекты-массивы (для представления векторов, матриц, изображений и т. п.), и функции из области линейной алгебры. Массивы NumPy будут использоваться почти во всех примерах ниже[1]. Объект массива позволяет выполнять такие важные операции, как умножение матриц, транспонирование, решение систем линейных уравнений, скалярное умножение и нормировку векторов. Все это необходимо для таких вещей, как совмещение изображений, деформирование, различные виды моделирования, классификация и группировка изображений и т. д.

Пакет `NumPy` можно бесплатно скачать по адресу *http://www.scipy.org/Download*, а в документации (*http://docs.scipy.org/doc/numpy/*) имеются ответы на большинство вопросов. Дополнительные сведения по `NumPy` можно найти в бесплатно распространяемой книге [24].

Представление изображения в виде массива

Загружая изображения в предыдущих примерах, мы преобразовывали их в массивы `NumPy` с помощью функции `array()`, хотя и не акцентировали на этом внимание. Массивы в `NumPy` многомерные и могут использоваться для представления векторов, матриц и изображений. Массив очень похож на список (или список списков), но может содержать только элементы одного типа. Если тип не указан при создании массива, то он автоматически выводится из данных.

Ниже показано, как это делается для изображений:

```
im = array(Image.open('empire.jpg'))
print im.shape, im.dtype

im = array(Image.open('empire.jpg').convert('L'),'f')
print im.shape, im.dtype
```

На консоли будет напечатано:

```
(800, 569, 3) uint8
(800, 569) float32
```

[1] `PyLab` на самом деле включается некоторые компоненты `NumPy`, в частности тип массива. Потому-то мы и могли пользоваться этим типом в примерах из раздела 1.2.

Первый кортеж описывает форму массива изображения (количество строк, столбцов и цветовых каналов), а следующая за ним строка – тип данных, хранящихся в элементах массива. Изображения обычно кодируются 8-разрядными целыми без знака (uint8), поэтому в первом случае после загрузки изображения и преобразования его в массив печатается тип «uint8». Во втором случае производится преобразование в полутоновое изображение, и при создании массива указан дополнительный аргумент «f». Эта короткая команда означает, что нужно использовать тип с плавающей точкой. Другие типы описаны в [24]. Обратите внимание, что для полутоновых изображений кортеж, описывающий форму, содержит только два элемента; понятно, что информация о цвете не нужна.

К элементам массива можно обращаться по индексам. Чтобы получить значение с координатами i, j и цветовым каналом k, нужно написать:

```
value = im[i,j,k]
```

Операция срезки массива позволяет обращаться сразу к нескольким элементам. Она возвращает представление части массива, определяемое интервалами индексов. Вот несколько примеров для полутонового изображения:

```
im[i,:] = im[j,:]    # скопировать значения из строки j в строку i
im[:,i] = 100 # присвоить всем элементам в столбце i значение 100
im[:100,:50].sum()    # просуммировать элементы в прямоугольнике,
                      # образованном первыми 100 строками
                      # и первыми 50 столбцами
im[50:100,50:100]     # строки 50-100, столбцы 50-100 (сотые не
                      # включаются)
im[i].mean()           # среднее значение в строке i
im[:,-1]               # последний столбец
im[-2,:] (или im[-2]) # предпоследняя строка
```

Обратите внимание на пример, где указан только один индекс. В таком случае он интерпретируется как индекс строки. Также обратите внимание на последние два примера. Отрицательный индекс отсчитывается от последнего элемента в обратном направлении. Мы часто будем использовать срезку для доступа к значениям пикселей, поэтому важно понимать, как она работает.

Есть много способов работы с массивами. Мы будем знакомиться с ними по ходу изложения. Более подробные сведения можно найти в онлайновой документации или в книге [24].

Преобразование уровня яркости

Прочитав изображение в массив NumPy, мы можем применить к нему различные математические операции. Простой пример – преобразование уровня яркости полутонового изображения. Возьмем произвольную функцию f, отображающую интервал 0 ... 255 (или, если угодно, 0 ... 1) в себя (т. е. область значений совпадает с областью определения). Вот несколько примеров:

```
from PIL import Image
from numpy import *

im = array(Image.open('empire.jpg').convert('L'))
im2 = 255 - im                  # инвертировать изображение

im3 = (100.0/255) * im + 100    # привести к интервалу 100…200
im4 = 255.0 * (im/255.0)**2     # применить квадратичную функцию
```

В первом примере уровни яркости инвертируются, во втором яркость приводится к интервалу 100 ... 200, а в третьем применяется квадратичная функция, которая уменьшает значения более темных пикселей. На рис. 1.4 показаны графики функций, а на рис. 1.5 – получающиеся изображения. Для нахождения минимального и максимального значения яркости можно написать:

```
print int(im.min()), int(im.max())
```

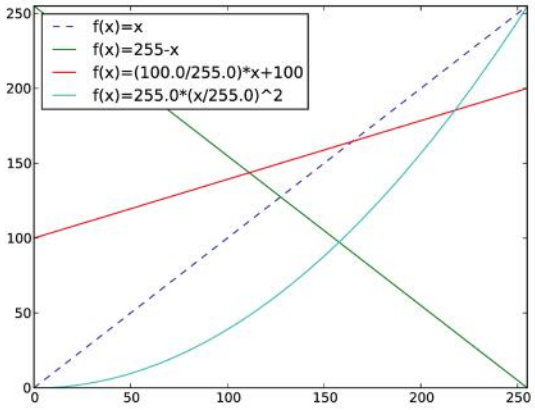


Рис. 1.4. Примеры преобразований уровня яркости. Показаны все три функции из примера выше, а также тождественная функция (штриховой линией)

Рис. 1.5. Преобразования уровня яркости. Результаты применения функций на рис. 1.4: инвертирование изображения – $f(x) = 255 - x$ (слева), приведение к ограниченному диапазону – $f(x) = (100/255)x + 100$ (в центре) и квадратичное преобразование – $f(x) = 255(x/255)^2$ (справа)

Выполнив это предложение для каждого из примеров выше, получим:

```
2 255
0 253
100 200
0 255
```

Функция `fromarray()` является обратной к `array()`:

```
pil_im = Image.fromarray(im)
```

Если тип «uint8» был заменен другим, как в примерах *im3* и *im4* выше, то перед созданием изображения нужно выполнить обратное преобразование:

```
pil_im = Image.fromarray(uint8(im))
```

Если вы не уверены в типе входных данных, то на всякий случай выполните эту операцию. Отметим, что `NumPy` всегда выбирает «самый узкий» тип, способный представить данные. При выполнении умножения или деления координат точек целочисленный тип изменяется на тип с плавающей точкой.

Изменение размера изображения

Массивы `NumPy` будут нашим основным средством при работе с изображениями и данными. Но для массивов нет простого способа изменить размер, хотя для изображений эта операция очень полезна. Чтобы написать функцию изменения размера, мы можем воспользоваться показанным выше преобразованием объектов изображений. Добавьте следующую функцию в файл *imtools.py*:

```
def imresize(im,sz):
  """ Изменить размер массива с помощью PIL. """
  pil_im = Image.fromarray(uint8(im))
  return array(pil_im.resize(sz))
```

Эта функция нам еще не раз пригодится.

Выравнивание гистограммы

Весьма полезным примером преобразования яркости является *выравнивание гистограммы*. Эта операция изменяет гистограмму яркости, так чтобы результирующая гистограмма содержала все возможные значения яркости и при этом примерно в одинаковом количестве. Она часто применяется для нормировки яркости перед последующей обработкой, а также для повышения контрастности.

В данном случае для преобразования используется *функция распределения* (cumulative distribution function, cdf) значений пикселей в изображении (нормированная так, чтобы привести значения к требуемому диапазону).

Добавьте следующую функцию в файл *imtools.py*:

```
def histeq(im,nbr_bins=256):
  """ Выравнивание гистограммы полутонового изображения. """

  # получить гистограмму изображения
  imhist,bins = histogram(im.flatten(),nbr_bins,normed=True)
  cdf = imhist.cumsum() # функция распределения
  cdf = 255 * cdf / cdf[-1] # нормировать

  # использовать линейную интерполяцию cdf для нахождения
  # значений новых пикселей
  im2 = interp(im.flatten(),bins[:-1],cdf)
  return im2.reshape(im.shape), cdf
```

Эта функция принимает полутоновое изображение и количество интервалов в гистограмме, а возвращает изображение, для которого

гистограмма выровнена с помощью функций распределения. Обратите внимание на использование последнего элемента (с индексом –1) для нормировки cdf на диапазон 0 ... 1. Попробуйте применить ее к какому-нибудь изображению:

```
from PIL import Image
from numpy import *

im=array(Image.open('AquaTermi_lowcontrast.jpg').convert('L'))
im2,cdf = imtools.histeq(im)
```

На рис. 1.6 и 1.7 приведены примеры выравнивания гистограммы. Сверху показаны гистограммы до и после выравнивания, а также сама функция распределения. Как видите, контрастность увеличилась, и детали в темных участках теперь видны более отчетливо.

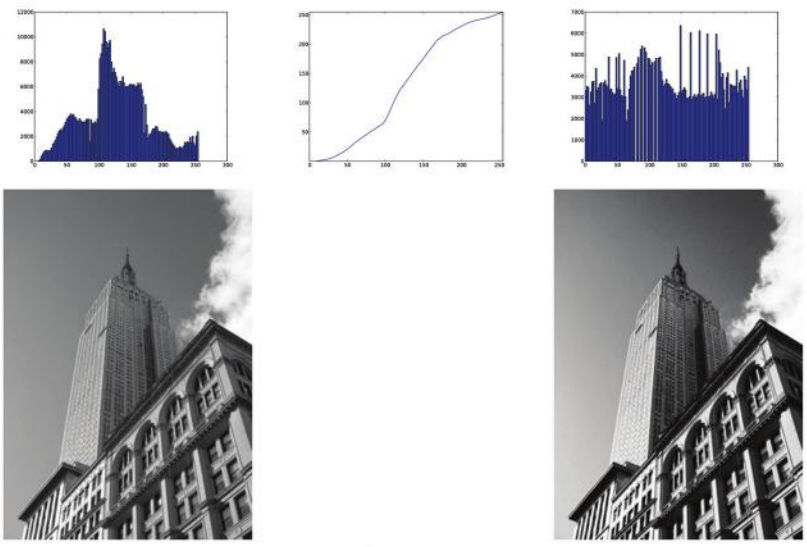


Рис. 1.6. Пример выравнивания гистограммы.
Слева – исходное изображение и его гистограмма.
В центре – функция преобразования уровня яркости.
Справа – изображение и гистограмма после выравнивания

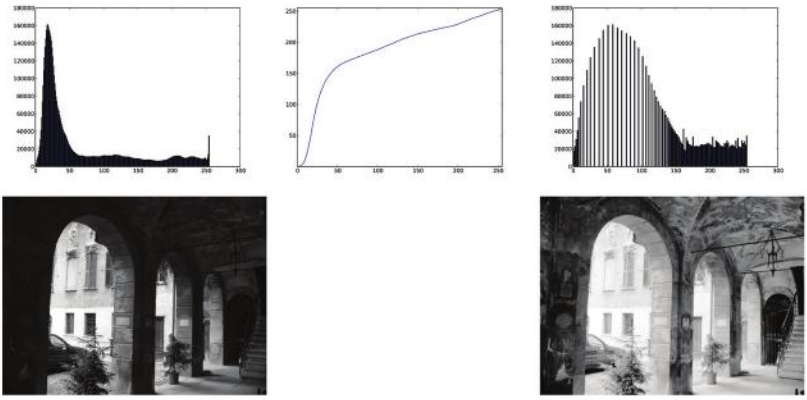


Рис. 1.7. Пример выравнивания гистограммы. Слева – исходное изображение и его гистограмма. В центре – функция преобразования уровня яркости. Справа – изображение и гистограмма после выравнивания

Усреднение изображений

Усреднение дает простой способ уменьшить зашумленность изображения, а также часто применяется для создания художественных эффектов. Вычислить среднее изображений из списка нетрудно. В предположении, что все они одного размера, мы можем просто вычислить сумму и разделить ее на число изображений. Добавьте следующую функцию в файл *imtools.py*:

```
def compute_average(imlist):
  """ Вычислить среднее списка изображений. """

  # открыть первое изображение и преобразовать в массив типа float
  averageim = array(Image.open(imlist[0]), 'f')

  for imname in imlist[1:]:
     try:
        averageim += array(Image.open(imname))
     except:
          print imname + '...пропущено'
          averageim /= len(imlist)

  # вернуть среднее в виде массива значений типа uint8
  return array(averageim, 'uint8')
```

Функция содержит простую обработку исключений, чтобы пропускать изображения, которые не удалось открыть. По-другому вычислить среднее изображение можно с помощью функции `mean()`. Она требует, чтобы все изображения были упакованы в массив и, если изображений много, то потребляет уйму памяти. Мы воспользуемся ей в следующем разделе.

Метод главных компонент для изображений

Метод главных компонент (*PCA*) – полезная техника понижения размерности, оптимальная в том смысле, что представляет изменчивость обучающих данных, используя наименьшее возможное число измерений. Даже у крохотного полутонового изображения размером 100×100 число измерений равно 10 000, поэтому его можно рассматривать как точку в 10 000-мерном пространстве. У мегапиксельного изображения измерений миллионы. Не удивительно, что понижение такой высокой размерности оказывается чрезвычайно кстати во многих приложениях компьютерного зрения. Матрицу проекции, получающуюся в результате применения метод PCA, можно рассматривать как переход к системе координат, в которой координаты упорядочены в порядке важности.

Для применения метода главных компонент к данным изображения их нужно сначала преобразовать в одномерный вектор, например, методом `NumPy flatten()`.

Линеаризованные изображения собираются в матрицу, по одной строке на каждое изображение. Перед тем как вычислять доминирующие направления, строки центрируются относительно среднего изображения. Для нахождения главных компонент обычно используется сингулярное разложение (SVD), но когда размерность очень высока, вычисление SVD оказывается слишком медленным, поэтому заменяется одним полезным приемом. Вот как это выглядит в коде:

```
from PIL import Image
from numpy import *

def pca(X):
  """ Метод главных компонент
  вход: матрица X, в которой обучающие данные хранятся в виде
    линеаризованных массивов, по одному в каждой строке
  выход: матрица проекции (наиболее важные измерения в начале),
```

```
    дисперсия и среднее."""

    # получить количество измерений
    num_data,dim = X.shape

    # центрировать данные
    mean_X = X.mean(axis=0)
    X = X - mean_X

    if dim>num_data:
        # PCA с компактным трюком
        M = dot(X,X.T) # ковариационная матрица
        e,EV = linalg.eigh(M) # собственные значения и собственные векторы
        tmp = dot(X.T,EV).T # это и есть компактный трюк
        V = tmp[::-1] # меняем порядок, потому что нам нужны
                      # последние собственные векторы
        S = sqrt(e)[::-1] # меняем порядок, потому что собственные
                          # значения перечислены в порядке возрастания
        for i in range(V.shape[1]):
            V[:,i] /= S
    else:
        # PCA с использованием сингулярного разложения
        U,S,V = linalg.svd(X)
        V = V[:num_data] # имеет смысл возвращать только первые
                         # num_data строк

    # вернуть матрицу проекции, дисперсию и среднее
    return V,S,mean_X
```

Эта функция сначала центрирует данные, вычитая среднее по каждому направлению. Затем вычисляются собственные векторы, соответствующие собственным значениям ковариационной матрицы, – с помощью компактного трюка или сингулярного разложения. Здесь мы воспользовались функцией `range()`, которая принимает целое число n и возвращает список целых чисел 0 ... (n – 1). Можно использовать также функцию `arange()`, возвращающую массив, или `xrange()`, возвращающую генератор (и, возможно, работающую быстрее). Но в этой книге мы будем пользоваться только функцией `range()`.

Мы переключаемся с сингулярного разложения на вычисление собственных векторов (меньшей) ковариационной матрицы XX^T, если количество точек меньше размерности векторов. Существует также возможность вычислять только собственные векторы, соответствующие k наибольшим собственным значениям (где k – желаемое число измерений), что дает дополнительный выигрыш в скорости. Оставляем исследование этой возможности любознательным читателям, поскольку она выходит за рамки книги. Строки матрицы V

ортогональны и соответствуют осям координат в порядке убывания дисперсии обучающих данных.

Опробуем этот метод на примере изображений шрифтов. Файл *fontimages.zip* содержит миниатюрные изображения буквы «a», напечатанные различными шрифтами и затем отсканированные. Использовано 2359 бесплатных шрифтов[2]. В предположении, что имена файлов изображений хранятся в списке *imlist* вместе с показанным выше кодом, и все это находится в файле *pca.py*, вычисление главных компонент производится следующим образом:

```
from PIL import Image
from numpy import *
from pylab import *
import pca

im = array(Image.open(imlist[0])) # открыть одно изображение
                                  # для получения размера
m,n = im.shape[0:2]  # получить размер изображений
imnbr = len(imlist) # получить число изображений

# создать матрицу для хранения всех линеаризованных изображений
immatrix = array([array(Image.open(im)).flatten()
for im in imlist],'f')

# выполнить метод главных компонент
V,S,immean = pca.pca(immatrix)

# показать несколько изображений (среднее и первые 7 мод)
figure()
gray()
subplot(2,4,1)
imshow(immean.reshape(m,n))
for i in range(7):
  subplot(2,4,i+2)
  imshow(V[i].reshape(m,n))

show()
```

Отметим, что изображения нужно восстановить из одномерного представления с помощью функции `reshape()`. В результате работы этого примера должно получиться восемь изображений в одном окне рисунка, как на рис. 1.8. Чтобы разместить несколько графиков в одном окне, мы воспользовались функцией `PyLab subplot()`.

[2] Изображения любезно подготовил Мартин Солли (Martin Solli) (*http://webstaff.itn.liu.se/~marso/*).

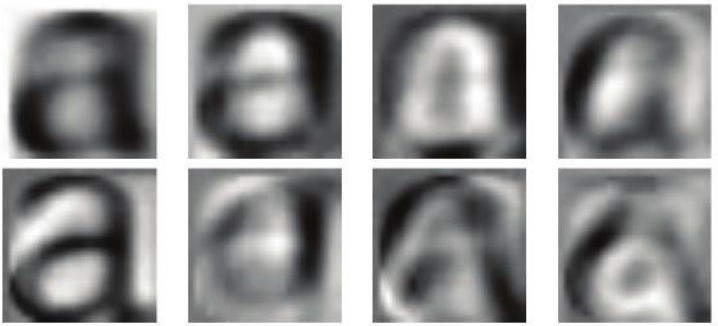


Рис. 1.8. Среднее изображение (слева вверху) и первые семь мод, т. е. направлений с наибольшей дисперсией

Использование модуля pickle

Если вы хотите сохранить результаты для последующего использования, то к вашим услугам входящий в состав Python модуль `pickle`. Этот модуль принимает почти любой объект Python и преобразует его в строку. Эта операция называется *сериализацией*[3], а обратная ей – восстановление объекта из строкового представления – *десериализацией*. А строки уже легко можно сохранить или передать по сети.

Проиллюстрируем на примере. Допустим, нам нужно сохранить среднее изображение и главные компоненты изображений шрифтов из предыдущего раздела. Это делается так:

```
# сохранить среднее изображение и главные компоненты
f = open('font_pca_modes.pkl', 'wb')
pickle.dump(immean,f)
pickle.dump(V,f)
f.close()
```

Как видите, в один файл можно поместить несколько сериализованных объектов. Для *pkl*-файлов существуют разные протоколы; если вы не уверены, то лучше всего использовать для чтения и записи двоичный формат. Чтобы загрузить данные в другом сеансе работы с Python, вызовите метод `load()`:

```
# загрузить среднее изображение и главные компоненты
f = open('font_pca_modes.pkl', 'rb')
immean = pickle.load(f)
V = pickle.load(f)
f.close()
```

[3] В Python употребляются термины *pickling* (консервирование) и *unpickling*. – *Прим. перев.*

Отметим, что порядок объектов должен быть точно таким же, как при сохранении! Существует также оптимизированный модуль `cpickle`, написанный на C и полностью совместимый со стандартным модулем pickle. Дополнительные сведения можно найти в документации по модулю pickle на странице *http://docs.python.org/library/pickle.html*.

Далее в этой книге мы будем использовать предложение `with` вместе с операциями чтения и записи файла. Эта конструкция, появившаяся в версии Python 2.5, автоматически закрывает файл, даже если во время работы с ним имели место ошибки. Вот как программируется сохранение и загрузка данных при использовании `with`:

```
# открыть файл и сохранить данные
with open('font_pca_modes.pkl', 'wb') as f:
  pickle.dump(immean,f)
  pickle.dump(V,f)
```

и

```
# открыть файл и загрузить данные
with open('font_pca_modes.pkl', 'rb') as f:
  immean = pickle.load(f)
  V = pickle.load(f)
```

На первый взгляд, кажется странным, но в действительности очень полезная штука. Если не нравится, можете пользоваться функциями `open` и `close`, как было показано выше.

В качестве альтернативы модулю pickle в `NumPy` имеются простые функции для чтения и записи текстовых файлов, полезные в случае, когда данные не содержат сложных структур, например, списка выбранных в изображении точек. Для сохранения массива x в файле нужно написать:

```
savetxt('test.txt',x,'%i')
```

Последний параметр означает, что следует использовать целочисленный формат. А вот как производится чтение из файла:

```
x = loadtxt('test.txt')
```

Дополнительные сведения можно найти в онлайновой документации по адресу *http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html*.

Наконец, в `NumPy` имеются специальные функции для сохранения и загрузки массивов. Поищите в документации функции `save()` и `load()`.

1.4. Пакет SciPy

SciPy (*http://scipy.org/*) – пакет математических программ с открытым исходным кодом. Он построен на базе NumPy и предоставляет эффективные функции для выполнения многих операций, в том числе численного интегрирования, оптимизации, статистики, обработки сигналов и, что для нас самое главное, обработки изображений. Как будет ясно из дальнейшего, в SciPy немало полезных модулей. Этот пакет можно бесплатно скачать со страницы *http://scipy.org/Download*.

Размытие изображений

Классический – и очень полезный – пример свертки изображения – *гауссово размытие*. Идея в том, что (полутоновое) изображение I сворачивается с гауссовым ядром, в результате чего получается размытое изображение:

$$I_\sigma = I * G_\sigma.$$

Здесь $*$ обозначает операцию свертки, G_σ – двумерное гауссово ядро со стандартным отклонением σ:

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

Гауссово размытие применяется для интерполяции, вычисления особых точек изображения и во многих других приложениях.

В состав SciPy входит модуль фильтрации scipy.ndimage.filters, который позволяет вычислять подобные свертки с помощью быстрого метода разделения переменных. Нам нужно только написать:

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters

im = array(Image.open('empire.jpg').convert('L'))
im2 = filters.gaussian_filter(im,5)
```

Второй параметр функции gaussian_filter() задает стандартное отклонение.

На рис. 1.9 показаны результаты размытия изображения с увеличивающимся σ. Чем больше стандартное отклонение, тем меньше остается деталей. Для размытия цветных изображений нужно применить гауссово размытие к каждому цветовому каналу:

```
im = array(Image.open('empire.jpg'))
im2 = zeros(im.shape)
for i in range(3):
  im2[:,:,i] = filters.gaussian_filter(im[:,:,i],5)
im2 = uint8(im2)
```

Последнее преобразование к типу uint8 не всегда обязательно, оно просто дает 8-разрядное представление пикселей. Того же эффекта можно было бы добиться, написав:

```
im2 = array(im2,'uint8')
```

Дополнительные сведения об этом модуле и различных параметрах можно найти в документации по SciPy на странице *http://docs.scipy.org/doc/scipy/reference/ndimage.html*.

Рис. 1.9. Пример гауссова размытия с применением модуля scipy.ndimage.filters: (a) исходное полутоновое изображение; (b) фильтра Гаусса с $\sigma = 2$; (c) с $\sigma = 5$; (d) с $\sigma = 10$

Производные изображения

То, как изменяется яркость изображения, – важная информация, необходимая во многих приложениях, и у нас будет не один случай убедиться в этом. Изменение яркости полутонового изображения I описывается его производными по x и по y – I_x и I_y (для цветных изображений производные берутся для каждого канала в отдельности).

Градиентом изображения называется вектор $\nabla I = [I_x, I_y]^T$. У вектора градиента есть два важных свойства: *модуль градиента*

$$|\nabla I| = \sqrt{I_x^2 + I_y^2},$$

который описывает, насколько сильно изменяется яркость, и *угол градиента*

$$\alpha = \arctan2(I_y, I_x),$$

описывающий направление наибольшего изменения яркости в каждой точке изображения (пикселе). Функция `NumPy arctan2()` возвращает угол в радианах в интервале от $-\pi$ до π.

Для вычисления производных изображения можно воспользоваться дискретными аппроксимациями. Проще всего реализовать их в виде сверток:

$$I_x = I * D_x \text{ и } I_y = I * D_y.$$

В качестве D_x и D_y часто берут *операторы Прюитта*:

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ и } D_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix},$$

или *операторы Собеля*

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ и } D_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Эти операторы легко реализовать, воспользовавшись стандартной сверткой из модуля `scipy.ndimage.filters`, например:

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters

im = array(Image.open('empire.jpg').convert('L'))

# Операторы Собеля
imx = zeros(im.shape)
filters.sobel(im,1,imx)

imy = zeros(im.shape)
filters.sobel(im,0,imy)

magnitude = sqrt(imx**2+imy**2)
```

В этом фрагменте производные по x и по y, а также модуль градиента вычисляются с помощью *оператора Собеля*. Его второй аргумент

определяет, по какому направлению – x или y – брать производную, а третий – где сохранять результат. На рис. 1.10 показаны изображение и его производные. В обоих производных изображениях положительным значениям производных соответствуют яркие пиксели, а отрицательным – темные. В серых областях значения близки к нулю.

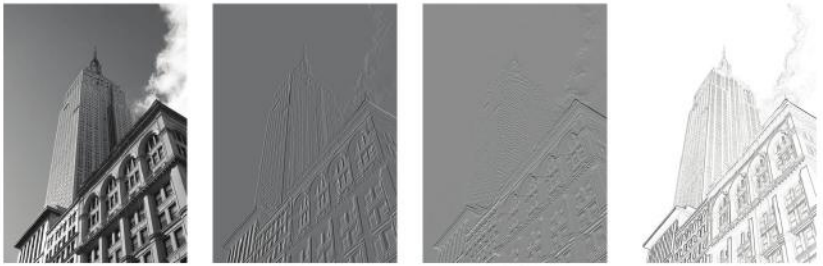


Рис. 1.10. Результаты вычисления производных изображений с помощью операторов Собеля: (a) исходное полутоновое изображение; (b) производная по *x*; (c) производная по *y*; (d) модуль градиента

У этого подхода есть недостаток: производные берутся в масштабе, определяемом разрешением изображения. Чтобы уменьшить зависимость от шума и вычислить производные в любом масштабе, можно воспользоваться *фильтром Гаусса*:

$$I_x = I * G_{\sigma x} \text{ и } I_y = I * G_{\sigma y},$$

где $G_{\sigma x}$ и $G_{\sigma y}$ производные по x и по y гауссовой функции G_σ со стандартным отклонением σ.

Функция `filters.gaussian_filter()`, которой мы раньше пользовались для размытия, может также вычислять гауссовы производные, если передать ей дополнительные аргументы. Для этого нужно написать:

```
sigma = 5 # стандартное отклонение

imx = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (0,1), imx)

imy = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (1,0), imy)
```

Третий аргумент определяет порядок производных в каждом направлении, а второй – стандартное отклонение. Детали смотрите в

документации. На рис. 1.11 показаны производные и модуль градиента для разных масштабов. Сравните с результатом размытия при тех же масштабах на рис. 1.9.

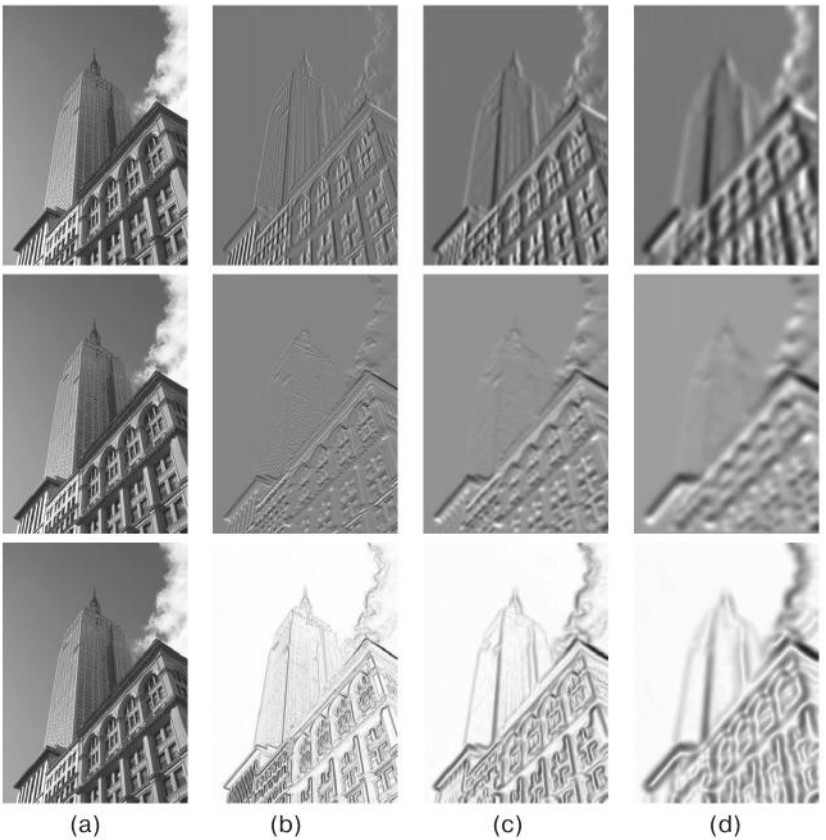


Рис.1.11. Результаты вычисления производных изображений с помощью фильтров Гаусса: производная по *x* (вверху), производная по *y* (в центре) и модуль градиента (внизу). (a) исходное полутоновое изображение; (b) фильтр Гаусса с $\sigma = 2$, (c) с $\sigma = 5$, (d) с $\sigma = 10$

Морфология – подсчет объектов

Морфология (или *математическая морфология*) – это инфраструктура и набор методов обработки изображений с целью измерения и анализа базовых форм. Обычно морфология применяется к *бинарным изображениям*, но может использоваться и для полутоно-

вых. Бинарным называется изображение, в котором каждый пиксель может принимать одно из двух значений, обычно 0 или 1. Такие изображения часто получаются в результате бинаризации имеющегося изображения, например, для подсчета объектов или определения их размеров. Хорошее введение в принципы работы морфологии имеется в статье из википедии по адресу *http://en.wikipedia.org/wiki/Mathematical_morphology*.

Морфологические операции включены в модуль `morphology` из пакета `scipy.ndimage`. Функции подсчета и измерения объектов для бинарных изображений находятся в модуле `measurements`. Рассмотрим простой пример их использования.

Возьмем бинарное изображение, показанное на рис. 1.12 (a)[4]. Для подсчета объектов в нем можно использовать такой код:

```
from scipy.ndimage import measurements,morphology

# загрузить изображение и выполнить бинаризацию
im = array(Image.open('houses.png').convert('L'))
im = 1*(im<128)

labels, nbr_objects = measurements.label(im)
print "Количество объектов:", nbr_objects
```

Здесь мы загружаем изображение и производим его бинаризацию. Умножение на 1 преобразует булев массив в бинарный. Затем функция `label()` находит отдельные объекты и сопоставляет пикселям метки в соответствии с объектами, которым они принадлежат. На рис. 1.12 (b) показан массив *labels*. Значения яркости соответствуют индексам объектов. Как видите, между некоторыми объектами имеются перемычки. Их можно устранить с помощью операции бинарного открытия:

```
# морфология - операция открытия разделяет объекты
im_open = morphology.binary_opening(im,ones((9,5)),iterations=2)

labels_open, nbr_objects_open = measurements.label(im_open)
print "Количество объектов:", nbr_objects_open
```

Второй аргумент функции `binary_opening()` определяет *структурный элемент* – массив, определяющий, каких соседей пикселя использовать. В данном случае мы использовали 9 пикселей (4 сверху, сам пиксель и четыре снизу) в направлении y и 5 пикселей в направлении x. В качестве структурного элемента можно задать любой мас-

[4] Это изображение на самом деле получено в результате "сегментации". Подробнее о том, как оно было создано, написано в разделе 9.3.

сив; его ненулевые элементы определяют соседей. Параметр *iterations* определяет, сколько раз выполнять операцию. Поэкспериментируйте и посмотрите, как изменяется количество объектов. Изображение после открытия и соответствующее меточное изображение показаны на рис. 1.12 (c) и (d). Как и следовало ожидать, существует функция `binary_closing()` для выполнения обратной операции. Изучение этой и других функции из модулей `morphology` и `measurements` оставляем в качестве упражнения. Прочитать о них можно в документации по адресу *http://docs.scipy.org/doc/scipy/reference/ndimage.html*.

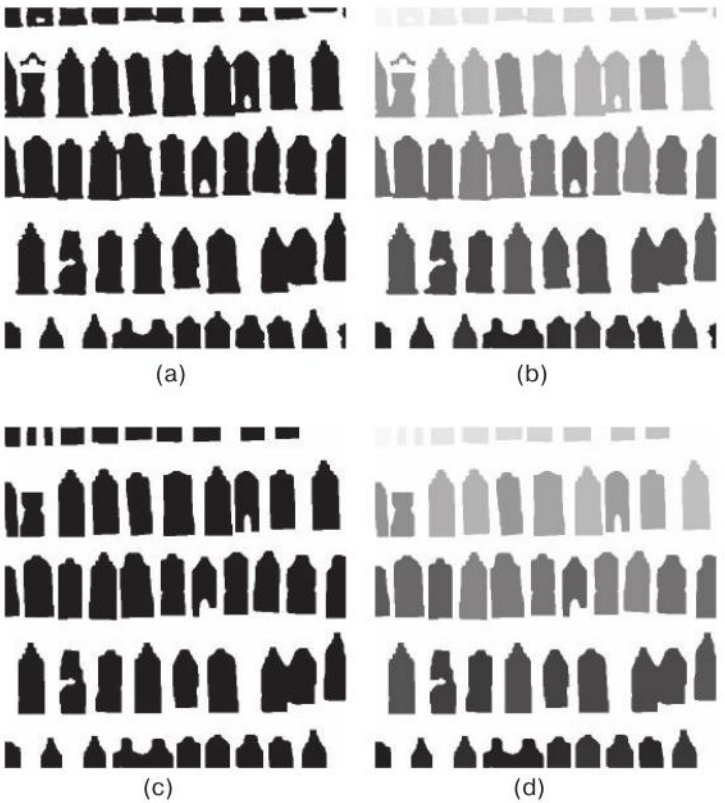


Рис. 1.12. Пример морфологии. Сначала выполнена операция бинарного открытия для разделения объектов, а затем произведен их подсчет. (a) исходное бинарное изображение; (b) меточное изображение, соответствующее исходному, уровень яркости равен индексу объекта; (c) бинарное изображение после открытия; (d) меточное изображение, соответствующее открытому

Полезные модули в пакете SciPy

В состав `SciPy` входят несколько полезных модулей для ввода и вывода, в частности `io` и `misc`.

Чтение и запись mat-файлов

Если в вас уже имеются или вы нашли в Интернете какие-то интересные данные в формате Matlab (с расширением *.mat*), то с помощью модуля `scipy.io` их можно будет прочитать:

```
data = scipy.io.loadmat('test.mat')
```

Возвращенный объект *data* содержит словарь, ключи которого соответствуют именам переменных в исходном *mat*-файле. Переменные хранятся в виде массива. Записать *mat*-файл тоже просто. Нужно лишь создать словарь, содержащий все переменные, которые требуется сохранить, и вызвать функции `savemat()`:

```
data = {}
data['x'] = x
scipy.io.savemat('test.mat',data)
```

При этом массив *x* сохраняется таким образом, что при чтении в Matlab он получит имя «x». Дополнительные сведения о модуле `scipy.io` можно найти в документации по адресу *http://docs.scipy.org/doc/scipy/reference/io.html*.

Сохранение массивов как изображений

Поскольку мы манипулируем изображениями, а вычисления производим над объектами-массивами, полезно иметь возможность сохранять массивы непосредственно как изображения[5]. Многие изображения в этой книге так и созданы.

В модуле `scipy.misc` имеется функция `imsave()`. Для сохранения массива *im* в файле достаточно написать:

```
from scipy.misc import imsave
imsave('test.jpg',im)
```

В модуле `scipy.misc` имеется также знаменитое тестовое изображение «Lena»:

```
lena = scipy.misc.lena()
```

[5] Любой рисунок `PyLab` можно сохранить в различных графических форматах, нажав кнопку "save" в окне рисунка.

Этот код возвращает массив, соответствующий полутоновому варианту изображения размером 512×512.

1.5. Более сложный пример: очистка изображения от шумов

Завершим эту главу очень полезным примером: очисткой изображения от шумов. Под *очисткой от шумов* понимается процесс удаления из изображения шума с сохранением всех деталей и общей структуры. Мы будем использовать *модель Рудина-Ошера-Фатеми* (*ROF*), впервые описанную в работе [28]. Удаление шума важно для многих приложений: от подчистки фотографий, сделанных на выходных, до повышения качества спутниковых снимков. Модель ROF обладает интересным свойством: она сглаживает изображение, сохраняя в то же время границы и структуры.

Математика, лежащая в основе модели ROF, весьма сложна и выходит за рамки этой книги. Мы приведем лишь краткое упрощенное описание, а затем покажем, как реализовать решатель ROF на базе алгоритма Шамболя (Chambolle) [5].

Полной вариацией (total variation, TV) полутонового изображения I называется сумма норм градиентов. В непрерывном представлении она имеет вид:

$$J(I) = \int |\nabla I| d\mathbf{x}.$$

А в дискретной форме записывается так:

$$J(I) = \sum_{\mathbf{x}} |\nabla I|,$$

где суммирование производится по всем точкам изображения $\mathbf{x} = [x, y]$.

В варианте модели ROF, предложенном Шамболем, целью является нахождение очищенного от шумов изображения U, для которого достигается минимум функции

$$\min_U \|I - U\|^2 + 2\lambda J(U),$$

где норма $\|I - U\|$ измеряет различие между U и исходным изображением I. Это означает, что модель ищет изображение, которое было бы «плоским», но допускает «скачки» на границах между участками.

Следующий код основан на приведенном в статье Шамболя алгоритме:

```
from numpy import *

def denoise(im,U_init,tolerance=0.1,tau=0.125,tv_weight=100):
  """ Реализация модели очистки от шумов Рудина-Ошера-Фатеми
    (ROF) с использованием численного алгоритма, приведенного
    в формуле (11) А. Шамболя (2005).

    Вход: зашумленное изображение (полутоновое), начальная
    гипотеза о U, вес члена, регуляризирующего TV, величина шага,
    допуск в условии остановки.

    Выход: очищенное от шума и текстуры изображение,
    остаточная текстура. """
    m,n = im.shape # размер зашумленного изображения

    # инициализация
    U = U_init
    Px = im # компонента x двойственной задачи
    Py = im # компонента y двойственной задачи
    error = 1

    while (error > tolerance):
       Uold = U
       # градиент переменной прямой задачи
       GradUx = roll(U,-1,axis=1)-U # компонента x градиента U
       GradUy = roll(U,-1,axis=0)-U # компонента y градиента U

       # изменить переменную двойственной задачи
       PxNew = Px + (tau/tv_weight)*GradUx
       PyNew = Py + (tau/tv_weight)*GradUy
       NormNew = maximum(1,sqrt(PxNew**2+PyNew**2))
       Px = PxNew/NormNew # изменить компоненту x (двойственной задачи)
       Py = PyNew/NormNew # изменить компоненту y (двойственной задачи)

       # изменить переменную прямой задачи
       RxPx = roll(Px,1,axis=1) # циклический сдвиг компоненты x
                                # вдоль оси x
       RyPy = roll(Py,1,axis=0) # циклический сдвиг компоненты y
                                # вдоль оси y

       DivP = (Px-RxPx)+(Py-RyPy) # дивергенция двойственного поля
       U = im + tv_weight*DivP # изменение переменной прямой задачи

       # пересчитать погрешность
       error = linalg.norm(U-Uold)/sqrt(n*m);

    return U,im-U # очищенное от шумов изображение и остаточная текстура
```

В этом примере мы воспользовались функцией `roll()`, которая циклически сдвигает элементы массива вдоль указанной оси. Это очень удобно при вычислении разности между соседями, в данном случае для вычисления производных изображений. Мы также использовали функцию `linalg.norm()`, которая измеряет различие двух массивов (в данном случае матриц изображений *U* и *Uold*). Сохраните функцию `denoise()` в файле *rof.py*.

Начнем с синтезированного примера зашумленного изображения:

```
from numpy import *
from numpy import random
from scipy.ndimage import filters
import rof

# синтезировать изображение с шумом
im = zeros((500,500))
im[100:400,100:400] = 128
im[200:300,200:300] = 255
im = im + 30*random.standard_normal((500,500))

U,T = rof.denoise(im,im)
G = filters.gaussian_filter(im,10)

# сохранить результат
from scipy.misc import imsave
imsave('synth_rof.pdf',U)
imsave('synth_gaussian.pdf',G)
```

Исходное изображение и результаты его обработки показаны на рис. 1.13. Как видим, модель ROF прекрасно сохраняет границы.

Рис. 1.13. Пример очистки от шумов синтезированного изображения с помощью модели ROF: (a) исходное зашумленное изображение; (b) изображение после гауссова размытия ($\sigma = 10$); (c) изображение после очистки от шумов

Теперь посмотрим, что произойдет с реальным изображением:

```
from PIL import Image
from pylab import *
import rof

im = array(Image.open('empire.jpg').convert('L'))
U,T = rof.denoise(im,im)

figure()
gray()
imshow(U)
axis('equal')
axis('off')
show()
```

Результат показан на рис. 1.14 (c) рядом с размытым изображением для сравнения. Как видим, очистка от шумов по методу ROF сохраняет границы и структуры внутри изображения, размывая в то же время «шумы».

(a) (b) (c)

Рис. 1.14. Пример очистки от шумов полутонового изображения с помощью модели ROF: (a) исходное зашумленное изображение; (b) изображение после гауссова размытия ($\sigma = 5$); (c) изображение после очистки от шумов

Упражнения

1. Возьмите какое-нибудь изображение и примените к нему гауссово размытие, как на рис. 1.9. Нарисуйте изолинии для возрастающих значений σ. Что происходит? Можете ли вы объяснить причину?

2. Реализуйте операцию *нерезкого маскирования* (*https://ru.wikipedia.org/wiki/Нерезкое_маскирование*), для чего сначала размойте изображение, а затем вычтите размытое изображение из исходного. Это повышает резкость изображения. Попробуйте применить операцию к полутоновому и цветному изображению.
3. Альтернативой выравниванию гистограммы является метод нормировки изображения с помощью вычисления *изображения-частного*. Оно получается путем деления изображения на его размытый вариант $I/(I * G_\sigma)$. Реализуйте эту операцию и протестируйте на каких-нибудь примерах.
4. Напишите функцию, которая находит контуры простых объектов внутри изображения (например, квадрата на белом фоне) с помощью градиентов.
5. Воспользуйтесь модулем и направлением градиента для распознавания отрезков прямых внутри изображения. Оцените длину отрезков и параметры соответствующих прямых. Нанесите отрезки поверх изображения на графике.
6. Примените функцию `label()` к бинаризованному изображению по своему выбору. Воспользовавшись гистограммами и результирующим меточным изображением, постройте график распределения размеров объектов внутри изображения.
7. Поэкспериментируйте с последовательным применением морфологических операций к бинаризованному изображению по своему выбору. Подобрав параметры, при которых получается хороший результат, попробуйте применить функцию `center_of_mass` из модуля `morphology` для нахождения координат центра каждого объекта и нанесите найденные точки на изображение.

Соглашения в примерах кода

Начиная с главы 2, мы будем предполагать, что в начале каждого файла импортированы библиотеки PIL, `NumPy` и `Matplotlib`:

```
from PIL import Image
from numpy import *
from pylab import *
```

Это позволит не отвлекаться на технические детали. В тех примерах, где требуются модули из пакета `SciPy`, мы будем импортировать их явно.

Ревнители чистоты, наверное, возразят против такого «огульного» включения и будут настаивать на чем-то вроде

```
import numpy as np
import matplotlib.pyplot as plt
```

чтобы можно было указать пространство имен (и, стало быть, знать, из какого модуля берется функция). При этом из `Matplotlib` импортируется только модуль `pyplot`, а части `NumPy`, импортированные посредством `PyLab`, вообще не нужны. Пуристы и опытные программисты знают, в чем разница, и могут выбирать тот вариант, который им больше нравится. Я же решил так не поступать, чтобы сделать примеры понятнее всем читателям.

Да будет покупатель бдителен!

ГЛАВА 2. Локальные дескрипторы изображений

Эта глава посвящена нахождению соответственных точек и участков изображений. Мы познакомимся с двумя типами локальных дескрипторов и методами сопоставления изображений. Эти локальные признаки еще не раз встретятся нам в книге, они являются важным строительным блоком во многих приложениях, например: создание панорам, дополненная реальность и вычисление трехмерных реконструкций.

2.1. Детектор углов Харриса

Алгоритм *обнаружения углов Харриса* (иногда его называют также детектором углов Харриса-Стивенса) – один из простейших детекторов углов. Идея заключается в том, чтобы найти особые точки, в окрестности которых имеются границы в нескольких направлениях, это и есть угловые точки.

Определим положительно-полуопределенную симметричную матрицу $\mathbf{M}_I = \mathbf{M}_I(\mathbf{x})$, где $\mathbf{x}$ – точка внутри изображения:

$$\mathrm{M}_I = \nabla I \nabla I^T = \begin{bmatrix} I_x \\ I_y \end{bmatrix} \begin{bmatrix} I_x & I_y \end{bmatrix} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \qquad (2.1)$$

Здесь ∇I – вектор градиента изображения, содержащий производные I_x и I_y (определения производных были даны выше). По построению, $\mathbf{M}_I$ имеет ранг 1, а ее собственные значения равны $\lambda_1 = |\nabla I|^2$ и $\lambda_2 = 0$. Таким образом, у нас имеется по одной матрице для каждой точки изображения.

Обозначим W матрицу весов (обычно фильтр Гаусса G_σ). Операция поэлементной свертки

$$\overline{\mathbf{M}}_I = W * \boldsymbol{M}_I \qquad (2.2)$$

дает локальное усреднение $\mathbf{M}_I$ по соседним пикселям. Получающаяся в результате матрица $\overline{\mathbf{M}}_I$ называется *матрицей Харриса*. Ширина W определяет размер представляющей интерес окрестности $\mathbf{x}$. Смысл усреднения матрицы $\mathbf{M}_I$ по области состоит в том, что собственные значения будут изменяться в зависимости от локальных свойств изображения. Если градиенты в этой области меняются, то второе собственное значение $\overline{\mathbf{M}}_I$ не будет равно 0. Если же градиенты одинаковы, то собственные значения будут такие же, как у $\mathbf{M}_I$.

В зависимости от значений ∇I в области существуют три случая (ниже λ_1 и λ_2 – собственные значения матрицы Харриса $\overline{\mathbf{M}}_I$).

- Если λ_1 и λ_2 – большие положительные числа, то в точке $\mathbf{x}$ имеется угол.
- Если λ_1 велико, а $\lambda_2 \approx 0$, то существует граница, и при усреднении $\mathbf{M}_I$ по области собственные значения изменяются не сильно.
- Если $\lambda_1 \approx \lambda_2 \approx 0$, то в точке нет никаких особенностей.

Чтобы отличить существенный случай от остальных, не вычисляя собственные значения, Харрис и Стивенс в работе [12] ввели такую индикаторную функцию:

$$\det(\overline{\mathbf{M}}_I) - \kappa \operatorname{trace}(\overline{\mathbf{M}}_I)^2.$$

Чтобы избавиться от весовой постоянной κ, проще бывает использовать в качестве индикатора частное

$$\frac{\det(\overline{\mathbf{M}})}{\operatorname{trace}(\overline{\mathbf{M}}_I)^2}.$$

Посмотрим, как это выглядит в коде. Для этого нам понадобится модуль `scipy.ndimage.filters` для вычисления производных с помощью фильтров Гаусса. Причина и на этот раз в том, чтобы подавить шумы в процессе обнаружения углов.

Сначала включим в файл *harris.py* функцию отклика Харриса, в которой будут использоваться фильтры Гаусса. Параметр σ, как и раньше, определяет масштаб фильтров. Эту функцию можно модифицировать, выбрав разный масштаб в направлениях x и y, а также другой масштаб для усреднения.

```
from scipy.ndimage import filters

def compute_harris_response(im,sigma=3):
  """ Вычислить функцию отклика детектора углов Харриса для
    каждого пикселя полутонового изображения. """

  # производные
  imx = zeros(im.shape)
  filters.gaussian_filter(im, (sigma,sigma), (0,1), imx)
  imy = zeros(im.shape)
  filters.gaussian_filter(im, (sigma,sigma), (1,0), imy)

  # вычислить элементы матрицы Харриса
  Wxx = filters.gaussian_filter(imx*imx,sigma)
  Wxy = filters.gaussian_filter(imx*imy,sigma)
  Wyy = filters.gaussian_filter(imy*imy,sigma)

  # определитель и след матрицы
  Wdet = Wxx*Wyy - Wxy**2
  Wtr = Wxx + Wyy

  return Wdet / Wtr
```

Тем самым мы получаем изображение, в котором каждый пиксель содержит значение функции отклика Харриса. Теперь осталось только извлечь из этого изображения нужную информацию. Хороший результат дает выборка всех точек со значениями выше порогового и дополнительным ограничением: углы должны отстоять друг от друга на некоторое минимальное расстояние. Чтобы реализовать эту идею, отберем все пиксели-кандидаты, отсортируем их в порядке убывания значений функции отклика и отделим области, слишком близкие к точкам, уже признанным углами. Добавьте в файл *harris.py* следующую функцию:

```
def get_harris_points(harrisim,min_dist=10,threshold=0.1):
  """ Возвращает углы на изображении, построенном по функции
      отклика Харриса.
      min_dist - минимальное число пикселей между углами и
      границей изображения """

  # найти точки-кандидаты, для которых функция отклика больше порога
  corner_threshold = harrisim.max() * threshold
  harrisim_t = (harrisim > corner_threshold) * 1

  # получить координаты кандидатов
  coords = array(harrisim_t.nonzero()).T

  # ... и их значения
```

```
    candidate_values = [harrisim[c[0],c[1]] for c in coords]

    # отсортировать кандидатов
    index = argsort(candidate_values)

    # сохранить данные о точках-кандидатах в массиве
    allowed_locations = zeros(harrisim.shape)
    allowed_locations[min_dist:-min_dist,min_dist:-min_dist] = 1

    # выбрать наилучшие точки с учетом min_distance
    filtered_coords = []
    for i in index:
       if allowed_locations[coords[i,0],coords[i,1]] == 1:
          filtered_coords.append(coords[i])
          allowed_locations[(coords[i,0]-min_dist):(coords[i,0]+min_dist),
             (coords[i,1]-min_dist):(coords[i,1]+min_dist)] = 0

    return filtered_coords
```

Теперь у нас есть все для определения угловых точек внутри изображения. Чтобы показать их, добавим в файл *harris.py* функцию построения графика, в которой используется библиотека `Matplotlib`:

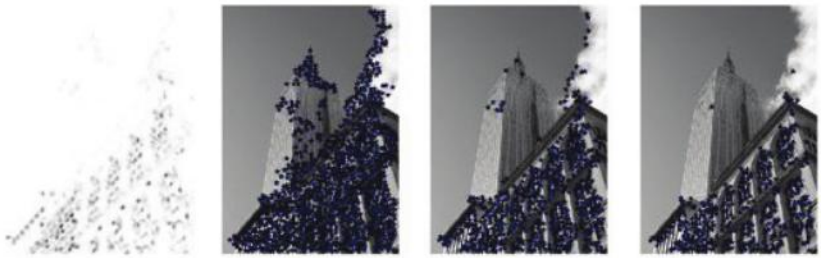
```
def plot_harris_points(image,filtered_coords):
  """ Нанести на график углы, найденные в изображении. """

  figure()
  gray()
  imshow(image)
  plot([p[1] for p in filtered_coords],[p[0] for p in filtered_coords],'*')
  axis('off')
  show()
```

Попробуйте выполнить такие команды:

```
im = array(Image.open('empire.jpg').convert('L'))
harrisim = harris.compute_harris_response(im)
filtered_coords = harris.get_harris_points(harrisim,6)
harris.plot_harris_points(im, filtered_coords)
```

Здесь мы читаем изображение и преобразуем его в полутоновое. Затем вычисляем функцию отклика и на основе ее значений выбираем точки. В конце точки наносятся на график поверх исходного изображения. Результаты показаны на рис. 2.1.

Обзор других подходов к обнаружению углов, включая усовершенствованный детектор Харриса и более поздние результаты, смотрите, например, в статье *http://en.wikipedia.org/wiki/Corner_detection*.



(a) (b) (c) (d)

Рис. 2.1. Пример работы детектора углов Харриса: (a) функция отклика Харриса; (b–d) обнаруженные углы при значениях порога 0.01, 0.05 и 0.1

Нахождение соответственных точек в изображениях

Детектор углов Харриса находит особые точки изображения, но не дает способа сравнить особые точки в разных изображениях и установить соответствие между углами. Нам необходимо снабдить каждую точку дескриптором и предложить способ сравнения дескрипторов.

Дескриптор особой точки – это сопоставленный точке вектор, который описывает окрестность этой точки в изображении. Чем лучше дескриптор, тем качественнее будет соответствие. Под *соответственными точками* мы понимаем точки разных изображений, относящиеся к одному и тому же объекту или точке сцены.

Угловые точки Харриса обычно комбинируются с дескриптором, состоящим из значений яркости в окружающем блоке изображения и нормированной взаимной корреляции. *Блок изображения* (image patch) – почти всегда прямоугольный участок изображения с центром в рассматриваемой точке.

В общем случае *корреляция* между двумя блоками изображения одинакового размера $I_1(\mathbf{x})$ и $I_2(\mathbf{x})$ определяется как

$$c(I_1, I_2) = \sum_{\mathbf{x}} f\,(I_1(\mathbf{x}), I_2(\mathbf{x})),$$

где функция f зависит от метода вычисления корреляции. Суммирование производится по всем точкам $\mathbf{x}$ в блоке изображения. В случае *взаимной корреляции* в качестве f берется функция $f\,(I_1, I_2) = I_1\, I_2$, так что $c(I_1, I_2) = I_1 \cdot I_2$, где точкой обозначено скалярное произведение

(блоков, представленных в виде векторов строк или столбцов). Чем больше значение $c(I_1, I_2)$, тем более похожи блоки I_1 и I_2.[1]

Нормированная взаимная корреляция – это вариант взаимной корреляции, она определяется следующим образом:

$$ncc(I_1, I_2) = \frac{1}{n-1}\sum_x \frac{(I_1(x)-\mu_1)}{\sigma_1}\cdot\frac{(I_2(x)-\mu_2)}{\sigma_2} \qquad (2.3)$$

где n – число пикселей в блоке, μ_1 и μ_2 – средние яркости, а μ_1 и μ_2 – стандартные отклонения в каждом блоке. Благодаря вычитанию средних и нормировке на стандартное отклонение этот метод слабо чувствителен к изменению яркости изображения.

Для выборки блоков изображения и их сравнения с помощью нормированной взаимной корреляции нам понадобятся еще две функции. Включите их в файл *harris.py*.

```
def get_descriptors(image,filtered_coords,wid=5):
    """ Для каждой точки вернуть значения пикселей в окрестности
     этой точки шириной 2*wid+1. (Предполагается, что выбирались
     точки с min_distance > wid). """
    desc = []
    for coords in filtered_coords:
        patch = image[coords[0]-wid:coords[0]+wid+1,
                      coords[1]-wid:coords[1]+wid+1].flatten()
        desc.append(patch)

    return desc

def match(desc1,desc2,threshold=0.5):
    """ Для каждого дескриптора угловой точки в первом изображении
     найти соответствующую ему точку во втором изображении,
     применяя нормированную взаимную корреляцию. """
    n = len(desc1[0])

    # попарные расстояния
    d = -ones((len(desc1),len(desc2)))
    for i in range(len(desc1)):
        for j in range(len(desc2)):
            d1 = (desc1[i] - mean(desc1[i])) / std(desc1[i])
            d2 = (desc2[j] - mean(desc2[j])) / std(desc2[j])
            ncc_value = sum(d1 * d2) / (n-1)
            if ncc_value > threshold:
                d[i,j] = ncc_value

    ndx = argsort(-d)
```

[1] Широко распространена также функция $f(I_1, I_2) = (I_1 - I_2)^2$, при которой корреляция является *суммой квадратов разностей*.

```
    matchscores = ndx[:,0]

    return matchscores
```

Первая функция принимает квадратный полутоновый блок с нечетной длиной стороны и центром в заданной точке, линеаризует его и добавляет в список дескрипторов. Вторая функция сопоставляет каждому дескриптору наилучшую пару в другом изображении, применяя нормированную взаимную корреляцию. Отметим, что перед сортировкой мы меняем знак расстояния на противоположный, поскольку чем больше значение, тем лучше соответствие. Чтобы еще улучшить результат, можно сопоставить второе изображение с первым и оставить только пары, оказавшиеся оптимальными в обоих случаях. Это делает такая функция:

```
def match_twosided(desc1,desc2,threshold=0.5):
    """ Двусторонний симметричный вариант match(). """

    matches_12 = match(desc1,desc2,threshold)
    matches_21 = match(desc2,desc1,threshold)

    ndx_12 = where(matches_12 >= 0)[0]

    # исключить несимметричные соответствия
    for n in ndx_12:
        if matches_21[matches_12[n]] != n:
            matches_12[n] = -1

    return matches_12
```

Получить наглядное представление о соответствиях можно, если расположить оба изображения рядом и соединить соответственные точки прямыми, как в следующем коде. Добавьте эти две функции в файл *harris.py*:

```
def appendimages(im1,im2):
    """ Вернуть новое изображение, на котором два исходных расположены рядом.

    # выбрать изображение с наименьшим числом строк и дополнить
    # его пустыми строками
    rows1 = im1.shape[0]
    rows2 = im2.shape[0]

    if rows1 < rows2:
        im1 = concatenate((im1,zeros((rows2-rows1,im1.shape[1]))),axis=0)
    elif rows1 > rows2:
        im2 = concatenate((im2,zeros((rows1-rows2,im2.shape[1]))),axis=0)

    # если ни то, ни другое, значит, число строк одинаково
```

```
    # и заполнять не нужно

    return concatenate((im1,im2), axis=1)

def plot_matches(im1,im2,locs1,locs2,matchscores,show_below=True):
    """ Показать рисунок, на котором соответственные точки
      соединены вход: im1,im2 (изображения в виде массивов),
        locs1,locs2 (координаты особых точек), matchscores
        (результат, возвращенный 'match()'), show_below (если нужно
        показать изображения под картиной соответствия)."""

    im3 = appendimages(im1,im2)
    if show_below:
        im3 = vstack((im3,im3))
    imshow(im3)

    cols1 = im1.shape[1]
    for i,m in enumerate(matchscores):
        if m>0:
            plot([locs1[i][1],locs2[m][1]+cols1],[locs1[i][0],locs2[m][0]],'c')
    axis('off')
```

На рис. 2.2 показан пример нахождения соответственных точек с помощью нормированной взаимной корреляции (в данном случае были взяты блоки размером 11×11 пикселей). Изображения были сгенерированы такой программой:

```
wid = 5
harrisim = harris.compute_harris_response(im1,5)
filtered_coords1 = harris.get_harris_points(harrisim,wid+1)
d1 = harris.get_descriptors(im1,filtered_coords1,wid)

harrisim = harris.compute_harris_response(im2,5)
filtered_coords2 = harris.get_harris_points(harrisim,wid+1)
d2 = harris.get_descriptors(im2,filtered_coords2,wid)

print 'начинается сопоставление'
matches = harris.match_twosided(d1,d2)
figure()
gray()
harris.plot_matches(im1,im2,filtered_coords1,filtered_coords2,matches)
show()
```

Если вы хотите нанести на график только часть соответствий, чтобы не загромождать рисунок, замените *matches*, например, на *matches[:100]* или случайную выборку индексов.

Легко видеть, что на рис. 2.2 много неправильных соответствий. Объясняется это тем, что взаимная корреляция блоков изображений дает не такие точные результаты, как более современные методы. По-

этому в реальном приложении следует использовать решения понадежнее. Еще одна проблема связана с тем, что эти дескрипторы не инвариантны относительно масштабирования или поворота, а выбор блоков влияет на результат.

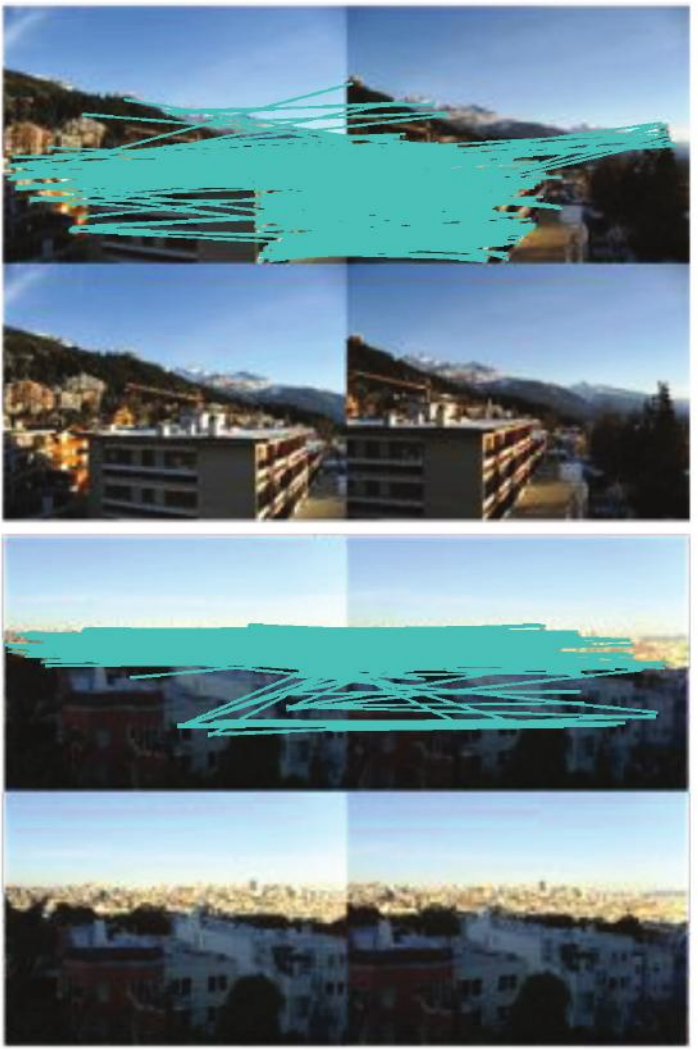


Рис. 2.2. Соответствие, получающееся в результате применения нормированной взаимной корреляции к блокам, окружающим угловые точки Харриса

В последние годы было проделано много работы по улучшению методов обнаружения и описания особых точек. В следующем разделе мы рассмотрим один из лучших алгоритмов.

2.2. SIFT – масштабно-инвариантное преобразование признаков

Один из самых удачных локальных дескрипторов изображения, открытых за последние десять лет, – метод *масштабно-инвариантного преобразования признаков* (Scale-Invariant Feature Transform, *SIFT*), сформулированный Дэвидом Лоуи (David Lowe) в работе [17]. Позже метод SIFT был усовершенствован и детально описан в статье [18] и в таком виде выдержал испытание временем. SIFT включает детектор особых точек и дескриптор. Дескриптор очень устойчив, и именно благодаря ему SIFT завоевал такую популярность. С момента его появления было предложено много альтернатив, в которых используется практически тот же самый дескриптор. Сегодня он сочетается с самыми разными детекторами особых точек (и областей), а иногда применяется даже ко всему изображению. Признаки, выделяемые методом SIFT, инварианты относительно масштабирования, поворота и изменения яркости. Они надежно сопоставляются даже для проекций трехмерных изображений при разном положении камеры и в присутствии шума. Краткое описание алгоритма можно найти на странице *http://en.wikipedia.org/wiki/Scale-invariant_feature_transform*.

Особые точки

Особые точки SIFT находит с помощью *разностей гауссианов*:

$$D(\mathbf{x}, \sigma) = [G_{k\sigma}(\mathbf{x}) - G_{\sigma}(\mathbf{x})] * I(\mathbf{x}) = [G_{k\sigma} - G_{\sigma}] * I = I_{k\sigma} - I_{\sigma},$$

где G_{σ} – двумерное гауссово ядро, описанное на стр. 39, I_{σ} – результат G_{σ}-размытия полутонового изображения, а k – постоянный множитель, определяющий разделение при масштабировании. Особые точки ищутся в виде минимумов и максимумов функции $D(\mathbf{x}, \sigma)$ от точки изображения и масштаба. Из найденных таким образом кандидатов исключаются неустойчивые точки. Для исключения применяется несколько критериев, в частности, низкая контрастность и по-

падание точки на границу. С деталями можно ознакомиться в тексте оригинальной статьи.

Дескриптор

Описанный выше детектор особых (ключевых) точек дает положение и масштаб. Чтобы добиться инвариантности относительно поворота, выбирается опорное направление, исходя из направления и модуля градиента изображения в окрестности точки. Опорным считается доминирующее направление, которое определяется с помощью гистограммы ориентаций (взвешенной по модулю градиента).

Следующий шаг – вычисление дескриптора на основе положения, масштаба и направления. Чтобы добиться независимости от яркости, в дескрипторе SIFT используются градиенты изображения (сравните с описанной ранее нормированной взаимной корреляцией, где используются яркости изображения). Мы берем сетку подобластей в окрестности точки и для каждой подобласти вычисляем гистограмму ориентаций градиента изображения. Эти гистограммы объединяются, и в результате получается вектор дескриптора. Стандартно берут подобласти размером 4×4 и гистограммы ориентаций с 8 интервалами, так что в итоге получается гистограмма со 128 интервалами ($4 * 4 * 8 = 128$). На рис. 2.3 показано построение дескриптора. Интересующийся читатель может найти детали в работе [18] или в статье википедии по адресу *http://en.wikipedia.org/wiki/Scale-invariant_feature_transform*.

Обнаружение особых точек

Для вычисления SIFT-признаков изображения мы воспользуемся двоичной программой, входящей в состав пакета VLFeat [36] с открытым исходным кодом. Реализация всех шагов алгоритма на чистом Python была бы не очень эффективна, да и в любом случае выходит за рамки книги. Пакет VLFeat можно скачать с сайта *http://www.vlfeat.org/*, где имеются также откомпилированные версии для всех основных платформ. Библиотека написана на C, но к ней имеется командный интерфейс, которым мы и воспользуемся. Существует также интерфейс к Matlab и обертка для Python (*http://github.com/mmmikael/vlfeat/*), если вы предпочитаете пользоваться ими, а не двоичными программами. Установка обертки для Python на некоторых платформах не вполне тривиальна из-за наличия зависимостей, поэтому мы остановимся на исполняемом файле. На сайте Лоуи по адресу *http://*

www.cs.ubc.ca/~lowe/keypoints/ имеется альтернативная реализация SIFT (только для Windows и Linux).

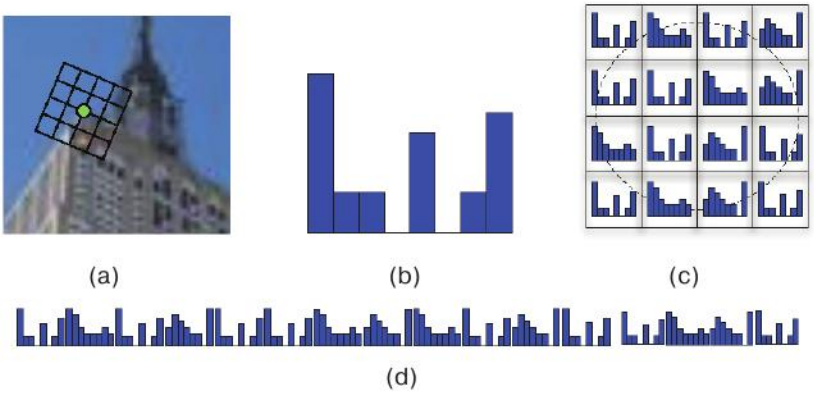


Рис. 2.3. Построение вектора признаков для SIFT-дескриптора: (a) сетка вокруг особой точки, ориентированная вдоль доминирующего направления градиента; (b) 8-интервальная гистограмма ориентаций градиента в части сетки; (c) гистограммы, построенные в каждой ячейке сетки; (d) гистограммы объединяются в один длинный вектор признаков

Создайте файл *sift.py* и добавьте в него функцию, которая вызывает исполняемую программу:

```
def process_image(imagename,resultname,
                  params="--edge-thresh 10 --peak-thresh 5"):
  """ Обработать изображения и сохранить результаты в файле. """

  if imagename[-3:] != 'pgm':
     # создать pgm-файл
     im = Image.open(imagename).convert('L')
     im.save('tmp.pgm')
     imagename = 'tmp.pgm'

     cmmd = str("sift "+imagename+" --output="+resultname+
           " "+params)
     os.system(cmmd)
     print 'processed', imagename, 'to', resultname
```

Программа ожидает получить полутоновое изображение в формате PGM, поэтому если изображение хранится в другом формате, то мы сначала преобразуем его и создаем временный *pgm*-файл. Результат сохраняется в текстовом формате, удобном для чтения. Выглядит это примерно так:

```
318.861 7.48227 1.12001 1.68523 0 0 0 1 0 0 0 0 0 11 16 0 ...
318.861 7.48227 1.12001 2.99965 11 2 0 0 1 0 0 0 173 67 0 0 ...
54.2821 14.8586 0.895827 4.29821 60 46 0 0 0 0 0 0 99 42 0 0 ...
155.714 23.0575 1.10741 1.54095 6 0 0 0 150 11 0 0 150 18 2 1 ...
42.9729 24.2012 0.969313 4.68892 90 29 0 0 0 1 2 10 79 45 5 11 ...
229.037 23.7603 0.921754 1.48754 3 0 0 0 141 31 0 0 141 45 0 0 ...
232.362 24.0091 1.0578 1.65089 11 1 0 16 134 0 0 0 106 21 16 33 ...
201.256 25.5857 1.04879 2.01664 10 4 1 8 14 2 1 9 88 13 0 0 ...
...
```

Здесь первые четыре значения в каждой строке – это координаты, масштаб и угол поворота для одной особой точки, а далее идут 128 значений дескриптора. Дескриптор представлен целыми числами и не нормирован. Именно такое представление нужно для сравнения дескрипторов, но об этом позже.

В примере выше показано начало первых восьми признаков, найденных для изображения. Обратите внимание, что координаты в первых двух строках одинаковы, но углы поворота различаются. Так может случиться, если в одной особой точке обнаружено несколько опорных направлений.

Следующая функция читает признаки из созданного файла в массивы `NumPy`. Добавьте ее в файл *sift.py*:

```
def read_features_from_file(filename):
  """ Прочитать признаки и вернуть их в виде матрицы. """

  f = loadtxt(filename)
  return f[:,:4],f[:,4:] # положения признаков, дескрипторы
```

Всю работу за нас делает функции `loadtxt()` из пакета `NumPy`.

После модификации дескрипторов в сеансе работы с Python полезно записать их обратно в файл. Для этого воспользуемся функцией `savetxt()` из пакета `NumPy`:

```
def write_features_to_file(filename,locs,desc):
  """ Сохранить положения и дескрипторы признаков в файле. """
  savetxt(filename,hstack((locs,desc)))
```

Здесь мы использовали функцию `hstack()`, которая сцепляет два массива по горизонтали, так чтобы в каждой строке дескриптор находился после положения точки.

После того как признаки прочитаны, нанести их положение на график поверх изображения совсем просто. Добавьте приведенную ниже функцию `plot_features()` в файл *sift.py*:

```
def plot_features(im,locs,circle=False):
  """ Показать изображение вместе с признаками. Вход: im
```

```
    (изображение в виде массива), locs (строка, столбец, масштаб,
    ориентация для каждого признака). """

def draw_circle(c,r):
    t = arange(0,1.01,.01)*2*pi
    x = r*cos(t) + c[0]
    y = r*sin(t) + c[1]
    plot(x,y,'b',linewidth=2)

imshow(im)
if circle:
    for p in locs:
        draw_circle(p[:2],p[2])
else:
    plot(locs[:,0],locs[:,1],'ob')
axis('off')
```

Эта функция рисует всех найденные SIFT точки на изображении. Если необязательный параметр *circle* равен «True», то вместо точек с помощью функции `draw_circle()` будут нарисованы окружности радиуса, равного масштабу признака.

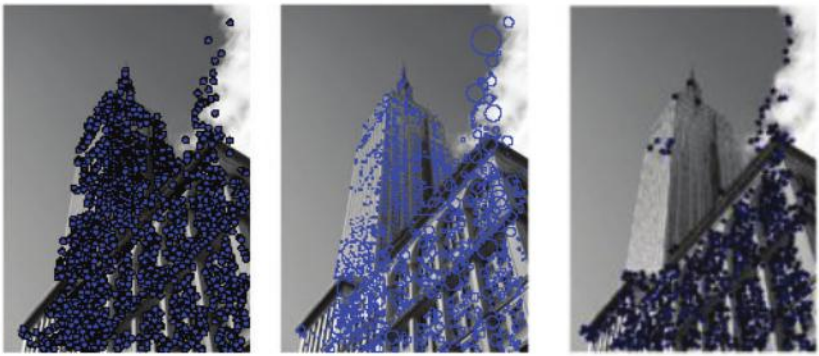


Рис. 2.4. Пример выделения SIFT-признаков из изображения: (a) SIFT-признаки; (b) SIFT-признаки, показанные окружностями, иллюстрирующими масштаб признака; (c) угловые точки Харриса для того же изображения

Показанный ниже код создает график, на котором показаны положения SIFT-признаков (рис. 2.4 (b)).

```
import sift

imname = 'empire.jpg'
```

```
im1 = array(Image.open(imname).convert('L'))
sift.process_image(imname,'empire.sift')
l1,d1 = sift.read_features_from_file('empire.sift')

figure()
gray()
sift.plot_features(im1,l1,circle=True)
show()
```

Для сравнения справа показано то же изображений с угловыми точками Харриса (рис. 2.4 (c)). Как видите, алгоритмы находят разные точки.

Сопоставление дескрипторов

Устойчивый критерий (тоже найденный Лоуи) для сопоставления признаков в двух изображениях – отношение расстояний до двух ближайших соответственных признаков. Тем самым гарантируется, что будут использованы только достаточно сильно различающиеся признаки. Поэтому количество ложных соответствий уменьшается. Ниже приведена функция сопоставления `match()`. Добавьте ее в файл *sift.py*:

```
def match(desc1,desc2):
  """ Для каждого дескриптора в первом изображении найти
    соответствующий ему во втором изображении.
    вход: desc1 (дескрипторы первого изображения),
          desc2 (дескрипторы второго изображения). """

  desc1 = array([d/linalg.norm(d) for d in desc1])
  desc2 = array([d/linalg.norm(d) for d in desc2])

  dist_ratio = 0.6
  desc1_size = desc1.shape
  matchscores = zeros((desc1_size[0],1),'int')
  desc2t = desc2.T # вычислить транспонированную матрицу
  for i in range(desc1_size[0]):
     dotprods = dot(desc1[i,:],desc2t) # вектор скалярных
                                       # произведений
     dotprods = 0.9999*dotprods
     # вычислить арккосинусы и отсортировать, получив в результате
     # индекс признаков во втором изображении
     indx = argsort(arccos(dotprods))

     # проверить, верно ли, что для ближайшего соседа угол меньше,
     # чем dist_ratio, умноженное на угол для следующего соседа
     if arccos(dotprods)[indx[0]] < dist_ratio * arccos(dotprods)[indx[1]]:
        matchscores[i] = int(indx[0])

  return matchscores
```

В этой функции в качестве меры расстояния используется угол между векторами дескрипторов. Но это имеет смысл только после нормирования векторов на единичную длину[2]. Поскольку сопоставление производится в одну сторону, т. е. мы сравниваем каждый признак одного изображения со всеми признаками другого, то можно заранее вычислить транспонированную матрицу, содержащую векторы признаков второго изображения, чтобы не выполнять одну и ту же операцию для каждого признака.

Чтобы еще повысить устойчивость результатов, мы можем инвертировать процедуру и провести сопоставление в другом направлении (сравнить признаки второго изображения с признаками первого), оставив только те соответствия, которые прошли обе проверки (как для точек Харриса). Это делает функция `match_twosided()`:

```
def match_twosided(desc1,desc2):
  """ Двусторонняя симметричная версия match(). """

  matches_12 = match(desc1,desc2)
  matches_21 = match(desc2,desc1)

  ndx_12 = matches_12.nonzero()[0]

  # удалить несимметричные соответствия
  for n in ndx_12:
     if matches_21[int(matches_12[n])] != n:
        matches_12[n] = 0

  return matches_12
```

Чтобы нанести соответствия на график, мы можем использовать функции из файла *harris.py*. Для удобства скопируйте функции `appendimages()` и `plot_matches()` в файл *sift.py*. При желании можете вместо этого импортировать также файл *harris.py* и брать эти функции из него.

На рис. 2.5 и 2.6 показаны примеры SIFT-признаков, найденных в двух изображениях, а также соответственные точки, возвращенные функцией `match_twosided()`.

На рис. 2.7 показаны результаты сопоставления признаков с помощью функций `match()` и `match_twosided()`. Видно, что симметричное (двустороннее) сопоставление позволило исключить неправильные соответствия, оставив хорошие (впрочем, некоторые правильные соответствия также удалены).

[2] Для единичных векторов скалярное произведение (без `arccos()`) эквивалентно обычному евклидову расстоянию.

Располагая средствами обнаружения и сопоставления особых точек, мы имеем все необходимое, чтобы применить локальные дескрипторы к решению разнообразных задач. В двух следующих главах мы добавим геометрические ограничения на соответствие, что позволит надежно отфильтровывать неправильно сопоставленные точки и применять локальные дескрипторы к таким задачам, как автоматическое создание панорам, оценка положения камеры и вычисление трехмерной структуры.

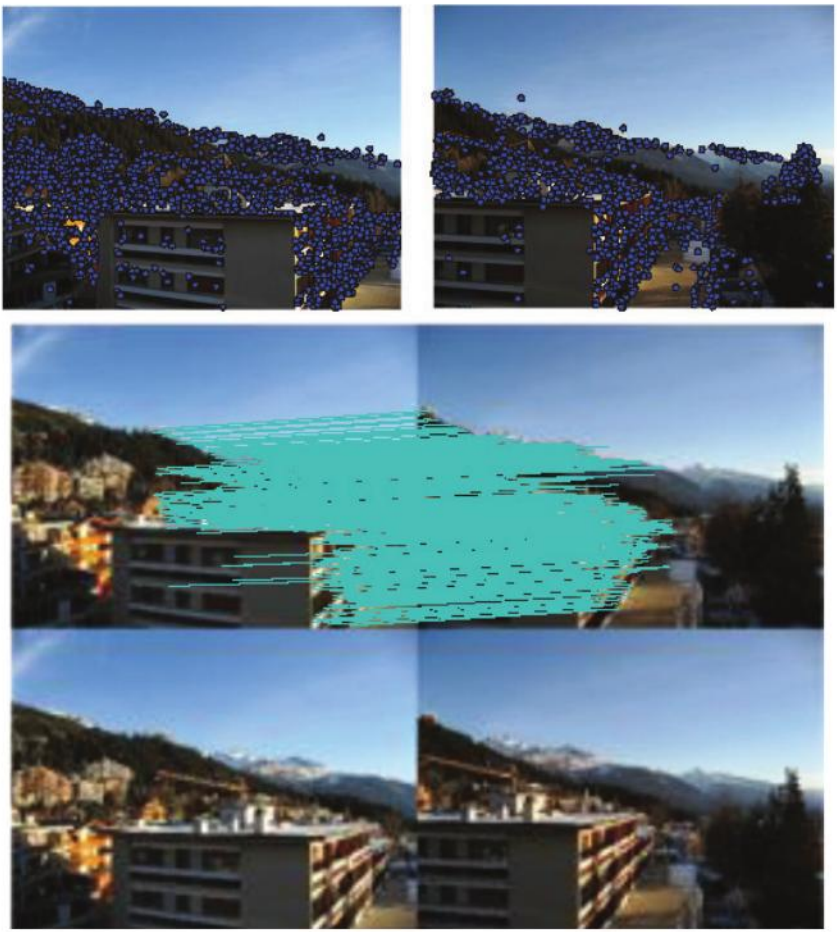


Рис. 2.5. Пример обнаружения и сопоставления SIFT-признаков в двух изображениях

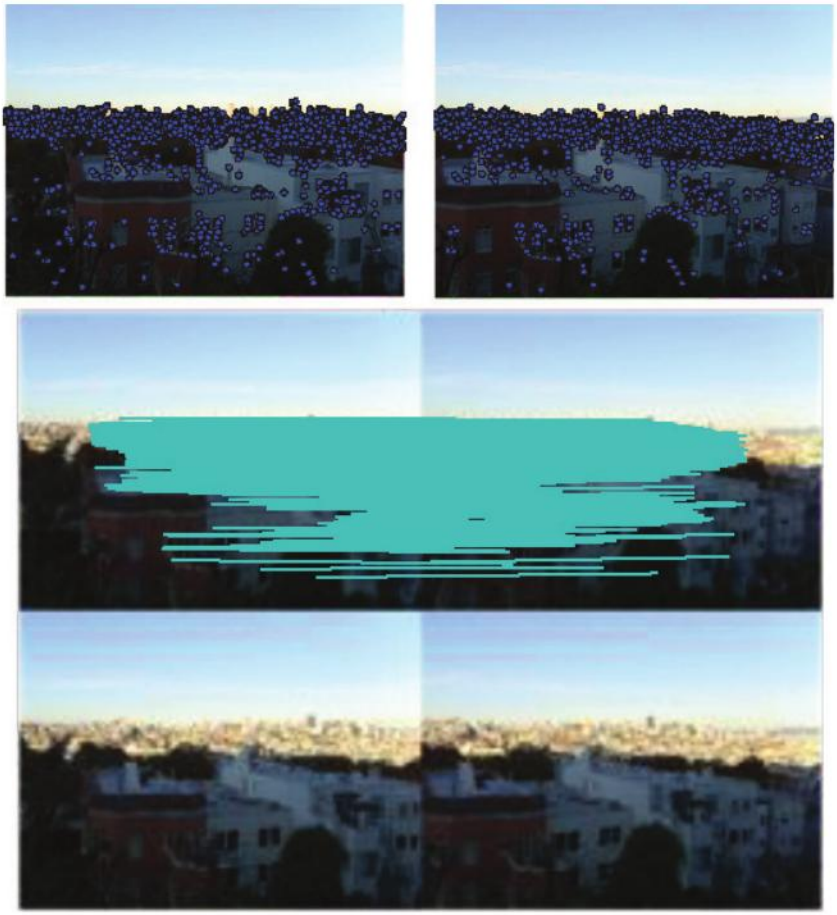
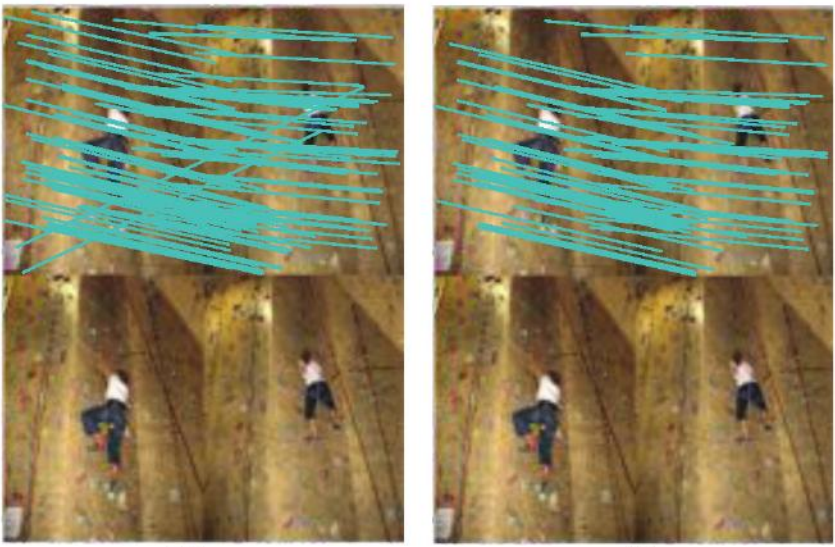


Рис. 2.6. Пример обнаружения и сопоставления SIFT-признаков в двух изображениях

2.3. Сопоставление изображений с геометками

Завершая эту главу, мы рассмотрим применение локальных дескрипторов для сравнения изображения с геометками.



(a) (b)

Рис. 2.7. Пример сопоставления SIFT-признаков в двух изображениях: (a) признаки из левого изображения сопоставляются с признаками из правого, но не наоборот; (b) соответствия, оставшиеся после применения двустороннего сопоставления

Загрузка изображений с геометками из Panoramio

Один из источников изображений с геометками – служба обмена фотографиями Panoramio (*http://www.panoramio.com/*), принадлежащая Google. Как у многих веб-служб, у Panoramio имеется API для доступа к содержимому из программы. API простой, его описание можно найти по адресу *http://www.panoramio.com/api/*. Отправив GET-запрос на URL-адрес вида

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity
&set=public&from=0&to=20&minx=-180&miny=-90&maxx=180&maxy=90&
size=medium
```

где *minx*, *miny*, *maxx*, *maxy* определяют территорию, к которой относятся геометки отбираемых фотографий (минимальная и максимальная широта и долгота соответственно), мы получим ответ в виде легко разбираемого JSON-документа. JSON – это стандартный формат обмена данными между службами, более простой, чем XML и другие

альтернативы. Подробнее прочитать об этом формате можно на странице *http://en.wikipedia.org/wiki/JSON*.

Интересное место с двумя разными видами – Белый дом в Вашингтоне, который обычно фотографируют с Пенсильвания-авеню с южной или северной стороны. Его координаты (широта, долгота):

```
lt=38.897661
ln=-77.036564
```

Для преобразования их в формат, необходимый для вызова API, нужно для каждой координаты прибавить и вычесть какое-нибудь число, чтобы получить квадрат вокруг Белого дома. В ответ на обращение

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&
set=public&from=0&to=20&minx=-77.037564&miny=38.896662&maxx=
-77.035564&maxy=38.898662&size=medium
```

будут возвращены первые 20 изображений в квадрате с разбросом координат ±0.001, упорядоченные по популярности. Ответ выглядит следующим образом:

```
{"count": 349,
"photos": [{"photo_id": 7715073, "photo_title": "White House", "photo_url":
"http://www.panoramio.com/photo/7715073", "photo_file_url":
"http://mw2.google.com/mw-panoramio/photos/medium/7715073.jpg",
"longitude":
-77.036583, "latitude": 38.897488, "width": 500, "height": 375, "upload_date":
"10 February 2008", "owner_id": 1213603, "owner_name": "***", "owner_url":
"http://www.panoramio.com/user/1213603"}
,
{"photo_id": 1303971, "photo_title": "White House balcony", "photo_url":
"http://www.panoramio.com/photo/1303971", "photo_file_url":
"http://mw2.google.com/mw-panoramio/photos/medium/1303971.jpg",
"longitude":
-77.036353, "latitude": 38.897471, "width": 500, "height": 336, "upload_date":
"13 March 2007", "owner_id": 195000, "owner_name": "***", "owner_url":
"http://www.panoramio.com/user/195000"}
…
]}
```

Чтобы разобрать этот ответ, мы воспользуемся пакетом `simplejson`, который можно скачать со страницы *http://github.com/simplejson/simplejson*. На этой же странице имеется документация.

Если вы работаете с версией Python 2.6 или более поздней, то использовать `simplejson` нет необходимости, потому что в дистрибутив уже включена библиотека для работы с JSON, которая импортируется следующим образом:

```
import json
```

Если вы хотите использовать `simplejson`, когда она доступна (эта библиотека быстрее и содержит больше возможностей, чем встроенная), то имеет смысл попытаться импортировать ее, но переключиться на встроенную в случае ошибки:

```
try: import simplejson as json
except ImportError: import json
```

В показанной ниже программе используется входящий в состав Python пакет `urllib` для обработки запросов, а затем полученный ответ разбирается с помощью `simplejson`:

```
import os
import urllib, urlparse
import simplejson as json

# запросить изображения
url = 'http://www.panoramio.com/map/get_panoramas.php?order=popularity&\
  set=public&from=0&to=20&minx=-77.037564&miny=38.896662&\
  maxx=-77.035564&maxy=38.898662&size=medium'
c = urllib.urlopen(url)

# выделить из JSON URL-адреса изображений
j = json.loads(c.read())
imurls = []
for im in j['photos']:
  imurls.append(im['photo_file_url'])

# загрузить изображения
for url in imurls:
  image = urllib.URLopener()
  image.retrieve(url, os.path.basename(urlparse.urlparse(url).path))
  print 'загружается:', url
```

Одного взгляда на JSON-документ достаточно, чтобы понять, что нас интересует поле «photo_file_url». В ходе выполнения показанной выше программы на консоли будут печататься строки такого вида:

```
загружается: http://mw2.google.com/mw-panoramio/photos/medium/7715073.jpg
загружается: http://mw2.google.com/mw-panoramio/photos/medium/1303971.jpg
загружается: http://mw2.google.com/mw-panoramio/photos/medium/270077.jpg
загружается: http://mw2.google.com/mw-panoramio/photos/medium/15502.jpg
...
```

На рис. 2.8 показано 20 изображений, загруженных этой программой. Теперь мы хотим для каждой пары изображений найти и сопоставить признаки.

Рис. 2.8. Фотографии, снятые в одном и том же месте земного шара (квадратная область с центром в Белом доме). Загружены с сайта *panoramio.com*.

Сопоставление с помощью локальных дескрипторов

Загрузив изображения, мы должны извлечь из них локальные дескрипторы. В данном случае будем использовать SIFT-дескрипторы, описанные в предыдущем разделе. Предположим, что изображение обработано по алгоритму SIFT и признаки сохранены в файле с таким же именем, но расширением «.sift», а не «.jpg». Будем считать, что списки *imlist* и *featlist* содержат имена файлов. Вот как можно произвести попарное сравнение изображений:

```
import sift

nbr_images = len(imlist)

matchscores = zeros((nbr_images,nbr_images))

for i in range(nbr_images):
  for j in range(i,nbr_images): # берем только верхний треугольник
     print 'сравниваются ', imlist[i], imlist[j]

     l1,d1 = sift.read_features_from_file(featlist[i])
```

```
        l2,d2 = sift.read_features_from_file(featlist[j])

        matches = sift.match_twosided(d1,d2)

        nbr_matches = sum(matches > 0)
        print 'число соответствий = ', nbr_matches
        matchscores[i,j] = nbr_matches

# копировать значения
for i in range(nbr_images):
  for j in range(i+1,nbr_images): # копировать диагональ не имеет смысла
      matchscores[j,i] = matchscores[i,j]
```

Число совпавших признаков для каждой пары мы сохраняем в массиве *matchscores*. Последний этап – заполнение нижнего треугольника матрицы – совершенно необязателен, потому что наша «метрика» симметрична, но так смотрится лучше. Матрица *matchscores* для этих конкретных изображений выглядит следующим образом:

```
662 0 0 2 0 0 0 0 1 0 0 1 2 0 3 0 19 1 0 2
0 901 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 2
0 0 266 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
2 1 0 1481 0 0 2 2 0 0 0 2 2 0 0 0 2 3 2 0
0 0 0 0 1748 0 0 1 0 0 0 0 0 2 0 0 0 0 0 1
0 0 0 0 0 1747 0 0 1 0 0 0 0 0 0 0 0 1 1 0
0 0 0 2 0 0 555 0 0 0 1 4 4 0 2 0 0 5 1 0
0 1 0 2 1 0 0 2206 0 0 0 1 0 0 1 0 2 0 1 1
1 1 0 0 0 1 0 0 629 0 0 0 0 0 0 0 1 0 0 20
0 0 0 0 0 0 0 0 0 829 0 0 1 0 0 0 0 0 0 2
0 0 0 0 0 0 1 0 0 0 1025 0 0 0 0 0 1 1 1 0
1 1 0 2 0 0 4 1 0 0 0 528 5 2 15 0 3 6 0 0
2 0 0 2 0 0 4 0 0 1 0 5 736 1 4 0 3 37 1 0
0 0 1 0 2 0 0 0 0 0 0 2 1 620 1 0 0 1 0 0
3 0 0 0 0 0 2 1 0 0 0 15 4 1 553 0 6 9 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2273 0 1 0 0
19 0 0 2 0 0 0 2 1 0 1 3 3 0 6 0 542 0 0 0
1 0 0 3 0 1 5 0 0 0 1 6 37 1 9 1 0 527 3 0
0 1 0 2 0 1 1 1 0 0 1 0 1 0 1 0 0 3 1139 0
2 2 0 0 1 0 0 1 20 2 0 0 0 0 0 0 0 0 0 499
```

Используя ее как простую меру расстояния между изображениями (для похожих изображений число совпавших признаков должно быть больше, чем для непохожих), мы можем связать изображения со схожим содержанием.

Визуализация связанных изображений

Теперь визуализируем связи между изображениями с сопоставившимися локальными дескрипторами. Для этого можно построить граф, ребра которого обозначают связи. Мы воспользуемся пакетом `pydot` (*http://code.google.com/p/pydot/*) – интерфейсом между Python и мощной библиотекой работы с графами GraphViz. Pydot нуждается в пакетах Pyparsing (*http://pyparsing.wikispaces.com/*) и GraphViz (*http://www.graphviz.org/*), но не переживайте – их установка займет всего несколько минут.

Работать с Pydot очень просто. В показанном ниже фрагменте показано, как создать дерево глубины 2 с коэффициентом ветвления 5 и пронумеровать узлы. Получившийся граф представлен на рис. 2.9. Есть много способов настроить расположение и внешний вид графа. Дополнительные сведения см. в документации по Pydot или в описании языка DOT, используемого в GraphViz, на странице *http://www.graphviz.org/Documentation.php*.

```
import pydot
g = pydot.Dot(graph_type='graph')
g.add_node(pydot.Node(str(0),fontcolor='transparent'))
for i in range(5):
  g.add_node(pydot.Node(str(i+1)))
  g.add_edge(pydot.Edge(str(0),str(i+1)))
  for j in range(5):
     g.add_node(pydot.Node(str(j+1)+'-'+str(i+1)))
     g.add_edge(pydot.Edge(str(j+1)+'-'+str(i+1),str(j+1)))
  g.write_png('graph.jpg',prog='neato')
```

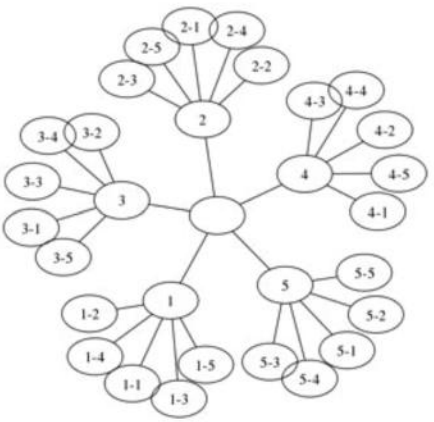


Рис. 2.9. Пример использования pydot для создания графа

Но вернемся к изображениям с геометками. Чтобы построить граф, иллюстрирующий потенциальные группы изображений, будем создавать ребро между узлами, представляющими изображения, если количество найденных соответствий превышает пороговое значение. Для включения в граф самих изображений нам понадобится полный путь к файлу (в примере ниже он представлен переменной *path*). Чтобы результат выглядел красиво, создадим для каждого изображения масштабную миниатюру с длиной наибольшей стороны 100 пикселей. Вот как это делается.

```
import pydot

threshold = 2 # мин. число соответствий для создания ребра

g = pydot.Dot(graph_type='graph') # не нужно строить ориентированный
                                  # граф, как предполагается по умолчанию
for i in range(nbr_images):
  for j in range(i+1,nbr_images):
     if matchscores[i,j] > threshold:
        # первое изображение в паре
        im = Image.open(imlist[i])
        im.thumbnail((100,100))
        filename = str(i)+'.png'
        im.save(filename) # нужны временные файлы подходящего размера
        g.add_node(pydot.Node(str(i),fontcolor='transparent',
             shape='rectangle',image=path+filename))

        # второе изображение в паре
        im = Image.open(imlist[j])
        im.thumbnail((100,100))
        filename = str(j)+'.png'
        im.save(filename) # нужны временные файлы подходящего размера
        g.add_node(pydot.Node(str(j),fontcolor='transparent',
             shape='rectangle',image=path+filename))

        g.add_edge(pydot.Edge(str(i),str(j)))

g.write_png('whitehouse.png')
```

Результат будет выглядеть, как на рис. 2.10, но, естественно, зависит от того, какие изображения загружены. В данном случае мы видим две группы изображений по одной для каждого ракурса Белого дома.

Это очень простой пример использования локальных дескрипторов для сопоставления областей изображений. В частности, мы не производили никакую проверку найденных соответствий.

Но это можно сделать (и весьма надежно), воспользовавшись идеями, которые мы изложим в двух следующих главах.

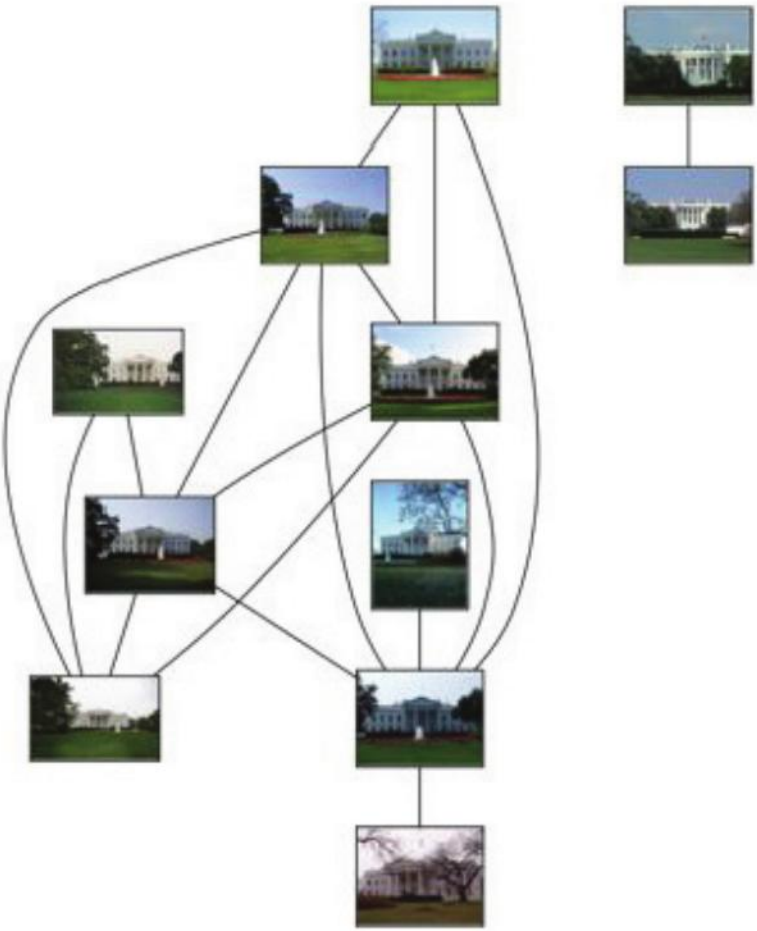


Рис. 2.10. Пример группировки снятых в одном месте фотографий с помощью локальных дескрипторов

Упражнения

1. Модифицируйте функцию сопоставления угловых точек Харриса, так чтобы она признавала точки соответственными, только если расстояние между ними в пикселях не больше порогового значения. Идея в том, чтобы повысить надежность сопоставления.

2. Постепенно повышайте степень размытия изображения (или применяйте очистку от шумов по методу ROF) и на каждой итерации находите углы Харриса. Что происходит?
3. Альтернативой методу Харриса является детектор углов FAST. Для него существует несколько реализаций, в т. ч. на чистом Python (*http://www.edwardrosten.com/work/fast.html*). Поэкспериментируйте с этим детектором, задавая различные пороги чувствительности, и сравните найденные углы с теми, что обнаруживает наша реализация детектора Харриса.
4. Создайте копии изображения с разной разрешающей способностью (например, несколько раз уменьшив размер вдвое). Выделите из каждого изображения SIFT-признаки. Нанесите признаки на график и проведите их сопоставление, чтобы понять, когда и как нарушается независимость от масштаба.
5. Среди командных утилит VLFeat имеется также реализация детектора MSER (Maximally Stable Extremal Regions – максимально устойчивые области экстремума) (*http://en.wikipedia.org/wiki/Maximally_stable_extremal_regions*), который ищет в изображении «блобы»[3]. Напишите функцию, которая находит MSER-области и передает их части алгоритма SIFT, строящей дескрипторы. Для этого воспользуйтесь параметром `--read-frames` и функцией нанесения на график эллиптических областей.
6. Напишите функцию, которая сопоставляет признаки в двух изображениях и, исходя из найденных соответствий, оценивает различие масштабов и угол поворота.
7. Загрузите фотографии, снятые в интересном вам месте, и произведите их сопоставление, как в примере с Белым домом. Можете ли вы найти лучший критерий связывания изображений? Как можно было бы использовать граф для выборки репрезентативных изображений места?

[3] Неформально «блобом» называется область изображения, в которой некоторые свойства постоянны или почти постоянны. Можно считать, что все точки блоба в некотором смысле похожи друг на друга. – *Прим. перев.*

ГЛАВА 3. Преобразования изображений

В этой главе описываются преобразования изображений и некоторые практические методы их вычисления. Преобразования используются, например, для деформирования и регистрации изображений. Кроме того, мы рассмотрим в качестве примера автоматическое создание панорам.

3.1. Гомографии

Гомографией называется проективное преобразование одной плоскости в другую. В нашем случае плоскостями являются изображения или плоские поверхности в трехмерном пространстве. У гомографии много практических применений, в том числе регистрация изображений, ректификация изображений, деформирование текстур и создание панорам. Мы часто будем пользоваться этим инструментом. По существу, гомография H отображает точки плоскости (в однородных координатах) согласно следующему уравнению:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \text{или} \quad \mathbf{x}' = H\mathbf{x}.$$

Однородные координаты – полезное представление точек в плоскости изображения (и в трехмерном пространстве, как мы увидим далее). В однородных координатах точка определена лишь с точностью до масштабного коэффициента, т. е. координаты $\mathbf{x} = [x, y, w] = [\alpha x, \alpha y, \alpha w] = [x/w, y/w, 1]$ определяют одну и ту же точку на плоскости. Следовательно, гомография H определена с точностью до масштаба и имеет восемь независимых степеней свободы. Часто производится

нормировка: полагают $w = 1$ и тем самым получают однозначное представление точки изображения с координатами x, y. Дополнительная координата упрощает представление преобразования одной матрицей.

Создайте файл *homography.py* и поместите в него следующие функции нормировки и преобразования в однородные координаты:

```
def normalize(points):
  """ Нормировать коллекцию точек в однородных координатах,
     так чтобы последняя строка была равна 1. """

  for row in points:
     row /= points[-1]
  return points

def make_homog(points):
  """ Преобразовать множество точек (массив dim*n) в
     однородные координаты. """

  return vstack((points,ones((1,points.shape[1]))))
```

При работе с преобразованиями мы будем хранить точки по столбцам, так что множество n точек на плоскости представляется массивом $3 \times n$ в однородных координатах. При таком представлении упрощается умножение матриц и преобразование точек. В остальных случаях данные, например признаки для кластеризации и классификации, обычно хранятся по строкам.

Существует несколько важных частных случаев таких проективных преобразований. *Аффинное преобразование*

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & t_x \\ a_3 & a_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{или} \quad \mathrm{x}' = \begin{bmatrix} A & \mathrm{t} \\ 0 & 1 \end{bmatrix} \mathrm{x},$$

сохраняет условие $w = 1$, но в отличие от проективного преобразования общего вида не позволяет описать сильные деформации. Аффинное преобразование распадается на обратимую матрицу A и вектор параллельного переноса $\mathbf{t} = [t_x, t_y]$. Такие преобразования применяются, например, для деформирования изображений.

Преобразования подобия

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s\cos(\theta) & -s\sin(\theta) & t_x \\ s\sin(\theta) & s\cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{или} \quad \mathrm{x}' = \begin{bmatrix} sR & \mathrm{t} \\ 0 & 1 \end{bmatrix} \mathrm{x},$$

включают изометрические преобразования и операции масштабирования. Скаляр s определяет коэффициент масштабирования, R описывает поворот на угол θ, а $\mathbf{t} = [t_x, t_y]$ снова является вектором параллельного переноса. При $s = 1$ расстояния между точками сохраняются, так что мы получаем *изометрическое преобразование*. Преобразования подобия применяются, например, при регистрации изображений.

Сначала рассмотрим алгоритмы вычисления гомографии, а затем перейдем к примерам использования аффинных преобразований для деформирования, преобразований подобия для регистрации и проективных преобразований общего вида для создания панорам.

Алгоритм прямого линейного преобразования

Гомографию можно вычислить непосредственно, зная соответственные точки двух изображений (или плоскостей). Как уже было сказано, у проективного преобразования общего вида восемь степеней свободы. Каждая пара соответственных точек дает два уравнения, по одному для координат x и y. Следовательно, для вычисления H необходимы четыре пары точек.

Прямое линейное преобразование (direct linear transformation, *DLT*) – это алгоритм вычисления H по четырем или более парам соответственных точек. Если переписать уравнение проективного преобразования, подставив в него несколько соответствий, то получим уравнение вида:

$$\begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' & x_1' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' & y_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' & x_2' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' & y_2' \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix}$$

или $A\mathbf{h} = \mathbf{0}$, где A – матрица, в которой число строк в два раза больше числа соответственных точек. Из этого уравнения можно найти зна-

чение H, минимизирующее среднеквадратичную ошибку, применив метод сингулярного разложения (SVD). Ниже приведена соответствующая функция, добавьте ее в файл *homography.py*:

```
def H_from_points(fp,tp):
  """ Найти гомографию H, отображающую fp в tp, методом DLT.
     Хорошая обусловленность обеспечивается автоматически. """

  if fp.shape != tp.shape:
     raise RuntimeError('расхождение в количестве точек')

  # обеспечить хорошую обусловленность точек (важно для численных расчетов)
  # --точки первого изображения--
  m = mean(fp[:2], axis=1)
  maxstd = max(std(fp[:2], axis=1)) + 1e-9
  C1 = diag([1/maxstd, 1/maxstd, 1])
  C1[0][2] = -m[0]/maxstd
  C1[1][2] = -m[1]/maxstd
  fp = dot(C1,fp)

  # --точки второго изображения--
  m = mean(tp[:2], axis=1)
  maxstd = max(std(tp[:2], axis=1)) + 1e-9
  C2 = diag([1/maxstd, 1/maxstd, 1])
  C2[0][2] = -m[0]/maxstd
  C2[1][2] = -m[1]/maxstd
  tp = dot(C2,tp)

  # создать матрицу для метода DLT, по две строки для каждой пары
  # соответственных точек
  nbr_correspondences = fp.shape[1]
  A = zeros((2*nbr_correspondences,9))
  for i in range(nbr_correspondences):
     A[2*i] = [-fp[0][i],-fp[1][i],-1,0,0,0,
        tp[0][i]*fp[0][i],tp[0][i]*fp[1][i],tp[0][i]]
     A[2*i+1] = [0,0,0,-fp[0][i],-fp[1][i],-1,
        tp[1][i]*fp[0][i],tp[1][i]*fp[1][i],tp[1][i]]

  U,S,V = linalg.svd(A)
  H = V[8].reshape((3,3))

  # обратное преобразование, компенсирующее обусловливание
  H = dot(linalg.inv(C2),dot(H,C1))

  # нормировать и вернуть
  return H / H[2,2]
```

Первым делом мы проверяем, что оба массива точек содержат одинаковое число элементов. Если это не так, возбуждаем исключение. Вообще говоря, всегда нужно проверять входные параметры для обе-

спечения надежности программы, но в этой книге мы пользуемся исключениями очень редко, чтобы не загромождать код. Подробнее о типах исключений можно прочитать на странице *http://docs.python.org/library/exceptions.html*, а о том, как ими пользоваться, – на странице *http://docs.python.org/tutorial/errors.html*.

Для обеспечения хорошей обусловленности точек мы нормируем их, так чтобы среднее было равно нулю, а стандартное отклонение – единице. Это очень важно с точки зрения вычислений, потому что от представления координат зависит устойчивость алгоритма. Затем по имеющимся соответствиям создаем матрицу A. Решение с наименьшей среднеквадратичной ошибкой будет в последней строке матрицы V сингулярного разложения. Это строка преобразуется обратно в матрицу H размерности 3×3. К матрице применяется преобразование, компенсирующее выполненное в начале обусловливание, затем она нормируется и возвращается.

Аффинные преобразования

У аффинного преобразования шесть степеней свободы, поэтому для вычисления H нужны три пары соответственных точек. Чтобы вычислить аффинное преобразование с помощью алгоритма DLT, нужно приравнять нулю последние два элемента: $h_7 = h_8 = 0$.

Но мы применим другой подход, подробно описанный в книге [13, стр. 130]. Добавьте в файл *homography.py* следующую функцию, которая вычисляет матрицу аффинного преобразования по парам соответственных точек.

```
def Haffine_from_points(fp,tp):
  """ Найти гомографию H, отображающую fp в tp. """
  if fp.shape != tp.shape:
     raise RuntimeError('расхождение в количестве точек')

  # обеспечить хорошую обусловленность точек
  # --точки первого изображения--
  m = mean(fp[:2], axis=1)
  maxstd = max(std(fp[:2], axis=1)) + 1e-9
  C1 = diag([1/maxstd, 1/maxstd, 1])
  C1[0][2] = -m[0]/maxstd
  C1[1][2] = -m[1]/maxstd
  fp_cond = dot(C1,fp)

  # --точки второго изображения--
  m = mean(tp[:2], axis=1)
  C2 = C1.copy() # в обоих наборах точек масштаб должен быть одинаков
  C2[0][2] = -m[0]/maxstd
```

```
    C2[1][2] = -m[1]/maxstd
    tp_cond = dot(C2,tp)

    # после обусловливания среднее точек равно 0, поэтому вектор
    # параллельного переноса нулевой
    A = concatenate((fp_cond[:2],tp_cond[:2]), axis=0)
    U,S,V = linalg.svd(A.T)

    # Создать матрицы B и C, как описано у Хартли и Циммермана
    # (второе издание), на стр. 130.
    tmp = V[:2].T
    B = tmp[:2]
    C = tmp[2:4]

    tmp2 = concatenate((dot(C,linalg.pinv(B)),zeros((2,1))), axis=1)
    H = vstack((tmp2,[0,0,1]))

    # обращение обусловливания
    H = dot(linalg.inv(C2),dot(H,C1))

    return H / H[2,2]
```

Как и в алгоритме DLT, точки подвергаются обусловливанию и обратному преобразованию. Посмотрим, что можно сделать с изображением с помощью аффинного преобразования.

3.2. Деформирование изображений

Применение матрицы аффинного преобразования H к блокам изображения называется *деформированием* (или *аффинным деформированием*). Эта операция часто применяется в машинной графике, а также в некоторых алгоритмах компьютерного зрения. Ее легко выполнить с помощью пакета `ndimage`, входящего в библиотеку `SciPy`. Команда

```
transformed_im = ndimage.affine_transform(im,A,b,size)
```

преобразует блок изображения `im`, применяя к нему линейное преобразование A и вектор параллельного переноса $\mathbf{b}$, как описано выше. Необязательный аргумент *size* позволяет задать размер выходного изображения.

По умолчанию результирующее изображение имеет такой же размер, как исходное. Чтобы посмотреть, как работает эта функция, выполните следующие команды:

```
from scipy import ndimage

im = array(Image.open('empire.jpg').convert('L'))
H = array([[1.4,0.05,-100],[0.05,1.5,-100],[0,0,1]])
im2 = ndimage.affine_transform(im,H[:2,:2],(H[0,2],H[1,2]))

figure()
gray()
imshow(im2)
show()
```

В результате получается изображение, показанное справа на рис. 3.1. Как видите, вместо отсутствующих пикселей в результирующее изображение добавлены нули.

Рис. 3.1. Пример деформирования изображения с помощью аффинного преобразования: справа показано изображение, получающееся в результате применения к левому функции `ndimage.affine_transform()`

Изображение внутри изображения

В качестве простого примера аффинного деформирования рассмотрим размещение изображений или их частей внутри другого изображения с выравниванием по определенным областям или опорным точкам.

Добавьте функцию `image_in_image()` в файл *warp.py*. Эта функция принимает два изображения и координаты углов области внутри второго изображения, куда следует поместить первое изображение.

```
def image_in_image(im1,im2,tp):
```

```
    """ Поместить im1 в im2 с помощью аффинного преобразования,
      так чтобы углы были как можно ближе к tp.
      Углы tp выражены в однородных координатах и перечисляются
      против часовой стрелки, начиная в левого верхнего. """

    # точки деформируемого изображения
    m,n = im1.shape[:2]
    fp = array([[0,m,m,0],[0,0,n,n],[1,1,1,1]])

    # вычислить и применить аффинное преобразование
    H = homography.Haffine_from_points(tp,fp)
    im1_t = ndimage.affine_transform(im1,H[:2,:2],
              (H[0,2],H[1,2]),im2.shape[:2])
    alpha = (im1_t > 0)

    return (1-alpha)*im2 + alpha*im1_t
```

Как видите, работы оказалось совсем немного. Смешивая два изображения, мы создаем *альфа-отображение*, которое определяет, какую часть цвета пикселя брать из каждого изображения. Здесь для создания бинарного альфа-отображения мы пользуемся тем фактом, что деформированное изображение заполнено нулями вне границ области совмещения. Для большей строгости можно было бы прибавить небольшое число к потенциально нулевым пикселям первого изображения, или вообще сделать все корректно (см. упражнения в конце главы). Обратите внимание, что точки изображения описываются в однородных координатах.

Рис. 3.2. Пример вставки одного изображения внутрь другого с помощью аффинного преобразования

Для тестирования этой функции поместим изображение на рекламный щит, находящийся в другом изображении. Показанный ниже код вставляет левое изображение на рис. 3.2 во второе изображение. Координаты были определены вручную путем изучения графика изо-

бражения (в PyLab в нижней части окна рисунка показываются координаты мыши). Разумеется, можно было бы также воспользоваться функцией PyLab ginput().

```
import warp

# пример аффинного деформирования im1 в im2
im1 = array(Image.open('beatles.jpg').convert('L'))
im2 = array(Image.open('billboard_for_rent.jpg').convert('L'))

# задать точки
tp = array([[264,538,540,264],[40,36,605,605],[1,1,1,1]])
im3 = warp.image_in_image(im1,im2,tp)

figure()
gray()
imshow(im3)
axis('equal')
axis('off')
show()
```

В результате изображение помещается в верхнюю часть рекламного щита. Снова отметим, что опорная точка *tp* выражена в однородных координатах. Если задать такие координаты

```
tp = array([[675,826,826,677],[55,52,281,277],[1,1,1,1]])
```

то изображение окажется в левом нижнем углу, занятом сейчас объявлением «сдается» (for rent).

Функция Haffine_from_points() находит наилучшее аффинное преобразование для заданных пар соответственных точек. В примере выше это были угловые точки изображения и углы области внутри рекламного щита. Если эффект перспективы мал, то это дает хорошие результаты. В верхней строке на рис. 3.3 показано, что произойдет, если попытаться с помощью аффинного преобразования поместить изображение на рекламный щит с большим эффектом перспективы. Невозможно с помощью одного аффинного преобразования совместить все четыре угловые точки (но можно было бы подобрать проективное преобразование общего вида). Если мы все же хотим воспользоваться аффинным деформированием, так чтобы совместить все четыре угла, то можно применить полезный прием.

Три пары точек с помощью аффинного преобразования можно совместить точно, поскольку у такого преобразования шесть степеней свободы, а три пары соответствий дают ровно шесть ограничений (уравнения для трех пар координат x и y). Поэтому, если мы хотим,

чтобы изображение точно заполнило весь рекламный щит, то можем разбить его на два треугольника и деформировать их по отдельности. Ниже приведен код.

```
# сопоставить исходные точки углам im1
m,n = im1.shape[:2]
fp = array([[0,m,m,0],[0,0,n,n],[1,1,1,1]])

# первый треугольник
tp2 = tp[:,:3]
fp2 = fp[:,:3]

# вычислить H
H = homography.Haffine_from_points(tp2,fp2)
im1_t = ndimage.affine_transform(im1,H[:2,:2],
          (H[0,2],H[1,2]),im2.shape[:2])

# альфа-отображение для треугольника
alpha = warp.alpha_for_triangle(tp2,im2.shape[0],im2.shape[1])
im3 = (1-alpha)*im2 + alpha*im1_t

# второй треугольник
tp2 = tp[:,[0,2,3]]
fp2 = fp[:,[0,2,3]]

# вычислить H
H = homography.Haffine_from_points(tp2,fp2)
im1_t = ndimage.affine_transform(im1,H[:2,:2],
         (H[0,2],H[1,2]),im2.shape[:2])

# альфа-отображение для треугольника
alpha = warp.alpha_for_triangle(tp2,im2.shape[0],im2.shape[1])
im4 = (1-alpha)*im3 + alpha*im1_t

figure()
gray()
imshow(im4)
axis('equal')
axis('off')
show()
```

Здесь мы просто создаем альфа-отображение для каждого треугольника, а затем объединяем все изображения. Чтобы вычислить альфа-отображение для треугольника, мы проверяем, можно ли записать координаты пикселя в виде выпуклой комбинации угловых точек треугольника[1]. Если да, то пиксель лежит внутри треугольника.

[1] *Выпуклой комбинацией* называется линейная комбинация вида $\sum_i \alpha_i \mathbf{x}_i$ (в данном случае углов треугольника) такая, что все коэффициенты α_i неотрицательны, а их сумма равна 1.

Добавьте функцию `alpha_for_triangle()`, использованную в предыдущем коде, в файл *warp.py*:

```
def alpha_for_triangle(points,m,n):
  """ Создает альфа-отображение размера (m,n)
     для треугольника, определенного своими вершинами
     (заданными нормированными однородными координатами). """

  alpha = zeros((m,n))
  for i in range(min(points[0]),max(points[0])):
     for j in range(min(points[1]),max(points[1])):
        x = linalg.solve(points,[i,j,1])
        if min(x) > 0: #all coefficients positive
           alpha[i,j] = 1
  return alpha
```

Графическая карта умеет выполнять такую операцию очень быстро. Python работает гораздо медленнее графической карты (да и реализации на C/C++), но для наших целей его быстродействия достаточно. На рис. 3.3 снизу видно, что углы теперь совпали.

Рис. 3.3. Сравнение аффинного деформирования всего изображения с аффинным деформированием с применением двух треугольников. Изображение помещается на рекламный счет с учетом эффекта перспективы. Когда аффинное преобразование применяется ко всему изображению, совмещение оказывается неполным. Для наглядности оба правых угла увеличены (сверху). Если применить аффинное деформирование к двум треугольникам, то удается достичь полного совмещения (снизу)

Кусочно-аффинное деформирование

В предыдущем примере мы видели, что с помощью аффинного деформирования частей изображения можно точно совместить угловые точки. Рассмотрим наиболее употребительную форму деформирования при заданном множестве пар соответственных точек – *кусочно-аффинное деформирование*. Произвольное изображение с опорными точками можно деформировать в другое изображение с соответственными опорными точками, если выполнить триангуляцию исходного изображения по опорным точкам, а затем произвести аффинное преобразование каждого треугольника. В любой графической библиотеке существуют стандартные средства для выполнения этой операции. Ниже показано, как это сделать с помощью `Matplotlib` и `SciPy`.

Часто применяется *триангуляция Делоне*. Ее реализация имеется в библиотеке `Matplotlib` (но не в части `PyLab`) и воспользоваться ей можно так:

```
import matplotlib.delaunay as md

x,y = array(random.standard_normal((2,100)))
centers,edges,tri,neighbors = md.delaunay(x,y)

figure()
for t in tri:
  t_ext = [t[0], t[1], t[2], t[0]] # первая точка совпадает с последней
  plot(x[t_ext],y[t_ext],'r')

plot(x,y,'*')
axis('off')
show()
```

На рис. 3.4 показан результат триангуляции множества точек. Триангуляция Делоне строится так, чтобы достигал максимума минимальный угол при вершинах всех треугольников[2]. Функция `delaunay()` возвращает кортеж из четырех элементов, но нам из него нужен только список треугольников (третий элемент). Добавьте функцию триангуляции в файл *warp.py*.

```
import matplotlib.delaunay as md

def triangulate_points(x,y):
  """ Триангуляция Делоне точек на плоскости. """

  centers,edges,tri,neighbors = md.delaunay(x,y)
  return tri
```

[2] На самом деле, ребра этой триангуляции образуют граф, двойственный диаграмме Вороного. См. статью Википедии *https://ru.wikipedia.org/wiki/Триангуляция_Делоне*.

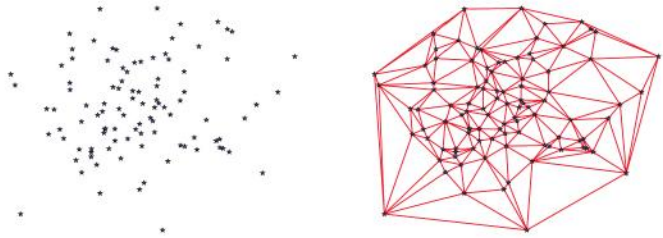


Рис. 3.4. Пример триангуляции Делоне для множества случайно выбранных точек на плоскости

На выходе получаем массив, в котором каждая строка содержит индексы в массивах *x* и *y* всех трех вершин каждого треугольника.

Воспользуемся этой функцией, чтобы перенести изображение на неплоский объект в другом изображении, взяв 30 контрольных точек, образующих сетку 5×6. На рис. 3.5 (b) показано изображение, которое мы хотим перенести на фасад небоскреба «закрученный торс», что находится в шведском городе Мальмё. Конечные точки были выбраны вручную с помощью функции `ginput()` и сохранены в файле *turningtorso_points.txt*.

Для начала нам понадобится общая функция кусочно-аффинного деформирования. Ее код приведен ниже, и в нем мы заодно воспользовались возможностью показать, как деформируются цветные изображения (нужно просто деформировать каждый цветовой канал по отдельности).

```
def pw_affine(fromim,toim,fp,tp,tri):
  """ Деформировать треугольные блоки изображения.
    fromim = деформируемое изображение
    toim = конечное изображение
    fp = исходные точки в однородных координатах
    tp = конечные точки в однородных координатах
    tri = триангуляция. """

  im = toim.copy()

  # является изображение полутоновым или цветным?
  is_color = len(fromim.shape) == 3

  # создать изображение, на которое переносится исходное
  # (необходимо в случае раздельного деформирования цветовых каналов)
  im_t = zeros(im.shape, 'uint8')
```

```
    for t in tri:
        # вычислить аффинное преобразование
        H = homography.Haffine_from_points(tp[:,t],fp[:,t])

        if is_color:
            for col in range(fromim.shape[2]):
                im_t[:,:,col] = ndimage.affine_transform(
                  fromim[:,:,col],H[:2,:2],(H[0,2],H[1,2]),im.shape[:2])
        else:
            im_t = ndimage.affine_transform(
                fromim,H[:2,:2],(H[0,2],H[1,2]),im.shape[:2])

        # альфа-отображение для треугольника
        alpha = alpha_for_triangle(tp[:,t],im.shape[0],im.shape[1])

        # добавить треугольник в изображение
        im[alpha>0] = im_t[alpha>0]

    return im
```

Здесь мы сначала проверяем, является изображение полутоновым или цветным. Для цветных изображений деформируется каждый цветовой канал по отдельности. Аффинное преобразование каждого треугольника определено однозначно, поэтому пользуемся функцией `Haffine_from_points()`. Добавьте следующую функцию в файл *warp.py*.

А следующий скрипт собирает все воедино:

```
import homography
import warp

# открыть деформируемое изображение
fromim = array(Image.open('sunset_tree.jpg'))
x,y = meshgrid(range(5),range(6))
x = (fromim.shape[1]/4) * x.flatten()
y = (fromim.shape[0]/5) * y.flatten()

# триангулировать
tri = warp.triangulate_points(x,y)

# открыть конечное изображение и файл конечных точек
open image and destination points
im = array(Image.open('turningtorso1.jpg'))
tp = loadtxt('turningtorso1_points.txt') # конечные точки

# преобразовать точки к однородным координатам
fp = vstack((y,x,ones((1,len(x)))))
```

```
tp = vstack((tp[:,1],tp[:,0],ones((1,len(tp)))))

# деформировать треугольники
im = warp.pw_affine(fromim,im,fp,tp,tri)

# построить график
figure()
imshow(im)
warp.plot_mesh(tp[1],tp[0],tri)
axis('off')
show()
```

Результат показан на рис. 3.5 (c). Треугольники рисуются с помощью такой вспомогательной функции (добавьте ее в файл *warp.py*):

```
def plot_mesh(x,y,tri):
  """ Нарисовать на графике треугольники. """

  for t in tri:
     t_ext = [t[0], t[1], t[2], t[0]] # добавить в конец первую точку
     plot(x[t_ext],y[t_ext],'r')
```

В этом примере демонстрируется все, что нужно знать для применения кусочно-аффинного деформирования изображений в собственных приложениях. Показанные функции можно усовершенствовать в различных направлениях. Некоторые из них упомянуты в упражнениях.

Рис. 3.5. Пример кусочно-аффинного деформирования с применением триангуляции Делоне по опорным точкам: (a) конечное изображение с опорными точками; (b) изображение с триангуляцией; (c) с наложенным изображением; (d) с наложенным изображением и триангуляцией

Регистрация изображений

Регистрацией изображения называется процесс приведения его к единой системе координат. Регистрация может быть изометрической или нет. Это важный начальный этап, позволяющий сравнивать изображения и проводить более сложный анализ.

Рассмотрим пример изометрической регистрации набора изображений лиц, так чтобы можно было осмысленно вычислить среднее лицо и отклонения от него. При таком виде регистрации мы ищем преобразование подобия (изометрическое преобразование, дополненное масштабированием), совмещающее соответственные пары точек. Дело в том, что размер, положение на фотографии и угол поворота лиц на изображениях могут различаться.

В файле *jkfaces.zip* имеется 366 изображений одного и того же лица (по одному на каждый день 2008 года)[3]. Каждое изображение аннотировано координатами глаз и рта, хранящимися в файле *jkfaces.xml*. По этим точкам можно вычислить преобразование подобия (которое, как уже было сказано, включает масштабирование). Для чтения XML-файлов мы воспользуемся модулем `minidom`, который входит в состав встроенного в Python пакета `xml.dom`.

XML-файл выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<faces>
  <face file="jk-002.jpg" xf="46" xm="56" xs="67" yf="38" ym="65" ys="39"/>
  <face file="jk-006.jpg" xf="38" xm="48" xs="59" yf="38" ym="65" ys="38"/>
  <face file="jk-004.jpg" xf="40" xm="50" xs="61" yf="38" ym="66" ys="39"/>
  <face file="jk-010.jpg" xf="33" xm="44" xs="55" yf="38" ym="65" ys="38"/>
  ...
</faces>
```

Для чтения координат создайте файл *imregistration.py* и добавьте в него такую функцию:

```
from xml.dom import minidom

def read_points_from_xml(xmlFileName):
  """ Читает контрольные точки для совмещения лиц. """

  xmldoc = minidom.parse(xmlFileName)
  facelist = xmldoc.getElementsByTagName('face')
  faces = {}
  for xmlFace in facelist:
```

[3] Изображения любезно предоставлены Дж. К. Келлером. Детали см. на сайте *http://jk-keller.com/daily-photo/*.

```
        fileName = xmlFace.attributes['file'].value
        xf = int(xmlFace.attributes['xf'].value)
        yf = int(xmlFace.attributes['yf'].value)
        xs = int(xmlFace.attributes['xs'].value)
        ys = int(xmlFace.attributes['ys'].value)
        xm = int(xmlFace.attributes['xm'].value)
        ym = int(xmlFace.attributes['ym'].value)
        faces[fileName] = array([xf, yf, xs, ys, xm, ym])
    return faces
```

Опорные точки возвращаются в виде словаря Python, в котором ключом является имя файла. Формат следующий: *xf*, *yf* – координаты левого глаза на изображении (т. е. правого глаза человека), *xs*, *ys* – координаты правого глаза, *xm*, *ym* – координаты рта.

Для вычисления параметров преобразования подобия воспользуемся методом наименьших квадратов. Для каждой точки $\mathbf{x}_i = [x_i, y_i]$ (в данном случае таких точек три) нам необходимо такое отображение на конечные точки $[\hat{x}_i, \hat{y}_i]$, что

$$\begin{bmatrix} \hat{x}_i \\ \hat{y}_i \end{bmatrix} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}.$$

Объединяя уравнения для всех трех точек, мы получаем систему уравнений с неизвестными a, b, t_x, t_y:

$$\begin{bmatrix} \hat{x}_1 \\ \hat{y}_1 \\ \hat{x}_2 \\ \hat{y}_2 \\ \hat{x}_3 \\ \hat{y}_3 \end{bmatrix} = \begin{bmatrix} x_1 & -y_1 & 1 & 0 \\ y_1 & x_1 & 0 & 1 \\ x_2 & -y_2 & 1 & 0 \\ y_2 & x_2 & 0 & 1 \\ x_3 & -y_3 & 1 & 0 \\ y_3 & x_3 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ t_x \\ t_y \end{bmatrix}.$$

Здесь мы использовали параметрическое представление матрицы подобия

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = s \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} = sR,$$

где $s = \sqrt{a^2 + b^2}$ – коэффициент масштабирования, а R – матрица поворота.

При большем числе соответственных пар точек все работает точно так же, только количество строк в матрице увеличивается. Решение по методу наименьших квадратов находит функция `linalg.lstsq()`. Использование этого метода – стандартный прием, с которым мы встретимся еще не раз. На самом деле, ту же идею мы применяли выше в алгоритме DLT. Ниже приведен код (добавьте его в файл *imregistration.py*):

```
from scipy import linalg

def compute_rigid_transform(refpoints,points):
  """ Вычисляет угол поворота, коэффициент масштабирования и вектор
     параллельного переноса для совмещения пар опорных точек. """

  A = array([ [points[0], -points[1], 1, 0],
              [points[1], points[0], 0, 1],
              [points[2], -points[3], 1, 0],
              [points[3], points[2], 0, 1],
              [points[4], -points[5], 1, 0],
              [points[5], points[4], 0, 1]])

  y = array([ refpoints[0],
              refpoints[1],
              refpoints[2],
              refpoints[3],
              refpoints[4],
              refpoints[5]])

  # метод наименьших квадратов минимизирует норму ||Ax - y||
  a,b,tx,ty = linalg.lstsq(A,y)[0]
  R = array([[a, -b], [b, a]]) # матрица поворота с масштабированием

  return R,tx,ty
```

Эта функция возвращает матрицу поворота, коэффициент масштабирования и вектор параллельного переноса в виде координат по осям x и y. Чтобы выполнить деформирование и сохранить новые совмещенные изображения, применим функцию `ndimage.affine_transform()` к каждому цветовому каналу (все эти изображения цветные). В качестве эталона для сравнения можно взять любые три точки. В следующем коде мы для простоты взяли опорные точки из первого изображения:

```
from scipy import ndimage
from scipy.misc import imsave

import os

def rigid_alignment(faces,path,plotflag=False):
```

```
    """ Изометрически совместить изображения и сохранить новые
       изображения. path задает путь к файлу, в котором сохранять
       совмещенные изображения. Флаг plotflag=True означает, что
       нужно нанести изображения на график. """

    # взять в качестве опорных точки из первого изображения
    refpoints = faces.values()[0]

    # деформировать каждое изображение с помощью аффинного преобразования
    for face in faces:
       points = faces[face]

    R,tx,ty = compute_rigid_transform(refpoints, points)
    T = array([[R[1][1], R[1][0]], [R[0][1], R[0][0]]])

    im = array(Image.open(os.path.join(path,face)))
    im2 = zeros(im.shape, 'uint8')

    # деформировать каждый цветовой канал
    for i in range(len(im.shape)):
       im2[:,:,i] = ndimage.affine_transform(im[:,:,i],linalg.inv(T),
                       offset=[-ty,-tx])

    if plotflag:
       imshow(im2)
       show()

    # обрезать рамку и сохранить совмещенные изображения
    h,w = im2.shape[:2]
    border = (w+h)/20

    # обрезать рамку
    imsave(os.path.join(path, 'aligned/'+face),im2[border:h-border,
                                                   border:w-border,:])
```

Здесь мы воспользовались функцией `imsave()` для сохранения совмещенных изображений в подкаталоге «aligned».

Показанный ниже коротенький скрипт читает XML-файл, содержащий имена файлов в качестве ключей, а сами точки в качестве значений, и регистрирует все изображения, совмещая их с первым.

```
import imregistration

# загрузить файл, содержащий положения контрольных точек
xmlFileName = 'jkfaces2008_small/jkfaces.xml'
points = imregistration.read_points_from_xml(xmlFileName)

# зарегистрировать
imregistration.rigid_alignment(points,'jkfaces2008_small/')
```

После прогона этого скрипта в подкаталоге будут находиться совмещенные изображения лиц. На рис. 3.6 показаны шесть примеров до и после регистрации. Зарегистрированные изображения немного обрезаны, чтобы убрать нежелательные черные пиксели по краям.

Рис. 3.6. Примеры изображений до (сверху) и после (снизу) изометрической регистрации

Теперь посмотрим, как это влияет на построение среднего изображения. На рис. 3.7 показано среднее изображение в отсутствие совмещения, а рядом – после совмещения (отметим, что размеры различаются из-за обрезки рамки). И хотя исходные изображения слабо варьируются по размеру лица, углу поворота и положению, на вычислении среднего эти колебания сказываются очень заметно.

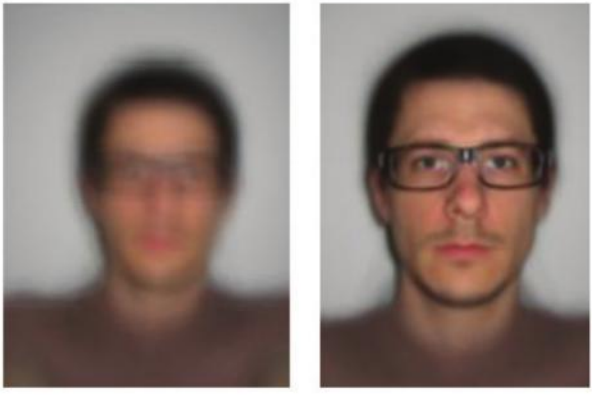


Рис. 3.7. Сравнение средних изображений: без совмещения (слева) и с изометрическим совмещением по трем точкам (справа)

Не удивительно, что использование плохо зарегистрированных изображений весьма негативно отражается на вычислении главных компонент. На рис. 3.8 показан результат применения метода PCA к первым 150 изображениям без регистрации и с регистрацией. Как и среднее изображение, PCA-моды размыты. При вычислении главных компонент мы использовали маску, состоящую из эллипса с центром в средней точке лица. Умножая изображения на эту маску перед объединением в общий массив, мы устраняем влияние различий в фоне на PCA-моды. Нужно лишь в примере из раздела 1.3 (стр. 36) заменить строку, в которой создается матрица:

```
immatrix = array([mask*array(Image.open(imlist[i]).convert('L')).flatten()
    for i in range(150)],'f')
```

где `mask` – бинарное изображение того же размера, уже линеаризованное.

Рис. 3.8. Сравнение PCA-мод незарегистрированных и зарегистрированных изображений: среднее изображение и первые девять главных компонент без предварительной регистрации (сверху); то же самое для зарегистрированных изображений (снизу)

3.3. Создание панорам

Два (или более) изображений, снятых в одном и том же месте (т. е. положение камеры одинаково для всех изображений), гомографически связаны (см. рис. 3.9). Этот факт часто используется для создания панорамных изображений, когда несколько отдельных изображений сшиваются в одно большое. В этом разделе мы узнаем, как это делается.

Рис. 3.9. Пять фотографий главного университетского здания в шведском городе Лунде. Все фотографии сделаны из одной и той же точки

RANSAC

RANSAC (RANdom SAmple Consensus) – итеративный метод подгонки моделей к данным, содержащим выбросы. Пусть имеется модель, например, гомография между двумя наборами точек. Идея в том, что данные могут содержать *регулярные точки*, описываемые моделью, и *выбросы* – точки, не укладывающиеся в модель.

Стандартный пример – аппроксимация прямой множества точек, содержащего выбросы. Простой метод наименьших квадратов в данном случае не годится, но RANSAC при некоторой удаче может выделить регулярные точки и построить правильную аппроксимацию. Рассмотрим использование программы *ransac.py*, описанной на странице *http://www.scipy.org/Cookbook/RANSAC*, где этот пример приведен в качестве тестового. На рис. 3.10 показан результат

выполнения функции `ransac.test()`. Как видите, алгоритм выбирает только точки, согласующиеся с линейной моделью, и правильно находит решение.

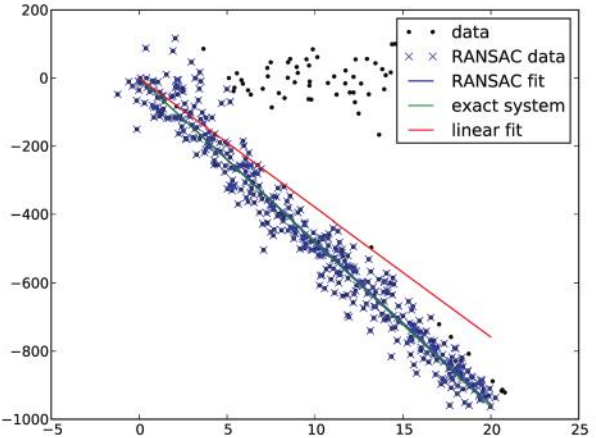


Рис. 3.10. Пример использования метода RANSAC для аппроксимации прямой множества точек с выбросами

RANSAC – очень полезный алгоритм, которым мы воспользуемся в следующем разделе для вычисления гомографии и в других примерах. Дополнительные сведения можно найти в оригинальной работе Фишлера и Боллеса [11], в статье *http://ru.wikipedia.org/wiki/RANSAC* или в отчете [40].

Устойчивое вычисление гомографии

Использовать модуль RANSAC можно для любой модели. От вас нужен лишь класс Python, содержащий методы `fit()` и `get_error()`, а обо всем остальном позаботится скрипт *ransac.py*. Сейчас нас будет интересовать автоматический поиск гомографии для панорамных изображений с использованием множества возможных соответствий. На рис. 3.11 показаны соответственные точки, найденные автоматически с помощью SIFT-признаков в результате выполнения следующей программы:

```
import sift

featname = ['Univ'+str(i+1)+'.sift' for i in range(5)]
imname = ['Univ'+str(i+1)+'.jpg' for i in range(5)]
```

```
l = {}
d = {}
for i in range(5):
  sift.process_image(imname[i],featname[i])
  l[i],d[i] = sift.read_features_from_file(featname[i])

matches = {}
for i in range(4):
  matches[i] = sift.match(d[i+1],d[i])
```

Рис. 3.11. Соответствия между последовательными парами изображений, найденные с помощью SIFT-признаков

Из рассмотрения изображений понятно, что не все соответствия правильны. Конечно, SIFT – очень надежный дескриптор, дающий меньше ложных соответствий, чем, скажем, детектор угловых точек Харриса с корреляцией блоков, но все же он далек от совершенства. Чтобы подобрать гомографию методом RANSAC, сначала нужно добавить в файл *homography.py* следующий класс модели:

```
class RansacModel(object):
  """ Класс для тестирования подбора гомографии с помощью скрипта
```

```
        ransac.py с сайта http://www.scipy.org/Cookbook/RANSAC. """

    def __init__(self,debug=False):
        self.debug = debug

    def fit(self, data):
        """ Подобрать гомографию по четырем выбранным соответствиям. """

        # транспонировать для аппроксимации функцией H_from_points()
        data = data.T

        # исходные точки
        fp = data[:3,:4]

        # конечные точки
        tp = data[3:,:4]

        # подобрать и вернуть гомографию
        return H_from_points(fp,tp)

    def get_error( self, data, H):
        """ Применить гомографию ко всем соответствиям,
            вернуть среднеквадратичную ошибку. """

        data = data.T

        # исходные точки
        fp = data[:3]

        # конечные точки
        tp = data[3:]

        # преобразовать fp
        fp_transformed = dot(H,fp)

        # нормировать однородные координаты
        for i in range(3):
            fp_transformed[i] /= fp_transformed[2]

        # вернуть среднеквадратичную ошибку
        return sqrt( sum((tp-fp_transformed)**2,axis=0) )
```

Этот класс содержит метод `fit()`, который принимает четыре соответствия, выбранных *ransac.py* (первые четыре точки в массиве *data*), и подбирает гомографию. Напомним, что четыре точки – это минимальное количество для вычисления гомографии. Метод `get_error()` применяет гомографию и возвращает сумму квадратов расстояний между каждой парой соответственных точек, чтобы RANSAC мог решить, какие точки является регулярными, а какие – выбросами. Для

простоты добавьте в файл *homography.py* следующую вспомогательную функцию:

```
def H_from_ransac(fp,tp,model,maxiter=1000,match_theshold=10):
  """ Устойчивое вычисление гомографии H по соответственным
     точкам с применением метода RANSAC (ransac.py на странице
     http://www.scipy.org/Cookbook/RANSAC).
     вход: fp,tp (массивы 3*n) - точки в однородных координатах. """

  import ransac

  # сгруппировать соответственные точки
  data = vstack((fp,tp))

  # вычислить и вернуть H
  H,ransac_data = ransac.ransac(data.T,model,4,maxiter,match_theshold,10,
                                return_all=True)
  return H,ransac_data['inliers']
```

Этой функции можно также задать порог и минимальное требуемое число точек. Самый важный параметр – максимальное число итераций: если закончить слишком рано, то может получиться плохое решение, а если задать слишком много итераций, то понадобится больше времени. Вместе с вычисленной гомографией возвращаются все регулярные точки.

Алгоритм RANSAC применяется к соответственным точкам следующим образом:

```
# эта функция преобразует соответственные точки в однородные координаты
def convert_points(j):
  ndx = matches[j].nonzero()[0]
  fp = homography.make_homog(l[j+1][ndx,:2].T)
  ndx2 = [int(matches[j][i]) for i in ndx]
  tp = homography.make_homog(l[j][ndx2,:2].T)
  return fp,tp

# вычислить гомографии
model = homography.RansacModel()

fp,tp = convert_points(1)
H_12 = homography.H_from_ransac(fp,tp,model)[0] #изображение 1 к 2

fp,tp = convert_points(0)
H_01 = homography.H_from_ransac(fp,tp,model)[0] #изображение 0 к 1

tp,fp = convert_points(2) #NB: reverse order
H_32 = homography.H_from_ransac(fp,tp,model)[0] #изображение 3 к 2

tp,fp = convert_points(3) #NB: reverse order
H_43 = homography.H_from_ransac(fp,tp,model)[0] #изображение 4 к 3
```

В этом примере изображение 2 является центральным, именно к нему мы хотим приложить остальные. Изображения 0 и 1 прикладываются справа, а изображения 3 и 4 слева. При вычислении соответствий исходным считается правое изображение в каждой паре, поэтому для изображений, прикладываемых слева, порядок соответствий инвертируется. Из двух возвращенных функцией результатов нас интересует только первый (сама гомография), а регулярные точки отбрасываются.

Сшивка изображений

После того как гомографии между изображениями вычислены (методом RANSAC), мы должны наложить все изображения на общую плоскость. Имеет смысл брать плоскость центрального изображения, иначе искажения будут слишком велики. Сделать это можно, например, следующим образом: создаем очень большое изображение, заполненное нулями, в плоскости, параллельной центральному изображению, и накладываем на него все изображения. Поскольку во время фотосъемки камера перемещалась только в горизонтальном направлении, мы можем даже упростить процедуру: достаточно дополнить центральное изображение слева и справа нулями, так чтобы было достаточно места для прикладывания остальных изображений. Добавьте в файл *warp.py* функцию, реализующую эту идею:

```
def panorama(H,fromim,toim,padding=2400,delta=2400):
  """ Создать горизонтальную панораму путем объединения двух
    изображений с применением гомографии H (вычисленной методом
    RANSAC). В результате получится изображение такой же высоты,
    как toim. Параметр 'padding'задает число пикселей заполнения,
    а 'delta' - дополнительный параллельный перенос. """

  # какое задано изображение - полутоновое или цветное?
  is_color = len(fromim.shape) == 3

  # гомографическое преобразование для функции geometric_transform()
  def transf(p):
     p2 = dot(H,[p[0],p[1],1])
     return (p2[0]/p2[2],p2[1]/p2[2])

  if H[1,2]<0: # fromim справа
     print 'warp - right'
     # преобразовать fromim
     if is_color:
        # дополнить конечное изображение нулями справа
```

```
            toim_t = hstack((toim,zeros((toim.shape[0],padding,3))))
            fromim_t = zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2]))
            for col in range(3):
                fromim_t[:,:,col] = ndimage.geometric_transform(fromim[:,:,col],
                        transf,(toim.shape[0],toim.shape[1]+padding))
        else:
            # дополнить конечное изображение нулями справа
            toim_t = hstack((toim,zeros((toim.shape[0],padding))))
            fromim_t = ndimage.geometric_transform(fromim,transf,
                        (toim.shape[0],toim.shape[1]+padding))
    else:
        print 'warp - left'
        # добавить параллельный перенос для компенсации дополнения слева
        H_delta = array([[1,0,0],[0,1,-delta],[0,0,1]])
        H = dot(H,H_delta)

        # преобразовать fromim
        if is_color:
            # дополнить конечное изображение нулями справа
            toim_t = hstack((zeros((toim.shape[0],padding,3)),toim))
            fromim_t = zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2]))
            for col in range(3):
                fromim_t[:,:,col] = ndimage.geometric_transform(fromim[:,:,col],
                        transf,(toim.shape[0],toim.shape[1]+padding))
        else:
            # дополнить конечное изображение нулями справа
            toim_t = hstack((zeros((toim.shape[0],padding)),toim))
            fromim_t = ndimage.geometric_transform(fromim,
                        transf,(toim.shape[0],toim.shape[1]+padding))

    # объединить и вернуть (поместив fromim над toim)
    if is_color:
        # все нечерные пиксели
        alpha = ((fromim_t[:,:,0] * fromim_t[:,:,1] * fromim_t[:,:,2] ) > 0)
        for col in range(3):
            toim_t[:,:,col] = fromim_t[:,:,col]*alpha + toim_t[:,:,col]*(1-alpha)
    else:
        alpha = (fromim_t > 0)
        toim_t = fromim_t*alpha + toim_t*(1-alpha)

    return toim_t
```

Обобщенной функции `geometric_transform()` необходимо передать функцию, которое описывает отображение пикселей. В данном случае это функция `transf()`, которая выполняет умножение на H и нормировку однородных координат. Чтобы узнать, с какой стороны дополнять изображение, мы смотрим, какой параллельный перенос указан в H. Если изображение дополняется нулями слева, то координаты точек в конечном изображении изменяются, поэтому к го-

мографии добавляется параллельный перенос. Для простоты мы по-прежнему используем прием с нулевыми пикселями для нахождения альфа-отображения.

Теперь применим эту функцию к нашим изображениям:

```
# деформировать изображения
delta = 2000 # для дополнения и параллельного переноса
im1 = array(Image.open(imname[1]))
im2 = array(Image.open(imname[2]))
im_12 = warp.panorama(H_12,im1,im2,delta,delta)
im1 = array(Image.open(imname[0]))
im_02 = warp.panorama(dot(H_12,H_01),im1,im_12,delta,delta)
im1 = array(Image.open(imname[3]))
im_32 = warp.panorama(H_32,im1,im_02,delta,delta)
im1 = array(Image.open(imname[j+1]))
im_42 = warp.panorama(dot(H_32,H_43),im1,im_32,delta,2*delta)
```

Обратите внимание, что в последней строке изображение *im_32* уже один раз переносилось. Получившаяся панорама показана на рис. 3.12. На границах изображений и по краям заметны эффекты разной выдержки. В коммерческие программы построения панорам включена дополнительная обработка для нормировки яркости и сглаживания переходов с целью повысить качество результата.

Рис. 3.12. Горизонтальная панорама, автоматически созданная по соответствиям, найденным SIFT: полная панорама (вверху) и вырезанная центральная часть (внизу)

Упражнения

1. Напишите функцию, которая принимает координаты квадратной (или прямоугольной) области внутри изображения (например, книги, плаката или двумерного штрих-кода) и вычисляет преобразование, которое показывает эту область целиком во фронтальной проекции в нормированной системе координат. Для нахождения точек воспользуйтесь функцией `ginput()` или возьмите самые четкие углы Харриса.
2. Напишите функцию, которая корректно определяет альфа-отображение для деформирования типа того, что показано на рис. 3.1.
3. Выберите сами набор изображений, содержащих три общие опорные точки (как в нашем примере с лицами или фотографии какого-нибудь знаменитого объекта, скажем, Эйфелевой башни). Постройте совмещенные изображения, в которых опорные точки находятся в одной и той же позиции. Вычислите среднее и медианное изображение и выведите их.
4. Реализуйте нормировку яркости и улучшите способ объединения изображений, устранив краевые эффекты, видные на рис. 3.12.
5. При построении панорам можно не прикладывать изображения к центральному, а помещать их на цилиндрическую поверхность. Попробуйте сделать так для примера на рис. 3.12.
6. С помощью метода RANSAC найдите нескольких доминирующих наборов регулярных точек для гомографии. Проще всего сделать это так: один раз прогнать RANSAC, найти гомографию с наибольшим согласованным подмножеством точек, удалить регулярные точки из множества соответствий, еще раз прогнать RANSAC для получения следующего наибольшего набора и т. д.
7. Модифицируйте метод вычисления гомографии RANSAC с целью вычисления аффинных преобразований по трем парам соответственных точек. С помощью этого метода определите, изображена ли на двух фотографиях плоская сцена, например, подсчитав количество регулярных точек. Если сцена плоская, то число регулярных точек для аффинного преобразования будет большим.
8. Постройте *панограф* (*http://en.wikipedia.org/wiki/Panography*) по набору изображений (например, с сайта Flickr), применив сопоставление локальных признаков и изометрическую регистрацию методом наименьших квадратов.

ГЛАВА 4. Модели камер и дополненная реальность

В этой главе мы будем рассматривать модели камер и их эффективное использование. В предыдущей главе мы познакомились с преобразованиями и сопоставлением изображений. Для обработки соответствия между трехмерными сценами и двумерными изображениями необходимо учитывать проекционные свойства камеры, с помощью которой было создано изображение. Мы покажем, как определить характеристики камеры и как использовать проекции изображений в таких приложениях, как дополненная реальность. В следующей главе мы применим модель камеры к обработке нескольких видов и установления соответствия между ними.

4.1. Модель камеры с точечной диафрагмой

Модель *камеры с точечной диафрагмой* (иногда употребляют также термин *проективная камера*) широко используется в компьютерном зрении. Она проста и в то же время достаточно точна для большинства приложений. Название происходит от типа камеры, в которую свет проникает через небольшое отверстие в темный внутренний ящик – как в камере-обскуре. В модели камеры с точечной диафрагмой свет проходит через одну точку – *центр камеры* **C** – и проецируется на *плоскость изображения*. На рис. 4.1 показана плоскость изображения перед центром камеры. В настоящей камере плоскость изображения позади центра камеры была бы перевернута, но для модели это неважно.

Проекционные свойства камеры с точечной диафрагмой можно вывести из этого рисунка в предположении, что ось изображения совпадает с осью x или y трехмерной системы координат. Тогда *оп-*

тическая ось камеры совпадает с осью z, а проекция определяется из рассмотрения подобных треугольников. Если добавить поворот и параллельный перенос, которые переводят точку трехмерного пространства в эту систему координат перед проецированием, то получится проективное преобразование общего вида. Интересующийся читатель сможем найти подробности в работах [13] и [25, 26].

В случае камеры с точечной диафрагмой проекция трехмерной точки $\mathbf{X}$ в точку изображения $\mathbf{x}$ (обе выражены в однородных координатах) описывается уравнением

$$\lambda \mathbf{x} = P\mathbf{X} \qquad (4.1)$$

Здесь матрица P размерности 3×4 называется *матрицей камеры* (или *проекционной матрицей*). Отметим, что трехмерная точка $\mathbf{X}$ описывается четырьмя однородными координатами $\mathbf{X} = [X, Y, Z, W]$. Скалярная величина λ называется *обратной глубиной* точки и нужна в том случае, когда мы хотим, чтобы координаты были однородными, а последняя была равна единице.

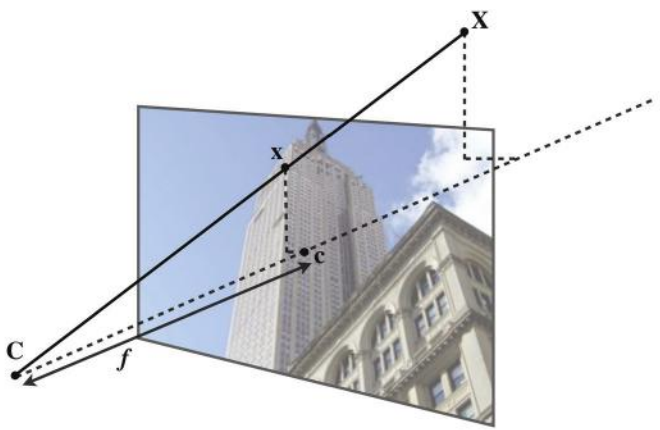


Рис. 4.1. Модель камеры с точечной диафрагмой. Точка изображения **x** находится на пересечении плоскости изображения с прямой, проходящей через точку трехмерного пространства **X** и центр камеры **C**. Пунктирной линией показана оптическая ось камеры

Матрица камеры

Матрицу камеры можно представить в виде

$$P = K\,[R \mid \mathbf{t}] \qquad (4.2)$$

где R – матрица поворота, описывающая ориентацию камеры, $\mathbf{t}$ – трехмерный вектор параллельного переноса, определяющий положение центра камеры, а K – *калибровочная матрица*, описывающая проекционные характеристики камеры.

Калибровочная матрица зависит только от характеристик камеры и в общем виде записывается так:

$$K = \begin{bmatrix} \alpha f & s & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Фокусное расстояние f – это расстояние между плоскостью изображения и центром камеры. Наклон s используется, только если светочувствительная пиксельная матрица наклонена, и в большинстве случаев его можно считать нулем. Таким образом, получаем

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \qquad (4.3)$$

Здесь мы воспользовались альтернативной нотацией: f_x и f_y, где $f_x = \alpha f_y$.

Отношение сторон α используется, когда пиксель не квадратный. Часто можно без опаски предполагать, что $\alpha = 1$. При таком предположении матрица принимает вид

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Помимо фокусного расстояния есть еще только два параметра: координаты *оптического центра* (иногда его еще называют *главной точкой*) – точки изображения $\mathbf{c} = [c_x, c_y]$, в которой оптическая ось пересекает плоскость изображения. Поскольку обычно она находится в центре изображения, а координаты изображения отсчитываются от левого верхнего угла, то эти параметры часто аппроксимируют половиной ширины и высоты изображения соответственно. Стоит отметить, что в таком случае неизвестной остается только одна переменная – фокусное расстояние f.

Проецирование точек трехмерного пространства

Создадим класс камеры, который будет отвечать за все операции, необходимые для моделирования камеры и проецирования.

```
from scipy import linalg

class Camera(object):
  """ Класс для представления камеры с точечной диафрагмой. """

  def __init__(self,P):
     """ Инициализировать модель камеры P = K[R|t]. """
     self.P = P
     self.K = None # калибровочная матрица
     self.R = None # поворот
     self.t = None # параллельный перенос
     self.c = None # центр камеры

  def project(self,X):
     """ Спроецировать точки из массива X (4*n) и нормировать
        координаты. """

     x = dot(self.P,X)
     for i in range(3):
        x[i] /= x[2]
     return x
```

В примере ниже показано, как спроецировать точки трехмерного пространства на плоскость изображения. В этом примере мы возьмем один из нескольких оксфордских многовидовых наборов данных, «Модель дома», который можно скачать со страницы *http://www.robots.ox.ac.uk/~vgg/data/data-mview.html*. Скачайте файл *house.p3d* в свой рабочий каталог.

```
import camera

# загрузить точки
points = loadtxt('house.p3d').T
points = vstack((points,ones(points.shape[1])))

# настроить камеру
P = hstack((eye(3),array([[0],[0],[-10]])))
cam = camera.Camera(P)
x = cam.project(points)

# изобразить проекцию на графике
```

```
figure()
plot(x[0],x[1],'k.')
show()
```

Сначала мы преобразуем координаты точек в однородные и создаем объект `Camera` с проекционной матрицей. Затем проецируем точки и наносим их на график. Результат показан на рис. 4.2 в центре.

Чтобы увидеть, как изменяется проекция при перемещении камеры, выполните следующий код, который в цикле поворачивает камеру вокруг случайно выбранной оси в трехмерном пространстве.

```
# создать преобразование
r = 0.05*random.rand(3)
rot = camera.rotation_matrix(r)

# повернуть камеру и спроецировать
figure()
for t in range(20):
  cam.P = dot(cam.P,rot)
  x = cam.project(points)
  plot(x[0],x[1],'k.')
show()
```

Здесь мы воспользовались вспомогательной функцией `rotation_matrix()`, которая создает матрицу поворота вокруг вектора в трехмерном пространстве (добавьте ее в файл *camera.py*):

```
def rotation_matrix(a):
  """ Создает матрицу поворота вокруг оси вектора a в трехмерном
     пространстве. """
  R = eye(4)
  R[:3,:3] = linalg.expm([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])
  return R
```

На рис. 4.2 показано одно изображение из этого набора данных, проекция точек и траектории проекций при повороте камеры вокруг случайно выбранной оси. Выполните эту программу несколько раз с различными случайными осями вращения, что получить представление о том, как перемещаются проекции точек.

Разложение матрицы камеры

Пусть дана матрица камеры P, и мы хотим восстановить внутренние параметры K, а также положение и угол поворота камеры $\mathbf{t}$ и R (см. уравнение (4.2)). Представление матрицы в виде произведения называется *разложением*. В данном случае нас будет интересовать так называемое *RQ-разложение*.

Рис. 4.2. Пример проецирования точек трехмерного пространства: исходное изображение (слева); проекции точек на плоскость (в центре); траектории проекций при повороте камеры (справа). Данные взяты из оксфордского набора «Модель дома»

Добавьте в класс *Camera* такой метод:

```
def factor(self):
  """ Представить матрицу камеры в виде P = K[R|t]. """

  # разложить левый верхний блок 3*3
  K,R = linalg.rq(self.P[:,:3])

  # сделать элементы на диагонали K положительными
  T = diag(sign(diag(K)))
  if linalg.det(T) < 0:
     T[1,1] *= -1

  self.K = dot(K,T)
  self.R = dot(T,R) # обратная к T матрица совпадает с ней самой
  self.t = dot(linalg.inv(self.K),self.P[:,3])

  return self.K, self.R, self.t
```

RQ-разложение не единственно, при выборе знака существует неоднозначность. Поскольку мы хотим, что определитель матрицы поворота R был положителен (в противном случае направление осей координат может поменяться на противоположное), то можем при необходимости добавить преобразование T, меняющее знак.

Выполните следующий код для нашего примера камеры:

```
import camera

K = array([[1000,0,500],[0,1000,300],[0,0,1]])
tmp = camera.rotation_matrix([0,0,1])[:3,:3]
Rt = hstack((tmp,array([[50],[40],[30]])))
cam = camera.Camera(dot(K,Rt))

print K,Rt
print cam.factor()
```

На консоли должны быть напечатаны ее параметры.

Вычисление центра камеры

Пусть дана проекционная матрица камеры P. Полезно знать, как вычислить положение камеры в пространстве. Центр камеры $\mathbf{C}$ – это точка, для которой $P\mathbf{C} = 0$. Если камера описывается матрицей $P = K[R|\mathbf{t}]$, то получаем

$$K[R|\mathbf{t}]\mathbf{C} = KR\mathbf{C} + K\mathbf{t} = 0,$$

и центр камеры можно вычислить в виде

$$\mathbf{C} = -R^T\mathbf{t}$$

Отметим, что центр камеры не зависит от калибровочной матрицы K, что и неудивительно.

Добавьте в класс `Camera` следующий метод для вычисления центра камеры по приведенной выше формуле.

```
def center(self):
  """ Вычислить и вернуть центр камеры. """

  if self.c is not None:
     return self.c
  else:
     # вычислить c с помощью разложения матрицы
     self.factor()
     self.c = -dot(self.R.T,self.t)
     return self.c
```

Итак, все основные функции класса `Camera` имеются. Теперь посмотрим, как работать с построенной моделью камеры с точечной диафрагмой.

4.2. Калибровка камеры

Под калибровкой камеры понимается определение ее внутренних параметров, в данном случае матрицы камеры K. Модель камеры можно обобщить, включив в нее радиальную дисторсию и другие артефакты, если приложение нуждается в прецизионных измерениях. Но для большинства приложений простой модели (4.3) вполне достаточно. Стандартный способ калибровки камеры – многократное фотографирование испытательной таблицы в виде шахматной доски. Так, например, устроены средства калибровки в библиотеке OpenCV (подробнее см. [3]).

Простой метод калибровки

Рассмотрим простой метод калибровки. Большинство параметров можно задать, приняв простые предположения (квадратные пиксели с прямолинейными границами, оптический центр совпадает с центром изображения), так что основную сложность представляет определение фокусного расстояния. Для описываемого метода необходимы плоский прямоугольный калибровочный объект (книга вполне подойдет), рулетка или линейка и плоская поверхность. Последовательность действий такова.

- Измерить стороны прямоугольного калибровочного объекта. Обозначим их dX и dY.
- Положить камеру и калибровочный объект на плоскую поверхность, так чтобы обратная сторона камеры была параллельна объекту и объект находился примерно в центре поля зрения камеры. Для лучшего совмещения, возможно, придется приподнять камеру или объект.
- Измерить расстояние между камерой и калибровочным объектом. Обозначим его dZ.
- Сделать снимок и проверить, что стороны объекта совмещаются со строками и столбцами матрицы изображения.
- Измерить ширину и высоту объекта в пикселях. Обозначим их dx и dy.

Организация измерения показана на рис. 4.3. Из подобия треугольников (взгляните на рис. 4.1, чтобы убедиться) вытекает следующая формула, дающая компоненты фокусного расстояния:

$$f_x = \frac{dx}{dX} dZ, \quad f_y = \frac{dy}{dY} dZ.$$

Конкретно для ситуации, показанной на рис. 4.3, объект имел размеры 130 × 185 мм, т. е. $dX = 130$, $dY = 185$. Расстояние от камеры до объекта составляло 460 мм, т. е. $dZ = 460$. Единица измерения может быть любой, важны только отношения длин. Воспользовавшись функцией `ginput()` для выбора четырех точек изображения, находим, что ширина и высота в пикселях равны соответственно 722 и 1040, т. е. $dx = 722$, $dy = 1040$. Подставляя все эти значения в формулу, получаем

$$f_x = 2555, \qquad f_y = 2586.$$

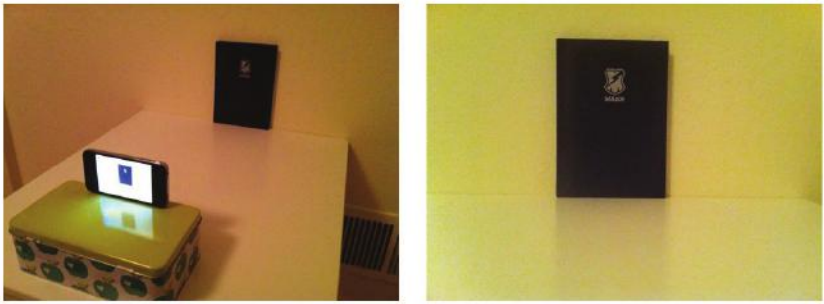


Рис. 4.3. Организация простой калибровки камеры: взаимное расположение предметов (слева); изображение, используемое для калибровки (справа). Для определения фокусного расстояния достаточно измерить ширину и высоту объекта калибровки на изображение и в натуре

Важно отметить, что это относится *к конкретному разрешению изображения*. В данном случае разрешение изображения было равно 2592×1936 пикселей. Напомним, что фокусное расстояние и оптический центр измеряются в пикселях и масштабируются при изменении разрешения изображения. Если взять другое разрешение, то значения изменятся. Удобно добавить в описание своей камеры эти константы с помощью такой вспомогательной функции:

```
def my_calibration(sz):
  row,col = sz
  fx = 2555*col/2592
  fy = 2586*row/1936
  K = diag([fx,fy,1])
  K[0,2] = 0.5*col
  K[1,2] = 0.5*row
  return K
```

Эта функция принимает кортеж с размерами и возвращает калибровочную матрицу. Здесь предполагается, что оптический центр совпадает с центром изображения. Если хотите, можете заменить фокусные расстояния их средним арифметическим, для большинства бытовых камер это приемлемо. Отметим, что эта калибровка применима к изображениям в альбомной ориентации. Для книжной ориентации нужно будет поменять константы местами. Сохраним эту функцию и воспользуемся ей в следующем разделе.

4.3. Оценивание положения по плоскостям и маркерам

В главе 3 мы видели, как вычислять гомографию между плоскими изображениями. В сочетании с калиброванной камерой это позволяет вычислить положение камеры (угол поворота и вектор параллельного переноса), если изображение содержит плоскостной маркерный объект, в качестве которого может выступать почти любой плоский объект.

Рис. 4.4. Пример вычисления проекционной матрицы для нового ракурса с использованием плоскостного объекта в качестве маркера. Сочетание признаков изображения с совмещенным маркером дает гомографию, которую можно использовать для вычисления положения камеры. Эталонное изображение с серым квадратом (слева вверху); фотография, снятая из неизвестной точки, с тем же квадратом, преобразованным в соответствии с вычисленной гомографией (справа вверху); куб, преобразованный с использованием вычисленной матрицы камеры (внизу)

Проиллюстрируем на примере. Рассмотрим два верхних изображения на рис. 4.4. Следующая программа выделяет из обоих изображений SIFT-признаки и вычисляет гомографию методом RANSAC:

```
import homography
import camera
import sift

# вычислить признаки
sift.process_image('book_frontal.JPG','im0.sift')
l0,d0 = sift.read_features_from_file('im0.sift')

sift.process_image('book_perspective.JPG','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

# сопоставить признаки и вычислить гомографию
matches = sift.match_twosided(d0,d1)
ndx = matches.nonzero()[0]
fp = homography.make_homog(l0[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
tp = homography.make_homog(l1[ndx2,:2].T)

model = homography.RansacModel()
H = homography.H_from_ransac(fp,tp,model)
```

Итак, у нас имеется гомография, которая совмещает точки маркера (в данном случае книги) на одном изображении с соответственными точками на другом изображении. Определим систему координат в пространстве, так чтобы маркер находился в плоскости X-Y ($Z = 0$), а начало координат совпадало с какой-нибудь точкой маркера.

Для проверки результатов нам понадобится простой трехмерный объект, помещенный на маркер. Мы воспользуемся для этой цели кубом и сгенерируем вершины куба с помощью такой функции:

```
def cube_points(c,wid):
  """ Создает список точек для нанесения куба на график. (Первые
      5 точек – нижняя грань, некоторые боковые грани повторяются.) """

  p = []

  # нижняя грань
  p.append([c[0]-wid,c[1]-wid,c[2]-wid])
  p.append([c[0]-wid,c[1]+wid,c[2]-wid])
  p.append([c[0]+wid,c[1]+wid,c[2]-wid])
  p.append([c[0]+wid,c[1]-wid,c[2]-wid])
  p.append([c[0]-wid,c[1]-wid,c[2]-wid]) # совпадает с первой для
                                         # замыкания ломаной
  # верхняя грань
  p.append([c[0]-wid,c[1]-wid,c[2]+wid])
  p.append([c[0]-wid,c[1]+wid,c[2]+wid])
  p.append([c[0]+wid,c[1]+wid,c[2]+wid])
  p.append([c[0]+wid,c[1]-wid,c[2]+wid])
```

```
    p.append([c[0]-wid,c[1]-wid,c[2]+wid]) # совпадает с первой для
                                           # замыкания ломаной

    # боковые грани
    p.append([c[0]-wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]-wid])

    return array(p).T
```

Некоторые точки повторяются несколько раз, чтобы функция `plot()` нарисовала куб правильно. Применяя гомографию и калибровочную матрицу камеры, мы можем определить относительное преобразование между двумя ракурсами:

```
# калибровка камеры
K = my_calibration((747,1000))

# точки в плоскости z=0, принадлежащие кубу с длиной стороны 0.2
box = cube_points([0,0,0.1],0.1)

# проецируем нижнюю грань на первое изображение
cam1 = camera.Camera( hstack((K,dot(K,array([[0],[0],[-1]])) )) )

# первые точки соответствуют нижней грани
box_cam1 = cam1.project(homography.make_homog(box[:,:5]))

# используем H для переноса точек на второе изображение
box_trans = homography.normalize(dot(H,box_cam1))

# вычисляем матрицу второй камеры по cam1 и H
cam2 = camera.Camera(dot(H,cam1.P))
A = dot(linalg.inv(K),cam2.P[:,:3])
A = array([A[:,0],A[:,1],cross(A[:,0],A[:,1])]).T
cam2.P[:,:3] = dot(K,A)

# проецируем с помощью второй камеры
box_cam2 = cam2.project(homography.make_homog(box))

# тест: проецирование точки на плоскость z=0 должно дать тот же результат
point = array([1,1,0,1]).T
print homography.normalize(dot(dot(H,cam1.P),point))
print cam2.project(point)
```

Здесь мы используем изображение с разрешением 747×1000 и сначала генерируем калибровочную матрицу для этого размера. За-

тем строятся точки куба с центром в начале координат. Первые пять точек, созданных функцией `cube_points()`, соответствуют нижней грани, которая в данном случае будет лежать в плоскости маркера $Z = 0$. Первое изображение (слева вверху на рис. 4.4), примерно соответствующее прямому фронтальному ракурсу, примем за эталонное. Поскольку масштаб координат сцены произволен, создадим первую камеру с матрицей

$$P_1 = K\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix},$$

так что ось координат совмещена с камерой и расположена над маркером. Спроецируем первые пять пространственных точек на изображение. С помощью вычисленной гомографии мы можем перенести эти точки на второе изображение. Нарисовав их на графике, мы увидим углы в соответственных позициях маркера (вверху справа на рис. 4.4).

Если в качестве матрицы камеры для второго изображения взять композицию P_1 и H

$$P_2 = HP_1,$$

то точки в плоскости маркера $Z = 0$ будут преобразованы правильно. Это означает, что первые два и четвертый столбец P_2 правильны. Поскольку мы знаем, что левый верхний блок размерности 3×3 должен совпадать с KR, где R – матрица поворота, то можем восстановить третий столбец, умножив P_2 на матрицу, обратную к калибровочной, и заменив третий столбец векторным произведением двух первых.

Для проверки мы можем спроецировать точку на плоскость маркера с помощью новой матрицы и убедиться, что получается такой же результат, как в случае преобразования точки, спроецированной первой камерой, с последующей гомографией. В обоих случаях на консоли должно быть напечатано одно и то же. Визуализировать спроецированные точки можно следующим образом:

```
im0 = array(Image.open('book_frontal.JPG'))
im1 = array(Image.open('book_perspective.JPG'))

# проекция нижней грани на плоскость
figure()
imshow(im0)
```

```
plot(box_cam1[0,:],box_cam1[1,:],linewidth=3)

# перенос проекции с помощью гомографии H
figure()
imshow(im1)
plot(box_trans[0,:],box_trans[1,:],linewidth=3)

# трехмерный куб
figure()
imshow(im1)
plot(box_cam2[0,:],box_cam2[1,:],linewidth=3)

show()
```

В результате должно получиться три рисунка, похожие на те, что изображены на рис. 4.4. Чтобы проделанными вычислениями можно было снова воспользоваться в последующих примерах, мы можем сохранить матрицы камер с помощью модуля Pickle:

```
import pickle

with open('ar_camera.pkl','w') as f:
  pickle.dump(K,f)
  pickle.dump(dot(linalg.inv(K),cam2.P),f)
```

Итак, мы научились вычислять матрицу камеры по заданному плоскостному объекту. Объединив сопоставление признаков, гомографию и калибровку камеры, мы смогли поместить на изображение куб. Умея вычислять положение камеры, мы теперь располагаем всеми инструментами, необходимыми для создания простых приложений дополненной реальности.

4.4. Дополненная реальность

Общим термином *дополненная реальность* называют размещение объектов и информации поверх изображения. Классический пример – размещение трехмерной графической компьютерной модели так, будто она является составной частью сцены и естественно перемещается вместе с камерой в случае видеосъемки. Имея изображение в плоскости маркера, как в предыдущем разделе, мы можем вычислить положение камеры и затем корректно отрисовывать графические компьютерные модели на сцене. В этом последнем разделе главы, посвященной камерам, мы построим простой пример дополненной реальности. Для этого нам понадобятся два инструмента: PyGame и PyOpenGL.

PyGame и PyOpenGL

PyGame – популярный пакет для разработки игр, который умеет работать с окнами, устройствами ввода, событиями и т. п. Исходный код PyGame открыт, а скачать его можно с сайта *http://www.pygame.org/*. На самом деле, это интерфейс из Python к игровому движку SDL. Инструкции по установке приведены в приложении А. Дополнительные сведения о программировании PyGame, см., например, в [21].

PyOpenGL – это интерфейс из Python к системе графического программирования OpenGL, которая входит в дистрибутив почти всех операционных систем и является важнейшей составной частью высокопроизводительных графических приложений. Система OpenGL кросс-платформенная, т. е. одинаково работает в разных операционных системах. Дополнительные сведения о ней можно найти на сайте *http://www.opengl.org/*. На странице *http://www.opengl.org/wiki/Getting_started* приведен список ресурсов для начинающих. PyOpenGL – пакет с открытым исходным кодом, простой в установке; детали см. в приложении Б. Дополнительную информацию можно найти на сайте проекта по адресу *http://pyopengl.sourceforge.net/*.

У нас нет места, чтобы сколько-нибудь подробно рассказать о программировании OpenGL. Вместо этого мы просто продемонстрируем некоторые важные аспекты, например, использование матриц камер и подготовку простой 3D-модели. Хорошие примеры и демонстрации имеются в пакете PyOpenGL-Demo (*http://pypi.python.org/pypi/PyOpenGL-Demo*). С него имеет смысл начать, если вы никогда раньше не работали с PyOpenGL.

Мы хотим поместить 3D-модель на сцену с помощью OpenGL. Чтобы воспользоваться пакетами PyGame и PyOpenGL, мы должны в начале программы импортировать следующие модули:

```
from OpenGL.GL import *
from OpenGL.GLU import *
import pygame, pygame.image
from pygame.locals import *
```

Как видите, нам понадобятся две основные части из OpenGL. Часть GL содержит все функции с префиксом «gl», которыми мы в основном и будем пользоваться. Часть GLU – это библиотека OpenGL Utility, содержащая функциональность верхнего уровня. Она нам будет нужна, главным образом, для настройки проекции камеры. Модуль `pygame` занимается подготовкой окон и событий, а модуль `pygame.image` за-

гружает изображения и создает текстуры OpenGL. Модуль `pygame.locals` нужен для настройки области отображения для OpenGL.

Две главные составные части подготовки сцены в OpenGL – проекционная матрица и матрица вида модели. Начнем с создания этих матриц для камеры с точечной диафрагмой.

От матрицы камеры к формату OpenGL

В OpenGL для представления преобразований (как трехмерного пространства, так и проецирования) используются матрицы 4×4. Это лишь немногим отличается от наших матриц камер размерности 3×4. Однако преобразование между камерой и сценой описывается двумя отдельными матрицами: *GL_PROJECTION* и *GL_MODELVIEW*. Матрица GL_PROJECTION отвечает за свойства формирования изображения и эквивалентна нашей внутренней калибровочной матрице K. А матрица GL_MODELVIEW отвечает за 3D-преобразование соотношений между объектами и камерой. Это примерно соответствует компонентам R и $\mathbf{t}$ нашей матрицы камеры. Одно из различий заключается в том, что начало системы координат предполагается совпадающим с камерой, так что матрица GL_MODELVIEW фактически определяет преобразование, которое помещает объекты перед камерой. При работе с OpenGL есть много тонкостей, мы будем отмечать их в комментариях.

В предположении, что камера откалибрована, т. е. ее калибровочная матрица K известна, для преобразования характеристик камеры с проекционную матрицу OpenGL применяется следующая функция:

```
def set_projection_from_camera(K):
  """ Настроить вид по калибровочной матрице камеры. """

  glMatrixMode(GL_PROJECTION)
  glLoadIdentity()

  fx = K[0,0]
  fy = K[1,1]
  fovy = 2*arctan(0.5*height/fy)*180/pi
  aspect = (width*fy)/(height*fx)

  # определить ближнюю и дальнюю плоскость отсечения
  near = 0.1
  far = 100.0

  # настроить перспективу
  gluPerspective(fovy,aspect,near,far)
  glViewport(0,0,width,height)
```

Мы предполагаем, что калибровочная матрица задана в упрощенной форме (4.3), когда оптический центр совпадает с центром изображения. Первая функция `glMatrixMode()` устанавливает в качестве рабочей матрицу GL_PROJECTION, так что последующие команды модифицируют именно эту матрицу[1]. Затем `glLoadIdentity()` инициализирует эту матрицу единичной, отменяя все предшествующие изменения. После этого мы вычисляем вертикальное поле зрения в градусах, зная высоту изображения, фокусное расстояние камеры и отношение сторон. В определение проекции в OpenGL входит также ближняя и дальняя плоскость отсечения, ограничивающие глубину сцены. Мы задали расстояние до ближней плоскости достаточно малым, чтобы захватить ближайший объект, а в качестве расстояния до дальней плоскости выбрали некоторое большое число. С помощью функции `gluPerspective()` мы задаем проекционную матрицу, а затем говорим, что портом просмотра (по сути дела, тем, что будет показано) является все изображение. Существует также возможность загрузить проекционную матрицу целиком с помощью функции `glLoadMatrixf()`, как матрицу вида модели ниже. Это полезно, когда простой формы калибровочной матрицы недостаточно.

Матрица вида модели (или просто *матрица вида*) должна представлять поворот и параллельный перенос, которые располагают объект перед камерой (как если бы камера находилась в начале координат). Эта матрица 4×4 обычно выглядит так:

$$\begin{bmatrix} R & \mathrm{t} \\ 0 & 1 \end{bmatrix},$$

где R – матрица поворота, столбцы которой совпадают с направлениями трех осей координат, а $\mathbf{t}$ – вектор параллельного переноса. При создании матрицы вида часть R должна содержать все повороты (объекта и системы координат), что достигается перемножением отдельных компонентов.

Следующая функция принимает матрицу камеры с точечной диафрагмой размерности 3×4, из которой исключена калибровочная матрица (путем умножения P на K^{-1}), и создает матрицу вида модели:

```
def set_modelview_from_camera(Rt):
  """ Строит матрицу вида модели по положению камеры. """

  glMatrixMode(GL_MODELVIEW)
```

[1] Это довольно странный способ, но поскольку переключаться можно только между двумя матрицами, GL_PROJECTION и GL_MODELVIEW, то справиться можно.

```
glLoadIdentity()

# повернуть чайник на 90 градусов вокруг оси x, так чтобы ось z
# была направлена вверх
Rx = array([[1,0,0],[0,0,-1],[0,1,0]])

# выбрать в качестве матрицы поворота наилучшее приближение
R = Rt[:,:3]
U,S,V = linalg.svd(R)
R = dot(U,V)
R[0,:] = -R[0,:] # изменить направление оси x

# задать параллельный перенос
t = Rt[:,3]

# задать матрицу вида модели 4*4
M = eye(4)
M[:3,:3] = dot(R,Rx)
M[:3,3] = t

# транспонировать и линеаризовать, получив порядок по столбцам
M = M.T
m = M.flatten()

# заменить матрицу вида новой матрицей
glLoadMatrixf(m)
```

Сначала мы делаем рабочей матрицу GL_MODELVIEW и инициализируем ее. Затем создаем матрицу поворота на 90 градусов, поскольку объект, который мы хотим расположить, необходимо повернуть (как будет видно ниже). Затем гарантируем, что часть матрицы камеры, описывающая поворот, действительно является матрицей поворота, – на случай, если при вычислении матрицы камеры имели место погрешности или шум. Для этого мы выполняем сингулярное разложение и в качестве наилучшего приближения матрицы поворота берем $R = UV^T$. В OpenGL система координат выбрана немного по-другому, поэтому мы меняем направление оси x. После этого матрица вида M получается путем перемножения поворотов. Функция `glLoadMatrixf()` задает матрицу вида модели, принимая 16 ее элементов, расположенных *по столбцам*. Для получения такого порядка нужно выполнить транспонирование и линеаризацию.

Помещение виртуальных на изображение

Первым делом нам нужно установить изображение, на которое мы хотим помещать виртуальные объекты, в качестве фонового.

В OpenGL для этого создается четырехугольник, заполняющий вид целиком. Проще всего это сделать, нарисовав четырехугольник, когда обе матрицы – проекционная и модели – сброшены, так что координаты изменяются от –1 до 1 в каждом направлении.

Следующая функция загружает изображение, преобразует его в текстуру OpenGL и накладывает эту текстуру на прямоугольник:

```
def draw_background(imname):
  """ Нарисовать фоновое изображение с помощью прямоугольника. """

  # загрузить фоновое изображение (в формате BMP) в текстуру OpenGL
  bg_image = pygame.image.load(imname).convert()
  bg_data = pygame.image.tostring(bg_image,"RGBX",1)
  glMatrixMode(GL_MODELVIEW)
  glLoadIdentity()
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

  # привязать текстуру
  glEnable(GL_TEXTURE_2D)
  glBindTexture(GL_TEXTURE_2D,glGenTextures(1))
  glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,width,height,0,GL_RGBA,
               GL_UNSIGNED_BYTE, bg_data)
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)

  # создать прямоугольник, заполняющий все окно
  glBegin(GL_QUADS)
  glTexCoord2f(0.0,0.0); glVertex3f(-1.0,-1.0,-1.0)
  glTexCoord2f(1.0,0.0); glVertex3f( 1.0,-1.0,-1.0)
  glTexCoord2f(1.0,1.0); glVertex3f( 1.0, 1.0,-1.0)
  glTexCoord2f(0.0,1.0); glVertex3f(-1.0, 1.0,-1.0)
  glEnd()

  # очистить текстуру
  glDeleteTextures(1)
```

Эта функция сначала вызывает функции PyGame для загрузки изображения и получения его строкового представления, которое можно передать PyOpenGL. Затем мы сбрасываем матрицу вида и очищаем буфер цвета и глубины. Далее мы привязываем текстуру, чтобы ее можно было наложить на прямоугольник, и задаем интерполяцию. Углы прямоугольника имеют координаты 1 и –1. Обратите внимание, что координаты изображения текстуры изменяются от 0 до 1. Напоследок мы очищаем текстуру, чтобы она не мешала последующим операциям рисования.

Теперь все готово к помещению объектов на сцену. Мы будем использовать пример, с которого начинаются все курсы машинной гра-

фики, – чайником из Юты (*http://en.wikipedia.org/wiki/Utah_teapot*). У этого чайника богатая история, и он включен в модуль GLUT в качестве одной из стандартных форм:

```
from OpenGL.GLUT import *
glutSolidTeapot(size)
```

Эти предложения создают модель сплошного чайника с относительным размером *size*.

Следующая функция задает цвет и другие свойства, чтобы получился симпатичный красный чайник:

```
def draw_teapot(size):
  """ Нарисовать красный чайник с центром в начале координат. """

  glEnable(GL_LIGHTING)
  glEnable(GL_LIGHT0)
  glEnable(GL_DEPTH_TEST)
  glClear(GL_DEPTH_BUFFER_BIT)

  # нарисовать красный чайник
  glMaterialfv(GL_FRONT,GL_AMBIENT,[0,0,0,0])
  glMaterialfv(GL_FRONT,GL_DIFFUSE,[0.5,0.0,0.0,0.0])
  glMaterialfv(GL_FRONT,GL_SPECULAR,[0.7,0.6,0.6,0.0])
  glMaterialf(GL_FRONT,GL_SHININESS,0.25*128.0)
  glutSolidTeapot(size)
```

В первых двух строчках включается освещение и источники света. Источники света нумеруются GL_LIGHT0, GL_LIGHT1 и т. д. В этом примере мы будем использовать только один источник. Функция `glEnable()` включает различные возможности OpenGL. Все они определены константами, записываемыми заглавными буквами. Для выключения возможностей предназначена функция `glDisable()`. Следующей включается проверка глубины, для того чтобы при рисовании объектов учитывалась их глубина (чтобы далекие объекты не рисовались перед близкими), и очищается буфер глубины. Затем задаются свойства материала, например рассеянный и отраженный свет. В последней строке на сцену добавляется сплошной чайник из Юты с заданными свойствами материала.

Собираем все вместе

Ниже приведена полная программа создания изображения, показанного на рис. 4.5 (предполагается, что все описанные выше функции находятся в том же файле).

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import pygame, pygame.image
from pygame.locals import *
import pickle

width,height = 1000,747

def setup():
  """ Настроить окно и окружение pygame. """

  pygame.init()
  pygame.display.set_mode((width,height),OPENGL | DOUBLEBUF)
  pygame.display.set_caption('OpenGL AR demo')

# загрузить данные о камере
with open('ar_camera.pkl','r') as f:
  K = pickle.load(f)
  Rt = pickle.load(f)

setup()
draw_background('book_perspective.bmp')
set_projection_from_camera(K)
set_modelview_from_camera(Rt)
draw_teapot(0.02)

while True:
  event = pygame.event.poll()
  if event.type in (QUIT,KEYDOWN):
     break
  pygame.display.flip()
```

Сначала программа загружает из файла в формате Pickle калибровочную матрицу камеры и часть матрицы камеры, описывающую поворот и параллельный перенос. Предполагается, что эти данные были сохранены, как описано на стр. 123. Функция `setup()` инициализирует PyGame, устанавливает размер окна равным размеру изображения и задает в качестве области рисования окно OpenGL с двойной буферизацией. Затем программа загружает фоновое изображение и заполняет им окно. Настраиваются матрицы камеры и вида модели и, наконец, в нужном месте рисуется чайник.

В PyGame обработка событий производится в бесконечном цикле опроса изменений. Бывают события клавиатуры, мыши и других типов. В данном случае мы проверяем, что приложение было завершено или нажата какая-то клавиша, и выходим из цикла. Функция `pygame.display.flip()` рисует объекты на экране.

Рис. 4.5. Дополненная реальность. Размещение графической компьютерной модели на сцене с изображением книги с использованием параметров камеры, вычисленных в результате сопоставления признаков: чайник из Юты нарисован в точке, совмещенной с началом координат (вверху); контрольная проверка, показывающая, где находится начало координат (внизу)

Результат должен выглядеть, как на рис. 4.5. Как видим, ориентация установлена правильно (чайник параллелен граням куба на рис. 4.4). Чтобы убедиться в правильности размещения, можно сделать чайник совсем маленьким, задав меньшее значение параметра *size*. Чайник должен оказаться рядом с вершиной куба [0, 0, 0] на рис. 4.4.

Загрузка моделей

Прежде чем закончить эту главу, мы коснемся еще одной, последней, детали: загрузки и показа 3D-моделей. Среди рецептов PyGame есть скрипт загрузки моделей в формате .obj, его можно найти на странице *http://www.pygame.org/wiki/OBJFileLoader*. Прочитать о формате .obj и соответствующем формате файла материалов можно на странице *http://en.wikipedia.org/wiki/Wavefront_.obj_file*.

Посмотрим, как этим пользоваться, на простом примере. Возьмем бесплатную модель игрушечного самолета со страницы *http://www.oyonale.com/modeles.php*[2]. Скачайте ее вариант в формате .obj и сохраните в файле *toyplane.obj*. Конечно, можете вместо этой модели взять любую понравившуюся, код при этом не изменится.

В предположении, что вы назвали скачанный файл *objloader.py*, добавьте в него следующую функцию, которой мы пользовались в примере с чайником:

```
def load_and_draw_model(filename):
  """ Загрузить модель из obj-файла с помощью objloader.py.
    Предполагается, что существует mtl-файл материалов с таким
    же именем. """

  glEnable(GL_LIGHTING)
  glEnable(GL_LIGHT0)
  glEnable(GL_DEPTH_TEST)
  glClear(GL_DEPTH_BUFFER_BIT)

  # установить цвет модели
  glMaterialfv(GL_FRONT,GL_AMBIENT,[0,0,0,0])
  glMaterialfv(GL_FRONT,GL_DIFFUSE,[0.5,0.75,1.0,0.0])
  glMaterialf(GL_FRONT,GL_SHININESS,0.25*128.0)

  # загрузить из файла
  import objloader
  obj = objloader.OBJ(filename,swapyz=True)
  glCallList(obj.gl_list)
```

Как и раньше, мы задаем освещение и цветовые свойства модели. Затем загружаем файл модели в объект типа `OBJ` и выполняем вызовы OpenGL, прописанные в файле.

[2] Модели любезно предоставил Жилль Тран (по лицензии Creative Commons).

Текстуру и свойства материала можно задать в соответствующем mtl-файле. На самом деле, модуль `objloader` требует, чтобы файл материала присутствовал. Вместо того чтобы модифицировать скрипт загрузки, мы просто создадим простейший файл материала, в котором задан только цвет.

Создайте файл *toyplane.mtl*, содержащий такие строки:

```
newmtl lightblue
Kd 0.5 0.75 1.0
illum 1
```

Здесь задается рассеянный свет и светло-синий цвет объекта. Не забудьте заменить строку «usemtl» в obj-файле такой:

```
usemtl lightblue
```

Добавление текстур мы оставим в качестве упражнения. Если заменить вызов `draw_teapot()` в примере выше таким:

```
load_and_draw_model('toyplane.obj')
```

то должно быть нарисовано окно, показанное на рис. 4.6.

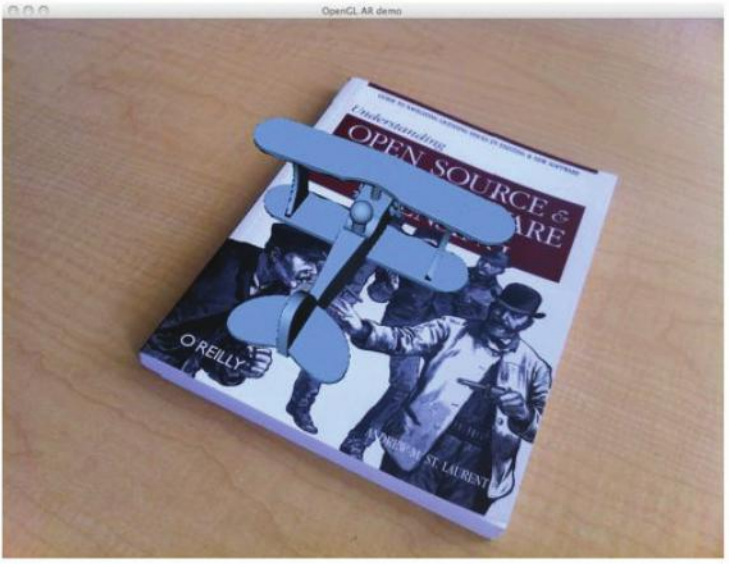


Рис. 4.6. Загрузка 3D-модели из obj-файла и размещение ее на сцене поверх книги с применением параметров камеры, вычисленных путем сопоставления признаков

Более глубоко заниматься OpenGL и дополненной реальностью в этой книге мы не можем. Но зная, как откалибровать камеру, вычислить ее положение, преобразовать свойства камеры в формат OpenGL и отрисовывать модели на сцене, вы обладаете достаточной подготовкой, чтобы самостоятельно продолжить изыскания в области дополненной реальности. В следующей главе мы продолжим заниматься моделью камеры и вычислением трехмерной структуры и положения камеры без использования маркеров.

Упражнения

1. Модифицируйте пример программы отслеживания перемещения камеры на рис. 4.2, так чтобы преобразовывались точки, а не камера. Должен получиться такой же график. Поэкспериментируйте с разными преобразованиями, каждый раз нанося результаты на график.
2. Некоторые из оксфордских многовидовых наборов данных сопровождаются матрицами камер. Вычислите положения камеры для одного из таких наборов и постройте график траектории камеры. Согласуется ли он с тем, что вы видите на изображениях?
3. Сделайте несколько снимков сцены с плоскостным маркером. Сопоставьте признаки с полным видом спереди, чтобы вычислить положение камеры для каждого снимка. Нарисуйте траекторию камеры и плоскость маркера. Если хотите, добавьте признаковые точки.
4. В примере дополненной реальности мы предполагали, что объект размещается в начале координат, и к матрице вида модели применяли только матрицу камеры. Модифицируйте пример, поместив в разные места несколько объектов. Для этого придется добавить в матрицу преобразование объекта. Например, поместите на маркер несколько чайников, расположенных в узлах прямоугольной сетки.
5. Ознакомьтесь с онлайновой документацией по формату файлов модели .obj и разберитесь, как работать с текстурированными моделями. Найдите такую модель (или создайте собственную) и поместите ее на сцену.

ГЛАВА 5. Многовидовая геометрия

В этой главе мы покажем, как обрабатывать несколько видов изображения и как, используя взаимосвязи между ними, восстановить положения камер и трехмерную структуру. Если есть несколько фотографий, снятых с разных точек, то координаты точек на трехмерной сцене и положения камер можно вычислять, сопоставляя признаки. Мы познакомимся с необходимыми для этого инструментами и приведем пример полной трехмерной реконструкции. А в последней части главы покажем, как реконструировать плотную карту глубины по стереоизображениям.

5.1. Эпиполярная геометрия

Многовидовая геометрия – это дисциплина, изучающая связи между камерами и признаками в ситуации, когда имеются соответствия между несколькими изображениями, снятыми с разных точек. Признаками изображения обычно являются особые точки, и в этой главе мы будем рассматривать только этот случай. На практике наиболее важна двухвидовая геометрия.

При наличии двух видов сцены и соответственных точек в них существуют геометрические ограничения на точки изображения, являющиеся следствием относительной ориентации и свойств камер, а также положения точек в пространстве. Описанием этих геометрических соотношений занимается *эпиполярная геометрия*. Ниже мы дадим очень краткий обзор основных компонентов, которые нам понадобятся. Дополнительные сведения см. в [13].

Если нет никакой информации о камерах, то существует принципиальная неоднозначность: если точка $\mathbf{X}$ преобразуется некоторой гомографией H (4×4) в точку $H\mathbf{X}$, то, будучи снята камерой PH^{-1}, она совпадает на изображении с исходной точкой, снятой камерой P. В терминах уравнения камеры это можно записать в виде:

$$\lambda \mathbf{x} = P\mathbf{X} = PH^{-1}H\mathbf{X} = \hat{P}\hat{\mathbf{X}}.$$

Из-за этой неоднозначности в двухвидовой геометрии всегда можно применять к камерам гомографическое преобразование, если это упрощает вычисления. Часто в роли такой гомографии выступает изометрическое преобразование, которое просто преобразует систему координат. Полезно совмещать начало координат с первой камерой, а ось располагать в направлении съемки, так что:

$$P_1 = K_1 [I \mid 0] \quad \text{и} \quad P_2 = K_2[R \mid \mathbf{t}].$$

Здесь используется та же нотация, что в главе 4: K_1 и K_2 – калибровочные матрицы, R описывает поворот второй камеры, а $\mathbf{t}$ – ее параллельный перенос. При таких матрицах камер можно вывести условие, при котором точка $\mathbf{X}$ проецируется в точки изображения $\mathbf{x}_1$ и $\mathbf{x}_2$ (при использовании камер P_1 и P_2 соответственно). Именно это условие позволяет восстановить матрицы камер по соответственным точкам изображения.

Должно удовлетворяться следующее соотношение:

$$\mathbf{x}_2^T F \mathbf{x}_1 = 0, \tag{5.1}$$

где

$$F = K_2^T S_t R K_1^{-1},$$

а S_t – кососимметричная матрица.

$$S_t = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}, \tag{5.2}$$

Уравнение (5.1) называется *эпиполярным ограничением*. Матрица F в нем называется *фундаментальной матрицей*, и, как легко видеть, она выражается через элементы матриц обеих камер (матрицу относительного поворота R и вектор параллельного переноса $\mathbf{t}$). Ранг фундаментальной матрицы равен 2, а определитель $\det(F) = 0$. Мы воспользуемся этим в алгоритмах вычисления F.

Приведенные уравнения означают, что матрицы камер можно восстановить, зная матрицу F, которую, в свою очередь, можно вычислить по соответственным точкам, как будет показано ниже. Если калибровочные матрицы (K_1 и K_2) неизвестны, то матрицы камер восстанав-

ливаются только с точностью до проективного преобразования. При известной калибровке реконструкция будет метрической. *Метрической* называется трехмерная реконструкция, правильно представляющая расстояния и углы[1].

Прежде чем применять теорию к изображениям, нам понадобится еще один геометрический факт. Если зафиксировать точку на одном изображении, например $\mathbf{x}_2$ во втором виде, то уравнение (5.1) определяет прямую линию на первом изображении, потому что

$$\mathrm{x}_2^T F\,\mathrm{x}_1 = \mathrm{l}_1^T \mathrm{x}_1 = 0.$$

Уравнение $\mathrm{l}_1^T \mathrm{x}_1 = 0$ определяет прямую, на которой лежат все точки $\mathbf{x}_1$ первого изображения, удовлетворяющие уравнению. Она называется *эпиполярной прямой*, соответствующей точке $\mathbf{x}_2$. Это означает, что точка, соответствующая $\mathbf{x}_2$, должна лежать на этой прямой. Поэтому фундаментальная матрица может помочь в поиске соответствий, т. к. ограничивает поиск этой прямой.

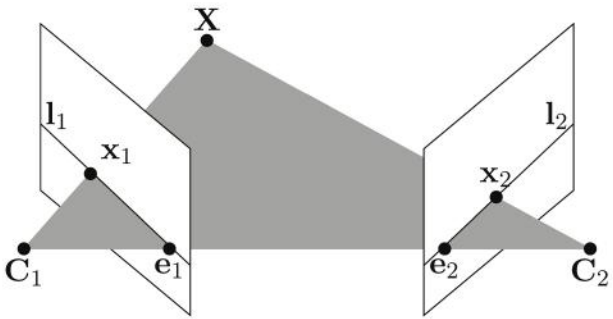


Рис. 5.1. Иллюстрация эпиполярной геометрии. Точки $\mathbf{x}_1$ и $\mathbf{x}_2$ – проекции **X** в двух видах. Линия, соединяющая центры двух камер, $\mathbf{C}_1$ и $\mathbf{C}_2$, пересекает плоскости изображений в двух эпиполюсах, $\mathbf{e}_1$ и $\mathbf{e}_2$. Прямые $\mathbf{l}_1$ и $\mathbf{l}_2$ называются эпиполярными прямыми

Все эпиполярные прямые пересекаются в точке **e**, называемой *эпиполюсом*. На самом деле, эпиполюс – это точка изображения, соответствующая проекции центра другой камеры. В зависимости от относительной ориентации камер эта точка может оказаться вне изображения. Поскольку эпиполюс принадлежит каждой эпиполярной прямой, он должен удовлетворять уравнению $F\mathbf{e}_1 = 0$. Следовательно,

[1] Абсолютный масштаб при реконструкции не восстанавливается, но это редко составляет проблему.

его можно вычислить, как собственный вектор F с нулевым собственным значением. Второй эпиполюс можно вычислить из уравнения $\mathbf{e}_2^T = 0$. Эпиполюсы и эпиполярные прямые показаны на рис. 5.1.

Демонстрационный набор данных

В следующих разделах для экспериментов и иллюстрации алгоритмов нам понадобится набор данных, содержащий точки изображения, точки трехмерного пространства и матрицы камер. Мы будем использовать один из оксфордских многовидовых наборов, предлагаемых на странице *http://www.robots.ox.ac.uk/~vgg/data/data-mview.html*. Скачайте zip-файл с набором «Merton College 1». Показанный ниже скрипт загружает данные:

```
import camera

# загрузить несколько изображений
im1 = array(Image.open('images/001.jpg'))
im2 = array(Image.open('images/002.jpg'))

# загрузить 2D-точки из каждого вида в список
points2D = [loadtxt('2D/00'+str(i+1)+'.corners').T for i in range(3)]

# загрузить 3D-точки
points3D = loadtxt('3D/p3d').T

# загрузить соответствия
corr = genfromtxt('2D/nview-corners',dtype='int',missing='*')

# загрузить камеры в список объектов Camera
P = [camera.Camera(loadtxt('2D/00'+str(i+1)+'.P')) for i in range(3)]
```

Здесь мы загружаем первые два изображения (из трех), все особые точки[2] для всех трех видов, реконструированные 3D-точки, соответствующие точкам изображения во всех трех видах, и, наконец, матрицы камер (для чего использован класс Camera из предыдущей главы). Для чтения текстовых файлов в массивы `NumPy` применяется функция `loadtxt()`. Некоторые данные о соответствии отсутствуют, потому что некоторые точки не видны, а другим не нашлось соответствия во всех видах. При загрузке соответствий следует помнить об этом. Функция `genfromtxt()` решает проблему, подставляя вместо отсутствующих данных (они обозначены в файле символом `'*'`) значение –1.

Чтобы выполнить этот скрипт и загрузить все данные, удобно сохранить код в файле, например *load_vggdata.py*, и с помощью функ-

[2] На самом деле, угловые точки Харриса, см. раздел 2.1.

ции `execfile()` запускать его в начале своего скрипта или в ходе экспериментов:

```
execfile('load_vggdata.py')
```

Посмотрим, на что похожи данные. Попробуем спроецировать 3D-точки на один вид и сравним результаты с наблюдавшимися точками изображения:

```
# представить 3D-точки в однородных координатах и спроецировать
X = vstack( (points3D,ones(points3D.shape[1])) )
x = P[0].project(X)

# нанести на график точки из вида 1
figure()
imshow(im1)
plot(points2D[0][0],points2D[0][1],'*')
axis('off')

figure()
imshow(im1)
plot(x[0],x[1],'r.')
axis('off')
show()
```

В результате будет создан график для первого вида и точек изображения на нем; для сравнения спроецированные точки показаны на отдельном рисунке. На рис. 5.2 мы видим получившиеся графики. Присмотревшись внимательно, мы обнаружим, что на втором графике, где показаны проекции 3D-точек, точек больше, чем на первом. Это особые точки, реконструированные по видам 2 и 3, но не обнаруженные на виде 1.

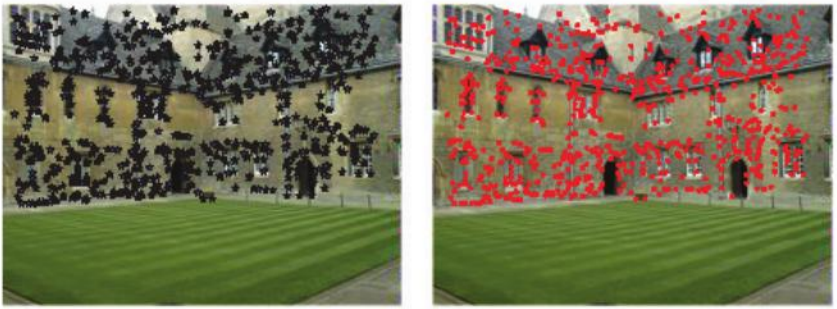


Рис. 5.2. Набор данных Merton1 из оксфордских многовидовых наборов; вид 1 с точками изображения (слева); вид 1 с проекциями 3D-точек (справа)

Построение трехмерных графиков в Matplotlib

Для визуализации трехмерных реконструкций нужно уметь строить трехмерные графики. Модуль `mplot3d`, входящий в состав `Matplotlib`, предоставляет средства построения трехмерных графиков, содержащих точки, прямые и кривые линии, поверхности и прочие графические элементы, а также возможность вращения и масштабирования – с помощью находящихся в окне элементов управления.

Для построения трехмерного графика нужно добавить ключевое слово `projection="3d"` к объекту оси:

```
from mpl_toolkits.mplot3d import axes3d

fig = figure()
ax = fig.gca(projection="3d")

# сгенерировать трехмерные тестовые данные
X,Y,Z = axes3d.get_test_data(0.25)

# нанести точки на трехмерный график
ax.plot(X.flatten(),Y.flatten(),Z.flatten(),'o')
show()
```

Функция `get_test_data()` генерирует тестовые данные на равномерной сетке x, y со стороной, равной заданному параметру. Линеаризация этих данных дает три списка точек, которые можно передать функции `plot()`. В результате будет построена поверхность, на которой лежат 3D-точки. Попробуйте и убедитесь сами.

Далее мы можем нанести на график набор данных Merton и посмотреть, как выглядят входящие в него точки:

```
# нанесение на график 3D-точек
from mpl_toolkits.mplot3d import axes3d
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(points3D[0],points3D[1],points3D[2],'k.')
```

На рис. 5.3 показаны 3D-точки из трех разных видов. Окно рисунка и элементы управления выглядят, как обычные окна графиков с двумерными изображениями, дополненные инструментом для вращения в трехмерном пространстве.

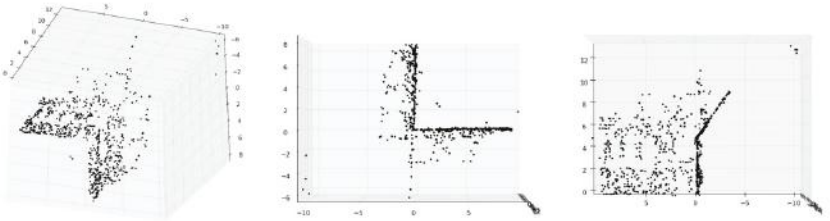


Рис. 5.3. 3D-точки из оксфордского многовидового набора данных Merton1, визуализированные средствами `Matplotlib`: вид сверху и сбоку (слева); вид сверху, на котором видны стены здания и точки на крыше (в центре); вид сбоку, на котором показан профиль одной стены и вид спереди точек на другой стене (справа)

Вычисление F – восьмиточечный алгоритм

Восьмиточечный алгоритм предназначен для вычисления фундаментальной матрицы по соответственным точкам. Ниже приведено его краткое описание, а детали можно найти в [14] и [13].

Эпиполярное ограничение (5.1) можно записать в виде системы линейных уравнений:

$$\begin{bmatrix} x_2^1 x_1^1 & x_2^1 y_1^1 & x_2^1 w_1^1 & \cdots & w_2^1 w_1^1 \\ x_2^2 x_1^2 & x_2^2 y_1^2 & x_2^2 w_1^2 & \cdots & w_2^2 w_1^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_2^n x_1^n & x_2^n y_1^n & x_2^n w_1^n & \cdots & w_2^n w_1^n \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ \vdots \\ F_{33} \end{bmatrix} = A\mathbf{f} = 0,$$

где $\mathbf{f}$ содержит элементы матрицы F, $\mathbf{x}_1^i = \left[x_1^i, y_1^i, w_1^i\right]$ и $\mathbf{x}_2^i = \left[x_2^i, y_2^i, w_2^i\right]$ – пара соответственных точек, а n – число таких пар. Фундаментальная матрица содержит девять элементов, но поскольку масштаб произволен, нужно только восемь уравнений. Следовательно, для вычисления F необходимо иметь восемь соответственных точек, отсюда и название алгоритма.

Создайте файл *sfm.py* и добавьте в него следующую функцию, которая реализует восьмиточечный алгоритм путем минимизации $|A\mathbf{f}\|$:

```
def compute_fundamental(x1,x2):
  """ Вычисляет фундаментальную матрицу по соответственным
    точкам (x1,x2 - массивы 3*n) с применением нормированного
    восьмиточечного алгоритма.
    Каждая строка строится в виде
    [x'*x, x'*y, x', y'*x, y'*y, y', x, y, 1] """

  n = x1.shape[1]
  if x2.shape[1] != n:
     raise ValueError("Неодинаковое число точек.")

  # построить матрицу
  A = zeros((n,9))
  for i in range(n):
     A[i] = [x1[0,i]*x2[0,i], x1[0,i]*x2[1,i], x1[0,i]*x2[2,i],
        x1[1,i]*x2[0,i], x1[1,i]*x2[1,i], x1[1,i]*x2[2,i],
        x1[2,i]*x2[0,i], x1[2,i]*x2[1,i], x1[2,i]*x2[2,i] ]

  # найти линейное решение, минимизирующее среднеквадратичную ошибку
  U,S,V = linalg.svd(A)
  F = V[-1].reshape(3,3)

  # наложить ограничения на F
  # сделать ранг F равным 2, обнулив последнее сингулярное значение
  U,S,V = linalg.svd(F)
  S[2] = 0
  F = dot(U,dot(diag(S),V))

  return F
```

Как обычно, мы вычисляем решение, минимизирующее среднеквадратичную ошибку, применяя сингулярное разложение. Поскольку ранг найденной матрицы может быть не равен 2, а это необходимое условие для фундаментальной матрицы, мы заменяем результат ближайшей аппроксимацией с рангом 2, обнуляя последнее сингулярное значение. Это стандартный прием, о котором полезно знать. В этой функции опущен важный этап – нормировка координат изображения, что может привести к проблемам при вычислениях с конечной точностью. Но оставим это на потом.

Эпиполюс и эпиполярные прямые

В начале этого раздела было сказано, что эпиполюс удовлетворяет уравнению $F\mathbf{e}_1 = 0$ и может быть вычислен, если известно ядро F. Добавьте в файл *sfm.py* следующую функцию:

```
def compute_epipole(F):
  """ Вычисляет (правый) эпиполюс по фундаментальной матрице F
```

```
    (для левого эпиполюса транспонировать F) """

  # вернуть ядро F (Fx=0)
  U,S,V = linalg.svd(F)
  e = V[-1]
  return e/e[2]
```

Если нужно найти эпиполюс, соответствующий коядру[3] (т. е. эпиполюс в другом изображении), то матрицу F нужно предварительно транспонировать.

Протестировать эти две функции на первых двух видах демонстрационного набора данных можно следующим образом:

```
import sfm

# индекс для точек из первых двух видов
ndx = (corr[:,0]>=0) & (corr[:,1]>=0)

# получить координаты и преобразовать в однородные
x1 = points2D[0][:,corr[ndx,0]]
x1 = vstack( (x1,ones(x1.shape[1])) )
x2 = points2D[1][:,corr[ndx,1]]
x2 = vstack( (x2,ones(x2.shape[1])) )

# вычислить F
F = sfm.compute_fundamental(x1,x2)

# вычислить эпиполюс
e = sfm.compute_epipole(F)

# построение графика
figure()
imshow(im1)

# рисуем каждую прямую по отдельности, при этом получаются
# симпатичные цвета
for i in range(5):
  sfm.plot_epipolar_line(im1,F,x2[:,i],e,False)
axis('off')

figure()
imshow(im2)

# рисуем каждую точку по отдельности, при этом получаются такие
# же цвета, как для прямых
for i in range(5):
  plot(x2[0,i],x2[1,i],'o')
```

[3] Коядром матрицы F называется множество векторов $\mathbf{x}$ таких, что $\mathbf{x}^T F = 0$. Коядро матрицы F совпадает с ядром матрицы F^T. – *Прим. перев.*

```
axis('off')

show()
```

Первым делом все соответственные точки обоих изображений преобразуются в однородные координаты. Здесь мы просто читаем их из файла, но на практике нужно было бы извлекать и сопоставлять признаки, как в главе 2. Отсутствующие значения в списке соответствий *corr* представлены числом –1, поэтому, выбирая только значения, большие или равные нулю, мы получаем точки, присутствующие в обоих видах. Два условия объединяются оператором `&`, применяемым к массивам.

Наконец, мы рисуем первые пять эпиполярных прямых на первом виде и соответствующие им точки на втором. При этом мы пользуемся такой вспомогательной функцией:

```
def plot_epipolar_line(im,F,x,epipole=None,show_epipole=True):
  """ Нарисовать эпиполюс и эпиполярную прямую F*x=0 на
     изображении.
     F - фундаментальная матрица, а x - точка на другом
     изображении."""

  m,n = im.shape[:2]
  line = dot(F,x)

  # параметр и значения эпиполярной прямой
  t = linspace(0,n,100)
  lt = array([(line[2]+line[0]*tt)/(-line[1]) for tt in t])

  # берем только точки прямой, находящиеся внутри изображения
  ndx = (lt>=0) & (lt<m)
  plot(t[ndx],lt[ndx],linewidth=2)

  if show_epipole:
     if epipole is None:
        epipole = compute_epipole(F)
     plot(epipole[0]/epipole[2],epipole[1]/epipole[2],'r*')
```

Эта функция параметризует прямую диапазоном координат на оси *x* и исключает части прямой выше и ниже границы изображения. Если последний параметр *show_epipole* равен true, то рисуется также эпиполюс (при этом он вычисляется, если не задан извне). Графики показаны на рис. 5.4. На обоих графиках используется один и тот же набор цветов, поэтому точка на одном изображении принадлежит соответствующей ей прямой того же цвета на другом изображении.

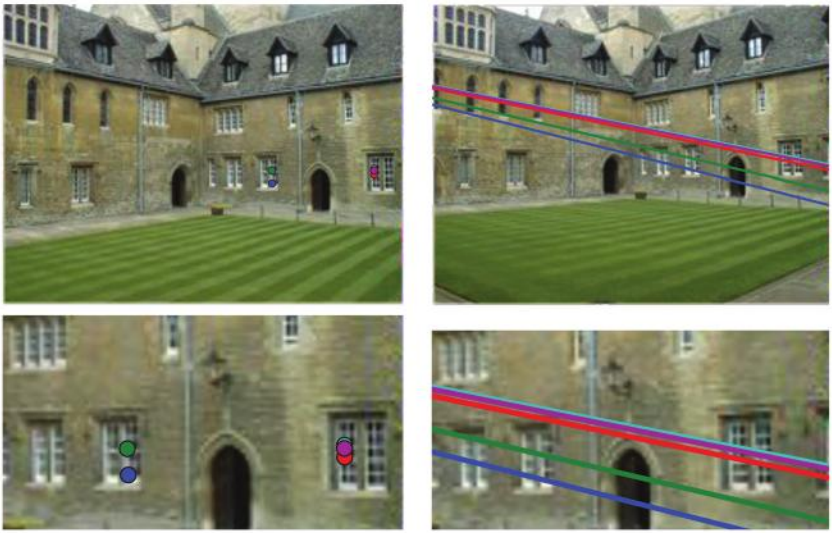


Рис. 5.4. На виде 1 показаны эпиполярные прямые для пяти точек на виде 2 из набора данных Merton1. В нижнем ряду показан крупный план окрестности точек. Видно, что прямые сходятся к точке, находящейся слева от изображения. Прямые показывают, где можно найти точки на другом изображении (цвета точек и соответствующих им прямых совпадают)

5.2. Вычисления, относящиеся к камерам и трехмерной структуре

В предыдущем разделе рассматривались связи между видами и вопрос о вычислении фундаментальной матрицы и эпиполярных прямых. А сейчас мы кратко опишем инструменты, необходимые для вычислений, относящиеся к камерам и трехмерной структуре.

Триангуляция

Если известны матрицы камер, то множество соответственных точек можно триангулировать для восстановления положения этих точек в пространстве. Базовый алгоритм довольно прост.

Если имеются два вида, снятые камерами с матрицами P_1 и P_2, и $\mathbf{x}_1$ и $\mathbf{x}_2$ – проекции одной и той же точки $\mathbf{X}$ на эти виды (в однородных координатах), то из уравнения камеры (4.1) следует:

$$\begin{bmatrix} P_1 & -x_1 & 0 \\ P_2 & 0 & -x_2 \end{bmatrix} \begin{bmatrix} \mathrm{X} \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = 0.$$

Из-за зашумленности изображения, погрешностей в определении матриц камер и других причин у этой системы уравнений может не быть точного решения. Но с помощью сингулярного разложения мы можем найти приближенное положение точки, минимизирующее среднеквадратичную ошибку.

Добавьте в файл *sfm.py* функцию, вычисляющую *триангуляцию по методу наименьших квадратов* для пары соответственных точек.

```
def triangulate_point(x1,x2,P1,P2):
  """ Триангуляция для пары точек по методу наименьших квадратов. """

  M = zeros((6,6))
  M[:3,:4] = P1
  M[3:,:4] = P2
  M[:3,4] = -x1
  M[3:,5] = -x2

  U,S,V = linalg.svd(M)
  X = V[-1,:4]

  return X / X[3]
```

Первые четыре элемента последнего собственного вектора – однородные координаты точки в пространстве. Для триангуляции при большом числе точек можете добавить такую вспомогательную функцию:

```
def triangulate(x1,x2,P1,P2):
  """ Двухвидовая триангуляция точек в массивах x1,x2
    (3*n в однородных координатах). """

  n = x1.shape[1]
  if x2.shape[1] != n:
     raise ValueError("Неодинаковое число точек.")

  X = [ triangulate_point(x1[:,i],x2[:,i],P1,P2) for i in range(n)]
  return array(X).T
```

Эта функция принимает два массива точек и возвращает массив трехмерных координат. Попробуем выполнить триангуляцию для набора данных Merton1:

```
import sfm

# индекс для точек из первых двух видов
ndx = (corr[:,0]>=0) & (corr[:,1]>=0)

# получить координаты и преобразовать в однородные
x1 = points2D[0][:,corr[ndx,0]]
x1 = vstack( (x1,ones(x1.shape[1])) )
x2 = points2D[1][:,corr[ndx,1]]
x2 = vstack( (x2,ones(x2.shape[1])) )

Xtrue = points3D[:,ndx]
Xtrue = vstack( (Xtrue,ones(Xtrue.shape[1])) )

# проверить первые 3 точки
Xest = sfm.triangulate(x1,x2,P[0].P,P[1].P)
print Xest[:,:3]
print Xtrue[:,:3]

# построить график
from mpl_toolkits.mplot3d import axes3d
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(Xest[0],Xest[1],Xest[2],'ko')
ax.plot(Xtrue[0],Xtrue[1],Xtrue[2],'r.')
axis('equal')
show()
```

Эта функция триангулирует соответственные точки из первых двух видов и печатает на консоли координаты первых трех реконструированных точек, а под ними истинные значения. Распечатка выглядит следующим образом:

```
[[ 1.03743725  1.56125273  1.40720017]
 [-0.57574987 -0.55504127 -0.46523952]
 [ 3.44173797  3.44249282  7.53176488]
 [ 1.          1.          1.        ]]
[[ 1.0378863   1.5606923   1.4071907 ]
 [-0.54627892 -0.5211711  -0.46371818]
 [ 3.4601538   3.4636809   7.5323397 ]
 [ 1.          1.          1.        ]]
```

Реконструированные точки близки к истинным. На рис. 5.5 показан график; как видно, совпадение точек вполне приемлемое.

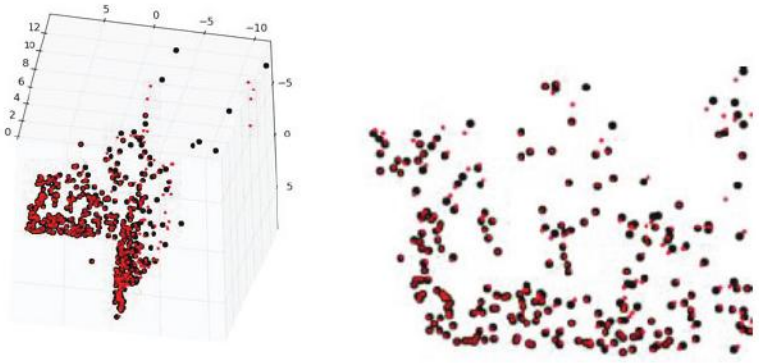


Рис. 5.5. Результат триангуляции точек с применением матриц камеры и известного соответствия. Восстановленные точки показаны сплошными кружками, а истинные – полыми. Вид сверху и сбоку (слева). Крупным планом показаны точки на одной из стен здания (справа)

Вычисление матрицы камеры по точкам в пространстве

Зная 3D-точки и их проекции на изображение, можно вычислить матрицу камеры P, применив алгоритм прямого линейного преобразования (DLT). По сути дела, это задача, обратная триангуляции, иногда ее называют *калибровкой камеры* (camera resectioning). Для восстановления матрицы камеры опять-таки применяется метод наименьших квадратов.

Согласно уравнению камеры (4.1), каждая видимая 3D-точка $\mathbf{X}_i$ (в однородных координатах) проецируется в точку изображения $\mathbf{x}_i = [x_i, y_i, 1]$, так что $\lambda_i \mathbf{x}_i = P\mathbf{X}_i$ и соответственные точки удовлетворяют соотношению

$$\begin{bmatrix} \mathbf{X}_1^T & 0 & 0 & -x_1 & 0 & 0 & \dots \\ 0 & \mathbf{X}_1^T & 0 & -y_1 & 0 & 0 & \dots \\ 0 & 0 & \mathbf{X}_1^T & -1 & 0 & 0 & \dots \\ \mathbf{X}_2^T & 0 & 0 & 0 & -x_2 & 0 & \dots \\ 0 & \mathbf{X}_2^T & 0 & 0 & -y_2 & 0 & \dots \\ 0 & 0 & \mathbf{X}_2^T & 0 & -1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \mathbf{p}_3^T \\ \lambda_1 \\ \lambda_2 \\ \vdots \end{bmatrix} = 0,$$

где $\mathbf{p}_1$, $\mathbf{p}_2$ и $\mathbf{p}_3$ – три строки матрицы P. Это соотношение можно записать компактнее:

$$M\mathbf{v} = 0.$$

Теперь оценку матрицы камеры можно вычислить с помощью сингулярного разложения. При описанном выше способе построения матриц код пишется прямолинейно. Добавьте следующую функцию в файл *sfm.py*:

```
def compute_P(x,X):
  """ Вычислить матрицу камеры по соответственным парам точек на
     плоскости и в пространстве (в однородных координатах). """

  n = x.shape[1]
  if X.shape[1] != n:
     raise ValueError("Неодинаковое число точек.")

  # создать матрицу для применения метода DLT
  M = zeros((3*n,12+n))
  for i in range(n):
     M[3*i,0:4] = X[:,i]
     M[3*i+1,4:8] = X[:,i]
     M[3*i+2,8:12] = X[:,i]
     M[3*i:3*i+3,i+12] = -x[:,i]

  U,S,V = linalg.svd(M)

  return V[-1,:12].reshape((3,4))
```

Эта функция принимает точки изображения и соответствующие им 3D-точки и строит описанную выше матрицу M. Первые 12 элементов последнего собственного вектора и есть элементы матрицы камеры, которая возвращается после изменения формы.

Протестируем эту функцию на нашем наборе данных. Приведенный ниже скрипт выбирает точки, присутствующие в первом виде (используя информацию об отсутствующих значениях в списке соответствия), преобразует их в однородные координаты и вычисляет оценку матрицы камеры:

```
import sfm, camera

corr = corr[:,0] # вид 1
ndx3D = where(corr>=0)[0] # отсутствующие значения равны -1
ndx2D = corr[ndx3D]

# выбрать видимые точки и преобразовать в однородные координаты
x = points2D[0][:,ndx2D] # вид 1
x = vstack( (x,ones(x.shape[1])) )
```

```
X = points3D[:,ndx3D]
X = vstack( (X,ones(X.shape[1])) )

# оценить P
Pest = camera.Camera(sfm.compute_P(x,X))

# сравнить!
print Pest.P / Pest.P[2,3]
print P[0].P / P[0].P[2,3]
xest = Pest.project(X)

# построить график
figure()
imshow(im1)
plot(x[0],x[1],'bo')
plot(xest[0],xest[1],'r.')
axis('off')

show()
```

Для проверки матрицы камер распечатываются на консоли в нормированном виде (после деления на последний элемент). Распечатка показана ниже:

```
[[ 1.06520794e+00 -5.23431275e+01  2.06902749e+01  5.08729305e+02]
 [ -5.05773115e+01 -1.33243276e+01 -1.47388537e+01  4.79178838e+02]
 [  3.05121915e-03 -3.19264684e-02 -3.43703738e-02  1.00000000e+00]]
[[  1.06774679e+00  -5.23448212e+01   2.06926980e+01  5.08764487e+02]
 [ -5.05834364e+01  -1.33201976e+01  -1.47406641e+01  4.79228998e+02]
 [  3.06792659e-03  -3.19008054e-02  -3.43665129e-02  1.00000000e+00]]
```

В первых трех строчках представлена наша оценка матрицы камеры, а в последних трех – матрица, вычисленная авторами набора данных. Как видим, результаты почти совпали. Далее функция проецирует 3D-точки с помощью найденной оценки камеры и строит график. Результат показан на рис. 5.6, где истинные точки представлены полыми кружками, а проекции с помощью оцененной камеры – сплошными.

Вычисление матрицы камеры по фундаментальной матрице

При наличии двух видов матрицы камер можно восстановить по фундаментальной матрице. В предположении, что матрица первой камеры нормирована, т. е. представлена в виде $P_1 = [I \mid 0]$, задача состоит в нахождении матрицы второй камеры P_2. Есть два случая: калиброванный и некалиброванный.

Рис. 5.6. Проекции точек на вид 1, вычисленные с помощью оценки матрицы камеры

Некалиброванный случай – проективная реконструкция

Без знания внутренних параметров камеры ее матрицу можно восстановить только с точностью до проективного преобразования. Это означает, что если для реконструкции 3D-точек используются две камеры, то реконструкция будет верна с точностью до проективного преобразования (т. е. можно получить любое решение из семейства проективных искажений сцены). Таким образом, углы и расстояния не сохраняются.

Поэтому в некалиброванном случае матрицу второй камеры можно выбирать с точностью до проективного преобразования (3×3). Вот самый простой выбор:

$$P_2 = [S_e F \mid \mathbf{e}],$$

где $\mathbf{e}$ – левый эпиполюс, $\mathbf{e}^T F = 0$, а S – кососимметричная матрица (5.2). Напомним, что триангуляция с помощью этой матрицы почти наверняка даст искажения, например, реконструкция может получиться скошенной.

Вот как это выглядит в коде:

```
def compute_P_from_fundamental(F):
  """ Вычисляет матрицу второй камеры (в предположении, что
     P1 = [I 0])по фундаментальной матрице. """

  e = compute_epipole(F.T) # left epipole
  Te = skew(e)
  return vstack((dot(Te,F.T).T,e)).T
```

Мы воспользовались следующей вспомогательной функцией `skew()`:

```
def skew(a):
  """ Кососимметричная матрица A такая, что a x v = Av для любого v. """

  return array([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])
```

Добавьте обе функции в файл *sfm.py*.

Калиброванный случай – метрическая реконструкция

Если калибровка известна, то реконструкция будет метрической, сохраняющей свойства евклидова пространства (за исключением глобального параметра масштаба). С точки зрения реконструкции трехмерной сцены калиброванный случай самый интересный.

Зная калибровочную матрицу K, мы можем применить обратную к ней матрицу K^{-1} к точкам изображения $\mathbf{x}_K = K^{-1}\mathbf{x}$, так что уравнение камеры принимает вид:

$$\mathbf{x}_K = K^{-1}K[R\,|\,\mathbf{t}]\mathbf{X} = [R\,|\,\mathbf{t}]\mathbf{X},$$

в новых координатах изображения. Точки в этих координатах удовлетворяют тому же фундаментальному уравнению, что и раньше:

$$\mathbf{x}_{K_2}^T F\,\mathbf{x}_{K_1} = 0.$$

Фундаментальная матрица для откалиброванных изображений называется *существенной матрицей* (essential matrix) и обычно обозначается E, а не F, чтобы было понятно, что речь идет о калиброванном случае, и координаты изображения нормированные.

Матрицы камер, восстановленные по существенной матрице, сохраняют метрические соотношения, но возможных решения четыре. Однако лишь в одном из них сцена расположена перед обеими камерами, так что выбрать правильное решение нетрудно.

Ниже приведена реализация алгоритма, вычисляющего все четыре решения (детали см. в [13]). Добавьте следующую функцию в файл *sfm.py*:

```
def compute_P_from_essential(E):
  """ Вычисляет матрицу второй камеры (в предположении, что
     P1 = [I|0]) по существенной матрице. Возвращает список из
     четырех возможных матриц камер. """

  # гарантировать, что ранг E равен 2
```

```
U,S,V = svd(E)
if det(dot(U,V))<0:
   V = -V
E = dot(U,dot(diag([1,1,0]),V))

# создать матрицы (Hartley стр. 258)
Z = skew([0,0,-1])
W = array([[0,-1,0],[1,0,0],[0,0,1]])

# вернуть все четыре решения
P2 = [vstack((dot(U,dot(W,V)).T,U[:,2])).T,
      vstack((dot(U,dot(W,V)).T,-U[:,2])).T,
      vstack((dot(U,dot(W.T,V)).T,U[:,2])).T,
      vstack((dot(U,dot(W.T,V)).T,-U[:,2])).T]

return P2
```

Первым делом эта функция предпринимает меры к тому, чтобы ранг существенной матрицы был равен 2 (с двумя равными ненулевыми сингулярными значениями), а затем находит все четыре решения, применяя метод из книги [13]. Возвращается список, содержащий четыре матрицы камер. Как выбрать из них правильную, мы покажем в примере ниже.

На этом мы завершаем изложение теории, необходимой для вычисления трехмерных реконструкций по набору изображений.

5.3. Многовидовая реконструкция

Посмотрим, как с помощью всего вышеизложенного вычислить трехмерную реконструкцию по двум изображениям. Эту операцию часто называют *определением структуры по движению* (structure from motion, SfM), поскольку представление о трехмерной структуре дает движение камеры (или нескольких камер).

В предположении, что камера откалибрована, нужно выполнить следующие шаги.

1. Определить особые точки и сопоставить их.
2. Вычислить фундаментальную матрицу по соответственным точкам.
3. Вычислить матрицы камер по фундаментальной матрице.
4. Триангулировать 3D-точки.

У нас уже есть все необходимые инструменты, но нужен устойчивый способ вычисления фундаментальной матрицы в случае, когда среди установленных соответствий между точками изображений есть ошибки.

Устойчивое вычисление фундаментальной матрицы

Напомним, что в разделе 3.3 нам был нужен устойчивый способ вычисления гомографий, и точно так же сейчас нам необходим способ вычисления фундаментальной матрицы при наличии шума и неправильных соответствий. Как и раньше, воспользуемся методом RANSAC, но на этот раз в комбинации с восьмиточечным алгоритмом. Следует сказать, что восьмиточечный алгоритм не работает для плоскостных сцен, т. е. его нельзя применить к сцене, где все точки лежат в одной плоскости.

Добавьте следующий класс в файл *sfm.py*:

```
class RansacModel(object):
  """ Класс для нахождения фундаментальной матрицы с помощью модуля
     ransac.py со страницы http://www.scipy.org/Cookbook/RANSAC. """

  def __init__(self,debug=False):
     self.debug = debug

  def fit(self,data):
     """ Вычислить фундаментальную матрицу по восьми выбранным
        соответствиям. """

     # транспонировать и разделить данные на два набора точек
     data = data.T
     x1 = data[:3,:8]
     x2 = data[3:,:8]

     # вычислить и вернуть фундаментальную матрицу
     F = compute_fundamental_normalized(x1,x2)
     return F

  def get_error(self,data,F):
     """ Вычислить x^T F x для всех соответствий, вернуть
        погрешность для каждой преобразованной точки. """

     # транспонировать и разделить данные на два набора точек
     data = data.T
     x1 = data[:3]
```

```
    x2 = data[3:]

    # для оценки погрешности используем расстояние Сэмпсона
    Fx1 = dot(F,x1)
    Fx2 = dot(F,x2)
    denom = Fx1[0]**2 + Fx1[1]**2 + Fx2[0]**2 + Fx2[1]**2
    err = ( diag(dot(x1.T,dot(F,x2))) )**2 / denom

    # вернуть погрешности для всех точек
    return err
```

Как и прежде, нам нужны методы `fit()` и `get_error()`. В качестве меры погрешности здесь используется расстояние Сэмпсона (см. [13]). Метод `fit()` теперь выбирает восемь точек и применяет нормированный вариант восьмиточечного алгоритма:

```
def compute_fundamental_normalized(x1,x2):
  """ Вычисляет фундаментальную матрицу по соответственным
     точкам (массивы 3*n x1,x2) с помощью нормированного
     восьмиточечного алгоритма. """

  n = x1.shape[1]
  if x2.shape[1] != n:
     raise ValueError("Неодинаковое число точек.")

  # нормировать координаты изображения
  x1 = x1 / x1[2]
  mean_1 = mean(x1[:2],axis=1)
  S1 = sqrt(2) / std(x1[:2])
  T1=array([[S1,0,-S1*mean_1[0]],[0,S1,-S1*mean_1[1]],[0,0,1]])
  x1 = dot(T1,x1)

  x2 = x2 / x2[2]
  mean_2 = mean(x2[:2],axis=1)
  S2 = sqrt(2) / std(x2[:2])
  T2=array([[S2,0,-S2*mean_2[0]],[0,S2,-S2*mean_2[1]],[0,0,1]])
  x2 = dot(T2,x2)

  # вычислить F по нормированным координатам
  F = compute_fundamental(x1,x2)

  # обратить операцию нормировки
  F = dot(T1.T,dot(F,T2))

  return F/F[2,2]
```

Эта функция нормирует точки изображения, так чтобы получить нулевое среднее и фиксированную дисперсию. Теперь класс можно передать функции. Добавьте следующую функцию в файл *sfm.py*:

```
def F_from_ransac(x1,x2,model,maxiter=5000,match_theshold=1e-6):
    """ Устойчивое вычисление фундаментальной матрицы F по соответственным
        точкам с помощью метода RANSAC (модуль ransac.py со страницы
        http://www.scipy.org/Cookbook/RANSAC).
        вход: x1,x2 (массивы 3*n) - точки в однородных координатах. """

    import ransac

    data = vstack((x1,x2))

    # вычислить F и вернуть вместе с индексом регулярных точек
    F,ransac_data = ransac.ransac(data.T,model,8,maxiter,match_theshold,20,
                                  return_all=True)
    return F, ransac_data['inliers']
```

Здесь мы возвращаем наилучшую оценку фундаментальной матрицы *F*, а также индекс регулярных точек, чтобы знать, какие соответствия оказались совместимы с *F*. По сравнению с вычислением гомографии мы увеличили подразумеваемое по умолчанию максимальное число итераций и изменили порог соответствия, который раньше выражался в пикселях, а теперь – в терминах расстояния Сэмпсона.

Пример трехмерной реконструкции

В этом разделе мы рассмотрим полный пример реконструкции трехмерной сцены от начала до конца. Мы будем использовать две фотографии, снятые камерой с известной калибровкой. На них изображена знаменитая тюрьма Алькатрас (рис. 5.7)[4].

Рис. 5.7. Пример двух изображений сцены, снятых с разных точек

Для удобства изложения разобьем код на несколько частей. Сначала мы выделяем признаки, сопоставляем их и вычисляем фундаментальную матрицу и матрицы камер.

[4] Изображения любезно предоставил Карл Олсон (*http://www.maths.lth.se/matematiklth/personal/calle/*).

```
import homography
import sfm
import sift

# калибровочная матрица
K = array([[2394,0,932],[0,2398,628],[0,0,1]])

# загрузить изображения и вычислить признаки
im1 = array(Image.open('alcatraz1.jpg'))
sift.process_image('alcatraz1.jpg','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

im2 = array(Image.open('alcatraz2.jpg'))
sift.process_image('alcatraz2.jpg','im2.sift')
l2,d2 = sift.read_features_from_file('im2.sift')

# сопоставить признаки
matches = sift.match_twosided(d1,d2)
ndx = matches.nonzero()[0]

# преобразовать в однородные координаты и нормировать с помощью inv(K)
x1 = homography.make_homog(l1[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
x2 = homography.make_homog(l2[ndx2,:2].T)

x1n = dot(inv(K),x1)
x2n = dot(inv(K),x2)

# вычислить E с помощью RANSAC
model = sfm.RansacModel()
E,inliers = sfm.F_from_ransac(x1n,x2n,model)

# вычислить матрицы камер (P2 содержит список из четырех решений)
P1 = array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])
P2 = sfm.compute_P_from_essential(E)
```

Калибровка известна, поэтому мы в начале программы просто зашили матрицу K в код. Как и в предыдущих примерах, выбираем соответственные точки. Затем нормируем их с помощью матрицы K^{-1} и выполняем оценивание методом RANSAC в сочетании с нормированным восьмиточечным алгоритмом. Поскольку точки нормированы, это дает нам существенную матрицу. Сохраняем индекс регулярных точек, т. к. он понадобится в дальнейшем. По существенной матрице вычисляем четыре возможных решения для матрицы второй камеры.

Из списка матриц камер выбираем ту, для которой после триангуляции большинство точек сцены оказывается перед обеими камерами:

```
# выбрать решение, в котором точки находятся перед камерами
ind = 0
maxres = 0
for i in range(4):
  # триангулировать регулярные точки и вычислить глубину для каждой камеры
  X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[i])
  d1 = dot(P1,X)[2]
  d2 = dot(P2[i],X)[2]
  if sum(d1>0)+sum(d2>0) > maxres:
     maxres = sum(d1>0)+sum(d2>0)
     ind = i
     infront = (d1>0) & (d2>0)

# триангулировать регулярные точки и исключить те, что не находятся
# перед обеими камерами
X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[ind])
X = X[:,infront]
```

Мы перебираем в цикле все четыре решения и на каждой итерации триангулируем 3D-точки, соответствующие регулярным точкам. Знак глубины дает третье значение в каждой точке изображения после проецирования триангулированной точки $\mathbf{X}$ обратно на изображения. Мы сохраняем индекс решения с наибольшим числом положительных значений глубины и массив булевых признаков для каждой точки в наилучшем решении, чтобы можно было отобрать только те, что действительно находятся перед камерами. Из-за шума и погрешностей оценивания может случиться, что некоторые точки все-таки окажутся за одной из камер, пусть даже матрицы камер правильные.

Теперь можно нанести реконструкцию на график:

```
# Трехмерный график
from mpl_toolkits.mplot3d import axes3d

fig = figure()
ax = fig.gca(projection='3d')
ax.plot(-X[0],X[1],X[2],'k.')
axis('off')
```

На трехмерных графиках, созданных с помощью `mplot3d`, направление первой оси противоположно тому, что принято в нашей системе координат, поэтому изменяем знак.

Теперь можно нарисовать результат обратного проецирования на каждый вид:

```
# нанести на график проекцию X
import camera

# спроецировать 3D-точки
```

```
cam1 = camera.Camera(P1)
cam2 = camera.Camera(P2[ind])
x1p = cam1.project(X)
x2p = cam2.project(X)

# выполнить операцию, обратную нормировке с помощью K
x1p = dot(K,x1p)
x2p = dot(K,x2p)

figure()
imshow(im1)
gray()
plot(x1p[0],x1p[1],'o')
plot(x1[0],x1[1],'r.')
axis('off')

figure()
imshow(im2)
gray()
plot(x2p[0],x2p[1],'o')
plot(x2[0],x2[1],'r.')
axis('off')
show()
```

После проецирования 3D-точек необходимо обратить выполненную в начале нормировку, для чего надо умножить результат на калибровочную матрицу.

Результат показан на рис. 5.8. Как видите, обратно спроецированные точки не точно совпадают с исходными особыми точками, но весьма близки к ним. Можно уточнить матрицы камер, чтобы улучшить качество реконструкции и обратного проецирования, но это уже выходит за рамки простого примера.

Обобщения и случай более двух видов

Существуют различные обобщения многовидовой реконструкции, которые мы не можем подробно рассмотреть в этой книге. Все же перечислим некоторые направления со ссылками на литературу для дальнейшего чтения.

Случай более двух видов

Когда имеется более двух видов одной сцены, трехмерная реконструкция обычно получается более точной и детальной. Но поскольку фундаментальная матрица связывает только два вида, процесс несколько отличается.

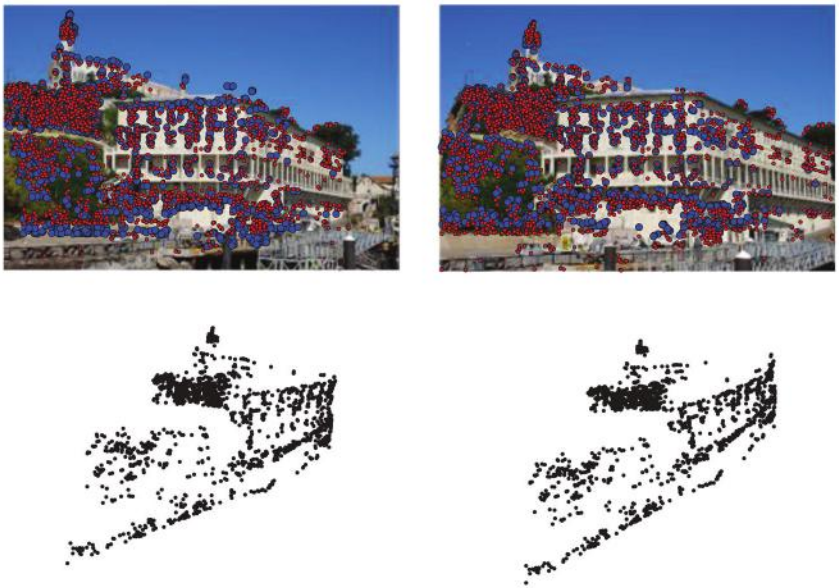


Рис. 5.8. Пример вычисления трехмерной реконструкции по двум изображениям с использованием соответственных точек: два изображения с особыми точками, показанными черными кружками, и обратно спроецированными 3D-точками реконструкции, показанными белыми кружками (вверху); трехмерная реконструкция (внизу)

Для последовательностей видеоизображений можно использовать хронологический порядок и сопоставлять признаки в последовательных парах кадров. При переходе к следующей паре необходимо инкрементно добавлять относительную ориентацию (подобно тому, как мы добавляли гомографии в примере построения панорам на рис. 3.12). Обычно такой подход хорошо работает, и для эффективного нахождения соответствий можно использовать трассировку (о трассировке см. раздел 10.4). Проблема в том, что с увеличением числа видов накапливаются ошибки. Это можно исправить на заключительном этапе оптимизации (см. ниже).

Для статических изображений можно применить такой подход: найти центральное опорное изображение и вычислить матрицы всех остальных камер относительно него. Другой способ – вычислить матрицы камер и трехмерную реконструкцию для одной пары изображений, а затем инкрементно добавлять новые изображения и 3D-точки

(см., например, [34]). Попутно отметим, что существуют способы вычисления позиций камер и трехмерной реконструкции по трем видам одновременно (см., например, [13]), но при большем числе видов все равно необходим инкрементный подход.

Блочное уравнивание

Из нашего простого примера трехмерной реконструкции на рис. 5.8 ясно, что при определении позиций восстановленных точек и вычислении матриц камер по найденной оценке фундаментальной матрицы неизбежны ошибки. При большем числе видов ошибки накапливаются. Поэтому на последнем этапе многовидовой реконструкции часто пытаются минимизировать ошибки обратного проецирования с помощью оптимизации позиций 3D-точек и параметров камеры. Этот процесс называется *блочным уравниванием* (bundle adjustment). Детали можно найти в работах [13] и [35], а краткий обзор – в статье википедии по адресу *http://en.wikipedia.org/wiki/Bundle_adjustment*.

Автокалибровка

В случае неоткалиброванной камеры иногда возможно вычислить параметры калибровки по особым точкам изображения. Эта процедура называется *автокалибровкой*. Есть много алгоритмов, различающихся допущениями о параметрах калибровочной матрицы камеры и типами имеющихся изображений (с сопоставленными признаками, с параллельными прямыми или плоскостями и т. д.). Интересующегося читателя отсылаем к книгам [13] и [26, глава 6].

И кстати о калибровке: в состав системы Bundler SfM (*http://phototour.cs.washington.edu/bundler/*) входит полезный скрипт *extract_focal.pl*, который пользуется справочной таблицей популярных камер и оценивает фокусное расстояние на основе EXIF-данных изображения.

5.4. Стереоизображения

Частным случаем многовидовых изображений является *стереозрение* (или *стереоизображения*), когда две камеры, снимающие одну и ту же сцену, расположены на горизонтальной плоскости. Если камеры настроены так, что плоскости обоих изображений совпадают, а их строки выровнены по вертикали, то пара изображений называется *ректифицированной*. Такая конфигурация часто встречается в робототехнике и называется *стереоспаркой* (stereo rig).

Любую конфигурацию стереокамер можно ректифицировать, если деформировать изображения в общую плоскость, так чтобы эпиполярными прямыми стали строки изображения (стереоспарка обычно изначально конструируется, так чтобы получались ректифицированные стереопары). Эта тема выходит за рамки книги, но интересующийся читатель может найти подробности в книгах [13, стр. 303] или [3, стр. 430].

В предположении, что два изображения ректифицированы, соответствия следует искать только в общих строках. После того как соответственная точка найдена, ее глубину (координату Z) можно вычислить непосредственно по горизонтальному смещению, которому она обратно пропорциональна:

$$Z = \frac{f\,b}{x_l - x_r},$$

где f – фокусное расстояние ректифицированного изображения, b – расстояние между центрами камер, а x_l и x_r – абсциссы соответственных точек левого и правого изображения. Расстояние между центрами камер называется *базисом*. На рис. 5.9 показано расположение камер для ректифицированной стереосъемки.

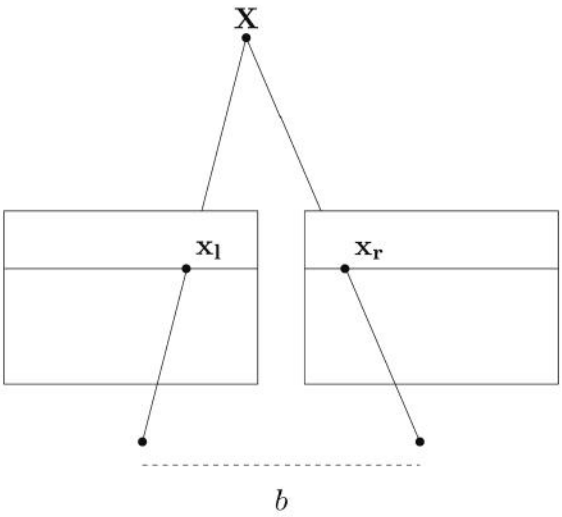


Рис. 5.9. Конфигурация для ректифицированной стереосъемки, когда соответственные точки находятся в одних и тех же строках изображений

Стереореконструкцией (или *плотной реконструкцией глубины*) называется задача восстановления карты глубины (или карты диспаратности), когда требуется оценить глубину (или диспаратность) каждого пикселя изображения. Это классическая задача компьютерного зрения, для решения которой существует много алгоритмов. На странице Мидлбери-колледжа, посвященной стереозрению (*http://vision.middlebury.edu/stereo/*), публикуется обновляемый обзор лучших алгоритмов с кодом и описанием многих реализаций. В следующем разделе мы реализуем алгоритм стереореконструкции на основе нормированной взаимной корреляции.

Вычисление карт диспаратности

В описываемом ниже алгоритме реконструкции мы опробуем диапазон смещений и запоминаем наилучшее смещение каждого пикселя, выбирая то, для которого оптимальна оценка, основанная на нормированной взаимной корреляции в локальной окрестности. Иногда эту идею называют заметанием плоскостью (plane sweeping), потому что каждый шаг вычисления смещения соответствует плоскости, расположенной на некоторой глубине. Хотя этот метод и не является последним словом в стереореконструкции, он прост и дает приличные результаты.

Нормированную взаимную корреляцию можно эффективно вычислить путем сплошного применения к изображениям. Это несколько отличается от случая, когда мы применяли ее к далеко отстоящим соответственным точкам в главе 2. Мы хотим вычислить нормированную взаимную корреляцию (NCC) в блоке вокруг каждого пикселя (по сути дела, в локальной окрестности). Для этого формулу NCC в окрестности пикселя (2.3) можно переписать в виде

$$\mathrm{ncc}(I_1, I_2) = \frac{\sum_x (I_1(x) - \mu_1)(I_2(x) - \mu_2)}{\sqrt{\sum_x (I_1(x) - \mu_1)^2 \sum_x (I_2(x) - \mu_2)^2}},$$

где исключена нормировочная константа (она здесь не нужна), а суммирование производится по всем пикселям в локальной окрестности данного.

И это нужно проделать для каждого пикселя изображения. Все три суммы берутся по локальному блоку и могут быть эффективно вычислены с помощью фильтров изображения, как делалось для размы-

тия и вычисления производных. Функция `uniform_filter()` в модуле `ndimage.filters` вычисляет эти суммы по прямоугольному блоку.

Ниже приведена функция, которая выполняет заметание плоскостью и возвращает оптимальную диспаратность для каждого пикселя. Создайте файл *stereo.py* и добавьте в него эту функцию:

```
def plane_sweep_ncc(im_l,im_r,start,steps,wid):
  """ Найти карту диспаратности с применением нормированной
      взаимной корреляции. """

  m,n = im_l.shape

  # массивы для хранения сумм
  mean_l = zeros((m,n))
  mean_r = zeros((m,n))
  s = zeros((m,n))
  s_l = zeros((m,n))
  s_r = zeros((m,n))

  # массив для хранения глубин плоскостей
  dmaps = zeros((m,n,steps))

  # вычислить среднее по блоку
  filters.uniform_filter(im_l,wid,mean_l)
  filters.uniform_filter(im_r,wid,mean_r)

  # нормированные изображения
  norm_l = im_l - mean_l
  norm_r = im_r - mean_r

  # пробовать разные диспаратности
  for displ in range(steps):
     # сдвинуть левое изображение вправо, вычислить суммы
     filters.uniform_filter(roll(norm_l,-displ-start)*norm_r,wid,s)
     filters.uniform_filter(roll(norm_l,-displ-start)*roll(norm_l,
                            -displ-start),wid, s_l)
     filters.uniform_filter(norm_r*norm_r,wid,s_r)

  # сохранить оценки ncc
  dmaps[:,:,displ] = s/sqrt(s_l*s_r)

  # выбрать наилучшую глубину для каждого пикселя
  return argmax(dmaps,axis=2)
```

Первым делом нужно создать массивы для хранения результатов фильтрации, поскольку функция `uniform_filter()` принимает их в качестве аргументов. Затем мы создаем массив для хранения плоскостей, чтобы можно было применить функцию `argmax()` к

последнему измерению и найти наилучшую глубину для каждого пикселя. Функция организует цикл по *steps* смещениям, начиная со смещения *start*. Одно изображение сдвигается с помощью функции `roll()`, и с помощью фильтрации вычисляются все три входящие в NCC суммы.

Ниже приведен полный код загрузки изображений и вычисления карты диспаратности с применением этой функции.

```
import stereo

im_l=array(Image.open('scene1.row3.col3.ppm').convert('L'),'f')
im_r=array(Image.open('scene1.row3.col4.ppm').convert('L'),'f')

# начальное смещение и количество шагов
steps = 12
start = 4

# ширина блока для ncc
wid = 9

res = stereo.plane_sweep_ncc(im_l,im_r,start,steps,wid)

import scipy.misc
scipy.misc.imsave('depth.png',res)
```

Здесь мы сначала загружаем классическую пару изображений «tsukuba» и преобразуем их в полутоновые. Затем задаются параметры, необходимые для функцию заметания плоскостью, количество рассматриваемых смещений, начальное значение смещения и ширина блока для NCC. Этот метод работает довольно быстро, по крайней мере, по сравнению с сопоставлением признаков с помощью нормированной взаимной корреляции. Объясняется это тем, что для вычислений применяются фильтры.

Описанный подход работает и для других фильтров. Равномерный фильтр назначает всем пикселям в квадратном блоке равные веса, но иногда для вычисления NCC предпочтительнее другие фильтры. Ниже показан вариант с фильтром Гаусса, который порождает более гладкие карты диспаратности. Добавьте его в файл *stereo.py*:

```
def plane_sweep_gauss(im_l,im_r,start,steps,wid):
  """ Найти карту диспаратности с применением нормированной
     взаимной корреляции с фильтром Гаусса. """

  m,n = im_l.shape

  # массивы для хранения сумм
```

```
    mean_l = zeros((m,n))
    mean_r = zeros((m,n))
    s = zeros((m,n))
    s_l = zeros((m,n))
    s_r = zeros((m,n))

    # массив для хранения глубин плоскостей
    dmaps = zeros((m,n,steps))

    # вычислить среднее
    filters.gaussian_filter(im_l,wid,0,mean_l)
    filters.gaussian_filter(im_r,wid,0,mean_r)

    # нормированные изображения
    norm_l = im_l - mean_l
    norm_r = im_r - mean_r

    # пробовать разные диспаратности
    for displ in range(steps):
        # сдвинуть левое изображение вправо, вычислить суммы
        filters.gaussian_filter(roll(norm_l,-displ-start)*norm_r,wid,0,s)
        filters.gaussian_filter(roll(norm_l,-displ-start)*roll(norm_l,
                                -displ-start),wid,0,s_l)
        filters.gaussian_filter(norm_r*norm_r,wid,0,s_r)

    # сохранить оценки ncc
    dmaps[:,:,displ] = s/sqrt(s_l*s_r)

    # выбрать наилучшую глубину для каждого пикселя
    return argmax(dmaps,axis=2)
```

Код такой же, как для равномерного фильтра за исключением дополнительного аргумента функции фильтрации `gaussian_filter()`, которой нужно передать нуль, означающий, что нас интересует стандартный фильтр Гаусса, а не его производные (подробности см. на стр. 41).

Эта функция используется так же, как предыдущая. На рис. 5.10 и 5.11 показаны результаты применения обеих реализаций алгоритма заметания плоскостью к эталонным стереопарам. Изображения взяты из работ [29] и [30] и имеются на странице *http://vision.middlebury.edu/stereo/data/*. Мы воспользовались изображениями «tsukuba» и «cones», задав параметр *wid* равным 9 для стандартной версии и равным 3 для версии с фильтром Гаусса. В верхнем ряду показана пара изображений, в нижнем ряду слева – результат стандартного заметания плоскостью, а справа – результат заметания с фильтром Гаусса.

Как видите, во втором случае меньше шума, но и меньше деталей, чем в первом.

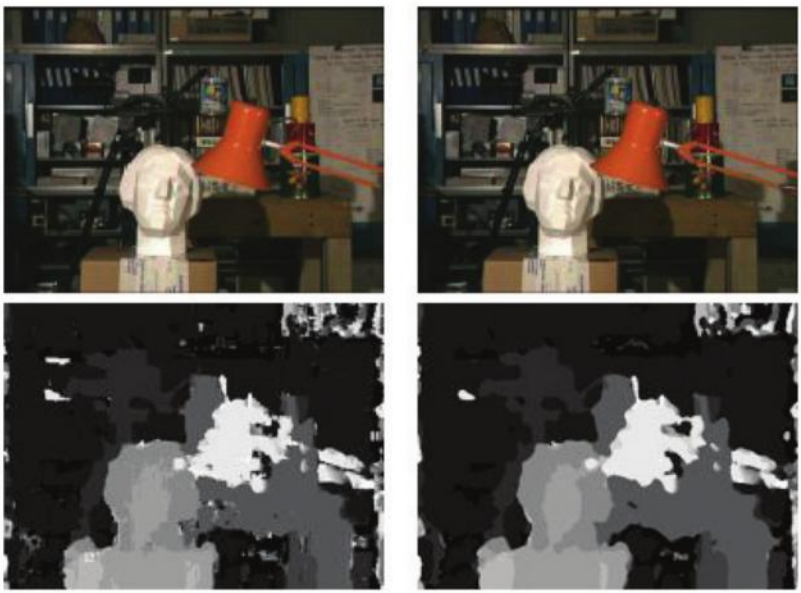


Рис. 5.10. Пример вычисления карт диспаратности для стереопары методом нормированной взаимной корреляции

Упражнения

1. Примените описанную в этой главе технику для проверки соответствий в примере с Белым домом на стр. 74 (а еще лучше к своему собственному примеру) и посмотрите, сможете ли вы улучшить результаты.
2. Вычислите соответственные признаки для какой-нибудь пары изображений и оцените фундаментальную матрицу. Воспользовавшись эпиполярными прямыми, выполните второй проход и найдите дополнительные соответствия, поискав наилучшие соответствия на эпиполярной прямой для каждого признака.
3. Возьмите набор, содержащий три или более изображений. Выберите одну пару и вычислите 3D-точки и матрицы камер. Сопоставьте признаки с остальными изображениями и найдите соответствия. Затем возьмите 3D-точки для найденных соответствий

и вычислите матрицы камер для остальных изображений, произведя калибровку. Нанесите на график позиции 3D-точек и камер. Используйте собственный набор изображений или один из оксфордских многовидовых наборов.

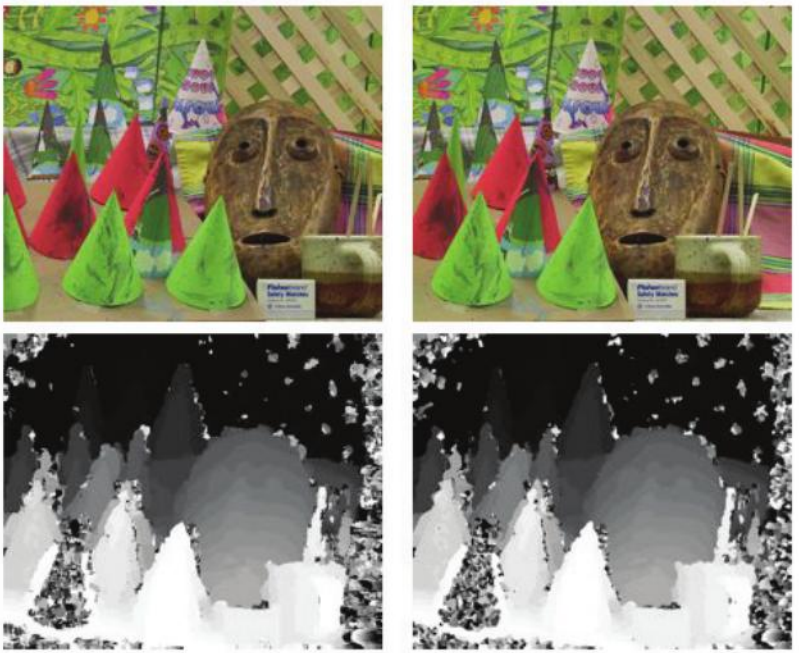


Рис. 5.11. Пример вычисления карт диспаратности для стереопары методом нормированной взаимной корреляции

4. Реализуйте вариант алгоритма стереореконструкции, в котором используется не нормированная взаимная корреляция, а сумма квадратов разностей, а фильтрация производится так же, как в варианте с NCC.
5. Попробуйте сгладить карты глубины с помощью алгоритма очистки от шумов ROF из раздела 1.5. Поэкспериментируйте с размером блоков в методе взаимной корреляции, чтобы получить резкие границы с таким уровнем шума, который можно устранить с помощью сглаживания.
6. Один из способов повышения качества карт диспаратности заключается в сравнении диспаратностей, получающихся при сдвиге левого изображения вправо и при сдвиге правого изображения

влево. При этом сохраняются только согласованные части. Таким образом удается, например, очистить части, где присутствует окклюзия. Реализуйте эту идею и сравните результаты с однонаправленным заметанием плоскостью.

7. В Нью-Йорской публичной библиотеке есть много исторических стереофотографий. Покопайтесь в галерее по адресу *http://stereo.nypl.org/gallery* и скачайте те, что вам понравятся (для этого щелкните по изображению правой кнопкой мыши и сохраните JPEG-файл). Изображения должны быть уже ректифицированы. Разрежьте изображение на две части и попробуйте применить код плотной реконструкции глубины

ГЛАВА 6. Кластеризация изображений

В этой главе мы познакомимся с несколькими методами кластеризации и покажем, как их применить для нахождения групп похожих изображений. Кластеризацию можно использовать для распознавания, для разделения наборов изображений или для организации и навигации. Мы также рассмотрим, как с помощью кластеризации наглядно представить сходство изображений.

6.1. Кластеризация методом K средних

Метод *K средних* – это очень простой алгоритм кластеризации, который пытается разбить данные на K кластеров. Принцип его работы основан на уточнении начальной оценки центроидов классов:

1. Инициализировать центроиды $\boldsymbol{\mu}_i$, $i = 1, ..., k$ случайным образом или исходя из некоторой гипотезы.
2. Назначить каждой точке класс c_i ближайшего к ней центроида.
3. Пересчитать центроиды, присвоив каждому среднее значение всех точек, отнесенных к тому же классу, что он сам.
4. Повторять шаги 2 и 3, пока алгоритм не сойдется.

В методе K средних ставится цель минимизировать *полную внутриклассовую дисперсию*:

$$V = \sum_{i=0}^{k} \sum_{\mathrm{x}_j \in C_i} (\mathrm{x}_j - \mu_i)^2,$$

где $\mathbf{x}_j$ – векторы данных. Описанный выше эвристический алгоритм уточнения работает в большинстве случаев, но не гарантируется, что

он найдет наилучшее решение. Чтобы уменьшить зависимость от неудачного выбора центроидов, алгоритм часто прогоняют несколько раз с различными начальными центроидами, а затем выбирают решение с наименьшей дисперсией *V*.

Основной недостаток этого алгоритма заключается в том, что количество кластеров необходимо задавать заранее, и, если оно выбрано неправильно, то результаты кластеризации окажутся неудовлетворительными. Ну а достоинства – простота реализации и распараллеливания, а также пригодность для широкого круга задач без какой-либо настройки.

Пакет кластеризации в SciPy

Хотя алгоритм просто реализовать самостоятельно, делать это ни к чему. В пакете векторного квантования `scipy.cluster.vq` уже имеется реализация метода *K* средних. Покажем, как ей пользоваться на примере двумерного набора данных.

```
from scipy.cluster.vq import *

class1 = 1.5 * randn(100,2)
class2 = randn(100,2) + array([5,5])
features = vstack((class1,class2))
```

Этот код генерирует два класса с нормальным распределением по обоим направлениям. Попробуем кластеризовать точки, выполнив алгоритм *K* средних с $K = 2$:

```
centroids,variance = kmeans(features,2)
```

Функция возвращает дисперсию, но она нам ни к чему, потому что реализация из SciPy вычисляет кластеры несколько раз (по умолчанию 20) и сама выбирает решение с наименьшей дисперсией. Теперь можно посмотреть, как точки распределились между кластерами, воспользовавшись функцией векторного квантования:

```
code,distance = vq(features,centroids)
```

Проверив значение *code*, мы можем узнать, встретились ли неправильные назначения. Для наглядности можно нанести на график сами точки и вычисленные центроиды:

```
figure()
ndx = where(code==0)[0]
plot(features[ndx,0],features[ndx,1],"*")
```

```
ndx = where(code==1)[0]
plot(features[ndx,0],features[ndx,1],"r.")
plot(centroids[:,0],centroids[:,1],"go")
axis('off")
show()
```

Здесь функция `where()` возвращает индексы для каждого класса. В результате получается график, изображенный на рис. 6.1.

Рис. 6.1. Пример кластеризации точек на плоскости методом *K* средних. Центроиды классов обозначены увеличенными кружками, а предсказанные классы – точками и звездочками

Кластеризация изображений

Попробуем применить метод *K* средних к изображениям буквы в разных шрифтах на рис. 1.8. В файле *selectedfontimages.zip* находятся 66 изображений из этого набора данных (они отобраны для более наглядной иллюстрации кластеров). В качестве дескрипторного вектора изображения мы возьмем коэффициенты проекций на первые 40 главных компонент, вычисленных ранее. Загрузим модель из pickle-файла, спроецируем изображения на главные компоненты и выполним кластеризацию:

```
import imtools
import pickle
from scipy.cluster.vq import *

# получить список изображений
imlist = imtools.get_imlist('selected_fontimages/")
```

```
imnbr = len(imlist)

# загрузить файл модели
with open('a_pca_modes.pkl","rb") as f:
  immean = pickle.load(f)
  V = pickle.load(f)

# создать матрицу для хранения линеаризованных изображений
immatrix = array([array(Image.open(im)).flatten()
                          for im in imlist],"f")

# спроецировать на первые 40 главных компонент
immean = immean.flatten()
projected = array([dot(V[:40],immatrix[i]-immean) for i in range(imnbr)])

# метод K средних
projected = whiten(projected)
centroids,distortion = kmeans(projected,4)

code,distance = vq(projected,centroids)
```

Как и раньше, *code* содержит сведения о том, какой кластер назначен каждому изображению. В данном случае мы выбрали $K = 4$. Мы также решили «обелить» данные с помощью функции `SciPy whiten()`, т. е. нормировать его, так чтобы дисперсии всех признаков были равны 1. Попробуйте поиграть с параметрами, например количеством главных компонент и значением K, и посмотреть, как будут меняться результаты кластеризации. Для визуализации кластеров выполним такой код:

```
# нанести кластеры на график
for k in range(4):
  ind = where(code==k)[0]
  figure()
  gray()
  for i in range(minimum(len(ind),40)):
     subplot(4,10,i+1)
     imshow(immatrix[ind[i]].reshape((25,25)))
     axis('off")
show()
```

Здесь каждый кластер показан в отдельном окне рисунка, причем из каждого кластера взято не более 40 изображений. Для определения сетки рисунков мы воспользовались функцией `subplot()` из пакета `PyLab`. На рис. 6.2 показаны получившиеся кластеры.

Дополнительные сведения о реализации метода K средних в `SciPy` и о пакете `scipy.cluster.vq` можно найти в справочном руководстве по адресу *http://docs.scipy.org/doc/scipy/reference/cluster.vq.html*.

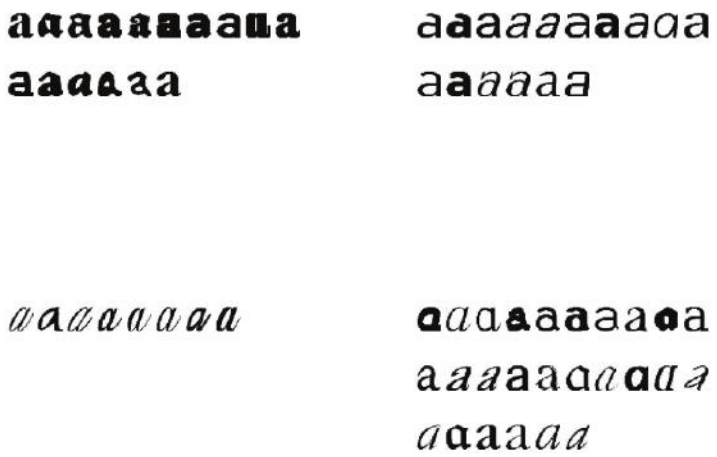


Рис. 6.2. Пример кластеризации изображений буквы разными шрифтами методом *K* средних при K = 4 с использованием 40 главных компонент

Визуализация проекций изображений на главные компоненты

Чтобы понять, как работает кластеризация на основе всего нескольких главных компонент, представим изображения с помощью координат их проекций на две главные компоненты. Чтобы спроецировать изображения на две компоненты вместо 40, немного изменим код:

```
projected = array([dot(V[[0,2]],immatrix[i]-immean) for i in range(imnbr)])
```

(здесь $V[[0, 2]]$ дает проекции на первую и третью компоненты).

Можно было бы вместо этого спроецировать изображения на все компоненты, а затем выбрать интересующие нас столбцы. Для визуализации воспользуемся модулем `ImageDraw` из библиотеки PIL. Для имеющихся у нас проекций и списка изображений следующий короткий скрипт сгенерирует график, показанный на рис. 6.3.

```
from PIL import Image, ImageDraw

# высота и ширина
h,w = 1200,1200

# создать новое изображение на белом фоне
```

```
img = Image.new('RGB',(w,h),(255,255,255))
draw = ImageDraw.Draw(img)

# нарисовать оси
draw.line((0,h/2,w,h/2),fill=(255,0,0))
draw.line((w/2,0,w/2,h),fill=(255,0,0))

# масштабировать координаты под размер рисунка
scale = abs(projected).max(0)
scaled = floor(array([ (p / scale) * (w/2-20,h/2-20) +
                       (w/2,h/2) for p in projected]))

# вставить миниатюры каждого изображения
for i in range(imnbr):
  nodeim = Image.open(imlist[i])
  nodeim.thumbnail((25,25))
  ns = nodeim.size
  img.paste(nodeim,(scaled[i][0]-ns[0]//2,scaled[i][1]-
     ns[1]//2,scaled[i][0]+ns[0]//2+1,scaled[i][1]+ns[1]//2+1))

img.save('pca_font.jpg')
```

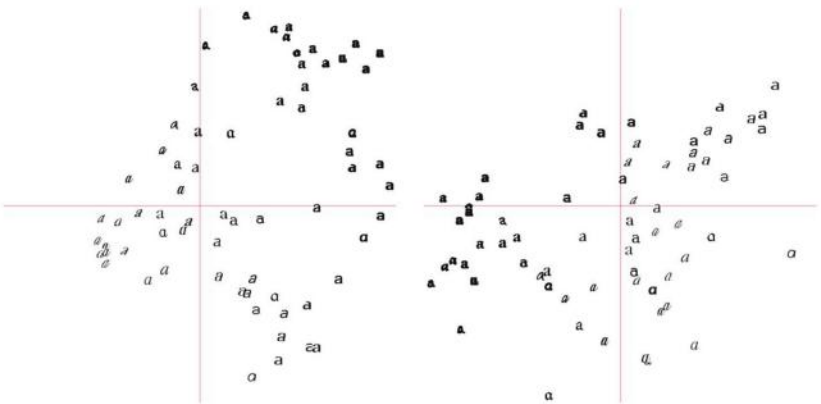


Рис. 6.3. Проекции изображений буквы на две главные компоненты: первую и вторую (слева); вторую и третью (справа)

Здесь мы воспользовались оператором целочисленного деления `//`, который возвращает целое число, описывающее позицию пикселя, отбрасывая все знаки после запятой.

Подобные графики иллюстрируют расположение изображений по 40 измерениям и могут оказаться очень полезны для выбора хорошего дескриптора. Даже при проецировании на плоскость отчетливо видна близость изображений, соответствующих похожим шрифтам.

Кластеризация пикселей

Прежде чем закончить эту тему, рассмотрим пример кластеризации отдельных пикселей, а не целых изображений. Группировка областей и пикселей изображений в «осмысленные» компоненты называется *сегментацией изображения* и будет подробно рассмотрена в главе 9. Наивное применение метода *K* средних к значениям пикселей не дает осмысленного результата – разве что для очень простых изображений. Для получения полезной сегментации нужно использовать более изощренные модели классов, учитывающие не только средний цвет или пространственное расположение пикселей. Но пока применим метод *K* средних к значениям RGB, а решением задач сегментации займемся позже (см. раздел 9.2).

Показанная ниже программа читает изображение, понижает его разрешение, усредняя пиксели (по квадратной сетке размером *steps* × *steps*) и выполняет кластеризацию областей методом *K* средних.

```
from scipy.cluster.vq import *
from scipy.misc import imresize

steps = 50 # разбить изображения на области размером steps*steps
im = array(Image.open('empire.jpg"))

dx = im.shape[0] / steps
dy = im.shape[1] / steps

# для каждой области вычислить цветовые признаки
features = []
for x in range(steps):
  for y in range(steps):
     R = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,0])
     G = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,1])
     B = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,2])
     features.append([R,G,B])
     features = array(features,"f") # собрать в массив

# кластеризовать
centroids,variance = kmeans(features,3)
code,distance = vq(features,centroids)

# построить изображение, содержащее метки кластеров
codeim = code.reshape(steps,steps)
codeim = imresize(codeim,im.shape[:2],interp="nearest")

figure()
imshow(codeim)
show()
```

На вход методу *K* средних подается массив, содержащий *steps*steps* строк, в каждой из которых находятся средние значения R, G и B. Для визуализации результата мы используем функцию `SciPy imresize()`, которая показывает изображение размера *steps*steps* в координатах исходного изображения. Параметр *interp* задает тип интерполяции; в данном случае интерполяция производится по ближайшему соседу, чтобы не создавать новых значений пикселей на границах между классами.

На рис. 6.4 показаны результаты для областей размера 50×50 и 100×100 и двух сравнительно простых изображений. Отметим, что порядок меток, назначенных методом *K* средних (в данном случае цветов в результирующих изображениях), произволен. Как видите, результат сильно зашумлен, несмотря на понижение разрешения. Не видно никакой пространственной согласованности и трудно разделить области, например мальчика и траву в нижнем примере. Пространственной согласованностью и улучшением разделения мы займемся позже, когда будем говорить о других алгоритмах сегментации изображений. А сейчас перейдем к следующему алгоритму кластеризации.

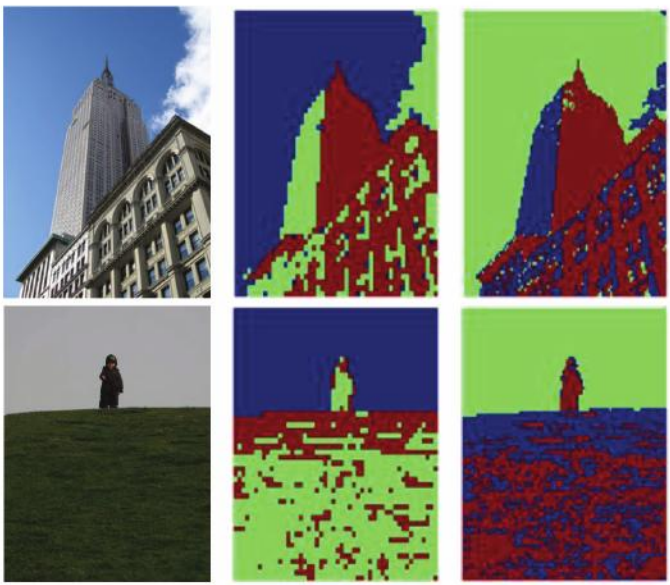


Рис. 6.4. Кластеризация пикселей на основе цвета методом *K* средних: исходное изображение (слева); результат кластеризации при *K* = 3 и разрешении 50×50 (в центре); результат кластеризации при *K* = 3 и разрешении 100×100 (справа)

6.2. Иерархическая кластеризация

Иерархическая (или *агломеративная*) кластеризация – еще один простой, но мощный алгоритм кластеризации. Идея в том, чтобы построить дерево сходства на основе попарных расстояний между точками. Сначала алгоритм объединяет в одну группу два ближайших объекта (исходя из расстояния между векторами признаков), а затем создает «средний» узел дерева, делая эти объекты его дочерними узлами. Среди оставшихся объектов и средних узлов ищется следующая ближайшая пара. И так далее. В каждом узле сохраняется расстояние между двумя его дочерними узлами. Теперь для получения кластеров остается обойти дерево, прекращая обход, когда расстояние станет меньше некоторого порога, определяющего размер кластера.

У иерархической кластеризации есть несколько достоинств. В частности, древовидную структуру можно использовать для визуализации связей между кластерами. Если векторы признаков выбраны удачно, то дерево покажет хорошее разделение. Другое преимущество заключается в том, что построенное дерево можно использовать с другим порогом размера кластеров, ничего не пересчитывая. Недостаток же состоит в том, что для выбора кластеров нужно указать некоторый порог.

Посмотрим, как это выглядит в коде[1]. Создайте файл *hcluster.py* и поместите в него следующий код (за образец взят код иерархической кластеризации из [31]):

```
from itertools import combinations

class ClusterNode(object):
  def __init__(self,vec,left,right,distance=0.0,count=1):
      self.left = left
      self.right = right
      self.vec = vec
      self.distance = distance
      self.count = count # используется только для взвешенного среднего

  def extract_clusters(self,dist):
      """ Извлечь из дерева hcluster список поддеревьев,
        определяющих кластеры, для которых distance<dist. """
      if self.distance < dist:
          return [self]
```

[1] В библиотеке SciPy также имеется вариант иерархической кластеризации, можете ознакомиться с ним, если хотите. Мы не воспользовались им, потому что нам нужен класс, который умеет рисовать дендрограммы и визуализировать кластеры с помощью миниатюр изображений.

```
        return self.left.extract_clusters(dist) +
               self.right.extract_clusters(dist)

    def get_cluster_elements(self):
        """ Вернуть идентификаторы элементов в поддереве кластеров. """
        return self.left.get_cluster_elements() +
               self.right.get_cluster_elements()

    def get_height(self):
        """ Вернуть высоту узла, т.е. сумму высот обеих ветвей. """
        return self.left.get_height() + self.right.get_height()

    def get_depth(self):
        """ Вернуть глубину узла, т.е. максимум из глубин дочерних
          узлов плюс величину distance самого узла. """
        return max(self.left.get_depth(), self.right.get_depth()) +
               self.distance

class ClusterLeafNode(object):
    def __init__(self,vec,id):
        self.vec = vec
        self.id = id

    def extract_clusters(self,dist):
        return [self]

    def get_cluster_elements(self):
        return [self.id]

    def get_height(self):
        return 1

    def get_depth(self):
        return 0

def L2dist(v1,v2):
    return sqrt(sum((v1-v2)**2))

def L1dist(v1,v2):
    return sum(abs(v1-v2))

def hcluster(features,distfcn=L2dist):
    """ Кластеризовать строки признаков методом
        иерархической кластеризации. """

    # кэшировать вычисление расстояний
    distances = {}

    # в начале каждая строка считается кластером
    node = [ClusterLeafNode(array(f),id=i) for i,f in enumerate(features)]
```

```
    while len(node)>1:
        closest = float('Inf")

        # перебрать в цикле все пары и найти наименьшее расстояние
        for ni,nj in combinations(node,2):
            if (ni,nj) not in distances:
                distances[ni,nj] = distfcn(ni.vec,nj.vec)

            d = distances[ni,nj]
            if d<closest:
                closest = d
                lowestpair = (ni,nj)
        ni,nj = lowestpair

        # усреднить оба кластера
        new_vec = (ni.vec + nj.vec) / 2.0

        # создать новый узел
        new_node = ClusterNode(new_vec,left=ni,right=nj,distance=closest)
        node.remove(ni)
        node.remove(nj)
        node.append(new_node)

    return node[0]
```

Мы создали два класса для представления узлов дерева кластеров: `ClusterNode` и `ClusterLeafNode`. Функция `hcluster()` строит дерево. Сначала создается список листовых узлов, затем в цикле объединяются пары, ближайшие с точки зрения выбранной метрики. Возвращенный функцией узел является корнем дерева. Если применить `hcluster()` к матрице векторов признаков, то она создаст и вернет дерево кластеров.

Выбор метрики зависит от признаков. В данном случае мы взяли евклидово расстояние (L_2) (предоставляется также функция для расстояния L_1), но разрешается написать любую функцию и передать ее в качестве параметра `hcluster()`. В качестве нового вектора признаков, который представляет поддерево и позволяет рассматривать его как объект, мы взяли среднее векторов признаков во всех узлах поддерева. Существуют и другие правила выбора узлов, объединяемых на очередной итерации, например *одиночная связь* (использовать наименьшее расстояние между объектами в двух поддеревьях) и *полная связь* (использовать наибольшее расстояние между объектами в двух поддеревьях). От выбора связи зависит вид порожденных кластеров.

Для извлечения кластеров нужно обойти дерево, начиная с корня и до узла с расстоянием, меньшим заданного порога. Проще всего это

сделать рекурсивно. Задачу решает метод `extract_clusters()` класса `ClusterNode`, который возвращает список, содержащий сам узел, если расстояние меньше порогового, или вызывает себя для дочерних узлов (для листовых узлов всегда возвращаются они сами). В результате вызова этого метода мы получим список поддеревьев, содержащих кластеры. Чтобы для каждого поддерева кластеров получить листовые узлы, содержащие идентификаторы объектов, нужно обойти поддерево и вернуть список листьев с помощью метода `get_cluster_elements()`.

Проиллюстрируем описанную стратегию на простом примере. Сначала создадим точки на плоскости (как для метода *K* средних выше):

```
class1 = 1.5 * randn(100,2)
class2 = randn(100,2) + array([5,5])
features = vstack((class1,class2))
```

Кластеризуем точки и выберем кластеры из списка, задав порог (в данном случае 5), а затем выведем кластеры на консоль.

```
import hcluster

tree = hcluster.hcluster(features)

clusters = tree.extract_clusters(5)

print 'число кластеров', len(clusters)
for c in clusters:
  print c.get_cluster_elements()
```

В результате будет выведена примерно такая распечатка:

```
число кластеров 2
[184, 187, 196, 137, 174, 102, 147, 145, 185, 109, 166, 152, 173, 180, 128, 163, 141,
178, 151, 158, 108, 182, 112, 199, 100, 119, 132, 195, 105, 159, 140, 171, 191, 164,
130, 149, 150, 157, 176, 135, 123, 131, 118, 170, 143, 125, 127, 139, 179, 126, 160,
162, 114, 122, 103, 146, 115, 120, 142, 111, 154, 116, 129, 136, 144, 167, 106, 107,
198, 186, 153, 156, 134, 101, 110, 133, 189, 168, 183, 148, 165, 172, 188, 138, 192,
104, 124, 113, 194, 190, 161, 175, 121, 197, 177, 193, 169, 117, 155]
[56, 4, 47, 18, 51, 95, 29, 91, 23, 80, 83, 3, 54, 68, 69, 5, 21, 1, 44, 57, 17, 90,
30, 22, 63, 41, 7, 14, 59, 96, 20, 26, 71, 88, 86, 40, 27, 38, 50, 55, 67, 8, 28, 79,
64, 66, 94, 33, 53, 70, 31, 81, 9, 75, 15, 32, 89, 6, 11, 48, 58, 2, 39, 61, 45,
65, 82, 93, 97, 52, 62, 16, 43, 84, 24, 19, 74, 36, 37, 60, 87, 92, 181, 99, 10, 49,
12, 76, 98, 46, 72, 34, 35, 13, 73, 78, 25, 42, 77, 85]
```

В идеале должно получиться два кластера, но в зависимости от данных их может оказаться три или даже больше. В нашем простом

примере один кластер должен содержать значения меньше 100, а другой – 100 и выше.

Кластеризация изображений

Рассмотрим пример кластеризации изображений по цвету. Файл *sunsets.zip* содержит 100 фотографий с сайта Flickr, найденных по метке «sunset» (закат) или «sunsets» (закаты). В этом примере мы в качестве вектора признаков будем использовать гистограмму цветов. Это грубовато и слишком упрощенно, но достаточно для иллюстрации работы иерархической кластеризации. Выполните следующую программу, находясь в папке, содержащей изображения заката:

```
import os
import hcluster

# создать список изображений
path = 'flickr-sunsets/'
imlist = [os.path.join(path,f) for f in os.listdir(path)
                                  if f.endswith('.jpg')]

# выделить признаки (8 интервалов на один цветовой канал)
features = zeros([len(imlist), 512])
for i,f in enumerate(imlist):
  im = array(Image.open(f))

  # многомерная гистограмма
  h,edges = histogramdd(im.reshape(-1,3),8,normed=True,
                        range=[(0,255),(0,255),(0,255)])
  features[i] = h.flatten()

tree = hcluster.hcluster(features)
```

Здесь в качестве вектора признаков мы берем цветовые каналы R, G, B и подаем их на вход функции `histogramdd()` из пакета `NumPy`. Она вычисляет многомерные (в данном случае трехмерные) гистограммы. Мы выбрали по 8 интервалов в каждом цветовом измерении (8×8×8), после линеаризации это дает вектор признаков с 512 элементами. Параметр «normed=True» изначает, что гистограммы нужно нормировать в случае, когда размеры изображений разнятся. Далее указаны диапазоны значений для каждого канала: 0 ... 255. Благодаря использованию функции `reshape()` с первым параметром –1 будет автоматически определен правильный размер и, следовательно, создан входной массив для вычисления гистограммы, содержащий в строках значения цветов в формате RGB.

Для визуализации дерева кластеров мы можем нарисовать *дендрограмму*. Так называется диаграмма, на которой показана структура дерева. Часто она дает полезную информацию о качестве дескрипторного вектора и о том, что в данном конкретном случае считается похожим. Добавьте следующий код в файл *hcluster.py*:

```
from PIL import Image,ImageDraw

def draw_dendrogram(node,imlist,filename='clusters.jpg'):
  """ Нарисовать дендрограмму кластеров и сохранить ее в файле. """

  # высота и ширина
  rows = node.get_height()*20
  cols = 1200

  # масштабный коэффициент, чтобы изображение уместилось по ширине
  s = float(cols-150)/node.get_depth()

  # создать изображение и нарисовать объект
  im = Image.new('RGB',(cols,rows),(255,255,255))
  draw = ImageDraw.Draw(im)

  # начальная вертикаль дерева
  draw.line((0,rows/2,20,rows/2),fill=(0,0,0))

  # рекурсивно нарисовать узлы
  node.draw(draw,20,(rows/2),s,imlist,im)
  im.save(filename)
  im.show()
```

Здесь для рисования всех узлов дендрограммы применяется метод `draw()`. Добавьте его в класс `ClusterNode`.

```
def draw(self,draw,x,y,s,imlist,im):
  """ Рекурсивно нарисовать узлы, помещая в листья
     миниатюры изображений. """

  h1 = int(self.left.get_height()*20 / 2)
  h2 = int(self.right.get_height()*20 /2)
  top = y-(h1+h2)
  bottom = y+(h1+h2)

  # вертикальный отрезок до дочерних узлов
  draw.line((x,top+h1,x,bottom-h2),fill=(0,0,0))

  # горизонтальные отрезки
  l1 = self.distance*s
  draw.line((x,top+h1,x+l1,top+h1),fill=(0,0,0))
```

```
    draw.line((x,bottom-h2,x+ll,bottom-h2),fill=(0,0,0))

    # рекурсивно нарисовать левый и правый дочерний узел
    self.left.draw(draw,x+ll,top+hl,s,imlist,im)
    self.right.draw(draw,x+ll,bottom-h2,s,imlist,im)
```

У листовых узлов имеется специальный метод для рисования миниатюр изображений. Добавьте его в класс `ClusterLeafNode`.

```
def draw(self,draw,x,y,s,imlist,im):
 nodeim = Image.open(imlist[self.id])
 nodeim.thumbnail([20,20])
 ns = nodeim.size
 im.paste(nodeim,[int(x),int(y-ns[1]//2),int(x+ns[0]),
                  int(y+ns[1]-ns[1]//2)])
```

Высота дендрограммы и ее частей определяется значениями расстояния. Их необходимо масштабировать, чтобы они помещались в изображение выбранного размера. Узлы рисуются рекурсивно, на следующий уровень передаются координаты. Листовые узлы представлены миниатюрными изображениями размером 20×20 пикселей. Для получения высоты и ширины дерева служат вспомогательные методы `get_height()` и `get_depth()`.

Для рисования дендрограммы нужно вызвать метод следующим образом:

```
hcluster.draw_dendrogram(tree,imlist,filename='sunset.pdf')
```

На рис. 6.5 показана дендрограмма изображений заката. Как видим, изображения похожего цвета расположены в дендрограмме близко друг к другу. На рис. 6.6 показаны три примера кластеров. Для их получения был выполнен такой код:

```
# визуализировать кластеры с некоторым (произвольно выбранным) порогом
clusters = tree.extract_clusters(0.23*tree.distance)

# нанести на график изображения кластеров, содержащих больше 3 элементов
for c in clusters:
  elements = c.get_cluster_elements()
  nbr_elements = len(elements)
  if nbr_elements>3:
     figure()
     for p in range(minimum(nbr_elements,20)):
        subplot(4,5,p+1)
        im = array(Image.open(imlist[elements[p]]))
        imshow(im)
        axis('off")
show()
```

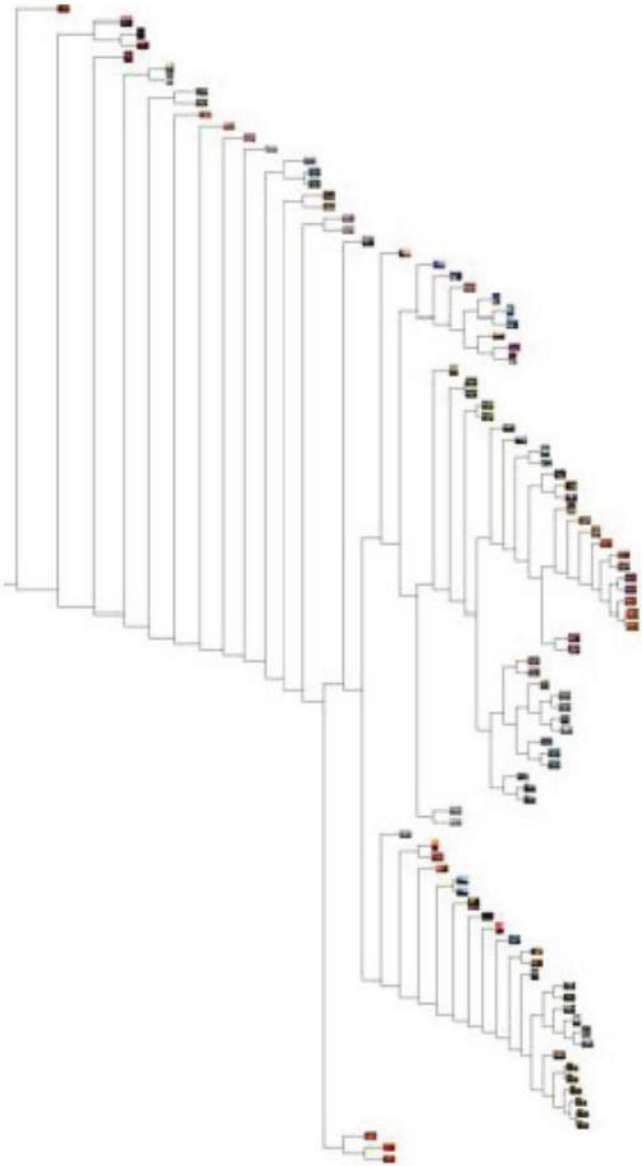


Рис. 6.5. Пример иерархической кластеризации 100 изображений заката с применением 512-интервальной гистограммы RGB-цветов в качестве вектора признаков. Изображения, оказавшиеся рядом в дереве, имеют схожее распределение цветов

Рис. 6.6. Примеры кластеров, выделенных из 100 изображений заказа с помощью иерархической кластеризации с порогом 23 % от максимального расстояния между узлами дерева

Напоследок создадим дендрограмму изображений буквы в разных шрифтах.

```
tree = hcluster.hcluster(projected)
hcluster.draw_dendrogram(tree,imlist,filename='fonts.jpg')
```

где *projected* и *imlist* – переменные из примера метода *K* средних в разделе 6.1. Результат показан на рис. 6.7.

6.3. Спектральная кластеризация

Методы *спектральной кластеризации* дают интересный пример алгоритмов, в которых применен подход, отличающийся от метода *K* средних и иерархической кластеризации.

Матрицей сходства (или *матрицей близости*, а иногда *матрицей расстояний*) для *n* объектов (например, изображений) называется матрица размерности *n*×*n*, содержащая оценки попарного сходства. Словом «спектральный» алгоритм обязан использованию спектра матрицы, построенной по матрице сходства. Для понижения размерности задачи и последующей кластеризации используются собственные векторы этой матрицы.

Одно из преимуществ спектральных методов кластеризации заключается в том, что на вход подается только эта матрица, а строить

ее можно на основе любой меры близости. Методы типа K средних или иерархической кластеризации вычисляют среднее векторов признаков, а это налагает ограничение: признаки (или дескрипторы) должны быть векторами (а как иначе вычислить среднее?). Для спектральных методов вообще не нужны никакие векторы признаков, достаточно только понятия «расстояния» или «сходства».

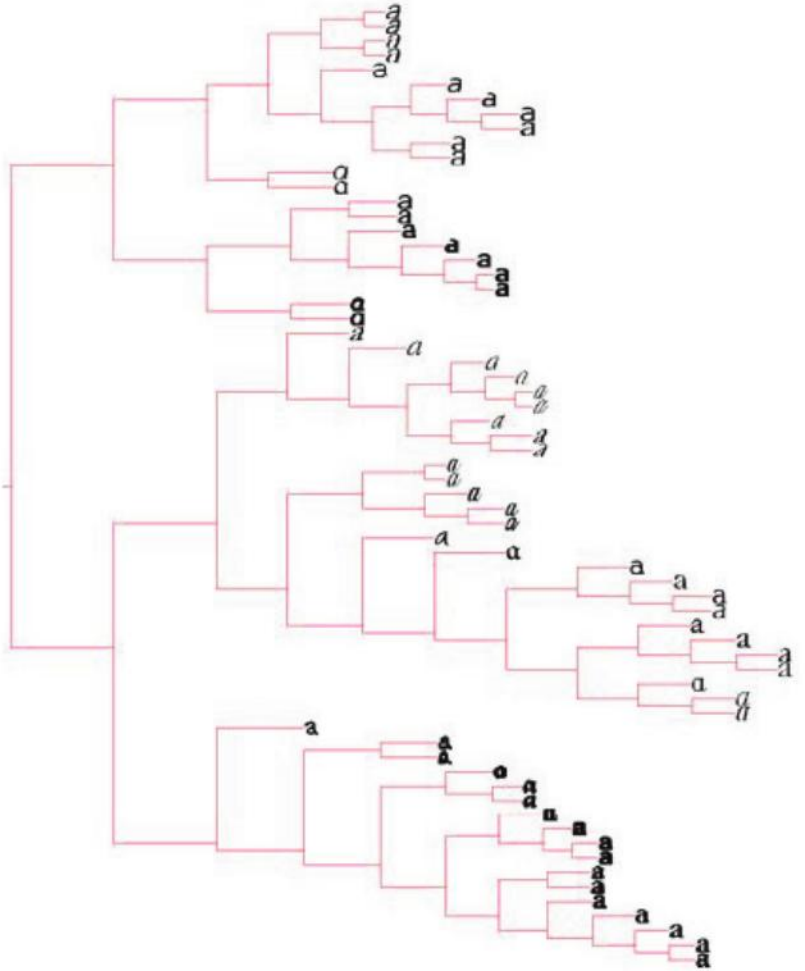


Рис. 6.7. Пример иерархической кластеризации 66 выбранных изображений буквы в разных шрифтах с применением 40 главных компонент в качестве вектора признаков

Работает это следующим образом. Если дана матрица сходства S размерности $n \times n$, содержащая элементы s_{ij}, то можно построить *матрицу Лапласа*[2]

$$L = I - D^{-1/2} S D^{-1/2},$$

где I – единичная матрица, а D – диагональная матрица, в которой на диагонали находятся суммы элементов в строках S, $D = \mathrm{diag}(d_i)$, $d_i = \sum_j s_{ij}$. Тогда матрица $D^{-1/2}$, участвующая в выражении для матрицы Лапласа, имеет вид

$$D^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{d_1}} & & & \\ & \frac{1}{\sqrt{d_2}} & & \\ & & \ddots & \\ & & & \frac{1}{\sqrt{d_n}} \end{bmatrix}.$$

Для простоты изложения будем использовать небольшие значения s_{ij} для похожих элементов и наложим ограничение $s_{ij} \geq 0$ (в этом случае больше подходит термин «матрица расстояний»).

Для нахождения кластеров нужно вычислить собственные векторы L и взять k собственных векторов, соответствующих наибольшим собственным значениям. Из них строится множество векторов признаков (напомним, что в начале у нас могло не быть вообще ничего!). Построим матрицу, в которой столбцами являются k собственных векторов. Ее строки будем рассматривать как новые векторы признаков (длины k). Эти векторы признаков можно подвергнуть кластеризации, например методом K средних, в результате чего будет получены окончательные кластеры. По существу, этот алгоритм преобразует исходные данные в новые векторы признаков, которые проще кластеризовать (в некоторых случаях можно применить алгоритмы, непригодные для исходных данных).

Но довольно теории; посмотрим, как выглядит код, работающий в реальной ситуации. Снова возьмем изображения буквы в разных шрифтах.

[2] Иногда матрицей Лапласа называют матрицу $L = D^{-1/2} S D^{-1/2}$, но это не имеет значения, т. к. изменяются только собственные значения, но не собственные векторы.

```
from scipy.cluster.vq import *

n = len(projected)

# вычислить матрицу расстояний
S = array([[ sqrt(sum((projected[i]-projected[j])**2))
          for i in range(n) ] for j in range(n)], 'f')

# построить матрицу Лапласа
rowsum = sum(S,axis=0)
D = diag(1 / sqrt(rowsum))
I = identity(n)
L = I - dot(D,dot(S,D))

# вычислить собственные векторы L
U,sigma,V = linalg.svd(L)

k = 5
# создать вектор признаков из первых k собственных векторов,
# сделав их столбцами матрицы
features = array(V[:k]).T

# применить метод K средних
features = whiten(features)
centroids,distortion = kmeans(features,k)
code,distance = vq(features,centroids)

# нанести кластеры на график
for c in range(k):
  ind = where(code==c)[0]
  figure()
  for i in range(minimum(len(ind),39)):
     im = Image.open(path+imlist[ind[i]])
     subplot(4,10,i+1)
     imshow(array(im))
     axis('equal')
     axis('off")
show()
```

В данном случае для построения S мы используем попарные евклидовы расстояния и применяем стандартную кластеризацию методом K средних к k собственным векторам (где $k = 5$). Напомним, что матрица V содержит собственные векторы, упорядоченные по убыванию собственных значений. В самом конце наносим кластеры на график. На рис. 6.8 показаны кластеры, получившиеся в этом примере (напомним, что метод K средних может при каждом запуске давать разные результаты).

Рис. 6.8. Спектральная кластеризация изображений буквы в разных шрифтах с применением собственных векторов матрицы Лапласа

Мы можем применить этот метод к примеру, где нет ни векторов признаков, ни строгого определения сходства. Изображения с сайта Panoramio с геометками (см. стр. 71) связывались, исходя из количества найденных соответственных локальных дескрипторов. Матрицу на стр. 75 можно рассматривать как матрицу сходства, в которой оценки равны количеству соответственных признаков (без какой-либо нормировки). Если список *imlist* содержит имена файлов изображений, а матрица сходства сохранена в файле с помощью функции `NumPy savetxt()`, то нам нужно лишь следующим образом модифицировать первые строки написанного ранее кода:

```
n = len(imlist)

# загрузить и переформатировать матрицу сходства
S = loadtxt('panoramio_matches.txt')
S = 1 / (S + 1e-6)
```

Здесь мы инвертируем элементы, чтобы у похожих изображений были небольшие значения оценок (тогда не придется модифицировать приведенный выше код). Прибавляем небольшое число, чтобы избежать деления на нуль. Больше ничего в коде не меняется.

В данном случае выбор k не вполне очевиден. Большинство людей сочли бы, что имеется только два класса (две стороны Белого дома) и какие-то посторонние изображения. При $k = 2$ получится картина, изображенная на рис. 6.9, где один большой кластер включает фотографии, снятые с одной стороны, а второй – фотографии, снятые с другой стороны, плюс все прочие. Если взять большее значение, скажем $k = 10$, то получатся несколько кластеров, содержащих по одному изображению (хочется надеяться, что постороннему), и несколько на-

стоящих кластеров. Результат прогона показан на рис. 6.10. В данном случае мы имеем только два реальных кластера, каждый из которых содержит изображения одной стороны Белого дома.

У представленного здесь алгоритма есть много вариантов, отличающихся способом построения матрицы L и действиями над собственными векторами. Дополнительные сведения о спектральной кластеризации и деталях некоторых распространенных алгоритмов можно почерпнуть, например, в обзорной статье [37].

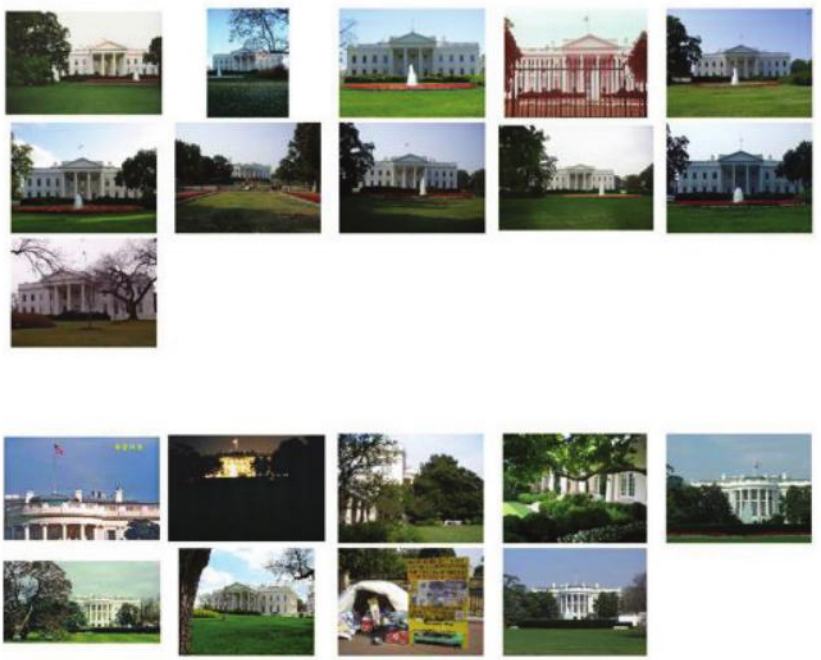


Рис. 6.9. Спектральная кластеризация изображений Белого дома с геометками при $k = 2$, когда в роли оценки сходства выступает количество соответственных локальных дескрипторов

Упражнения

1. *Иерархическим методом K средних* называется метод кластеризации, который рекурсивно применяет метод K средних к кластерам для создания дерева постепенно уточняемых кластеров. Каждый узел дерева в этом случае имеет K дочерних узлов. Реализуйте этот алгоритм и примените его к изображениям буквы в разных шрифтах.

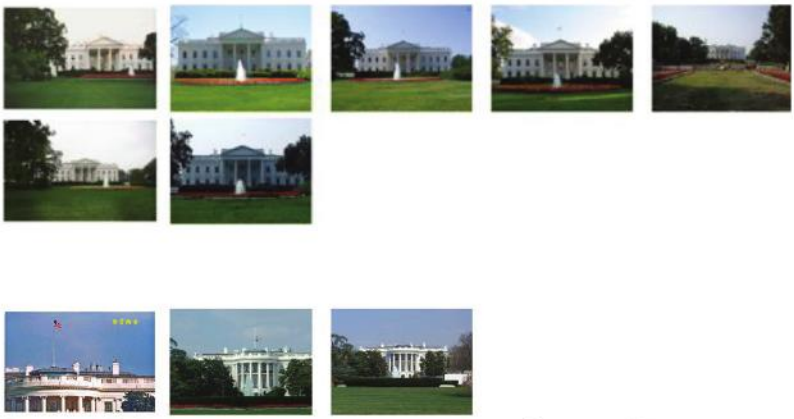


Рис. 6.10. Спектральная кластеризация изображений Белого дома с геометками при $k = 10$, когда в роли оценки сходства выступает количество соответственных локальных дескрипторов. Показаны только кластеры, содержащие больше одного изображения

2. Применив иерархический метод K средних из предыдущего упражнения, визуализируйте дерево (по типу дендрограммы для иерархической кластеризации), которое показывает среднее изображение для кластера в каждом узле.
   Подсказка: можно взять вектор признаков, составленный из средних коэффициентов PCA, и, используя базис PCA, синтезировать изображение для каждого вектора признаков.
3. Если модифицировать класс, использованный для иерархической кластеризации, включив в него количество изображений ниже данного узла, то мы получим простой и быстрый способ нахождения компактных групп похожих изображений заданного размера. Реализуйте это изменение и протестируйте его на каких-нибудь реальных данных. Каково качество алгоритма?
4. Поэкспериментируйте с применением одиночной и полной связи к построению дерева кластеров. Насколько сильно отличаются получающиеся кластеры?
5. В некоторых алгоритмах спектральной кластеризации вместо L применяется матрица $D^{-1}S$. Попробуйте заменить ей матрицу Лапласа и применить новый алгоритм к различным наборам данных.
6. Скачайте с сайта Flickr наборы изображений с разными метками. Постройте гистограмму RGB, как мы делали для изображений заката. Кластеризуйте изображения одним из описанных в этой главе методов. Удается ли с помощью кластеров разделить классы?

ГЛАВА 7. Поиск изображений

В этой главе мы покажем, как методы анализа текста позволяют искать изображения по визуальному содержанию. Излагаются основополагающие идеи об использовании визуальных слов, объясняется, как настроить инфраструктуру и протестировать ее на демонстрационном наборе данных.

7.1. Поиск изображений по содержанию

Поиск изображений по содержанию (Content-based image retrieval, CBIR) – это раздел компьютерного зрения, в котором решается задача нахождения визуально похожих изображений в большой базе данных. Это могут быть изображения примерно одного цвета, с похожей текстурой или содержащие похожие объекты или сцены – в общем, может использоваться любая информация, являющаяся частью изображения.

Для таких запросов высокого уровня, как поиск похожих объектов, практически невозможно выполнить полное сравнение (например, сопоставление признаков) изображения-запроса со всеми изображениями в базе. Если база данных велика, то на это ушло бы недопустимо много времени. Но в последние годы были найдены способы применить методы, заимствованные из анализа текстов, к задачам поиска изображений, в результате чего стало возможно осуществлять поиск среди миллионов похожих изображений.

Векторная модель – инструмент анализа текста

Векторная модель служит для представления и поиска текстовых документов. Но, как мы увидим, она применима к объектам практиче-

ски любого вида, в т. ч. к изображениям. Название связано с тем, что текстовые документы представляются векторами, описывающими гистограммы распределения частот слов в тексте[1]. Иначе говоря, вектор содержит количество вхождений слова в позиции, соответствующей этому слову, и нули в остальных позициях. Эта модель называется также *моделью набора слов* (bag-of-words), поскольку порядок и местоположение слов игнорируются.

Для индексирования документов подсчитываются слова и строится вектор гистограммы **v**, причем часто употребительные слова типа «the», «and», «is» и т. д. игнорируются. Такие слова называются *стоп-словами*. Для учета различий в длине документов векторы можно нормировать на единичную длину, разделив на сумму всех значений в гистограмме. Отдельным элементам вектора гистограммы обычно назначается вес в соответствии с важностью слова. Как правило, важность увеличивается для слов, которые часто встречаются в документе, но уменьшается, если слово часто встречается во всех документах из имеющегося набора (или «корпуса»).

Чаще всего употребляется схема взвешивания tf-idf (частота терма – обратная частота документа), в которой *частота терма* w в документе d определяется как

$$\mathrm{tf}_{w,d} = \frac{n_w}{\sum_j n_j},$$

где n_w – число вхождений w в d. Для нормировки эта величина делится на суммарное число вхождений всех слов в данном документе.

Обратная частота документа определяется по формуле

$$\mathrm{idf}_{w,d} = \log \frac{|D|}{|\{d : w \in d\}|},$$

где $|D|$ – число документов в корпусе D, а в знаменателе находится число документов d из D, содержащих терм w. Перемножение обеих величин дает вес tf-idf, который становится одним из элементов **v**. Подробнее об этом можно прочитать в статье википедии *https://ru.wikipedia.org/wiki/TF-IDF*.

Пока это все, что нужно знать. А теперь применим эту модель к индексированию и поиску изображений по визуальному содержанию.

[1] Часто говорят не «слово», а «терм», но смысл от этого не меняется.

7.2. Визуальные слова

Для применения методов анализа текста к изображениям нужно, прежде всего, придумать визуальный эквивалент слов. Обычно это делается с помощью локальных дескрипторов, например SIFT-дескрипторов, описанных в разделе 2.2. Идея в том, чтобы разбить пространство дескрипторов на некоторое количество типичных примеров и сопоставить каждому дескриптору изображения один из этих примеров. Типичные примеры определяются путем анализа обучающего набора изображений, их можно рассматривать как *визуальные слова*. Тогда множество всех слов составляет *визуальный словарь* (который иногда называют *визуальной кодовой книгой*). Такой словарь можно создать как для конкретной задачи или типа изображений, так и для представления визуального содержания в целом.

Для построения визуальных слов к дескрипторам-признакам, выделенным из большого обучающего набора изображений, применяется тот или иной алгоритм кластеризации. Чаще всего это алгоритм *K* средних[2], мы тоже выберем его. Визуальные слова – это не что иное, как набор векторов в данном пространстве дескрипторов; в случае алгоритма *K* средних это будут центроиды кластеров. Представление изображения с помощью гистограммы визуальных слов тогда называется моделью *набора визуальных слов*.

Проиллюстрируем эту идею на конкретном примере набора данных. Файл *first1000.zip* содержит первые 1000 изображений из эталонного набора для распознавания объектов, созданного в Кентуккийском университете (его еще называют «ukbench»). Полный набор, отчеты о качестве распознавания и сопровождающий код можно найти на странице *http://www.vis.uky.edu/~stewe/ukbench/*. Набор ukbench содержит много четверок изображений одной и той же сцены или объекта (четверки хранятся последовательно, т. е. изображения с индексами 0 ... 3, 4 ... 7 и т. д. относятся к одному и тому же объекту). На рис. 7.1 приведено несколько примеров из этого набора данных. В приложении А сам набор и способ получения данных описаны более детально.

Создание словаря

Для создания словаря визуальных слов нам необходимо, прежде всего, выделить дескрипторы. В данном случае мы будем использовать SIFT-дескрипторы. В результате выполнения следующего кода, в котором список `imlist` как обычно содержит имена файлов изображений:

[2] В более сложных случаях может применяться иерархический алгоритм *K* средних.

Рис. 7.1. Примеры изображений из набора ukbench (эталонный набор Кентуккийского университета для распознавания объектов)

```
nbr_images = len(imlist)
featlist = [ imlist[i][:-3]+'sift' for i in range(nbr_images)]

for i in range(nbr_images):
  sift.process_image(imlist[i],featlist[i])
```

мы получаем дескрипторные файлы для каждого изображения. Создайте файл *vocabulary.py* и поместите в него показанный ниже код класса, который служит для обучения словаря на некотором наборе изображений:

```
from scipy.cluster.vq import *
import vlfeat as sift

class Vocabulary(object):

  def __init__(self,name):
     self.name = name
     self.voc = []
     self.idf = []
     self.trainingdata = []
     self.nbr_words = 0

  def train(self,featurefiles,k=100,subsampling=10):
     """ Обучить словарь на признаках, хранящихся в файлах из
       списка featurefiles, применяя метод K средних, где
       k - число слов. Для ускорения можно использовать подвыборку
       из обучающих данных. """

     nbr_images = len(featurefiles)

     # читать признаки из файла
     descr = []
     descr.append(sift.read_features_from_file(featurefiles[0])[1])
     descriptors = descr[0] #stack all features for k-means
     for i in arange(1,nbr_images):
       descr.append(sift.read_features_from_file(featurefiles[i])[1])
       descriptors = vstack((descriptors,descr[i]))

     # K средних: последнее число определяет количество прогонов
     self.voc,distortion = kmeans(descriptors[::subsampling,:],k,1)
     self.nbr_words = self.voc.shape[0]

     # перебрать все обучающие изображения и спроецировать на словарь
     imwords = zeros((nbr_images,self.nbr_words))
     for i in range( nbr_images ):
       imwords[i] = self.project(descr[i])

     nbr_occurences = sum( (imwords > 0)*1 ,axis=0)

     self.idf = log( (1.0*nbr_images) / (1.0*nbr_occurences+1) )
     self.trainingdata = featurefiles

  def project(self,descriptors):
     """ Спроецировать дескрипторы на словарь для
       создания гистограммы слов. """

     # гистограмма визуальных слов
     imhist = zeros((self.nbr_words))
     words,distance = vq(descriptors,self.voc)
     for w in words:
```

```
            imhist[w] += 1

        return imhist
```

Класс `Vocabulary` содержит центроиды кластеров векторов слов *voc*, а также значения idf для каждого слова. Для обучения словаря на наборе изображений метод `train()` принимает список файлов с SIFT-дескрипторами и число k, желаемое количество слов в словаре. Есть также возможность использования подвыборки из обучающих данных при вызове функции `kmeans`, которая работает долго, если признаков слишком много.

Если изображения и файлы признаков находятся в одной папке на вашем компьютере, то следующая программа создаст словарь длины $k \approx 1000$ (мы снова предполагаем, что *imlist* – список имен файлов изображений):

```
import pickle
import vocabulary

nbr_images = len(imlist)
featlist = [ imlist[i][:-3]+'sift' for i in range(nbr_images) ]

voc = vocabulary.Vocabulary('ukbenchtest')
voc.train(featlist,1000,10)

# сохранение словаря
with open('vocabulary.pkl', 'wb') as f:
  pickle.dump(voc,f)
print 'словарь:', voc.name, voc.nbr_words
```

В последней части весь словарь сохраняется в pickle-файле для последующего использования.

7.3. Индексирование изображений

Прежде чем приступать к поиску, необходимо подготовить базу изображений и их представления с помощью визуальных слов.

Подготовка базы данных

Перед тем как индексировать изображения, нужно подготовить базу данных. Под индексированием в этом контексте понимается выделение дескрипторов из изображений, преобразование их в визуаль-

ные слова и сохранение визуальных слов и гистограмм слов вместе с информацией о том, из какого изображения они взяты. После этого можно будет опрашивать базу данных, предъявив изображение, и получать в ответ похожие изображения.

В качестве базы данных мы используем SQLite. В этой базе все данные хранятся в одном файле, поэтому настроить ее и работать с ней очень легко. Мы выбрали SQLite, потому что это самый простой способ сделать дело, не отвлекаясь на вопросы конфигурирования сервера и прочие детали, не относящиеся к теме книги. Версию для Python, `pysqlite`, можно найти на странице *http://code.google.com/p/pysqlite/* и во многих репозиториях пакетов для Mac и Linux. В SQLite используется язык запросов SQL, поэтому переход на другую базу – если возникнет такое желание – не должен вызвать трудностей.

Для начала создадим таблицы и индексы, а также класс индексатора для записи изображения в базу данных. Создайте файл *imagesearch.py* и поместите в него такой код:

```
import pickle
from pysqlite2 import dbapi2 as sqlite

class Indexer(object):

  def __init__(self,db,voc):
     """ Инициализировать, задав имя базы данных и объект словаря. """

     self.con = sqlite.connect(db)
     self.voc = voc

  def __del__(self):
     self.con.close()

  def db_commit(self):
     self.con.commit()
```

Прежде всего, нам понадобится модуль `pickle` для кодирования массивов в виде строк и обратного декодирования. SQLite импортируется из модуля `pysqlite2` (сведения об установке см. в приложении А). Класс `Indexer` открывает соединение с базой данных и сохраняет объекта словаря (в методе `__init__()`). Метод `__del__()` закрывает соединение, а метод `db_commit()` записывает изменения в файл базы данных.

Нам нужна очень простая схема базы, содержащая всего три таблицы. В таблице `imlist` хранятся имена файлов всех индексированных изображений, а в таблице `imwords` – *индекс слова*, какой словарь ис-

пользовался и в каком изображении встретилось слово. Наконец, в таблице `imhistograms` хранятся полные гистограммы слов для каждого изображения. Они нужны для сравнения изображений, как принято в векторной модели. Схема базы показана в табл. 7.1.

Таблица 7.1. Простая схема базы данных для хранения изображений и визуальных слов

imlist	imwords	imhistograms
rowid	imid	imid
filename	wordid	histogram
	vocname	vocname

Следующий метод класса `Indexer` создает таблицы и некоторые полезные индексы для ускорения поиска:

```
def create_tables(self):
  """ Создать таблицы базы данных. """
  self.con.execute('create table imlist(filename)')
  self.con.execute('create table imwords(imid,wordid,vocname)')
  self.con.execute('create table imhistograms(imid,histogram,vocname)')
  self.con.execute('create index im_idx on imlist(filename)')
  self.con.execute('create index wordid_idx on imwords(wordid)')
  self.con.execute('create index imid_idx on imwords(imid)')
  self.con.execute('create index imidhist_idx on imhistograms(imid)')
  self.db_commit()
```

Добавление изображений

После того как таблицы созданы, мы можем приступить к добавлению изображений в индекс. Для этого понадобится метод `add_to_index()`, который мы включим в класс `Indexer`. Добавьте его в файл *imagesearch.py*:

```
def add_to_index(self,imname,descr):
  """ Взять изображение с дескрипторами, спроецировать на
      словарь и добавить в базу данных. """

  if self.is_indexed(imname): return
  print 'производится индексирование', imname

  # получить imid
  imid = self.get_id(imname)

  # получить слова
  imwords = self.voc.project(descr)
```

```
    nbr_words = imwords.shape[0]

    # связать каждое слово с изображением
    for i in range(nbr_words):
        word = imwords[i]

        # wordid - это номер слова
        self.con.execute("insert into imwords(imid,wordid,vocname)
              values (?,?,?)", (imid,word,self.voc.name))

    # сохранить гистограмму слов для изображения
    # использовать pickle для представления массивов NumPy в виде строк
    self.con.execute("insert into imhistograms(imid,histogram,vocname)
          values (?,?,?)", (imid,pickle.dumps(imwords),self.voc.name))
```

Этот метод принимает имя файла изображения и массив `NumPy`, содержащий найденные в изображении дескрипторы. Дескрипторы проецируются на словарь и вставляются в таблицы `imwords` (пословно) и `imhistograms`. Мы воспользовались двумя вспомогательными функциями: `is_indexed()` проверяет, было ли уже изображение индексировано, а `get_id()` возвращает идентификатор изображения по имени файла. Добавьте их в файл *imagesearch.py*:

```
def is_indexed(self,imname):
  """ Возвращает True, если imname уже проиндексировано. """
  im = self.con.execute("select rowid from imlist where
        filename='%s'" % imname).fetchone()
  return im != None

def get_id(self,imname):
  """ Получить имя файла и добавить в таблицу, если его еще нет. """
  cur = self.con.execute(
     "select rowid from imlist where filename='%s'" % imname)
  res=cur.fetchone()
  if res==None:
     cur = self.con.execute(
        "insert into imlist(filename) values ('%s')" % imname)
     return cur.lastrowid
  else:
     return res[0]
```

Вы заметили, что мы использовали `pickle` в методе `add_to_index()`? В таких базах данных, как SQLite, нет стандартного типа данных для хранения объектов или массивов. Но мы можем получить строковое представление объекта с помощью функции `dumps()` из модуля `pickle` и сохранить его в базе. А после чтения строки из базы ее нужно будет преобразовать обратно в объект. Подробнее об этом в следующем разделе.

Приведенная ниже программа перебирает все изображения в наборе ukbench и добавляет их в индекс. Здесь предполагается, что в списках *imlist* и *featlist* хранятся имена файлов изображений и дескрипторов и что обученный ранее словарь был сериализован и сохранен в файле *vocabulary.pkl*:

```
import pickle
import sift
import imagesearch

nbr_images = len(imlist)

# загрузить словарь
with open('vocabulary.pkl', 'rb') as f:
  voc = pickle.load(f)

# создать индексатор
indx = imagesearch.Indexer('test.db',voc)
indx.create_tables()

# перебрать все изображения, спроецировать признаки на словарь и вставить
for i in range(nbr_images)[:100]:
  locs,descr = sift.read_features_from_file(featlist[i])
  indx.add_to_index(imlist[i],descr)

# зафиксировать в базе данных
indx.db_commit()
```

Теперь можно просмотреть содержимое нашей базы данных:

```
from pysqlite2 import dbapi2 as sqlite
con = sqlite.connect('test.db')
print con.execute('select count (filename) from imlist').fetchone()
print con.execute('select * from imlist').fetchone()
```

На консоли будет напечатано:

```
(1000,)
(u'ukbench00000.jpg',)
```

Если в последней строке вызвать метод `fetchall()`, а не `fetchone()`, то будет выведен длинный список всех имен файлов.

7.4. Поиск изображений в базе данных

После того как набор изображений проиндексирован, мы можем искать похожие изображения в базе. В данном случае мы использовали

представление всего изображения в виде набора слов, но описанная ниже процедура общая и может быть применена к поиску похожих предметов, похожих лиц, похожих цветов и т. д. Все зависит от изображений и того, какие дескрипторы использовались.

Для поиска добавим в файл *imagesearch.py* класс `Searcher`:

```
class Searcher(object):

  def __init__(self,db,voc):
     """ Инициализировать, передав имя базы данных. """

     self.con = sqlite.connect(db)
     self.voc = voc

  def __del__(self):
     self.con.close()
```

Объект `Searcher` открывает соединение с базой данных и закрывает его в момент удаления – точно так же, как объект `Indexer`.

Если количество изображений велико, то на практике невозможно сравнить полные гистограммы для всех хранящихся в базе изображений. Нужно как-то отыскать множество кандидатов приемлемого размера (что такое «приемлемый», определяется допустимым временем реакции, требованиями к памяти и т. д.). Вот тут-то и выходят на сцену индексы. С помощью индекса мы можем отобрать множество кандидатов и уже для них провести полное сравнение.

Использование индекса для получения кандидатов

Используя индекс, мы можем найти все изображения, содержащие определенное слово. Это простой запрос к базе данных. Добавьте в класс `Searcher` метод `candidates_from_word()`:

```
def candidates_from_word(self,imword):
  """ Получить список изображений, содержащих слово imword. """

  im_ids = self.con.execute(
     "select distinct imid from imwords where wordid=%d" % imword).fetchall()
  return [i[0] for i in im_ids]
```

Этот метод дает идентификаторы всех изображений, содержащих указанное слово. Чтобы получить кандидатов для нескольких слов, например всех ненулевых элементов гистограммы, мы можем в цикле перебрать все слова, получить изображения для каждого слова и

агрегировать списки[3]. Здесь следует также запоминать, сколько раз каждый идентификатор слова встречается в агрегированном списке, поскольку это говорит о том, сколько слов совпадает с теми, что представлены в гистограмме. Все это делает следующий метод из класса `Searcher`:

```
def candidates_from_histogram(self,imwords):
  """ Получить список изображений с похожими словами. """

  # получить идентификаторы слов
  words = imwords.nonzero()[0]

  # найти кандидаты
  candidates = []
  for word in words:
     c = self.candidates_from_word(word)
     candidates+=c

  # взять все уникальные слова и отсортировать по убыванию числа вхождений
  tmp = [(w,candidates.count(w)) for w in set(candidates)]
  tmp.sort(cmp=lambda x,y:cmp(x[1],y[1]))
  tmp.reverse()

  # вернуть отсортированный список, поместив лучших кандидатов в начало
  return [w[0] for w in tmp]
```

Этот метод создает список идентификаторов слов в ненулевых элементах гистограммы слов изображения. Кандидаты для каждого слова ищутся в индексе и агрегируются в список *candidates*. Затем создается список кортежей (ид слова, счетчик), содержащий количество вхождений каждого слова в список кандидатов, и этот список сортируется (на месте – для большей эффективности) с помощью метода `sort()` со специализированной функцией сравнения, которая сравнивает вторые элементы кортежа. Функция сравнения анонимна, т. е. объявляется тут же с помощью нотации лямбда-выражений. Результат возвращается в виде списка идентификаторов изображений, в начале которого находятся лучшие кандидаты.

Рассмотрим следующий пример:

```
src = imagesearch.Searcher('test.db')
locs,descr = sift.read_features_from_file(featlist[0])
iw = voc.project(descr)

print 'запрос с использованием гистограммы...'
print src.candidates_from_histogram(iw)[:10]
```

[3] Если не хотите использовать все слова, попробуйте ранжировать их по весу idf и взять только слова с наибольшими весами.

Печатаются первые 10 найденных в индексе изображений в таком виде (для вашего словаря результат может быть другим):

```
запрос с использованием гистограммы...
[655, 656, 654, 44, 9, 653, 42, 43, 41, 12]
```

Все десять лучших кандидатов неправильны. Но не огорчайтесь; мы можем теперь взять из этого списка любое число элементов и сравнить гистограммы. Как вы увидите, это даст существенное улучшение.

Запрос по изображению

Для поиска изображений, похожих на предъявленное, больше почти ничего не нужно. Чтобы сравнить гистограммы, объект `Searcher` должен уметь читать гистограммы слов из базы. Добавьте следующий метод в класс `Searcher`:

```
def get_imhistogram(self,imname):
  """ Вернуть гистограмму слов для изображения. """

  im_id = self.con.execute(
    "select rowid from imlist where filename='%s'" % imname).fetchone()
  s = self.con.execute(
    "select histogram from imhistograms where rowid='%d'" % im_id).fetchone()

  # с помощью pickle декодировать строковое представление массивов NumPy
  return pickle.loads(str(s[0]))
```

И снова мы используем модуль `pickle` – на этот раз для преобразования строки в массив `NumPy` с помощью функции `loads()`.

Теперь можно собрать все в метод запроса:

```
def query(self,imname):
  """ Построить список изображений, похожих на imname. """

  h = self.get_imhistogram(imname)
  candidates = self.candidates_from_histogram(h)

  matchscores = []
  for imid in candidates:
     # получить имя
     cand_name = self.con.execute(
        "select filename from imlist where rowid=%d" % imid).fetchone()
     cand_h = self.get_imhistogram(cand_name)
     cand_dist = sqrt( sum( (h-cand_h)**2 ) ) #используется метрика L2
     matchscores.append( (cand_dist,imid) )

     # вернуть отсортированный список расстояний и ид в базе данных
```

```
        matchscores.sort()
        return matchscores
```

Этот метод принимает имя файла изображения и читает из базы гистограмму слов и список кандидатов (размер которого следует ограничить, если набор данных очень велик). Для каждого кандидата мы сравниваем гистограммы, применяя стандартную евклидову метрику, и возвращаем отсортированный список кортежей, содержащих расстояние и идентификатор изображения.

Попробуем выполнить запрос по тому же изображению, что в предыдущем разделе:

```
src = imagesearch.Searcher('test.db')
print 'выполняется запрос...'
print src.query(imlist[0])[:10]
```

Снова будут напечатаны 10 лучших результатов, включая расстояние:

```
выполняется запрос...
[(0.0, 1), (100.03999200319841, 2), (105.45141061171255, 3), (129.47200469599596, 708),
(129.73819792181484, 707), (132.68006632497588, 4), (139.89639023220005, 10),
(142.31654858097141, 706), (148.1924424523734, 716), (148.22955170950223, 663)]
```

Гораздо лучше. Расстояние от изображения до него самого равно нулю, и в первых двух позициях находятся два из трех изображений той же самой сцены. Третье изображение оказалось в пятой позиции.

Эталонное тестирование и построение графика

Чтобы лучше понять, насколько хороши результаты поиска, вычислим количество правильных изображений в первых четырех позициях. Именно этот показатель использовался для оценки качества при работе с набором данных ukbench. Ниже показана функция для его вычисления. Добавьте ее в файл *imagesearch.py* и можете приступать к оптимизации запросов.

```
def compute_ukbench_score(src,imlist):
  """ Возвращает среднее число правильных изображений среди
      первых четырех результатов запроса."""

  nbr_images = len(imlist)
  pos = zeros((nbr_images,4))

  # получить первые четыре результата для каждого изображения
```

```
    for i in range(nbr_images):
        pos[i] = [w[1]-1 for w in src.query(imlist[i])[:4]]

    # вычислить оценку и вернуть среднее
    score = array([ (pos[i]//4)==(i//4) for i in range(nbr_images)])*1.0
    return sum(score) / (nbr_images)
```

Эта функция получает первые четыре результата и вычитает единицу из индекса, возвращенного функцией `query()`, потому что нумерация в базе данных начинается с единицы, а в списке изображений – с нуля. Затем мы вычисляем оценку с помощью операции целочисленного деления, памятуя о том, что правильные изображения объединены в группы по четыре. Если результат идеальный, то получится оценка 4, если не найдено ни одного правильного изображения, то 0, а если возвращаются только в точности совпадающие изображения, то оценка будет равна 1. Если найдено в точности совпадающее изображение и еще два из трех прочих, то получится оценка 3.

Попробуйте выполнить такое предложение:

```
imagesearch.compute_ukbench_score(src,imlist)
```

А если не хотите ждать (для выполнения 1000 запросов потребуется некоторое время), возьмите подмножество всех изображений:

```
imagesearch.compute_ukbench_score(src,imlist[:100])
```

В данном случае можно считать хорошей оценку, близкую к 3. Применение самых современных методов, о которых сообщается на сайте ukbench, дает оценку чуть выше 3 (но при этом они используют больше изображений, а наша оценка с увеличением размера набора данных снижается).

Наконец, полезна функция показа найденных результатов. Добавьте ее в файл.

```
def plot_results(src,res):
  """ Показать изображения из списка результатов res. """

  figure()
  nbr_results = len(res)
  for i in range(nbr_results):
      imname = src.get_filename(res[i])
      subplot(1,nbr_results,i+1)
      imshow(array(Image.open(imname)))
      axis('off')
  show()
```

При вызове ей передается список *res*, содержащий произвольное число результатов поиска, например:

```
nbr_results = 6
res = [w[1] for w in src.query(imlist[0])[:nbr_results]]
imagesearch.plot_results(src,res)
```

Вспомогательная функция

```
def get_filename(self,imid):
  """ Вернуть имя файла изображения с идентификатором id. """

  s = self.con.execute(
     "select filename from imlist where rowid='%d'" % imid).fetchone()
  return s[0]
```

преобразует идентификатор изображения в имя соответствующего файла, которое необходимо при загрузке изображений для нанесения на график. На рис. 7.2 показаны результаты запросов к нашему набору данных, построенные с помощью `plot_results()`.

Рис. 7.2. Примеры результатов поиска в наборе данных ukbench. Слева показано изображение-запрос, а за ним – первые пять найденных изображений

7.5. Ранжирование результатов с применением геометрических соображений

Рассмотрим кратко распространенный способ улучшения результатов, полученных с помощью модели набора визуальных слов. Один из недостатков этой модели заключается в том, что представление изображения с помощью визуальных слов не содержит позиций признаков. Это та цена, которую мы платим за быстродействие и масштабируемость.

Чтобы улучшить результаты, учтя позиции точек-признаков, мы можем изменить ранжирование лучших результатов, применив некоторый критерий, принимающий в расчет геометрические связи между признаками. Чаще всего подбирают гомографии между позициями признаков в изображении-запросе и в лучших найденных изображениях-результатах.

Чтобы это эффективно работало, мы можем сохранить в базе данных позиции признаков и определять соответствия по идентификаторам слов признаков (это работает, только если словарь достаточно велик, так что множество соответствий по идентификаторам слов содержит в основном правильные соответствия). Но такое изменение потребовало бы существенной переделки схемы базы и переписывания всего кода, что усложнило бы изложение. Для иллюстрации мы просто заново загрузим признаки первых найденных результатов и сопоставим их.

Ниже приведен полный код загрузки всех файлов модели и повторного ранжирования лучших результатов с помощью гомографий.

```
import pickle
import sift
import imagesearch
import homography

# загрузить список изображений и словарь
with open('ukbench_imlist.pkl','rb') as f:
  imlist = pickle.load(f)
  featlist = pickle.load(f)

nbr_images = len(imlist)

with open('vocabulary.pkl', 'rb') as f:
```

```
  voc = pickle.load(f)

src = imagesearch.Searcher('test.db',voc)

# индекс изображения-запроса и количество возвращаемых результатов
q_ind = 50
nbr_results = 20

# регулярный запрос
res_reg = [w[1] for w in src.query(imlist[q_ind])[:nbr_results]]
print 'лучшие соответствия (обычное решение):', res_reg

# загрузить признаки изображения-запроса
q_locs,q_descr = sift.read_features_from_file(featlist[q_ind])
fp = homography.make_homog(q_locs[:,:2].T)

# модель RANSAC для подбора гомографии
model = homography.RansacModel()

rank = {}

# загрузить признаки изображений-результатов
for ndx in res_reg[1:]:
  locs,descr = sift.read_features_from_file(featlist[ndx])

  # найти соответствия
  matches = sift.match(q_descr,descr)
  ind = matches.nonzero()[0]
  ind2 = matches[ind]
  tp = homography.make_homog(locs[:,:2].T)

  # вычислить гомографию и регулярные точки. Если соответствий
  # недостаточно, вернуть пустой список.
  try:
     H,inliers = homography.H_from_ransac(fp[:,ind],tp[:,ind2],model,
                                          match_theshold=4)
  except:
     inliers = []

  # сохранить число регулярных точек
  rank[ndx] = len(inliers)

# отсортировать словарь, так чтобы сначала шли результаты с
# наибольшим числом регулярных точек
sorted_rank = sorted(rank.items(), key=lambda t: t[1], reverse=True)
res_geom = [res_reg[0]]+[s[0] for s in sorted_rank]
print 'лучшие результаты (с гомографией):', res_geom

# нанести лучшие результаты на график
imagesearch.plot_results(src,res_reg[:8])
imagesearch.plot_results(src,res_geom[:8])
```

Сначала загружаются список изображений, список признаков (имена файлов изображений и файлы с SIFT-признаками соответ-

ственно) и словарь. Затем создается объект `Searcher`, выполняется обычный запрос, и его результаты сохраняются в списке *res_reg*. Далее загружаются признаки для изображения-запроса. Затем для каждого изображения из списка результатов загружаются признаки и производится сопоставление с изображением-запросом. По найденным соответствиям вычисляются гомографии и регулярные точки. Если подобрать гомографию не удалось, то список регулярных точек делается пустым. Наконец, мы сортируем словарь *rank*, который содержит индекс изображения и число регулярных точек, в порядке убывания последнего. Списки результатов выводятся на консоль, а лучшие изображения еще и показываются на графике.

Вывод выглядит следующим образом:

```
  лучшие результаты (обычное решение): [39, 22, 74, 82, 50, 37,
38, 17, 29, 68, 52, 91, 15, 90, 31, ... ]
  лучшие результаты (с гомографией): [39, 38, 37, 45, 67, 68, 74,
82, 15, 17, 50, 52, 85, 22, 87, ... ]
```

На рис. 7.3 показаны примеры результатов обычного поиска и после повторного ранжирования лучших изображений.

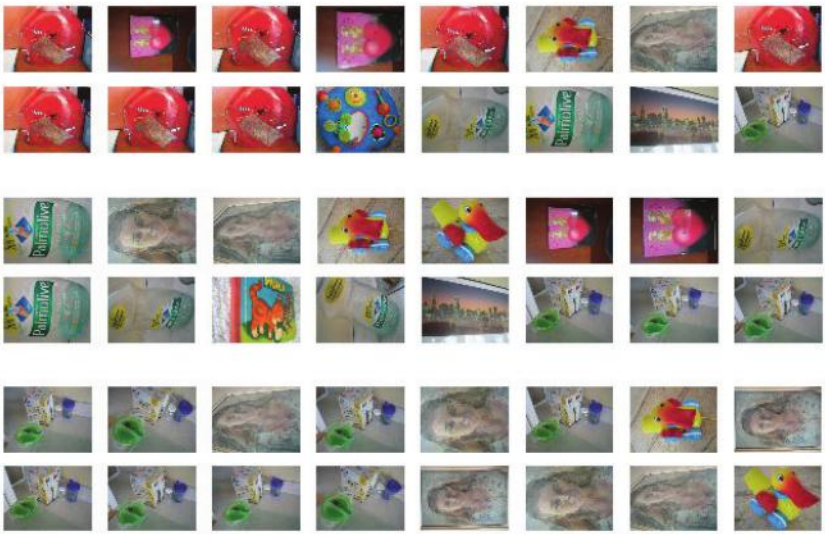


Рис. 7.3. Примеры результатов поиска с повторным ранжированием с учетом геометрической согласованности, устанавливаемой путем вычисления гомографии. В каждом случае верхняя строка содержит результат обычного поиска, а нижняя – результат после повторного ранжирования

7.6. Создание демонстраций и веб-приложений

В последнем разделе главы, посвященной поиску, мы рассмотрим простой способ создания демонстраций и веб-приложений с помощью Python. Оформив демонстрацию в виде веб-страницы, мы автоматически получаем кросс-платформенную поддержку и возможность показать свой проект и поделиться им, не предъявляя чрезмерных требований. Ниже мы опишем построение простой системы поиска изображений.

Создание веб-приложений с помощью CherryPy

Для создания демонстраций мы воспользуемся пакетом `CherryPy`, который можно скачать с сайта *http://www.cherrypy.org/*. Это написанный на чистом Python облегченный веб-сервер на основе объектно-ориентированной модели. О том, как установить и настроить `CherryPy`, написано в приложении А. В предположении, что вы изучили примеры в пособии и в общих чертах понимаете, как работает `CherryPy`, построим веб-демонстрацию поиска изображений на основе системы, разработанной нами ранее.

Демонстрация поиска изображений

Начать следует с включения начальных html-тегов и загрузки данных из pickle-файла. Нам понадобится словарь для объекта `Searcher`, который будет взаимодействовать с базой данных. Создайте файл *searchdemo.py* и поместите в него следующий класс с двумя методами:

```
import cherrypy, os, urllib, pickle
import imagesearch

class SearchDemo(object):

  def __init__(self):
     # загрузить список изображений
     with open('webimlist.txt') as f:
        self.imlist = f.readlines()

     self.nbr_images = len(self.imlist)
     self.ndx = range(self.nbr_images)

     # загрузить словарь
     with open('vocabulary.pkl', 'rb') as f:
```

```
            self.voc = pickle.load(f)

        # задать максимальное число отображаемых результатов
        self.maxres = 15

        # html-код верхнего и нижнего колонтитула
        self.header = """
            <!doctype html>
            <head>
            <title>Image search example</title>
            </head>
            <body>
            """

        self.footer = """
            </body>
            </html>
            """

    def index(self,query=None):
        self.src = imagesearch.Searcher('web.db',self.voc)
        html = self.header
        html += """
            <br />
            Click an image to search.
            <a href='?query='>Random selection</a> of images.
            <br /><br />
            """

        if query:
            # опросить базу данных и получить лучшие изображения
            res = self.src.query(query)[:self.maxres]
            for dist,ndx in res:
                imname = self.src.get_filename(ndx)
                html += "<a href='?query="+imname+"'>"
                html += "<img src='"+imname+"' width='100' />"
                html += "</a>"
        else:
            # показать случайную выборку, если запроса не было
            random.shuffle(self.ndx)
            for i in self.ndx[:self.maxres]:
                imname = self.imlist[i]
                html += "<a href='?query="+imname+"'>"
                html += "<img src='"+imname+"' width='100' />"
                html += "</a>"

        html += self.footer
        return html

    index.exposed = True

cherrypy.quickstart(SearchDemo(), '/',
    config=os.path.join(os.path.dirname(__file__), 'service.conf'))
```

Как видите, эта простая демонстрация состоит из единственного класса с двумя методами: для инициализации и для вывода «индексной» страницы (в данном случае страница всего одна). Методы автоматически отображаются на URL-адреса, а их аргументы передаются в виде параметров URL. У метода `index` есть параметр, определяющий изображение-запрос, с которым сравниваются остальные. Если он не задан, то показывается просто случайная выборка изображений. Строка

```
index.exposed = True
```

делает URL индексной страницы доступным, а последняя строка запускает веб-сервер `CherryPy` в конфигурации, прочитанной из файла *service.conf*. В этом примере конфигурационный файл выглядит следующим образом:

```
[global]
server.socket_host = "127.0.0.1"
server.socket_port = 8080
server.thread_pool = 50
tools.sessions.on = True

[/]
tools.staticdir.root = "tmp/"
tools.staticdir.on = True
tools.staticdir.dir = ""
```

В первом разделе задается IP-адрес и номер прослушиваемого порта, а во втором открывается доступ для чтения к локальной папке (в данном случае «tmp/»). В этой папке должны находиться изображения.

> Не помещайте ничего конфиденциального в эту папку, если планируете показывать ее содержимое пользователям. Сервер CherryPy делает доступным все, что находится в папке.

Запустите веб-сервер командой

```
$ python searchdemo.py
```

Откройте браузер и перейдите на URL-адрес *http://127.0.0.1:8080/* – вы должны увидеть начальную страницу, содержащую случайную выборку изображений (см. рис. 7.4). Щелчок по изображению запускает

поиск, после чего будут показаны результаты. Щелчок по одному из найденных изображений запускает новый поиск, уже по нему, и так далее. Имеется ссылка, позволяющая вернуться в начальное состояние, когда показывается случайная выборка (путем передачи пустого запроса).

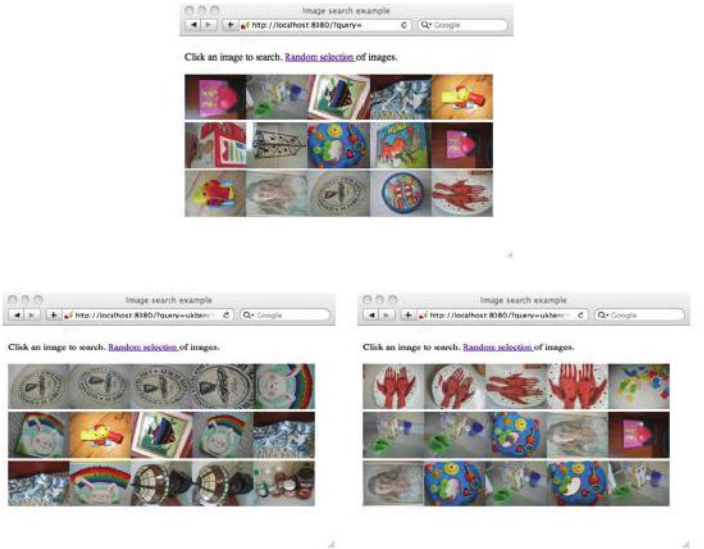


Рис. 7.4. Примеры результатов поиска по набору изображений ukbench: начальная страница со случайной выборкой изображений (вверху); результаты поиска (внизу). Изображение-запрос показано в левом верхнем углу, а за ним, в той же строке – лучшие найденные результаты

Этот пример демонстрирует интеграцию веб-страницы с запросами к базе данных и представлением результатов. Разумеется, мы не увлекались стилизацией и дополнительными возможностями, так что возможностей для улучшения много. Например, можно добавить таблицу стилей, чтобы сделать страницу симпатичнее, или загружать файлы изображений, используемых в качестве запроса.

Упражнения

1. Попробуйте ускорить поиск, используя только часть слов в изображении-запросе для построения списка кандидатов. В качестве критерия отбора слова возьмите вес idf.

2. Реализуйте список визуальных стоп-слов, т. е. наиболее часто встречающихся слов в словаре (скажем, первые 10 %), и игнорируйте эти слова при поиске. Как при этом изменится качество поиска?
3. Наглядно представьте визуальные слова, сохранив все признаки изображения, которые отображаются на заданный идентификатор слова. Вырежьте блоки изображения в окрестности каждого признака (в заданном масштабе) и нанесите их на график. Одинаково ли выглядят блоки для данного слова?
4. Поэкспериментируйте с использованием разных метрик и весов в методе `query()`. Для измерения качества воспользуйтесь функцией `compute_ukbench_score()`.
5. В этой главе мы включали в словарь только SIFT-признаки. При этом полностью игнорируется цветовая информация, что видно из результатов поиска на рис. 7.2. Попробуйте добавить цветовые дескрипторы и посмотрите, улучшатся ли при этом результаты поиска.
6. В случае больших словарей использование массивов для представления частот визуальных слов неэффективно, поскольку большинство элементов будут равны нулю (представьте, что слов несколько сотен тысяч, а типичное изображение содержит порядка тысячи признаков). Один из способов устранить эту неэффективность – воспользоваться словарями как средством представления разреженных массивов. Замените массив своим разреженным классом и добавьте необходимые методы. Или воспользуйтесь модулем `scipy.sparse`.
7. При увеличении размера словаря чрезмерно возрастает время кластеризации, замедляется также проецирование признаков на слова. Реализуйте иерархический словарь, воспользовавшись иерархическим методом *K* средних, и посмотрите, насколько улучшится масштабируемость. Подробности и идеи для работы можно найти в статье [23].

ГЛАВА 8. Классификация изображений по содержанию

В этой главе мы познакомимся с алгоритмами классификации изображений по содержанию. Мы рассмотрим как простые, но эффективные методы, так и современные классификаторы и применим их к задачам бинарной и многоклассовой классификации. В качестве примеров будут приведены приложения, связанные с распознаванием жестов и объектов.

8.1. Метод k ближайших соседей

Один из самых простых и наиболее употребительных методов – классификация по *k ближайшим соседям* (*kNN*). Этот алгоритм сравнивает подлежащий классификации объект (например, вектор признаков) со всеми объектами из обучающего набора с известными метками классов и назначает ему класс, исходя из классов k ближайших соседей. Зачастую метод показывает хорошие результаты, но имеет ряд недостатков. Как и в случае алгоритма кластеризации методом K средних, число k нужно задавать заранее, а от него зависит качество работы. Кроме того, требуется хранить весь обучающий набор, а если он велик, то поиск становится медленным. Для больших обучающих наборов обычно используют ту или иную форму разбиения на интервалы с целью сократить количество необходимых сравнений[1]. К числу плюсов следует отнести отсутствие ограничений на выбор метрики; подойдет практически любая (правда, это не значит, что результаты окажутся хорошими). К тому же, алгоритм тривиально распараллеливается.

[1] Еще один вариант – хранить только подмножество обучающего набора. Однако при этом может снизиться точность.

Реализация kNN в простейшей форме вполне прямолинейна. Показанный ниже код решает задачу, если известны обучающие примеры и соответствующие им метки. Примеры и метки можно передать в виде строк массива или просто списками. Это могут быть числа, строки – вообще все, что угодно. Добавьте следующий класс в файл *knn.py*:

```
class KnnClassifier(object):

  def __init__(self,labels,samples):
     """ Инициализировать классификатор, передав обучающие данные. """

     self.labels = labels
     self.samples = samples

  def classify(self,point,k=3):
     """ Классифицировать точку по k ближайшим к ней точкам
        обучающего набора, вернуть метку. """

     # вычислить расстояния до всех точек из обучающего набора
     dist = array([L2dist(point,s) for s in self.samples])

     # отсортировать их
     ndx = dist.argsort()

     # сохранить k ближайших в словаре
     votes = {}
     for i in range(k):
        label = self.labels[ndx[i]]
        votes.setdefault(label,0)
        votes[label] += 1

     return max(votes)

  def L2dist(p1,p2):
     return sqrt( sum( (p1-p2)**2) )
```

Проще всего определить класс и инициализировать его обучающими данными. Тогда нам не придется хранить и передавать обучающие данные в виде аргументов всякий раз, как нужно будет что-то классифицировать. Если для хранения k ближайших меток используется словарь, то сами метки могут быть строками или числами. В данном случае в качестве метрики мы взяли евклидово расстояние (L_2). Другие метрики можно добавить в виде дополнительных функций.

Простой двумерный пример

Создадим несколько простых наборов точек на плоскости и наглядно покажем, как работает классификатор. Следующий скрипт

создает два набора точек, принадлежащих двум классам, и сохраняет данные с помощью модуля `pickle`:

```
from numpy.random import randn
import pickle

# создать демонстрационные наборы точек на плоскости
n = 200

# два нормальных распределения
class_1 = 0.6 * randn(n,2)
class_2 = 1.2 * randn(n,2) + array([5,1])
labels = hstack((ones(n),-ones(n)))

# сохранить с помощью pickle
with open('points_normal.pkl', 'w') as f:
  pickle.dump(class_1,f)
  pickle.dump(class_2,f)
  pickle.dump(labels,f)

# нормальное распределение и кольцо вокруг него
class_1 = 0.6 * randn(n,2)
r = 0.8 * randn(n,1) + 5
angle = 2*pi * randn(n,1)
class_2 = hstack((r*cos(angle),r*sin(angle)))
labels = hstack((ones(n),-ones(n)))

# сохранить с помощью pickle
with open('points_ring.pkl', 'w') as f:
  pickle.dump(class_1,f)
  pickle.dump(class_2,f)
  pickle.dump(labels,f)
```

Выполните скрипт дважды с разными именами файлов, например, сначала `points_normal_test.pkl`, а затем `points_ring_test.pkl`. Теперь у вас четыре файла с наборами данных, по два для каждого распределения. Один будем использоваться для обучения, другой – для тестирования.

Посмотрим, как решить задачу с помощью kNN-классификатора. Введите такой скрипт:

```
import pickle
import knn
import imtools

# загрузить точки с помощью pickle
with open('points_normal.pkl', 'r') as f:
class_1 = pickle.load(f)
```

```
class_2 = pickle.load(f)
labels = pickle.load(f)
model = knn.KnnClassifier(labels,vstack((class_1,class_2)))
```

Он создает *модель* kNN-классификатора на основе данных в pickle-файле. Добавьте далее такой код:

```
# загрузить тестовые данные с помощью pickle
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# протестировать на первой точке
print model.classify(class_1[0])
```

Здесь мы загружаем другой набор данных (тестовый) и печатаем на консоли предсказанную метку класса для первой точки.

Чтобы визуализировать классификацию всех тестовых точек и показать, насколько хорошо классификатор разделяет два класса, добавим такие строчки:

```
# определить функцию построения графика
def classify(x,y,model=model):
  return array([model.classify([xx,yy]) for (xx,yy) in zip(x,y)])

# нанести на график границу между классами
imtools.plot_2D_boundary([-6,6,-6,6],[class_1,class_2],classify,[1,-1])
show()
```

Здесь мы создали простенькую вспомогательную функцию, которая принимает массивы x и y, содержащие координаты точек, а также классификатор и возвращает массив предсказанных меток классов. Эту функцию можно передать в качестве аргумента функции, которая строит график. Добавьте показанную ниже функцию в файл *imtools*:

```
def plot_2D_boundary(plot_range,points,decisionfcn,labels,values=[0]):
  """ plot_range – это диапазон(xmin,xmax,ymin,ymax), points – список
      точек, decisionfcn – функция, принимающая решение,
      labels – массив меток классов, который возвращает decisionfcn,
      values – список подлежащих показу изолиний решающей функции. """

  clist = ['b','r','g','k','m','y'] # цвета, соответствующие классам

  # вычислить и нанести на сетку изолинию решающей функции
  x = arange(plot_range[0],plot_range[1],.1)
  y = arange(plot_range[2],plot_range[3],.1)
```

```
xx,yy = meshgrid(x,y)
xxx,yyy = xx.flatten(),yy.flatten() # списки x,y на сетке
zz = array(decisionfcn(xxx,yyy))
zz = zz.reshape(xx.shape)

# нанести на график изолинии из списка values
contour(xx,yy,zz,values)

# для каждого класса нанести на график точки, обозначив '*'
# правильные, 'o' - неправильные
for i in range(len(points)):
  d = decisionfcn(points[i][:,0],points[i][:,1])
  correct_ndx = labels[i]==d
  incorrect_ndx = labels[i]!=d
  plot(points[i][correct_ndx,0],points[i][correct_ndx,1],'*',
       color=clist[i])
  plot(points[i][incorrect_ndx,0],points[i][incorrect_ndx,1],'o',
       color=clist[i])

axis('equal')
```

Эта функция принимает решающую функцию (классификатор) и вычисляет ее в узлах сетки с помощью функции `meshgrid()`. На график можно нанести изолинии решающей функции, чтобы показать, где проходят границы. По умолчанию подразумевается нулевая изолиния. Получается график, показанный на рис. 8.1. Как видим, решающая граница kNN-классификатора может адаптироваться к распределению классов без явного построения модели.

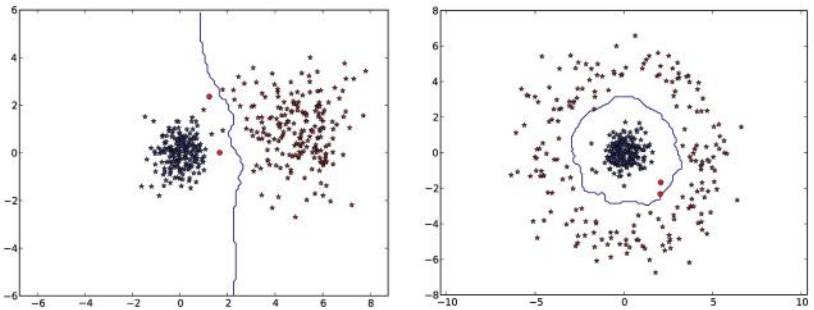


Рис. 8.1. Классификация точек на плоскости с помощью классификатора по *k* ближайшим соседям. Цветом кодируются классы меток. Правильно классифицированные точки показаны звездочками, неправильные – кружочками. Кривая на графике – решающая граница

Плотные SIFT-дескрипторы в качестве признаков изображения

Рассмотрим теперь классификацию изображений. Нам понадобится вектор признаков изображения. В главе, посвященной кластеризации, мы работали с векторами признаков, составленными из средних значений RGB-цветов и коэффициентов PCA. А сейчас введем так называемое *плотное SIFT-представление*.

Плотное SIFT-представление создается посредством применения дескрипторной части алгоритма SIFT к регулярной сетке, наложенной на все изображение[2]. Мы можем воспользоваться кодом из раздела 2.2 и получить SIFT-признаки на плотной сетке, добавив дополнительные параметры. Создайте файл *dsift.py* и поместите в него такую функцию:

```
import sift

def process_image_dsift(imagename,resultname,size=20,steps=10,
        force_orientation=False,resize=None):
  """ Обработать изображение с помощью SIFT-дескрипторов на
     плотной сетке и сохранить результаты в файле. Необязательные
     параметры: размер признаков, количество шагов между точками,
     принудительное вычисление ориентации дескриптора (False
     означает, что все дескрипторы направлены вверх), кортеж с
     новыми размерами изображения. """

  im = Image.open(imagename).convert('L')
  if resize!=None:
     im = im.resize(resize)
  m,n = im.size

  if imagename[-3:] != 'pgm':
     # создать pgm-файл
     im.save('tmp.pgm')
     imagename = 'tmp.pgm'

  # создать фреймы и сохранить во временном файле
  scale = size/3.0
  x,y = meshgrid(range(steps,m,steps),range(steps,n,steps))
  xx,yy = x.flatten(),y.flatten()
  frame = array([xx,yy,scale*ones(xx.shape[0]),zeros(xx.shape[0])])
  savetxt('tmp.frame',frame.T,fmt='%03.3f')

  if force_orientation:
```

[2] У этого представления есть и другое название: *гистограмма направленных градиентов* (Histogram of Oriented Gradients, HOG).

```
        cmmd = str("sift "+imagename+" --output="+resultname+
            " --read-frames=tmp.frame --orientations")
    else:
        cmmd = str("sift "+imagename+" --output="+resultname+
            " --read-frames=tmp.frame")
    os.system(cmmd)
    print 'processed', imagename, 'to', resultname
```

Сравните эту функцию с функцией `process_image()` из раздела 2.2. Мы вызываем функцию `savetxt()` для сохранения массива `frame` в текстовом файле для последующей обработки в пакетном режиме. Последний параметр этой функции можно использовать для изменения размера изображения перед выделением дескрипторов. Например, если передать *imsize*=(100, 100), то изображения будут преобразованы в квадратные размером 100×100 пикселей. Наконец, если параметр *force_orientation* равен True, то дескрипторы нормируются в направлении доминирующего локального градиента. В противном случае все дескрипторы просто направлены вверх.

Воспользуемся этой функцией для вычисления плотных SIFT-дескрипторов и визуализации места их расположения:

```
import dsift,sift

dsift.process_image_dsift('empire.jpg','empire.sift',90,40,True)
l,d = sift.read_features_from_file('empire.sift')

im = array(Image.open('empire.jpg'))
sift.plot_features(im,l,True)
show()
```

Этот код вычисляет SIFT-дескрипторы на плотной сетке, покрывающей все изображение, ориентируя их вдоль направления локального градиента (поскольку параметр *force_orientation* равен True). Места расположения дескрипторов показаны на рис. 8.2.

Классификация изображений – распознавание жестов

В этом приложении мы будем рассматривать применение плотных SIFT-дескрипторов к изображениям жестов руки с целью построения простой системы распознавания жестов. Для иллюстрации воспользуемся изображениями из базы данных статических положений руки (по адресу *http://www.idiap.ch/resource/gestures/*). Скачайте малый

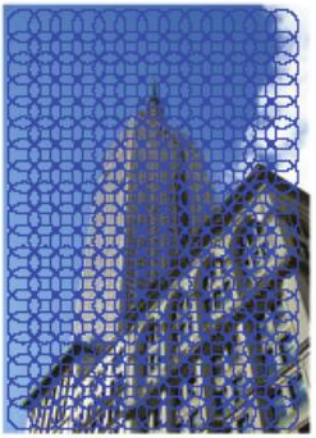


Рис. 8.2. Пример применения SIFT-дескрипторов на плотной сетке, покрывающей все изображение

тестовый набор (на веб-странице ссылка на него называется «test set 16.3Mb»), найдите все папки с именем «uniform». Изображения, находящиеся в каждой из них, поровну разложите по папкам «train» и «test» (предварительно создав их).

Обработаем изображения приведенной выше функцией, чтобы получить векторы признаков. В предположении, что имена файлов хранятся в списке *imlist*, это делается следующим образом:

```
import dsift

# обработать изображения, приведенные к фиксированному размеру (50,50)
for filename in imlist:
  featfile = filename[:-3]+'dsift'
  dsift.process_image_dsift(filename,featfile,10,5,resize=(50,50))
```

В результате для каждого изображения создается файл его признаков с расширением «.dsift». *Обратите внимание на приведение всех изображений к единому размеру.* Это очень важно, иначе количество дескрипторов в изображениях, а значит, и длины векторов признаков были бы различны, что привело бы впоследствии к ошибкам при их сравнении. Нанесенные на график изображения с дескрипторами показаны на рис. 8.3.

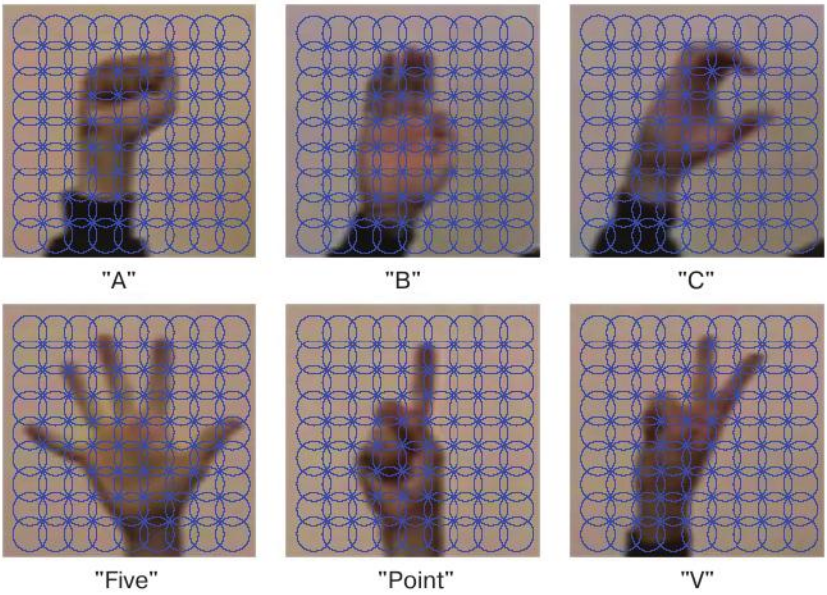


Рис. 8.3. Плотные SIFT-дескрипторы для изображений из шести категорий жестов руки (взяты из базы данных Static Hand Posture Database)

Определим вспомогательную функцию для чтения файлов плотных SIFT-дескрипторов:

```
import os, sift

def read_gesture_features_labels(path):
  # создать список всех файлов с расширением .dsift
  featlist = [os.path.join(path,f) for f in os.listdir(path)
          if f.endswith('.dsift')]

  # прочитать признаки
  features = []
  for featfile in featlist:
     l,d = sift.read_features_from_file(featfile)
     features.append(d.flatten())
  features = array(features)

  # создать метки
  labels = [featfile.split('/')[-1][0] for featfile in featlist]

  return features,array(labels)
```

Теперь можно прочитать признаки и метки для тестового и обучающего набора:

```
features,labels = read_gesture_features_labels('train/')

test_features,test_labels=read_gesture_features_labels('test/')

classnames = unique(labels)
```

Здесь мы создаем метки классов, исходя из первой буквы имени файла. А с помощью функции `NumPy unique()` получаем отсортированный список уникальных имен классов.

Настало время опробовать наш код классификации по ближайшим соседям на этих данных:

```
# протестировать kNN
k = 1
knn_classifier = knn.KnnClassifier(labels,features)
res = array([knn_classifier.classify(test_features[i],k) for i in
     range(len(test_labels))])

# верность
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Верность:', acc
```

Сначала создаем объект классификатора, передавая ему обучающие данные и метки. Затем обходим тестовый набор и классифицируем каждое изображение, вызывая метод `classify()`. Верность классификации вычисляется путем умножения булевого массива на единицу и суммирования. В данном случае истинные значения равны 1, поэтому подсчитать количество правильно назначенных классов нетрудно. Получается такой результат:

```
Верность: 0.811518324607
```

т. е. 81 % примеров классифицировано правильно. Значение зависит от выбора k и параметров алгоритма вычисления плотных SIFT-дескрипторов.

Полученное выше значение верности показывает, сколько примеров из тестового набора было классифицировано правильно, но ничего не говорит ни о том, какие жесты оказалось трудно классифицировать, ни о типичных ошибках. *Матрица неточностей* содержит информацию о том, как именно были классифицированы примеры из каждого класса. Она показывает, как распределились ошибки и какие классы классификатор чаще всего «путает» с другими.

Следующая функция печатает метки и матрицу неточностей:

```
def print_confusion(res,labels,classnames):

  n = len(classnames)

  # матрица неточностей
  class_ind = dict([(classnames[i],i) for i in range(n)])

  confuse = zeros((n,n))
  for i in range(len(test_labels)):
     confuse[class_ind[res[i]],class_ind[test_labels[i]]] += 1

  print 'Матрица неточностей для'
  print classnames
  print confuse
```

В результате вызова

```
print_confusion(res,test_labels,classnames)
```

будет напечатано:

```
Матрица неточностей для
['A' 'B' 'C' 'F' 'P' 'V']
[[ 26.  0.  2.  0.  1.  1.]
 [  0. 26.  0.  1.  1.  1.]
 [  0.  0. 25.  0.  0.  1.]
 [  0.  3.  0. 37.  0.  0.]
 [  0.  1.  2.  0. 17.  1.]
 [  3.  1.  3.  0. 14. 24.]]
```

Отсюда видно, к примеру, что «P» («Point» – указание) часто неправильно классифицируется как «V».

8.2. Байесовский классификатор

Еще один пример простого, но эффективного классификатора дает *байесовский классификатор*[3] (или *наивный байесовский классификатор*). Это вероятностный классификатор, основанный на теореме Байеса об условных вероятностях. Предполагается, что все признаки независимы и не связаны между собой (отсюда и слово «наивный»). Байесовский классификатор очень эффективно обучается, поскольку выбранная модель применяется независимо к каждому признаку. Несмотря на упрощающие предположения, байесовские классифика-

[3] В честь Томаса Байеса, английского математика и священника XVIII века.

торы весьма успешно применяются на практике, особенно для фильтрации почтового спама. Дополнительное преимущество состоит в том, что после того как модель обучена, обучающие данные можно не хранить. Нужны только параметры модели.

Для построения классификатора условные вероятности отдельных признаков перемножаются, и результат считается полной вероятностью класса. Затем выбирается класс с наибольшей вероятностью.

Рассмотрим простую реализацию байесовского классификатора с использованием модели нормального распределения вероятностей. Это означает, что среднее и дисперсия распределения каждого признака независимо вычисляются по обучающему набору данных. Добавьте следующий класс в файл *bayes.py*:

```
class BayesClassifier(object):

  def __init__(self):
     """ Инициализировать классификатор обучающими данными. """

     self.labels = [] # метки классов
     self.mean = []   # средние классов
     self.var = []    # дисперсии классов
     self.n = 0       # число классов

  def train(self,data,labels=None):
     """ Обучить на данных (список массивов n*dim).
       Параметр labels необязателен, по умолчанию 0...n-1. """

     if labels==None:
        labels = range(len(data))
     self.labels = labels
     self.n = len(labels)
     for c in data:
        self.mean.append(mean(c,axis=0))
        self.var.append(var(c,axis=0))

  def classify(self,points):
     """ Классифицировать точки - вычислить вероятности
       каждого класса и вернуть метку самого вероятного класса. """

     # вычислить вероятности каждого класса
     est_prob = array([gauss(m,v,points)
                       for m,v in zip(self.mean,self.var)])

     # получить индекс наибольшей вероятности, она определяет метку класса
     ndx = est_prob.argmax(axis=0)
     est_labels = array([self.labels[n] for n in ndx])

     return est_labels, est_prob
```

В этой модели класс описывается двумя переменными: средним и дисперсией. Метод `train()` принимает списки массивов признаков (по одному на каждый класс) и вычисляет среднее и дисперсию. Метод `classify()` вычисляет вероятности классов для массива данных и выбирает класс с наибольшей вероятностью. Возвращаются метки и вероятности предсказанных классов. Понадобится также вспомогательная функция для вычисления нормального распределения.

```
def gauss(m,v,x):
  """ Вычислить d-мерное нормальное распределение со средним m
     и дисперсией v в точках (строках) x. """
  if len(x.shape)==1:
     n,d = 1,x.shape[0]
  else:
     n,d = x.shape

  # ковариационная матрица, вычесть среднее
  S = diag(1/v)
  x = x-m

  # произведение вероятностей
  y = exp(-0.5*diag(dot(x,dot(S,x.T))))

  # нормировать и вернуть
  return y * (2*pi)**(-d/2.0) / ( sqrt(prod(v)) + 1e-6)
```

Эта функция вычисляет произведение отдельных нормальных распределений и возвращает вероятность для заданной пары параметров модели *m*, *v*. Дополнительные сведения об этой функции см., например, в статье *https://ru.wikipedia.org/wiki/Многомерное_нормальное_распределение*.

Опробуем этот байесовский классификатор на двумерных данных из предыдущего раздела. Приведенный ниже скрипт загружает те же самые наборы данных и обучает классификатор:

```
import pickle
import bayes
import imtools

# загрузить обучающие данные из pickle-файлов
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# обучить байесовский классификатор
bc = bayes.BayesClassifier()
bc.train([class_1,class_2],[1,-1])
```

Теперь можно загрузить другой набор данных и протестировать классификатор:

```
# загрузить тестовые данные из pickle-файлов
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# протестировать на нескольких точках
print bc.classify(class_1[:10])[0]

# нанести на график точки и решающую границу
def classify(x,y,bc=bc):
  points = vstack((x,y))
  return bc.classify(points.T)[0]

imtools.plot_2D_boundary([-6,6,-6,6],[class_1,class_2],classify,[1,-1])
show()
```

Будут напечатаны результаты классификации для первых 10 точек. Выглядеть это может так:

```
[1 1 1 1 1 1 1 1 1 1]
```

Как и раньше, мы воспользовались вспомогательной функцией `classify()`, которую передаем функции построения графика для визуализации результатов классификации на сетке. Графики для обоих наборов данных показаны на рис. 8.4. В данном случае решающие границы – эллипсообразные линии уровня двумерной гауссовой функции.

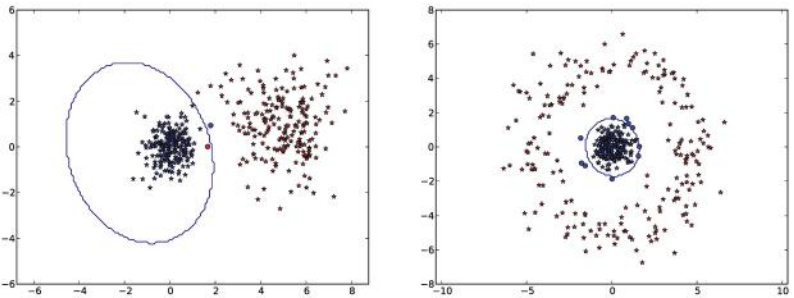


Рис. 8.4. Байесовская классификация двумерных данных. Для каждого примера цветом обозначается метка класса. Правильно классифицированные точки показаны звездочками, неправильно классифицированные – кружочками. Кривая линия – решающая граница классификатора

Использование метода главных компонент для понижения размерности

Теперь попробуем решить задачу о распознавании жестов. Поскольку векторы признаков для плотных SIFT-дескрипторов очень велики (в примере выше более 10 000 параметров), то будет разумно сначала понизить размерность и только потом обучать модель. Обычно с этим неплохо справляется метод главных компонент (PCA, см. раздел 1.3). Выполните следующий скрипт, в котором используется реализация PCF из файла *pca.py* (стр. 34):

```
import pca

V,S,m = pca.pca(features)

# оставить наиболее важные измерения
V = V[:50]
features = array([dot(V,f-m) for f in features])
test_features = array([dot(V,f-m) for f in test_features])
```

Здесь *features* и *test_features* – те же массивы, что мы загружали для классификатора kNN. В данном случае мы применяем метод PCA к обучающим данным и оставляем 50 измерений с наибольшей дисперсией. Для этого вычитаем среднее *m* (вычисленное по обучающим данным) и умножаем на базисные векторы *V*. Такое же преобразование применяется к тестовым данным.

Обучим и протестируем байесовский классификатор:

```
# протестировать байесовский классификатор
bc = bayes.BayesClassifier()
blist = [features[where(labels==c)[0]] for c in classnames]

bc.train(blist,classnames)
res = bc.classify(test_features)[0]
```

Поскольку `BayesClassifier` принимает список массивов (по одному массиву для каждого класса), то перед тем как передавать данные методу `train()`, мы их преобразуем. Так как сейчас нам не нужны вероятности, мы возвращаем только найденные в результате классификации метки.

Вычисляем и печатаем верность:

```
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Верность:', acc
```

Получается такое значение:

```
Верность: 0.717277486911
```

Распечатанная матрица неточностей

```
print_confusion(res,test_labels,classnames)
```

выглядит так:

```
Матрица неточностей для
['A' 'B' 'C' 'F' 'P' 'V']
 [[ 20.  0.  0.  4. 0.  0.]
 [  0. 26.  1.  7.  2.  2.]
 [  1.  0. 27.  5.  1.  0.]
 [  0.  2.  0. 17.  0.  0.]
 [  0.  1.  0.  4. 22.  1.]
 [  8.  2.  4.  1.  8. 25.]]
```

Не так хорошо, как в случае классификатора kNN, но зато мы не должны хранить обучающие данные, нужны лишь параметры модели для каждого класса. Результат сильно зависит от выбора измерений после прогона PCA.

8.3. Метод опорных векторов

Метод опорных векторов (Support Vector Machines, SVM) – мощный современный алгоритм классификации, который во многих случаях дает отличные результаты. В простейшей форме SVM находит гиперплоскость (плоскость в многомерном пространстве), которая наилучшим образом разделяет классы. Решающая функция для вектора признаков $\mathbf{x}$ имеет вид

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b,$$

где $\mathbf{w}$ – вектор, нормальный к гиперплоскости, а b – постоянное смещение. Плоскость, на которой эта функция равна 0, идеально разделяет два класса, так что для одного значения функции положительны, а для другого отрицательны. Параметры $\mathbf{w}$ и b ищутся путем решения задачи оптимизации на обучающем наборе векторов признаков $\mathbf{x}_i$, снабженных метками $y_i \in \{-1, 1\}$, цель которой – максимизировать зазоры между классами и гиперплоскостью. Нормаль представляет собой линейную комбинацию некоторых обучающих векторов

$$\mathrm{w} = \sum_i \alpha_i y_i \mathrm{x}_i,$$

так что решающую функцию можно записать в виде:

$$f(\mathrm{x}) = \sum_i \alpha_i y_i \mathrm{x}_i \cdot \mathrm{x} - b.$$

Здесь i пробегает выборку обучающих векторов; выбранные векторы $\mathbf{x}_i$ называются *опорными векторами*, поскольку они определяют решающую границу.

Одна из сильных сторон метода SVM заключается в том, что путем использования *ядерных функций*, т. е. функций, отображающих векторы признаков в пространство другой (более высокой) размерности, можно эффективно решать нелинейные или очень трудные задачи классификации, сохраняя в то же время некоторый контроль над решающей функцией. Ядерные функции заменяют скалярное произведение $\mathbf{x}_i \cdot \mathbf{x}$ функцией $K(\mathbf{x}_i, \mathbf{x})$.

Приведем примеры наиболее употребительных ядерных функций:

- *линейная*, гиперплоскость в пространстве признаков $K(\mathbf{x}_i, \mathbf{x}) = \mathbf{x}_i \cdot \mathbf{x}$, это самый простой случай;
- *полиномиальная*, признаки отображаются с помощью полиномов степени d, $K(\mathbf{x}_i, \mathbf{x}) = (\gamma \mathbf{x}_i \cdot \mathbf{x} + r)^d$, $\gamma > 0$;
- *радиальная базисная функция*, экспоненциальная функция, обычно очень эффективна: $K(\mathbf{x}_i, \mathbf{x}) = e^{(-\gamma |\mathrm{x}_i - \mathrm{x}|^2)}$, $\gamma > 0$;
- *сигмоидная*, сглаженная альтернатива гиперплоскости, $K(\mathbf{x}_i, \mathbf{x}) = \tanh(\gamma \mathbf{x}_i \cdot \mathbf{x} + r)$.

Параметры ядра также определяются во время обучения. Для многоклассовых задач обычно обучают несколько SVM, так чтобы каждый отделял один класс от всех остальных (такие классификаторы еще называют «один против всех»). Дополнительные сведения о методе опорных векторов можно найти, например, в книге [9] и в сетевых руководствах, упомянутых на странице *http://www.support-vector.net/references.html*.

Использование библиотеки LibSVM

Мы воспользуемся одной из лучших и самых распространенных реализаций, LibSVM [7]. Эта библиотека имеет элегантный интерфейс к Python (а также ко многим другим языкам программирования). Инструкции по установке приведены в разделе А.4. Применим LibSVM к демонстрационному набору двумерных данных, чтобы посмотреть, как это делается. Скрипт загружает те же данные, что и

выше, и обучает SVM-классификатор, применяя радиальные базовые функции:

```
import pickle
from svmutil import *
import imtools

# загрузить обучающие данные из pickle-файлов
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# преобразовать в списки для libsvm
class_1 = map(list,class_1)
class_2 = map(list,class_2)
labels = list(labels)
samples = class_1+class_2 # конкатенировать два списка

# создать SVM-классификатор
prob = svm_problem(labels,samples)
param = svm_parameter('-t 2')

# обучить SVM-классификатор на данных
m = svm_train(prob,param)

# как прошло обучение?
res = svm_predict(labels,samples,m)
```

Загрузка набора данных производится так же, как и раньше, но на этот раз мы должны преобразовать массивы в списки, поскольку LibSVM не принимает объекты-массивы. Здесь мы с помощью встроенной в Python функции `map()` применяем функцию `list()` к каждому элементу. В следующей строке создается объект SVM-задачи и задаются его параметры. В результате обращения к функции `svm_train()` решается задача оптимизации и определяются параметры модели. Затем модель можно использовать для предсказания. Наконец, вызов `svm_predict()` классифицирует обучающие данные с помощью модели *m* и сообщает, насколько успешным оказалось обучение. Печатается такое сообщение:

```
Accuracy = 100% (400/400) (classification)
```

Это означает, что классификатор полностью разделяет обучающие данные и правильно классифицирует все 400 точек.

Обратите внимание, что при вызове функции обучения классификатора мы задали строку параметров. Они служат для выбора типа

ядра, степени и других настроек классификатора. Большая их часть выходит за рамки этой книги, но о параметрах «t» и «k» следует знать. Параметр «t» задает тип ядра и может принимать следующие значения:

Значение -t	Ядро
0	Линейное
1	Полиномиальное
2	Радиальное базисное (по умолчанию)
3	Сигмоидное

Параметр «k» определяет степень полинома (по умолчанию 3). Теперь загрузим другой набор точек и протестируем классификатор:

```
# загрузить тестовые данные из pickle-файлов
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# преобразовать в списки для libsvm
class_1 = map(list,class_1)
class_2 = map(list,class_2)

# определить функцию для построения графика
def predict(x,y,model=m):
  return array(svm_predict([0]*len(x),zip(x,y),model)[0])

# нанести на график решающую границу
imtools.plot_2D_boundary([-6,6,-6,6],[array(class_1),array(class_2)],
                         predict,[-1,1])
show()
```

И снова необходимо преобразовать массивы в списки для LibSVM. Как и раньше, определяем вспомогательную функцию `predict()` для нанесения на график решающей границы классификатора. Обратите внимание на использование списка нулей `[0]*len(x)` вместо списка меток на случай, если истинные метки недоступны. Можно взять любой список, лишь бы он был правильной длины. Точки из обоих наборов показаны на рис. 8.5.

И снова о распознавании жестов

К нашей многоклассовой задаче о распознавании жестов руки LibSVM применяется без труда. Наличие нескольких классов обрабатывается автоматически, так что нам остается только подготовить данные, чтобы вход и выход соответствовали требованиям LibSVM.

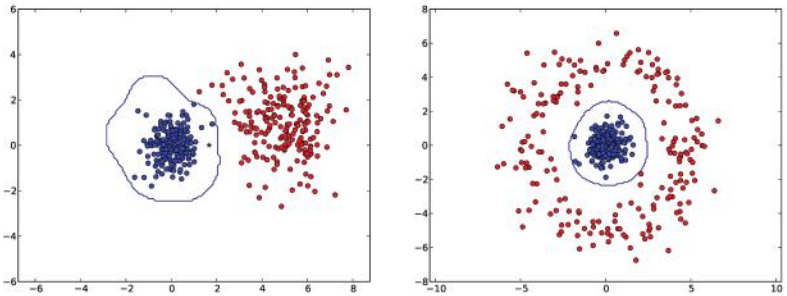


Рис. 8.5. Классификация точек на плоскости методом опорных векторов. Для каждого примера цветом обозначается метка класса. Правильно классифицированные точки показаны звездочками, неправильно классифицированные – кружочками. Кривая линия – решающая граница классификатора

В предположении, что обучающие и тестовые данные находятся в массивах *features* и *test_features*, как в предыдущих примерах, следующая программа загружает данные и обучает линейный SVM-классификатор:

```
features = map(list,features)
test_features = map(list,test_features)

# создать функцию преобразования для меток
transl = {}
for i,c in enumerate(classnames):
  transl[c],transl[i] = i,c

# создать SVM-классификатор
prob = svm_problem(convert_labels(labels,transl),features)
param = svm_parameter('-t 0')

# обучить SVM-классификатор на данных
m = svm_train(prob,param)

# как прошло обучение?
res = svm_predict(convert_labels(labels,transl),features,m)

# протестировать SVM-классификатор
res = svm_predict(convert_labels(test_labels,transl),test_features,m)[0]
res = convert_labels(res,transl)
```

Как и раньше, мы преобразуем данные в списке, вызывая функцию `map()`. Затем необходимо преобразовать метки, потому что LibSVM не умеет работать со строковыми метками. В словаре *transl* хранится

соответствие между строковыми и целочисленными метками. Попробуйте распечатать его на консоли и посмотрите, что получится. Параметр «-t 0» говорит, что классификатор должен быть линейным, т. е. решающая граница будет гиперплоскостью в исходном пространстве признаков с 10 000 измерений.

Теперь, как и раньше, сравним метки:

```
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Верность:', acc
print_confusion(res,test_labels,classnames)
```

При использовании такого линейного ядра получаются следующие результаты:

```
Верность: 0.916230366492
Матрица неточностей для
['A' 'B' 'C' 'F' 'P' 'V']
[[ 26.  0.  1.  0.  2.  0.]
 [  0. 28.  0.  0.  1.  0.]
 [  0.  0. 29.  0.  0.  0.]
 [  0.  2.  0. 38.  0.  0.]
 [  0.  1.  0.  0. 27.  1.]
 [  3.  0.  2.  0.  3. 27.]]
```

Если же, применив метод главных компонент, уменьшить число измерений до 50, как мы делали в разделе 8.2, то верность изменится:

```
Верность: 0.890052356021
```

Неплохо, если принять во внимание, что векторы признаков стали в 200 раз меньше, чем были (и пропорционально уменьшился объем памяти, необходимой для хранения опорных векторов).

8.4. Оптическое распознавание символов

В качестве примера многоклассовой задачи рассмотрим интерпретацию изображений игру судоку. *Оптическим распознаванием символов* (optical character recognition, OCR) называется процесс интерпретации изображений рукописного или машинного текста. Типичный пример – распознавание текста в отсканированных документах, например почтовых индексов на конвертах или страниц книги, как в библиотеке Google Books (*http://books.google.com/*). В этом разделе мы рассмотрим простую задачу распознавания чисел в напечатанных

изображениях судоку. Напомним, что судоку – это логическая головоломка, в которой требуется заполнить сетку 9×9 цифрами от 1 до 9, так чтобы в каждой строке, каждом столбце и каждом квадрате 3×3 встречались все девять цифр[4]. Нас будет интересовать только чтение и правильная интерпретация игрового поля. Решение же головоломки оставляем вам.

Обучение классификатора

В этой задаче классификации у нас есть десять классов – цифры от 1 до 9 и пустые клетки. Назначим пустым клеткам метку 0. Для обучения этого десятиклассового классификатора нам понадобится набор изображений с отдельными клетками судоку[5]. В файле *sudoku_images.zip* находятся две папки, «ocr_data» и «sudokus». Во второй из них вы найдете изображения судоку при различных условиях. Ими мы займемся позже. А пока обратимся к папке «ocr_data». Она содержит две подпапки с изображениями, по одной для обучения и тестирования. Первая буква в имени файла изображения совпадает с меткой класса (0 ... 9). На рис. 8.6 приведены примеры изображений из обучающего набора. Все они полутоновые размером примерно 80×80 пикселей (с небольшими вариациями).

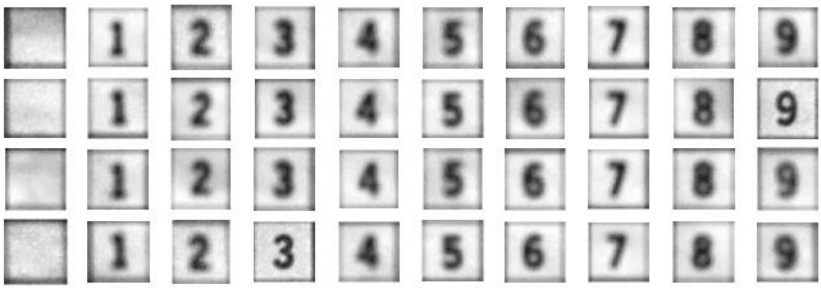


Рис. 8.6. Примеры обучающих изображений для десятиклассового классификатора судоку

Отбор признаков

Необходимо решить, какой вектор признаков использовать для представления изображений клеток. Хороших вариантов много, мы

[4] Если вы не знакомы с этой игрой, загляните на страницу *https://ru.wikipedia.org/wiki/судоку*.

[5] Изображения приводятся с любезного разрешения Мартина Бюрёда (Martin Byröd) [4], *http://www.maths.lth.se/matematiklth/personal/byrod/*, который собрал и разрезал фотографии реального поля судоку.

возьмем что-нибудь простое, но эффективное. Следующая функция принимает изображение и возвращает вектор признаков, содержащий линеаризованные полутоновые значения пикселей:

```
def compute_feature(im):
  """ Вернуть вектор признаков для OCR. """

  # изменить размер и убрать рамку
  norm_im = imresize(im,(30,30))
  norm_im = norm_im[3:-3,3:-3]

  return norm_im.flatten()
```

Эта функция вызывает функцию `imresize()` из пакета *imtools*, чтобы уменьшить размер и, следовательно, длину вектора признаков. Кроме того, мы обрезаем рамку – по 10 % пикселей с каждой стороны – потому что она часто содержит следы линий сетки (см. рис. 8.6).

Теперь можно прочитать обучающие данные:

```
def load_ocr_data(path):
  """ Вернуть метки и признаки OCR для всех изображений
     в указанной папке. """

  # создать список всех файлов с расширением .jpg
  imlist = [os.path.join(path,f) for f in os.listdir(path)
           if f.endswith('.jpg')]

  # создать метки
  labels = [int(imfile.split('/')[-1][0]) for imfile in imlist]

  # создать признаки по изображениям
  features = []
  for imname in imlist:
     im = array(Image.open(imname).convert('L'))
     features.append(compute_feature(im))
  return array(features),labels
```

Метками считаются первые символы имен JPEG-файлов, они сохраняются как целые числа в списке *labels*. Затем с помощью показанной выше функции вычисляются и сохраняются в массиве векторы признаков.

Многоклассовый SVM-классификатор

Подготовив обучающие данные, можно приступить к обучению классификатора. В данном случае мы воспользуемся методом опорных векторов. Код такой же, как в предыдущем разделе:

```
from svmutil import *

# ОБУЧАЮЩИЕ ДАННЫЕ
features,labels = load_ocr_data('training/')

# ТЕСТОВЫЕ ДАННЫЕ
test_features,test_labels = load_ocr_data('testing/')

# обучить линейный SVM-классификатор
features = map(list,features)
test_features = map(list,test_features)

prob = svm_problem(labels,features)
param = svm_parameter('-t 0')

m = svm_train(prob,param)

# как прошло обучение?
res = svm_predict(labels,features,m)

# а как на тестовом наборе?
res = svm_predict(test_labels,test_features,m)
```

Мы обучили линейный SVM-классификатор и протестировали его на не предъявлявшихся ранее изображениях из тестового набора. Последние два вызова функции `svm_predict()` печатают следующие сообщения:

```
Accuracy = 100% (1409/1409) (classification)
Accuracy = 99.2979% (990/997) (classification)
```

Великолепно! Все 1409 изображений обучающего набора идеально разделены на 10 классов, а качество распознавания на тестовом наборе составляет 99 %. Теперь можно использовать этот классификатор на вырезанных клетках новых изображений судоку.

Выделение клеток и распознавание символов

Имея классификатор, который распознает содержимое клетки, мы далее должны научиться автоматически находить клетки. Решив эту задачу, мы сможем вырезать их и передать классификатору. Предположим пока, что изображение судоку выровнено так, что горизонтальные и вертикальные линии сетки параллельны сторонам изображения (как слева на рис. 8.8). При таких условиях можно произвести бинаризацию изображения и просуммировать значения пикселей по

строкам и по столбцам. Поскольку значения пикселей на границах равны 1, а в остальных частях – нулю, то мы должны получить резкий всплеск на границах, который подскажет, как разрезать изображение.

Следующая функция принимает полутоновое изображение и направление и возвращает десять границ в этом направлении:

```
from scipy.ndimage import measurements

def find_sudoku_edges(im,axis=0):
  """ Найти границы клеток в выровненном изображении судоку. """

  # произвести бинаризацию и просуммировать по строкам и по столбцам
  trim = 1*(im<128)
  s = trim.sum(axis=axis)

  # найти центры наиболее темных линий
  s_labels,s_nbr = measurements.label(s>(0.5*max(s)))
  m = measurements.center_of_mass(s,s_labels,range(1,s_nbr+1))
  x = [int(x[0]) for x in m]

  # если обнаружены только темные линии, добавить еще и промежуточные
  if len(x)==4:
     dx = diff(x)
     x = [x[0],x[0]+dx[0]/3,x[0]+2*dx[0]/3,
          x[1],x[1]+dx[1]/3,x[1]+2*dx[1]/3,
          x[2],x[2]+dx[2]/3,x[2]+2*dx[2]/3,x[3]]

  if len(x)==10:
     return x
  else:
     raise RuntimeError('Границы не обнаружены.')
```

Сначала производим бинаризацию по средней точке, чтобы получить единицы в темных областях. Затем суммируем пиксели в заданном направлении (axis=0 или 1). В пакете `scipy.ndimage` имеется модуль `measurements`, очень полезный для подсчета и обмера областей в двоичных или меточных массивах. Сначала функция `labels()` находит связные компоненты двоичного массива, вычисленного в результате бинаризации. Затем функция `center_of_mass()` вычисляет среднюю точку каждой независимой компоненты. В зависимости от графического оформления судоку (все линии одинаково темные или границы клеток во внутренних квадратах светлее), мы можем получить четыре или десять линий. В первом случае добавляем промежуточные линии через равные интервалы. Если в конце получилось не десять линий, то возбуждается исключение.

В папке «sudokus» находится коллекция изображений судоку различной степени сложности. Для каждого изображения имеется также файл, содержащий правильные значения в клетках, чтобы можно было проверить полученные результаты. Границы некоторых судоку параллельны краям изображения. Выбрав любое из них, мы сможем проверить качество вырезания и классификации.

```
imname = 'sudokus/sudoku18.jpg'
vername = 'sudokus/sudoku18.sud'
im = array(Image.open(imname).convert('L'))

# найти границы клеток
x = find_sudoku_edges(im,axis=0)
y = find_sudoku_edges(im,axis=1)

# вырезать и классифицировать клетки
crops = []
for col in range(9):
  for row in range(9):
     crop = im[y[col]:y[col+1],x[row]:x[row+1]]
     crops.append(compute_feature(crop))

res = svm_predict(loadtxt(vername),map(list,crops),m)[0]
res_im = array(res).reshape(9,9)

print 'Результат:'
print res_im
```

Мы нашли границы и вырезали отдельные клетки. Эти клетки передаются той же функции выделения признаков, которая использовалась для обучения, и результаты сохраняются в массиве. Векторы признаков классифицируются функцией `svm_predict()`, которая загружает метки из файла с помощью `loadtxt()`. На консоли печатается результат.

```
Accuracy = 100% (81/81) (classification)
Результат:
[[ 0. 0. 0. 0. 1. 7. 0. 5. 0.]
 [ 9. 0. 3. 0. 0. 5. 2. 0. 7.]
 [ 0. 0. 0. 0. 0. 0. 4. 0. 0.]
 [ 0. 1. 6. 0. 0. 4. 0. 0. 2.]
 [ 0. 0. 0. 8. 0. 1. 0. 0. 0.]
 [ 8. 0. 0. 5. 0. 0. 6. 4. 0.]
 [ 0. 0. 9. 0. 0. 0. 0. 0. 0.]
 [ 7. 0. 2. 1. 0. 0. 8. 0. 9.]
 [ 0. 5. 0. 2. 3. 0. 0. 0. 0.]]
```

Это было простое изображение. Попробуйте другие и посмотрите, что происходит в случае ошибки и когда классификатор ошибается.

Если нанести вырезанные клетки на подграфик 9×9, то получится картинка, показанная на рис. 8.7.

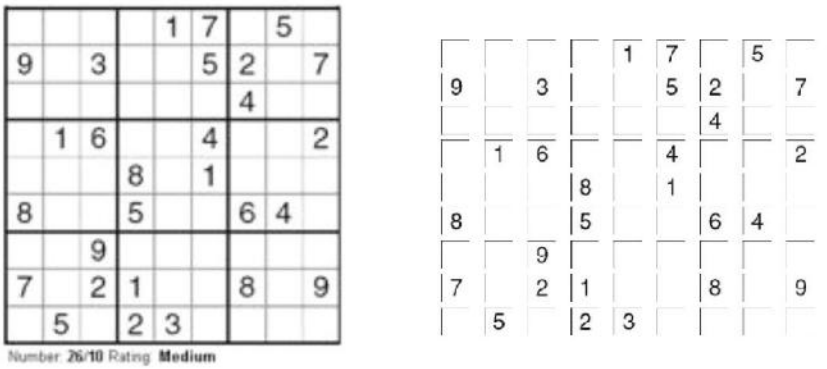


Рис. 8.7. Пример обнаружения и вырезания клеток сетки судоку: изображение игрового поля судоку (слева); вырезанные изображения отдельных клеток, передаваемые классификатору OCR (справа)

Выпрямление изображений

Если вы довольны качеством классификатора, то можно переходить к следующему этапу – применению к изображениям, расположенным под углом. В заключение мы покажем простой способ выпрямления изображения при условии, что четыре угловые точки были обнаружены автоматически или отмечены вручную. Слева на рис. 8.8 показан пример изображения судоку с заметным эффектом перспективы.

С помощью гомографии можно преобразовать сетку, так чтобы границы клеток стали параллельны границам изображения, нужно только вычислить подходящее преобразование. В примере ниже рассмотрен случай отметки четырех угловых точек вручную и последующей деформации в конечное квадратное изображение размером 1000×1000 пикселей:

```
from scipy import ndimage
import homography

imname = 'sudoku8.jpg'
im = array(Image.open(imname).convert('L'))

# отметить углы
figure()
imshow(im)
gray()
```

```
x = ginput(4)

# левый верхний, правый верхний, правый нижний, левый нижний
fp = array([array([p[1],p[0],1]) for p in x]).T
tp = array([[0,0,1],[0,1000,1],[1000,1000,1],[1000,0,1]]).T

# вычислить гомографию
H = homography.H_from_points(tp,fp)

# вспомогательная функция для геометрического преобразования
def warpfcn(x):
  x = array([x[0],x[1],1])
  xt = dot(H,x)
  xt = xt/xt[2]
  return xt[0],xt[1]

# деформировать изображение проективным преобразованием общего вида
im_g = ndimage.geometric_transform(im,warpfcn,(1000,1000))
```

Для большинства примеров аффинного преобразования, как в главе 3, недостаточно. Здесь мы использовали более общую функцию преобразования `geometric_transform()` из пакета `scipy.ndimage`. Она принимает не матрицу преобразования, а отображение плоскости в плоскость, так что необходима вспомогательная функция (использование аффинного деформирование по треугольникам в данном случае привело к образованию артефактов). Деформированное изображение показано на рис. 8.8 справа.

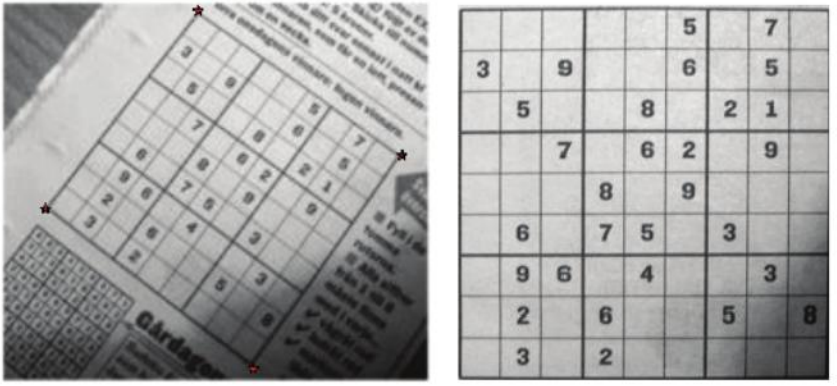


Рис. 8.8. Пример выпрямления изображения с применением проективного преобразования общего вида: исходное изображение судоку, на котором отмечены четыре угловых точки (слева); выпрямленное изображение, деформированное в квадрат размером 1000 × 1000 пикселей (справа)

На этом мы завершаем пример оптического распознавания судоку. Здесь открывается простор для усовершенствования и исследования альтернатив. Кое-что упомянуто в упражнениях ниже, остальное оставляем вам.

Упражнения

1. Качество kNN-классификатора зависит от значения k. Попробуйте поиграть с ним и посмотрите, как меняется верность. Нанесите на график решающие границы множеств точек на плоскости и понаблюдайте за их изменением.
2. Набор данных для распознавания жестов руки на рис. 8.3 содержит изображения с более сложным задним планом (в папке «complex/»). Попробуйте обучить и протестировать классификатор на этих изображениях. Как изменилось качество? Можете ли вы предложить улучшенный дескриптор изображения?
3. Попробуйте изменить количество измерений при проецировании на главные компоненты признаков распознавания жестов для байесовского классификатора. Какое число дает хорошие результаты? Нанесите на график сингулярные значения S, должна получиться типичная «коленообразная» кривая, как на рис. 8.9. Хороший компромисс между способностью справляться с изменчивостью обучающих данных и поддержанием относительно небольшого числа измерений обычно дает число на оси x непосредственно перед местом, начиная с которого кривая становится пологой.

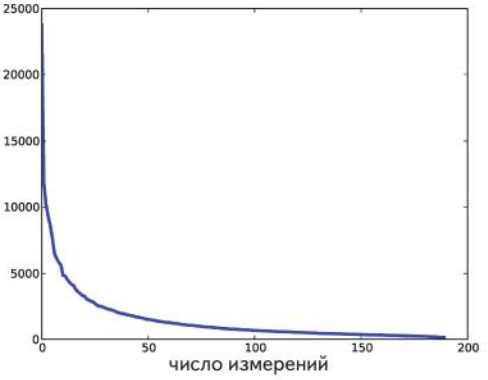


Рис. 8.9. График для упражнения 3

4. Модифицируйте байесовский классификатор, так чтобы использовалась модель распределения вероятностей, отличная от нормальной. Например, попробуйте взять счетчики частот каждого признака в обучающем наборе. Сравните результаты с получающимися при использовании нормального распределения для различных наборов данных.
5. Поэкспериментируйте с нелинейными SVM-классификаторами в задаче распознавания жестов. Попробуйте полиномиальные ядра, постепенно повышая степень полинома (с помощью параметра «-d»). Что происходит с качеством классификации на обучающем и на тестовом наборе? Нелинейный классификатор сопряжен с риском чрезмерной подгонки под обучающий набор, так что на нем качество классификации близко к идеальному, но на тестовом наборе резко снижается. Это явление, при котором теряется способность классификатора к обобщению, называется *переобучением*; его необходимо избегать.
6. Попробуйте более изощренные векторы признаков в задаче о распознавании символов в изображениях судоку. Если необходим источник вдохновения, почитайте работу [4].
7. Реализуйте метод автоматического выпрямления сетки судоку. Например, попробуйте обнаруживать признаки методом RANSAC, находить линии или клетки с помощью морфологических операций или измерений, имеющихся в пакете `scipy.ndimage` (*http://docs.scipy.org/doc/scipy/reference/ndimage.html*). Дополнительное задание: разрешите неоднозначность поворота при обнаружении направления «вверх». Например, можно попробовать повернуть выпрямленную сетку и дать классификатору OCR возможность проголосовать за оптимальную ориентацию путем вычисления верности.
8. Если вам интересна задача классификации посложнее судоку, ознакомьтесь с базой данных рукописных цифр MNIST по адресу *http://yann.lecun.com/exdb/mnist/*. Попробуйте выделить какие-нибудь признаки и применить к этому набору метод опорных векторов. Посмотрите, какое место достигнутое вами качество занимает на шкале лучших методов (некоторые из них дают потрясающие результаты).
9. Если хотите углубиться в проблемы классификации и алгоритмом машинного обучения, поинтересуйтесь пакетом `scikit.learn` (*http://scikit-learn.org/*) и примените некоторые реализованные в нем алгоритмы к данным из этой главы.

ГЛАВА 9. Сегментация изображений

Сегментацией называется разбиение изображения на значимые области. Это может быть передний и задний план изображения или отдельные объекты. При построении областей используются такие признаки, как цвета, границы или сходство соседей. В этой главе мы рассмотрим некоторые методы сегментации.

9.1. Разрезание графов

Графом называется множество вершин (иногда называемых также узлами) и соединяющих их ребер. На рис. 9.1 изображен пример графа[1]. Ребра могут быть направленными, и тогда граф называется *ориентированным*, или орграфом (на рис. 9.1 направление ребер показано стрелками), или не направленными. С каждым ребром может быть ассоциирован вес.

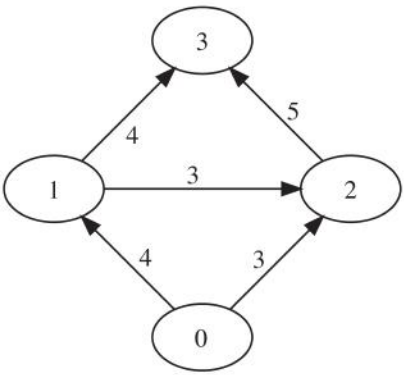


Рис. 9.1. Простой ориентированный граф, построенный с помощью пакета `python-graph`

[1] Мы уже встречались с графами в разделе 2.3. Но на этот раз воспользуемся ими для сегментации изображений.

Разрезанием графа называется разбиение ориентированного графа на непересекающиеся подмножества. Разрезания находят применение при решении различных задач компьютерного зрения, например стереореконструкции, сшивке и сегментации изображений. Если создать граф по пикселям изображения и их соседям и ввести понятие энергии или «стоимости», то граф можно будет использовать для сегментации изображения на несколько областей. Идея заключается в том, что похожие пиксели, расположенные близко друг к другу, должны принадлежать одному сегменту.

Стоимостью разреза графа C (где C – множество ребер) называется сумма весов ребер, принадлежащих разрезу

$$E_{cut} = \sum_{(i,j)\in C} w_{ij}, \qquad (9.1)$$

где w_{ij} – вес ребра (i, j), ведущего из вершины i в вершину j, а сумма вычисляется по всем ребрам, принадлежащим разрезу C.

Идея сегментации состоит в том, чтобы разрезать граф, представляющий изображение, так чтобы минимизировать стоимость разреза E_{cut}. Представляющий граф пополняется двумя вершинами, источником и стоком, и рассматриваются только разрезания, в которых источник и сток разделены.

Нахождение *минимального разреза* эквивалентно нахождению *максимального потока* между источником и стоком (подробнее см. [2]). Существуют эффективные алгоритмы для решения этих задач.

В наших примерах мы будем пользоваться пакетом `python-graph`, в который входит много полезных алгоритмов на графах. Скачать пакет и документацию можно с сайта *http://code.google.com/p/python-graph/*. Нам понадобится функция `maximum_flow()`, которая вычисляет максимальный поток (минимальный разрез) по алгоритму Эдмондса-Карпа (*https://ru.wikipedia.org/wiki/Алгоритм_Эдмондса_–_Карпа*). Достоинством пакета, написанного на чистом Python, является простота установки и совместимость, недостатком – низкое быстродействие. Для наших целей его производительности хватает, но для сколько-нибудь больших изображений необходима более быстрая реализация.

Ниже приведен пример использования пакета `python-graph` для вычисления максимального потока (минимального разреза) в небольшом графе[2].

[2] Этот граф приведен в качестве примера в статье *https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem*.

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow

gr = digraph()
gr.add_nodes([0,1,2,3])

gr.add_edge((0,1), wt=4)
gr.add_edge((1,2), wt=3)
gr.add_edge((2,3), wt=5)
gr.add_edge((0,2), wt=3)
gr.add_edge((1,3), wt=4)

flows,cuts = maximum_flow(gr,0,3)
print 'поток:', flows
print 'разрез:', cuts
```

Сначала создается граф с четырьмя вершинами, пронумерованными от 0 до 3. Затем с помощью метода `add_edge()` добавляются ребра с заданными весами. Вес определяет максимальную пропускную способность ребра. При вычислении максимального потока вершина 0 считается источником, а вершина 3 – стоком. В конце печатаются найденные поток и разрез:

```
поток: {(0, 1): 4, (1, 2): 0, (1, 3): 4, (2, 3): 3, (0, 2): 3}
разрез: {0: 0, 1: 1, 2: 1, 3: 1}
```

Эти словари содержат потоки, протекающие через каждое ребро, и метки вершин: 0 для части графа, содержащей исток, и 1 – для части, содержащей сток. Можете вручную проверить, что этот разрез действительно минимальный. Граф показан на рис. 9.1.

Графы изображений

При заданной структуре окрестностей можно определить граф, в котором вершинами будут пиксели изображения. Здесь мы ограничимся простейшим случаем 4-окрестностей пикселей и двумя областями изображения (которые будем называть передним и задним планом). *4-окрестностью* называется четверка пикселей слева, справа, выше и ниже данного[3].

Помимо вершин, соответствующих пикселям нам понадобятся две специальные вершины: «источник» и «сток», представляющие соответственно передний и задний план. Мы будем использовать простую модель, в которой все пиксели соединены с источником и стоком.

[3] Часто рассматриваются также 8-окрестности, в которые включены еще диагональные пиксели.

Граф строится следующим образом:

- для каждой вершины-пикселя существует входящее ребро из источника;
- для каждой вершины-пикселя существует исходящее ребро в сток;
- каждая вершина-пиксель соединена входящим и исходящим ребром с каждым своим соседом.

Для определения весов ребер нам понадобится модель сегментации, задающая веса (описывающие максимально допустимый поток по каждому ребру) ребер между пикселями и между пикселями и источником и стоком. Как и раньше, обозначим w_{ij} вес ребра между пикселями i и j, w_{si} – вес ребра от источника к пикселю i, а w_{it} – вес ребра от пикселя i к стоку.

Рассмотрим наивный байесовский классификатор по цветам пикселей, описанный в разделе 8.2. Предварительно обучив классификатор на пикселях переднего и заднего плана (того же или других изображений), мы сможем вычислить вероятности $p_F(I_i)$ и $p_B(I_i)$ принадлежности к переднему и заднему плану. Здесь I_i – вектор цветов пикселя i.

Теперь можно создать модель весов ребер:

$$w_{si} = \frac{p_F(I_i)}{p_F(I_i)+p_B(I_i)},$$

$$w_{it} = \frac{p_B(I_i)}{p_F(I_i)+p_B(I_i)},$$

$$w_{ij} = \kappa e^{-|I_i - I_j|^2/\sigma|}.$$

В этой модели каждый пиксель соединен с передним и задним планом (источником и стоком), а веса равны нормированной вероятности принадлежности к классу. Величина w_{ij} описывает сходство соседних пикселей; у похожих пикселей вес близок к κ, у непохожих – к нулю. Параметр σ определяет скорость приближения значения к нулю при увеличении несходства.

Создайте файл *graphcut.py* и добавьте в него следующую функцию, которая создает граф изображения:

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow
import bayes

def build_bayes_graph(im,labels,sigma=1e2,kappa=2):
```

```
""" Построить граф по 4-окрестностям пикселей.
  Передний и задний план определяются по меткам
  (1 для переднего плана, -1 для заднего, 0 в остальных
  случаях), в качестве модели используется наивный байесовский
  классификатор. """

m,n = im.shape[:2]

# Вектор RGB (один пиксель на строку)
vim = im.reshape((-1,3))

# RGB переднего и заднего плана
foreground = im[labels==1].reshape((-1,3))
background = im[labels==-1].reshape((-1,3))
train_data = [foreground,background]

# обучить наивный байесовский классификатор
bc = bayes.BayesClassifier()
bc.train(train_data)

# получить вероятности для всех пикселей
bc_lables,prob = bc.classify(vim)
prob_fg = prob[0]
prob_bg = prob[1]

# создать граф, имеющий m*n+2 вершин
gr = digraph()
gr.add_nodes(range(m*n+2))
source = m*n # предпоследняя вершина - источник
sink = m*n+1 # последняя - сток

# нормировать
for i in range(vim.shape[0]):
   vim[i] = vim[i] / linalg.norm(vim[i])

# обойти все вершины и добавить ребра
for i in range(m*n):
   # добавить ребро, исходящее из источника
   gr.add_edge((source,i), wt=(prob_fg[i]/(prob_fg[i]+prob_bg[i])))

   # добавить ребро, входящее в сток
   gr.add_edge((i,sink), wt=(prob_bg[i]/(prob_fg[i]+prob_bg[i])))

   # добавить ребра, соединяющие соседей
   if i%n != 0: # левый сосед существует
      edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-1])**2)/sigma)
      gr.add_edge((i,i-1), wt=edge_wt)
   if (i+1)%n != 0: # правый сосед существует
      edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+1])**2)/sigma)
      gr.add_edge((i,i+1), wt=edge_wt)
```

```
        if i//n != 0: # верхний сосед существует
            edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-n])**2)/sigma)
            gr.add_edge((i,i-n), wt=edge_wt)
        if i//n != m-1: # нижний сосед существует
            edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+n])**2)/sigma)
            gr.add_edge((i,i+n), wt=edge_wt)

    return gr
```

Здесь мы использовали в качестве обучающих данных меточное изображение, в котором значение 1 соответствует переднему плану, а –1 – заднему плану. При таком соглашении о метках байесовский классификатор обучается на значениях RGB. Для каждого пикселя вычисляются вероятности принадлежности к классам. Они становятся весами ребер, ведущих из источника и в сток. Создается граф, содержащий $n * m + 2$ вершин. Обратите внимание, что источнику и стоку назначены соответственно предпоследний и последний индекс, потому что так упрощается индексация пикселей.

Для визуализации меток, наложенных на изображение, можно воспользоваться функцией `contourf()`, которая заполняет области между изолиниями изображения (в данном случае меточного изображения). Параметр *alpha* задает прозрачность. Добавьте следующую функцию в файл *graphcut.py*:

```
def show_labeling(im,labels):
    """ Показать изображение с областями переднего и заднего плана.
        labels = 1 для переднего плана, -1 для заднего, 0 для прочих точек. """

    imshow(im)
    contour(labels,[-0.5,0.5])
    contourf(labels,[-1,-0.5],colors='b',alpha=0.25)
    contourf(labels,[0.5,1],colors='r',alpha=0.25)
    axis('off')
```

После того как граф построен, необходимо найти его оптимальное разрезание. Следующая функция вычисляет минимальный разрез и переформатирует результат в бинарное изображение, составленное из меток пикселей.

```
def cut_graph(gr,imsize):
    """ Найти максимальный поток графа gr и вернуть бинарное
        изображение, составленное из меток пикселей получившейся
        сегментации. """

    m,n = imsize
    source = m*n # предпоследняя вершина - источник
```

```
    sink = m*n+1 # последняя вершина - сток

    # разрезать граф
    flows,cuts = maximum_flow(gr,source,sink)

    # преобразовать граф в меточное изображение
    res = zeros(m*n)
    for pos,label in cuts.items()[:-2]: # не добавлять источник и сток
        res[pos] = label

    return res.reshape((m,n))
```

Снова обращаем внимание на индексы источника и стока. Нам нужно знать размер изображения для их вычисления и переформатирования результата перед возвратом сегментации. Разрез возвращается в виде словаря, который нужно скопировать в изображение, содержащее метки сегментации. Это делает метод `.items()`, который возвращает список пар (ключ, значение). Последние два элемента списка пропускаются.

Посмотрим, как использовать эти функции для сегментации изображения. Ниже приведен полный пример чтения изображения и создания графа с оценками вероятностей принадлежности к классам, вычисленными по двум прямоугольным областям.

```
from scipy.misc import imresize
import graphcut

im = array(Image.open('empire.jpg'))
im = imresize(im,0.07,interp='bilinear')
size = im.shape[:2]

# добавить две прямоугольные обучающие области
labels = zeros(size)
labels[3:18,3:18] = -1
labels[-18:-3,-18:-3] = 1

# создать граф
g = graphcut.build_bayes_graph(im,labels,kappa=1)

# разрезать граф
res = graphcut.cut_graph(g,size)

figure()
graphcut.show_labeling(im,labels)

figure()
imshow(res)
gray()
```

```
axis('off')

show()
```

Мы воспользовались функций `imresize()`, чтобы уменьшить изображение до размера, с которым может работать наша графическая библиотека, написанная на Python, – в данном случае до 7 % оригинала. Граф разрезается, и результат наносится на график вместе с изображением обучающих областей. На рис. 9.2 показаны обучающие области, наложенные на изображение, и окончательный результат сегментации.

Рис. 9.2. Пример сегментации графа с применением байесовской вероятностной модели. Изображение уменьшено до размера 54×38. Меточное изображение для обучения модели (слева); обучающие области поверх изображения (в центре); сегментация (справа)

Переменная *kappa* (символ κ в формулах) определяет относительный вес ребер между соседними пикселями. На рис. 9.3 можно наблюдать результат изменения *kappa*. По мере увеличения граница сегментации сглаживается, но детали теряются. Выбор оптимального значения оставлен на усмотрение пользователя. Он зависит от характера приложения и желаемого результата.

Сегментация с привлечением пользователя

Сегментацию методов разрезания графа можно различными способами сочетать с интерактивным взаимодействием с пользователем.

Например, пользователь может поставить на изображении маркеры переднего и заднего плана. Или выбрать область, содержащую передний план, с помощью ограничивающего прямоугольника или инструмента «лассо».

Рис. 9.3. Влияние изменения относительного веса ребер между соседними пикселями на вероятности классов. Одна и та же сегментация при (a) к = 1, (b) к = 2, (c) к = 5, (d) к = 10

В качестве последнего примера рассмотрим несколько изображений из набора Grab Cut, созданного научно-исследовательским центром Microsoft Research в Кембридже; подробности см. в [27] и приложении Б.5.

Эти изображения сопровождаются контрольными метками для проверки качества сегментации. Они также содержат аннотации, имитирующие поведение пользователя, который выбирает прямоугольную область изображения или с помощью инструмента «лассо» отмечает области переднего и заднего плана. Мы можем использовать сообщаемую пользователем информацию для получения обучающих данных и разрезания графа с целью сегментации изображения.

Информация, полученная от пользователя, закодирована в растровых изображениях со следующими значениями пикселей:

Значение пикселя	Интерпретация
0,64	задний план
128	неизвестно
255	передний план

Ниже приведен полный код загрузки изображения и аннотаций и их передачи нашей функции сегментации путем разрезания графа.

```
from scipy.misc import imresize
import graphcut

def create_msr_labels(m,lasso=False):
  """ Создать меточную матрицу для обучения на
     пользовательских аннотациях. """

  labels = zeros(im.shape[:2])

  # задний план
  labels[m==0] = -1
  labels[m==64] = -1

  # передний план
  if lasso:
     labels[m==255] = 1
  else:
     labels[m==128] = 1

  return labels

# загрузить изображение и аннотацию
im = array(Image.open('376043.jpg'))
m = array(Image.open('376043.bmp'))

# изменить размер
scale = 0.1
im = imresize(im,scale,interp='bilinear')
m = imresize(m,scale,interp='nearest')

# создать обучающие метки
labels = create_msr_labels(m,False)

# построить граф с помощью аннотаций
g = graphcut.build_bayes_graph(im,labels,kappa=2)

# разрезать граф
res = graphcut.cut_graph(g,im.shape[:2])

# удалить части, принадлежащие заднему плану
res[m==0] = 1
res[m==64] = 1

# нанести результат на график
figure()
imshow(res)
gray()
xticks([])
yticks([])
savefig('labelplot.pdf')
```

Прежде всего, определим вспомогательную функцию для чтения и форматирования аннотирующих изображений, так чтобы их можно было передать нашей функции обучения моделей переднего и заднего плана. Ограничивающие прямоугольники содержат только метки заднего плана. В данном случае мы устанавливаем обучающую область переднего плана, так чтобы она совпадала со всей «неизвестной» областью (внутренностью прямоугольника). Затем строим и разрезаем граф. Поскольку у нас есть информация от пользователя, удаляем результаты, в которых пиксели переднего плана встречаются в отмеченной области заднего плана. В самом конце наносим результат сегментации на график и стираем риски на осях, присваивая им пустые списки. Тем самым мы получаем аккуратный ограничивающий прямоугольник (иначе границы изображения было бы трудно различить на черно-белом рисунке).

На рис. 9.4 показаны некоторые результаты, когда в качестве признака использовался вектор RGB: оригинальное изображение, уменьшенная маска и уменьшенный результат сегментации.

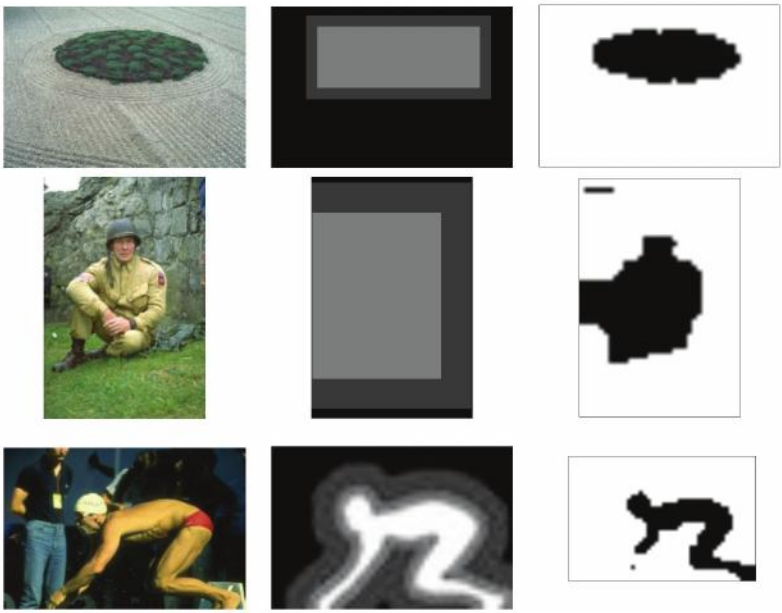


Рис. 9.4. Результаты сегментации с помощью разрезания графа для изображений из набора Grab Cut: уменьшенное исходное изображение (слева); маска, использованная для обучения (в центре); результат сегментации с применением значений RGB в качестве вектора признаков (справа)

9.2. Сегментация с применением кластеризации

Метод разрезания графа из предыдущего раздела находит решение задачи сегментации в результате поиска максимального потока (минимального разреза) графа изображения. В этом разделе мы рассмотрим другой способ разрезания графа изображения. Алгоритм нормализованного разреза, основанный на спектральной теории графов, для сегментации изображения комбинирует сходство пикселей с пространственной близостью.

Идея заключается в том, чтобы определить стоимость разреза, принимая во внимание размер частей. Формула нормализованного разреза выглядит так (где E_{cut} – стоимость разреза, определенная в формуле (9.1)):

$$E_{ncut} = \frac{E_{cut}}{\sum_{i \in A} w_{ix}} + \frac{E_{cut}}{\sum_{j \in B} w_{jx}},$$

Здесь A и B – две части разреза, по которым суммируются веса ребер, ведущих в другие вершины графа (которые в данном случае являются пикселями изображения). Эта сумма называется *ассоциацией*, и для изображений, пиксели которых имеют примерно одинаковое число связей с другими пикселями, она является грубой оценкой размера частей разреза. Эта функция стоимости была предложена в работе [32] вместе с алгоритмом ее минимизации. Описываемый ниже алгоритм формулируется для двухклассовой сегментации.

Определим W как матрицу реберных весов, в которой элемент w_{ij} содержит вес ребра между пикселями i и j. Обозначим D диагональную матрицу, содержащую суммы элементов W по строкам, $D = \mathrm{diag}(d_i)$, где $d_i = \sum_j w_{ij}$ (как в разделе 6.3). Сегментация на основе нормализованного разреза ищется как решение следующей задачи оптимизации:

$$\min_{\mathbf{y}} \frac{\mathbf{y}^T (D - W)\mathbf{y}}{\mathbf{y}^T D \mathbf{y}},$$

где вектор $\mathbf{y}$ содержит дискретные метки, удовлетворяющие ограничениям: $y_i \in \{1, -b\}$ для некоторой константы b (это означает, что $\mathbf{y}$ принимает только два дискретных значения) и сумма $\mathbf{y}^T D$ равна 0. Из-за этих ограничений задача оказывается трудной[4].

[4] На самом деле, NP-трудной.

Однако если ослабить ограничения и разрешить **y** принимать любые вещественные значения, то задача сводится к поиску собственных значений и легко решается. Недостаток в том, что результат придется подвергнуть бинаризации или кластеризации, чтобы получить дискретную сегментацию.

С ослабленными ограничениями задача состоит в нахождении собственных векторов матрицы Лапласа

$$L = D^{-1/2}WD^{-1/2},$$

как и в случае спектральной кластеризации. Единственная остающаяся сложность – определить веса w_{ij} ребер, соединяющих пиксели. Нормализованные разрезы имеют много общего со спектральной кластеризацией, и лежащие в их основе теории в чем-то пересекаются. Объяснения и детали см. в работе [32].

Воспользуемся весами ребер, предложенными в оригинальной статье [32] о нормализованных разрезах. Вес ребра, соединяющего пиксели i и j, определяется по формуле

$$w_{ij} = e^{-|Ii - Ij|^2/\sigma_g} e^{-|\mathbf{x}i - \mathbf{x}j|^2/\sigma_d}.$$

Первый сомножитель измеряет сходство между пикселями, где I_i и I_j обозначают либо RGB-векторы, либо полутоновые значения. Второй сомножитель измеряет близость пикселей в изображении, а $\mathbf{x}_i$ и $\mathbf{x}_j$ обозначают их координатные векторы. Масштабные коэффициенты σ_g и σ_d определяют относительный масштаб и скорость приближения элементов вектора к нулю.

Теперь рассмотрим код. Добавьте следующую функцию в файл *ncut.py*:

```
def ncut_graph_matrix(im,sigma_d=1e2,sigma_g=1e-2):
  """ Создать матрицу нормализованного разреза. Параметрами
      являются веса для расстояния между пикселями и сходства
      пикселей. """

  m,n = im.shape[:2]
  N = m*n

  # нормировать и создать вектор признаков: RGB или полутоновый
  if len(im.shape)==3:
     for i in range(3):
        im[:,:,i] = im[:,:,i] / im[:,:,i].max()
     vim = im.reshape((-1,3))
  else:
     im = im / im.max()
```

```
    vim = im.flatten()

# координаты x,y для вычисления расстояния
xx,yy = meshgrid(range(n),range(m))
x,y = xx.flatten(),yy.flatten()

# создать матрицу реберных весов
W = zeros((N,N),'f')
for i in range(N):
    for j in range(i,N):
        d = (x[i]-x[j])**2 + (y[i]-y[j])**2
        W[i,j] = W[j,i] = exp(-1.0*sum((vim[i]-vim[j])**2)/sigma_g) *
                          exp(-d/sigma_d)

return W
```

Эта функция принимает массив, представляющий изображение, и создает вектор признаков, используя RGB или полутоновые значения в зависимости от вида изображения. Поскольку вес ребра включает компонент, зависящий от расстояния, мы используем функцию `meshgrid()`, чтобы получить значения координат x и y пикселей. Затем в цикле обходим все N пикселей и заполняем матрицу нормализованного разреза W размерности $N \times N$.

Для вычисления сегментации можно либо последовательно разрезать каждый собственный вектор, либо взять сколько-то собственных векторов и применить кластеризацию. Мы выбрали второй подход, потому что он работает безо всяких изменений для любого числа сегментов. Мы берем первые *ndim* собственных векторов матрицы Лапласа, соответствующей W, и кластеризуем пиксели.

Следующая функция реализует кластеризацию. Как видите, она мало отличается от функции спектральной кластеризации из раздела 6.3:

```
from scipy.cluster.vq import *

def cluster(S,k,ndim):
  """ Спектральная кластеризация по матрице сходства. """

  # проверить симметричность
  if sum(abs(S-S.T)) > 1e-10:
     print 'не симметрична'

  # создать матрицу Лапласа
  rowsum = sum(abs(S),axis=0)
  D = diag(1 / sqrt(rowsum + 1e-6))
  L = dot(D,dot(S,D))

  # вычислить собственные векторы L
```

```
    U,sigma,V = linalg.svd(L)

    # создать вектор признаков из первых ndim собственных векторов,
    # рассматриваемых как столбцы
    features = array(V[:ndim]).T

    # кластеризация методом K средних
    features = whiten(features)
    centroids,distortion = kmeans(features,k)
    code,distance = vq(features,centroids)

    return code,V
```

Здесь мы воспользовались кластеризацией методом *K* средних (см. раздел 6.1), чтобы сгруппировать пиксели на основе изображения, построенного из собственных векторов. Если хотите поэкспериментировать, попробуйте другой алгоритм кластеризации или другой критерий группировки.

Теперь мы готовы применить описанный метод к нескольким изображениям. Ниже приведен полный код.

```
import ncut
from scipy.misc import imresize

im = array(Image.open('C-uniform03.ppm'))
m,n = im.shape[:2]

# привести к размеру (wid,wid)
wid = 50
rim = imresize(im,(wid,wid),interp='bilinear')
rim = array(rim,'f')

# создать матрицу нормализованного разреза
A = ncut.ncut_graph_matrix(rim,sigma_d=1,sigma_g=1e-2)

# кластеризовать
code,V = ncut.cluster(A,k=3,ndim=3)

# восстановить исходный размер изображения
codeim = imresize(code.reshape(wid,wid),(m,n),interp='nearest')

# нанести результат на график
figure()
imshow(codeim)
gray()
show()
```

Здесь мы изменяем размер изображения (в данном примере 50×50), чтобы ускорить вычисление собственных векторов. Функция `linalg.`

`svd()` из пакета NumPy `linalg.svd()` недостаточно быстрая, поэтому с большими матрицами работать не может (и иногда дает на таких матрицах неточные результаты). При изменении размера исходного изображения мы использовали билинейную интерполяцию, а при изменении размера результирующего меточного изображения сегментации – интерполяцию по ближайшему соседу, потому что не хотим интерполировать метки классов. Обратите внимание, что сначала мы преобразовали одномерный массив в матрицу (*wid*, *wid*), а затем восстановили изображение до исходного размера.

В этом примере мы использовали одно из изображений жеста из базы данных статических положений руки (см. раздел 8.1) и взяли $k = 3$. Получившаяся сегментация показана на рис. 9.5 вместе с первыми четырьмя собственными векторами.

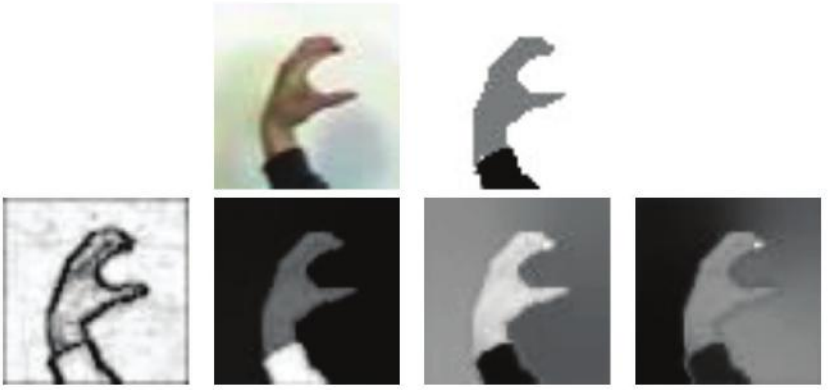


Рис. 9.5. Сегментация изображения с применением алгоритма нормализованного разреза: исходное изображение и результат сегментации с тремя классами (сверху); первые четыре собственных вектора матрицы графа сходства (снизу)

Собственные векторы возвращены в виде массива *V*, и визуализировать их можно следующим образом:

```
imshow(imresize(V[i].reshape(wid,wid),(m,n),interp='bilinear'))
```

В результате *i*-ый собственный вектор будет показан в виде изображения исходного размера.

На рис. 9.6 показаны дополнительные примеры применения этого скрипта. Изображение самолета взято из категории «airplane» в наборе данных Caltech 101. Для этих примеров мы оставили те же

значения параметров σ_d и σ_g, что и выше. Если их изменить, то получатся более плавные, регуляризованные изображения и совершенно другие изображения собственных векторов. Поэкспериментируйте сами.

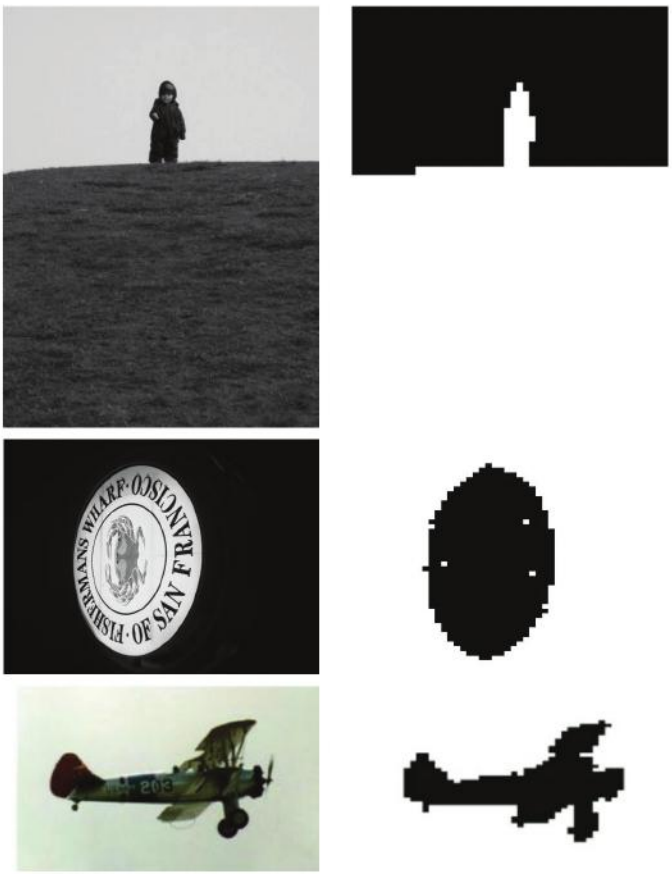


Рис. 9.6. Примеры сегментации с двумя классами с применением алгоритма нормализованного разреза: исходное изображение (слева); результат сегментации (справа)

Стоит отметить, что даже в этих сравнительно простых примерах бинаризация изображения дала бы другой результат, как и кластеризация RGB или полутоновых значений. Дело в том, что оба эти метода не учитывают соседних пикселей.

9.3. Вариационные методы

В этой книге мы встречали несколько примеров минимизации стоимости или энергии для решения задач компьютерного зрения. В предыдущих разделах мы занимались поиском минимального разреза графа, но аналогичные методы применялись для очистки от шумов по методу ROF, для кластеризации методом K средних и классификации методом опорных векторов. Все это примеры задач оптимизации.

Если оптимизация применяется в пространстве функций, то соответствующая задача называется *вариационной*, а алгоритмы ее решения – *вариационными методами*. Рассмотрим простую, но эффективную вариационную модель.

В алгоритме сегментации Чана-Везе [6] предполагается кусочно-постоянная модель областей сегментируемого изображения. Мы будем рассматривать только две области, скажем передний и задний план, но модель обобщается на любое число областей (см., например, [38]). Модель можно описать следующим образом.

Если рассмотреть набор кривых Γ, разделяющих изображение на две области Ω_1 и Ω_2, как на рис. 9.7, то сегментация в модели Чана-Везе определяется минимумом энергии:

$$E(\Gamma) = \lambda \,\text{length}(\Gamma) + \int_{\Omega_1} (I - c_1)^2 d\mathbf{x} + \int_{\Omega_2} (I - c_2)^2 d\mathbf{x},$$

которая измеряет отклонения от постоянных уровней яркости в каждой области, c_1 и c_2. Здесь интегралы берутся по каждой области, а включение суммарной длины разделяющих кривых нужно для нахождения более гладких решений.

Для кусочно-постоянного изображения $U = \chi_1 c_1 + \chi_2 c_2$ это выражение можно переписать в виде

$$E(\Gamma) = \lambda \frac{|c_1 - c_2|}{2} \int |\nabla U| d\mathbf{x} + \|I - U\|^2,$$

где χ_1 и χ_2 – характеристические (индикаторные) функций обеих областей[5]. Это преобразование нетривиально и требует солидных познаний в математике, далеко выходящих за рамки этой книги. Однако для понимания сути дела они не нужны.

А суть в том, что теперь это выражение имеет такой же вид, как выражение (1.1) в алгоритме ROF, только вместо λ стоит $\lambda|c_1 - c_2|$. Един-

[5] *Характеристическая функция* принимает значение 1 внутри области и 0 вне нее.

ственное различие заключается в том, что в алгоритме Чана-Везе мы ищем кусочно-постоянное изображение U. Можно показать, что бинаризация решения ROF дает хороший минимум. Интересующегося читателя отсылаем к работе [8].

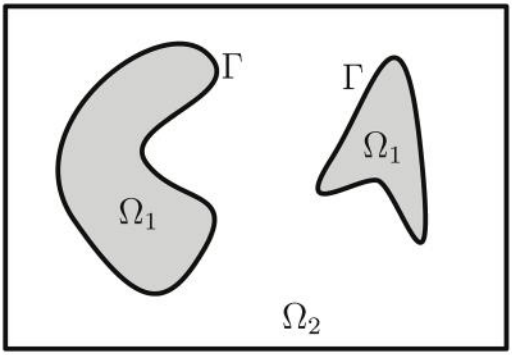


Рис. 9.7. Кусочно-постоянная модель сегментации Чана-Везе

Минимизация модели Чана-Везе теперь свелась к очистке от шумов по методу ROF с последующей бинаризацией:

```
import rof

im = array(Image.open('ceramic-houses_t0.png').convert("L"))
U,T = rof.denoise(im,im,tolerance=0.001)
t = 0.4 # бинаризация

import scipy.misc
scipy.misc.imsave('result.pdf',U < t*U.max())
```

В данном случае мы уменьшили пороговое значение прекращения итераций ROF, чтобы гарантировать достаточное число итераций. На рис. 9.8 показаны результаты для двух довольно трудных изображений.

Упражнения

1. Вычисление минимального разреза графа можно ускорить, уменьшив число ребер. Такая процедура построения графа описана в разделе 4.2 работы [16]. Выполните ее и посмотрите, как изменится размер графа и время сегментации по сравнению с простым построением, описанным в книге.

Рис. 9.8. Примеры сегментации изображений методом минимизации модели Чана-Везе с применением алгоритма ROF очистки от шумов: (a) исходное изображение; (b) изображение после очистки от шумов по методу ROF; (c) окончательная сегментация

2. Разработайте пользовательский интерфейс или имитируйте выбор пользователем областей для сегментации методом разрезания графа. Затем попробуйте «зашить» в код передний и задний план, присвоив весам некоторое большое значение.
3. Измените вектор признаков в алгоритме сегментации методом разрезания графа, взяв вместо RGB-вектора какой-нибудь другой дескриптор. Сможете ли вы улучшить результаты сегментации?
4. Реализуйте итеративный подход к сегментации методом разрезания графа, использовав текущую сегментацию как обучающие данные для следующей модели переднего и заднего плана. Улучшится ли при этом качество сегментации?
5. Набор данных Grab Cut научно-исследовательского центра Microsoft содержит контрольные данные для проверки сегментации. Реализуйте функцию, которая измеряет погрешность сегментации и оценивает различные комбинации параметров и идеи, высказанные в упражнениях.

6. Попробуйте другие параметры вычисления весов ребер нормализованного разреза и посмотрите, как они влияют на изображения собственных векторов и результаты сегментации.
7. Вычислите градиенты изображений первых собственных векторов нормализованного разреза. Комбинируя эти градиентные изображения, найдите контуры изображений объектов.
8. Реализуйте линейный поиск по значениям порога бинаризации для очищенного от шумов изображения в методе сегментации Чана-Везе. Для каждого порога сохраните энергию $E(\Gamma)$ и выберите сегментацию с наименьшим значением.

ГЛАВА 10. OpenCV

В этой главе дается краткий обзор работы с популярной библиотекой компьютерного зрения OpenCV через интерфейс с Python. Оригинальная версия библиотеки OpenCV была написана на C++ корпорацией Intel, а теперь сопровождается компанией Willow Garage. Она предназначена для компьютерного зрения в реальном масштабе времени. Исходный код OpenCV открыт и распространяется по лицензии BSD, подразумевающей бесплатность для использования в академических и коммерческих целях. Начиная с версии 2.0, ее поддержка в Python значительно улучшена. Мы рассмотрим несколько простых примеров и уделим больше внимания трассировке и видео.

10.1. Интерфейс между OpenCV и Python

OpenCV написана на C++ и содержит модули, охватывающие различные области компьютерного зрения. Но имеется и постоянно развивается поддержка интерфейса с Python как более простым скриптовым языком. Этот интерфейс продолжает разрабатываться – еще не все части OpenCV охвачены, и многие функции не документированы. Но в будущем положение, вероятно, изменится, т. к. за разработкой интерфейса стоит активное сообщество. Интерфейс с Python документирован на странице *http://opencv.willowgarage.com/documentation/python/index.html*. Инструкции по установке приведены в приложении А.

Текущая версия OpenCV (2.3.1) поставляется с двумя интерфейсами к Python. В старом модуле `cv` используются внутренние типы данных OpenCV, и работа с ними из `NumPy` может доставить немало хлопот. В новом модуле `cv2` используются массивы `NumPy`, так что он

интуитивно намного понятнее[1]. Для доступа к новому модулю нужно написать

```
import cv2
```

а к старому

```
import cv2.cv
```

В этой главе мы будем работать только с модулем `cv2`. Помните, что в будущих версиях имена модулей, функций и классов могут измениться. OpenCV и интерфейс между ней и Python активно разрабатываются.

10.2. Основы OpenCV

В библиотеке OpenCV имеются функции для чтения и записи изображений и операций над матрицами, а также математические библиотеки. Для изучения OpenCV (только на C++) имеется прекрасная книга [3]. Рассмотрим некоторые основные компоненты и порядок работы с ними.

Чтение и запись изображений

Следующая короткая программа загружает изображение, печатает его размер, преобразует и сохраняет в формате PNG:

```
import cv2

# прочитать изображение
im = cv2.imread('empire.jpg')
h,w = im.shape[:2]
print h,w

# сохранить изображение
cv2.imwrite('result.png',im)
```

Функция `imread()` возвращает изображение в виде стандартного массива `NumPy` и понимает различные графические форматы. Ей можно пользоваться вместо функций из библиотеки PIL. Функция `imwrite()` автоматически преобразует изображение в формат, соответствующий расширению выходного файла.

[1] Имена и места расположения этих модулей в будущем вполне могут измениться. Справляйтесь с документацией в сети.

Цветовые пространства

В OpenCV изображения хранятся не с применением традиционных цветовых каналов RGB, а в обратном порядке BGR. При чтении изображения по умолчанию подразумевается порядок BGR, но предоставляются необходимые конвертеры. Для преобразования цветового пространства служит функция `cvtColor()`. Например, преобразование в полутоновый формат выполняется так:

```
im = cv2.imread('empire.jpg')
# создать полутоновое изображение
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
```

Первый параметр функции – исходное изображение, второй – код преобразования цветов, определенный в OpenCV. Приведем несколько наиболее употребительных кодов:

- `cv2.COLOR_BGR2GRAY`
- `cv2.COLOR_BGR2RGB`
- `cv2.COLOR_GRAY2BGR`

В каждом случае число цветовых каналов в результирующем изображении соответствует коду преобразования (один канал для полутонового и три для RGB и BGR). Последний код предназначен для преобразования полутоновых изображений в формат BGR и полезен, когда требуется нанести изображение на график или наложить на него цветные объекты. Мы еще воспользуемся им в примерах ниже.

Отображение изображений и результатов обработки

Рассмотрим несколько примеров использования OpenCV для обработки изображений и представления результатов с помощью средств построения графиков и управления окнами.

В первом примере мы читаем изображение из файла и создаем его интегральное представление:

```
import cv2

# прочитать изображение
im = cv2.imread('fisherman.jpg')
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

# вычислить интегральное изображение
```

```
intim = cv2.integral(gray)

# нормировать и сохранить
intim = (255.0*intim) / intim.max()
cv2.imwrite('result.jpg',intim)
```

После того как изображение прочитано и преобразовано в полутоновое, функция `integral()` создает изображение, в котором значение каждого пикселя равно сумме яркостей пикселей сверху и слева от него. Это очень полезный прием для быстрой оценки признаков. Интегральные изображения используются в классе OpenCV `CascadeClassifier`, основанном на подходе, который был предложен в работе Виолы и Джонса [39]. Перед сохранением результирующего изображения производится нормировка с целью приведения к диапазону 0 ... 255, для чего выполняется деление на максимальное значение. Результат применения этого кода к изображению показан на рис. 10.1.

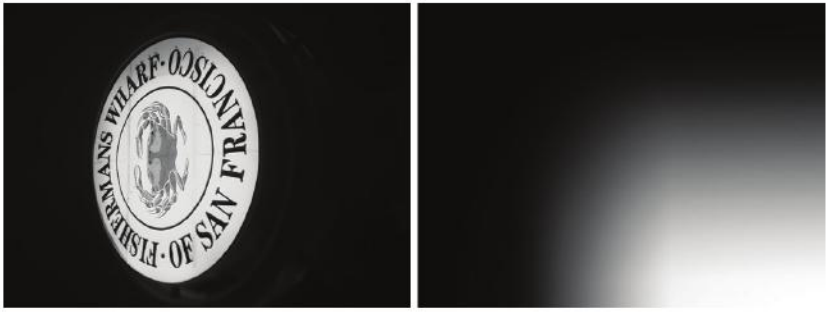


Рис. 10.1. Пример вычисления интегрального изображения с помощью функции OpenCV `integral()`

Во втором примере мы выполним заливку, начиная с заданного пикселя:

```
import cv2

# прочитать изображение
filename = 'fisherman.jpg'
im = cv2.imread(filename)
h,w = im.shape[:2]

# пример заливки
diff = (6,6,6)
mask = zeros((h+2,w+2),uint8)
cv2.floodFill(im,mask,(10,10),(255,255,0),diff,diff)

# показать результат в окне OpenCV
```

```
cv2.imshow('flood fill',im)
cv2.waitKey()

# сохранить результат
cv2.imwrite('result.jpg',im)
```

В этом примере к изображению применена заливка, а результат показан в окне OpenCV. Функция `waitKey()` ждет нажатия клавиши, после чего окно закрывается. Функция `floodFill()` принимает изображение (полутоновое или цветное), маску, в которой ненулевые пиксели обозначают области, которые не нужно заливать, начальный пиксель и новое значение цвета заливаемых пикселей, а также ограничения снизу и сверху на разность между яркостью или цветом текущего и начального пикселя, при которой заливка возможна. Ограничения задаются в виде кортежей (R, G, B). Результат показан на рис. 10.2.

Рис. 10.2. Заливка цветного изображения. В подсвеченной области на правом рисунке залиты все пиксели примерно такого же цвета, как у начальной точки в левом верхнем углу

В качестве последнего примера рассмотрим выделение SURF-признаков – более быстрый вариант алгоритма SIFT, описанный в работе [1]. Заодно покажем, как использовать простые команды построения графиков в OpenCV:

```
import cv2

# прочитать изображение
im = cv2.imread('empire.jpg')

# понизить разрешение
im_lowres = cv2.pyrDown(im)

# преобразовать в полутоновое
```

```
gray = cv2.cvtColor(im_lowres,cv2.COLOR_RGB2GRAY)

# найти особые точки
s = cv2.SURF()
mask = uint8(ones(gray.shape))
keypoints = s.detect(gray,mask)

# показать изображение и точки
vis = cv2.cvtColor(gray,cv2.COLOR_GRAY2BGR)
for k in keypoints[::10]:
  cv2.circle(vis,(int(k.pt[0]),int(k.pt[1])),2,(0,255,0),-1)
  cv2.circle(vis,(int(k.pt[0]),int(k.pt[1])),int(k.size),(0,255,0),2)

cv2.imshow('local descriptors',vis)
cv2.waitKey()
```

Прочитав изображение, мы понижаем его разрешение с помощью функции `pyrDown()`, которая уменьшает размер оригинала вдвое, если новый размер не задан явно. Затем изображение преобразуется в полутоновое и передается объекту обнаружения особых точек `SURF`. Параметр *mask* определяет, к каким областям применять детектор особых точек. Для построения графика мы преобразуем полутоновое изображение в цветное и зеленым цветом наносим особые точки. В цикле отбирается каждая десятая точка и рисуется маленький круг с центром в ней и еще одна окружность, показывающая масштаб (размер) особой точки. Функция `circle()` принимает изображение, кортеж с координатами изображения (целочисленными), радиус, кортеж, задающий цвет окружности, и толщину линии (−1 означает сплошной круг). Результат показан на рис. 10.3.

10.3. Обработка видео

Обработать видео на чистом Python трудно. Необходимо принимать во внимание быстродействие, кодеки, камеры, операционные системы и форматы файлов. В настоящее время для Python не существует библиотеки работы с видео. OpenCV с интерфейсом к Python – единственный приемлемый вариант. В этом разделе мы рассмотрим несколько простых примеров.

Ввод видео

В OpenCV имеется отличная поддержка считывания видео с камеры. Ниже приведен полный пример захвата кадров и показа их в окне OpenCV:

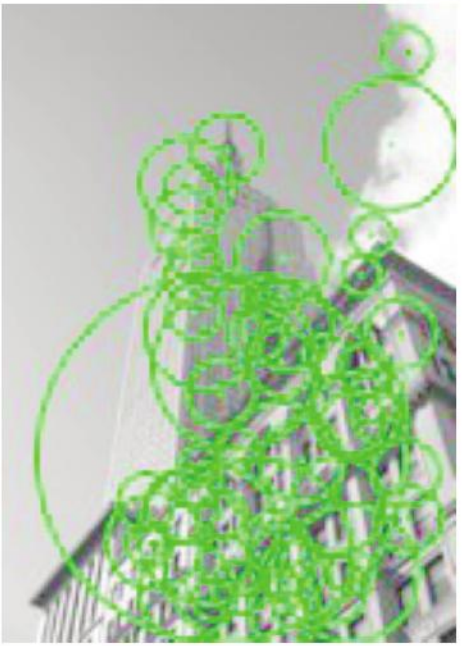


Рис. 10.3. Пример выделения и нанесения на график SURF-признаков с помощью OpenCV

```
import cv2

# настроить захват видеосигнала
cap = cv2.VideoCapture(0)

while True:
  ret,im = cap.read()
  cv2.imshow('video test',im)
  key = cv2.waitKey(10)
  if key == 27:
     break
  if key == ord(' '):
     cv2.imwrite('vid_result.jpg',im)
```

Объект `VideoCapture` читает видео с камеры или из файла. В данном случае мы передаем ему целое число – идентификатор видеоустройства; при наличии всего одной подключенной камеры оно равно 0. Метод `read()` декодирует и возвращает следующий видеокадр. Первое значение – код завершения, второе – собственно массив, содержащий изображение. Функция `waitKey()` ждет нажатия клавиши

и завершает программу, если нажата клавиша **Esc** (ее ASCII-код равен 27), или сохраняет кадр, если нажат пробел.

Разовьем этот пример, добавив простую обработку: покажем в окне OpenCV результат размытия цветного изображения, полученного от камеры. Для этого программу придется лишь немного модифицировать:

```
import cv2

# настроить захват видеосигнала
cap = cv2.VideoCapture(0)

# получить кадр, применить гауссово сглаживание, показать результат
while True:
  ret,im = cap.read()
  blur = cv2.GaussianBlur(im,(0,0),5)
  cv2.imshow('camera blur',blur)
  if cv2.waitKey(10) == 27:
     break
```

Каждый кадр передается функции `GaussianBlur()`, которая применяет фильтр Гаусса к изображению. Поскольку мы передаем цветное изображение, каждый канал размывается по отдельности. Функция принимает кортеж, содержащий размер фильтра и стандартное отклонение гауссовой функции (в данном случае 5). Если размер фильтра задан равным 0, то он будет автоматически определен по стандартному отклонению. Результат показан на рис. 10.4.

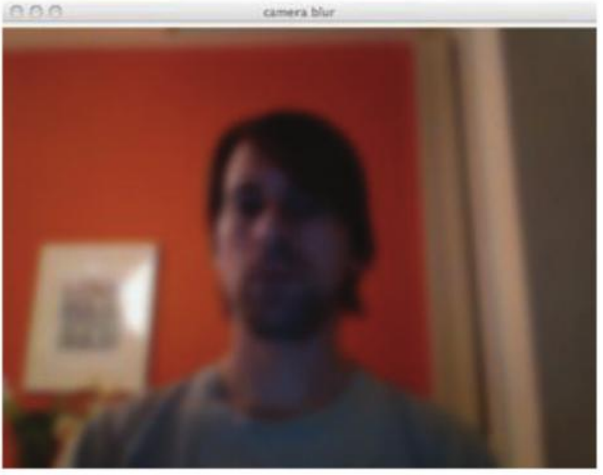


Рис. 10.4. Размытый видеокадр с изображением автора, пишущего эту главу

Чтение видео из файла производится точно так же, только функции `VideoCapture()` передается не номер устройства, а имя файла:

```
capture = cv2.VideoCapture('filename')
```

Чтение видео в массивы NumPy

С помощью OpenCV можно прочитать видеокадры с камеры или из файла и преобразовать их в массивы `NumPy`. Ниже приведен пример захвата видео с камеры и сохранения кадров в массиве `NumPy`:

```
import cv2

# настроить захват видеосигнала
cap = cv2.VideoCapture(0)
frames = []

# получить кадр, сохранить в массиве
while True:
  ret,im = cap.read()
  cv2.imshow('video',im)
  frames.append(im)
  if cv2.waitKey(10) == 27:
     break
frames = array(frames)

# проверить форму массива
print im.shape
print frames.shape
```

Массив, содержащий очередной кадр, добавляется в конец списка, и так продолжается, пока захват не будет остановлен. Результирующий массив имеет форму (число кадров, высота, ширина, 3), что и подтверждает распечатка.

```
(480, 640, 3)
(40, 480, 640, 3)
```

В данном случае было записано 40 кадров. Массивы, содержащие видеоданные, полезны для обработки видео, например, вычисления различий между кадрами и трассировки.

10.4. Трассировка

Трассировкой (tracking) называется процесс прослеживания объектов на последовательности изображений или видео.

Оптический поток

Оптическим потоком называет перемещение объектов на изображении по мере того, как сами объекты, сцена или камера меняют положение. Поток представляет собой двумерное векторное поле параллельных переносов внутри изображения. Это классическая, хорошо изученная область компьютерного зрения, имеющая много успешных приложений, в том числе для сжатия видео, анализа параметров движения, трассировки объектов и сегментации изображений.

Уравнения оптического потока опираются на три основных допущения.

1. Постоянство яркости. Яркости пикселей объекта не изменяются при переходе от предыдущего изображения к следующему.
2. Регулярность во времени. Время между кадрами достаточно мало, чтобы применять дифференциалы к рассмотрению изменения изображений (используются при выводе основного уравнения ниже).
3. Пространственная согласованность. Соседние пиксели движутся похожим образом.

Часто эти предположения нарушаются, но для малых перемещений и коротких временных интервалов между соседними изображениями эта модель подходит. Предположение о том, что пиксель объекта $I(x, y, t)$ в момент t имеет такую же яркость, как в момент $t + \delta t$ после перемещения на расстояние $[\delta x, \delta y]$ означает, что $I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t)$. Дифференцируя обе части, получаем *уравнение оптического потока*:

$$\nabla I^T \mathbf{v} = -I_t,$$

где $\mathbf{v} = [u, v]$ – вектор перемещения, а I_t – производная по времени. Для отдельных точек изображения это неопределенное уравнение, которое невозможно решить (в одном уравнении два неизвестных элемента $\mathbf{v}$). Но если добавить условие пространственной согласованности, то найти решение можно. Мы покажем, как используется это предположение в алгоритме Лукаса-Канаде.

В OpenCV есть несколько реализаций оптического потока: `CalcOpticalFlowBM()` с сопоставлением блоков; `CalcOpticalFlowHS()`, в которой используются результаты работы [15] (в настоящее время обе реализации существуют только в старом модуле `cv`); пирамидальный алгоритм Лукаса-Канаде [19] `calcOpticalFlowPyrLK()` и, наконец,

`calcOpticalFlowFarneback()`, основанная на работе [10]. Последняя реализация считается одним из лучших методов получения плотных оптических полей. Рассмотрим пример его применения к нахождению векторов перемещения в видео (алгоритм Лукаса-Канаде будет рассмотрен в следующем разделе).

Выполните следующий скрипт:

```
import cv2

def draw_flow(im,flow,step=16):
  """ Построить график оптического потока в нескольких точках,
     отстоящих друг от друга на step пикселей. """

  h,w = im.shape[:2]
  y,x = mgrid[step/2:h:step,step/2:w:step].reshape(2,-1)
  fx,fy = flow[y,x].T

  # создать конечные точки отрезков
  lines = vstack([x,y,x+fx,y+fy]).T.reshape(-1,2,2)
  lines = int32(lines)

  # создать и нарисовать изображение
  vis = cv2.cvtColor(im,cv2.COLOR_GRAY2BGR)
  for (x1,y1),(x2,y2) in lines:
     cv2.line(vis,(x1,y1),(x2,y2),(0,255,0),1)
     cv2.circle(vis,(x1,y1),1,(0,255,0), -1)
  return vis

# настроить захват видеосигнала
cap = cv2.VideoCapture(0)

ret,im = cap.read()
prev_gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

while True:
  # получить полутоновое изображение
  ret,im = cap.read()
  gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

  # вычислить оптический поток
  flow = cv2.calcOpticalFlowFarneback(prev_gray,gray,None,
                  0.5,3,15,3,5,1.2,0)
  prev_gray = gray

  # нанести на график векторы потока
  cv2.imshow('Optical flow',draw_flow(gray,flow))
  if cv2.waitKey(10) == 27:
     break
```

Здесь мы захватываем изображения с веб-камеры и вызываем функцию вычисления оптического потока для каждой пары последо-

вательных кадров. Векторы перемещения сохраняются в двухканальном изображении *flow*, которое возвращает функция `calcOpticalFlowFarneback()`. Помимо предыдущего и текущего кадра, эта функция принимает дополнительные параметры. Если интересно, почитайте о них в документации. Вспомогательная функция `draw_flow()` наносит на график векторы перемещения в точках изображения, отстоящих на одинаковое расстояние. Для этого используются входящие в OpenCV функции рисования `line()` и `circle()`, а параметр *step* определяет промежуток между выборочными точками потока. Результат показан на рис. 10.5. Здесь положения выборочных точек показаны в виде сетки окружностей, а векторы потока, представленные отрезками, показывают, как эти точки движутся.

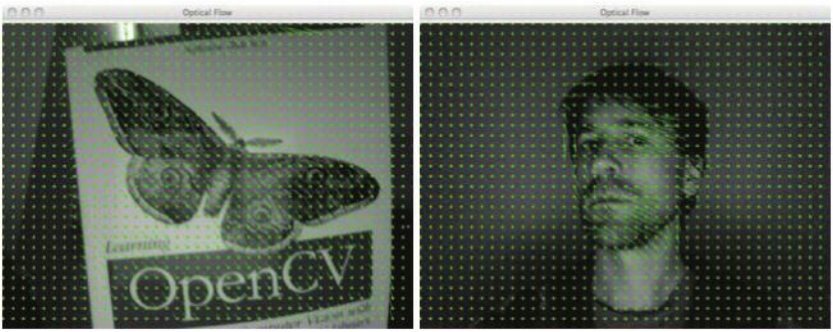


Рис. 10.5. Векторы оптического потока (с выборкой в каждом шестнадцатом пикселе), показанные на видео передвигаемой книги и поворачивающейся головы

Алгоритм Лукаса-Канаде

Самая простая форма трассировки – следить за перемещением особых точек, например углов. Для этой цели часто применяется *алгоритм трассировки Лукаса-Канаде*, в котором используется понятие разреженного оптического потока.

Трассировку методом Лукаса-Канаде можно применять к признакам любого типа, но обычно вместе с ней используются угловые точки, например углы Харриса из раздела 2.1. Функция `goodFeaturesToTrack()` обнаруживает углы методом Ши-Томаси [33], в котором углами считаются собственные векторы, соответствующие двум наибольшим собственным значениям структурного тензора (матрицы Харриса) (2.2), при условии, что меньшее собственное значение превышает заданный порог.

Уравнение оптического потока неопределенное (т. е. в нем больше неизвестных, чем уравнений), если его применять к каждому пикселю. Но в предположении, что соседние пиксели движутся одинаково, можно составить из нескольких таких уравнений систему:

$$\begin{bmatrix} \nabla I^T(\mathrm{x}_1) \\ \nabla I^T(\mathrm{x}_2) \\ \vdots \\ \nabla I^T(\mathrm{x}_n) \end{bmatrix} \mathrm{v} = \begin{bmatrix} I_x(\mathrm{x}_1) & I_y(\mathrm{x}_1) \\ I_x(\mathrm{x}_2) & I_y(\mathrm{x}_2) \\ \vdots & \vdots \\ I_x(\mathrm{x}_n) & I_y(\mathrm{x}_n) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(\mathrm{x}_1) \\ I_t(\mathrm{x}_2) \\ \vdots \\ I_t(\mathrm{x}_n) \end{bmatrix}$$

для некоторой окрестности, состоящей из n пикселей. В такой системе уравнений больше, чем неизвестных, поэтому ее можно решить методом наименьших квадратов. Обычно вкладам окружающих пикселей назначаются веса, так что чем дальше отстоит пиксель, тем меньше его влияние. Чаще всего применяют гауссово взвешивание. Тогда приведенная выше матрица превращается в структурный тензор в уравнении (2.2), и мы получаем соотношение:

$$\bar{\mathrm{M}}_I \mathrm{v} = - \begin{bmatrix} I_t(\mathrm{x}_1) \\ I_t(\mathrm{x}_2) \\ \vdots \\ I_t(\mathrm{x}_n) \end{bmatrix} \quad \text{или} \quad \text{проще } A\mathbf{v} = \mathbf{b}.$$

Эту переопределенную систему уравнений можно решить в смысле метода наименьших квадратов, тогда вектор перемещения описывается формулой:

$$\mathbf{v} = (A^T A)^{-1} A^T \mathbf{b}.$$

Система имеет решение, только если матрица $A^T A$ обратима, а это так по построению, если она применяется в угловых точках Харриса или в «хороших точках для трассировки» в смысле алгоритма Ши-Томаси. Именно так вычисляются векторы перемещения в алгоритме трассировки Лукаса-Канаде.

Стандартный алгоритм трассировки Лукаса-Канаде работает для малых смещений. Для обработки больших смещений применяется иерархический подход. В этом случае оптический поток вычисляется для вариантов изображения с возрастающей детализацией. Для этой цели предназначена функция OpenCV `calcOpticalFlowPyrLK()`.

Функции Лукаса-Канаде включены в OpenCV. Посмотрим, как их использовать для построения класса-трассировщика на Python. Создайте файл *lktrack.py* и добавьте в него следующий класс и конструктор:

```
import cv2

# константы и параметры по умолчанию
lk_params = dict(winSize=(15,15),maxLevel=2,
    criteria=(cv2.TERM_CRITERIA_EPS|cv2.TERM_CRITERIA_COUNT,10,0.03))

subpix_params = dict(zeroZone=(-1,-1),winSize=(10,10),
    criteria = (cv2.TERM_CRITERIA_COUNT | cv2.TERM_CRITERIA_EPS,20,0.03))

feature_params = dict(maxCorners=500,qualityLevel=0.01,minDistance=10)


class LKTracker(object):
  """ Класс трассировки Лукаса-Канаде с пирамидальным
      оптическим потоком. """

  def __init__(self,imnames):
      """ Инициализировать списком имен файлов изображений. """

      self.imnames = imnames
      self.features = []
      self.tracks = []
      self.current_frame = 0
```

Конструктору объекта-трассировщика передается список имен файлов. В списках *features* и *tracks* будут храниться угловые точки и их трассы. В переменной *current_frame* хранится номер текущего кадра. Мы определили три словаря, содержащих параметры выделения признаков, трассировки и определения особой точки с субпиксельной точностью.

Для того чтобы начать обнаружение точек, мы должны загрузить само изображение, преобразовать его в полутоновое и выделить «хорошие точки для трассировки». Основную работу выполняет функция OpenCV `goodFeaturesToTrack()`. Добавьте в класс метод `detect_points()`:

```
  def detect_points(self):
    """ Найти 'хорошие точки для трассировки' (углы) в текущем
        кадре с субпиксельной точностью. """

    # загрузить изображение и преобразовать его в полутоновое
    self.image = cv2.imread(self.imnames[self.current_frame])
```

```
    self.gray = cv2.cvtColor(self.image,cv2.COLOR_BGR2GRAY)

    # искать хорошие точки
    features = cv2.goodFeaturesToTrack(self.gray, **feature_params)

    # уточнить положение углов
    cv2.cornerSubPix(self.gray,features, **subpix_params)
    self.features = features
    self.tracks = [[p] for p in features.reshape((-1,2))]

    self.prev_gray = self.gray
```

Мы уточняем положение точек с помощью функции `cornerSubPix()` и сохраняем в переменных-членах *features* и *tracks*. Отметим, что после выполнения этой функции история трассировки очищается.

Итак, особые точки мы знаем, теперь надо трассировать их. Прежде всего, необходимо получить следующий кадр, применить к нему функцию OpenCV `calcOpticalFlowPyrLK()`, которая определяет, какие точки переместились, а затем удалить пропавшие точки и очистить их списки трассировки. Все это делает метод `track_points()`:

```
def track_points(self):
  """ Трассировать обнаруженные признаки. """

  if self.features != []:
     self.step() # перейти к следующему кадру

     # загрузить изображение и преобразовать в полутоновое
     self.image = cv2.imread(self.imnames[self.current_frame])
     self.gray = cv2.cvtColor(self.image,cv2.COLOR_BGR2GRAY)

     # изменить форму в соответствии требованиями к входному формату
     tmp = float32(self.features).reshape(-1, 1, 2)

     # вычислить оптический поток
     features,status,track_error = cv2.calcOpticalFlowPyrLK(self.prev_gray,
                    self.gray,tmp,None,**lk_params)

     # удалить пропавшие точки
     self.features = [p for (st,p) in zip(status,features) if st]

     # очистить трассы пропавших точек
     features = array(features).reshape((-1,2))
     for i,f in enumerate(features):
        self.tracks[i].append(f)
     ndx = [i for (i,st) in enumerate(status) if not st]
     ndx.reverse() # удалить из конца
     for i in ndx:
```

```
            self.tracks.pop(i)

        self.prev_gray = self.gray
```

Здесь мы пользуемся вспомогательным методом `step()` для перехода к следующему кадру.

```
def step(self,framenbr=None):
  """ Перейти к другому кадру. Если аргумент не задан,
     то перейти к следующему кадру. """

  if framenbr is None:
     self.current_frame = (self.current_frame + 1) % len(self.imnames)
  else:
     self.current_frame = framenbr % len(self.imnames)
```

Этот метод позволяет перейти к указанному кадру или к следующему, если кадр не указан явно. Наконец, мы хотим еще нарисовать результат в окне OpenCV. Добавьте в класс `LKTracker` такой метод `draw()`:

```
def draw(self):
  """ Нарисовать текущее изображение вместе с точками,
     пользуясь функциями OpenCV.
     Для закрытия окна нажать любую клавишу. """

  # нарисовать точки зелеными кружочками
  for point in self.features:
    cv2.circle(self.image,(int(point[0][0]),int(point[0][1])),3,(0,255,0),-1)

  cv2.imshow('LKtrack',self.image)
  cv2.waitKey()
```

Теперь у нас есть полная замкнутая система трассировки, основанная на функциях OpenCV.

Использование трассировщика

Соберем все вместе, воспользовавшись написанным классом трассировщика в реальной ситуации. Показанный ниже скрипт инициализирует объект, обнаруживает и трассирует точки в последовательности кадров, а затем рисует результат.

```
import lktrack

imnames = ['bt.003.pgm', 'bt.002.pgm', 'bt.001.pgm', 'bt.000.pgm']

# создать объект-трассировщик
```

```
lkt = lktrack.LKTracker(imnames)

# найти особые точки в первом кадре и трассировать их в последующих
lkt.detect_points()
lkt.draw()
for i in range(len(imnames)-1):
  lkt.track_points()
  lkt.draw()
```

Мы рисуем по одному кадру за раз и показываем точки, которые еще трассируются. Для перехода к следующему кадру нужно нажать любую клавишу. На рис. 10.6 показаны первые четыре изображения коридора в Оксфорде (один из многовидовых оксфордских наборов данных, доступный по адресу *http://www.robots.ox.ac.uk/~vgg/data/data-mview.html*).

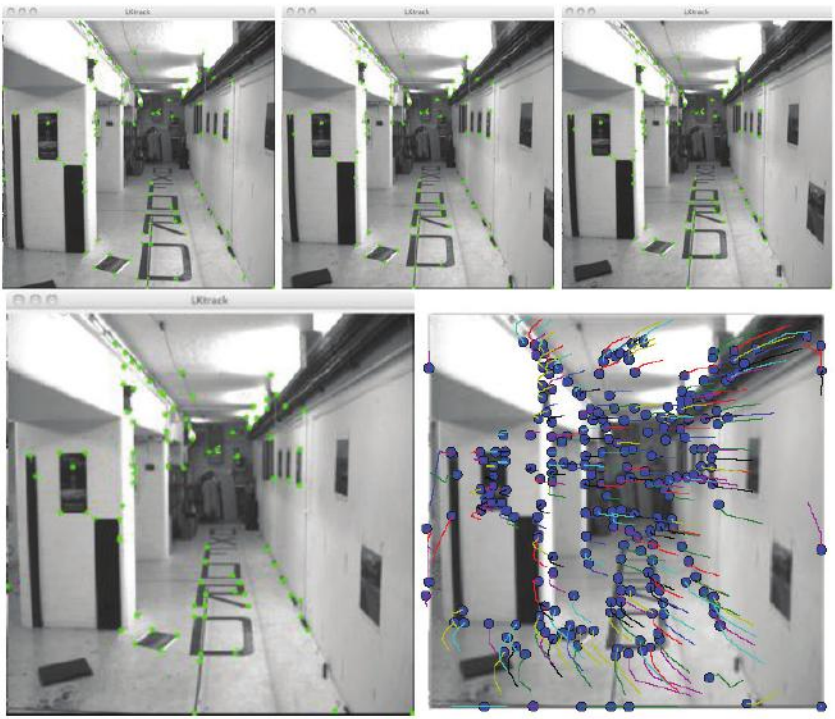


Рис. 10.6. Трассировка методом Лукаса-Канаде с помощью класса LKTrack

Применение генераторов

Добавьте в класс `LKTracker` следующий метод:

```
def track(self):
  """ Генератор для обхода последовательности. """
  for i in range(len(self.imnames)):
     if self.features == []:
        self.detect_points()
     else:
        self.track_points()

  # создать копию в формате RGB
  f = array(self.features).reshape(-1,2)
  im = cv2.cvtColor(self.image,cv2.COLOR_BGR2RGB)
  yield im,f
```

Здесь создается генератор, который упрощает обход последовательности и отдает трассы и изображения в виде массивов RGB, чтобы было проще рисовать результат. Ниже показано его применение к классической последовательности динозавров, созданной в Оксфорде (находится на той же странице, что и изображения коридора выше):

```
import lktrack
imnames = ['viff.000.ppm', 'viff.001.ppm',
           'viff.002.ppm', 'viff.003.ppm', 'viff.004.ppm']

# трассировать с помощью генератора LKTracker
lkt = lktrack.LKTracker(imnames)
for im,ft in lkt.track():
  print 'трассируется %d признаков' % len(ft)

# нанести трассы
figure()
imshow(im)
for p in ft:
  plot(p[0],p[1],'bo')
for t in lkt.tracks:
  plot([p[0] for p in t],[p[1] for p in t])
axis('off')
show()
```

Этот генератор заметно упрощает использование класса трассировщика и полностью скрывает от пользователя функции OpenCV. В нашем примере строятся графики, показанные на рис. 10.7 и справа внизу на рис. 10.6.

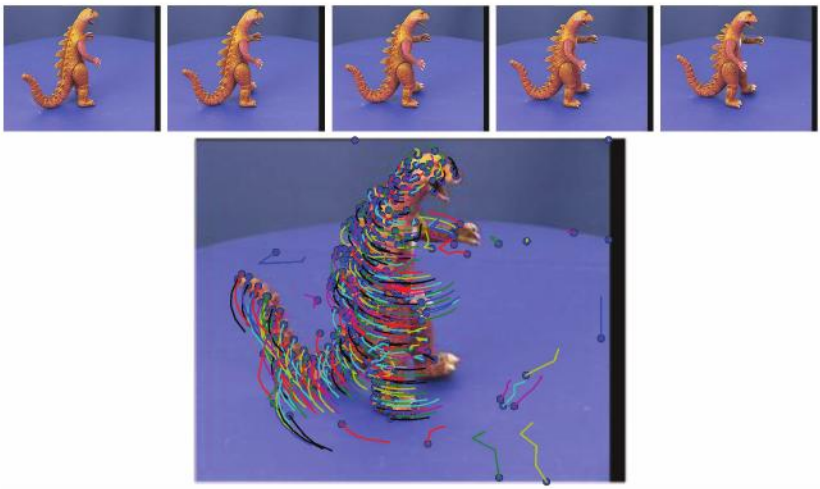


Рис. 10.7. Пример трассировки последовательности изображений поворачивающегося объекта методом Лукаса-Канаде с графиком трассы точек

10.5. Другие примеры

В состав OpenCV входит ряд полезных примеров использования интерфейса с Python. Они находятся в подкаталоге samples/python2/. Их изучение – отличный способ ближе познакомиться с OpenCV. Ниже приведено несколько примеров, иллюстрирующих другие возможности OpenCV.

Ретуширование

Реконструкция утраченных или испорченных частей изображения называется *ретушированием*. Сюда относятся как собственно алгоритмы восстановления дефектных частей, так и устранение красных глаз или отдельных объектов в фоторедакторах. Обычно некий участок изображения помечается как «дефектный», т. е. нуждающийся в заполнении с помощью данных из других частей изображения.

Выполните такую команду:

```
$ python inpaint.py empire.jpg
```

Она открывает интерактивное окно, в котором можно пометить области, нуждающиеся в ретушировании. Результаты показываются в отдельном окне. Пример приведен на рис. 10.8.

Рис. 10.8. Пример ретуширования с помощью OpenCV. На левом изображении пользователь пометил отдельные участки как «дефектные». На правом изображении показан результат после ретуширования

Сегментация по морфологическим водоразделам

В обработке изображений *водоразделом* называется метод, применяемый для сегментации (см. рис. 10.9). Изображение рассматривается как топографический ландшафт, «затопленный» из нескольких начальных областей. Обычно используется изображение, построенное по модулю градиента, потому что на нем гребни выглядят как отчетливые границы, а сегментация останавливается на границах изображений.

В OpenCV используется алгоритм Мейера [22]. Выполните следующую команду:

```
$ python watershed.py empire.jpg
```

Она открывает интерактивное окно, в котором можно очертить начальные области, являющиеся входными данными для алгоритма. Результат показывается во втором окне, где цветом обозначены выделенные сегменты.

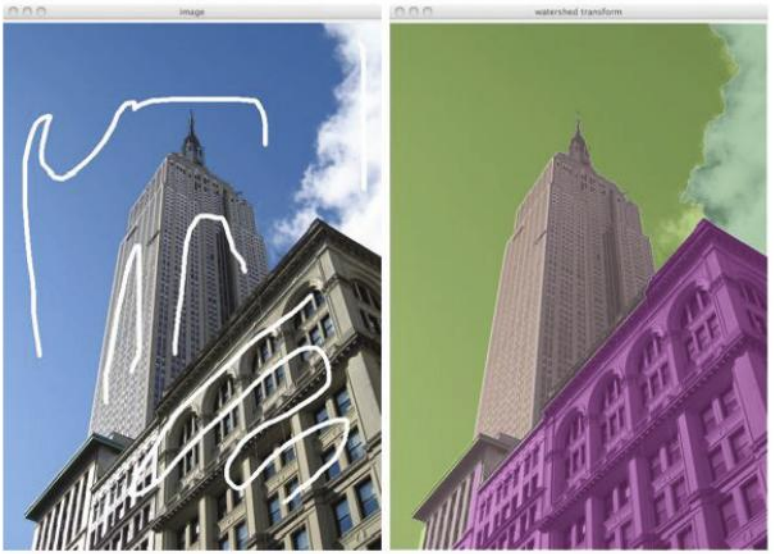


Рис. 10.9. Пример сегментации изображения по морфологическим водоразделам. Слева показано исходное изображение с начальными областями, а справа – результирующее изображение, в котором сегменты обозначены цветами

Обнаружение фигур с помощью преобразования Хафа

Преобразование Хафа (*https://ru.wikipedia.org/wiki/Преобразование_Хафа*) – это метод нахождения фигур в изображении. В нем используется процедура голосования в параметрическом пространстве фигур. Наиболее распространенное применение – поиск линейных структур в изображениях. В этом случае границы и отрезки прямых можно сгруппировать вместе, дав им возможность проголосовать за одни и те же линейные параметры в двумерном параметрическом пространстве прямых.

Пример в составе OpenCV обнаруживает прямые именно таким способом[2]. Выполните следующую команду:

```
$ python houghlines.py empire.jpg
```

В результаты вы увидите два окна, показанных на рис. 10.10. В одном окне показано исходное полутоновое изображение, а во вто-

[2] В текущей версии этот пример находится в папке */samples/python*.

ром – карта краёв вместе с обнаруженными прямыми, получившими максимальное количество голосов в параметрическом пространстве. Отметим, что прямые всегда бесконечны; если вы захотите найти концы отрезков этих прямых в изображении, то можете воспользоваться картой краёв.

Рис. 10.10. Пример обнаружения прямых с помощью преобразования Хафа. Слева показано исходное полутоновое изображение, а справа – карта краёв с обнаруженными прямыми

Упражнения

1. С помощью оптического потока разработайте простую систему распознавания жестов. Например, можно произвести выборку из потока, как в функции построения графика, и использовать выборочные векторы в качестве входных данных.
2. В OpenCV есть две функции деформирования: `cv2.warpAffine()` и `cv2.warpPerspective()`. Попробуйте применить их к примерам из главы 3.
3. Используйте функцию заливки для вычитания заднего плана на оксфордских изображениях динозавра (рис. 10.7). Создайте новые

изображения, в которых динозавр нарисован на фоне другого цвета или поверх другого изображения.

4. В OpenCV имеется функция `cv2.findChessboardCorners()`, которая автоматически находит углы шахматного узора. Воспользуйтесь ей для нахождения соответствий при калибровке камеры с помощью функции `cv2.calibrateCamera()`.
5. Если у вас есть две камеры, соберите из них стереоспарку и захватите стереопары изображения с помощью функции `cv2.VideoCapture()` с разными идентификаторами видеоустройств. Для начала попробуйте идентификаторы 0 и 1. Вычислите глубину изображений для нескольких изменяющихся сцен.
6. Воспользуйтесь моментами Ху, полученными от функции `cv2.HuMoments()`, как признаками для решения задачи об оптическом распознавании судоку из раздела 8.4. Оцените качество классификации.
7. В OpenCV имеется реализация алгоритма сегментации Grab Cut. Примените функцию `cv2.grabCut()` к набору данных Grab Cut научно-исследовательского центра Microsoft (см. раздел 9.1). Хочется надеяться, что результаты будут лучше полученных нами с помощью сегментации изображений низкого разрешения.
8. Модифицируйте класс трассировщика Лукаса-Канаде, так чтобы он принимал на входе видеофайл, и напишите скрипт, который трассирует точки на последовательности кадров и через каждые k кадров обнаруживает новые точки.

ПРИЛОЖЕНИЕ А. Установка пакетов

Ниже приведены краткие инструкции по установке пакетов, используемых в книге. Они рассчитаны на версии, которые были последними на момент написания. Но все в мире меняется (включая URL-адреса!), поэтому если какие-то инструкции устареют, обращайтесь к самому сайту проекта.

На большинстве платформ работает также Python-скрипт easy_install, которым можно воспользоваться, вместо того чтобы читать инструкции. Если при выполнении инструкций возникнут проблемы, попробуйте easy_install. Подробнее прочитать о нем можно на сайте пакета *http://packages.python.org/distribute/easy_install.html*.

A.1. NumPy и SciPy

Порядок установки `NumPy` и `SciPy` зависит от операционной системы. На большинстве платформ текущими являются версии NumPy 2.0 и SciPy 0.11. На всех основных платформах работает бесплатный дистрибутив Enthought EPD Free, облегченный вариант коммерческого дистрибутива Enthought, доступный по адресу *http://enthought.com/products/epd_free.php*.

Windows

Самый простой способ обзавестись `NumPy` и `SciPy` – скачать и установить двоичные дистрибутивы с сайта *http://www.scipy.org/Download*.

Mac OS X

В последние версии Mac OS X (начиная с 10.7.0 [Lion]) пакет `NumPy` уже включен.

Установить `NumPy` и `SciPy` для Mac OS X позволяет «суперпакет» по адресу *https://github.com/fonnesbeck/ScipySuperpack*. Заодно вы получаете и `Matplotlib`.

Альтернатива – воспользоваться системой управления пакетами MacPorts (*http://www.macports.org/*). Она работает и для `Matplotlib` тоже.

Если ни один из этих вариантов не дал результата, зайдите на страницу проекта *http://scipy.org/*, где описан еще ряд способов.

Linux

Для установки необходимы права администратора. В некоторые дистрибутивы `NumPy` уже входит, в другие – нет. И `NumPy`, и `SciPy` проще всего установить с помощью встроенного менеджера пакетов (например, Synaptic в Ubuntu). Он же поможет установить `Matplotlib` – как альтернатива приведенным ниже инструкциям.

A.2. Matplotlib

Ниже приведены инструкции по установке `Matplotlib` в том случае, когда она не установилась вместе с `NumPy/SciPy`. Библиотеку `Matplotlib` можно бесплатно получить на сайте `http://matplotlib.sourceforge.net/`. Щелкнув по ссылке «download», вы скачаете установщик последней версии для своей системы и версии Python. В настоящее время последней является версия 1.1.0.

Можно также просто скачать и распаковать исходный код. После выполнения команды

```
$ python setup.py install
```

все должно заработать. Общие рекомендации по установке на разные системы можно найти на странице *http://matplotlib.sourceforge.net/users/installing.html*, но описанная процедура должна работать для большинства платформ и версий Python.

A.3. PIL

Библиотека PIL, Python Imaging Library, доступна по адресу *http://www.pythonware.com/products/pil/*. Последняя бесплатная версия имеет номер 1.1.7. Скачайте исходный код и распакуйте архив. Перейдя в распакованную папку, выполните команду

```
$ python setup.py install
```

Для сохранения изображений, созданных PIL, на компьютере должны быть установлены библиотеки для работы с JPEG (libjpeg) и PNG (zlib). Если возникнут проблемы, почитайте файл README или зайдите на сайт PIL.

A.4. LibSVM

Номер текущей версии 3.1 (выпущена в апреле 2011). Скачайте zip-файл с сайта LibSVM (*http://www.csie.ntu.edu.tw/~cjlin/libsvm/*). После его распаковки будет создан каталог libsvm-3.1). В окне терминала перейдите в этот каталог и наберите `make`:

```
$ cd libsvm-3.0
$ make
```

Затем перейдите в каталог python и сделайте то же самое:

```
$ cd python/
$ make
```

Больше ничего делать не надо. Для проверки работоспособности запустите Python из командной строки и выполните команду

```
import svm
```

Авторы написали практическое руководство по работе с LivSVM [7], которое послужит хорошей отправной точкой.

A.5. OpenCV

Установка OpenCV зависит от операционной системы. Следуйте инструкциям ниже.

Для проверки работоспособности запустите Python и попробуйте выполнить примеры, описанные на странице *http://opencv.willowgarage.com/documentation/python/cookbook.html*. Онлайновое справочное руководство по OpenCV Python на странице *http://opencv.willowgarage.com/documentation/python/index.html* содержит дополнительные примеры и подробные сведения о том, как использовать OpenCV совместно с Python.

Windows и Unix

В репозитории SourceForge по адресу *http://sourceforge.net/projects/opencvlibrary/* имеются установщики для Windows и Unix.

Mac OS X

Для Mac OS X поддержки пока нет, но работы ведутся. Есть несколько способов собрать библиотеку из исходного кода, они описаны на вики-сайте OpenCV по адресу *http://opencv.willowgarage.com/wiki/InstallGuide*. Альтернативой является система MacPorts, которая хорошо работает, если вы устанавливали Python, NumPy, SciPy и Matplotlib тоже с помощью MacPorts. Для сборки OpenCV из исходного кода нужно выполнить следующие команды:

```
$ svn co https://code.ros.org/svn/opencv/trunk/opencv
$ cd opencv/
$ sudo cmake -G "Unix Makefiles" .
$ sudo make -j8
$ sudo make install
```

Если все зависимости присутствуют, то компиляция и установка должны завершиться успешно. Если выдается ошибка вида

```
import cv2
Traceback (most recent call last):
 File "", line 1, in
ImportError: No module named cv2
```

то необходимо добавить в переменную окружения PYTHONPATH каталог, содержащий файл *cv2.so*, например:

```
$ export PYTHONPATH=$PYTHONPATH:/usr/local/lib/python2.7/site-packages/
```

Linux

Пользователи Linux могут попробовать установить пакет с помощью менеджера пакетов для своего дистрибутива (пакет обычно называется «opencv») или собрать его из исходного кода, как описано в разделе, посвященном Mac OS X.

A.6. VLFeat

Для установки VLFeat скачайте и распакуйте последнюю версию двоичного дистрибутива со страницы *http://vlfeat.org/download.html* (в настоящий момент это версия 0.9.14). Добавьте в переменную окружения пути или скопируйте двоичные файлы в каталог, уже указанный в списке путей. Двоичные файлы находятся в каталоге *bin/*, вам остается только выбрать подкаталог для своей платформы.

Как работать с командами VLFeat, написано в файле, находящемся в подкаталоге *src/*. Документация также имеется на странице *http://vlfeat.org/man/man.html*.

A.7. PyGame

Пакет PyGame можно скачать со страницы *http://www.pygame.org/download.shtml*. Номер последней версии 1.9.1. Проще всего забрать двоичный установочный пакет для своей системы и версии Python.

Можно вместо этого скачать исходный код и в созданной папке выполнить команду

```
$ python setup.py install
```

A.8. PyOpenGL

Для установки PyOpenGL проще всего скачать пакет со страницы *http://pypi.python.org/pypi/PyOpenGL*, как описано на сайте проекта *http://pyopengl.sourceforge.net/*. Номер последней версии 3.0.1.

В созданной папке, как обычно, выполните команду

```
$ python setup.py install
```

Если возникли трудности или нужна информация о зависимостях, то дополнительную документацию можно найти на странице *http://pyopengl.sourceforge.net/documentation/installation.html*. Хорошие демонстрационные примеры, которые помогут освоиться, есть на странице *http://pypi.python.org/pypi/PyOpenGL-Demo*.

A.9. Pydot

Сначала установите зависимости: GraphViz и Pyparsing. Зайдите на сайт *http://www.graphviz.org/* и скачайте двоичный дистрибутив GraphViz для своей платформы. Установочные файлы должны установить GraphViz автоматически.

Затем перейдите на страницу проекта Pyparsing *http://pyparsing.wikispaces.com/*. Страница загрузки находится по адресу *http://sourceforge.net/projects/pyparsing/*. Скачайте последнюю версию (в настоящее время 1.5.5) и распакуйте ее в какой-нибудь каталог. Выполните команду

```
$ python setup.py install
```

Наконец, перейдите на страницу проекта *http://code.google.com/p/pydot/* и щелкните по ссылке «download». Со страницы загрузки скачайте последнюю версию (в настоящее время 1.0.4). Распакуйте и снова выполните команду

```
$ python setup.py install
```

Теперь команда `import pydot` в Python-скриптах должна работать.

A.10. Python-graph

Python-graph – это модуль Python для работы с графами, он содержит много полезных алгоритмов, например: обход графа, поиск кратчайшего пути, вычисление PageRank и максимального потока. Номер последней версии 1.8.1, ее можно скачать с сайта проекта *http://code.google.com/p/pythongraph/*.

Если в вашей системе имеется скрипт easy_install, то установить python-graph проще всего так:

```
$ easy_install python-graph-core
```

Но можно скачать исходный код со страницы *http://code.google.com/p/python-graph/downloads/list* и выполнить команду:

```
$ python setup.py install
```

Для создания и визуализации графов (на языке DOT) понадобится пакет python-graphdot, который можно скачать самостоятельно или установить с помощью easy_install:

```
$ easy_install python-graph-dot
```

Пакет python-graph-dot зависит от pydot; см. выше. Документация (в формате html) находится в папке *docs/*.

A.11. Simplejson

Simplejson – независимо сопровождаемая версия модуля JSON, который входит в последние версии Python (начиная с 2.6). Синтаксис в обоих модулях одинаковый, но simplejson лучше оптимизирован и работает быстрее.

Для установки перейдите на страницу проекта *https://github.com/simplejson/simplejson* и нажмите кнопку Download. Затем выберите последнюю версию в разделе «Download Packages» (в настоящее время 2.1.3). Распакуйте в каталог и выполните команду

```
$ python setup.py install
```

A.12. PySQLite

PySQLite – интерфейс SQLite к Python. SQLite – это облегченная дисковая база данных, к которой можно обращаться с запросами на языке SQL. Отличается простотой установки и использования. Номер последней версии 2.6.3; дополнительные сведения смотрите на сайте проекта *http://code.google.com/p/pysqlite/*.

Для установки скачайте пакета со страницы *http://code.google.com/p/pysqlite/downloads/list* и распакуйте в какой-нибудь каталог. Выполните команду

```
$ python setup.py install
```

A.13. CherryPy

CherryPy (*http://www.cherrypy.org/*) – быстрый, стабильный и потребляющий мало ресурсов веб-сервер, написанный на Python с использованием объектно-ориентированной модели. Установить CherryPy просто; скачайте последнюю версию (в настоящее время 3.2.0) со страницы *http://www.cherrypy.org/wiki/CherryPyInstall*. Распакуйте и выполните команду

```
$ python setup.py install
```

После установку ознакомьтесь с примерами в очень кратком пособии, которое находится в каталоге *cherrypy/tutorial/*. Из них вы узнаете, как передавать переменные из GET и POST-запросов, как наследовать свойства страницы, скачивать и закачивать файлы и т. д.

ПРИЛОЖЕНИЕ Б. Наборы изображений

Б.1. Flickr

Невероятно популярный сайт обмена фотографиями Flickr (*http://flickr.com/*) – золотая жила для специалистов и любителей, занимающихся компьютерным зрением. Располагая миллионами изображений, многие из которых аннотированы пользователями, этот сайт является неоценимым ресурсом для получения обучающих данных и экспериментирования с реальными данными. Flickr предлагает API доступа к службе, позволяющий скачивать, закачивать и аннотировать изображения (наряду со многими другими вещами). Полное описание API можно найти по адресу *http://flickr.com/services/api/*. Существуют его адаптации для многих языков программирования, включая Python.

Рассмотрим библиотеку flickrpy, которую бесплатно предлагается на странице *http://code.google.com/p/flickrpy/*. Скачайте файл *flickr.py*. Для работы с ним необходимо получить ключ API Key от Flickr. Ключи бесплатны для некоммерческого использования, а для коммерческого их можно запросить. Щелкните по ссылке «Apply for a new API Key» на странице Flickr API и следуйте инструкциям. Получив ключ API, откройте файл *flickr.py* и подставьте в строку

```
API_KEY = ''
```

свой ключ, например:

```
API_KEY = '123fbbb81441231123cgg5b123d92123'
```

Напишем простую командную утилиту, которая будет скачивать изображения, помеченные указанной меткой. Создайте файл *tagdownload.py* и добавьте в него следующий код:

```
import flickr
import urllib, urlparse
```

```
import os
import sys

if len(sys.argv)>1:
  tag = sys.argv[1]
else:
  print 'не указана метка'

# скачивание данных об изображениях
f = flickr.photos_search(tags=tag)
urllist = []  # сохранить загруженное в списке

# скачивание самих изображений
for k in f:
  url = k.getURL(size='Medium', urlType='source')
  urllist.append(url)
  image = urllib.URLopener()
  image.retrieve(url, os.path.basename(urlparse.urlparse(url).path))
  print 'скачивается:', url
```

Если вы хотите также сохранить в текстовом файле список URL-адресов, добавьте в конец такие строчки:

```
# вывести список URL-адресов в файл
fl = open('urllist.txt', 'w')
for url in urllist:
  fl.write(url+'\n')
fl.close()
```

В результате выполнения команды

```
$ python tagdownload.py goldengatebridge
```

вы получите последние 100 изображений с меткой «goldengatebridge» (мост Золотые ворота). Мы задали размер «Medium». Но это лишь один из многих вариантов размера, при желании можно получить миниатюры или полноразмерные оригиналы; см. документацию на странице *http://flickr.com/api/*.

Нас здесь интересует только скачивание изображений; для вызовов API, требующих аутентификации, процедура несколько сложнее. О том, как открыть аутентифицированный сеанс, см. документацию по API.

Б.2. Panoramio

Хорошим источником изображений с геометками является веб-служба Google по обмену фотографиями Panoramio (*http://www.panoramio.com/*). Она предоставляет API для программного доступа

к содержимому. API описан на странице *http://www.panoramio.com/api/*. Можно получать веб-виджеты и обращаться к данным с помощью объектов JavaScript. Проще всего загрузить изображения с помощью GET-запроса, например:

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&
set=public&from=0&to=20&minx=-180&miny=-90&maxx=180&maxy=90&
size =medium
```

где *minx*, *miny*, *maxx*, *maxy* – географические координаты области, для которой выбираются фотографии (минимальная долгота и широта, максимальная долгота и широта соответственно). Ответ придет в формате JSON, например:

```
{"count": 3152, "photos":
[{"upload_date": "02 February 2006", "owner_name": "***", "photo_id": 9439,
"longitude": -151.75, "height": 375, "width": 500, "photo_title": "***",
"latitude": -16.5, "owner_url": "http://www.panoramio.com/user/1600",
"owner_id": 1600,
"photo_file_url": "http://mw2.google.com/mw-panoramio/photos/medium/9439.jpg",
"photo_url": "http://www.panoramio.com/photo/9439"},
{"upload_date": "18 January 2011", "owner_name": "***", "photo_id": 46752123,
"longitude": 120.52718600000003, "height": 370, "width": 500, "photo_title": "***",
"latitude": 23.327833999999999, "owner_url": "http://www.panoramio.com/
user/2780232",
"owner_id": 2780232,
"photo_file_url": "http://mw2.google.com/mw-panoramio/photos/medium/46752123.jpg",
"photo_url": "http://www.panoramio.com/photo/46752123"},
{"upload_date": "20 January 2011", "owner_name": "***", "photo_id":
46817885,
"longitude": -178.13709299999999, "height": 330, "width": 500, "photo_title":
"***",
"latitude": -14.310613, "owner_url": "http://www.panoramio.com/user/919358",
"owner_id": 919358,
"photo_file_url": "http://mw2.google.com/mw-panoramio/photos/medium/46817885.jpg",
"photo_url": "http://www.panoramio.com/photo/46817885"},
...
], "has_more": true}
```

С помощью пакета JSON можно выделить из результата поле «photo_file_url». Пример см. в разделе 2.3.

Б.3. Оксфордская группа Visual Geometry

Исследовательская группа Visual Geometry из Оксфордского университета опубликовала много наборов данных на странице *http://www.*

robots.ox.ac.uk/~vgg/data/. В этой книге мы использовали несколько многовидовых наборов, в частности «Merton1», «Model House», «dinosaur» и «corridor». Данные, доступные для загрузки (некоторые содержат матрицы камер и трассы точек), выложены на странице *http://www.robots.ox.ac.uk/~vgg/data/data-mview.html*.

Б.4. Эталонные изображения для распознавания Кентуккийского университета

Набор эталонных изображений, подготовленный Кентуккийским университетом (часто его называют «ukbench»), содержит 2550 групп изображений. В каждой группе четыре фотографии объекта или сцены, снятые с разных точек. Этот набор удобен для тестирования алгоритмов распознавания объектов и поиска изображений по содержанию. Набор данных (объем полного набора составляет примерно 1,5 ГБ) можно скачать со страницы *http://www.vis.uky.edu/~stewe/ukbench/*. Подробное описание имеется в статье [23].

В этой книге мы использовали подмножество, включающее первые 1000 изображений.

Б.5. Другие наборы

Пражский генератор данных и эталонный набор для сегментации текстур

Этот набор данных использовался в главе о сегментации, он позволяет генерировать различные типы текстур для тестирования сегментации. Доступен по адресу *http://mosaic.utia.cas.cz/index.php*.

Набор данных Grab Cut научно-исследовательского центра Microsoft в Кембридже

Первоначально использованный в статье [27], этот набор содержит изображения для исследования сегментации, снабженные пользовательскими аннотациями. Сам набор данных и несколько связанных с ним работ имеются на странице *http://research.microsoft.com/*

en-us/um/cambridge/projects/visionimagevideoediting/ segmentation/grabcut.htm. Оригинальные изображения взяты из набора данных, который теперь вошел в состав набора Berkeley Segmentation Dataset (*http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/*).

Caltech 101

Это классический набор данных, содержащий фотографии объектов из 101 категории. Он применяется для тестирования алгоритмов распознавания объектов. Скачать можно со страницы *http://www.vision.caltech.edu/Image_Datasets/Caltech101/*.

База данных статических положений руки

Этот набор данных, подготовленный Себастьеном Марселем, выложен на странице *http://www.idiap.ch/resource/gestures/* вместе с несколькими другими наборами, содержащими изображения рук и жестов.

Наборы стереоизображений Мидлбери-колледжа

Эти наборы данных используются для тестирования алгоритмов работы со стереоизображениями. Его можно скачать со страницы *http://vision.middlebury.edu/stereo/data/*. Каждая стереопара сопровождается контрольными картами глубины для сравнения результатов.

ПРИЛОЖЕНИЕ В. Благодарности авторам изображений

В этой книге мы пользовались доступными всем желающим наборами данных и изображениями из различных веб-служб, все они перечислены в приложении Б. Мы высоко ценим заслуги исследователей, подготовивших эти наборы данных.

Некоторые изображения принадлежат самому автору. Вы можете использовать их на условиях лицензии Creative Commons Attribution 3.0 (CC BY 3.0) (*http://creativecommons.org/licenses/by/3.0/*), например, упомянув эту книгу в качестве источника.

Ниже перечислены эти изображения:

- изображение небоскреба Эмпайр Стейт Билдинг, встречающееся во многих примерах;
- слабоконтрастное изображение на рис. 1.7;
- примеры для сопоставления признаков на рис. 2.2, 2.5, 2.6 и 2.7;
- знак Рыбацкой пристани на рис. 9.6, 10.1 и 10.2;
- мальчик на вершине холма на рис. 6.4 и 9.6;
- изображение книги для калибровки на рис. 4.3;
- два изображения открытой книги издательства O'Reilly на рис. 4.4, 4.5 и 4.6.

В.1. Изображения с сайта Flickr

Мы воспользовались несколькими изображениями с сайта Flickr, публикуемыми по лицензии Creative Commons Attribution 2.0 Generic (CC BY 2.0) (*http://creativecommons.org/licenses/by/2.0/deed.en*). Выражаем благодарность фотографам.

Ниже перечислены изображения с сайта Flickr (указаны имена, использованные в примерах из книги, а не имена оригинальных файлов):

- billboard_for_rent.jpg от @striatic, *http://flickr.com/photos/striatic/21671910/*, использовано на рис. 3.2;
- blank_billboard.jpg от @mediaboytodd, *http://flickr.com/photos/23883605@N06/2317982570/*, использовано на рис. 3.3;
- beatles.jpg от @oddsock, *http://flickr.com/photos/oddsock/82535061/*, использовано на рис. 3.2, 3.3;
- turningtorso1.jpg от @rutgerblom, *http://www.flickr.com/photos/rutgerblom/2873185336/*, использовано на рис. 3.5;
- sunset_tree.jpg от @jpck, *http://www.flickr.com/photos/jpck/3344929385/*, использовано на рис. 3.5.

В.2. Прочие изображения

- Изображения лиц на рис. 3.6, 3.7 и 3.8 публикуются с разрешения Дж. К. Келлера. Аннотации глаз и рта принадлежат автору.
- Фотографии зданий Лундского университета на рис. 3.9, 3.11 и 3.12 взяты из набора данных, используемого группой математической обработки изображений из Лундского университета. Фотографировал, скорее всего, Магнус Оскарсон.
- Трехмерная модель игрушечного самолета на рис. 4.6 принадлежит Жиллю Трану (Creative Commons License By Attribution).
- Изображения тюрьмы Алькатрас на рис. 5.7 и 5.8 публикуются с разрешения Карла Олсона.
- Наборы шрифтовых данных на рис. 1.8, 6.2, 6.3, 6.7 и 6.8 публикуются с разрешения Мартина Солли.
- Изображения судоку на рис. 8.6, 8.7 и 8.8 публикуются с разрешения Мартина Бюрёда.

В.3. Иллюстрации

Иллюстрация эпиполярной геометрии на рис. 5.1 основана на рисунке Класа Джозефсона и адаптирована для этой книги.

ЛИТЕРАТУРА

[1] Herbert Bay, Tinne Tuytelaars, Luc Van Gool. SURF: Speeded up robust features. В сборнике *European Conference on Computer Vision*, 2006.

[2] Yuri Boykov, Olga Veksler, Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on PatternAnalysis andMachine Intelligence*, 23:2001, 2001.

[3] Gary Bradski, Adrian Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008.

[4] Martin Byröd. An optical Sudoku solver. В сборнике *Swedish Symposium on Image Analysis, SSBA. http://www.maths.lth.se/matematiklth/personal/byrod/papers/ sudokuocr.pdf*, 2007.

[5] Antonin Chambolle. Total variation minimization and a class of binary mrf models. В сборнике *Energy Minimization Methods in Computer Vision and Pattern Recognition*, Lecture Notes in Computer Science, pages 136–152. Springer Berlin / Heidelberg, 2005.

[6] T. Chan, L. Vese. Active contours without edges. *IEEE Trans. Image Processing*, 10(2):266–277, 2001.

[7] Chih-Chung Chang, Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Программное обеспечение доступно по адресу *http://www.csie.ntu.edu.tw/~cjlin/libsvm*.

[8] D. Cremers, T. Pock, K. Kolev, A. Chambolle. Convex relaxation techniques for segmentation, stereo and multiview reconstruction. В сборнике *Advances in Markov Random Fields for Vision and Image Processing*. MIT Press, 2011.

[9] Nello Cristianini, John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[10] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, стр. 363–370, 2003.

[11] M. A. Fischler, R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–95, 1981.

[12] C. Harris, M. Stephens. A combined corner and edge detector. В сборнике *Proceedings Alvey Conference*, стр. 189–192, 1988.

[13] R. I. Hartley, A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, второе издание, 2004.

[14] Richard Hartley. In defense of the eight-point algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:580–593, 1997.

[15] Berthold K. P. Horn, Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[16] Vladimir Kolmogorov, Ramin Zabih. What energy functions can be minimized via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:65–81, 2004.

[17] David G. Lowe. Object recognition from local scale-invariant features. В сборнике *International Conference on Computer Vision*, стр. 1150–1157, 1999.

[18] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[19] Bruce D. Lucas, Takeo Kanade. An iterative image registration technique with an application to stereo vision, стр. 674–679, 1981.

[20] Mark Lutz. *Learning Python*. O'Reilly Media Inc., 2009.

[21] Will McGugan. *Beginning Game Development with Python and Pygame*. Apress, 2007.

[22] F.Meyer. Color image segmentation. В сборнике *Proceedings of the 4th Conference on Image Processing and its Applications*, стр. 302–306, 1992.

[23] D. Nistér, H. Stewénius. Scalable recognition with a vocabulary tree. В сборнике *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, том 2, стр. 2161–2168, 2006.

[24] Travis E. Oliphant. *Guide to NumPy. http://www.tramy.us/numpybook.pdf*, 2006.

[25] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, R. Koch. Visual modeling with a hand-held camera. *International Journal of Computer Vision*, 59(3):207–232, 2004.

[26] Marc Pollefeys. Visual 3d modeling from images–tutorial notes. Technical report, University of North Carolina–Chapel Hill. *http://www.cs.unc.edu/~marc/tutorial.pdf*

[27] Carsten Rother, Vladimir Kolmogorov, Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23: 309–314, 2004.

[28] L. I. Rudin, S. J. Osher, E. Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D*, 60:259–268, 1992.

[29] Daniel Scharstein, Richard Szeliski. A taxonomy and evaluation of dense two frame stereo correspondence algorithms. *International Journal of ComputerVision*, 2001.

[30] Daniel Scharstein, Richard Szeliski. High-accuracy stereo depth maps using structured light. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003.

[31] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media, 2007.

[32] Jianbo Shi, Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22:888–905, August 2000.

[33] Jianbo Shi, Carlo Tomasi. Good features to track. In *1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, стр. 593–600, 1994.

[34] Noah Snavely, Steven M. Seitz, Richard Szeliski. Photo tourism: Exploring photo collections в сборнике 3d. In *SIGGRAPH Conference Proceedings*, стр. 835–846. ACM Press, 2006.

[35] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, Andrew W. Fitzgibbon. Bundle adjustment – a modern synthesis. В сборнике *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, ICCV '99, стр. 298–372. Springer-Verlag, 2000.

[36] A. Vedaldi, B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. *http://www.vlfeat.org/*, 2008.

[37] Deepak Verma, Marina Meila. A comparison of spectral clustering algorithms. Technical report, 2003.

[38] Luminita A. Vese, Tony F. Chan. A multiphase level set framework for image segmentation using the mumford and shah model. *International Journal of Computer Vision*, 50:271–293, December 2002.

[39] Paul Viola, Michael Jones. Robust real-time object detection. В сборнике *International Journal of Computer Vision*, 2001.

[40] Marco Zuliani. Ransac for dummies. Technical report, Vision Research Lab, UCSB, 2011.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.alians-kniga.ru.

Оптовые закупки: тел. **(499) 782-38-89**

Электронный адрес: **books@alians-kniga.ru**.

Ян Эрик Солем

Программирование компьютерного зрения на языке Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 19,5.
Тираж 500 экз.

Веб-сайт издательства: www.дмк.рф