O'REILLY®



学习OpenCV 3

（中文版）

Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library

[美] 安德里安·凯勒 (Adrian Kaehler)
加里·布拉德斯基 (Gary Bradski) 著

阿丘科技 刘昌祥 吴雨培 王成龙 崔玉芳 等译

清华大学出版社

学习OpenCV 3（中文版）

Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library

[美] 安德里安·凯勒 (Adrian Kaehler)
加里·布拉德斯基 (Gary Bradski) 著

阿丘科技 刘昌祥 吴雨培 王成龙 崔玉芳 等译

Beijing • (l • Tokyo O'REILLY®

ia, Inc.授权清华大学出版社出版

清华大学出版社
北京

内容简介

计算机视觉是在图像处理的基础上发展起来的新兴学科。OpenCV是一个开源的计算机视觉库，是英特尔公司资助的两大图像处理利器之一。它为图像处理、模式识别、三维重建、物体跟踪、机器学习和线性代数提供了各种各样的算法。

本书由OpenCV发起人所写，站在一线开发人员的角度用通俗易懂的语言解释了OpenCV的缘起和计算机视觉基础结构，演示了如何用OpenCV和现有的自由代码为各种各样的机器进行编程，这些都有助于读者迅速入门并渐入佳境，兴趣盎然地深入探索计算机视觉领域。

本书可作为信息处理、计算机、机器人、人工智能、遥感图像处理、认知神经科学等有关专业的高年级学生或研究生的教学用书，也可供相关领域的研究工作者参考。

Copyright © 2016 Adrian Kaehler and Gary Bradski. All rights reserved.
Authorized Simplified Chinese translation edition, by O'Reilly Media, Inc., is published by Tsinghua University Press, 2017. Authorized translation of the original English edition, 2016 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.
本书之英文原版由O'Reilly Media, Inc.于2016出版。
本中文简体翻译版由 O'Reilly Media, Inc.授权清华大学出版社于2018年出版。此翻译版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc.的许可。
版权所有，未经书面许可，本书的任何部分和全部不得以任何形式复制。

北京市版权局著作权合同登记　图字：01-2017-5006号
本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目（CIP）数据

学习OpenCV 3（中文版）/（美）安德里安·凯勒（Adrian Kaehler），（美）加里·布拉德斯基（Gary Bradski）著; 刘昌祥等译 --北京：清华大学出版社，2018
书名原文：Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library
ISBN 978-7-302-50418-4
I. ①学… II. ①安… ②加… ③刘… III. ①图像处理软件－程序设计 IV. ①TP391.41
中国版本图书馆CIP数据核字(2018)第124218号

责任编辑：文开琪
封面设计：Karen Montgomery，张健
责任校对：周剑云
责任印制：李红英
出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮政编码**：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62795954，jsjjc@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：178mm × 233mm　　**印　　张**：54.5　　**字　　数**：1046千字
版　　次：2018年7月第1版　　**印　　次**：2018年7月第1次印刷
定　　价：149.00元

产品编号：048245-01

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——*Wired*

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

译者序

计算机视觉是一个又新又酷的研究领域，而 OpenCV 是辅助研究计算机视觉最出色的开源工具。

1966 年，MIT 的明斯基[注 1] 给他的几个本科生萨斯曼[注 2] 指派了暑期作业：把相机连接到电脑上，并让电脑描述相机所看到的东西。[注 3] 他觉得在夏天结束之前学生们就会解决问题，结果半个世纪之后，我们依然在为此而努力。时至今日，计算机视觉领域虽然已经取得了诸多成果，但依然满足不了我们的好奇心，计算机视觉还能做什么，它的极限在哪里？如果你在今年的夏天收到了这样一份相同的暑假作业，你又会如何处理那个问题呢？

首先，得弄明白要解决的是什么问题。计算机视觉需要解决的任务可谓纷繁复杂，从区分目标和背景的背景分割（第 15 章）、追踪，到使用特征描述（第 16 章）以及机器学习方法（第 20 章和第 21 章）来进行的目标检测（第 22 章），从相机标定（第 18 章）到三维视觉（第 19 章），基于不同的目的，就有了不同的任务。这往往会让初学者感到慌乱。每时每刻，往往都有所谓的热点影响着视听。人脸识别技术，此时炙手可热，工业缺陷检测、三维视觉等技术也在日益升温。可是，没有哪一个方向永远只会上升不

注 1：Marvin Minsky（1927—2016），人工智能之父，美国工程院院士，美国科学院院士。1969 年图灵奖获得者，2001 年富兰克林奖章获得者，著作有《情感机器》和《心智社会》等。

注 2：Gerald Jay Sussman（1947— ），著名计算机科学家，MIT 教授，自由软件基金会创始人之一。

注 3：传说往往都比事实夸大一些，如果对当时那份暑假作业有兴趣，不妨查看 *http://publications.csail.mit.edu/ai/browse/0100browse.shtml*。

是一味地下降，与其朝三暮四，不如选择一个领域深扎下去，耐心与研究从来都是结伴而生的兄弟。

在决定要研究的方向以后，硬件条件也许又会成为很多人的拦路虎。虽然在 1966 年的夏天，那时候集成电路计算机已经开始普及，并且相对于晶体管计算机已经有了长足发展，但是低下的计算能力依然是遏制计算机技术发展的一个瓶颈，过于超前的想法可能确实难以实现。即使是今天，深度学习的初学者也可能会为没有性能足够强劲的计算机发愁。当然，我们说机器的性能固然重要，可是算法超然于硬件，那才是硬件的灵魂。1998 年，LeNet5 已经可以在只有几十兆内存（这样的硬件条件似乎和深度学习根本沾不上边）的机器上运行，并且被应用到了实际的项目中。诸多看起来非常耗费计算力的图像分析算法（第 12 章），也由于人们的巧妙智慧在工业界得到了应用。另一方面，如 IPP 等针对于指令集的优化，也让轮廓查找（第 14 章）、模板匹配（第 13 章），图像变换（第 11 章）和卷积（第 10 章）等操作速度有了质的提升。

所以现在，哪怕手里只有一台老旧的奔腾，也已经阻挡不了你研究计算机视觉的脚步。更幸运的是，今天的我们已经不必使用打孔带在 0 和 1 之间挣扎，甚至不必从最基础的图像结构开始。只需要简单的运行可执行文件，或者输入 apt-get 就可以得到 OpenCV，一个具备跨平台特性（第 9 章）且支持包括摄像机、视频文件、图像等多种类型输入（第 8 章）的计算式视觉开源库。它不仅提供最基础的图像结构（第 3 章和第 4 章），还支持多样的矩阵操作（第 5 章和第 7 章），甚至还有一定能力的绘图功能（第 6 章）。

如此这般，在 1966 年的夏天，看起来几乎不可能的任务，在如今很可能已经变成某个本科生的一次课程作业。当然，难度都是依据标准的不同而变化的，本科生用某些工具，以 80% 的正确率识别出一辆车，这都是值得竖起大拇指点赞的；自动驾驶的从业人员，对车辆的识别率达到了 99%，仍然不足以让所有人安心。不论繁简，这其中都蕴含着视觉工作者们无数的奉献。OpenCV，正是这一种奉献的集大成者，它就像是一座极大的宝库，既能让初学者从中获取到入门之道，也能让资深专家汲取到许多灵感。在 Learning OpenCV 巨大成功的十年之后，我们欣然接受了 Learning OpenCV3 的翻译工作。

尽管依然继承了 Learning OpenCV 之名，本书的内容已经和上一版本有了质的不同。作者重构了旧的章节，并将所有的例子都用符合 OpenCV 2.4 版本以上的风格重写，还增加了近十章的新章节。书中不可避免的涉及大量专业术语，为了便于读者日后进一步学习时，不对这些名词产生陌生感，我们尽最大努力做了保留。

本书的翻译工作也涉及到众多朋友的付出和努力，在此一并列出。本书前言及第 1 章、第 2 章、第 4 章由刘昌祥翻译，第 3 章由吴雨培翻译，第 5 章和第 18 章由崔玉芳翻译，第 6 章由董奭翻译，第 7 章由孙曦翻译，第 8 章由徐磊翻译，第 9 章由饶强翻译，第 10 章、第 12 章由何闻达翻译，第 11 章由李明磊、白涛翻译，第 13 章、第 15 章由田来翻译，

第 14 章吴昭翻译，第 16 章、第 17 章由王成龙翻译、第 19 章、第 22 章、第 23 章由岳嘉欣翻译，第 20 章、第 21 章由薛希哲翻译，附录由马文科翻译。部分人员参与了全书的校对工作，全书由刘昌祥定稿。本书译制过程中，文开琪女士做出了大量指导工作，郑慧伟、王岳谦、张辉、崔坤、孔涛、孙超等前辈和同事也提出了许多有价值的修改意见，阿丘科技全体技术团队为本书翻译提供了技术支持。

在近千页的原著翻译完成之后，笔者十分欣慰能参与并且主导这次视觉的远征。这本书的出版可以说是笔者人生的一个新的起点，也更希望它能成为更多人计算机视觉之路上的一个起点。感谢本书原作 Adrian Kaehler 和 Gary Bradski 在这条传道之路上的坚持，巨人燃起篝火，追随者从他们的薪火中获取了火种，撒播到懵懂的黑暗中，希望能够聚起更多的光明。

计算机视觉发展极其迅速，罕有人士能对其众多分支领域均有精深的理解。笔者自认才疏学浅，更兼时间精力有限，书中错谬之处在所难免，诚望读者朋友不吝告知，不胜感激。

译者团队代表：刘昌祥

2017 年 11 月 24 日

阿丘科技

阿丘科技是一家以机器学习、3D 视觉和机器人技术为核心的高科技企业，拥有业内领先的机器人 3D 视觉技术、机器学习算法及标准行业解决方案。

人工智能，已经成为全球技术发展、应用拓展的前沿领域，也是引领未来的一枚关键的棋子。计算机视觉作为人工智能的关键应用点之一，近年来兴起的创业公司主要集中在人脸识别、无人驾驶、增强现实等领域。而阿丘科技独辟蹊径，将人工智能技术与机器人结合，将其应用于工业自动化领域，并在短短一年内在多个场景下落地，取得了出色的成果。

相较于传统的机器视觉，阿丘科技将深度学习用于工业视觉检测，在缺陷检测分类等场景下具有突出优势。而三维计算机视觉与机器人的结合，更是极大地扩展了工业机器人的应用场景。凭借国际顶尖的技术人才，领先的计算机视觉、机器人技术，阿丘科技走在了智能工业视觉技术的前列。随着人工智能和智能制造行业的兴起，我司将助力中国工业自动化以及智能化的发展。

尽管由于某些原因，OpenCV 并没有在正式版本中提供自身完备的深度学习工具（我们很高兴这一点在 3.3 版本中有了巨大改善），但是 OpenCV 作为从事机器视觉应用技术开发的必备工具，同样，也是阿丘科技技术开发的基础工具之一，有着不可替代的作用。在很多固定场景下，基于 OpenCV 等视觉工具的开发的算法依然具有极大的应用潜力。尤其是对于初学者而言，我们还是非常推荐使用 OpenCV 完成一些视觉项目，而不是直接用深度学习构建空中楼阁。我司能够在一年内推出经受复杂应用场景检验、市场普遍认可的产品，有一大份功劳，也归属于 OpenCV 对项目完成和开发研究的推动。高效、精准、便捷等诸多特质，使得 OpenCV 在学术研究和商业应用中占据了重要地位。

阿丘科技致力于构建领先的智能机器人视觉平台，以视觉为切入点，将 AI 与机器人结合。创始团队源自清华大学计算机系人工智能实验室，聚集了来自清华、CMU 等全球高等院校的顶尖人才，以及工业机器人和自动化资深行业从业者。我们期待，能有更多热爱视觉、工业自动化和机器人技术的朋友加入阿丘科技，共同去探索和扩展机器人应用的边界！

前言

这本书提供了C++开源计算机视觉库（OpenCV）3.0版本的工作指南，介绍了计算机视觉相关领域的通用背景，以便读者更有效地使用OpenCV。

本书的目标

计算机视觉在以下四个趋势的引导下，已经成为一个快速发展的领域：

- 智能手机的出现让数以百万计的用户都拥有了相机
- 互联网和搜索引擎聚集了海量的图像和视频数据
- 计算资源变成一种廉价的商品
- 视觉算法发展得更加成熟（由于深度神经网络的出现，OpenCV也正在逐渐支持这方面技术，详情可以参考opencv_contrib中的*dnn*）

OpenCV通过帮助成千上万的视觉工作者去做更多富有创造性的工作，在图像处理领域扮演着重要的角色。由于OpenCV 3.x的存在，不管你是学生还是研究人员，是专家还是初学者，都可以快速建立应用，并且在OpenCV所提供的基于C++的多平台视觉基础架构上实现一个跨越性的发展。

本书的目标如下：

- 通过详细讲述函数的惯用用法以及正确用法，成为一本更好的OpenCV经典参考文档
- 帮助读者对各种计算机视觉算法的工作原理有一个基础的理解
- 培养读者，使其知道什么算法工具可以使用并且应该在什么时候使用

- 提供给读者许多有效的代码，提升其应用计算机视觉和机器学习算法的开发效率
- 针对问题代码给读者提供建议，教读者如何解决一些简单或复杂的问题

本书的写作方式是力求帮助读者快速在计算机视觉领域做一些有意思的事情，因此直观地解释了算法是如何工作的，可以指导读者设计和调试计算机视觉程序，还能够使计算机视觉和机器学习算法的形式化描述更容易理解和记忆。

本书受众

本书包含描述、工作代码示例和OpenCV 3.x库中包含的C++计算机视觉工具说明。因此，本书对各种不同类型的用户都有所帮助。

专业人士和开发者

对于需要快速设计原型或实现专业计算机视觉系统的专家而言，示例代码提供的框架可以帮助他们在此基础上进行快速开发。我们对算法的描述可以快速教会或者提醒读者如何使用它们。由于OpenCV 3.x的设计基于位于硬件加速层（HAL）之上，所以它所实现的算法都可以高效运行，并且能够无缝应用各种硬件平台的诸多加速特性。

学生

这是我们希望能广泛应用于学校的教程。直观的解释、详细的文档以及示例代码将能够帮助你在计算机视觉之路上更快地前进，做更有趣的项目，最终为这个领域开拓新的研究方向。

教师

计算机视觉是一个快速发展的领域。我们发现，让学生快速浏览容易读的文章并且教师在必要的地方进行说明，再辅之以现阶段的论文或讲座，是一种非常高效的授课方法。与此同时，学生可以提前开始课程设计，并尝试做一些更有挑战性的任务。

爱好者

计算机视觉超有趣的，让我们一起深入探索它吧。

我们致力于为读者提供丰富的指导、文档以及代码来快速建立实时的视觉应用。

本书不是什么

这本书不是正式的论文，我们的确在很多地方讨论了数学上的细节[注1]，但这都是为了建

注1：通常，这些部分都有警告标识，以提示用户可以跳过这些内容。

立一种基于此的更深层次的直觉以及阐明算法中所构建的假设之含义。在这里，我们没有尝试进行一些更加正式的数学解释，因为过多的数学解释可能会使一些人觉得枯燥。

这本书本质上来说更偏向于“应用”，它将作为一个基础的帮助，但是并不着力于介绍某个具体的计算机视觉领域分类（比如医学图像或者遥感图像分析）。

也就是说，我们相信通过阅读这里的解释，学生不仅能更好地学习理论，而且还能记住更多、更久。因此，这本书将是一个很好的理论课程的辅助教材，并且很适合入门课程或以项目为中心的课程使用。

本书例程

本书所有例程都基于OpenCV 3.x。这些代码在Linux，Windows 以及 OS X都可以运行。OpenCV也有充足的对安卓和IOS 的支持。本书所有代码都可以在本书网站（*http://bit.ly/learningOpenCV3*）获取，OpenCV的源码可以在GitHub（*https://github.com/opencv/opencv*）获取，OpenCV的预编译版本可以在SourceFroge（*https://sourceforge.net/projects/opencvlibrary/*）获取。

OpenCV现在也处于开发过程中，每个季度都有新的官方版本发布。如果希望得到最新版本，可以从前述的GitHub网站得到代码更新。OpenCV的官方网站是*http://opencv.org*，如果是开发人员，还可以浏览它的维基页面：*https://github.com/opencv/opencv/wiki*。

预备知识

在大多数情况下，读者只需要了解如何使用C++编程。本书许多数学相关的段落都不是必须的并且也做出了相应的标记。本书需要的数学基础包括简单的线性代数以及基础矩阵运算，并且假设读者了解最小二乘优化、高斯分布的基础知识、贝叶斯法则以及一些简单方程的衍生和变化。

本书对数学的介绍是为了支持建立一种对算法的直觉。读者可以跳过数学部分以及算法描述部分，只阅读函数解释以及代码示例，也可以构建视觉应用并且使其正常工作。

如何使用本书

本书并不一定要按照章节顺序阅读，它可以作为一本用户手册：需要的时候，可以用来查询函数，如果你希望了解它的工作原理，请阅读该函数相关的描述。本书的意图在于提供一本指南。它提供对计算机视觉的基本理解以及如何在何时使用选择的算法。

本书的写作目的是作为计算机视觉的本科或研究生课程的辅助教材或教科书。采取的基本策略是先让学生能够快速阅读课本，然后在其他教科书或者文章中用更正式的章节进行补充阅读。每一章最后都配有练习题，可以帮助测试学生对知识的掌握程度并培养更多图像处理方面的意识。

可以采用如下任意一种方法来阅读本书。

抓取重点

先阅读第1~5章，之后只阅读你所需要的章节。除了第18章和第19章（涵盖了相机校正和双目视觉）以及第20章~第22章（涵盖机器学习），这本书并不需要按照章节顺序阅读。以项目为导向的学生和研发人员可以用这种方式阅读。

精读

坚持每周阅读两章，这样你就可以在11周之内读完第1章~第22章（第23章很短）。从项目开始，深入到具体研究领域，适当地使用额外的文献和论文作为补充。

速成

在可以理解的基础上尽可能快地浏览本书，阅读第1章~第23章，之后开始项目，并使用相关文献和论文在某一个领域进行更深入的研究。这也许是专业人员的一个选择，但也可能适合更高级的计算机视觉课程。

第20章对机器学习进行了简要的介绍，在此之后的第21章和第22章将对机器学习算法及其在OpenCV中的实现和实战进行更详细的解释。当然，机器学习是目标识别以及计算机视觉的重要组成部分，相关知识非常丰富以至于可以独立成书。专业人员会发现这本书是进一步探索的一个合适的起点，或者也可以仅仅对这一部分的OpenCV代码进行深入研究。在OpenCV 3.x中，机器学习的接口已经大大简化和统一。

我们喜欢的计算机视觉教学方式是，在学生基本能够理解算法工作原理的基础上尽可能快速展开教学，然后让他们进一步加深理解，进而开始学习有意义的课程设计，同时通过相关领域的文献和论文让学生对所选择的领域拥有更深刻和系统的认识。这种方法适用于半学期、一学期以及两学期的课程。学生可以快速开始，在对视觉项目仅仅有一般认识的时候就开始编码，等到进行更富挑战性和更耗时的工作的时候，指导教师可以提供适当的帮助。

而对于计划时间更长的课程来说，OpenCV项目本身就可以成为项目管理的指导。先建立工作系统，然后用更多的知识、细节和研究进一步加以完善。这类课程的目的是让每个项目都有一个值得自己发表的地方并且其中一些真的能够发展壮大起来。OpenCV

3.x的代码框架，Buildbots、GitHub的使用，pull request[译注1]，单元测试和回归测试，文档，这些对商业软件架构来说都是很好的示例，值得参考。

本书约定

本书中使用下列约定。

斜体（*Italic*）

指示新的术语、URL、电子邮件地址、文件名、拓展名、路径名、文件夹和Unix工具。

固定宽度（Constant Width）

指示指令、选项、选择、变量、属性、键、函数、类型、类、命名空间、方法、模块、参数、值、对象、时间、事件句柄、XML标记、HTML标记、文件的内容或者指令的输出。

加粗固定宽度（**Constant width bold**）

指示指令以及其他应该由用户输入的文本。也用来着重显示代码片段。

斜体固定宽度（*Constant Width Italic*）

指示应当由用户提供的值取代的文本。

[…]

指示参考目录的引用。



这个标识表示建议或者一般性的笔记。



这个标识表示警告或需要注意的地方。

译注1：拉取请求（pull request）是Github上的功能，该功能在模块修改完成后，将会专门针对修改的模块建立讨论并进行整合修改。

使用代码例程

补充材料（代码例程和练习等）都可以在*https://github.com/oreillymedia/Learning-OpenCV-3_examples*下载。

OpenCV不管是对于商业还是学术使用都是免费的，本书的代码我们也使用了相同的许可。不管是作为作业、研究还是商业产品，都可以使用。我们会非常感谢你在自己的文章中引用这本书，不过这不是必须的。引用通常包含标题、作者、出版商以及ISBN。比如："*Learning OpenCV 3* by Adrian Kaehler and Gary Bradski (O'Reilly)。Copyright 2017 Adrian Kaehler, Gary Bradski, 978-1-491-93799-0。"

除了了解如何帮助你完成你的作业（最好保密），我们很乐意听到你将计算机视觉用于学术研究或者课堂教学以及使用OpenCV来帮助你开发商业产品。同样，这不是必须的，我们始终欢迎你们的来信。

O'Reilly Safari



*Safari*是企业、政府、教育工作者和个人的会员制培训平台。该平台的成员可以获取上千种图书、培训视频、学习路线、交互教程以及来自超过250个出版商的播放列表，包括O'Reilly Media，Harvard Business Review，Prentice Hall Professional，Addison-Wesley Professional，Microsoft Press，Sams，Que，Peachpit Press，Adobe，Focal Press，Cisco Press，John Wiley & Sons，Syngress，Morgan Kaufmann，IBM Redbooks，Packt，Adobe Press，FT Press，Apress，Manning，New Riders，McGraw-Hill，Jones & Bartlett，Course Technology以及其他一些出版商。

更多信息请访问*http://oreilly.com/safari*。

联系方式

请把你对本书的意见和疑问发给出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

我们还有一个服务于本书的网站。那里有示例的列表以及未来版本的计划。点击*http://bit.ly/learningOpenCV3*了解这些信息。

评论或是询问这本书技术相关的问题，请发送邮件到*bookquestions@oreilly.com*。

关于本书、课程、会议以及新闻更多信息，请访问我们的网站：*http://www.oreilly.com*

在FaceBook上找到我们：*http://facebook.com/oreilly*

在Twitter上关注我们：*http://twitter.com/oreillymedia*

在YouTube上观看我们：*http://www.youtube.com/oreillymedia*

致谢

一个长期的开源项目通常经历很多人的加入和离开。每个人都从不同方面做出了贡献。这个库的贡献列表已经太长而不适合在这里展示，但是可以访问与OpenCV绑定的文档*/opencv/docs/HTML/Contributors/doc_contributors.html*，查看贡献列表。

感谢对OpenCV的帮助

英特尔是OpenCV诞生的地方，感谢英特尔在这个库发起和成长的过程中所做出的巨大支持。时光荏苒，英特尔始终在为OpenCV筹备竞赛并做出贡献。英特尔还捐赠了内置的性能原语代码，为英特尔的架构提供了无缝加速。感谢英特尔所做的这一切。

谷歌一直是OpenCV开发者的稳定资助者，它为谷歌编程之夏（Google Summer of Code，GSoC）项目的Opencv实习生提供了赞助。这场活动贡献了许多杰出的工作成果。柳树车库（Willow Garage）公司提供了资金，致力于将OpenCV 2.0版本提升到3.0版本成果。在这些日子里，计算机视觉公司Itseez（最近被英特尔收购）提供了大量的工程师支持并承担了网站维护工作。英特尔已经表示继续支持维护网站（谢谢）。

在软件层面，特别需要感谢一些机构和个人，尤其是（英特尔）俄罗斯的软件团队。其中最主要的是俄罗斯首席程序员Vadim Pisarevsky（瓦迪姆·彼萨里夫斯）。他是这个库最大的独立贡献者。Vadim还在项目引爆前的萧条时期管理并且发展了OpenCV。如果要把有些人称为OpenCV的英雄，那么一定非他莫属。他的技术见解在我们写书的时候

也起到了巨大的帮助，在管理方面他也受到了Victor Eruhimov的支持，俄罗斯视觉公司Itseez的联合创始人，现在是Itseez3D（已被英特尔收购）的CEO。

在每周会议上，有几个人一直在帮助管理库，他们是Grace Vesom，Vincent Rabaud，Stefano Fabri，当然还有Vadim Pisarevsky。这些会议可以在*https://github.com/opencv/opencv/wiki/Meeting_notes*看到。

在此期间，许多人都为OpenCV库做出了贡献，最近一些贡献者分别是Dinar Ahmatnurov，Pablo Alcantarilla，Alexander Alekhin，Daniel Angelov，Dmitriy Anisimov，Anatoly Baksheev，Cristian Balint，Alexandre Benoit，Laurent Berger，Leonid Beynenson，Alexander Bokov，Alexander Bovyrin，Hilton Bristow，Vladimir Bystritsky，Antonella Cascitelli，Manuela Chessa，Eric Christiansen，Frederic Devernay，Maria Dimashova，Roman Donchenko，Vladimir Dudnik，Victor Eruhimov，Georgios Evangelidis，Stefano Fabri，Sergio Garrido，Harris Gasparakis，Yuri Gitman，Lluis Gomez，Yury Gorbachev，Elena Gvozdeva，Philipp Hasper，Fernando J. Iglesias Garcia，Alexander Kalistratov，Andrey Kamaev，Alexander Karsakov，Rahul Kavi，Pat O'Keefe，Siddharth Kherada，Eugene Khvedchenya，Anna Kogan，Marina Kolpakova，Kirill Kornyakov，Ivan Korolev，Maxim Kostin，Evgeniy Kozhinov，Ilya Krylov，Laksono Kurnianggoro，Baisheng Lai（赖百胜），Ilya Lavrenov，Alex Leontiev，Gil Levi，Bo Li（李波），Ilya Lysenkov，Vitaliy Lyudvichenko，Bence Magyar，Nikita Manovich，Juan Manuel Perez Rua，Konstantin Matskevich，Patrick Mihelich，Alexander Mordvintsev，Fedor Morozov，Gregory Morse，Marius Muja，Mircea Paul Muresan，Sergei Nosov，Daniil Osokin，Seon-Wook Park，Andrey Pavlenko，Alexander Petrikov，Philip aka Dikay900，Prasanna，Francesco Puja，Steven Puttemans，Vincent Rabaud，Edgar Riba，Cody Rigney，Pavel Rojtberg，Ethan Rublee，Alfonso Sanchez-Beato，Andrew Senin，Maksim Shabunin，Vlad Shakhuro，Adi Shavit，Alexander Shishkov，Sergey Sivolgin，Marvin Smith，Alexander Smorkalov，Fabio Solari，Adrian Stratulat，Evgeny Talanin，Manuele Tamburrano，Ozan Tonkal，Vladimir Tyan，Yannick Verdie，Pierre Emmanuel Viel，Vladislav Vinogradov，Pavel Vlasov，Philipp Wagner，Yida Wang（王一达），Jiaolong Xu（许骄龙），Marian Zajko和Zoran Zivkovic。

这里（*https://github.com/opencv/opencv/wiki/ChangeLog*）还列出了其他的贡献者。最后，Arraiy现在也在协助维护OpenCV.org（免费开源的代码库）。

感谢对本书提供帮助的人

在准备这本书和这本书的前一个版本时，我们得到了《纽约时报》科技记者约翰·马

尔科夫（John Markoff）[译注1]的大力帮助，感谢他的鼓励、不厌其烦的指导以及多年所累积的一线写作经验。我们还要感谢O'Reilly（奥莱利）的许多编辑，特别是Dawn Schanafelt（道恩·施纳菲尔特），在叛逆的作者准备创业的时候，他满怀耐心地继续跟进本书相关工作。这本书是一个由OpenCV 2.x到3.x到长期项目。非常感谢O'Reilly始终站在我们这一边。

Adrian致谢

在第1版（*Learning OpenCV*）的写作过程中，我邀请一些优秀的教师来协助我，使得本书工作变成了可能。在这些年，从各位那里获得的指导变得愈发珍贵。我对他们每个人都怀有诚挚的谢意。我还特别想在这份感谢名单上加上我的导师汤姆·汤贝雷诺[译注2]，我欠他非常多，在他的记忆里，我会把我的贡献献给这本书。他是一个具有非凡智慧的人，我很骄傲自己有机会追随他的足迹。最后，深深感谢OpenCV社区，感谢你们对第1版的欢迎，感谢你们耐心等待，这一版历经波折总算最终完成。

这一版花了很长时间才完成，在这几年里，我有幸与数十家不同的公司合作，担任顾问并进行技术研发，先后作为董事会的成员、顾问团的成员、技术总监、研发人员以及始创人。我有幸看到并热爱着研发的方方面面。我在Applied Minds公司很多年，在那里创建并且运行我们的机器人部门，更准确地说是Applied Minds子公司。我很欣慰地看到OpenCV是许多优秀项目的核心，涉及领域包括医疗、农业、航空、国防和国土安全。我同样高兴地发现这本书的第1版已经成为几乎所有这些机构中的案头必备。Gary和我用来构建斯坦利（Stanley）的技术已经成为无数项目的一部分，其中至少包含许多正在研发的自动驾驶项目。它们其中的某一个，或许所有都是为了改变或改善无数人的日常生活而准备的，成为其中的一员是多么让人开心啊！在这些年我遇到过无数惊人的故事——有关于第1版如何在人们所工作或教学的领域让他们受益，帮助他们建立事业，帮助他们完成巨大的成就——这些事情成为快乐和奇迹的源泉。我希望这本新书将继续为你们所有人服务，并激励和支持新一代科学家、工程师和发明家前行。

在本书的最后一章，我们开始了在机器人、人工智能、视觉和其他领域工作的新篇章。就我个人而言，我深深地感激那些引领和激励我的人：教师、导师以及相关书籍的作

译注1：被誉为“硅谷独家大王”，对互联网的发展富有洞察力和敏锐度，长期专注于对机器人和人工智能领域进行报告。他是2013年普利策奖的得主，代表作有《与机器人共舞》。

译注2：Tom Tombrello（1936—2014），从上世纪60年代开始研究核反应。1998—2008年间主持物理、数学和天文系的工作。他帮助创建了物理11课程，旨在鼓励新生突破常规思维，勇于创新和想象。这堂课的确激发了许多年轻人，包括埃隆·马斯克的明星前任莱利。

者。我希望我们这本书的新版本能够让人在他们自己的生活中迈出重要的一步，希望看到你们迈向成功！

Gary致谢

我在1999年创立了OpenCV项目，目的是能够向所有人提供我当时只有在顶级实验室里才能见到的基础框架，并借此来加速计算机视觉和人工智能技术的发展。尽管很少有目标能够和预期一样实现，但我庆幸这一计划在时隔17年后终于实现了。完成这一目标还得归功于多方的帮助，有太多的朋友和贡献者，我没办法在此一一感谢[注1]。但我还是希望在这里特别感谢我在英特尔开始工作时所在的俄罗斯研究团队，它们造就了一个成功的计算机视觉公司（Itseez.com），并最终被英特尔收购。我们作为同事开始工作并建立了深厚的友谊。

由于家中有三个十几岁的孩子，我的妻子Sonya Bradski（桑雅·布拉德斯基）做了更多的工作来支持我写作本书，对于她，我的感谢与爱难以言表。我也一样爱我的孩子，但要说他们加速了我这本书的出版，委实说不出口:)。

我从帮助建立Industrial Perception公司（2013年被谷歌收购）的时候就萌生了创作第2版的想法。从此以后，写书的工作就不定期地在周末和深夜进行。不知不觉，现在已经是2016年，真是时光飞逝如电啊！我在第23章结束时所做的一些思考，灵感来源于我亲自参与过的机器人项目，一个由柳树车库制造的叫PR2的双臂机器人以及斯坦福的斯坦利（Stanley）项目，这个机器人赢得了DARPA机器人挑战赛200万美元的大奖。

读者朋友们，在我们结束本书创作之际，由衷地希望在创业公司、实验室、学术网站、会议、车间、风险投资以及正在研发的很酷的项目中看到你们的身影。然后轻松地互相致意，谈一谈你正在做的新东西。我发起了OpenCV，为了共同的利益来支持并加速计算机视觉以及人工智能技术，接下来要看你们的了。我们生活在一个充满创意的世界里，一个人可以创造一个罐子，下一个人就能把这个罐子变成一个鼓，如此等等，都是创造！用OpenCV来创造一些可以造福于人类的东西吧！

注2：　OpenCV项目现在有许多贡献者，可以在*https://github.com/opencv/opencv/wiki/ChangeLog*查阅。独立的*opencv_contrib*项目也得到了许多新算法和新程序。

目录

译者序 xvii

前言 xxi

第1章 概述 1

什么是OpenCV 1

OpenCV怎么用 2

什么是计算机视觉 3

OpenCV的起源 6

OpenCV的结构 7

使用IPP来加速OpenCV 8

谁拥有OpenCV 9

下载和安装OpenCV 9

安装 9

从Git获取最新的OpenCV 12

更多的OpenCV文档 13

提供的文档 13

在线文档和维基资源 13

OpenCV贡献库 15

下载和编译Contributed模块 16

可移植性 16

小结 17

练习 17

第2章 OpenCV初探 19

头文件 19

资源 20

第一个程序：显示图片 21

第二个程序：视频 23

跳转 24

简单的变换 28

不那么简单的变换 30

从摄像头中读取 32

写入AVI文件 33

小结 34

练习 35

第3章 了解OpenCV的数据类型 37

基础知识 37

OpenCV的数据类型 37

基础类型概述 38

深入了解基础类型 39

辅助对象 46

工具函数 53

模板结构 60

小结 61

练习 61

第4章 图像和大型数组类型 63

动态可变的存储 63

cv::Mat类N维稠密数组 64

创建一个数组 65

独立获取数组元素 69

数组迭代器NAryMatIterator 72

通过块访问数组元素 74

矩阵表达式：代数和cv::Mat 75

饱和转换 77

数组还可以做很多事情 78

稀疏数据类cv::SparesMat 79
访问稀疏数组中的元素 79
稀疏数组中的特有函数 82
为大型数组准备的模板结构 83
小结 85
练习 86

第5章 矩阵操作 87
矩阵还可以做更多事情 87
cv::abs() 90
cv::add() 91
cv::addWeighted() 92
cv::bitwise_and() 94
cv::bitwise_not() 94
cv::bitwise_or() 94
cv::bitwise_xor() 95
cv::calcCovarMatrix() 95
cv::cartToPolar() 97
cv::checkRange() 97
cv::compare() 98
cv::completeSymm() 99
cv::convertScaleAbs() 99
cv::countNonZero() 100
cv::Mat cv::cvarrToMat() 100
cv::dct() 101
cv::dft() 102
cv::cvtColor() 103
cv::determinant() 106
cv::divide() 106
cv::eigen() 106
cv::exp() 107
cv::extractImageCOI() 107
cv::flip() 108
cv::gemm() 108

cv::getConvertElem()和cv::getConvertScaleElem() 109
cv::idct() 110
cv::inRange() 110
cv::insertImageCOI() 111
cv::invert() 111
cv::log() 112
cv::LUT() 112
cv::Mahalanobis() 113
cv::max() 114
cv::mean() 115
cv::meanStdDev() 116
cv::merge() 116
cv::min() 116
cv::minMaxIdx() 117
cv::minMaxLoc() 118
cv::mixChannels() 119
cv::mulSpectrums() 120
cv::multiply() 121
cv::mulTransposed() 121
cv::norm() 122
cv::normalize() 123
cv::perspectiveTransform() 125
cv::phase() 125
cv::polarToCart() 126
cv::pow() 126
cv::randu() 127
cv::randn() 127
cv::repeat() 129
cv::scaleAdd() 129
cv::setIdentity() 130
cv::solve() 130
cv::solveCubic() 131
cv::solvePoly() 132
cv::sort() 132

cv::sortIdx()....133
cv::split()....133
cv::sqrt()....134
cv::subtract()....135
cv::sum()....135
cv::trace()....135
cv::transform()....136
cv::transpose()....136
小结....137
练习....137

第6章 绘图和注释....139
绘图....139
艺术线条和填充多边形....140
字体和文字....146
小结....148
练习....148

第7章 OpenCV中的函数子....151
操作对象....151
主成分分析（cv::PCA）....151
奇异值分解cv::SVD....154
随机数发生器cv::RNG....157
小结....160
练习....160

第8章 图像、视频与数据文件....163
HighGUI模块：一个可移植的图形工具包....163
图像文件的处理....164
图像的载入与保存....165
关于codecs的一些注释....167
图片的编码与解码....168
视频的处理....169
使用cv::VideoCapture对象读取视频流....169
使用cv::VideoWriter对象写入视频....175

数据存储 176
cv::FileStorage的写入 177
使用cv::FileStorage读取文件 179
cv::FileNode 180
小结 183
练习 183

第9章 跨平台和Windows系统 187
基于Windows开发 187
HighGUI原生图形用户接口 188
通过Qt后端工作 199
综合OpenCV和全功能GUI工具包 209
小结 222
练习 222

第10章 滤波与卷积 225
概览 225
预备知识 225
滤波、核和卷积 225
边界外推和边界处理 227
阈值化操作 230
Otsu算法 233
自适应阈值 233
平滑 235
简单模糊和方框型滤波器 236
中值滤波器 238
高斯滤波器 239
双边滤波器 240
导数和梯度 242
索贝尔导数 242
Scharr滤波器 244
拉普拉斯变换 245
图像形态学 246

膨胀和腐蚀 247
通用形态学函数 250
开操作和闭操作 251
形态学梯度 254
顶帽和黑帽 256
自定义核 258
用任意线性滤波器做卷积 259
用cv::filter2D()进行卷积 259
通过cv::sepFilter2D使用可分核 260
生成卷积核 260
小结 262
练习 262

第11章 常见的图像变换 267

概览 267
拉伸、收缩、扭曲和旋转 267
均匀调整 268
图像金字塔 269
不均匀映射 273
仿射变换 274
透视变换 279
通用变换 282
极坐标映射 282
LogPolar 283
任意映射 287
图像修复 287
图像修复 288
去噪 289
直方图均衡化 292
cv::equalizeHist()用于对比均衡 294
小结 295
练习 295

第12章 图像分析 297

概览 297

离散傅里叶变换 297

cv::dft()离散傅里叶变换 298

cv::idft()用于离散傅里叶逆变换 300

cv::mulSpectrums()频谱乘法 300

使用傅里叶变换进行卷积 301

cv::dct()离散余弦变换 303

cv::idct()离散余弦逆变换 304

积分图 304

cv::integral()标准求和积分 306

cv::integral()平方求和积分 306

cv::integral()倾斜求和积分 307

Canny边缘检测 307

cv::Canny() 309

Hough变换 309

Hough线变换 309

Hough圆变换 313

距离变换 316

cv::distanceTransform()无标记距离变换 317

cv::distanceTransform()有标记距离变换 317

分割 318

漫水填充 318

分水岭算法 322

Grabcuts算法 323

Mean-Shift分割算法 325

小结 326

练习 326

第13章 直方图和模板 329

OpenCV中直方图的表示 331

cv::calcHist()：从数据创建直方图 332

基本直方图操作 334

直方图归一化 334

直方图二值化.....335
找出最显著的区间.....335
比较两个直方图.....337
直方图用法示例.....339
一些复杂的直方图方法.....342
EMD距离.....342
反向投影.....347
模板匹配.....350
方差匹配方法（cv::TM_SQDIFF）.....351
归一化方差匹配方法（cv::TM_SQDIFF_NORMED）.....352
相关性匹配方法（cv::TM_CCORR）.....352
归一化的互相关匹配方法（cv::TM_CCORR_NORMED）.....352
相关系数匹配方法（cv::TM_CCOEFF）.....352
归一化的相关系数匹配方法（cv::TM_CCOEFF_NORMED）.....352
小结.....355
练习.....355
第14章 轮廓.....359
轮廓查找.....359
轮廓层次.....360
绘制轮廓.....364
轮廓实例.....365
另一个轮廓实例.....366
快速连通区域分析.....368
深入分析轮廓.....370
多边形逼近.....370
几何及特性概括.....372
几何学测试.....377
匹配轮廓与图像.....378
矩.....378
再论矩.....380
使用Hu矩进行匹配.....383
利用形状场景方法比较轮廓.....384

小结.....388
练习.....389
第15章 背景提取.....391
背景提取概述.....391
背景提取的缺点.....392
场景建模.....392
像素.....393
帧间差分.....396
平均背景法.....397
累计均值，方差和协方差.....403
更复杂的背景提取方法.....410
结构.....413
进行背景学习.....414
存在移动的前景物体时进行背景学习.....417
背景差分：检测前景物体.....418
使用码书法的背景模型.....419
关于码书法的其他想法.....419
使用连通分量进行前景清理.....420
小测试.....423
两种背景方法的对比.....425
OpenCV中的背景提取方法的封装.....425
cv::BackgroundSubstractor基类.....426
KB方法.....427
Zivkovic方法.....428
小结.....431
练习.....431
第16章 关键点和描述子.....433
关键点和跟踪基础.....433
角点检测.....434
光流简介.....437
Lucas-Kanade稀疏光流法.....438

广义关键点和描述符.....448
光流，跟踪和识别.....450
OpenCV一般如何处理关键点和描述符.....451
核心关键点检测方法.....461
关键点过滤.....497
匹配方法.....499
结果显示.....505
小结.....508
练习.....508

第17章 跟踪.....511
跟踪中的概念.....511
稠密光流.....512
Farneback多项式扩展算法.....513
Dual TV-L[1]模型.....515
简单光流算法.....519
Mean-Shift算法和Camshift 追踪.....522
Mean-Shift算法.....522
Camshift.....526
运动模板.....526
估计.....533
卡尔曼滤波器.....534
扩展卡尔曼滤波器简述.....549
小结.....551
练习.....551

第18章 相机模型与标定.....553
相机模型.....554
射影几何基础.....556
Rodrigues变换.....558
透镜畸变.....559
标定.....562
旋转矩阵和平移向量.....563
标定板.....566

单应性.....572
相机标定.....576
矫正.....587
矫正映射.....587
使用cv::convertMaps()在不同表示方式之间转换矫正映射.....588
使用cv::initUndistortRectifyMap()计算矫正映射.....589
使用cv::remap()矫正图像.....591
使用cv::undistort()进行矫正.....591
使用cv::undistortPoints()进行稀疏矫正.....591
与标定结合.....592
小结.....595
练习.....596

第19章 投影与三维视觉.....599

投影.....600
仿射变换与透视变换.....601
鸟瞰图变换实例.....602
三维姿态估计.....606
单摄像机姿态估计.....607
立体成像.....609
三角测量.....610
对极几何.....613
本征矩阵和基本矩阵.....615
计算极线.....624
立体校正.....624
立体校正.....628
立体匹配.....638
立体校正、标定和对应的示例代码.....650
来自三维重投影的深度映射.....657
来自运动的结构.....659
二维与三维直线拟合.....659
小结.....662
练习.....662

第20章 机器学习基础.....665
什么是机器学习.....665
训练集和测试集.....666
有监督学习和无监督学习.....667
生成式模型和判别式模型.....669
OpenCV机器学习算法.....669
机器学习在视觉中的应用.....671
变量的重要性.....673
诊断机器学习中的问题.....674
ML库中遗留的机器学习算法.....678
K均值.....679
马氏距离.....684
小结.....687
练习.....687
第21章 StatModel：OpenCV中的基准学习模型.....689
ML库中的常见例程.....689
训练方法和cv::ml::TrainData的结构.....691
预测.....697
使用cv::StatModel的机器学习算法.....698
朴素贝叶斯分类器.....699
二叉决策树.....703
Boosting方法.....716
随机森林.....721
期望最大化算法.....725
K近邻算法.....729
多层感知机.....731
支持向量机.....739
小结.....749
练习.....750
第22章 目标检测.....753
基于树的目标检测技术.....753
级联分类器.....754

有监督学习和boosting理论 756
学习新目标 764
使用支持向量机的目标识别 772
Latent SVM用于目标识别 772
Bag of Words算法与语义分类 775
小结 780
练习 780

第23章 OpenCV的未来 783
过去与未来 783
OpenCV 3.x 784
我们上一次预测怎么样？ 784
未来应用 785
目前GSoC的进展 787
社区贡献 788
OpenCV.org 789
一些关于AI的猜测 790
结语 793

附录A 平面划分 795

附录B opencv_contrib模块概述 809

附录C 标定图案 813

参考文献 819

第1章

概述

什么是OpenCV

OpenCV是一个开源的计算机视觉库，可以从*http://opencv.org*获取。1999年，Gary Bradski（加里·布拉德斯基）当时在英特尔任职，怀着通过为计算机视觉和人工智能的从业者提供稳定的基础架构并以此来推动产业发展的美好愿景，他启动了OpenCV项目。OpenCV库用C和C++语言编写，可以在Windows、Linux、Mac OS X等系统运行。同时也在积极开发Python、Java、Matlab以及其他一些语言的接口，将库导入安卓和iOS中为移动设备开发应用。OpenCV自项目成立以来获得了来自英特尔和谷歌的大力支持，尤其需要感谢Itseez[编注1]，该公司完成了早期开发的大部分工作。此后，Arraiy[编注2]加入该项目并负责维护始终开源和免费的OpenCV.org。

OpenCV设计用于进行高效的计算，十分强调实时应用的开发。它由C++语言编写并进行了深度优化，从而可以享受多线程处理的优势。如果希望得到更多在英特尔架构上的自动优化，可以购买英特尔的集成性能基元（IPP）库，该库包含了许多算法领域的底层优化程序。在库安装完毕的情况下OpenCV在运行的时候会自动调用合适的IPP库。从OpenCV 3.0开始，英特尔许可OpenCV研发团队和OpenCV社区拥有一个免费的IPP库的子库（称IPPICV），该子库默认集成在OpenCV中并在运算时发挥效用。

编注1：俄罗斯视觉公司，专门从事计算机视觉算法。2016年5月，英特尔收购该公司，以“帮助英特尔的用户打造创新型深度学习的CV应用，如果自动驾驶、数字安全监控和工业检测”（英特尔物联网总经理Doug Dacies语）。

编注2：团队成员有Ethan Rublee（CEO）、Gary Bradski博士（CTO）、Brendan Dowdle（COO）科学家Michael Tetelman博士、工程师9人（含一只小狗）。

OpenCV的一个目标是提供易于使用的计算机视觉接口，从而帮助人们快速建立精巧的视觉应用。OpenCV库包含从计算机视觉各个领域衍生出来的500多个函数，包括工业产品质量检验、医学图像处理、安保领域、交互操作、相机校正、双目视觉以及机器人学。因为计算机视觉和机器学习经常在一起使用，所以OpenCV也包含一个完备的、具有通用性的机器学习库（ML模块）。这个子库聚焦于统计模式识别以及聚类。ML模块对OpenCV的核心任务（计算机视觉）相当有用，但是这个库也足够通用，可以用于任意机器学习问题。

OpenCV怎么用

许多计算机科学家和经验丰富的程序员多多少少都了解计算机视觉的某些方面，但是很少有人熟谙计算机视觉的每一个应用。比如，很多人了解计算机视觉在安保行业的应用，一些人也知道它在网页端的图像和视频处理中的应用在逐渐增加。但很少有人知道计算机视觉在游戏交互中的应用。同时，也很少有人认识到大部分航空图像和街景图像（比如说谷歌街景）已经大量应用相机校正和图像拼接技术。有一些人略微知道一点视觉在自动监控、无人机或者生物制药分析上的应用，但很少有人知道计算机视觉早已经在制造业普遍使用。事实上，批量制造的所有东西都已经利用计算机视觉在进行某些方面的质检工作了。

OpenCV的开源许可允许任何人利用OpenCV包含的任何组件构建商业产品。你也没有义务开源自己的产品或者对该产品所涉及领域进行反馈和改进，虽然我们希望你这样做。在这种自由许可的影响下，项目有着极其庞大的用户社区，社区用户包括一些来自大公司的员工（IBM、微软、英特尔、索尼、西门子和谷歌等）以及一些研究机构（例如斯坦福大学、麻省理工学院、卡内基梅隆大学、剑桥大学以及法国国家信息与自动化研究所）。项目还有一个雅虎论坛组为用户提供提问和讨论的地方，该论坛组有超过50 000名成员。OpenCV在世界范围内都非常流行，尤其是在中国、日本、俄罗斯、欧洲和以色列有着庞大的用户社区。

自从测试版本在1999年1月发布以来，OpenCV已经广泛用于许多应用、产品以及科研工作中。这些应用包括在卫星和网络地图上拼接图像，图像扫描校准，医学图像的降噪，目标分析，安保以及工业检测系统，自动驾驶和安全系统，制造感知系统，相机校正，军事应用，无人空中、地面、水下航行器。它也被运用于声音和音乐的识别，在这些场景中，视觉识别方法被运用于声音的频谱图像。OpenCV亦是斯坦福大学的机器人斯坦

利（Stanley）[编注1]至关重要的一部分，这个机器人赢得了美国国防部高级研究计划署主持的DARPA机器人挑战赛野外机器人竞速的200万美元大奖[编注2]。

什么是计算机视觉

计算机视觉[注1]这种技术可以将静止图像或视频数据转换为一种决策或新的表示。所有这样的转换都是为了完成某种特定的目的而进行的。输入数据可能包含一些场景信息，例如“相机是搭载在一辆车上的”或者“雷达发现了一米之外有一个目标”。一个新的表示，意思是将彩色图像转换为黑白图像，或者从一个图像序列中消除相机运动所产生的影响。

因为我们是被赋予了视觉的生物，所以很容易误认为“计算机视觉也是一种很简单的任务”。计算机视觉究竟有多困难呢？请说说你是如何从一张图像中观察到一辆车的。你最开始的直觉可能具有很强的误导性。人类的大脑将视觉信号划分为许多通道，好让不同的信息流输入大脑。大脑已经被证明有一套注意力系统，在基于任务的方式上，通过图像的重要部分检验其他区域的估计。在视觉信息流中存在巨量的信息反馈，并且到现在我们对此过程也知之甚少。肌肉控制的感知器和其他所有感官都存在着广泛的相互联系，这让大脑能够利用人在世界上多年生活经验所产生的交叉联想，大脑中的反馈循环将反馈传递到每一个处理过程，包括人体的感知器官（眼睛），通过虹膜从物理上控制光线的量来调节视网膜对物体表面的感知。

然而在机器视觉系统中，计算机会从相机或者硬盘接收栅格状排列的数字，也就是说，最关键的是，机器视觉系统不存在一个预先建立的模式识别机制。没有自动控制焦距和光圈，也不能将多年的经验联系在一起。大部分的视觉系统都还处于一个非常朴素原始的阶段。图1-1展示了一辆汽车。在这张图片中，我们看到后视镜位于驾驶室旁边。但是对于计算机而言，看到的只是按照栅格状排列的数字。所有在栅格中给出的数字还有大量的噪声，所以每个数字只能给我们提供少量的信息，但是这个数字栅格就是计算机所能够“看见”的全部了。我们的任务变成将这个带有噪声的数字栅格转换为感知结果“后视镜”。图1-2给出了为什么计算机视觉如此困难的另一些解释。

编注1：2005年，这辆自动驾驶汽车成功越野行驶了212公里，在无人驾驶机器人挑战赛中第一个到达终点。

编注2：DARPPA机器人挑战赛（DRC）是机器人领域的一项重大赛事，堪称“机器人的奥林匹克”。时隔12年，雷锋网独家采访了当事人（本书作者之一）Adrian，文章标题为“重访2005 DARPA挑战赛斯坦福车队成员：Stanley夺冠背后那些不为人知的细节”。

注1：计算机视觉是一个广阔的领域。本书将帮助读者对该领域建立一个最基础的认识。同时我们也推荐Trucco[Trucco98]以获得更全面的介绍、Faugeras[Faugeras93]作为综合参考以及Hartley[Hartley06]和Faugeras[Faugeras93]关于三维视觉的专著。



但相机看到的则是这样：

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	74	65
20	41	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

图1-1. 对于计算机来说，汽车的后视镜就是一组栅格状排列的数字

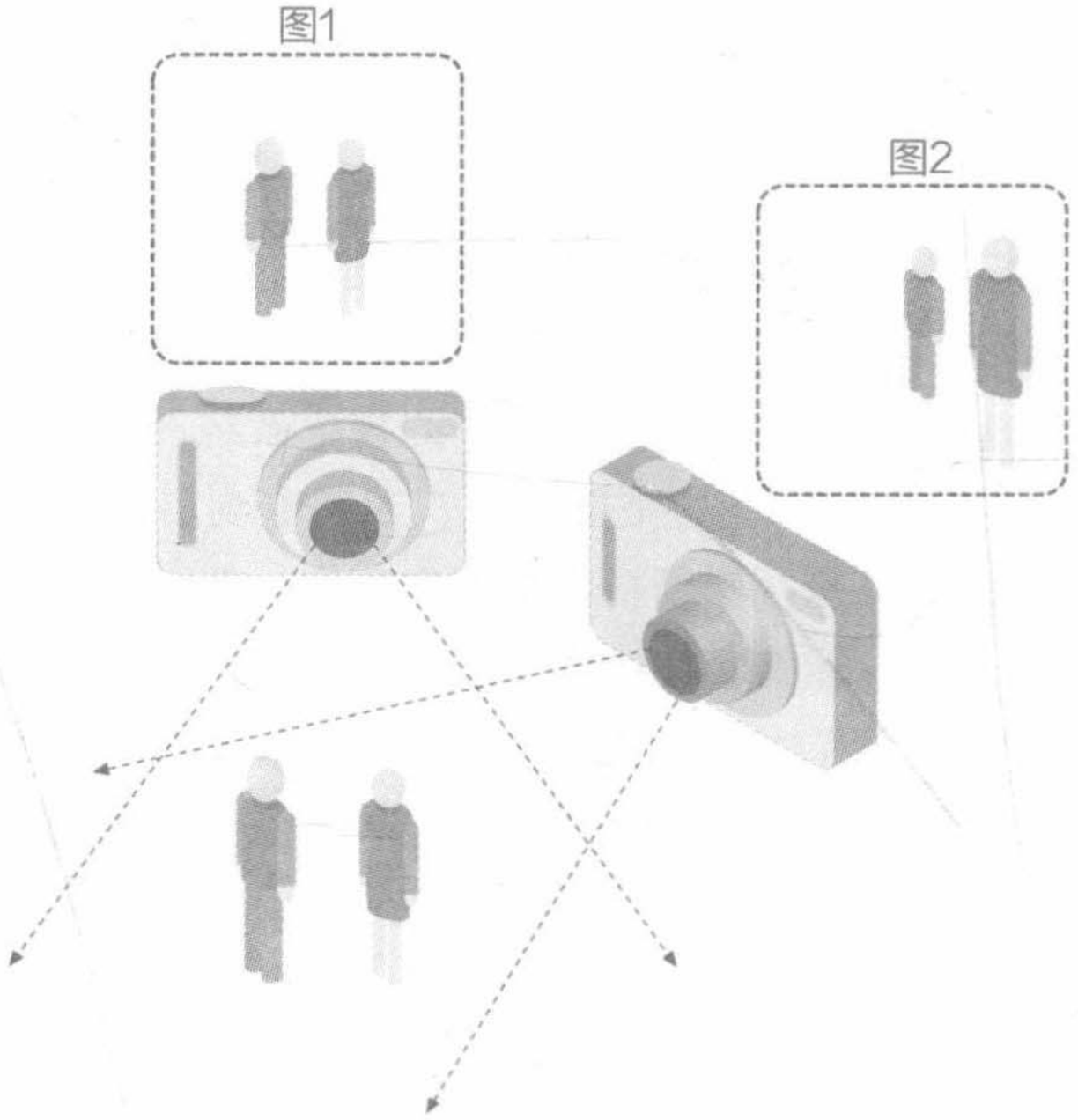


图1-2. 视觉的不适定[译注1]问题，物体的二维表示可能随着视点的不同完全改变

译注1：一个数学物理定解问题的解如果存在，唯一并且稳定的，则说明该问题是适定的（well-posed）；如果不满足，则说明该问题是不适定的（ill-posed）。

实际上，这一问题，正如我们之前所提出的，用“困难”已经不足以形容它了，它在很多情况下根本不可能解决。给定一个对于3D世界的二维（2D）观测，就不存在一个唯一的方式来重建三维信号。即使数据是完美的，相同的二维图像也可能表示一个无限的3D场景组合中的任一种情况。而且，前面也提到过，数据会被噪声和畸变所污染。这样的污染源于现实生活中的很多方面（天气、光线、折射率和运动），还有传感器中的电路噪声以及其他的一些电路系统影响，还有在采集之后对于图像压缩产生的影响。在这一系列的影响之下，我们又该如何推动事情的进展呢？

在经典的系统设计中，额外场景信息可以帮助我们从传感器的层面改善获取信息的质量。考虑这样一个例子，一个移动机器人需要在一栋建筑中找到并且拿起一个订书机。机器人就可能用到这样的事实：桌子通常放在办公室里，而订书机通常收纳在桌子里。这也同样给出了一个关于尺寸的推断：订书机的大小一定可以被桌子所收纳。更进一步，这还可以帮助减少在订书机不可能出现的地方错误识别订书机的概率（比如天花板或者窗口）。机器人可以安全忽略掉200英尺高的订书机形状的飞艇，因为飞艇没有满足被放置在木制桌面上的先验信息。相对的，在诸如图像检索等任务中，数据集中所有的订书机图像都是来自真实的订书机，这样不合常理的尺寸以及一些奇形怪状的造型都会在我们进行图片采集的时候隐式消除——因为摄影师只会去拍摄普通的正常尺寸的订书机。人们同样倾向于在拍摄的时候将拍摄目标放在图片的中间，并且倾向于在最能够展现目标特征的角度拍摄。因此，通常也有很多无意的附加信息在人们拍摄照片的时候无意加进去。

场景信息同样可以（尤其是通过机器学习技术）进行建模。隐式的变量（比如尺寸、重力的方向等不容易被直接观测到的）可以从带标记的数据集中发现关系并推测出来。或者，可以尝试使用附加的传感器测量隐式变量的值，比如利用激光雷达来测量深度，从而准确得到目标的尺寸。

计算机视觉所面临的下一个问题是噪声，我们一般使用统计的方法来对抗噪声。比如，我们很难通过单独的像素点和它的相邻像素点判断其是否是一个边缘点，但如果观察它在一个区域的统计规律，边缘检测就会变得更加简单了。一个真正的边缘应该表现为一个区域内一连串独立的点，所有点的朝向都与其最接近的点保持一致。我们也可以通过时间上的累计统计对噪声进行抑制，当然也有通过现有数据建立噪声模型来消除噪声的方法。例如，因为透镜畸变很容易建模，我们只需要学习一个简单的多项式模型来描述畸变就可以几乎完美矫正失真图像。

基于摄像机的数据，计算机视觉准备做出的动作或决定是在特定的目的或者任务的场景环境中执行的。我们也许想要移除噪声或者修复被损坏的照片，这样安全系统就可以对试图爬上栏杆等危险行为发出警报，或者对于穿过某个游乐场区域的人数进行统计。而

在大楼中漫游的机器人的视觉软件将会采取和安全系统完全不同的策略，因为两种策略处于不同的语境中。一般来说，视觉系统所处的环境约束越严格，我们就越能够依赖这些约束来简化问题，我们最终的解决方案也越可靠。

OpenCV的目标是为计算机视觉需要解决的问题提供工具。在某些情况下，函数库中的高级功能可以有效解决计算机视觉中的问题。即使遇到不能够一次性解决的问题，函数库中的基础组件也具有足够的完备性来增强解决方案的性能，以应对任意的计算机视觉难题。在后一种情况下，也存在一些使用库的可靠方法，所有的这些方法都是从尽量多使用不同的组件库开始。通常，在开发了第一个粗糙的解决方案之后，就可以发现解决方案存在哪些缺陷并且使用自己的代码与聪明才智修复那些缺陷（更为熟知的说法是“解决真正存在的问题，而不是你想象中的那些问题”）。在此之后可以使用粗糙的解决方案作为一个评判标准，评价改善水平。从这一点出发，你可以解决任意问题。

OpenCV的起源

OpenCV缘起于英特尔想要增强CPU集群性能的研究。该项目的结果是英特尔启动了许多项目，包括实时光线追踪算法以及三维墙体的显示。其中一位研究员Gary Bradski（加里·布拉德斯基），当时正在为英特尔工作，他在访问大学的时候注意到很多顶尖大学研究机构，比如MIT的媒体实验室，拥有非常完备的内部公开的计算机视觉开发接口——代码从一个学生传到另一个学生手中，并且会给每个新来的学生一个有价值的由他们自己开发的视觉应用方案。相较于从头开始设计并完成基本功能，新来的学生可以在之前的基础上进行很多新的工作。

所以，OpenCV怀着为计算机视觉提供通用性接口这一思想开始了策划。在英特尔性能实验室（Performance Library）团队的帮助下，[注2]OpenCV最初的核心代码和算法规范是英特尔俄罗斯实验室团队完成的，这就是OpenCV的缘起，从英特尔软件性能组的实验研究开始，俄罗斯的专家负责实现和优化。

俄罗斯专家团队的负责人是Vadim Pisarevsky（瓦迪姆·彼萨里夫斯基），他负责规划、编程以及大部分OpenCV的优化工作，并且到现在他仍是很多OpenCV项目的核心人物。与他一同工作的Victor Eruhimov（维克托·伊拉西莫夫）帮助构建了早期框架，Valery Kuriakin（瓦勒利·库里阿基恩）负责管理俄罗斯实验室并且为项目提供了非常大的助力。以下是OpenCV想要完成的一些目标。

注2：　Shinn Horng Lee（李信弘）是主要推动者，时任IPP（集成性能基元）首席架构师。

- 为高级的视觉研究提供开源并且优化过的基础代码，不再需要重复造轮子。
- 以提供开发者可以在此基础上进行开发的通用接口为手段传播视觉相关知识，这样代码有更强的可读性和移植性。
- 以创造可移植的、优化过的免费开源代码来推动基于高级视觉的商业应用，这些代码可以自由使用，不要求商业应用程序开放或免费。

这些目标阐述了Opencv创建的目的。启用计算机视觉程序，将会增加对运算更快处理器的需求，从而使得用户购买更快的处理器，相较于售卖额外的软件，能够更快增加英特尔的收入。也许这就是这样一个开源并且免费的代码库是由一个硬件厂商而非软件厂商开发的原因。有时，硬件厂商内部拥有更多针对软件的创新空间。

对于开源项目来说，项目需要超过一个临界质量使其能够自我维持，这是开源项目非常重要的一点。目前，OpenCV已经拥有接近1100万次的下载量（2009年3月，200万次），并且这个数字还在以平均每个月160 000次下载的速度增长。OpenCV得到了来自很多用户的贡献，研发主力也很大部分转移到英特尔之外。[注3]OpenCV发展的时间线如图1-3所示。[译注1]在发展过程中，OpenCV受到互联网泡沫的影响，也受到管理层和方向变更等诸多变化的影响，在这些波动的过程中，有时候根本就没有英特尔公司任何人员参与。然而，随着多核处理器的出现以及计算机视觉的许多新应用的问世，OpenCV的价值开始上升。类似地，机器人领域的快速增长也推动了许多用户开始使用和开发这个库。在成为一个开源的函数库之后，OpenCV经过柳树车库几年的开发，现在已经得到了OpenCV基金会的支持。在今天，OpenCV由基金会以及一些上市公司和私人机构开发，更多关于OpenCV未来的信息，请参阅涉及OpenCV未来的相关章节。

OpenCV的结构

OpenCV是由层级结构组织的。处于最上层的是OpenCV和操作系统的交互。接下来是语言绑定和示例应用程序。再下一层是*opencv_contrib*模块所包含的OpenCV由其他开发人员所贡献的代码，其包含大多数高层级的函数功能。这就是OpenCV的核心。底层是基于硬件加速层（HAL）的各种硬件优化。图1-4显示了这个组织关系。

注3：在撰写本书的过程中，柳树车库（Willow Garage）这个机器人研究机构和孵化器，正在积极推动日常的OpenCV维护以及视觉在机器人领域的新进展。

译注1：2016年12月发布OpenCV 3.2；2017年8月发布OpenCV 3.3；2017年12月发布OpenCV 3.4。更详细的发展历程可以访问*https://opencv.org/releases.html*。

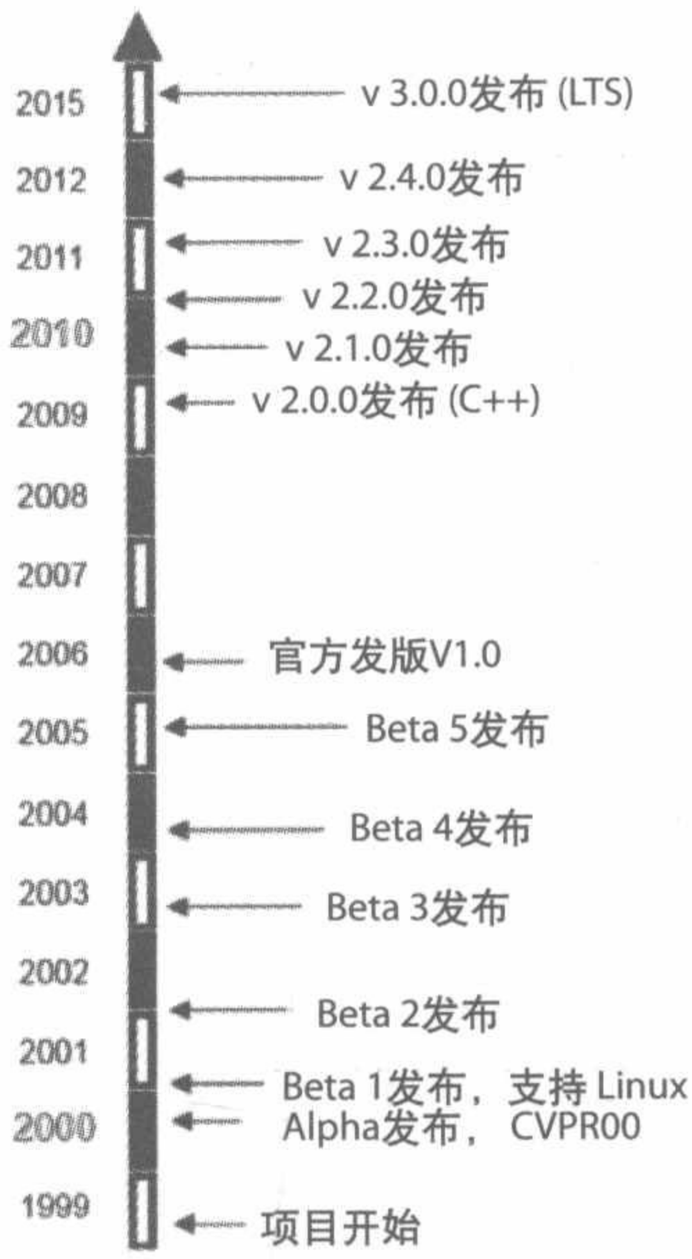


图1-3. OpenCV发展历程

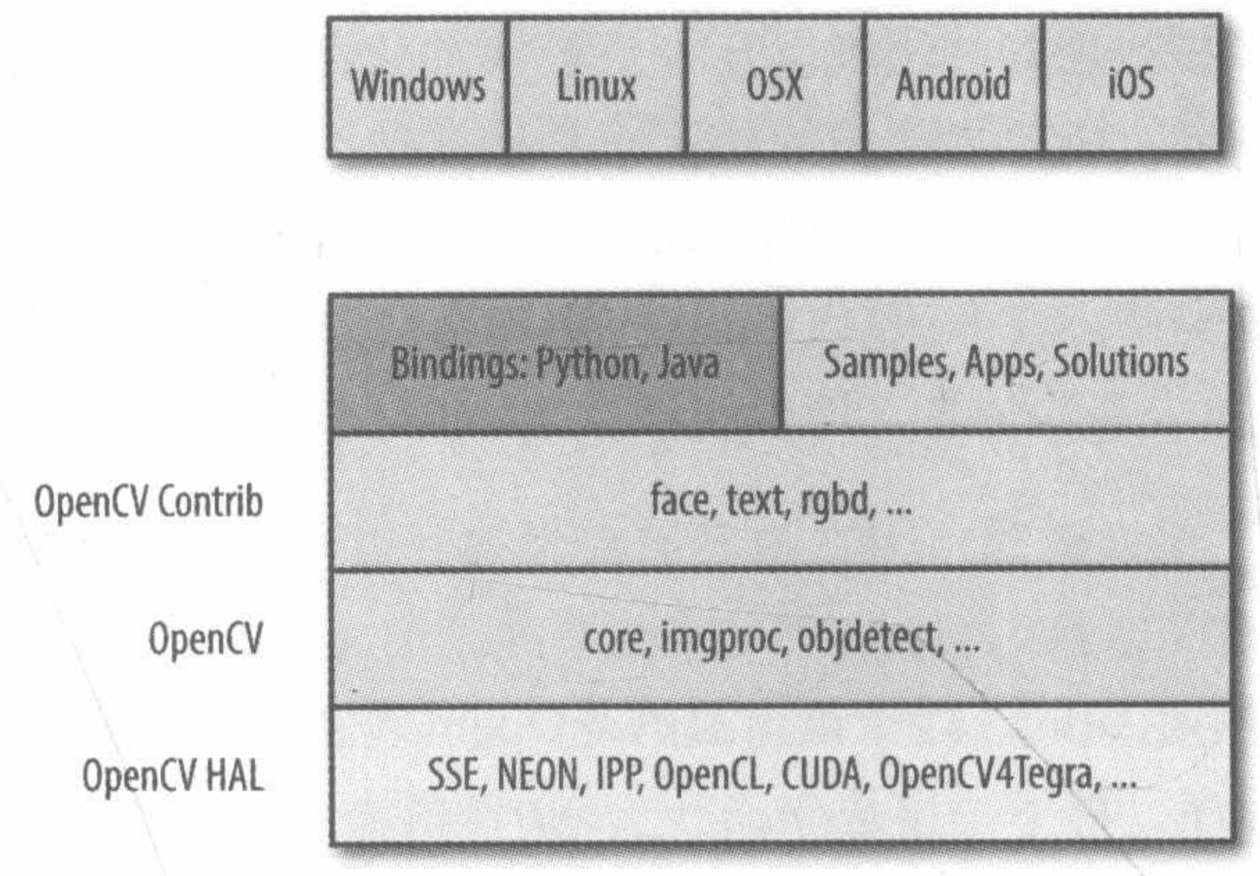


图1-4. OpenCV层级结构及其支持的操作系统

使用IPP来加速OpenCV

如果在英特尔的处理器上使用，OpenCV就会自动使用一种免费的英特尔集成性能原语库（IPP）的子集，IPP 8.x(IPPICV)。IPPICV可以在编译阶段链接到OpenCV，这样一来，会替代相应的低级优化的C语言代码（在cmake中设置`WITH_IPP=ON/OFF`开启或者关闭这一功能，默认情况为开启）。使用IPP获得的速度提升非常可观，图1-5显示了使用IPP之后得到的加速效果。

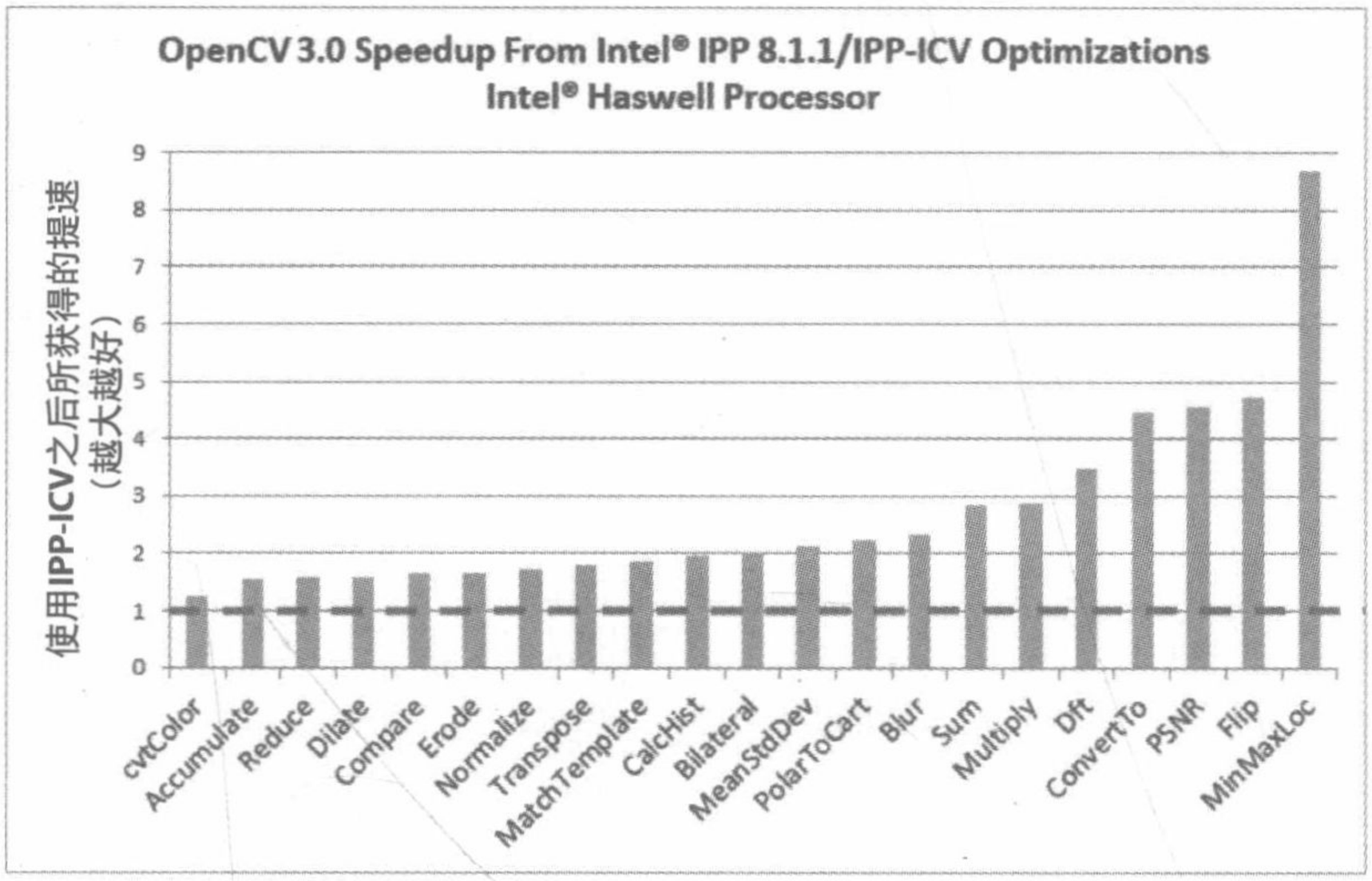


图1-5. 当OpenCV在Intel Haswell处理器上使用IPPICV时的加速效果

谁拥有OpenCV

尽管Gary Bradski在英特尔发起了OpenCV项目，但这个函数库的宗旨是促进商业和研究，这正是它的使命。因此，它是开源并且免费的，并且无论是研究还是商业目的代码都可以（全部或者部分的）使用或者被嵌入到其他程序中。它并不强制你声明基于此开发的代码必须是免费或者开源的，也不要求你将自己的改进反馈回OpenCV中，虽然我们希望可以。

下载和安装OpenCV

在OpenCV官方网站（*http://opencv.org/*）可以下载最新的且完整的源码以及大部分的release版本源码。下载链接可以通过下载页面（*http://opencv.org/downloads.html*）访问。当然，最新的代码会在github（*https://github.com/opencv/opencv*）上进行即时更新。如果想要获取最新的高级函数功能，也可以下载和编译*opencv_contrib*模块（*https://github.com/opencv/opencv_contrib*）。

安装

现在，OpenCV使用Git作为版本管理工具，使用Cmake来构建工程。[注4]当然，这些日子都一去不复返了。在许多情况下，你不必担心构建问题，因为许多环境中都有预编译好

注4：在早些时候，OpenCV开发者使用Subversion作为版本管理工具以及automake来构建工程项目。

的库。但是，一旦成为更加专业的使用者，势必需要重新编译库，并且根据具体的应用程序定制特定的选项。

Windows系统

在*http://opencv.org/downloads.html*，可以发现最新的为Windows准备的最新版本OpenCV下载链接。可以通过这个链接下载一个EXE文件，该文件会把预编译好的OpenCV解压到你的电脑上，预编译版本支持不同版本的Visual Studio环境。现在，你很快就可以开始使用OpenCV了。[注5]

另一个额外的细节是，需要再添加一个名为OPENCV_DIR的环境变量来告诉编译器在哪里找到OpenCV的二进制文件。可以通过命令行工具对此进行设置。[注6]

```
setx -m OPENCV_DIR D:\OpenCV\Build\x64\vc10
```

如果希望静态链接OpenCV，就只需要做到这一步。如果希望使用OpenCV的动态链接库（DLL），就需要告诉系统在哪里找到它的二进制库。为了完成这一目标，只需要在库路径中添加%OPENCV_DIR%\bin。（比如说，在Windows 10中，右击计算机图标，选择“属性”，然后单击“高级系统设置”，最后选择“环境变量”，把OpenCV二进制文件的路径添加到“系统变量”的path变量中。）

OpenCV3集成了IPP，所以如果使用最新的x86或者x64CPU，就可以获得或多或少的性能优势。

也可以按照如下操作从源码编译OpenCV。

1. 运行CMake GUI。
2. 指定OpenCV源码所在路径以及构建目标文件夹（必须和源码所在路径不同）。
3. 按两次Configure键（选择可以使用的Visual Studio编译器或者MinGW构建文件，如果正在使用MinGW的话），直到所有条目没有红色警示。
4. 使用Visual Studio打开生成的解决方案并构建。如果使用的是MinGW，则按照Linux的安装指导进行。

注5：有一点很重要的事情是，尽管Windows环境拥有预编译的release版本的库，但是它并不包含debug版本的库。所以，在开发OpenCV之前，需要打开解决方案并且自行编译那些库。

注6：当然，具体路径会根据安装的不同而有所不同，比如说在32位机器上安装，那么该路径就该是x86而不是x64。

Linux系统

由于GCC和GLIBC在不同Linux版本（SuSE，Debian，Ubuntu，等等）下拥有不同的版本，OpenCV的Linux的预编译版本不包含linux的版本号。然而，在很多情况下，你的Linux版本会提供OpenCV。如果你的版本不提供OpenCV，你将不得不从源代码中构建，与Windows安装一样，可以从*http://opencv.org/downloads.html*下载源代码，但是在这种情况下，该链接会把你转向SourceForge，在这里可以为当前的OpenCV源代码选择压缩文件。

为了编译这个库和示例程序，需要GTK+2.x或者更高的版本，还需要*gcc*以及必要的开发包*cmake*以及*libtbb*（英特尔线程构建模块）以及一些可选项目诸如*zlib*，*libpng*，libjpeg，*libtiff*以及*libjasper*的开发者版本（例如模块名称后带有*-dev*的版本）。你也需要用到Python 2.6或者更高的版本（开发者包）以及NumPy使OpenCV可以在Python环境下工作。此外，还需要来自*ffmpeg*的*libavcodec*以及其他的*libav**库（包含头文件）。

对于后者，请安装Linux发行版本所提供的的*libav/ffmpeg*包，或者前往*http://www.ffmpeg.org*下载。*ffmpeg*库拥有较低的通用公开（LGPL）许可证，但是它的一些组件拥有更严格的通用公开许可证（GPL）。为了结合使用非GPL的软件，你需要构建并且使用一个共享的*ffmpeg*库：

```
$> ./configure --enable-shared
$> make
$> sudo make install
```

当动态链接一个LGPL的库时，不需要为自己的代码使用GPL许可证。最终得到*/usr/local/lib/libavcodec.so.**，*/usr/local/lib/libavformat.so.**，*/usr/local/lib/libavutil.so.**以及在*/usr/local/include/libav**路径下的头文件。

为了编译这个库，需要解压*tar.gz*文件并且切换到解压过程创建的源码文件夹中，然后进行如下操作：

```
mkdir release
cd release
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..
make
sudo make install #  optional
```

第一个命令和第二个命令将创建名为*release*的子目录并且切换到*release*中，第三个命令告诉*cmake*工具如何设置你的工程。我们提供的例子可能是让你入门的正确方法，但是其他选项允许你启用更多的设置：确定哪些例子需要被构建；添加对Python的支持；添加CUDA GPU的支持，等等。默认情况下，OpenCV的*cmake*配置脚本会尝试查找和使用尽可能多的第三方库，例如如果它探测到CUDA SDK的存在，就会自动支持GPU加速的OpenCV功能。最后两个指令将编译整个库并将其安装到正确的位置。注意，如果使用

CMake来建工程，就没有必要安装OpenCV，只需要指定生成的*OpenCVConfig.cmake*文件的路径就可以了。在前面的例子中，文件将被安装在*/usr/local/share/OpenCV*。

正如Windows的例子所示，Linux建立的OpenCV将自动利用IPP进行加速（如果有安装的话）。从OpenCV 3.0版本开始，OpenCV的*cmake*设置脚本将自动下载并且链接一个IPP的免费子库（IPPICV）。如果想要禁用IPP加速，请在执行CMake的时候加上`-D WITH_IPP=OFF`指令。

Mac系统

在Mac上安装步骤和Linux上的安装步骤十分接近，不同的是，Mac拥有自己的开发环境Xcode，包含大部分在CMake过程中需要的东西。你不需要GTK+、TBB，*libjpeg*，并且：

- 在默认情况下，Cocoa会取代GTK+；
- 在默认情况下，QTKit会取代*ffmpeg*；
- GDC会取代TBB以及OpenMP。

安装步骤和Linux下安装一致。需要添加`-G Xcode`指令到CMake中来生成一个Xcode工程，从而可构建和debug工程。

从Git获取最新的OpenCV

OpenCV现在也处于积极的开发状态中，当bug报告中包含有准确的描述以及代码复现bug的时候，该bug通常会被迅速修复。然而，官方的OpenCV通常每年只发布一次到两次，如果你正在开发一个项目或产品，可能想要OpenCV能够尽快修复bug并更新。为了完成这些目的，需要在GitHub网站上访问OpenCV的Git仓库。

本节并不打算引导你如何使用Git，如果你还在使用另外的一些开源项目，也许对这套操作已经很熟悉了。如果对此并不熟悉，请查阅Jon Loeliger（乔恩·罗力格）所著的《Git版本控制管理》（*http://shop.oreilly.com/product/0636920022862.do*）。Git的命令行工具有Linux，OS X以及大部分的类UNIX系统所支持。针对Windows，我们推荐TortoiseGit（*https://tortoisegit.org/*）；针对OS X，SourceTree也许适合。

在Windows上，如果想要从Git获得OpenCV最新的版本，你需要访问*https://github.com/opencv/opencv.git*。

在Linux上，只需要输入如下指令：

```
git clone https://github.com/opencv/opencv.git
```

更多的OpenCV文档

OpenCV基础文档可以在*http://opencv.org*获取。另外，在*http://docs.opencv.org/2.4.13/doc/tutorials/tutorials.html*，也可以获得更多深入教程，OpenCV的维基网站页面现在位于*https://github.com/opencv/opencv/wiki*。

提供的文档

OpenCV 2.x自身就提供PDF格式的完整的引用手册以及丰富的教程，查看*opencv/doc*目录就可以得到。从OpenCV 3.x开始，就不再提供离线的文档了。

在线文档和维基资源

正如我们之前提到的，在*https://opencv.org*上有大量的文档和维基资源。文档分为以下几个主要部分。

参考（http://docs.opencv.org/）

这个部分包含函数、它们的参数以及如何使用它们的一些信息。

教程（http://docs.opencv.org/trunk/d9/df8/tutorial_root.html）

这个部分是许多教程的集合，这些教程会告诉你如何完成各种各样的事情。这里有一些基础课程的教程（比如如何在不同平台上安装OpenCV或者创建OpenCV项目）以及一些更高级的话题（比如目标检测的背景提取算法）。

快速指南（http://opencv.org/quickstart.html）

本部分包含一个精心编制的快速指南，只包含帮助你在特定平台上获取和运行OpenCV的内容。

速查表（http://docs.opencv.org/3.0-last-rst/opencv_cheatsheet.pdf）

只有一页篇幅的PDF文档，包含整个库的高度压缩的参考。感谢Vadim Pisarevsky制作了这个出色的速查表，这样你就可以把这漂亮的两页纸钉在小隔间的墙上随时查阅。

维基（https://github.com/opencv/opencv/wiki）

维基包含所有你可能想要的东西，甚至可能超乎你的想象。在这里，可以找到路线图、新闻、开放问题、bug追踪以及无数个更深入的主题，比如如何成为OpenCV的贡献者。

问答社区（http://answers.opencv.org/questions/）

问答社区是一个庞大的、包含数千个人遇到过的问题的档案集合。可以在这里向OpenCV社区提问，或者通过回答问题帮助其他人。

以上所有在OpenCV.org的底部的有超链接可以访问。在这些高质量的资源中，有一个更值得我们讨论——“参考”。参考划分为几个部分，每个部分都和库中一个模块有关。具体的模块列表随着时间推移而不断的发展，但模块始终是组成这个库的基本单位。每个函数都是一个模块的一部分，以下是当前OpenCV所拥有的模块。

Core

该模块包含OpenCV库的基础结构以及基本操作。

Improc

图像处理模块包含基本的图像转换，包括滤波以及类似的卷积操作。

Highgui（在OpenCV 3.0中，分割为imcodecs、videoio以及highgui三部分）

这个模块包含可以用来显示图像或者简单的输入的用户交互函数。这可以看作是一个非常轻量级的Windows UI工具包。

Video

该模块包含读取和写视频流的函数。

Calib3d

这个模块包括校准单个、双目以及多个相机的算法实现。

Feature2d

这个模块包含用于检测、描述以及匹配特征点的算法。

Objdectect

这个模块包含检测特定目标，比如人脸或者行人的算法。也可以训练检测器并用来检测其他物体。

Ml

机器学习模块本身是一个非常完备的模块，包含大量的机器学习算法实现并且这些算法都能和OpenCV的数据类型自然交互。

Flann

Flann的意思是“快速最邻近库”。这个库包含一些你也许不会直接使用的方法，但是其他模块中的函数会调用它在数据集中进行最邻近搜索。

GPU（在*OpenCV 3.x*中被分割为多个cuda*模块）

GPU模块主要是函数在CUDA GPU上的优化实现，此外，还有一些仅用于GPU的功能。其中一些函数能够返回很好的结果，但是需要足够好的计算资源，如果硬件没有GPU，则不会有什么提升。

Photo

这是一个相当新的模块，包含计算摄影学的一些函数工具。

`Stitching`

本模块是一个精巧的图像拼接流程实现。这是库中的新功能，但是，就像Photo模块一样，这个领域未来预计有很大的增长。

`Nonfree`（在*OpenCV 3.0*中，被移到*opencv_contrib/xfeatures2d*）

OpenCV包含一些受到专利保护的或者受到使用限制的（比如SIFT算法）算法。这些算法被隔离到它们自己的模块中，以表明你需要做一些特殊的工作，才可以在商业产品中使用它们。

`Contrib`（在*OpenCV 3.0*中，融合进了*opencv_contrib*）

这个模块包含一些新的、还没有被集成进OpenCV库的东西。

`Legacy`（在*OpenCV 3.0*中，被取消）

这个模块包含一些老的尚未被完全取消的东西。

`Ocl`（在*OpenCV 3.0*中，被取消，取而代之的是*T-API*）

这是一个较新的模块，可以认为它和GPU模块相似，它实现了开放并行编程的Khronos OpenCL标准。虽然现在模块的特性比GPU模块少很多，但`Ocl`模块的目标是提供可以运行在任何GPU或者是其他可以搭载Khronos的并行设备。这与GPU模块形成了鲜明的对比，后者使用Nividia CUDA工具包进行开发，因此只能在Nividia GPU设备上工作。

尽管在线文档的质量越来越高，但并没有充分解释算法的实现细节及期所要求的参数的准确含义。这本书的目的是提供这些信息以及帮助读者更深入地理解库中所有的基本模块。

OpenCV贡献库

在OpenCV 3.0中，先前单一的库分成两部分：成熟的OpenCV和*opencv_contrib*中的最新模块。前者由核心的OpenCV团队维护，并且包含(大部分)稳定的代码，而后者则不成熟，主要由社区维护和开发，可能有非OpenCV许可的部分，并且可能包括受专利保护的算法。

以下是一些可以在*opencv_contrib*中获得的模块，参阅附录B，可以查看完整的最新列表。

`Dnn`

深度神经网络。

`Face`

人脸识别。

`Text`

文本检测以及识别，基于许多开源的OCR算法。

`Rgbd`

处理由Kinect或者其他深度传感器（或者简单的由双目匹配得到的）获取的RGB+深度图像。

`Bioinspired`

一些基于生物学启发的视觉技术。

`ximgproc`和`xphoto`

先进的图像处理以及计算摄影学方法。

`Tracking`

现代目标追踪算法。

下载和编译Contributed模块

在Linux和OS X中，可以直接使用下面的指令来下载*opencv_contrib*模块。

```
git clone https://github.com/opencv/opencv_contrib.git
```

在Windows，将这个地址输入TortoiseGit或者其他客户端，接下来只需要再次配置OpenCV的Cmake：

```
cmake -D CMAKE_BUILD_TYPE=Release \
  -D OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules ..
```

然后和平常一样再次编译。编译好的Contributed会和普通的OpenCV模块处于同样的目录，不需要任何特殊的配置就可以使用它们。

可移植性

OpenCV基于可移植的理念设计，最初可以被任意通用的C++编译器编译。这表示它的C和C++代码都必须是相当标准的，以便跨平台支持更加容易。表1-1显示OpenCV在不同运行平台的支持，其中对于Intel和AMD 32位，64位（x86，x64）的支持是最成熟的，对ARM的支持也在迅速改善中。对于操作系统，OpenCV完全支持Windows、Linux、OS X、Android和iOS。如果一个体系结构或者操作系统没有出现在这个表中，并不意味着这个系统无法使用OpenCV。事实上，OpenCV已经被移植到几乎所有商用系统中，从亚马逊云以及40核的英特尔Xeon Phi到树莓派和机器狗中。

表1-1：OpenCV 1.0发行版本移植性引导

	x86/x64	ARM	其他：MIP，PPC
Windows	SIMD，IPP，Parallel，I/O	SIMD，Parallel (3.0)，I/O	N/A
Linux	SIMD，IPP，Parallel，[a] I/O	SIMD，Parallel，[a] I/O	Parallel，[a] I/O*
Android	SIMD，IPP (3.0)，Parallel，[b] I/O	SIMD，Parallel，[b] I/O	MIPS—基本支持
OS X/iOS	SIMD，IPP (3.0)，Parallel，I/O	SIMD，Parallel，I/O	N/A
其他：BSD，QNX等	SIMD	SIMD	

a. Linux中的并行化是通过第三方库或启用OpenMP来实现的

b. Android设备中并行是通过TBB进行的

对表1-1的说明如下。

SIMD

用于加速的向量指令：基于x86，x64，NEON以及ARM的SSE指令集。

IPP

可否使用英特尔IPP加速，从3.0版本开始，函数库自带一个IPP的免费子库（IPPICV）。

Parallel

一些用于在多核进行线程控制的标准或者第三方线程框架。

I/O

一些用于抓取或者写视频的标准或第三方API。

小结

在这一章中，我们对OpenCV的历史进行了研究，从1999年由Gary Bradski（加里·布拉德斯基）在英特尔公司始创到现在Arraiy的支持。我们讨论了OpenCV及其部分内容的动机。讨论了OpenCV核心库是什么样的以及*opencv_contrib*模块（参考附录B）。本章还介绍了如何下载和安装OpenCV，以及它的性能和可移植性如何。

练习

1. 下载并且安装最新的OpenCV，并分别在debug模式和release模式下进行编译。
2. 从Git下载并且编译最新的OpenCV代码。
3. 描述至少三个将3D输入转换为2D表示时引起的问题，你如何解决这些问题？

第2章

OpenCV初探

头文件

在安装OpenCV库以及设置好编程环境之后，下一个任务就是用代码来做一些有趣的事情。为了完成这个工作，我们需要先讨论一下头文件。幸运的是，头文件是按照第1章所述模块一样组织的。而我们最感兴趣的头文件莫过于*.../include/opencv2/opencv.hpp*，它包含OpenCV各个模块的头文件：

```
#include "opencv2/core/core_c.h"
```

旧式C风格的结构以及运算

```
#include "opencv2/core/core.hpp"
```

新式C++风格的结构以及数学运算

```
#include "opencv2/flann/miniflann.hpp"
```

最邻近搜索匹配函数

```
#include "opencv2/imgproc/imgproc_c.h"
```

旧式C风格的图像处理函数

```
#include "opencv2/imgproc/imgproc.hpp"
```

新式C++风格图像处理函数

```
#include "opencv2/video/photo.hpp"
```

操作和恢复照片相关算法

```
#include "opencv2/video/video.hpp"
```

视觉追踪以及背景分割

```
#include "opencv2/features2d/features2d.hpp"
```

用于追踪的二维特征

```
#include "opencv2/objdetect/objdetect.hpp"
```

级联人脸分类器、latent SVM分类器、HoG特征和平面片检测器

```
#include "opencv2/calib3d/calib3d.hpp"
```

校准以及双目视觉相关

```
#include "opencv2/ml/ml.hpp"
```

机器学习、聚类以及模式识别相关

```
#include "opencv2/highgui/highgui_c.h"
```

旧式C风格的显示、滑动条、鼠标操作以及输入输出相关

```
#include "opencv2/highgui/highgui.hpp"
```

新式C++风格的显示、滑动条、鼠标操作以及输入输出相关

```
#include "opencv2/contrib/contrib.hpp"
```

用户贡献的代码、皮肤检测、模糊Mean-Shift追踪、spin image算法及自相似特征等

你也许会使用头文件*opencv.hpp*来包含所有可能在OpenCV函数中用到的头文件，但是这会减慢编译的速度。如果只使用一个，比如说图像处理相关的函数，只包含*opencv2/imgproc/imgproc.hpp*所消耗的编译时间会比包含*opencv.hpp*消耗的编译时间少很多。这些头文件在*.../modules*文件夹中。比如，*imgproc.hpp*存在于*.../modules/imgproc/include/opencv2/imgproc/imgproc.hpp*。类似，函数对应的源文件也存在于对应的*src*文件夹中。比如，improc模块的`cv::Canny()`函数存在于*.../modules/improc/src/canny.cpp*。

有了前述头文件之后，就可以开始写我们的第一个C++的OpenCV程序了。



可以使用*opencv2/legacy/legacy.hpp*来调用遗留代码，比如说比较旧的斑点追踪、隐马尔科夫模型（HMM）检测、聚合追踪以及特征目标。其他位于*.../modules/legacy/include/opencv2/legacy/legacy.hpp*。

资源

网上有几个不错的OpenCV资源：

- 对整个OpenCV库进行概述：*https://www.slideshare.net/zblair/opencv-introduction*或OpenCV3概述，*http://slideplayer.com/slide/10302485/*，也可以发送邮件到*coo@netease.com*，大家共同探讨

- 关于加速的讨论：*https://www.slideshare.net/embeddedvision/making-opencv-code-run-fast-a-presentation-from-intel*
- 关于模块的讨论：*https://docs.opencv.org/3.1.0/*

第一个程序：显示图片

OpenCV提供了一些实用工具来读取从视频流到摄像机画面的各种各样的图像格式，这些工具都是HighGUI的一部分。我们将使用其中的一些工具来创建一个简单的程序，这个程序将打开一张图像并且将其显示在屏幕上（如示例2-1所示）。

示例2-1：一个简单的加载并显示图像的OpenCV程序

```
#include <opencv2/opencv.hpp> //Include file for every supported OpenCV function

int main( int argc, char** argv ) {
  cv::Mat img = cv::imread(argv[1],-1);
  if( img.empty() ) return -1;
  cv::namedWindow( "Example1", cv::WINDOW_AUTOSIZE );
  cv::imshow( "Example1", img );
  cv::waitKey( 0 );
  cv::destroyWindow( "Example1" );
  return 0;
}
```

注意，OpenCV的函数都位于cv这一命名空间下，为了调用OpenCV的函数，你需要在每个函数前加上cv::，向编译器说明你所调用的函数处于cv命名空间。为了摆脱这种繁琐的工作，我们可以像示例2-2一样用using namespace cv;指令，告诉编译器假设所有函数都位于cv命名空间下。[注1]你还需要注意示例2-1和示例2-2在头文件上的不同，在前者中，我们使用了通用的*opencv.hpp*，而在后者中，我们只使用了必须的头文件来节约编译时间。

示例2-2：与示例2-1不同的是直接使用了using namespace

```
#include "opencv2/highgui/highgui.hpp"

using namespace cv;

int main( int argc, char** argv ) {

  Mat img = imread( argv[1], -1 );
  if( img.empty() ) return -1;
```

注1：当然，如果这样做，需要冒着和其他潜在的命名空间冲突的风险。如果函数foo()已经存在于cv和std的命名空间，你必须指定自己使用的是位于cv::foo()的函数还是位于std::foo()的函数。在本书中，除了示例2-2，都会指明cv::命名空间，并以此保持一个良好的编程风格。

```
  namedWindow( "Example1", cv::WINDOW_AUTOSIZE );
  imshow( "Example1", img );
  waitKey( 0 );

  destroyWindow( "Example1" );

}
```

当使用命令行编译和运行时，[注2]示例2-1将加载一张图像到内存中并且显示到屏幕上。它会保持显示直到用户按下一个键，之后程序才会销毁窗口并退出。现在让我们来一行一行地解释代码，并花一些时间来理解每一行代码都在做什么工作。

```
cv::Mat img = cv::imread( argv[1], -1 );
```

这一行将会载入图像。[注3]函数cv::imread()是高级的；依据文件名来决定载入图像格式的处理。这也会自动地申请图像需要的内存，注意，cv::imread()可以读取很多种图像格式，包括BMP，DIP，JPEG，JPE，PNG，PBM，PGM，PPM，SR，RAS以及TIFF。函数会返回一个cv::Mat结构，这个结构是OpenCV中你将会接触最多的自带结构。OpenCV使用这个结构来处理所有类型的图像：单通道、多通道、整型、浮点数以及各种类型。紧接着的下面这一行：

```
if( img.empty() ) return -1;
```

检查这个图像是否真的被载入了。另一个高层级的函数cv::namedWindow()将会在屏幕打开一个窗口，其中可以包含需要显示的图片。

```
cv::namedWindow( "Example1", cv::WINDOW_AUTOSIZE );
```

该函数由HighGUI模块提供，会将一个名称赋予窗口（在这里窗口名为"Example1"）。未来HighGUI的和这个窗口的交互函数将由这个名称来指定要与哪个窗口交互。

cv::namedWindow第二个参数说明了Windows的特性。这可以全部设置为0（默认情况下），也可以设置为cv::WINDOW_AUTOSIZE。在之前的例子中，窗口的大小将会和载入图像的大小一致，图像将会被缩放以适应窗口的大小。在之后的例子中，窗口将会在图像载入的时候被自动缩放以适应图像的真实大小，也可能由用户自行调整。

注2：当然，如果这样做的话，你需要冒着和其他潜在的命名空间冲突的风险。如果函数foo()已经存在于cv和std的命名空间，必须指定你使用的是位于cv::foo()的函数还是位于std::foo()的函数。在本书中，除了示例2-2，都会指明cv::命名空间，并以此保持一个良好的编程风格。

注3：好的程序会检查argv[1]是否存在并且给用户反馈一个错误信息，但是在这里没有。我们在本书中将会简略掉这些处理并且假设读者都有足够的的知识来理解处理错误代码的方式及其重要性。

```
cv::imshow( "Example1", img );
```

不论何时，只要在cv::Mat中拥有一个图像结构，我们都可以通过cv::imshow()进行显示。cv::imshow()将建一个窗口（如果这个窗口不存在，它会自动调用cv::namedWindow()新建一个窗口）。在调用cv::imshow()的时候，窗口将被重绘上要求的图片，并且窗口会按照要求自动调整大小（如果使用cv::WINDOW_AUTOSIZE参数）。

```
cv::waitKey(0);
```

cv::waitKey(0);函数告诉系统暂停并且等待键盘事件。如果其传入了一个大于零的参数，它将会等待等同于该参数的毫秒时间，然后继续执行程序。如果参数被设置为0或者一个负数，程序将会无限等待直到有键被按下。

因为有cv::Mat，图像将会在生命周期结束的时候自动释放，其行为类似于标准模板库（STL）中的容器类。这种自动的内存释放由内部的引用指针所控制，最重要的是，这表示我们用不着担心图像的内存申请和释放，这将程序员从OpenCV 1.0 IplImage结构繁琐的维护工作中解放了出来。

```
cv::destroyWindow( "Example1" );
```

最后，我们可以让窗口自行销毁。函数cv::destroyWindow()将会关闭窗口并且释放掉相关联的内存空间。为了更简洁的编码，我们将会在之后的例子中略过这一步。在更长、更复杂的代码中，程序员应该在窗口的生命周期自然结束之前自主销毁窗口以防止内存泄漏。

我们下一个任务是创建一个非常简单的、几乎和本例一样的程序来读取视频文件。在此之后，我们将会开始对实际图像进行更多的操作。

第二个程序：视频

用OpenCV播放视频就像显示图像一样简单。唯一不同的是，我们需要某种循环来读取视频序列中的每一帧。我们也许还需要一些方法在电影太无聊的时候来帮助跳出循环。

示例2-3：一个简单的播放视频文件的OpenCV程序

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

int main( int argc, char** argv ) {

  cv::namedWindow( "Example3", cv::WINDOW_AUTOSIZE );
  cv::VideoCapture cap;
  cap.open( string(argv[1]) );
```

```
  cv::Mat frame;
  for(;;) {
    cap >> frame;
    if( frame.empty() ) break;                  // Ran out of film
    cv::imshow( "Example3", frame );
    if( cv::waitKey(33) >= 0 ) break;
  }

  return 0;
}
```

main函数从创建一个窗口开始（在本例中窗口名为"Example3"），视频读取结构cv::VideoCapture在其后被加载进来。这个结构可以打开和关闭很多类型的*ffmpeg*支持的视频文件。

```
cap.open(string(argv[1]));
cv::Mat frame;
```

视频读取结构通过传入字符串打开文件，这个字符串指示了想要打开的视频文件的路径。一旦视频被打开，视频读取结构将会包含所有的关于这个视频文件可以读取的属性，包括状态信息。以这样的方式创建以后，cv::VideoCapture结构将会在视频的开头被初始化。在这个程序中，cv::Mat frame声明了一个可以保存视频帧的结构。

```
cap >> frame;
if( frame.empty() ) break;
cv::imshow( "Example3", frame );
```

一旦内部的while()循环开始执行，视频文件会按照帧从视频流中被读取。这个程序通过if(frame.empty())检查数据是不是真的从视频中读了出来，如果没有，程序将会退出。如果视频帧被成功读取，将通过cv::imshow()显示。

```
if( cv::waitKey(33) >= 0 ) break;
```

一旦显示了这帧图片，我们会等待33毫秒。[注4]如果用户在这段时间在键盘有任何输入，我们将退出循环。如果没有发生，33毫秒之后我们会跳过并执行下一个循环。在退出的时候，所有数据的内存空间将会由于生命周期的结束被自动释放掉。

跳转

为了强化前面这个简单的程序以及发现更多有用的函数，现在是时候反思一下了。我们也许已经注意到示例2-3中的视频播放器不可以快速在视频中进行跳转。所以我们下一个

注4：你也许想要定义其他时间长度，在这个情况下，我们简单说明33毫秒是因为这能让视频以30FPS的速率播放，并且能够允许用户在播放的时候打断。根据以往的经验，最好检查cv::VideoCapture结构来确定视频真正的帧率（更多详情请参阅第8章）。

任务就是添加一个滑动条，使得用户可以进行视频跳转。为了执行更多的控制，我们将会允许用户通过按下S键来执行单步模式，以及按下R键来恢复连续视频播放模式，无论何时用户通过滑动条跳转到视频一个新的时间点，我们都会使用单步模式在那个点进行播放。

HighGUI工具提供了许多简单的工具用于处理视频和图像，而不仅仅是我们刚才演示的那些功能。一个特别有用的工具就是我们前面所提到的滑动条，它能够使用户轻松从视频的一个部分跳转到另一个部分。要创建一个滑动条，我们会调用cv::createTrackbar()并指明要在哪个窗口中进行显示，为了完成所需要的功能，我们还需要一个能够执行重新定位的回调函数，示例2-4给出了代码表述。

示例2-4：加入了滑动条的基本浏览窗口

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <fstream>

using namespace std;

int g_slider_position = 0;
int g_run = 1, g_dontset = 0; //start out in single step mode
cv::VideoCapture g_cap;

void onTrackbarSlide( int pos, void *) {

  g_cap.set( cv::CAP_PROP_POS_FRAMES, pos );

  if( !g_dontset )
    g_run = 1;
  g_dontset = 0;

}

int main( int argc, char** argv ) {

  cv::namedWindow( "Example2_4", cv::WINDOW_AUTOSIZE );
  g_cap.open( string(argv[1]) );
  int frames = (int) g_cap.get(cv::CAP_PROP_FRAME_COUNT);
  int tmpw   = (int) g_cap.get(cv::CAP_PROP_FRAME_WIDTH);
  int tmph   = (int) g_cap.get(cv::CAP_PROP_FRAME_HEIGHT);
  cout << "Video has " << frames << " frames of dimensions("
       << tmpw << ", " << tmph << ")." << endl;

  cv::createTrackbar("Position", "Example2_4", &g_slider_position, frames,
                     onTrackbarSlide);

  cv::Mat frame;
  for(;;) {

    if( g_run != 0 ) {
```

```
    g_cap >> frame; if(frame.empty()) break;
    int current_pos = (int)g_cap.get(cv::CAP_PROP_POS_FRAMES);
    g_dontset = 1;

    cv::setTrackbarPos("Position", "Example2_4", current_pos);
    cv::imshow( "Example2_4", frame );

    g_run-=1;

  }

  char c = (char) cv::waitKey(10);
    if( c == 's' ) // single step
      {g_run = 1; cout << "Single step, run = " << g_run << endl;}
    if( c == 'r' ) // run mode
      {g_run = -1; cout << "Run mode, run = " << g_run <<endl;}
    if( c == 27 )
      break;

  }
  return(0);

}
```

本质上，代码中添加一个全局变量来表示滑动条的位置并且添加一个回调函数来更改这个变量来重新定位视频读取的位置。在创建滑动条和回调函数之后，我们就开始运行程序，[注5]让我们从全局变量开始更详细地查看整个程序。

```
int g_slider_position = 0;
int g_run             = 1;
int g_dontset         = 0;      // start out in single-step mode
VideoCapture g_cap;
```

在程序的开始处，我们定义了一个全局变量g_slider_position来存储滑动条位置。由于回调函数需要访问帧读取结构g_cap，所以我们最好将这个结构也提升为全局变量。为了保证代码的易读性。我们会在每一个全局变量前加上g_前缀以表明这是一个全局变量。同理，全局变量g_run将在新的跳转触发之后置0。当它为正的时候，指示在停止之前程序要播放多少张图片；当它为负的时候，表示系统处于连续视频播放模式。

为了防止冲突，在用户单击了滑动条跳转到一个视频的新位置之后，我们将会通过设置g_run变量为1使视频进入单步模式。即使是这样，也存在一个小问题：当视频前进的时候，我们希望滑动条也能够随着视频的当前进度前进。我们通过在程序中调用滑动条的回调函数实现这一功能。但是，我们并不希望在这个时候进入单步模式。为了避免这样的事情发生，我们引入最后一个全局变量g_dontset来避免在调整进度条位置的时候触发单步模式。

注5：　注意，有一些AVI格式和MPEG格式的文件不允许在视频中向后跳转。

```
void onTrackbarSlide(int pos, void *) {

  g_cap.set(cv::CAP_PROP_POS_FRAMES, pos);

  if( !g_dontset )
    g_run = 1;
  g_dontset = 0;

}
```

现在，定义一个用户调整滑动条的时候执行的回调程序。这个程序将会传入一个32位的整型数值以表示当前的位置。这将会是进度条存在的新的位置。在回调函数的内部，我们在新的位置通过调用g_cap.set()来真正使进度条移到我们希望的位置。if()语句设置程序是否进入单步模式。只有在用户触发滑动条事件的时候，这个设置才会生效；在系统自动调用该回调函数的时候，单步模式不会生效。

调用g_cap.set()是我们以后经常使用的一个操作，它通常配合g_mcap.get()使用。这些程序允许我们配置（或是询问，这将在稍后介绍）cv::VideoCapture结构中的许多变量。在本例中，我们输入参数CV::CAP_PROP_POS_FRAMES，这个参数指示我们想要帧集合的读取位置。[注6]

```
int frames = (int) g_cap.get(cv::CAP_PROP_FRAME_COUNT);
int tmpw   = (int) g_cap.get(cv::CAP_PROP_FRAME_WIDTH);
int tmph   = (int) g_cap.get(cv::CAP_PROP_FRAME_HEIGHT);
cout << "Video has " << frames << " frames of dimensions("
     << tmpw << ", " << tmph   << ")." << endl;
```

主程序的核心和示例2-3一致，所以我们将把重点放在我们添加了什么上面。第一个不同是在打开视频之后我们用g_cap.get()来确定总帧数以及视频的高和宽。这些数字打印出来之后，我们需要这些视频中和帧有关的信息来校准滑动条（在下一步）。

```
createTrackbar("Position", "Example2_4", &g_slider_position, frames,
               onTrackbarSlide);
```

下一步我们将创建一个滑动条，这个函数cv::createTrackbar()允许我们给滑动条一个标签[注7]（在本例中，是Position）并且指明在哪一个窗口放置我们的滑动条。我们提供一个和滑动条绑定的变量——滑动条能够达到的最大值（视频的总帧数）以及一个在滑动条移动时候的回调函数（不需要的时候也可以输入NULL）。

```
if( g_run != 0 ) {
```

注6：因为HighGUI的代码是高度自动化的，所以当一个新的视频位置被请求的时候，将自动处理一些（诸如请求的帧并非是一个关键帧等）问题。如果遇到这种情况，它会从请求位置之前的一个关键帧开始播放并且快进到请求的位置，免得我们考虑这些繁琐的问题。

```
    g_cap >> frame; if(!frame.data) break;
    int current_pos = (int)g_cap.get(cv::CAP_PROP_POS_FRAMES);
    g_dontset = 1;

    cv::setTrackbarPos("Position", "Example2_4", current_pos);
    cv::imshow( "Example2_4", frame );

    g_run-=1;

  }
```

在while循环中，为了读取和显示视频中的帧，我们还会获取在视频中的当前位置，设置g_dontset使下一个滑动条的回调函数不会让我们进入单步模式，之后我们将会委托滑动条的回调函数来更新滑动块在滑动条上的位置以便显示给用户。全局变量g_run将会减1，它的效果是让我们保持单步模式或者依据用户的设置让视频正常播放。接下来我们将会看到以下结构：

```
char c = (char) cv::waitKey(10);
if( c == 's' ) // single step
  {g_run = 1; cout << "Single step, run = " << g_run << endl;}
if( c == 'r' ) // run mode
  {g_run = -1; cout << "Run mode, run = " << g_run <<endl;}
if( c == 27 )
  break;
```

在while循环的底部，我们会等待用户的键盘输入。如果S键被按下，我们将会进入单步模式（g_run将会被设置为1，会使得程序只读取一张图片）。如果R被按下，我们将会进入连续视频模式（g_run将会被设置为-1，之后进一步递减使其对于之后任意帧都为负）。最后，如果Esc被按下，程序将会终止。再次提醒你注意，为了让代码更简洁，我们略过了回收窗口垃圾这一步cv::destroyWindow()。

简单的变换

很好，现在你已经可以用OpenCV来自己动手建立一个视频播放器了，这和现有的那些播放器差不多，但是我们的关注点在计算机视觉，所以希望做一些计算机视觉相关的工作。许多基础的计算机视觉工作都包括对视频流使用滤波器。我们将会对已有的程序进行修改以对视频中的每一帧实现一些简单的操作。

一个最简单的操作就是对图像的平滑处理，这将有通过高斯核或者其他核卷积效减小图像的信息量。OpenCV使得这样使用高斯核的卷积非常容易实现。如示例2-5所示，我们

注7：因为HighGUI是一个轻量级的、易于使用的工具，所以cv::createTrackbar()不会区分滑动条的名称和真正显示在屏幕上的滑动条旁边的标签，你也许已经注意到cv::namedWindow()也不区分窗口的名称以及真正在窗口上显示的名称。

可以新建一个名为"Example4-out"的窗口开始，并且在这个窗口显示处理结果。所以，在我们调用cv::imshow()在输入窗口并显示新获得的图像之后，我们就可以在output窗口计算并且显示处理结果。

示例2-5：加载图像并且在显示之前平滑处理图像

```
#include <opencv2/opencv.hpp>

void example2_5( const cv::Mat & image ) {

  // Create some windows to show the input
  // and output images in.
  //
  cv::namedWindow( "Example2_5-in", cv::WINDOW_AUTOSIZE );
  cv::namedWindow( "Example2_5-out", cv::WINDOW_AUTOSIZE );

  // Create a window to show our input image
  //
  cv::imshow( "Example2_5-in", image );

  // Create an image to hold the smoothed output
  //
  cv::Mat out;

  // Do the smoothing
  // ( Note: Could use GaussianBlur(), blur(), medianBlur() or bilateralFilter(). )
  //
  cv::GaussianBlur( image, out, cv::Size(5,5), 3, 3);
  cv::GaussianBlur(   out, out, cv::Size(5,5), 3, 3);

  // Show the smoothed image in the output window
  //
  cv::imshow( "Example2_5-out", out );

  // Wait for the user to hit a key, windows will self destruct
  //
  cv::waitKey( 0 );

}
```

第一个cv::imshow()的调用和之前我们的使用方法没有什么差别。但是下一个调用的过程中，我们申请了另一个图像结构，接下来，C++对象cv::Mat使我们的工作变得轻松了。我们只需要实例化一个输出矩阵out，这将由程序自行管理并在合适的时候自动分配，释放内存。为了让这点更清晰，我们将其输入到两个cv::GaussianBlur()函数中。在第一次调用中，输入图像被5×5大小的高斯核模糊并且被写入out变量中。高斯核的大小必须是奇数，因为高斯卷积会在其覆盖区域的中心输出结果。在下一次调用cv::GaussianBlur()的时候，由于被分配了临时的存储空间，所以out可以同时作为输入和输出。执行了两次模糊操作的图像被作为结果输出，在终止之前等待用户进行键盘事件，然后在对象生命周期结束的时候清理申请的内存。

不那么简单的变换

接下来，我们要了解如何做更多有意思的事情。在示例2-5中，我们没有任何特殊目的地使用了高斯模糊。现在我们将使用高斯模糊来对一张图像实现基2的降采样[Rosenfeld80]。如果我们对图像进行多次降采样，就要建立一个尺度空间（也称为“图像金字塔”），这一方法是计算机视觉用于处理传感器和目标尺度变化的常用手段之一。

对于那些了解信号处理和香农-奈奎斯特采样理论[Shannon49]的读者，对信号的降采样(在本例中，我们创建一个图像并对每个像素进行采样)等效于和一系列脉冲函数进行卷积(将这些函数视为“峰值”)。这样的采样会把高频分量引入输出信号（图像）。为了避免这样的事情发生，我们希望首先通过一个高通滤波器来限制信号带宽，使其能够在采样频率之内。在OpenCV中，高斯模糊以及降采样通过cv::pyrDown()函数来实现，我们的实现在示例2-6中。

示例2-6：使用cv::pyrDown()来创建一个新的图像，其宽高均为原始图像的一半

```
#include <opencv2/opencv.hpp>

int main( int argc, char** argv ) {

  cv::Mat img1,img2;
  cv::namedWindow( "Example1", cv::WINDOW_AUTOSIZE );
  cv::namedWindow( "Example2", cv::WINDOW_AUTOSIZE );

  img = cv::imread( argv[1] );
  cv::imshow( "Example1", img1 );

  cv::pyrDown( img1, img2);
  cv::imshow( "Example2", img2 );

  cv::waitKey(0);

  return 0;

};
```

然后，让我们来分析一段相似但更加复杂的代码，示例2-7这段代码包含有Canny边缘检测器[Canny86]cv::Canny()。在示例2-7中，边缘检测器通过cv::cvtColor()函数生成一个和原图一样大小但只有一个通道的图像，从而将图像从BGR图像转换为灰度图，这个操作在OpenCV中定义为宏cv::COLOR_BGR2GRAY。

示例2-7：Canny边缘检测器输出一个单通道的（灰度）图像

```
#include <opencv2/opencv.hpp>

int main( int argc, char** argv ) {
```

```
cv::Mat img_rgb, img_gry, img_cny;

cv::namedWindow( "Example Gray",  cv::WINDOW_AUTOSIZE );
cv::namedWindow( "Example Canny", cv::WINDOW_AUTOSIZE );

img_rgb = cv::imread( argv[1] );

cv::cvtColor( img_rgb, img_gry, cv::COLOR_BGR2GRAY);
cv::imshow( "Example Gray", img_gry );

cv::Canny( img_gry, img_cny, 10, 100, 3, true );
cv::imshow( "Example Canny", img_cny );

    cv::waitKey(0);

}
```

这时，我们可以将很多简单的操作串联起来。比如说，如果我们想要收缩两次图像然后寻找收缩过两次的图像中的边缘，我们可以像示例2-8一样处理。

示例2-8：在一个简单图像处理流程中结合图像金字塔操作（两次）和Canny边缘检测器

```
cv::cvtColor( img_rgb, img_gry, cv::BGR2GRAY );
cv::pyrDown( img_gry, img_pyr );
cv::pyrDown( img_pyr, img_pyr2 );
cv::Canny( img_pyr2, img_cny, 10, 100, 3, true );
// do whatever with 'img_cny'
//
...
```

在示例2-9中，我们展示了一个简单的方法来读写示例2-8的像素值。

示例2-9：读写示例2-8中的像素值

```
int x = 16, y = 32;
cv::Vec3b intensity = img_rgb.at< cv::Vec3b >(y, x);

// ( Note: We could write img_rgb.at< cv::Vec3b >(x,y)[0] )
//
uchar blue  = intensity[0];
uchar green = intensity[1];
uchar red   = intensity[2];

std::cout << "At (x,y) = (" << x << ", " << y <<
          "): (blue, green, red) = (" <<
          (unsigned int)blue <<
          ", " << (unsigned int)green << ", " <<
          (unsigned int)red << ")" << std::endl;

std::cout << "Gray pixel there is: " <<
             (unsigned int)img_gry.at<uchar>(y, x) << std::endl;

x /= 4; y /= 4;
```

```
std::cout << "Pyramid2 pixel there is: " <<
             (unsigned int)img_pyr2.at<uchar>(y, x) << std::endl;

img_cny.at<uchar>(x, y) = 128; // Set the Canny pixel there to 128
```

从摄像头中读取

“视觉”在计算机世界中可以表示很多东西，在一些情况下，我们分析从任意地方加载的静止的图像。在另一些情况下，我们会分析从硬盘中读取的视频。然而在更多的情况下，我们想要和从某种摄像头中读取的实时数据流进行交互。

OpenCV——或更准确地说，OpenCV中的HighGUI模块——为我们提供了一个简单的方式来驾驭这种情况。这个方法很接近于我们使用cv::VideoCapture从硬盘读取视频的方式，事实上，cv::VideoCapture对磁盘上的文件和摄像头是有一致接口的。对于前者来说，需要给它一个指示读取文件名的路径，对于后者来说，需要给它一个相机ID号（如果只有一个摄像头连接，这个ID号通常为0），ID的默认值是-1，这意味着“随意选择一个”，当然，当只有一个摄像头可以选择的时候这能够很好地工作（参考第8章，查阅详细解释），示例2-10展示了从文件或者摄像头中读取视频。

示例2-10：同一个对象可以读取视频文件，也可以连接摄像头

```
#include <opencv2/opencv.hpp>
#include <iostream>

int main( int argc, char** argv ) {

  cv::namedWindow( "Example2_10", cv::WINDOW_AUTOSIZE );

  cv::VideoCapture cap;
  if (argc==1) {
    cap.open(0);            // open the first camera
  } else {
    cap.open(argv[1]);
  }
  if( !cap.isOpened() ) {  // check if we succeeded
    std::cerr << "Couldn't open capture." << std::endl;
    return -1;
  }

  // The rest of program proceeds as in #simple_opencv_program_for_playing_a_vide
  ...
```

在示例2-10中，如果一个文件名被提供，OpenCV将如示例2-3所示打开指定的文件。如果没有给予任何文件名，程序将尝试打开一个摄像头。我们加入一段检查代码来确定程序是否真的开启了一些东西，如果没有，将报错。

写入AVI文件

在许多应用中，我们都希望记录一个输入流或完全不同的一些图像到视频流中。OpenCV提供一种简单的方法来实现这一点。正如我们创建一个从视频流中捕获帧的对象一样，我们可以创建一个写入对象以便将帧依次输入到一个视频文件中。允许我们进行这个工作的对象是cv::VideoWriter。

一旦将其实例化，我们就可以将每一帧图像输入到cv::VideoWriter对象中。然后在完成写入之后调用cv::VideoWriter.release()方法。为了让事情更有趣一些，示例2-11展示了一个程序，该程序会打开一个视频文件，读取它的内容后将其转换为对数极坐标（log-polar）形式（类似于人眼真正捕捉到的图像的形式，详细信息请参阅第11章），然后将对数极坐标图像写入一个新的视频文件。

示例2-11：一个完整的读取彩色视频并转换为对数极坐标视频的程序

```
#include <opencv2/opencv.hpp>
#include <iostream>

int main( int argc, char* argv[] ) {

  cv::namedWindow( "Example2_11", cv::WINDOW_AUTOSIZE );
  cv::namedWindow( "Log_Polar",   cv::WINDOW_AUTOSIZE );

  // ( Note: could capture from a camera by giving a camera id as an int.)
  //
  cv::VideoCapture capture( argv[1] );

  double fps = capture.get( cv::CAP_PROP_FPS );
  cv::Size size(
    (int)capture.get( cv::CAP_PROP_FRAME_WIDTH ),
    (int)capture.get( cv::CAP_PROP_FRAME_HEIGHT )
  );

  cv::VideoWriter writer;
  writer.open( argv[2], CV_FOURCC('M','J','P','G'), fps, size );

  cv::Mat logpolar_frame, bgr_frame;
  for(;;) {

    capture >> bgr_frame;
    if( bgr_frame.empty() ) break; // end if done

    cv::imshow( "Example2_11", bgr_frame );

    cv::logPolar(
      bgr_frame,                    // Input color frame
      logpolar_frame,               // Output log-polar frame
      cv::Point2f(                  // Centerpoint for log-polar transformation
        bgr_frame.cols/2,           //  x
        bgr_frame.rows/2            //  y
      ),
```

```
      40,                               // Magnitude (scale parameter)
      cv::WARP_FILL_OUTLIERS            // Fill outliers with 'zero'
    );

    cv::imshow( "Log_Polar", logpolar_frame );
    writer << logpolar_frame;

    char c = cv::waitKey(10);
    if( c == 27 ) break;          // allow the user to break out
  }
  capture.release();
}
```

这个程序里面有很多我们非常熟悉的元素。首先是打开视频并且读取一些在cv::VideoWriter建立时用到的属性（每秒播放的帧数以及图像宽高）。在此之后，我们将会从cv::VideoReader中逐帧读取视频，并将每一帧转换为对数极坐标形式，然后将转换的对数极坐标图像写入新的视频文件，循环操作直到源文件读完或者用户按下Esc键。

cv::VideoWriter的调用有几个需要理解的参数。第一个参数是新建视频文件的文件名，第二个参数是视频编码方式，指明视频将以何种方式进行压缩。现在有非常多的编码器可以选择，但是选择的任何编码器都必须确保可以在你的机器上使用（编码器是与OpenCV分开安装的）。在我们的例子中，我们选择了通用的MJPG编码器，我们通过使用OpenCV所提供的的宏CV_FOURCC()指定它，这个宏将四个字符作为参数，所有编码器都有类似的四个字符作为其标识。比如本例中用于运动jpeg图像编码的字符为“MJPG”，所以我们指定CV_FOURCC('M', 'J', 'P', 'G')。接下来的参数是帧率以及图像的大小。在我们的例子中，这两个值和原始视频一致。

小结

在进入下一章之前，我们应该花些时间来盘点一下我们所处的阶段，并看看未来会发生什么。我们已经看到，OpenCV API为我们提供了各种易于使用的工具，用于读取和写入静态图像和视频以及从相机捕捉视频。我们还看到，库包含用于操作这些图像的基本功能。我们还没有看到库的强大元素，它允许对整个抽象数据类型进行更复杂的操作，这些数据类型对于解决实际的视觉问题非常重要。

在接下来的几章中，我们将更深入地探究基础知识，更详细地了解与接口相关的函数和图像数据类型。我们将研究原始图像处理操作符，然后再进行一些更高级的操作。此后，我们将准备探索API为各种任务提供的多种专门服务，如相机的校准、跟踪和识别。准备好了吗？赶紧上车吧!

练习

下载并安装OpenCV，如果之前还没有这么做的话。系统性地浏览它的文件夹结构。尤其是*doc*文件夹，在那里可以加载*index.html*，它可以链接到库的主文档。在此之后，浏览库的主要部分：core模块包含基础数据类型以及算法，improc模块包含图像处理和视频处理算法，ml模块包含机器学习和聚类算法，highgui模块包含输入输出功能。检查.../*samples/cpp*文件夹，那里有许多有用的例子。

1. 使用这本书或者*http://opencv.org*所提供的的安装、构建指令，在debug模式和release模式下编译这个库。这也许会花一些时间，但是你需要编译得到的库和*dll*文件。同时注意设置*cmake*编译位于.../*opencv/samples/*文件夹下的例子。

2. 切换到.../*opencv/samples/*的编译输入目录（对我们来说，位于.../*trunk/eclipse_build/bin*）然后查看*lkdemo.cpp*（这是一个运动追踪的示例）。连接上一个相机，然后运行代码，它会提示选择窗口，按下**r**来初始化追踪。你可以通过用鼠标点击视频中的位置来添加点。也可以通过按下**n**来切换到只查看追踪点的模式。再次按下**n**会在“白天”和“夜晚”模式中切换。

3. 将示例2-11中的代码和示例2-6中的代码连接起来，建立一个读取视频并存储降采样后彩色图像的程序。

4. 修改练习3中的程序，并结合示例2-2中窗口显示的代码来显示处理的图像。

5. 修改练习4中的代码，添加一个示例2-4中的滑动条，这样一来，用户可以动态控制金字塔的降采样等级（从2到8）。可以跳过存储的步骤，但要将处理结果显示出来。

第3章

了解OpenCV的数据类型

基础知识

在接下来的几章中， 我们会学习OpenCV中的几种基础数据类型，涵盖最基本的原语容器到用于处理像图像与大型矩阵这样的数据结构。在这期间，我们还要介绍许多有用的函数，使得我们可以用一系列有效的方式来处理这些数据。本章中，我们将从学习基础数据类型开始，学习OpenCV库提供的一些有用的工具函数。

OpenCV的数据类型

OpenCV有很多数据类型，它们都基于一些重要视觉概念的抽象而设计，以此提供相对简单、直观的表示和处理。同时，许多算法开发者需要一些相对有效的，可以推广或拓展以满足他们特定需求的基本数据结构。OpenCV库使用基础数据类型模板构建并特化这些模板，从而使每个人能更简单地执行操作并满足自己的需求。

从组织结构的角度来看，OpenCV的基础数据类型主要分为三类。第一类是直接从C++原语中继承的基础数据类型（如`int`和`float`等）。这些类型包括简单的数组和矩阵，同时也代表一些简单的几何概念，比如点、矩形、大小等；第二类是辅助对象。这些对象代表更抽象的概念， 比如垃圾收集指针类、用于数据切片的范围对象（range objects）以及抽象的终止条件类等。第三类可以称为大型数组类型。这些对象原本目的是涵盖数组或一些其他的原语、程序集或更常见的基础数据类型。这一类的典型代表是`cv::Mat`类，该类用来代表任意维度的包含任意基础元素的数组。存储图片对象是`cv::Mat`类的特殊用途，但不同于早期OpenCV版本（比如，2.1以前版本），这样的专门应用不需要不同的类或类型。除了`cv::Mat`, 这个类别还涵盖其相关的类型，比如

稀疏矩阵cv::SparseMat，顾名思义，它更适用于直方图这样的稀疏数据。cv::Mat和cv::SparseMat类将在下一章主要介绍。

除了这些类型，OpenCV还使用了很多标准模板库（STL）。OpenCV相当依赖vector类，许多OpenCV库函数在参数列表中有vector模板类。我们不会在这本书中介绍STL，[注1]但会在必要的时候解释相关功能。如果已经对STL很熟悉，很快就能够适应OpenCV的模板机制。

基础类型概述

最简单的基本数据类型是模板类cv::Vec<>，它是一个原语的容器类，[注2]也就是我们所称的固定向量类。为何不直接使用STL的类呢？它们之间主要的不同点在于，固定向量类用于在编译时已经知道维度的小型向量。这允许代码以更高的效率处理与此类向量相关的操作。在实际使用中，“小”意味着，如果用来处理多个元素，说明你可能错误使用了这个类。（事实上，在2.2版本中，元素的数量在任何实例中都不能超过9）。在下一章，我们会介绍cv::Mat类，这是一个可以用来处理任何维度的数的大型数组，但现在，我们把固定向量类看作是处理小型向量的一种更加方便和快捷的手段。

即使cv::Vec<>是模板，但大部分的时间你都不会倾向于使用它的这个形式。相对的，有很多它的别名（typedef）以便用于通用的实例。它们有着cv::Vec2i，cv::Vec3i和cv::Vec4d这样的名称（分别对应于2个元素的整型向量、3个元素的整型向量和4个元素的双精度浮点向量）。任何有着cv::Vec{2,3,4,6}{b,w,s,i,f,d}形式的声明，对于2个到6个维度的6种数据类型的任何组合都是有效的。[注3]

除了固定向量类，库中还有固定矩阵类。它们与模板类cv::Matx<>相关联。与固定向量类一样，cv::Matx<>并不是用于大型数组的，相反，它是设计用于一些特定的小型矩阵操作。在计算机视觉中，有很多的2×2或3×3的矩阵，同时有少量的4×4矩阵用于各种

注1：不熟悉标准模板库（STL）的读者可以找到许多出色的在线参考资源。此外，作者特别推荐Nicolai M. Josuttis（尼古拉·约祖蒂斯）的《The C++标准库》（第2版由侯捷翻译）或Scott Meyers（司各特·梅耶斯）的《Effective STL中文版》（由潘爱民翻译）。

注2：事实上，这是一种简单化，我们会在后面的章节中介绍。cv::Vec<>是一个适用任何类型的容器，并使用模板来创建此功能。所以，cv::Vec<>可以包含其他类对象，不论是来自OpenCV还是别的地方。在大部分使用场合，cv::Vec都被用作一个C语言原语类型（如int或float）的容器。

注3：这里提到的6种数据类型在库中一般缩写为b = unsigned char, w = unsigned short, s = short, i = int, f = float, d = double。

变换。cv::Matx<>设计用于容纳这些对象。与cv::Vec<>类似，cv::Matx<>一般都以别名cv::Matx{1,2,3,4,6}{1,2,3,4,6}{f,d}的形式应用。关于固定矩阵类（类似固定向量类，而不是下一章要讲的cv::Mat类），有一点需要注意的是，固定矩阵类的维度在编译之前必须已知。当然，正是这种知识使得固定矩阵类的运算效率高，同时消除了许多动态内存分配操作。

与固定向量类关系非常类似的是cv::Point类，它是两到三个原语类型的容器。Point类不是从固定向量类继承下来的，而是由自己的模板派生的，但同时它们也可以从固定向量类转换得到。Point类与固定向量类之间最大的不同是它们的成员是通过名称变量访问的（mypoint.x，mypoint.y等）而不是通过下标访问（myvec[0]，myvec[1]等）。和cv::Vec<>一样，Point类是通过别名调用作为一个正确模板的实例。这些别名可以是cv::Point2i，cv::Point2f，cv::Point2d或cv::Point3i，cv::Point3f，cv::Point3d。

cv::Scalar本质上是一个四维Point类。与Point类相似，cv::Scalar实际上与一个可以产生任意四元素向量的模板相关，但cv::Scalar一般是双精度四元素向量的别名。不像Point类，cv::Scalar对象的元素是通过整数下标来访问的，这与cv::Vec<>相同。这是因为cv::Scalar直接继承cv::Vec<>实例（更准确地说，继承自cv::Vec<double,4>）。

我们下一个要讲的是cv::Size和cv::Rect。与Point类相似，这两个类继承自自己的模板。它们的主要区别在于cv::Size有width和height这两个属性而不是x和y，而cv::Rect包含了这四个。cv::Size类实际上是cv::Size2i的别名，在宽度和高度为整数的情况下，它本身就是一个更通用的模板的别名。对于浮点类型的宽度和高度，使用别名cv::Size2f。类似的，cv::Rect是一个整数类型的矩形的别名。还有一个类用来表示非轴对称的矩形，它便是cv::RotateRect，它含有一个cv::Point2f类型的中心点，一个cv::Size2f类型的size，还有一个额外的浮点类型的角度。

深入了解基础类型

每一个基础类型实际上都是一个相对复杂的类，支持它自己的接口函数、重载以及类似操作。在这一节，我们将更全面地看看每个类型所提供的东西，同时还要看看到这些看起来相似的类型应该如何区分。

在研究这些类时，我们将重点关注它们的接口而不是太具体的的细节。同时，我们会提供一个例子来告诉你能对这些对象做什么和不能做什么。对于一些底层的细节，建议查询 *.../opencv2/core/core.hpp*文件。

Point类

作为OpenCV的基础类型，Point类可能是最简单的。正如我们之前提到的，它们的实现

是基于一个基础模板结构，所以它们可以成为任何类型的点，如整型和浮点型等。这里实际上有两个这样的模板，一个是给二维点提供的，另一个是给三维点提供的。Point类的巨大优势在于它们简单且开销小。原生上并没有对它们定义很多操作，但它们可以转换成更广义的类型，比如在需要的时候可以转换为固定向量类或者是固定矩阵类（接下来要讨论到）。

在大多数程序中，Point类是通过类似于cv::Point2i和cv::Point3f这类别名来实例化的，这些别名的最后一个字母表示构造该点所需要的原语（在这里，b是一个无符号字符，s是一个短整型，i是一个32位整型，f是一个32位浮点数，d是一个64位浮点数）。

表3-1是Point类原生支持的函数列表（相对较短）。注意，这里有几个非常重要的受支持的操作，但它们是间接地通过隐式转换到固定向量类而受支持（在固定向量类中介绍）。这些操作显著包含所有的向量（vector）和标量(singleton)，[注4]重载了线性代数的操作符以及比较操作符。

表3-1：直接受Point类支持的操作

操作	示例
默认构造函数	cv::Point2i p; cv::Point3i p;
复制构造函数	cv::Point3f p2(p1);
值构造函数	cv::Point2i(x0,x1); cv::Point3d p(x0, x1, x2);
构造成固定向量类	(cv::Vec3f) p;
成员访问	p.x, p.y, // and for three-dimensionalpoint classes: P.Z
点乘	float x = p1.dot(p2);
双精度点乘	double x = p1.dot(p2);
叉乘	p1.cross(p2); // (for three-dimensional point classes only)
判断一个点*p*是否在举行*r*内	p.inside(r); // (for two-dimensional point classes only)

这些类型可以与C接口类型CvPoint和CvPoint2D32f互相转换。若一个Point类的浮点值实例被转换成整型，那么那个值将被自动取整。

注4：你可能以为我们会在这里使用*scalar*一词，但是我们避免使用它，因为cv::scalar是库中已经存在的类。并且很快你就会看到，cv::scalar是一个四个数组成的数组，与含有四个元素的cv::vector非常接近。在本章的语境下，*singleton*表示“任何类型的向量都是一个数组的实例”。

cv::Scalar类

cv::Scalar是四维点类。与其他类型类似，它实际上与一个模板类相关，但访问它的别名返回一个该模板的实例，在这个模板中，所有的成员都是双精度浮点数。在计算机视觉中，cv::Scalar类有一些与四元向量相关的特殊成员函数。表3-2列举了一些cv::Scalar支持的操作。

表3-2：cv::Scalar类支持的操作

操作	示例
默认构造函数	`cv::Scalar s;`
复制构造函数	`cv::Scalar s2(s1);`
值构造函数	`cv::Scalar s(x0); cv::Scalar s(x0, x1, x2, x3);`
元素相乘	`s1.mul(s2);`
（四元数）共轭	`s.conj(); //(returns cv::Scalar(s0, -s1, -s2, -s2));`
（四元数）真值测试	`s.isReal(); //(returns true, if s1==s2==s3==0)`

你将注意到，对于cv::Scalar，转换成固定向量类的操作并没有出现在表3-2中（但在表3-1中有出现）。这是因为，不像Point类，cv::Scalar直接从固定向量类模板实例中继承而来。因此，它继承了所有的向量代数操作、成员访问函数（比如，操作符[]）和一些固定向量类的特性。我们之后会介绍固定向量类，但现在，只需要记住cv::Scalar是四维双精度向量的快速表示，它有很少几个对各种类型的四维向量有用的成员函数。

cv::Scalar类可以与旧式C语言接口CvScalar自由地互换。

size类

size类在实际操作时与Point类相似，而且可以与Point类互相转换。这两者之间主要的区别在于Point类的数据成员是x和y，而size类中对应的成员是width和height。size类的三个别名分别是cv::Size，cv::Size2i，cv::Size2f。前面两个是等价的，表示整数大小，而最后一个是表示32位浮点大小。与Point类相似，size类可以与对应的旧类型的OpenCV类（CvSize和CvSize2D32f）互相转换。表3-3列举了size类支持的操作。

表3-3：size类直接支持的操作

操作	示例

默认构造函数	`cv::Size sz;`
	`cv::Size2i sz;`
	`cv::Size2f sz;`

复制构造函数	`cv::Size sz2(sz1);`

表3-3：size类直接支持的操作（续）

操作	示例
值构造函数	`cv::Size2f sz(w, h);`
成员访问	`sz.width; sz.height;`
计算面积	`sz.area();`

不像Point类，size类不支持转换到固定向量类。这意味着size类有更受限的用途。另一方面，Point类和固定向量类毫无疑问可以被转换成size类。

cv:Rect 类

矩形类包含Point类的成员x和y（矩形左上角）和size类的成员width和height（代表了矩形的大小）。然而矩形类并不是从Point类或size类继承过来的，所以它并没有从它们那里继承操作（表3-4）。

表3-4：cv::Rect直接支持的操作

操作	示例
默认构造函数	`cv::Rect r;`
复制构造函数	`cv::Rect r2(r1);`
值构造函数	`cv::Rect(x,y,w,h);`
由起始点和大小构造	`cv::Rect(p, sz);`
由两个对角构造	`cv::Rect(p1, p2);`
成员访问	`r.x; r.y; r.width; r.height;`
计算面积	`r.area();`
提取左上角	`r.tl();`
提取右下角	`r.br();`
判断点*p*是否在矩形*r*内	`r.contains(p);`

转换操作符和复制构造函数的目的是让cv::Rect能从旧的类型cv::CvRect转换。cv::Rect实际上是用整数实例化的一个矩形模板的别名。

表3-5显示，cv::Rect也支持一系列的覆写操作符，它们可以用于计算两个矩形、一个矩形或别的对象的各种各样的几何特性。

表3-5：cv::Rect对象的覆写操作符

操作	示例
矩形*r1*和矩形*r2*的交集	`cv::Rect r3 = r1 & r2;  r1 &= r2;`
同时包含矩形*r1*和矩形*r2*的最小面积矩形	`cv::Rect r3 = r1 | r2; r1 |= r2;`

表3-5：cv::Rect对象的覆写操作符（续）

操作	示例
平移矩形*r*x个数量	`cv::Rect rx = r + x; r += x;`
扩大矩形*r*s大小	`cv::rect rs = r + s; r += s;`
比较矩形*r1*与矩形*r2*是否相等	`bool eq = (r1 == r2);`
比较矩形*r1*和矩形*r2*是否不相等	`bool ne = (r1 != r2);`

cv::RotatedRect类

cv::RotatedRect类是OpenCV中少数底层没有使用模板的C++接口类之一。同时，它是包含一个中心点cv::Point2f、一个大小cv::Size2f和一个额外的角度float的容器。其中浮点类型（float）的角度代表图形绕中心点旋转的角度。cv::RotatedRect和cv::Rect有一个非常重要的不同点是cv::RotatedRect是以中心为原点的，而cv::Rect则位于左上角为原点。表3-6列举了cv::RotatedRect类直接支持的操作。

表3-6：cv::RotatedRect直接支持的操作

操作	示例
默认构造函数	`cv::RotatedRect rr();`
复制构造函数	`cv::RotatedRect rr2(rr1);`
从两个点构造	`cv::RotatedRect(p1, p1);`
值构造函数，需要一个点（point）、一个大小（size）和一个角度（angle）	`cv::RotatedRect rr(p, sz, theta);`
成员访问	`rr.center, rr.size, rr.angle`
返回四个角的列表	`rr.points(pts[4]);`

固定矩阵类

固定矩阵类是为编译时就已知维度的矩阵打造的，这也是它称为“固定”的原因。因为它们内部的所有数据都是在堆栈上分配的，所以它们的分配和清除都很快。对固定矩阵类的操作运行很快，而且它们还在小型矩阵（2×2，3×3，等等）上做过特别的优化。固定矩阵类同时也是在OpenCV的C++接口中基本类型的核心。固定向量类继承自固定矩阵类，而其他类的重要操作，要么是继承自固定向量类（类似cv::Scalar），要么是依赖于转换成固定向量类。通常，固定矩阵类实际上是一个模板。这个模板称为cv::Matx<>，但独立的矩阵则通常通过别名分配。这些别名的基础格式是

cv::Matx{1,2,...}{1,2,...}{f,d}，其中的数字可以是1到6之间的任何数，尾部的字母与之前的类型一样。[注5]

一般来说，当你要表示一个用来处理矩阵代数的矩阵的时候，你就会用到固定矩阵类。而如果你的对象是一个像图像或大型点列表这样的大数组，使用固定矩阵类就不是个好主意。这种情况下你应该使用cv::Mat（我们将在下一章介绍）。固定矩阵类是你在编译的时候就知道矩阵大小的时候使用的（比如相机矩阵）。表3-7列举了cv::Matx支持的操作。

表3-7：cv::Matx支持的操作

操作	示例
默认构造函数	`cv::Matx33f m33f; cv::Matx43d m43d;`
复制构造函数	`cv::Matx22d m22d(n22d);`
值构造函数	`cv::Matx21f m(x0,x1); cv::Matx44d m(x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15);`
含相同元素的矩阵	`m33f = cv::Matx33f::all( x );`
全零矩阵	`m23d = cv::Matx23d::zeros();`
全一矩阵	`m16f = cv::Matx16f::ones();`
创建一个单位矩阵	`m33f = cv::Matx33f::eye();`
创建一个可以容纳另一个矩阵对角线的矩阵	`m31f = cv::Matx33f::diag();`
创建一个均匀分布的矩阵	`m33f = cv::Matx33f::randu( min, max );`
创建一个正态分布的矩阵	`m33f = cv::Matx33f::nrandn( mean, variance );`
成员访问	`m( i, j ), m( i ); // one argument for // one-dimensional matrices only`
矩阵代数运算	`m1 = m0; m0 * m1; m0 + m1; m0 - m1;`
单例（Singleton）代数	`m * a; a * m; m / a;`
比较	`m1 == m2; m1 != m2;`
点积	`m1.dot( m2 ); // (sum of element-wise // multiplications, precision of m)`
点积	`m1.ddot( m2 ); // (sum of element-wise multiplications, // double precision)`
改变矩阵形状	`m91f = m33f.reshape<9,1>();`

注5：在本书写作的时候，相关的头文件称为Core.hpp，实际上并没有包含这些整数之所有可能的组合。例如，没有1×1矩阵的别名，也没有5×5。这在以后的版本中可能会改变，也可能不会改变，但你当然不想丢弃其中的任何一个。如果真的想要一个奇数构造的矩阵，可以自己实例化模板。例如，cv::Matx < 5,5,float>。

表3-7：cv::Matx支持的操作（续）

操作	示例
变换操作符	`m44f = (Matx44f) m44d`
提取(*i,j*)处的2*2子矩阵	`m44f.get_minor<2, 2>( i, j );`
提取第*i*行	`m14f = m44f.row( i );`
提取第*j*列	`m41f = m44f.col( j );`
提取矩阵对角线	`m41f = m44f.diag();`
计算转置	`n44f = m44f.t();`
逆矩阵	`n44f = m44f.inv( method );  // (default method is // cv::DECOMP_LU)`

解线性系统	`m31f = m33f.solve( rhs31f, method )`
	`m32f = m33f.solve<2>( rhs32f, method ); // (template forma[a])`
	`// default method is DECOMP_LU)`

每个元素的乘法	`m1.mul( m2 );`

a. 当隐含的矩阵方程的右边有多个列时使用模板的形式。在这个案例中，本质上是在同时解决*k*个不同的系统。*K*的值必须作为模板参数传递给solve<>()。它还将决定结果矩阵中的列数

注意，很多固定矩阵的函数相对类是静态的（比如，你访问它们是作为类而不是特定对象的成员）。举个例子，如果想构造一个3×3的单位矩阵，这有一个方便的类函数cv::Mat33f::eye()。注意，在这个例子中，eye()不需要任何参数，因为它是类的成员，并且已经是cv::Matx<>模板的3×3实例。

固定向量类

固定向量类是从固定矩阵类派生出来的。它们其实只是为了更方便地使用cv::Matx<>。在C++继承的意义上，可以说固定向量模板cv::Vec是列为1的cv::Matx<>。为特定实例准备好的别称有这样的形式cv::Vec{2,3,4,6}{b,s,w,I,f,d}，最后一个字母意义如前文所述（另外加上w，表示unsigned short）。表3-8展示了cv::Vec支持的操作。

表3-8：cv::Vec支持的操作

操作	示例
默认构造函数	`Vec2s v2s; Vec6f v6f;`
复制构造函数	`Vec3f u3f( v3f );`
值构造函数	`Vec2f v2f(x0,x1); Vec6d v6d(x0,x1,x2,x3,x4,x5);`
成员访问	`v4f[ i ]; v3w( j ); // (operator() and operator[] both work)`
向量叉乘	`v3f.cross( u3f );`

固定向量类的最主要的便利是它可以通过单个数索引各项，以及几个特定的额外的函数

使常规矩阵运算有意义（比如叉乘）。可以从表3-8中看出，该类拥有相对少的新方法以及大量继承自固定矩阵类的方法。

复数类

还有一个应该包含在基础类型中的类，它就是复数类。OpenCV的复数类与STL复数类模板complex<>不一样，但是与之兼容，可以相互转换。它们最大的区别在于成员获取。在STL类中，虚部和实部是通过成员函数real()和imag()获取的，而在OpenCV中，直接通过成员变量re和im获取。表3-9列出了复数类支持的操作。

表3-9：OpenCV复数类支持的操作

操作	示例
默认构造函数	`cv::Complexf z1; cv::Complexd z2;`
复制构造函数	`cv::Complexf z2( z1 );`
值构造函数	`cv::Complexd z1(re0); cv::Complexd(re0,im1) ;`
复制构造函数	`cv::Complexf u2f( v2f );`
成员访问	`z1.re; z1.im;`
复共轭	`z2 = z1.conj();`

与很多基本类型一样，复数类为重要的模板取了别名，cv::Complexf和cv::Complexd分别是单精度和双精度复数的别名。

辅助对象

除了基本类型和大容器（下节我们将会介绍），还有一系列的辅助对象，它们对控制各种各样的算法（比如终止条件）和各种在容器上的操作（比如ranges或者slices）非常有用。还有一种非常重要的对象，智能指针cv::Ptr。深入cv::Ptr，我们会检查已经被集成进OpenCV的C++接口的垃圾收集系统（garbage collecting system）。这个系统将我们从对象的申请和释放中解放出来，完全不同于早期基于C语言的接口（比如版本2.1）。

cv::TermCriteria类

很多算法需要一个终止条件以确定何时退出。通常，终止条件的形式要么是达到允许的有限迭代次数（称为COUNT或MAX_ITER），要么是某种形式的误差参数（如果接近于如此程度，就可以退出，称为EPS，即epsilon的简称）。cv::TermCriteria对象把一个或两个终止条件封装，以方便传进OpenCV算法函数里。它们有三个成员变量（type，maxCount以及epsilon），都可以直接设置（它们是公有的），更常见的做法是只通过它们的构造函数TermCriteria(int type, int maxCount, double epsilon)进行设置。变量type设置为cv::TermCriteria::COUNT或者TermCriteria::EPS。

也可以把两个条件并（使用位运算符:|）在一起。cv::TermCriteria::COUNT是cv::TermCriteria::MAX_ITER的同义词，所以如果喜欢，也可以使用后者。如果终止条件包含cv::TermCriteria::COUNT，就是告诉算法在maxCount次迭代之后终止。如果终止条件包含cv::TerCritera::EPS，就是告诉算法在与算法收敛相关的某些度量降到epsilon以下后终止。[注6]type参数必须进行相应的设置以使用maxCount或epsilon。

cv::Range类

Cv::Range类用于确定一个连续的整数序列。cv::Range对象有两个元素start和end，与cv::TermCriteria类似，它们经常在构造函数cv::Range(int start, int end)中设定，范围包含初始值start，但不包含终止值end，因此cv::Range rng(0, 4)包含值0,1,2,3，但不包含4。

使用size()函数可以得到一个range类的元素数量，在上面的例子中，rgb.size()等于4。在cv::Range类中有一个成员函数empty()，用于测试一个range是否含有元素。最后，cv::Range::all()可以用在任何需要获得对象可用范围的时候。

cv::Ptr模板和垃圾收集

智能指针（smart pointer）是C++中一个非常有用的类型。[注7]这个指针允许我们创建一个对象的引用，然后把它传递到各处。你可以创建更多的对该对象的引用，然后所有这些引用都会被计数。当引用超出范围，智能指针的引用计数就会减少。一旦所有的引用（指针的实例）消失，这个对象将自动清理（释放）。而作为程序员的你，不再需要记录这些东西。

接下来我们将了解它如何工作。首先，需要为你想要封装的对象定义一个指针模板的实例。可以通过调用类似cv::Ptr<Matx33f> p(new cv::Matx33f)或者 cv::Ptr<Matx33f> p = makePtr<cv::Matx33f>()的形式实现。这个模板类的构造函数就拥有指向对应对象的指针。进行这样的操作之后，你就会得到一个智能指针p，这是一个类似指针的对象，你可以随意传递，并且像使用标准指针一样使用它（它支持operator*()和operator->()等操作）。一旦得到p，就可以创建其他相同类型的对象，而不需要把一个指向新的对象的指针传递给它们。举个例子，你可以创建Ptr<Mat33f> q，当你把p的值传递给q时，在后台的某个地方，智能指针就开始执行它的“智能”行为。你可以发现，像一个正常的指针一样，这里仍然只有一个实际上的cv::Mat33f实例，而p和q同时

注6：确切的终止标准显然依赖于算法，但文档会清楚地说明特定的算法是如何解释的。

注7：如果熟悉C++标准最近的一些内容，你会注意到OpenCV cv::Ptr<>模板和smart_ptr<>模板之间的相似性。类似地，Boost库中也有一个智能指针shared_ptr<>。它们或多或少都是相同的。

指向它。不同之处在于，p和q都知道它们只是两个指针中的一个。当p被释放（比如超出了其生命周期），q就能够知道它是原始矩阵仅存的引用。如果这时候q也要被释放，这时它的析构函数将被调用（隐式），由于q发现它是遗留下来的最后一个引用，所以它会释放它的原始矩阵。可以把这个过程想成建筑中的最后一个人离开的时候有责任关灯和关门（在这个例子中，也需要把建筑物夷为平地）。

cv::Ptr()<>模板类支持多个接口中的与智能指针的引用计数功能相关的附加函数。特别地，函数addref()和release()增加和减少指针内部的引用计数。这些都是相对危险的函数，但需要自己管理引用计数的时候还是可用的。

还有empty()函数，可用于确定一个智能指针是否指向一个已经被释放掉的对象。这种情况会发生在你对一个对象调用一次或多次release()的时候。在这种情况下，你仍然有一个智能指针，但该智能指针指向的对象可能已经被销毁了。empty()函数有第二种应用，确定智能指针对象内部对象指针是否出于其他原因而为NULL。举个例子，这种情况可能发生在你给智能指针赋值时使用了一个刚好首先返回NULL的函数（cvLoadImage()，fopen()等）。[注8]

Ptr<>中最后一个要了解的成员是delete_obj()。这是一个引用计数为零时自动调用的函数。默认情况下，这个函数定义了但不做任何事情。这样你就可以在实例化指向一个需要一些特定操作来清除指向的类对象的cv::Ptr<>时重载该函数。举个例子，假设你在使用旧类型的（2.1版本以前）的IplImage。[注9]在以前，你可能调用cvLoadImage()从硬盘中加载图像，像下面这样：

```
IplImage* img_p = cvLoadImage( ... );
```

现代版本的调用方式（仍使用IplImage而不是cv::Mat）看起来像下面这样：

```
cv::Ptr<IplImage> img_p = cvLoadImage( "an_image" );
```

或者（如果你喜欢）这种：

```
cv::Ptr<IplImage> img_p( cvLoadImage("an_image" ) );
```

注8：为了本例的目的，我们将参考IplImage和cvLoadImge()，这两种构造都来自于现在已弃用的老版本2.1之前的接口。我们不会在本书中详细讨论它们，但你需要知道，IplImage是图像的旧数据结构，而cvLoadImage()是从磁盘获取图像的旧的函数，并返回指向结果图像结构的指针。

注9：这个示例看起来有点生造，但实际上，如果你有一个很大的2.1版本之前的程序，而且想要实现版本提升，你可能发现自己需要经常这样做。

现在可以以与指针相同的方式使用img_p（也就是说，对2.1版本之前接口有经验的读者，你可以“保持自己当时的样子”）。为了更加方便的使用，这个模板实例实际已经在组成OpenCV的大量的头文件中定义好了。如果要把它搜索出来，会发现如下模板函数定义：

```
template<> inline void cv::Ptr<IplImage>::delete_obj() {
    cvReleaseImage(&obj);
}
```

变量obj是Ptr<>内的实际上指向于已分配对象的成员类变量。这种定义的结果是，你不需要释放从cvLoadImage()中得到的IplImage*指针。相反，当img_p离开范围时它会被自动释放。

这个例子从某种程度来说特殊（虽然高度相关）的情况，在这个例子中IplImage的智能指针相当明显，它是由库为你定义的。在更典型的例子中，不存在你需要的清除函数时，你需要自己定义。考虑使用一个指向FILE的智能指针创建一个文件句柄。[注10]在这个案例中，我们为cv::Ptr<FILE>模板定义自己的重载版本delete_obj()函数：

```
template<> inline void cv::Ptr<FILE>::delete_obj() {
    fclose(obj);
}
```

然后，可以下一步使用这个指针来打开一个文件并对它做任意操作，然后，让指针离开范围（在什么时间文件句柄会自己关闭）：

```
{
  cv::Ptr<FILE> f(fopen("myfile.txt", "r"));
  if(f.empty())
    throw ...;          // Throw an exception, we will get to this later on...
  fprintf(f, ...);
  ...
}
```

在最后的大括号，f离开范围，f内部的引用计数变为0，delete_obj()被f的析构函数调用，（因此）fclose()在文件句柄指针（存储在obj中）中被调用。



给大师的提示：认真的程序员可能担心引用计数的增加和减少不足以让Ptr<>模板在多线程应用保持安全。实际上，大可不必担心，Ptr<>是线程安全的。相似地，OpenCV中的其他引用计数对象在同样意义上也是线程安全的。

注10：在这里，我们的FILE指的是C语言所定义的文件句柄结构。

cv::Exception类和异常处理

OpenCV使用异常来处理错误。OpenCV定义了它自己的异常类型cv::Exception，它继承自STL的异常类std::exception。实际上，这个异常类除了在cv::命名空间内以外没有任何特别的，只是处于cv::命名空间，使其有别于其他从std::exception中继承而来的对象。

cv::Exception类型有成员code，err，func，file和line，分别由一个数字的错误码代表，对应表示产生异常的错误的字符串、错误发生的函数名、错误发生的文件和错误出现在文件中的行数。err，func和files全部都是STL字符串。

有几个内置宏用于自己生成异常。CV_Error(errorcode, description)会生成并抛出一个有固定文字描述的异常。CV_Error_(errorcode, printf_fmt_str, [printf-args])以同样的方式工作，但允许你使用类似printf格式字的符串和参数代替固定的描述。最后，CV_Assert(condition)和CV_DbgAssert(condition)都会测试程序中所设定的条件（condition），如果条件不符合，则抛出异常。遗憾的是，在较新的OpenCV版本中，这些功能只能在调试（debug）版本使用。这些宏是抛出异常的首选方法，因为它们会自动为你处理函数、文件和行。

cv::DataType<>模板

当Opencv库函数需要传递特定数据类型的概念时，它们会创建一个cv::DataType<>类型的对象。cv::DataType<>本身是一个模板，传递的实际对象是这个模板的特化。这是一个在C++里被称为*traits*的例子。这允许cv::DataType<>对象同时包含该类型的运行信息，也包含允许它在编译的时候指向相同类型的它自己定义时的typedef声明。

这听起来可能有点令人迷惑，的确是的，但这是尝试将程序运行时信息和C++编译时的信息混合所带来的不可避免的结果。[注11]以下这个为DataType定义的模板类的示例有助于说明清楚这个问题：

```
template<typename _Tp> class DataType
{
typedef _Tp value_type;
typedef value_type work_type;
typedef value_type channel_type;
typedef value_type vec_type;

enum {
  generic_type = 1,
  depth        = -1,
  channels     = 1,
```

注11：在支持变量内省和具有运行时类型检查概念的语言中，没有这种问题。

```
    fmt             = 0,
    type            = CV_MAKETYPE(depth, channels)
  };
};
```

让我们试着理解其中的含义，然后用一个例子来说明它。首先，我们可以看到cv::DataType<>是一个模板，并希望被一个叫_Tp的类特化。它有四个typedef声明，它们允许cv::DataType<>和其他相关的类型在编译的时候就从cv::DataType<>实例对象中提取出来。在模板定义中，这些都是相同的，但在我们的模板特化示例中，它们不需要这么做（一般都不应该）。下一部分是一个枚举（enum），它包含了几个部分，分别为generic_type、depth、channels的数量，[注12]格式fmt和类型type。为了看到这些部分的含义，我们将研究两个cv::DataType<>特化的例子，这些例子都来自*core.hpp*。第一个是为float类型定义的cv::DataType<>：

```
template<> class DataType<float>
{
public:
  typedef float       value_type;
  typedef value_type work_type;
  typedef value_type channel_type;
  typedef value_type vec_type;

  enum {
    generic_type = 0,
    depth        = DataDepth<channel_type>::value,
    channels     = 1,
    fmt          = DataDepth<channel_type>::fmt,
    type         = CV_MAKETYPE(depth, channels)
  };
};
```

首先需要注意的是，这是一个C++内置类型的定义。为内置类型作这样的定义是有意义的，我们可以基于此为更复杂的对象作定义。在这个例子中，value_type当然是float，work_type、channel_type和vec_type都是相同的。在下一个例子中，我们将更清楚地看到这些是什么。对于枚举中的常数量来说，这个例子很好说明。第一个变量generic_type被设为0，因为在*core.hpp*中所有类型都是零。depth变量是OpenCV使用的数据类型标识符。举个例子，cv::DataDepth<float>::value解析为常量CV_32F。入口channels是1，因为float仅仅是一个数，下一个例子中我们会见到一个替代，变量fmt给出了一个结构的单字符表示。在这个例子中，cv::DataDepth<float>::fmt解析为f。最后的入口是type，这是一个与depth相似的表示，但包含channels的数量（在这个例子中是1）。CV_MAKETYPE(CV_32F, 1)解析成CV_32FC1。

注12：如果这个构造方式对你来说难以理解，请记住，你总是可以在enum声明中为“选项”分配整数值。实际上，这是一种将一些整数常量放在编译时固定的方法。

关于DataType<>，还有一件重要的事情是与更复杂的结构实现交互。比如说允许以与传入数据类型无关的方式实现算法（比如，使用内省来确定如何处理传入数据的算法）。

考虑cv::Rect<>中的cv::DataType<>实例的例子（它自身包含了一个相似但又不专业的类型 _Tp）：

```
template<typename _Tp> class DataType<Rect_<_Tp> >
{
public:
  typedef Rect_<_Tp>                                value_type;
  typedef Rect_<typename DataType<_Tp>::work_type> work_type;
  typedef _Tp                                       channel_type;
  typedef Vec<channel_type, channels>               vec_type;

  enum {
    generic_type = 0,
    depth        = DataDepth<channel_type>::value,
    channels     = 4,
    fmt          = ((channels-1)<<8) + DataDepth<channel_type>::fmt,
    type         = CV_MAKETYPE(depth, channels)
  };
};
```

这是一个更复杂的例子。首先要注意cv::Rect自身并没有出现。回想我们之前提到的，cv::Rect实际上是一个模板的别名，而这个模板是cv::Rect_<>，所以这个模板可以被特化成cv::DataType<Rect>或形如cv::DataType<Rect_<float> >。在本例中的cv::DataType<Rect>，考虑到所有的元素都是整型，所以针对于这个案例，所有的模板参数_Tp的实例都解析成int类型。

我们可以看到value_type仅仅是在编译时用来描述cv::DataType<>的名称（即Rect）。至于work_type，被定义为cv::DataType<int>的work_type（不意外的就是int）。我们可以看见，work_type告诉我们cv::DataType<>是由什么类型的数据组成的。channel_type也是int类型。这表明如果我们想要通过这个变量表示多通道对象，它应该被表示成几个int类型的对象。最后，正如channel_type告诉我们的，如果将cv::DataType<>表示成多通道对象，vec_type将会告诉我们如何把它表示成cv::Vec<>类型对象。别名cv::DataType<Rect>::vec_type会解析成cv::Vec<int, 4>。

至于运行时常量：generic_type仍然是0，depth是CV_32S，channel是4（因为实际上有4个数，相同的原因为何vec_type实例化成大小为4的cv::Vec<>），fmt解析成0x3069（因为i是0x69），type则解析成CV_32SC4。

cv::InputArray类和cv::OutputArray类

许多OpenCV的函数都把数组作为变量并把数组作为返回值，但在OpenCV中有各种

各样的数组。我们已经看到OpenCV支持一些小的数组类型（cv::Scalar，cv::Vec，cv::Matx），还有STL的std::vector<>和将在下一章讨论的大型数组类型（cv::Mat和cv::SparseMat）。为了防止接口变得复杂，OpenCV定义了cv::InputArray和cv::OutputArray类型，它们表示支持“上面的任何一个”；同时，甚至还存在cv::InputOutputArray，指定用于就地计算的数据。

cv::InputArray和cv::OutputArray之间的主要区别在于前者假设为常量（只读），而后者是可变量。你会经常看到这两种类型被用于库例程的定义中。或许你不会尝试亲自使用它们，但当你引入库函数时，它们的存在意味着你可以使用任何数组类型，包括单一的cv::Scalar。

与cv::InputArray相关的是返回cv::InputArray的特殊的函数cv::noArray()。它的返回对象可以被传递给需要cv::InputArray的地方，以强调输入的是并没有被使用的输入量。一些函数有可选的输出数组，当你不需要对应的输出时，可以传递cv::noArray()。

工具函数

除了本章前面所提供的专门的原始数据类型外，OpenCV库还提供了一些专用功能，可用于更有效地处理计算机视觉应用中普遍出现的数学和其他问题。在库的环境中，它们被称为工具函数。工具函数包含数学操作、测试、错误生成、内存与线程处理、优化及其他的工具。表3-10列举了这些函数并概括了它们的功能，详细描述紧随其后。

表3-10：工具函数和系统函数

函数名称	描述
cv::alignPtr()	对齐指针到给定字节数
cv::alignSize()	将缓冲区大小与给定的字节数对齐
cv::allocate()	分配一个C风格的数组对象
cvCeil()[a]	近似一个浮点数*x*到不小于*x*的最近的整数（向上取整）
cv::cubeRoot()	计算一个数的立方根
cv::CV_Assert()	如果给定的条件不为真，则抛出异常
CV_Error()	构造cv::Exception（从固定的字符串）并抛出异常的一个宏
CV_Error_()	构造cv::Exception（从格式化的字符串）并抛出异常的一个宏
cv::deallocate()	释放一个c风格的数组对象
cv::error()	指示错误并抛出异常
cv::fastAtan2()	向量的二维角度的计算
cv::fastFree()	释放一个内存缓存区

表3-10：工具函数和系统函数（续）

函数名称	描述
cv::fastMalloc()	分配一个对齐的内存缓存区
cvFloor()	近似一个浮点数x到不大于x的最近的整数（向下取整）
cv::format()	以sprintf类似格式创建一个STL字符串
cv::getCPUTickCount()	从内部CPU计时器获得tick计数
cv::getNumThreads()	获得当前OpenCV使用的线程数
cv::getOptimalDFTSize()	计算要传递给cv::DFT()的数组的最适宜大小
cv::getThreadNum()	获得当前线程的索引
cv::getTickCount()	获得系统的tick计数
cv::getTickFrequency()	获得每秒的tick数
cvIsInf()	判断一个浮点数x是否无穷
cvIsNaN()	判断一个浮点数x是否不是一个数
cvRound()	近似一个浮点数x到最近的整数
cv::setNumThreads()	设定OpenCV使用的线程数
cv::setUseOptimized()	开启或关闭优化代码（SSE2等）
cv::useOptimized()	指示代码优化的启用（请见cv::setUseOptimized()）

a. 由于一些兼容性能的考虑，这个函数使用的是C风格的声明而非C++风格（参考*core .../types_c.h*），并且被声明为内联函数。其他几个C风格声明也是类似的情况。

cv::alignPtr()

```
template<T> T* cv::alignPtr(       // Return aligned pointer of type T*
  T*  ptr,                         // pointer, unaligned
  int n   = sizeof(T)              // align to block size, a power of 2
);
```

给定任何类型的指针，该函数根据如下公式计算一个相同类型的对齐指针：

```
(T*)(((size_t)ptr + n+1) & -n)
```



在某些架构下，从一个不被对象大小均匀分割的地址中读取多字节对象是不可能的（比如从32位整型中读取4比特）。在像x86这样的架构下，CPU通过多次读取并从这些读取中获取你的值来自动处理这种情况，但代价是显著降低了性能。

cv::alignSize()

```
size_t cv::alignSize(                    // minimum size >='sz' divisible by 'n'
  size_t sz,                             // size of buffer
  int n    = sizeof(T)                   // align to block size, a power of 2
);
```

给定一个数量n（一般是从sizeof()返回的值）和一个来自缓存sz的大小，cv::alignSize()计算此缓冲区应该的大小，以便包含大小为n个整数的对象。这便是最小的大于或等于sz除以n的数。计算使用了下面的公式：

```
(sz + n-1) & -n
```

cv::allocate()

```
template<T> T* cv::allocate(             // Return pointer to allocated buffer
  size_t sz                              // buffer size, multiples of sizeof(T)
);
```

cv::allocate()函数与数组形式的new相似，都分配了含n个T类型对象的C风格数组，为每个对象调用默认构造函数并返回指向数组中第一个对象的指针。

cv::deallocate()

```
template<T> void cv::deallocate(
  T*     ptr,                            // Pointer to buffer to free
  size_t sz                              // size of buffer, multiples of sizeof(T)
);
```

cv::deallocate()函数与数组形式的delete相似，都释放含n个T类型对象的c风格数组，为每个对象调用了析构函数。cv::deallocate()用于释放由cv::allocate()分配的对象。传递给cv::deallocate()的元素n必须与一开始cv::allocate()分配的对象数量相同。

cv::fastAtan2()

```
float cv::fastAtan2(                     // Return value is 32-bit float
  float y,                               // y input value (32-bit float)
  float x                                // x input value (32-bit float)
);
```

这个函数计算了x,y对的反正切并返回从圆点到指示点的角度。这个结果以0.0~360.0之间的角度表示，包括0.0但不包括360.0。

cvCeil()

```
int cvCeil(                              // Return the smallest int >= x
  float x                                // input value (32-bit float)
);
```

给定一个浮点数x，cvCeil()计算不小于x的最小整数。如果输入超出了32位整型的表示范围，则结果返回未定义（undefined）。

cv::cubeRoot()

```
float cv::cubeRoot(                    // Return value is 32-bit float
  float x                              // input value (32-bit float)
);
```

这个函数计算变量x的立方根。负数值的x也能正确处理（返回的是负数）。

cv::CV_Assert()和CV_DbgAssert()

```
// example
CV_Assert( x!=0 )
```

CV_Assert()是一个宏，它会测试传递给它的表达式，如果那个表达式是False（或0），它会抛出一个异常。CV_Assert()宏一般是要测试的。作为替代，你可以使用CV_DbgAssert()，它只在debug版本中测试。

cv::CV_Error()和CV_Error_()

```
// example
CV_Error( ecode, estring )
CV_Error_( ecode, fmt, ... )
```

CV_Error()宏允许你传递一个错误代码ecode和一个固定C风格的字符串estring，然后它们会被打包进cv::Exception，进而传递给cv::error()进行处理。如果需要在运行过程中构建消息，那么可以使用不同的宏CV_Error_()。CV_Error_()接受与CV_Error()相同的ecode，但需要一个sprintf()风格的字符串紧跟着各种变量参数，就是sprintf()所需要的。

cv::error()

```
void cv::error(
  const cv::Exception& ex                    // Exception to be thrown
);
```

这个函数大多数是由CV_Error()和CV_Error_()调用的。如果你的代码在非调试版本被编译，它会抛出一个异常ex。如果你的代码在debug版本编译，它会故意引发内存访问违规，使得调试器可以访问执行堆栈和所有的参数。

你可能不会直接调用cv::error()，但会依赖于宏CV_Error()和CV_Error_()去抛出异常。这些宏携带着你希望在异常中展示的信息，为你打包好，然后传递最终的异常结果给cv::error()。

cv::fastFree()

```
void cv::fastFree(
  void* ptr                                   // Pointer to buffer to be freed
);
```

这个例程释放由cv::fastMalloc()（下面即将提到）分配的缓存。

cv::fastMalloc()

```
void* cv::fastMalloc(                         // Pointer to allocated buffer
  size_t size                                 // Size of buffer to allocate
);
```

cv::fastMalloc()工作机制与你熟悉的malloc()类似，但因为它做了缓存区内存大小对齐，所以执行速度更快。这意味着如果传递的缓存区大小超过16比特，返回的缓存区会被对齐到16比特的边界。

cvFloor()

```
int cvFloor(                                  // Return the largest int <= x
  float x                                     // input value (32-bit float)
};
```

给定一个浮点数x，cv::Floor()函数计算不大于x的最大整数。如果输入的值超过32位整数能表示的范围，结果会是undefined。

cv::format()

```
string cv::format(                            // Return STL-string
  const char* fmt,                            // formatting string, as sprintf()
  ...                                         // vargs, as sprintf()
);
```

这个函数本质上与标准库中的sprintf()相同，但是，它不需要从访问者中获得一个字符缓存区，而是构建一个STL字符串并返回它。它对Exception()构造函数（需要STL字符串作为它的参数）格式化错误信息很有用。

```
int64 cv::getCPUTickCount( void );          // long int CPU for tick count
```

这个函数报告CPU的ticks的数量（包括但不限于x86架构）。你需要知道，这个函数的返回值在许多体系结构上很难解释。特别是，因为在多核系统中一个线程可以放在一个核中睡眠（sleep）并在另一个核中唤醒，两次对cv::getCPUTickCount()的后续调用的不同结果会产生误导或者完全没有意义。所以，除非对自己在做什么很有把握，否则最

好用cv::getTickCount()来做时间测量。[注13]这个函数对于初始化随机数生成器这样的任务很适用。

cv::getNumThreads()

```
int cv::getNumThreads( void );          // total threads allocated to OpenCV
```

返回当前OpenCV适用的线程数。

cv::getOptimalDFTSize()

```
int cv::getOptimalDFTSize( int n );     // best size array to use for dft, >= n
```

当你调用cv::dtf()时，OpenCV使用的用来计算变换的算法是传递给cv::dft()的数组的大小很敏感。首选的大小的确遵循它们生成的规则，但是这个规则相当复杂，使得每次计算数组的正确大小都很麻烦。cv::getOptimalDFTSize()的一般输入为图片的实际大小，并返回你应该传递给cv::dtf()的数组大小（规则的依据为：当图像的尺寸是2、3、5的整数倍时，计算速度最快，所以计算的方法是扩充输入的尺寸到最接近的2、3、5的倍数）。OpenCV使用此信息创建一个更大的数组，让你可以在其中复制数据并用零填充其余部分。

cv::getThreadNum()

```
int cv::getThreadNum( void );           // int, id of this particular thread
```

如果你的OpenCV库编译时添加了OpenMP支持，则会返回当前执行的线程的索引（从零开始）。

cv::getTickCount()

```
int64 cv::getTickCount( void );         // long int CPU for tick count
```

这个函数返回了与一些体系结构相关的时间的tick计数。tick的速率也是与体系结构和操作系统相关的，然而，每个tick的时间可以用cv::getTickFrequency()计算（下面会描述）。比起cv::getCPUTickCount()函数，这个函数有更多的机会被应用，因为它不会受到线程跑在哪个核和CPU频率自动节流（大部分现代的处理器出于能源管理原因而做了这件事）等底层问题的影响。

cv::getTickFrequency()

```
double cv::getTickFrequency( void );    // Tick frequency in seconds as 64-bit
```

注13：当然，如果真的知道自己在做什么，从CPU的时钟读取时间显然是更更准确的事情。

当cv::getTickCount()被用于时间分析时，一个tick的确切定义是，一般是由架构决定的。函数cv::getTickFrequenty()计算时钟时间（比如秒）和抽象的ticks之间的转换。

为了计算某些事件发生的时间（比如函数的执行），只需要在函数调用的前后调用cv::getTickCount()，两个时间相减并除以cv::getTickFrequency()的返回值就可以了。

cvIsInf()

```
int cvIsInf( double x );                    // return 1 if x is IEEE754 "infinity"
```

如果x是正负无穷，cvIsInf()的返回值就是1，否则返回0。无穷测试是由IEEE754标准提供的。

cvIsNaN()

```
int cvIsNan( double x );                    // return 1 if x is IEEE754 "Not a number"
```

如果x不是一个数，cvIsNaN()就返回1，否则返回0。NaN测试是由IEEE754标准提供的测试。

cvRound()

```
int cvRound( double x );                    // Return integer nearest to 'x'
```

给定一个浮点数x，cvRound()计算与x最近的整数。如果输入值超出32位整型表示的范围，结果就是undefined。在OpenCV 3.0中，有一个覆写的cvRound(float x)函数（cvFloor和cvCeil也有对应函数），这个函数在ARM上执行更快。

cv::setNumThreads()

```
void cv::setNumThreads( int nthreads );   // Set number of threads OpenCV can use
```

当OpenCV编译时加入了OpenMP的支持，这个函数可以设定OpenCV在并行的OpenMP区域使用的线程数。使用的线程数的默认值是CPU的逻辑核心数（例如，如果我们有四个核，每个核有两个超线程，那么默认会有8个线程）。如果nthreads设为0，线程数量会被重新定为默认值。

cv::setUseOptimized()

```
void cv::setUseOptimized( bool on_off ); // If false, turn off optimized routines
```

尽管早期版本的OpenCV依赖于外部库（比如IPP库，Intel Performance Primitives）来达到高性能的优化，比如SSE2指令集，之后的版本越来越倾向于将代码移植到OpenCV自

身。默认情况下，这些优化例程是启用的，除非你在安装库时特别禁用它。然而，你可以用cv::setUseOptimized()在任何时候开启或关闭这些优化。



对优化用法的全局标志的测试是在opencv库函数中相对较高的级别上完成的。这表示在任何其他的例程（或函数）运行的时候，你都不应该调用cv::setUseOptimized()。最好从应用程序的最高级别确切知道哪些在运行，哪些不在运行，在这种情况下，才能调用这个例程。

cv::useOptimized()

```
bool cv::useOptimized( void );    // return true if optimizations are enabled
```

在任何时候，你都可以检查全局标记的状态，这允许通过调用cv::useOptimized()来实现高性能优化。如果这些优化在当前是开启的，就会返回True，否则，这个函数会返回False。

模板结构

在本章中，我们几乎提到了现在所有的基本模板类型。事实上，大多数程序员可能在并不熟悉模板的情况下就接触了OpenCV。[注14]

OpenCV 2.1版本以及后续的版本建立在类似于STL、boost及其他类似库的模板元编程风格之上。这些库的设计十分强劲，不管是在质量上还是速度上都非常领先，并且对开发者保持了最大限度的自由。从惯例来说，OpenCV中所使用的模板结构允许算法以一种抽象的形式表达并且不需要依赖于C++的基本结构，甚至不依赖于OpenCV的基本结构。

在本章中，我们从cv::Point类开始。尽管这个类是作为基本类型介绍的，但是当你实例化一个cv::Point类的时候，实际上是在实例化一个更基础的类cv::Point_<int>。[注15]显然，这个模板不仅可以以int类型初始化 而且可以以支持如同int类型的基本操作的任何类型初始化。比如说，OpenCV提供了类型cv::Complex，你也可能使用过OpenCV的复数完全不一样的STL的复数类型std::complex。对于其他的自己构造的类型也是如此。同样的概念可以推广到cv::Scalar_<>以及cv::Rect_<>，cv::Matx_<>和cv::Vec_<>。

注14：如果你C++的编程技巧并不纯熟，可以跳过这一小段。

注15：注意后面的下划线，这是一个常见的但是不通用的用来表示模板的记法。在2.X版本，这个记法基本是通用的，而在3.x版本中在不需要特别说明的地方下划线通常会被删除。因此cv::Point_<>仍然有下划线来区别于非模板类cv::Point；而cv::Vec<>没有下划线，3.x版本已经不支持cv::Vec_<>。

当实例化自己这些模板的时候，必须提供实例化类型以建立模板以及模板的维度。通用模板的参数如表3-11所示。

表3-11：通用的固定长度模板

函数名称	描述
cv::Point_<Type T>	包含两个类型T对象的点
cv::Rect_<Type T>	位置、宽度和高度，均为类型T
cv::Vec<Type T, int H>	一个类型T的集合
cv::Matx<Type T, int H, int W>	一个H*W的类型T
cv::Scalar_<Type T>	一个包含四个类型T对象的集合（和cv::Vec<T, 4>）

在下一章中，我们将会看到大型数组类型cv::Mat和cv::SpareMat，还会以相同的方式把它与cv::Mat<> 和cv::SparseMat_<>进行比较。

小结

在这一章，我们详细学习了OpenCV库中用以处理紧凑数据的数据类型，包括点、低维向量、常用来表示颜色或者坐标的矩阵、以及在这些空间上操作的小矩阵。我们学习了用于此类对象主要是内部的模板表示以及这些模板的特化类。这些特化类最为常用。除了这些数据类之外，我们还学习了帮助对象，这些对象允许我们表示终止条件和值范围等概念。最后，我们通过提供一些实用函数来总结本章。这些函数提供了计算机视觉应用程序经常遇到的重要任务的优化实现。我们还提供了包括特殊的运算和内存管理工具的重要操作例子。

练习

1. 找到并打开.../*opencv/cxcore/include/cxtypes.h*，通读并找到很多转换帮助函数。
   a. 选择一个负的浮点数。
   b. 计算它的绝对值，近似，向上取整，向下取整。
   c. 产生一些随机数。
   d. 生成一个浮点数的cv::Point2f，把它转换成整型的cv::Point，把cv::Pointz转成cv::Point2f。
2. 紧凑型矩阵和向量类。
   a. 用cv::Mat33f和cv::Vec3f对象，对应生成一个3 × 3的矩阵和3行的向量。
   b. 可以直接把它们两个相乘吗？如果不行，为什么不行？

3. 紧凑型矩阵和向量模板类。

   a. 用cv::Mat<>和cv::Vec模板，对应生成一个3×3矩阵和3行的向量。

   b. 可以直接把它们俩相乘吗？如果不行，为什么？

   c. 尝试类型映射，用cv::Mat<>模板把向量对象映射到3×1的矩阵，看看会发生什么情况？

第4章

图像和大型数组类型

动态可变的存储

本章，我们将进入大型数组类型的世界，它们之中最主要的当属cv::Mat，这个结构可以视为是OpenCV所有C++实现的核心，OpenCV所有主要函数都或是cv::Mat类的成员，或是将cv::Mat作为参数，或是返回一个cv::Mat类型。很少有函数和这三者都没有关系。

cv::Mat类用于表示任意维度的稠密数组。在本章中，“稠密”表示该数组的所有部分都有一个值存储，即使这个值是0。对于大多数图像来说，都是以稠密数组的形式存储的；与此相对的当然还有稀疏数组，在*spare* array中被实现，稀疏数组中只有非0的数值会被存储。从结论上来说，如果数组的很多地方都是0，那么稀疏数组会非常节约内存。但是在数组比较稠密的时候，稀疏数组反而会浪费大量内存。一个常用的使用稀疏数组比稠密数组好的例子是，统计直方图大部分直方图的大部分数值都为0，我们没有必要存储所有值为0的部分。



如果对OpenCV的C接口非常熟悉（2.1版本之前的实现），可能记得IplImage类型和CvMat类型。你也许还记得CvArr在C++版本的实现中，它们都不存在了，取而代之的是cv::Mat，这意味着在转换void指针的时候不必再迟疑。总的来说，这对库接口的一致性是一个巨大的提升。

cv::Mat类N维稠密数组

cv::Mat类可以作为任意维度的数组使用，其数据可以看做是以按照栅格扫描顺序存储的n维数组。这意味着在一维数组中，元素是按顺序排列的；而在一个二维数组中，数据按行组织，每一行也按顺序排列，对于三维数组来说，所有的通道都被行填充，每一个通道同样按顺序排列。

所有的矩阵都含有一个表示它所包含数组类型的元素flag，一个表示其维度的元素dims，分别表示行和列的数目的元素rows和cols（在dims大于2的时候无效），一个指示数据真正存储位置的data指针，以及一个表示该内存区域有多少个引用的refcount元素，类似于cv::Ptr<>的引用计数器。这个成员cv::Mat像智能指针一样管理内存区域。数据实体data的结构被step[]所描述。数据数组按如下公式来排列数据：

$$\&\left(mtx_{i_0,i_1,\ldots,i_{N_d-1}N_d}\right) = mtx.data + mtx.step[0]^*i_0 + mtx.step[1]^*i_1 + \ldots$$

$$+ mtx.step[N_d-1]^*i_{N_d-1}$$

比如对于一个二维数组，这个公式可以如下简化为：

$$\&(mtx_{i,j}) = mtx.data + mtx.step[0]^*i + mtx.step[1]^*j$$

包含在cv::Mat中的数据不要求必须是简单基元。cv::Mat中的元素可以是一个简单的数字，也可以是多个数字。在包含多个数字的时候，它就被称为“多通道数组”。事实上，一个n维数组和一个$(n-1)$维多维数组有着非常接近的结构。但是在很多情况下，将数组看成一个带有数值的向量的数组是很有用的，因此OpenCV库包含对此类结构的特殊规定。[注1]

造成这个差别的原因是内存读取方式。从定义上来说，一个数组的元素是许多带有数值的向量的一部分。比如说，一个数组也许会被声明为二维三通道32位浮点型数组，在这个情况下，数组的元素就是大小为12字节的32位浮点数据。当在内存中储存的时候，数组的行可能不是绝对按顺序存储的，由于数据对齐的原因，每一行之间可能存在一个很小的间隔。[注2]n维单通道数组和$(n-1)$维多通道数组的区别在于，这种填充始终出现于每一行的末尾（比如说，每个元素中的通道总是按顺序存储的）。

注1：在2.1版本前，OpenCV的数组类型有一个需要直接声明的元素IplImage::nChannels，这个元素指明了通道的数量，因为cv::Mat结构在设计时考虑要有一个更通用性的概念，所以该信息不再直接存储在类变量中，而是由一个成员函数cv::channels()来返回。

注2：数据对齐是为了内存的存取效率。

创建一个数组

可以通过实例化变量cv::Mat来创建一个数组，通过这种方式创建的数组没有大小和数据类型。当然，之后也可以再次使用成员函数create()来申请一个内存区域。一个create的变体是通过指定行数和列数以及数据类型来配置二维数组的规模。数组的类型(type)决定了它含有什么样的元素。一个有效的数据类型需要同时指明数据的类型和通道数。所有这些数据类型都在库的头文件中声明，包括CV_{8U,16S,16U,32S,32F,64F}C{1,2,3}[注3]的多种组合。比如说，CVC_32FC3表示一个三通道的32位浮点数据。

如果愿意，也可以在第一次声明矩阵的时候指定这些事情。Cv::Mat有非常多的构造函数，其中一种和create()函数有相同的参数（该接口还存在第4个可选参数来指定如何初始化数组的所有元素），比如：

```
cv::Mat m;

// Create data area for 3 rows and 10 columns of 3-channel 32-bit floats
m.create( 3, 10, CV_32FC3 );

// Set the values in the 1st channel to 1.0, the 2nd to 0.0, and the 3rd to 1.0
m.setTo( cv::Scalar( 1.0f, 0.0f, 1.0f ) );
```

等效于：

```
cv::Mat m( 3, 10, CV_32FC3, cv::Scalar( 1.0f, 0.0f, 1.0f ) );
```



本书最重要的一段

理解数组中的数据与数组对象的解耦时，需要以辩证的观点来考虑。对象cv::Mat是数据实体的头，原则上来说，它和数据实体是完全不同的两个东西。比如，我们可以将一个矩阵n赋值给矩阵m（比如用语句m＝n）。在这里，m内部的数据指针之前所指向的数据实体m（如果存在的话）将会被释放。[注4]与此同时，m和n所共享的内存区域的引用指针会增加一个引用计数。然后，m的与数据实体相关的成员依据新数据进行更新（比如行列与数据类型）。这将引导出一个非常方便的结果，数组可以被赋值给另一个数组，并且这个工作能够在后台运行，以确保获得正确的结果。

注3：OpenCV允许定义超过3通道的数据类型，但是为了创建它，你需要调用函数CV_{8U,16S,32S,32F,64F}C()。这些函数只有一个参数，表明通道数，所以CV_8UC(3)等效于CV_8UC3，但由于没有宏CV_8UC7，所以为了调用它，你必须使用函数C_8UC(7)。

注4：一般来说，如果m是一个指向特定数据区域的cv::Mat，就会被释放。

表4-1是一个完整的关于cv::Mat构造函数的列表，它也许显得有些繁琐，但实际上大多数时候都只需要使用其中几个常用的构造函数。只是需要查阅一些不常用的构造方式的时候，这个表才会帮助到你。

表4-1：cv::Mat的构造函数（非复制构造函数）

构造函数	说明
cv::Mat	默认构造函数
cv::Mat(int rows, int cols, int type);	指定类型的二维数组
cv::Mat(int rows, int cols, int type, const Scalar&s);	指定类型的二维数组，并指定初始化值
cv::Mat(int rows, int cols, int type, void* data, size_t step=AUTO_STEP);	指定类型的二维数组，并指定预先存储的数据
cv::Mat(cv::Size sz, int type);	指定类型的二维数组（大小由sz指定）
cv::Mat(cv::Size sz, int type, const Scalar&s);	指定类型的二维数组，并指定初始化值（大小由sz指定）
cv::Mat(cv::Size sz, int type, void* data, size_t step=AUTO_STEP);	指定类型的二维数组，并指定预先存储的数据（大小由sz指定）
cv::Mat(int ndims, const int* sizes, int type);	指定类型的多维数组
cv::Mat(int ndims, const int* sizes, int type, const Scalar&s);	指定类型的多维数组，并指定初始化值
cv::Mat(int ndims, const int* sizes, int type, void* data, size_t step=AUTO_STEP);	指定类型的多维数组，并指定预先存储的数据

表4-1列举了cv::Mat的基本构造函数。除了默认构造函数，它们主要分为如下几个类型：要求输入行数和列数来构造一个二维数组的、使用cv::Size对象来构造一个二维数组的以及构造*n*维数组并且要求你通过一个整型的序列来确定每一维数据维度的。

补充说明，以上一些构造函数允许你确定初始化的数据，你既可以通过指定一个cv::Scalar（在这个情况下，所有的数据都会被初始化为这个值），也可以提供一个指向数据块的指针。对于后者来说，相当于只创建了一个指向已经存在数据的头（不会发生数据复制的行为，成员data会被设置为data参数所指向的区域）。

复制构造函数（表4-2）展示了如何从一个数组创建另一个数组。作为复制构造函数的补充，有几种方法可以从一个已经存在的数组的子区域创建一个数组，或者从一些矩阵表达中生成一个新的矩阵。

表4-2：cv::Mat从其他cv::Mat复制数据的构造函数

构造函数	描述
`cv::Mat( const Mat&mat );`	复制构造函数

`cv::Mat(`
`  const Mat&mat,`
`  const cv::Range&rows,`
`  const cv::Range&cols`
`);`
	只从指定的行列中复制数据的复制构造函数

`cv::Mat(`
`  const Mat&mat,`
`  const cv::Rect&roi`
`);`
	只从感兴趣的区域中复制数据的复制构造函数

`cv::Mat(`
`  const Mat&mat,`
`  const cv::Range* ranges`
`);`
	服务于*n*维数组的，从泛化的感兴趣区域中复制数据的复制构造函数

`cv::Mat( const cv::MatExpr&expr );`	从其他矩阵的线性代数表述中生成新矩阵的复制构造函数

子区域（也称为“感兴趣区域”，ROI）构造函数也分为三个类别，分别是一种输入为行和列的范围（这只在二维矩阵的情况下有效）、一种使用cv::Rect来指定一个矩形的子区域（这也只对于二维矩阵有效）以及最后一种输入一个range数组的。在后者中，range所指向的有效范围的维度必须和mat的维度相等。如果你的mat是一个多维（维度大于2）的数组，就得使用第三种方法进行复制。

如果还在维护基于2.1版本之前的C风格数据结构的代码，也许想要从旧的cvMat或者IplImage建立一个新的C++风格的cv::Mat结构。这种情况下，你有两种选择（表4-3）：可以在现有数据上建立一个头m（通过设置copyData为false）；也可以设置copyData为true（这样一来，新的内存区域将会为m所申请，所有数据集都会被复制到新的内存区域）。

表4-3：2.1版本之前的构造函数如何构造cv::Mat

构造函数	描述

`cv::Mat(`
`  const CvMat* old,`
`  bool copyData=false`
`);`
	从旧的CvMat结构中构造一个新的对象m并可选择是否拷贝数据

`cv::Mat(`
`  const IplImage* old,`
`  bool copyData=false`
`);`
	从旧的IplImage结构中构造一个新的对象m并可选择是否拷贝数据



这些构造函数所起的作用比你最初所理解的多很多。特别是，它允许C++和C数据类型混用的表达式，并且通过隐式的构造函数转换为需要的C++类型。因此，可以简单使用一个指向C结构的指针，只要这个结构拥有对应的C++结构，函数就会正常运转（这就是copyData默认为false的原因）。除了这些构造函数，还有相应的操作将cv::Mat转换为CvMat或者IplImage，这些操作也不会复制数据。

最后一组构造函数是带模板的构造函数（表4-4），它们也被称为“模板构造函数”，这些函数并不会从cv::Mat创建一个模板，而是根据模板创建一个cv::Mat的实例。这些构造函数允许通过模板类cv::Vec<>或cv::Matx<>来创建对应维度和类型的cv::Mat，或者使用一个STL vector<>来创建一个相同类型的数组。

表4-4：cv::Mat模板构造函数

构造函数	描述

`cv::Mat(`
`  const cv::Vec<T,n>&vec,`
`  bool copyData=true`
`);`
	构造一个如同cv::Vec所指定的数据类型为T、大小为*n*的一维数组

`cv::Mat(`
`  const cv::Matx<T,m,n>&vec,`
`  bool copyData=true`
`);`
	构造一个如同cv::Matx所指定的数据类型为T、大小为*m*×*n*的二维数组

`cv::Mat(`
`  const std::vector<T>&vec,`
`  bool copyData=true`
`);`
	构造STL的vector所指定的数据类型为T、大小为vector元素数的一维数组

cv::Mat类也提供许多静态方法来创建一些常用的数组（表4-5）。这些方法包括zeros()，ones()，eye()，对应的地，将建立一个全为0的矩阵、全为1的矩阵以及一个单位矩阵。[注5]

表4-5：构造cv::Mat的静态方法

函数	描述
`cv::Mat::zeros(rows, cols, type);`	构造一个大小为rows×cols、数据类型为type指定类型的、值全为0的矩阵
`cv::Mat::ones(rows, cols, type);`	构造一个大小为rows×cols、数据类型为type指定类型的、值全为1的矩阵
`cv::Mat::eye(rows, cols, type);`	构造一个大小为rows×cols、数据类型为type指定类型的单位矩阵

独立获取数组元素

OpenCV有几种不同的访问矩阵的方法，这些方法均是为了方便对某一种数据类型进行访问而设计。然而，最近的OpenCV版本投入大量精力使其即使不完全相同，也可以进行对比，并进行高效访问。访问一个元素的两种主要方法是通过位置或迭代器访问。

直接访问是通过模板函数at<>()来实现的。这个函数有很多种变体，对不同维度的数组有不同的参数要求。这个函数的工作方式是先将at<>()特化到矩阵所包含的数据类型，然后使用你所想要的数据的行和列的位置访问该元素。这里有一个简单的例子：

```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC1 );

printf(
  "Element (3,3) is %f\n",
  m.at<float>(3,3)
);
```

多通道数组的操作与单通道数组相类似：

```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC2 );

printf(
  "Element (3,3) is (%f,%f)\n",
  m.at<cv::Vec2f>(3,3)[0],
  m.at<cv::Vec2f>(3,3)[1]
);
```

注意，当你想要对多维数组指定一个类似于at<>()的模板函数时，最好的方式是使用cv::Vec<>对象（使用预先创建的别名或模板形式）。

注5：使用cv::Mat::eye()和cv::Mat::ones()的时候，如果要求创建的是一个多维数组，就只有第一通道会被设置为1，其余通道保持为0。

和向量的情形类似，你可以创建一个由更加精巧的元素构建的数组，比如复数：

```
cv::Mat m = cv::Mat::eye( 10, 10, cv::DataType<cv::Complexf>::type );

printf(
  "Element (3,3) is %f + i%f\n",
  m.at<cv::Complexf>(3,3).re,
  m.at<cv::Complexf>(3,3).im,
);
```

同样值得注意的是，这里使用了cv::DataType<>模板，矩阵的构造函数需要一个类型为int的值来作为运行值，这个值也恰好作为某些“魔法”值可以被构造函数所理解。相反，cv::Complexf是真正的数据对象的类型，纯粹是在编译过程中构造的。当我们需要从（编译时）的表示生成另一个（运行时的）表示时，就需要cv::DataType<>。表4-6列举了at<>()函数可用的数据类型。

表4-6：at<>()访问器函数的变体

示例	描述
M.at<int>(i);	整型数组M中的元素i
M.at<float>(i, j);	浮点型数组M中的元素(i, j)
M.at<int>(pt);	整型矩阵M中处于(pt.x, pt.y)的元素
M.at<float>(i, j, k);	三维浮点型矩阵M中处于(I, j, k)位置的元素
M.at<uchar>(idx);	无符号字符数组M中位于idx[]所索引的*n*维位置的元素

为了访问二维数组，你可以使用C风格的指针来指定某一行。这个工作由cv::Mat类的成员函数ptr<>()完成（再次强调，数组中的数据是按行连续组织的，因此不可以通过这种方式访问一个指定的列，但稍后我们可以看到访问列的正确方式）由于at<>(), ptr<>()都是模板函数，所以需要一个类型名来进行实例化。函数接收一个整型参数来指示希望指针指向的行，函数将返回一个和矩阵原始数据类型相同的数据指针（比如说，如果数组类型是CV_32FC3，它将会返回一个float*）。因此，给定一个类型为float三通道的矩阵mtx，结构体mtx.ptr<Vec3f>(3)将会返回mtx的第三行指向第一个元素第一个（浮点）通道的指针，这通常是访问数组最快的一种方式，[注6]因为一旦你拥有指针，就可以向指定的位置写入数据。

注6：使用at<>和利用指针访问的差距取决于编译器的优化程度。使用at<>进行存储性能更接近于好的优化器所能够达到的效果（尽管稍微慢一些），但是如果优化器被关闭了，其性能相较于没有优化器优化会有一个数量级的提升。而通过迭代器的访问几乎总是比这两种方法都要慢，然而在几乎所有情况下，使用内置的OpenCV函数都比你写的所有通过循环来控制直接访问的方法快，所以在任何情况下，都要避免通过循环来大量访问矩阵内部结构。



有两种方式可以获得一个指向矩阵mtx的数据区域的指针。一种是使用ptr<>()成员函数，另一种是直接使用数据指针data，然后使用成员数组step来计算地址，后者更接近于我们在C语言中所做的操作。但是一般来说，由于at<>()和ptr<>()以及迭代器的存在，这种做法已经不再推荐了。但是不得不说，直接计算地址始终是最有效率的做法，尤其是要处理多于二维的数组时。

你需要时刻记住C风格指针式存取方式的一些问题。如果想要访问一个数组中的所有东西，可能需要一次性迭代一行，因为这些行在数组中可能是连续的，但也有可能不连续。成员函数isContinuous()将告诉你数组是否被连续的打包。如果是连续的，你就可以通过获取第一行第一个元素的指针，然后在整个数组中遍历，仿佛它是一个巨大的一维数组。

另一种序列存储的方式是使用cv::Mat内嵌的迭代器机制。这种机制在某种程度上是基于STL容器所提供的机制。基础想法是OpenCV提供一对迭代器模板，一个用于只读（const）数组的和一个用于非只读的（ono-const）数组的 上述迭代器分别被命名为cv::MatConstIterator<>和cv::MatIterator<>。Cv::Mat的成员函数begin()和end()会返回这种类型的对象 因为迭代器具有足够的智能来处理连续的内存区域和非连续的内存区域，所以这种用法非常方便，不管在哪一种维度的数组中都非常有效。

所有的迭代器都必须在数组建立的时候声明并且指定一个对象类型。下面有一个简单的使用迭代器来计算三通道三维数组中“最长”元素（一个三维向量域）的例子：

```
int sz[3] = { 4, 4, 4 };
cv::Mat m( 3, sz, CV_32FC3 );  // A three-dimensional array of size 4-by-4-by-4
cv::randu( m, -1.0f, 1.0f );   // fill with random numbers from -1.0 to 1.0

float max = 0.0f;                  // minimum possible value of L2 norm
cv::MatConstIterator<cv::Vec3f> it = m.begin();
while( it != m.end() ) {

  len2 = (*it)[0]*(*it)[0]+(*it)[1]*(*it)[1]+(*it)[2]*(*it)[2];
  if( len2 > max ) max = len2;
  it++;

}
```

基于习惯，你可能会使用迭代器来进行针对整个数组的操作或者针对多个数组的访问。但是在两个数组相加。或是将RGB颜色空间的数组转换到HSV颜色空间的情况下，数组中的每个元素都会进行这样的操作。

数组迭代器NAryMatIterator

还有另一种形式的迭代器，尽管不像cv::MatIterator<>那样将不连续内存区域打包以同时处理多个数组的迭代。这个迭代器称为cv::NaryMatIterator，它只要求被迭代的数组有相同的几何结构（维度以及每一个维度的范围）。

该迭代器不会返回一个用于迭代的单独元素，而通过返回一堆数组来进行*N*-ary迭代器操作，这些返回的数组也称为“面”（plane）。一个面表示输入数组有连续内存的部分（一般来说是一维或者二维的片段）。[注7]这也是如何处理非连续内存的方法：可以一个接一个处理其中连续的内存片段，对于每一个面，你既可以通过数组操作进行处理，也可以通过自己的方式进行正常的迭代（在这里，“正常的”表示不需要检查该区域内存是否连续）。

面所包含的内容全部都是从被迭代的多维度数组的内容中分离出来的。示例4-1演示了如何按面将一个多维数组相加。

示例4-1：按面进行多维数组相加

```
const int n_mat_size = 5;
const int n_mat_sz[] = { n_mat_size, n_mat_size, n_mat_size };
cv::Mat n_mat( 3, n_mat_sz, CV_32FC1 );

cv::RNG rng;
rng.fill( n_mat, cv::RNG::UNIFORM, 0.f, 1.f );

const cv::Mat* arrays[] = { &n_mat, 0 };
cv::Mat my_planes[1];
cv::NAryMatIterator it( arrays, my_planes );
```

在这里，你有了*N*-ary迭代器。继续我们的例子，我们要计算m0和m1的和，并将结果放在m2中。我们将按面来进行这个操作：

```
    // On each iteration, it.planes[i] will be the current plane of the
    // i-th array from 'arrays'.
    //
    float s = 0.f;                                  // Total sum over all planes
    int   n = 0;                                    // Total number of planes
    for (int p = 0; p < it.nplanes; p++, ++it) {
      s += cv::sum(it.planes[0])[0];
      n++;
    }
```

在本例中，我们首先创建了一个三维数组n_mat并使用125个大小介于0到1的随机数进行填充。为了初始化cv::NAryMat迭代器对象，我们需要准备两件事。首先，我们需要指向包含指向所有我们想要迭代的cv::Mat的C风格指针数组，这个数组必须以0或者NULL

注7：事实上，面的维度并不限于二维，它可以更大，只要它所占内存区域是连续分布的。

终止。然后，我们需要另一个可以作为面的参考的C风格的cv::Mat数组，从而可以通过它们进行迭代（在这里，数组长度为1）。

我们创建了*N*-ary迭代器之后，我们就可以通过它进行迭代。回想一下，这种迭代是基于我们给迭代器的数组的面进行迭代的。面的数量（每一个数组都一样，因为它们有相同的几何结构）始终由it,nplanes所决定。*N*-ary迭代器包含有名为planes的C风格数组，它为每个输入数组保存当前平面的头。在本例中，只有一个数组被迭代，所以我们只需要引用it.plnes[0]（在这里平面只有一个且只含有一个数组）。在本例中，我们将会对每一个面调用cv::sum()并得到最终结果。

为了看到*N*-ary迭代器的实际效用，考虑将上个例子稍微拓展一下，变成两个数组相加（示例4-2）。

示例4-2：使用N-ary将两个数组相加

```
const int n_mat_size = 5;
const int n_mat_sz[] = { n_mat_size, n_mat_size, n_mat_size };
cv::Mat n_mat0( 3, n_mat_sz, CV_32FC1 );
cv::Mat n_mat1( 3, n_mat_sz, CV_32FC1 );

cv::RNG rng;
rng.fill( n_mat0, cv::RNG::UNIFORM, 0.f, 1.f );
rng.fill( n_mat1, cv::RNG::UNIFORM, 0.f, 1.f );

const cv::Mat* arrays[] = { &n_mat0, &n_mat1, 0 };
cv::Mat my_planes[2];
cv::NAryMatIterator it( arrays, my_planes );

float s = 0.f;                          // Total sum over all planes in both arrays
int   n = 0;                            // Total number of planes
for(int p = 0; p < it.nplanes; p++, ++it) {
  s += cv::sum(it.planes[0])[0];
  s += cv::sum(it.planes[1])[0];
  n++;
}
```

在第二个例子中，你可以看到名为arrays C风格的数组被每个输入序列赋予一个指针以及my_planes所提供的两个矩阵。当在面上进行迭代时，对于每一步来说，planes[0]包含来自n_mat0的一个面，planes[1]包含来自n_mat1的一个面。在这个简单的例子中，我们只是将两个面相加并添加到累加器中。在一个稍微拓展的情况下，我们可以使用针对于每个元素的加法来对两个平面求和，并将结果放在第三个平面的对应位置。

在之前的例子中没有提到但非常重要的是成员it.size。它将指明每一个面的大小，这个变量显示了面中元素的变量，所以它不会包含通道数这一要素。在之前的例子中，如果it.planes是4，it.size将会是16：

```
////////////// compute dst[*] = pow(src1[*], src2[*]) ///////////////
const Mat* arrays[] = { src1, src2, dst, 0 };
float* ptrs[3];

NAryMatIterator it(arrays, (uchar**)ptrs);
for( size_t i = 0; i < it.nplanes; i++, ++it )
{
      for( size_t j = 0; j < it.size; j++ )
      {
            ptrs[2][j] = std::pow(ptrs[0][j], ptrs[1][j]);
      }
}
```

通过块访问数组元素

在上一节，我们了解了按照独立元素和按照序列进行迭代访问的方法。还有一种常见的情况是，你可能需要将一个数组的子集作为另一个数组访问。这个子集可能是一行或者一列，也可能是原始数组的一个子集。

如表4-7所示，有很多种方法都可以让我们完成这个目的，所有的方法都是cv::Mat的成员函数，并且都返回我们所请求的数组的子集。最简单的方法是row()和col()，它将一个整型变量作为参数并返回这个变量所索引的行或者列。这个方法只适用于二维数组，但稍后我们还会看到更复杂的例子。

当你（对数组m）使用m.row()或者m.col()，或者我们要讨论的其他方法的时候，很重要的一点是你需要理解m的数据并没有被复制到新的数组中。比如表达式m2＝m.row(3)，这个表达式将创建一个新的数组头，并且分配它的data指针、step数组以及其他一些东西，这样它将可以访问m中的第三行数据。如果修改了m2中得到的数据，也会修改处于m中的数据。稍后，我们将学习copyTo()方法，这个方法会真正拷贝数据。OpenCV这样操作的原因是创建一个新数组的时间不仅很少，而且和旧数组的大小无关。

与row()还有col()密切相关的函数是rowRange()和colRange()。这些函数本质上与其表兄弟是一样的，只不过它们会从多个连续的行（列）中提取数组。你可以用两种方式调用这两个函数，既可以指定一个整数的开始和结束的行（列），也可以传递指明想要的行（列）的cv::Range对象。在输入两个整数的情况下，数据范围包括开始的索引但不包括结束索引所指向的行（列）。在使用cv::Range的时候，也会受到类似的处理。

成员函数diag()类似于row()或col()，不同的是从m.diag()返回的数组指向矩阵m的对角元素，调用m.diag()的时候，需要输入一个整型参数来说明哪一个对角需要被提取 如果参数为0，那么它将会是主对角。如果输入为一个正数，它将相对于主对角线向数组上半部分偏移。如果它是负的，它将相对于主对角线向下半部分偏移。

最后一种提取子矩阵的方式是运算符()。通过这个运算符，你可以传递一对范围（一个指示行范围的cv::Range和一个指示列范围的cv::Range）或者一个cv::Rect来指定你想要的区域。这是唯一允许你从高维矩阵中获取一个子区域的方法，这时需要有一个指向C风格的cv::Range数组的指针，并且，这个数组必须和你进行操作的数组具有相同的维度。

表4-7：cv::Mat区块访问

示例	描述
m.row(i);	m中第i行数组
m.col(j);	m中第j列数组
m.rowRange(i0, i1);	M中第i0行到第i1-1行所构成的数组
m.rowRange(cv::Range(i0, i1));	m中第i0行到第i1-1行所构成的数组
m.colRange(j0, j1);	m中第j0列到第j1-1列所构成的数组
m.colRange(cv::Range(j0, j1));	m中第j0列到第j1-1列所构成的数组
m.diag(d);	m中偏移为d的对角线所组成的数组
m(cv::Range(i0,i1), cv::Range(j0,j1));	m中从点(i0, j0)到点(i1-1, j1-1)所包含数据构成的数组
m(cv::Rect(i0,i1,w,h));	m中从点(i0, j0)到点(i0+w-1, j0+h-1)所包含数据构成的数组
m(ranges);	M中依据ranges[0]到ranges[ndim-1]所索引区域构成的数组

矩阵表达式：代数和cv::Mat

在2.1版本之后，代码迁移到C++，让运算符重载这一特性得以实现，这使得我们可以创建包含单个矩阵序列[注8]和单个矩阵的矩阵表达式。它可以让代码变得更加清晰，因为这可以将许多操作结合成为一个更加复杂和有意义的表达式。

OpenCV在后台使用许多自身特性来实现这些操作。比如说，在需要的时候自动创建矩阵头，但是工作区域的内存只有在必要的时候申请。当其不再需要的时候，数据区域会被自动释放掉。计算结果由操作符=()放到最终的结果中。然而，很重要的区别在于，操作符=()不是对cv::Mat或cvMat（这也有可能出现）的赋值，而是更接近于cv::Mat

注8：为了清晰起见，我们使用“数组”这个词来表示一般的cv::Mat的对象，用“矩阵”这个词来表示作为一个数学对象的情况。这种区别只是语义上的，并没有在OpenCV的代码结构中体现出来。

的cv::MatExpr（自身的表达[注9]）。这个区别非常重要，因为数据始终会被复制到结果（左边的）数组中去。再想想m2=m1这一操作，尽管合理，但表达的意思却完全不一样。对于后者来说，m2只是对于m1的一个引用。相比而言，m2＝m1+m0表达的意思则完全不一样。因为m1+m0是矩阵表达式，这会被计算并且将结果的指针指向m2。结果将处于一个新的内存区域中[注10]。

表4-8列出了可使用的代数操作的样例。作为简单代数表达的补充，有许多操作用于矩阵的创建（比如我们之前学习过的cv::Mat::eye()），以及一些用于计算矩阵转置和求逆的高级操作。重点是，你需要尽量简洁的使用一行代码，来组合这一系列相似但不普通的矩阵操作来描述你在计算机视觉中所面对的问题。

表4-8：矩阵表达式可用的运算操作

示例	描述
`m0 + m1, m0 - m1;`	矩阵的加法和减法
`m0 + s; m0 - s; s + m0, s - m1;`	矩阵和单个元素的加和减
`-m0;`	矩阵取负
`s * m0; m0 * s;`	Scaling of a matrix by a singleton
`m0.mul( m1 ); m0/m1;`	按元素将m0和m1相乘，按元素将m0和m1相除
`m0 * m1;`	m0和m1进行矩阵乘法
`m0.inv( method );`	对m0矩阵求逆（默认使用DECONP_LU）
`m0.t()`	对m0矩阵求转置
`m0>m1; m0>=m1; m0==m1; m0<=m1; m0<m1;`	按元素进行比较，返回元素只有0和255的uchar类型矩阵
`m0&m1; m0|m1; m0^m1; ~m0;` `m0&s; s&m0; m0|s; s|m0; m0^s; s^m0;`	矩阵和矩阵之间或者矩阵和单个元素之间按位进行的逻辑操作
`min(m0,m1); max(m0,m1); min(m0,s);` `min(s,m0); max(m0,s); max(s,m0);`	矩阵和矩阵之间或者矩阵和单个元素之间按元素取最大或者最小值
`cv::abs( m0 );`	对m0按元素取绝对值

注9：cv::MatExpr的底层机制比我们这里讨论的复杂很多，但是你可以认为cv::MatExpr是一个位于等号右侧的代数的符号表达。它的最大优点是，可以自动移除或者修改一些不必要的表达式（比如矩阵转置的转置，加0矩阵，乘以其逆矩阵等）。

注10：如果你是IT圈内一名真正的“老司机”，不会有什么感到惊讶的。显然，必须创建一个临时的数组来存储m1+m0的结果。m2实际上只是另一个引用，一个对临时数组的引用。当运算符+()推出的时候，它对临时数组的引用被丢弃，但是引用计数不为0，这时m2保留有对这个数组的唯一引用。

表4-8：矩阵表达式的运算操作（续）

示例	描述
`m0.cross( m1 ); m0.dot( m1 );`	向量叉乘和点乘操作（叉乘操作只适用于3×1矩阵）

`cv::Mat::eye( Nr, Nc, type );`
`cv::Mat::zeros( Nr, Nc, type );`
`cv::Mat::ones( Nr, Nc, type );`
	用于返回规定类型的N×N矩阵的静态方法

矩阵求逆操作inv()实际上是一系列矩阵求逆操作的接口。现阶段主要有三种操作，第一种是cv::DECOMP_LU，表示使用LU分解，这种方法对任意的非奇异矩阵都有效。第二种是cv::DECOMP_CHOLESKY，表示使用Cholesky分解 Cholesky之在矩阵半正定的时候有效，而且在处理大型矩阵的时候比LU分解快很多。最后一种操作是cv::DECOMP_SVD，使用奇异值分解（SVD）进行求逆。SVD是唯一一种在矩阵奇异或非方阵的情况下都可工作的方法（可以求出矩阵的伪逆）。

还有一些没有包含在表4-8中的操作，比如cv::norm()，cv::mean()，cv::sum()以及其他的一些函数（这些函数我们可能没有介绍过，但是你应该可以猜到它的用法）。这些函数可以将矩阵转换为其他矩阵或数值。所有这样的对象都可以在矩阵的运算表达式中使用。

饱和转换

在OpenCV中，常常有一些操作有溢出的风险。不仅在处理无符号类型数据的时候经常发生，在其他任何情况下也可能发生。为了处理这个问题，OpenCV实现了一个称为“饱和转换”（saturation casting）的构造。

这意味着OpenCV在进行算术和其他操作的时候会自动检查是否上溢出和下溢出。在这些情况下，这个库函数会将结果值转换为相对最小或者最大的可行值。记住，这并不是C语言的通常操作，也不是一般我们需要做的。

你也可能想在自己的函数实现这种特殊的行为。OpenCV提供一些手动的模板转换操作来使得这些操作对你来说更容易进行。这一操作的模板函数是cv::saturate_cast<>()，它允许你指定自己喜欢的类型来转换参数。这里有一个例子：

```
uchar&Vxy = m0.at<uchar>( y, x ); Vxy = cv::
saturate_cast<uchar>((Vxy-128)*2 + 128);}
```

在这个示例代码中，我们首先将数组m0中一个8-bit元素的引用赋值给变量Vxy，之后，我们减去128并乘以2（将其放大），最后加上128（这样，结果将是原来这个量和128距离的两倍）。常用的C运算法则会把Vxy－128赋值给一个有符号的整型，然后再和2进行整数乘法以及和128进行整数加法。注意，如果Vxy原始的量是10，那么Vxy－128将是

−118。最终的计算结果将会是−108。这个数值不属于8-bit无符号整型的范围。这就是我们使用cv::saturation_cast<uchar>()的原因，它会判定输入值−108小于无符号整型的范围，并将其转换为0。

数组还可以做很多事情

在这里，我们已经接触了cv::Mat类的大部分成员函数。当然，我们还错过了一些事情，因为这些事情并不属于之前我们所讨论的任何一种类别。表4-9列举了这些可能在OpenCV编程中用到的“遗老” 。

表4-9：cv::Mat类的更多函数成员

示例	描述
`m1 = m0.clone();`	从m0进行完全复制，该复制将复制所有的数据元素
`m0.copyTo(m1);`	将m0复制给m1，如果有必要，将给m1重分配内存空间（等同于m1=m0.clone()）
`m0.copyTo(m1, mask );`	和m0.copyTo(m1)一样，但是只复制mask所指示的区域
`m0.convertTo( m1, type, scale, offset );`	转换m0中元素的类型（比如CV_32F）并且在尺度变换（默认为1）和增加偏置（默认为0）之后赋值给m1
`m0.assignTo( m1, type );`	只在内部使用（集成在convertTo中）
`m0.setTo( s, mask );`	设置m0所有元素为s，如果存在mask，则只对mask指示区域进行操作
`m0.reshape( chan, rows )`	改变二维数组的有效形状，chan和rows变量可能为0，表示不做更改
`m0.push_back( s );`	在末尾增加一个m×1大小的数组
`m0.push_back( m1 );`	向m×n大小的矩阵m0增加k行并且复制到m1中，m1大小必须是k×n
`m0.pop_back( n );`	从m×n大小的矩阵移除n行（默认是1）[a]
`m0.locateROI( size, offset );`	将m0的全尺寸写入cv::Size 变量size 如果m0只是一个大矩阵的一块小区域，还会写入一个Point类型的offset
`m0.adjustROI( t, b, l, r );`	通过四个值t（最上），b（最下），l（最左），r（最右）调整ROI范围
`m0.total();`	计算数组序列的元素的数目（不包括通道）
`m0.isContinuous();`	如果m0的行之间没有空隙，将返回true
`m0.elemSize();`	返回m0的位长度（比如三通道浮点矩阵将返回12）
`m0.elemSize1();`	返回m0最基本元素的位长度（比如三通道浮点矩阵将返回4）

表4-9：cv::Mat类的更多函数成员（续）

示例	描述
m0.type();	返回m0元素的类型（比如CV_32FC3）
m0.depth();	返回m0通道中的元素类型（比如CV_32F）
m0.channels();	返回m0的通道数目
m0.size();	以cv::Size返回m0的大小
m0.empty();	如果数组没有元素，将返回true（比如m0.total==0或者m0.data==NULL）

a. 许多实现中，pop会返回被弹出的元素，但在这里不是，这里将返回void类型

稀疏数据类cv::SparesMat

cv::SparesMat类在数组非0元素非常少的情况下使用。这种情况在线性代数中称为“稀疏矩阵”，在表示数据的过程中同样也会出现，尤其是在直方图或者高维数组中，在一些特殊的应用中，大部分的元素都将为空。一个稀疏表示只存储有数据的部分，可以节约大量的内存 一般来说，许多稀疏对象在元素全部以稠密形式表示的时候会过于巨大。稀疏表示的坏处在于计算速度更慢（基于每个元素进行计算）。最后，很重要的一点，稀疏矩阵的计算并不是一直都慢，如果已经知道哪些操作不需要做，会节约很多计算时间。

OpenCV的稀疏矩阵类cv::SparseMat的函数在很多方面都类似于稠密矩阵类cv::Mat，它们的定义十分相似，支持大多数相同的操作，并且可以包含相同的数据类型。但从内部来说，它们数据的组织方式有着非常大的不同；当cv::Mat使用接近C风格的（数据按照序列被打包并且地址可以通过元素的索引计算出来）数组的时候，cv::SpareMat使用哈希表来存储非0元素[注11]，这个哈希表会自动维护，所以当数组中的非0数据变得太多以致于无法高效进行查找的时候，表也会自动增长。

访问稀疏数组中的元素

稀疏数组和稠密数组最后一点重要的区别在于他们的元素的访问。稀疏数组提供四种访问机制：cv::SparseMat::ptr()，cv::SparseMat::ref()，cv::SparseMat::value()和cv::SparseMat::find()。

Cv::SparesMat::Ptr()方法有几种变体，最简单的如下所示：

```
uchar* cv::SparseMat::ptr( int i0, bool createMissing, size_t* hashval=0 );
```

注11：事实上，当元素在某些计算之后变为0之后，这些0元素也会被存储。如果希望清理掉这些元素，你必须自己进行处理。函数SparseMat::erase()将在稍后介绍。

这个特定的版本用于访问一维数组。第一个参数i0，是请求元素的索引。第二个参数createMissing，表示这个元素是否应该被创建（如果它事先不存在于数组中）。当cv::SpareMat::Ptr()被调用的时候，如果这个元素已经在数组中定义，它将直接返回指向这个元素的指针，如果createMissing参数为真，这个元素将会被创建，并且一个合理的非0指针将指向这个新的元素。为了理解最后一个参数hashval，有必要记得cv::SparseMat的数据存储格式为哈希表。查找在哈希表中的对象需要两步：第一，计算哈希key（在本例中，key会依据索引计算）；第二，在key所指向的列表中进行查找 一般来说，这个列表将会很短（理想状态下只有一个元素），所以基础运算的时间主要耗费在查找和计算哈希Key。如果key已经计算出来了（比如调用cv::SparseMat::hash()，这将在稍后介绍），时间就可以通过砍掉重复的计算节约下来。在cv::SparseMat::Ptr()的例子中，如果参数hashval是默认的NULL，哈希key将被计算，如果主动提供一个key，它将会被使用。

这里还有一些cv::SparseMat::ptr()的变体允许使用两个或者三个索引，也有第一个参数是一个整型数组指针的版本（比如const int* idx），它要求这个索引的元素数量和被访问的数组的维度一样多。

在以上所有情况下，cv::SparseMat::ptr()都会返回一个指向无符号字符型的指针（uchar*），一般需要再次转换为正确的类型。

访问器模板函数SparseMat::ref<>()用于返回一个指向数组中特定元素的引用。这个函数类似于SparseMat::ptr()，可以接受一个、两个或者三个索引，或者一个索引数组，并且同样支持一个可选的用于查找的哈希值。因为这是一个模板函数，所以必须指定对象的类别。在这里，如果你的数组是CV_32F类型的，就可以像这样调用SparseMat::ref<>()：

```
a_sparse_mat.ref<float>( i0, i1 ) += 1.0f;
```

模板方法cv::SparseMat::value<>()和SparseMat::ref<>()彼此独立，它将返回一个值而不是返回值的引用。因此，这个方法也叫“只读方法”[注12]。

最后一个访问器函数叫cv::SparseMat::find<>()，它类似于cv::SparseMat::ref<>()和cv::SparseMat::value<>()，不同的是会返回一个请求对象的指针。和cv::SparseMat::ptr()不一样，cv::SparseMat::find<>()的指针类型是由模板指定的，

注12：对于不知道“只读正确性”的人来说，这意味着该方法在其原型中声明，因而传递给SparseMat::value<>()的指针务必确保是const的。因此SparseMat::value<>()可以被只读（const）的对象所调用，SparseMat::ref<>()这些函数不可以。下一个函数SparseMat::find<>()同样是一个const函数。

所以不需要再次转换。为了让代码更加清晰，相比cv::SparseMat::ptr()，更推荐使用cv::SparseMat::find<>()。然而，由于cv::Sparse Mat::find<>()是一个只读方法，返回一个只读的（const）指针，所以在很多情况下这两种方法都不可以互换。

除了刚刚讲授的四个直接访问数据的方法之外，还可以通过迭代器访问稀疏数组的元素。与密集数组类型一样，迭代器通常是模板化的，模板化的迭代器是cv::SparseMatIterator_<>和cv::SparseMatConstIterator_<>（const形式的begin()和end()将返回const的迭代器）。也有非模板化的迭代器cv::SparseMatIterator和cv::SparseMatConstIt erator，它们将返回一个非模板的SparseMat::begin()和SparseMat::end()。

在示例4-3中，我们将打印一些稀疏矩阵的元素。

示例4-3：打印一个稀疏矩阵中的所有非0元素

```
// Create a 10x10 sparse matrix with a few nonzero elements
//
int size[] = {10,10};
cv::SparseMat sm( 2, size, CV_32F );

for( int i=0; i<10; i++ ) {           // Fill the array
  int idx[2];
  idx[0] = size[0] * rand();
  idx[1] = size[1] * rand();

  sm.ref<float>( idx ) += 1.0f;
}

// Print out the nonzero elements
//
cv::SparseMatConstIterator_<float> it     = sm.begin<float>();
cv::SparseMatConstIterator_<float> it_end = sm.end<float>();

for(; it != it_end; ++it) {
  const cv::SparseMat::Node* node = it.node();
  printf(" (%3d,%3d) %f\n", node->idx[0], node->idx[1], *it );
}
```

在这个例子中，我们还略过了定义在迭代器中的方法node()。node()返回一个指向被迭代器索引指向的稀疏矩阵的实际数据区域。它将返回一个类型为cv::SparseMat::Node的对象，其定义如下：

```
struct Node
{
  size_t hashval;
  size_t next;
  int idx[cv::MAX_DIM];
};
```

这个结构既包含指向对应元素的哈希值的索引（注意，元素索引是一个int数组），还包含元素对应的哈希值（hashval元素与SparseMat::ptr(), Sparse Mat::ref(), SparseMat::value()和SparseMat::find()中所使用的相同）。

稀疏数组中的特有函数

如前所述，稀疏矩阵支持许多和稠密矩阵相同的操作。此外，还有一些方法是稀疏矩阵所独有的，这些方法在表4-10中列出，前几节中提到的函数也包含在表4-10内。

表4-10：cv::SparseMat中额外的类成员函数

示例	描述
`cv::SparseMat sm;`	跳过初始化，创建一个稀疏矩阵
`cv::SparseMat sm( 3, sz, CV_32F );`	创建一个由sz指定维度大小的三通道稀疏浮点矩阵
`cv::SparseMat sm( sm0 );`	从稀疏矩阵sm0复制一个稀疏矩阵
`cv::SparseMat( m0, try1d );`	从已有的稠密矩阵m0创建一个稀疏矩阵。如果布尔变量try1d为真，它将转换m0到一维稀疏矩阵（如果稠密矩阵是n×1或者1×n大小的）
`cv::SparseMat( &old_sparse_mat );`	从一个2.1版本之前的C风格稀疏矩阵CvSparseMat创建一个新的稀疏矩阵
`CvSparseMat* old_sm = (cv::SparseMat*) sm;`	转换操作将创建一个2.1版本之前的C风格稀疏矩阵CvSparseMat对象并且所有数据都会被复制到新对象中，最后返回对象的指针
`size_t n = sm.nzcount();`	返回sm中的非0元素数量

`size_t h = sm.hash( i0 );`
`size_t h = sm.hash( i0, i1 );`
`size_t h = sm.hash( i0, i1, i2 );`
`size_t h = sm.hash( idx );`
	返回一维稀疏矩阵中索引i0的所指向数据哈希值；返回二维稀疏举证中索引i0,i1所指向数据的哈希值；返回三维稀疏矩阵中索引i0,i1,i2所指向数据的哈希值；返回多维稀疏矩阵中索引idx数组所指向数据的哈希值

`sm.ref<float>( i0 ) = f0;`
`sm.ref<float>( i0, i1 ) = f0;`
`sm.ref<float>( i0, i1, i2 ) = f0;`
`sm.ref<float>( idx ) = f0;`
	设置一维稀疏矩阵中索引i0所指向元素的值为f0；设置二维稀疏矩阵中索引i0,i1所指向元素的值为f0；设置三维稀疏矩阵中索引i0,i1,i2所指向元素的值为f0；设置多维稀疏矩阵中索引为idx数组所指向元素的值为f0

`f0 = sm.value<float>( i0 );`
`f0 = sm.value<float>( i0, i1 );`
`f0 = sm.value<float>( i0, i1, i2 );`
`f0 = sm.value<float>( idx );`
	设置一维稀疏矩阵中索引i0所指向元素的值为f0；设置二维稀疏矩阵中索引i0,i1所指向元素的值为f0；设置三维稀疏矩阵中索引i0,i1,i2所指向元素的值为f0；设置多维稀疏矩阵中索引为idx数组所指向元素的值为f0

`p0 = sm.find<float>( i0 );`
`p0 = sm.find<float>( i0, i1 );`
`p0 = sm.find<float>( i0, i1, i2 );`
`p0 = sm.find<float>( idx );`
	将一维稀疏矩阵中索引i0所指向的元素赋值给p0；将二维稀疏矩阵中索引i0,i1所指向的元素赋值给p0；将三维稀疏矩阵中索引i0,i1,i2所指向的元素赋值给p0；将多维稀疏矩阵中索引为数组idx所指向的元素赋值给p0

表4-10：cv::SparseMat中额外的类成员函数（续）

示例	描述

`sm.erase( i0, i1, &hashval );`
`sm.erase( i0, i1, i2, &hashval );`
`sm.erase( idx, &hashval );`
	移除二维稀疏矩阵中索引为(i0,i1)的元素；移除三维稀疏矩阵中索引为(i0,i1,i2)的元素；移除多维稀疏矩阵中索引为数组idx的元素

`cv::SparseMatIterator<float> it = sm.begin<float>();`	创建一个浮点型稀疏矩阵迭代器it并指向其第一个元素
`cv::SparseMatIterator<uchar> it_end = sm.end<uchar>();`	创建一个无符号字符型稀疏矩阵迭代器it_end并将其初始化指向数组sm的最后一个元素的后一个元素

为大型数组准备的模板结构

我们在之前的章节中所看到的概念，其公共库类和模板类相关，可以一般化为cv::Mat和cv::SparseMat，以及模板cv::Mat_<>和cv::SparseMat_<>，而不是变成一些不常见的操作来实现。当你使用cv::Point2i时，会感觉这和cv::Point_<int>没什么两样 在使用cv::Mat和cv::Mat_<>的时候，它们的关系就没有那么简单了。cv::Mat已经拥有了表示基本数据类型的能力，但是它在创建的时候还是会指明其基于的类型。在这里，cv::Mat_<>这个实例模板其实是从cv::Mat集成来的一个类，并且是这个类的特化，这样就简化了接口和其他需要模板化的成员函数。

值得重申的是，使用模板cv::Mat_<>以及cv::SparseMat_<>的目的是不必在使用其成员函数的时候调用其模板形式。为了理解这个例子，我们定义一个矩阵：

```
cv::Mat m( 10, 10, CV_32FC2 );
```

对于这个矩阵，独立元素的访问需要指明矩阵的类型，需要我们进行这样的操作：

```
m.at< Vec2f >( i0, i1 ) = cv::Vec2f( x, y );
```

相对而言，如果已经使用模板类定义了一个矩阵m，就可以不进特化就使用at()函数，或者简单使用操作符()：

```
cv::Mat_<Vec2f> m( 10, 10 );
m.at( i0, i1 ) = cv::Vec2f( x, y );

// or...
m( i0, i1 ) = cv::Vec2f( x, y );
```

相比使用特化模板，这将会极大简化你的代码。



这两种声明矩阵的方法，以及各自的.at的方法在效率上是一样的。第二种方法普遍认为更“正确”，因为在m被输入函数并且需要一个具体的矩阵类型时允许编译器检查类型匹配的错误：

```
cv::Mat m(10, 10, CV_32FC2 );
```

将它传入函数：

```
void foo((cv::Mat_<char> *)myMat);
```

在运行时如果有未定义的行为，会出现错误，如果使用：

```
cv::Mat_<Vec2f> m( 10, 10 );
```

在编译的时候，就会提示错误。

模板可以用于创建操作一个特定类型数组的模板函数。想想我们在之前章节的例子，我们创建了一个小的稀疏矩阵并且打印其所有非0数值。我们可以尝试写一个函数来完成这个功能：

```
void print_matrix( const cv::SparseMat* sm ) {

  cv::SparseMatConstIterator_<float> it     = sm.begin<float>();
  cv::SparseMatConstIterator_<float> it_end = sm.end<float>();

  for(; it != it_end; ++it) {
    const cv::SparseMat::Node* node = it.node();
    printf(" (%3d,%3d) %f\n", node->idx[0], node->idx[1], *it );
  }
}
```

虽然这个函数很方便，并且如果它的输入是一个类型为CV_32F的二维数组，就可以正确执行。但是如果传入错误的矩阵类型，会出现错误。让我们来看看如何让这个函数变得更加通用。

我们首先要明白的是可能的数据类型。我们可以直接使用cv::SparseMat_<float>，但保持使用函数模板是一个更好的方法。我们还需要掌控printf()函数的使用，并且指明*it是浮点型。一个更好的函数应该更接近于示例4-4。

示例4-4：一个打印矩阵更好的方式

```
template <class T> void print_matrix( const cv::SparseMat_<T>* sm ) {

  cv::SparseMatConstIterator_<T> it     = sm->begin();
  cv::SparseMatConstIterator_<T> it_end = sm->end();

  for(; it != it_end; ++it) {
    const typename cv::SparseMat_<T>::Node* node = it.node();
    cout <<"( " <<node->idx[0] <<", " <<node->idx[1]
      <<" ) = " <<*it <<endl;
```

```
  }
}

void calling_function1( void ) {
  ...
  cv::SparseMat_<float> sm( ndim, size );
  ...
  print_matrix<float>( &sm );
}

void calling_function2( void ) {
  ...
  cv::SparseMat sm( ndim, size, CV_32F );
  ...
  print_matrix<float>( (cv::SparseMat_<float>*) &sm );
}
```

现在我们有必要梳理一下代码所做的改变 首先，在对比改变之前，注意函数的模板输入的指针，类型为const cv::Sparse Mat_<t>*且指向稀疏数组。一个很好的使用指针而不是使用引用的理由是，调用者可能是cv::Mat对象（正如在calling_function2()中的使用那样）而不是一个cv::Mat_<>模板对象（正如在calling_function1()中的使用那样）。Cv::Mat类型可以被取消引用，然后显式转换为一个指向稀疏矩阵模板对象类型的指针。

通过模板化的原型，我们已经将函数改造成类T的模板，并且输入变成一个cv::SparseMat_<T>*指针。在接下来的两行，我们声明使用模板类型的迭代器，但是begin()和end()不再具有实例化的模板，原因是sm现在已经是一个实例化的模板，并且由于直接指定，sm“知道”自己是什么样子的矩阵，因此特化begin()和end()就没那么有必要了。Node的声明也进行了类似的更改，这样我们所使用的Node将直接从cv::SparseMat_<T>的实例中获取[注13]。最后，我们将printf()表达式改为流式输出cout，这样做的优点是不需要知道*it的类型。

小结

在本章中，我们介绍了OpenCV最重要的数组结构cv::Mat，它用来表示矩阵、图像和多维数组。我们看到cv::Mat类可以包含任何原始类型的组合，例如数字、向量和其他类型。作为图像的情况下，它们仅仅如Vec3b一样在cv::Mat类对象构建的包含固定长度的向量。这个类有各种各样的成员函数，用于简化许多简单操作的表达。数组中的其他常见操作，也有各种各样的函数可以使用，这些我们在本章也讨论过。在学习稀疏矩阵的

注13：typename关键字的出现也许会让大多数读者感到很困扰。这是C++中依赖循环机制的结果。如果不慎忘记也不要紧，大多数编译器都会给出一个友好的提示。

过程中，我们发现它和cv::Mat一样几乎可以在任何地方正常使用，就像我们可以对大多数函数使用STL向量对象一样。最后，我们花了一点时间深入研究大型数组类型的模板类的确切功能，我们了解到，虽然原始类型是由它们的模板派生出来的，但大型数组模板却派生自基类。

练习

1. 建立一个500 × 500大小的单通道图像，每个像素的值都为0。
   a. 创建一个ASCII数字打字机，你可以在自己的电脑上输入数字，并在一个20像素高、10像素宽的方块中显示数字。当你键入时，数字将从左到右显示，直到到达图像的末尾才停止。
   b. 允许键入回车和退格。
   c. 允许使用箭头来编辑每个数字。
   d. 创建一个键来将输出图像转换为彩色图像，每个不同的数字由不同颜色显示。
2. 我们想要创建一个函数，通过创建一个统计图像来让矩形区域内的像素求和操作更有效率。这个统计图像的每个像素将储存从原点到它所包含矩形区域的所有像素值的和。这个图像也被称为“积分图”，通过使用积分图中四个点的值，我们可以得到它包含的矩形区域像素值的和。
   a. 创建一个100 × 200单通道uchar图像，并使用随机数填充，创建一个100 × 200单通道浮点数并将所有成员置0。
   b. 按照定义计算积分图。
   c. 考虑如何有效利用已经计算得到的积分图加上新的数值来加速积分图计算，实现这个更高效的方法。
   d. 使用积分图像快速计算原始图像中任意矩形的像素之和。
   e. 考虑如何修改积分图像，以便可以有效计算出一个45度旋转矩形的和。详细描述算法。

第5章

矩阵操作

矩阵还可以做更多事情

由前面的章节可知，矩阵类的成员函数可以进行很多基本的矩阵操作。然而，除此以外，也有很多操作被表示为“友元”函数，它们的输入为矩阵类型，或者输出为矩阵类型，或者输入输出同为矩阵类型。这些函数及其参数将在表5-1以后详细介绍。

表5-1：基本的矩阵和图像算子

函数名称	描述
cv::abs()	计算矩阵中所有元素的绝对值
cv::absdiff()	计算两个矩阵差值的绝对值
cv::add()	实现两个矩阵逐元素相加
cv::addWeighted()	实现两个矩阵逐元素加权求和（以为混合值）
cv::bitwise_and()	计算两个矩阵逐元素按位与
cv::bitwise_not()	按位非
cv::bitwise_or()	计算两个矩阵逐元素按位或
cv::bitwise_xor()	计算两个矩阵逐元素按位异或
cv::calcCovarMatrix()	计算一组*n*维向量的协方差
cv::cartToPolar()	计算二维向量的角度和幅度
cv::checkRange()	检查矩阵的无效值
cv::compare()	对两个矩阵中的所有元素应用所选择的比较运算符
cv::completeSymm()	通过将一半元素复制到另一半来使矩阵对称
cv::convertScaleAbs()	缩放矩阵，取绝对值，然后转换为8位无符号数

表5-1：基本的矩阵和图像算子（续）

函数	描述
cv::countNonZero()	计算矩阵中非零元素个数
cv::arrToMat()	将2.1版本的矩阵类型转换为cv::Mat
cv::dct()	计算矩阵的离散余弦变换
cv::determinant()	计算方阵的行列式
cv::dft()	计算矩阵的离散傅里叶变换
cv::divide()	实现两个矩阵逐元素相除
cv::eigen()	计算方阵的特征值和特征向量
cv::exp()	实现矩阵的逐元素求指数幂
cv::extractImageCOI()	从2.1版本的矩阵类型中提取单个通道
cv::flip()	绕选定的轴翻转矩阵
cv::gemm()	实现广义矩阵乘法
cv::getConvertElem()	获取单像素类型转换函数
cv::getConvertScaleElem()	获取单像素类型的转换和缩放函数
cv::idct()	计算矩阵的离散余弦逆变换
cv::idft()	计算矩阵的离散傅里叶逆变换
cv::inRange()	测试矩阵的元素是否在两个其他矩阵的值之间
cv::invert()	求方阵的逆
cv::log()	计算矩阵逐元素的自然对数
cv::magnitude()	计算二维向量的幅度
cv::LUT()	将矩阵转换为查找表的索引
cv::Mahalanobis()	计算两个向量之间的马氏距离
cv::max()	计算两个矩阵逐元素的最大值
cv::mean()	计算矩阵元素的平均值
cv::meanStdDev()	计算矩阵元素的均值和标准差
cv::merge()	将多个单通道矩阵合并成一个多通道矩阵
cv::min()	计算两个矩阵逐元素的最小值
cv::minMaxLoc()	在矩阵中寻找最小值和最大值
cv::mixChannels()	打乱从输入矩阵到输出矩阵的通道
cv::mulSpectrums()	计算两个傅里叶谱的逐元素乘积
cv::multiply()	计算两个矩阵的逐元素乘积
cv::mulTransposed()	计算矩阵和矩阵的转置的乘积
cv::norm()	计算两个矩阵的归一化相关系数
cv::normalize()	将矩阵中的元素标准化到某一数值内

表5-1：基本的矩阵和图像算子（续）

函数	描述
cv::perspectiveTransform()	实现一系列向量的透视矩阵变换
cv::phase()	计算二维向量的方向
cv::polarToCart()	已知角度和幅度，求出对应的二维向量
cv::pow()	对矩阵内的每个元素求幂
cv::randu()	用均匀分布的随机数填充给定的矩阵
cv::randn()	用正态分布的随机数填充给定的矩阵
cv::randShuffle()	随机打乱矩阵元素
cv::reduce()	通过特定的操作将二维矩阵缩减为向量
cv::repeat()	将一个矩阵的内容复制到另一个矩阵
cv::saturate_cast<>()	转换原始类型（模板函数）
cv::scaleAdd()	逐元素计算两个矩阵的和并且第一个矩阵可以选择缩放
cv::setIdentity()	将矩阵中对角线上的元素设为1，其他置0
cv::solve()	求出线性方程组的解
cv::solveCubic()	找到（唯一的）三次方程的实根
cv::solvePoly()	找到多项式方程的复根
cv::sort()	在矩阵中排序任意行或列的元素
cv::sortIdx()	与cv::sort()的目的相同，除了矩阵是未修改的，并返回索引
cv::split()	将一个多通道矩阵分割成多个单通道矩阵
cv::sqrt()	计算矩阵逐元素的平方根
cv::subtract()	实现两个矩阵逐元素相减
cv::sum()	对矩阵中的所有元素求和
cv::theRNG()	返回随机数生成器
cv::trace()	计算一个矩阵的迹
cv::transform()	在矩阵的每个元素上应用矩阵变换
cv::transpose()	矩阵的转置运算

表5-1中的这些函数都遵循一些通用的规则，特殊情况会在函数描述中注明。因为这些规则中的一个或多个几乎适用于本章所述的所有函数，为了方便，我们将它们列在下文。

饱和

计算的输出值被整合到输出矩阵的类型范围内。[译注1]

译注1：saturetion_cast是OpenCV提供的一种转换方法，它可以将高于或者低于目标类型取值区间的数值限制为取值类型的最大值和最小值。对于uchar来说，任何高于255的值都会被设置为255，而任何小于0的值都会被转化为0，而0~255之间的数值会正常取整。

输出

如果输出矩阵的类型和尺寸不符合所要求的类型和尺寸，将使用cv::Mat::create()重新创建输出矩阵。所要求的输出类型和尺寸与输入矩阵的往往都相同，但是对于一些函数来说，尺寸可能不同（比如cv::transpose）或者类型可能不同（比如cv::split）。

标量

许多函数，比如cv::add()，都允许两个矩阵相加或者一个矩阵和一个标量相加。原型可以很清楚地解释这个参数，提供一个标量参数的计算结果等同于提供一个所有元素都是这个标量的矩阵。

掩码

在一个函数中存在掩码参数时，则只会计算对应于掩码值不为0的位置的元素计算的矩阵。

dtype

许多算法和同类的函数不需要输入矩阵的类型相同，即使它们相同，输出矩阵也可能有与输入不同的类型。在这些情况下，输出矩阵必须明确指定其深度。这是通过dtype参数完成的。当dtype参数存在时，可以将dtype设置为任何基本类型（例如，CV_32F），结果矩阵将为该类型。如果dtype设置的类型与输入矩阵的类型相同，那么dtype可以设置为默认值-1，结果的类型将与输入矩阵的类型相同。

原位操作

除非另有说明，否则如果有一个矩阵作为输入，一个矩阵作为输出，且其尺寸和类型都相同的任何操作，则可以对输入输出使用相同的矩阵（即，可以将输出写入输入矩阵中）。

多通道

对于不需要使用多通道的操作，如果给定多通道参数，则每个通道都被单独处理。

cv::abs()

```
cv::MatExpr cv::abs( cv::InputArray src );
cv::MatExpr cv::abs( const cv::MatExpr& src );          // Matrix expression
```

这些函数计算矩阵或矩阵的某些表达式的绝对值。最常见的用法是计算矩阵中每个元素的绝对值。因为cv::abs()可以采用矩阵表达式，它能够识别某些特殊情况并适当地处理它们。实际上，对cv::abs()的调用会转换为对cv::absDiff()或其他函数的调用，并由这些函数处理。特别是下列特殊情况：

- m2 = cv::abs(m0 - m1)被转换为cv::absdiff(m0, m1, m2)

- `m2 = cv::abs( m0 )`被转换为`m2 = cv::absdiff( m0,cv::Scalar::all(0), m2 )`
- `m2 = cv::Mat_<Vec<uchar,n> >( cv::abs( alpha*m0 + beta ) )`（alpha与beta为实数）被转换为`cv::convertScaleAbs( m0, m2, alpha,beta )`

第三种情况可能看起来难以理解，但这只是计算*n*通道矩阵的尺度变换和偏移（其中任何一种都可以省去）的例子。例如，在计算图像的对比度校正时，可能会采用这种方式。

在由`cv::absdiff()`实现的情况下，结果矩阵将有与输入矩阵相同的尺寸和类型。然而，在`cv::convertScaleAbs()`实现的情况下，返回矩阵的结果类型将始终为`CV_U8`。

`cv::absdiff()`

```
void cv::absdiff(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst                      // Result array
)
```

$$dst_i = saturate(|\ src1_i - src2_i\ |)$$

给定两个矩阵，`cv::absdiff()`计算这两个矩阵中每对相应元素之间的差值，并将该差值的绝对值放入目标矩阵的相应元素中。

cv::add()

```
void cv::add(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst,                     // Result array
  cv::InputArray  mask  = cv::noArray(),   // Optional, do only where nonzero
  int             dtype = -1               // Output type for result array
);
```

$$dst_i = saturate(|\ src1_i + src2_i\ |)$$

`cv::add()`是一个简单的加法函数：它将`src1`中的所有元素添加到`src2`中的相应元素中，并将结果放入`dst`中。



对于简单的情况，矩阵运算可以得到相同的结果：

```
dst = src1 + src2;
```

也支持累加的情况：

```
dst += src1;
```

cv::addWeighted()

```
void cv::addWeighted(
  cv::InputArray  src1,          // First input array
  double          alpha,         // Weight for first input array
  cv::InputArray  src2,          // Second input array
  double          beta,          // Weight for second input array
  double          gamma,         // Offset added to weighted sum
  cv::OutputArray dst,           // Result array
  int             dtype = -1     // Output type for result array
);
```

函数cv::addWeighted()与cvAdd()类似，除了根据以下公式计算写入dst的结果：

$$dst_i = saturate(src1_i * \alpha + src2_i * \beta + \gamma)$$

只要两个源图像src1和src2类型相同，它们可以是任意像素类型，也可以有任意数量的通道（灰度与彩色等）。

此函数可以用于实现*alpha*混合[Smith79;Porter84]；也就是说，它可以将一个图像与另一个图像混合。在这种情况下，参数α是src1的混合强度，β是src2的混合强度。将α设置在0和1之间，将β设置为1－α，将γ设置为0，即可转换为标准的alpha混合方程。可以导出：

$$dst_i = saturate(src1_i * \alpha + src2_i * (1 - \alpha))$$

然而，cv::addWeighted()应用更加灵活——无论是我们如何对混合图像进行加权还是确定能够对所得到的目标图像进行附加偏移的额外参数γ。对于一般形式，你或许希望将alpha和beta的值设在0或0以上，并且其总和不超过1；可以根据像素所要调整到的平均或最大值来设置gamma。使用alpha混合的程序如示例5-1所示。

示例5-1：从src2中(0,0)开始的ROI与从src1中(x,y)开始的ROI进行alpha混合的完整程序

```
// alphablend <imageA> <image B> <x> <y> <width> <height> <alpha> <beta>
//
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv) {

  cv::Mat src1 = cv::imread(argv[1],1);
  cv::Mat src2 = cv::imread(argv[2],1);

  if( argc==9 && !src1.empty() && !src2.empty() ) {

    int    x     = atoi(argv[3]);
    int    y     = atoi(argv[4]);
    int    w     = atoi(argv[5]);
    int    h     = atoi(argv[6]);
```

```
  double alpha = (double)atof(argv[7]);
  double beta  = (double)atof(argv[8]);

  cv::Mat roi1( src1, cv::Rect(x,y,w,h) );
  cv::Mat roi2( src2, cv::Rect(0,0,w,h) );

  cv::addWeighted( roi1, alpha, roi2, beta, 0.0, roi1 );

  cv::namedWindow( "Alpha Blend", 1 );
  cv::imshow( "Alpha Blend", src2 );
  cv::waitKey( 0 );
}

  return 0;
}
```

示例5-1中的代码需要两个源图像：原始图像（src1）和混合图像（src2）。它读取src1的矩形ROI，并将相同尺寸的ROI应用于src2，但初始位置位于原点。读取alpha和beta的值，并将gamma设置为0。使用cv::addWeighted()实现阿尔法混合（Alpha Blend），并将结果放入src1并显示。示例输出如图5-1所示，小孩的脸部合成到猫的脸上。请注意，代码与示例5-1中的ROI相同。这次，我们使用ROI作为目标混合区域。

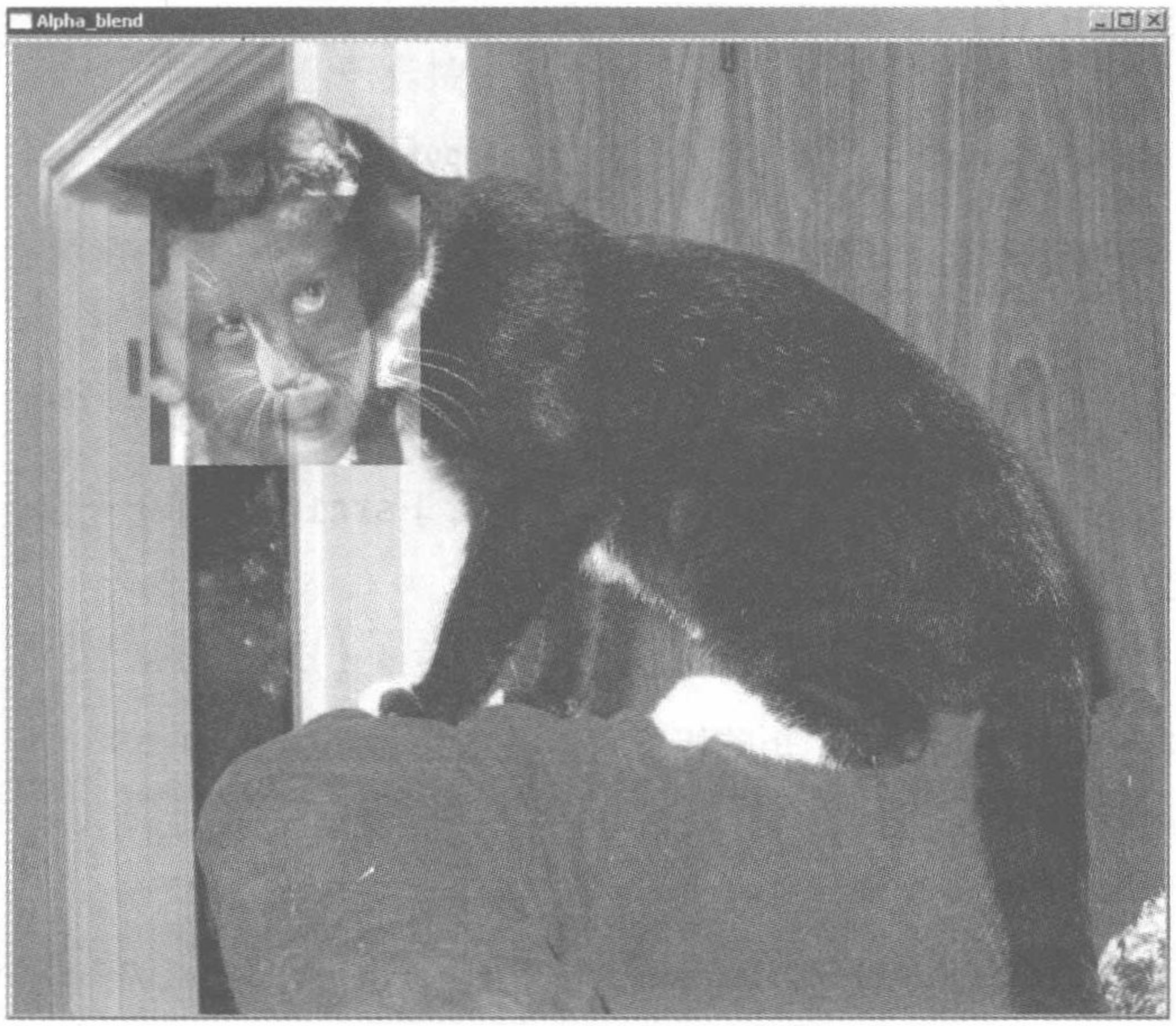


图5-1. 一个孩子的脸部阿尔法混合到一只猫的脸上

cv::bitwise_and()

```
void cv::bitwise_and(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst,                     // Result array
  cv::InputArray  mask  = cv::noArray(),   // Optional, do only where nonzero
);
```

$$dst_i = src1_i \wedge src2_i$$

cv::bitwise_and()是一个逐元素按位与操作。对于src1中的每个元素，使用src2中的相应元素计算按位与，并将其放入dst的相应元素中。



如果不使用掩码，使用矩阵操作可以获得相同的结果：

```
dst = src1 & src2;
```

cv::bitwise_not()

```
void cv::bitwise_not(
  cv::InputArray  src,                     // Input array
  cv::OutputArray dst,                     // Result array
  cv::InputArray  mask  = cv::noArray(),   // Optional, do only where nonzero
);
```

$$dst_i = \sim src1_i$$

cv::bitwise_not()是一个逐元素按位反转操作。对于src1中的每个元素，进行逻辑反转计算并将结果放到dst的相应元素中。



如果不使用掩码，使用矩阵操作可以获得相同的结果：

```
dst = !src1;
```

cv::bitwise_or()

```
void cv::bitwise_and(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst,                     // Result array
  cv::InputArray  mask  = cv::noArray(),   // Optional, do only where nonzero
);
```

$$dst_i = src1_i \vee src2_i$$

cv::bitwise_or()是一个逐元素按位或操作。对于src1中的每个元素，使用src2中的相应元素计算按位或，并将其放入dst的相应元素中。



如果不使用掩码，使用矩阵操作可以获得相同的结果：

```
dst = src1 | src2;
```

cv::bitwise_xor()

```
void cv::bitwise_and(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst,                     // Result array
  cv::InputArray  mask  = cv::noArray(),   // Optional, do only where nonzero
);
```

$$dst_i = src1_i \oplus src2_i$$

cv::bitwise_xor()是一个逐元素按位“异或”（XOR）操作。对于src1中的每个元素，使用src2中的相应元素计算按位异或，并将其放入dst的相应元素中。



如果不使用掩码，使用矩阵操作可以获得相同的结果：

```
dst = src1 ^ src2;
```

cv::calcCovarMatrix()

```
void cv::calcCovarMatrix(
  const cv::Mat* samples,              // C-array of n-by-1 or 1-by-n matrices
  int            nsamples,             // num matrices pointed to by 'samples'
  cv::Mat&       covar,                // ref to return array for covariance
  cv::Mat&       mean,                 // ref to return array for mean
  int            flags,                // special variations, see Table 5-2
  int            ctype = cv::F64       // output matrix type for covar
);

void cv::calcCovarMatrix(
  cv::InputArray samples,              // n-by-m matrix, but see 'flags' below
  cv::Mat&       covar,                // ref to return array for covariance
  cv::Mat&       mean,                 // ref to return array for mean
  int            flags,                // special variations, see Table 5-2
  int            ctype = cv::F64       // output matrix type for covar
);
```

给定一些向量，假设这些向量表示的点是近似高斯分布的，cv::calcCovarMatrix()将计算这些点的均值和协方差矩阵。这当然可以应用到很多方面，OpenCV有很多附加的flag值，在特定情况下会起作用。这些flag可以用标准的布尔或操作组合在一起。

表5-2：cv::calcCovarMatrix()可能用到的flag参数的值

flags参数的具体flag值	意义
cv::COVAR_NORMAL	计算均值和协方差
cv::COVAR_SCRAMBLED	快速PCA“scrambled”协方差
cv::COVAR_USE_AVERAGE	输入均值而不是计算均值
cv::COVAR_SCALE	重新缩放输出的协方差矩阵
cv::COVAR_ROWS	使用样本的行作为输入向量
cv::COVAR_COLS	使用样本的列作为输入向量

cv::calcCovarMatrix()有两个基本的调用方法，第一种是指向cv::Mat对象矩阵的指针和表示矩阵中的矩阵数量的nsamples一起传入函数。在这种情况下，矩阵可以是$n \times 1$或$1 \times n$。第二种是传入一个$n \times m$的矩阵。在这种情况下，应该提供flag为cv::COVAR_ROWS以指示存在长度为m的n（行）个向量，或者应该提供flag为cv::COVAR_COLS以指示存在长度为n的m（列）个向量。

在所有情况下，结果将放在covar中，但avg的确切含义取决于flag的值（见表5-2）。

标志cv::COVAR_NORMAL和cv::COVAR_SCRAMBLED是相互排斥的；只能使用其中一种，不能两者同时使用。如果使用cv::COVAR_NORMAL，函数只计算均值和协方差。

$$\Sigma^2_{normal} = z \begin{bmatrix} v_{0,0} - \overline{v}_o & \cdots & v_{m,0} - \overline{v}_o \\ \vdots & \ddots & \vdots \\ v_{0,n} - \overline{v}_n & \cdots & v_{m,n} - \overline{v}_n \end{bmatrix} \begin{bmatrix} v_{0,0} - \overline{v}_o & \cdots & v_{m,0} - \overline{v}_o \\ \vdots & \ddots & \vdots \\ v_{0,n} - \overline{v}_n & \cdots & v_{m,n} - \overline{v}_n \end{bmatrix}^T .$$

因此，标准的协方差由长度为n的m个向量计算得到，被定义为平均值向量的第n个元素，由此产生的协方差矩阵是一个$n \times n$的矩阵。因子z是一个可选比例因子，除非使用cv::COVAR_SCALE标志，否则它将被设置为1。

如果使用cv::COVAR_SCRAMBLED标志，cv::calcCovarMatrix()将如下计算：

$$\Sigma^2_{scrambled} = z \begin{bmatrix} v_{0,0} - \overline{v}_o & \cdots & v_{m,0} - \overline{v}_o \\ \vdots & \ddots & \vdots \\ v_{0,n} - \overline{v}_n & \cdots & v_{m,n} - \overline{v}_n \end{bmatrix}^T \begin{bmatrix} v_{0,0} - \overline{v}_o & \cdots & v_{m,0} - \overline{v}_o \\ \vdots & \ddots & \vdots \\ v_{0,n} - \overline{v}_n & \cdots & v_{m,n} - \overline{v}_n \end{bmatrix} .$$

这个矩阵不是一般的协方差矩阵（注意转置运算符的位置）。这个矩阵的计算同样来自长度为n的m个向量，但由此计算得到的协方差矩阵是$m \times m$的矩阵。这个矩阵是用在一些特定的算法中，比如对非常大的向量进行快速PCA（比如在人脸识别技术中）。

如果输入向量的平均值是已知的，则使用cv::COVAR_USE_AVG标志，在这种情况下，参数avg用来作为输入而不是作为输出，以减少计算时间。

最后，标志cv::COVAR_SCALE用于对计算得到的协方差矩阵进行统一缩放，这是上述方程式中的因子z。同标志cv::COVAR_NORMAL一起使用时，应用的缩放比例是1.0/m（或等效于1.0/nsamples）。如果使用cv::COVAR_SCRAMBLED，那么z的值将是1.0/n（向量长度的倒数）。

cv::calcCovarMatrix()的输入输出矩阵都应该是相同的浮点型。结果矩阵的尺寸应该是$n \times n$或者$m \times m$，取决于计算的是标准协方差还是scrambled协方差。应该注意的是，使用cv::Mat*形式时，samples的“向量”输入不一定要是一维的，它们也可以是二维对象（例如图像）。

cv::cartToPolar()

```
void cv::cartToPolar(
  cv::InputArray  x,
  cv::InputArray  y,
  cv::OutputArray magnitude,
  cv::OutputArray angle,
  bool            angleInDegrees = false
);
```

$$magnitude_i = \sqrt{x_i^2 + y_i^2}$$

$$angle_i = atan2(y_i, x_i)$$

函数cv::cartToPolar()使用两个输入矩阵x和y，它们分别表示一个向量场的x和y分量（注意，这不是一个双通道矩阵，而是两个单通道的矩阵）。矩阵x和y必须有相同的尺寸。然后cv::cartToPolar()计算每个向量的极坐标表示。每个向量的幅度被放置在相应的位置上，每个向量的方向以角度的形式放置在相应的位置上。除非布尔变量angleInDegrees设置为true，否则返回的角度以弧度表示。

cv::checkRange()

```
bool cv::checkRange(
  cv::InputArray src,
  bool           quiet  = true,
```

```
    cv::Point*     pos    = 0,            // if non-Null, location of first outlier
    double         minVal = -DBL_MAX,     // Lower check bound (inclusive)
    double         maxVal =  DBL_MAX      // Upper check bound (exclusive)
  );
```

函数cv::checkRange()检查输入矩阵src的每个元素，并确定该元素是否在给定范围内。范围由minVal和maxVal设置，但任何NaN或inf值也被认为超出范围。如果找到超出范围的值，除非将quiet设置为true，否则抛出异常。在这种情况下，如果所有值都在范围内，cv::checkRange()的返回值就为true；如果有任何值超出范围，cv::checkRange()的返回值则为false。如果指针pos不为NULL，则第一个异常值的位置将存储在pos中。

cv::compare()

```
bool cv::compare(
  cv::InputArray  src1,                 // First input array
  cv::InputArray  src2,                 // Second input array
  cv::OutputArray dst,                  // Result array
  int             cmpop                 // Comparison operator, see Table 5-3
);
```

该函数在两个矩阵src1和src2中的对应元素之间进行逐元素比较，并将结果放在图像dst中。cv::compare()将最后一个参数作为比较运算符，它可以是表5-3中列出的任何类型。在每种情况下，结果dst的每个像素都是一个8位序列，匹配的像素被标记为255，不匹配的像素被设置为0。

表5-3：cv::compare()使用的cmp_op值以及由此产生的比较操作

cmp_op的值	比较操作
cv::CMP_EQ	(src1i == src2i)
cv::CMP_GT	(src1i > src2i)
cv::CMP_GE	(src1i >= src2i)
cv::CMP_LT	(src1i < src2i)
cv::CMP_LE	(src1i <= src2i)
cv::CMP_NE	(src1i != src2i)

列出的所有比较操作都是通过相同的函数实现的，只需要传递合适的参数来说明你想怎么做。

这些比较函数很有用，例如，在去除背景中创建一个改变像素的掩码（例如，安保系统中的摄像机），从而只从图像中提取出有用的信息。



通过矩阵运算可以实现相同的结果：

```
dst = src1 == src2;
dst = src1  > src2;
dst = src1 >= src2;
dst = src1  < src2;
dst = src1 <= src2;
dst = src1 != src2;
```

cv::completeSymm()

```
bool cv::completeSymm(
  cv::InputArray mtx,
  bool           lowerToUpper = false
);
```

$$mtx_{ij} = mtx_{ji} \forall\ i > j \text{ (lowerToUpper = false)}$$

$$mtx_{ij} = mtx_{ji} \forall\ j < i \text{ (lowerToUpper = true)}$$

给定一个（二维）矩阵mtx，cv::completeSymm()通过复制来使矩阵对称[注1]。具体来说，来自上三角形的所有元素都被复制到它们在矩阵下三角形对应的转置位置。mtx的对角元素保持不变。如果标志lowerToUpper设置为true，则来自下三角形的元素将被复制到上三角形中。

cv::convertScaleAbs()

```
void cv::convertScaleAbs(
  cv::InputArray  src,                  // Input array
  cv::OutputArray dst,                  // Result array
  double          alpha = 1.0,          // Multiplicative scale factor
  double          beta  = 0.0           // Additive offset factor
);
```

$$dst_i = saturate_{uchar}(\,|\,\alpha * src_i + \beta\,|\,)$$

函数cv::convertScaleAbs()实际上融多种功能于一体。它将按顺序执行四个操作。第一个操作是通过因子alpha来重新调整源图像，第二个操作是通过（加）因子beta来偏移，第三个操作是计算上述所求和的绝对值，第四个操作是将该结果（饱和）映射到一个无符号字符型（8位）。

注1：数学好的读者会意识到，对于矩阵来说，有一些其他的对称过程比此操作更“自然”，但这种特定的操作有其自身的意义，例如，只计算一半就完成对称矩阵的运算，所以OpenCV中包含了此函数。

如果只是传递默认值（alpha = 1.0或beta = 0.0），则不必担心性能；OpenCV足够聪明，能意识到这种情况而不会在无用的操作上浪费处理器的时间。



通过以下循环可以获得类似的结果，有更强的通用性：

```
for( int i = 0; i < src.rows; i++ )
     for( int j = 0; j < src.cols*src.channels(); j++ )
          dst.at<dst_type>(i, j) = satuarate_cast<dst_type>(
               (double)src.at<src_type>(i, j) * alpha + beta
          );
```

cv::countNonZero()

```
int cv::countNonZero(                    // Return number of nonzero elements in mtx
  cv::InputArray mtx,                    // Input array
);
```

$$count = \sum_{mtx_i \neq 0} 1$$

cv::countNonZero()返回矩阵mtx中的非0像素数。

cv::Mat cv::cvarrToMat()

```
cv::Mat cv::cvarrToMat(
  const CvArr* src,                 // Input array: CvMat, IplImage, or CvMatND
  bool         copyData = false, // if false just make new header, else copy data
  bool         allowND  = true,  // if true and possible, convert CvMatND to Mat
  int          coiMode  = 0      // if 0: error if COI set, if 1: ignore COI
);
```

cv::cvarrToMat()可以将“旧式”（2.1版本以前）的图像或矩阵类型转换为“新式”（使用C++的2.1或更高版本）cv::Mat对象。默认情况下，仅构建新矩阵的头节点，而不复制数据。相反，新头节点中的数据指针指向现有数据矩阵（因此在使用cv::Mat头节点时不要释放它）。如果要复制数据，只需要将copyData设置为true，就可以任意删除原始数据对象。

cv::cvarrToMat()也可以采用CvMatND结构，但它不能处理所有的情况。转换的关键要求是矩阵应该是连续的或至少可以表示为连续矩阵的序列。具体来说，对于所有i，或者最坏除一个i以外，A.dim[i].size*A.dim.step[i]应该等于A.dim.step[i-1]。如果allowND设置为true（默认），cv::cvarrToMat()将在遇到CvMatND结构时尝试进行转换，如果无法进行转换（前提条件），则会引发异常。如果allowND设置为false，则在遇到CvMatND结构时将抛出异常。

因为COI[注2]概念在版本2.1以后的库中进行了不同处理（也就是说，它不再存在），所以在转换期间必须处理COI。如果参数coiMode为0，则当src包含活动COI时，将抛出异常。如果coiMode不为零，则不会报告错误，而是返回对应于整个图像的cv::Mat头，忽略COI。如果要正确处理COI，则必须检查图像是否具有COI集合，如果是，请使用cv::extractImageCOI()为该通道创建头。



大多数时候，这个函数用于帮助将旧式代码迁移到新的代码。在这种情况下，你可能需要将旧式CvArr*结构转换为cv::Mat，并进行相反的操作。相反的操作使用转换运算符完成。例如，如果你有一个定义为cv::Mat的矩阵A，则可以将其转换为IplImage*指针：

```
Cv::Mat A( 640, 480, cv::U8C3 );
// casting is implicit on assignment
IplImage. my_img = A;
iplImage* img = &my_img;
```

cv::dct()

```
void cv::dct(
  cv::InputArray  src,                  // Input array
  cv::OutputArray dst,                  // Result array
  int             flags,                // for inverse transform or row-by-row
);
```

该函数根据flags参数的值执行离散余弦变换或离散余弦逆变换。源矩阵src必须是一维或二维的，并且尺寸应该是偶数（如果需要，可以填充矩阵）。结果矩阵dst将具有与src相同的类型和尺寸。参数flags是一个位域值，可以设置为DCT_INVERSE或DCT_ROWS中的一个或两个。如果设置为DCT_INVERSE，则实现逆变换而不是前向变换。如果设置标志为DCT_ROWS，则将二维$n \times m$的输入视为长度为m的n个不同的一维向量。在这种情况下，每个这样的向量将被独立地变换。



cv::dct()的性能很大程度上取决于传递给它的矩阵的确切尺寸，这种关系（性能与尺寸的关系）并不是单调的。只有一些尺寸是比其他尺寸表现出的性能更好。建议在将矩阵传递给cv::dct()时，首先在比当前矩阵大的尺寸中确定最佳尺寸，然后将矩阵扩展为该尺寸。OpenCV为你提供了一个合适的例程来计算这个值，称为cv::getOptimalDFTSize()。

注2：“COI”是一个2.0版本之前的旧概念，意味着“感兴趣的通道”。在过去的IplImage类中，COI类似于ROI（感兴趣的区域），可以设置为使某些函数只在指定的通道上起作用。

正如具体实现过程所描述的，长度为n的向量的离散余弦变换是通过长度为$n/2$的向量的离散傅里叶变换（cv::dft()）计算得来的。这意味着要获得调用cv::dct()的最佳尺寸，你应该这样计算：

```
size_t opt_dft_size = 2 * cv::getOptimalDFTSize((N+1)/2);
```

这个函数（和一般的离散变换）会在第11章中详细介绍。在那一章中，我们将详细讨论如何压缩和解压输入和输出，以及什么时候和为什么要使用离散余弦变换。

cv::dft()

```
void cv::dft(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             flags        = 0,       // for inverse transform or row-by-row
  int             nonzeroRows = 0         // only this many entries are nonzero
);
```

cv::dft()函数实现离散傅里叶变换以及其逆变换（取决于flags参数）。源矩阵src必须是一维或二维的。结果矩阵dst将具有与src相同的类型和尺寸。参数flags是一个位域值，可以设置为DFT_INVERSE，DFT_ROWS，DFT_SCALE，DFT_COMPLEX_OUTPUT或DFT_REAL_OUTPUT中的一个或多个。如果设置为DFT_INVERSE，则完成逆变换。如果设置标志为DFT_ROWS，则二维$n \times m$输入被视为长度为m的n个不同的一维向量，并且每个这样的向量将独立变换。标志DFT_SCALE通过将结果除以矩阵中的元素数来标准化结果，这通常用于DFT_INVERSE，因为它保证逆的逆将具有正确的标准化。

标志DFT_COMPLEX_OUTPUT和DFT_REAL_OUTPUT是有用的，因为当计算实数矩阵的傅里叶变换时，结果将有复共轭对称性。因此，即使结果是复数，结果矩阵的元素数量等于输入矩阵中的元素数量，而不是该数量的两倍。这样的压缩是cv::dft()的默认行为。若要强制输出复数的形式，则需设置标志DFT_COMPLEX_OUTPUT。在逆变换的情况下，输入（通常）为复数，输出也为复数。然而，如果输入矩阵（对逆变换的情况）具有复共轭对称性（例如，如果它本身是实数矩阵的傅里叶变换的结果），那么逆变换将是一个实数矩阵。如果知道是这种情况，并且希望结果矩阵表示为一个实数矩阵（从而使用一半的内存量），则可以设置DFT_REAL_OUTPUT标志。请注意，如果设置此标志，cv::dft()不会检查输入矩阵是否具有必要的对称性，它只是假定具有对称性。

cv::dft()的最后一个参数是nonzeroRows，它默认为0，但如果设置它为任何非0值，将导致cv::dft()认为只有输入矩阵的前nonzeroRows行是有意义的。如果DFT_INVERSE被设置，那么就认为只有输出矩阵的前nonzeroRows行是非零的。在使用cv::dft()计算卷积的互相关时，这个标志特别方便。



cv::dft()的性能很大程度上取决于传递给它的矩阵的确切尺寸，这种关系（性能与尺寸的关系）并不是线性的。只有一些尺寸是比其他尺寸表现出的性能更好。建议在将矩阵传递给cv::dft()时，首先在比当前矩阵大的尺寸中确定最佳尺寸，然后将矩阵扩展为该尺寸。OpenCV提供了一个合适的例程来计算这个值，称为cv::getOptimalDFTSize()。

同样，这个函数（和一般的离散变换）会在第11章中详细介绍。在那一章中，我们将详细讨论如何压缩和解压输入和输出，以及什么时候和为什么要使用离散傅里叶变换。

cv::cvtColor()

```
void cv::cvtColor(
  cv::InputArray  src,                  // Input array
  cv::OutputArray dst,                  // Result array
  int             code,                 // color mapping code, Table 5-4.
  int             dstCn = 0             // channels in output (0='automatic')
);
```

cv::cvtColor[注3]是在保留相同数据类型的同时从一个颜色空间（通道数）转换到另一个颜色空间[Wharton71]。输入矩阵src可以是8位矩阵，16位无符号矩阵或32位浮点矩阵。输出矩阵dst将具有与输入矩阵相同的尺寸和深度。要完成的转换操作由编码参数指定，可能的值如表5-4()所示。最终参数dstCn是目标图像中所需的通道数。如果给出默认值0，则通道数由src中的通道数和转换编码确定。

表5-4：可通过cv::cvtColor()实现的转换

转换编码	意义
cv::COLOR_BGR2RGB cv::COLOR_RGB2BGR cv::COLOR_RGBA2BGRA cv::COLOR_BGRA2RGBA	在RGB或BGR颜色空间之间转换（包括或者不包括alpha通道）
cv::COLOR_RGB2RGBA cv::COLOR_BGR2BGRA	在RGB或BGR图像中加入alpha通道
cv::COLOR_RGBA2RGB cv::COLOR_BGRA2BGR	从RGB或BGR图像中删除alpha通道
cv::COLOR_RGB2BGRA cv::COLOR_RGBA2BGR cv::COLOR_BGRA2RGB cv::COLOR_BGR2RGBA	在加入或删除Alpha通道时将RGB转换为BGR颜色空间

注3：用过IPL的用户会注意到一点：cvCvtColor()会忽略IplImage头中的colorModel和channelSeq属性。完全根据所提供的参数code进行这样的转换。

表5-4：可通过cv::cvtColor()实现的转换（续）

转换编码	意义
cv::COLOR_RGB2GRAY cv::COLOR_BGR2GRAY	转换RGB或BGR颜色空间为灰度空间
cv::COLOR_GRAY2RGB cv::COLOR_GRAY2BGR cv::COLOR_RGBA2GRAY cv::COLOR_BGRA2GRAY	转换灰度空间为RGB或BGR颜色空间（在转换过程中可选择删除alpha通道）
cv::COLOR_GRAY2RGBA cv::COLOR_GRAY2BGRA	将灰度空间转换为RGB或BGR颜色空间，并加入alpha通道
cv::COLOR_RGB2BGR565 cv::COLOR_BGR2BGR565 cv::COLOR_BGR5652RGB cv::COLOR_BGR5652BGR cv::COLOR_RGBA2BGR565 cv::COLOR_BGRA2BGR565 cv::COLOR_BGR5652RGBA cv::COLOR_BGR5652BGRA	将RGB或BGR颜色空间转换为BGR565颜色表示，可选择加入或删除alpha通道（16位图像）
cv::COLOR_GRAY2BGR565 cv::COLOR_BGR5652GRAY	将灰度空间转换为BGR565颜色表示或者反变换（16位图像）
cv::COLOR_RGB2BGR555 cv::COLOR_BGR2BGR555 cv::COLOR_BGR5552RGB cv::COLOR_BGR5552BGR cv::COLOR_RGBA2BGR555 cv::COLOR_BGRA2BGR555 cv::COLOR_BGR5552RGBA cv::COLOR_BGR5552BGRA	从RGB或BGR颜色空间转换为BGR555颜色表示，可选择加入或删除alpha通道（16位图像）
cv::COLOR_GRAY2BGR555 cv::COLOR_BGR5552GRAY	将灰度空间转换为BGR555颜色表示或者反变换（16位图像）
cv::COLOR_RGB2XYZ cv::COLOR_BGR2XYZ cv::COLOR_XYZ2RGB cv::COLOR_XYZ2BGR	将RGB或BGR图像转为CIE XYZ表示或者反转（Rec 709和D65白点）
cv::COLOR_RGB2YCrCb cv::COLOR_BGR2YCrCb cv::COLOR_YCrCb2RGB cv::COLOR_YCrCb2BGR	转换RGB或BGR图像到luma-chroma（a.k.a. YCC）颜色表示或反转

表5-4：可通过cv::cvtColor()实现的转换（续）

转换编码	意义
cv::COLOR_RGB2HSV cv::COLOR_BGR2HSV cv::COLOR_HSV2RGB cv::COLOR_HSV2BGR	将RGB或BGR图像转为HSV（hue saturation value）颜色表示或反转
cv::COLOR_RGB2HLS cv::COLOR_BGR2HLS cv::COLOR_HLS2RGB cv::COLOR_HLS2BGR	将RGB或BGR图像转为HLS (hue lightness saturation)颜色表示或反转
cv::COLOR_RGB2Lab cv::COLOR_BGR2Lab cv::COLOR_Lab2RGB cv::COLOR_Lab2BGR	将RGB或BGR图像转换为CIE Lab颜色表示或者反变换
cv::COLOR_RGB2Luv cv::COLOR_BGR2Luv cv::COLOR_Luv2RGB cv::COLOR_Luv2BGR	将RGB或BGR图像转换为CIE Luv颜色表示或者反变换
cv::COLOR_BayerBG2RGB cv::COLOR_BayerGB2RGB cv::COLOR_BayerRG2RGB cv::COLOR_BayerGR2RGB cv::COLOR_BayerBG2BGR cv::COLOR_BayerGB2BGR cv::COLOR_BayerRG2BGR cv::COLOR_BayerGR2BGR	从Bayer模式（单通道）转为RGB或BGR图像

我们不讲述这些转换的细节，也不在这里介绍一些颜色表示（特别是Bayer颜色空间和CIE颜色空间）的细微之处。我们只注意OpenCV包含转换为各种颜色空间和从各种颜色空间转换的工具，这对各类用户来说很重要。

颜色空间的转换都用到以下约定：8位图像范围是0~255，16位图像范围是0~65536，浮点数的范围是0.0~1.0。当灰度图像转换为彩色图像时，最终图像的所有通道都是相同的；但是逆变换（例如RGB或BGR到灰度），使用加权公式来计算灰度值。

$$Y = (0.299)R + (0.587)G + (0.114)B$$

在HSV或HLS表示的情况下，色调通常为0~360。[注4]在8位图像中，这可能出现问题，因此，转换到HSV颜色空间，并以8位图的形式输出时，色调应该除以2。

注4：　不包括360。

cv::determinant()

```
double cv::determinant(
  cv::InputArray mat
);
```

$d = \det(mat)$

cv::determinant()用于计算一个方阵的行列式。这个矩阵必须是浮点数据类型之一，而且必须是单通道的。如果是小的矩阵，则直接用标准公式计算。然而对于大型矩阵，这样就不是很有效，此时行列式的计算使用高斯清除法。



如果已知一个矩阵是对称正定的，也可以通过奇异值分解（SVD）的策略来解决。欲了解更多信息，请参阅第7章对奇异值分解（cv::SVD）的讨论一节。但这个策略是将U和V设置为NULL，然后计算矩阵W的乘积来获得所求的行列式。

cv::divide()

```
void cv::divide(
  cv::InputArray  src1,                  // First input array (numerators)
  cv::InputArray  src2,                  // Second input array (denominators)
  cv::OutputArray dst,                   // Results array (scale*src1/src2)
  double          scale = 1.0,           // Multiplicative scale factor
  int             dtype = -1             // dst data type, -1 to get from src2
);

void cv::divide(
  double          scale,                 // Numerator for all divisions
  cv::InputArray  src2,                  // Input array (denominators)
  cv::OutputArray dst,                   // Results array (scale/src2)
  int             dtype = -1             // dst data type, -1 to get from src2
);
```

$$dst_i = saturate\left(scale * \frac{src1_i}{src2_i}\right)$$

$$dst_i = saturate(scale / src2_i)$$

cv::divide()是一个实现除法的简单函数，它将src1中的所有元素除以src2中的相应元素，然后将结果放在dst中。

cv::eigen()

```
bool cv::eigen(
  cv::InputArray  src,
```

```
  cv::OutputArray eigenvalues,
  int             lowindex     = -1,
  int             highindex    = -1
);

bool cv::eigen(
  cv::InputArray  src,
  cv::OutputArray eigenvalues,
  cv::OutputArray eigenvectors,
  int             lowindex    = -1,
  int             highindex   = -1
);
```

给定一个对称矩阵mat，cv::eigen()会计算出该矩阵的特征向量和特征值。矩阵必须为浮点类型之一。特征值矩阵以递减的顺序包含mat的特征值。如果要提供矩阵的特征向量，特征向量则以行的形式存储在矩阵中，并且与对应的特征值在特征值矩阵中的顺序相同。附加参数lowindex和highindex允许只计算一些特征值（两者必须一起使用）。例如，如果lowindex=0和highindex=1，就只计算最大的两个特征向量。

cv::exp()

```
void cv::exp(
  cv::InputArray  src,
  cv::OutputArray dst
);
```

求src中所有元素的指数，并将结果放在dst中。

cv::extractImageCOI()

```
bool cv::extractImageCOI(
  const CvArr*    arr,
  cv::OutputArray dst,
  int             coi = -1
);
```

函数cv::extractImageCOI()从旧版风格（版本2.1以前）矩阵中提取指定的COI，例如由arr给出的IplImage或CvMat，并将结果放到dst中。如果提供参数coi，该特定COI将被提取。否则，检查src中的COI字段以确定要提取哪个通道。



这里描述的cv::extractImageCOI()方法专门用于旧版的矩阵。如果需要从现在的cv::Mat对象中提取单个通道，请使用cv::mixChannels()或cv::split()。

cv::flip()

```
void cv::flip(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array, size and type of 'src'
  int             flipCode = 0            // >0: y-flip, 0: x-flip, <0: both
);
```

该函数将图像绕着*x*轴或*y*轴或者同时绕着*x*轴和*y*轴翻转。默认情况下，flipCode被设置为0，图像只绕*x*轴翻转。

如果flipCode被设置为大于0的数（例如，+1），图像会绕*y*轴翻转，如果被设置为一个负数（例如，-1），图像将围绕*x*轴和*y*轴翻转。

在Win32系统上进行视频处理时，你会发现自己经常使用此函数来进行图像格式变换，也就是坐标原点在左上角和左下角的变换。

cv::gemm()

```
void cv::gemm(
void cv::gemm(
  cv::InputArray  src1,                   // First input array
  cv::InputArray  src2,                   // Second input array
  double          alpha,                  // Weight for 'src1' * 'src2' product
  cv::InputArray  src3,                   // Third (offset) input array
  double          beta,                   // Weight for 'src3' array
  cv::OutputArray dst,                    // Results array
  int             flags = 0              // Use to transpose source arrays
);
```

广义矩阵乘法（GEMM）在OpenCV中由cv::gemm()来实现，可实现矩阵乘法、转置后乘法、比例乘法等。最常见的情况下，cv::gemm()计算如下：

$$D = \alpha \cdot op(\mathrm{src}_1)^{*}\ \mathrm{op}(src_2) + \beta \cdot \mathrm{op}(\mathrm{src}_3)$$

其中src1，src2和src3是矩阵，α和β是数值系数，op()是对所含矩阵的可选转置操作。转置是由可选参数flags来控制的，它的值可以是0或者是cv::GEMM_1_T，cv::GEMM_2_T和cv::GEMM_3_T（每一个标志都与一个矩阵转置相对应）的任意组合（通过布尔OR操作）。

每一个矩阵都必须有合适的尺寸来进行（矩阵）乘法，并且所有数据类型都应该是浮点型。cv::gemm()函数也支持双通道矩阵，在这种情况下，它将双通道视为一个复数的两个部分。



可以使用矩阵代数运算得到相似的结果。

例如：

```
cv::gemm(
  sc1, src2, alpha, src3, bets, dst,cv::GEMM_1_T | cv::GEMM_3_T
);
```

相当于：

```
dst = alpha * src1.T() * src2 + beta * src3.T()
```

cv::getConvertElem()和cv::getConvertScaleElem()

```
cv::convertData cv::getConvertElem(       // Returns a conversion function (below)
  int fromType,                           // Input pixel type (e.g., cv::U8)
  int toType                              // Output pixel type (e.g., CV_32F)
);

cv::convertScaleData cv::getConvertScaleElem(  // Returns a conversion function
  int fromType,                           // Input pixel type (e.g., cv::U8)
  int toType                              // Output pixel type (e.g., CV_32F)
);

// Conversion functions are of these forms:
//
typedef void (*ConvertData)(
  const void* from,                       // Pointer to the input pixel location
  void*       to,                         // Pointer to the result pixel location
  int         cn                          // number of channels
);

typedef void (*ConvertScaleData)(
  const void* from,                       // Pointer to the input pixel location
  void*       to,                         // Pointer to the result pixel location
  int         cn,                         // number of channels
  double      alpha,                      // scale factor
  double      beta                        // offset factor
);
```

函数cv::getConvertElem()和cv::getConvertScaleElem()返回OpenCV中用于特定类型转换的函数的函数指针。cv::getConvertElem()的函数返回值被定义（通过typedef）为cv::ConvertData类型，可以传递一个指向两个数据区域和“通道”数的指针。通道数由转换函数的参数cn给定，该数实际上是要转换的fromType类型的对象所占连续内存的数量。这意味着你可以简单地将通道数设置为矩阵中的元素总数，以此来转换整个（连续的内存）矩阵。

cv::getConvertElem()和cv::getConvertScaleElem()以两种类型作为参数：fromType和toType。这些类型用整数常数（例如，CV_32F）指定。

在cv::getConvertScaleElem()的情况下，返回的函数需要两个额外的参数alpha和beta。这些值由转换函数在转换为所需类型之前使用，从而重新缩放（alpha）和偏移（beta）输入值。

cv::idct()

```
void cv::idct(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             flags,                  // for row-by-row
);
```

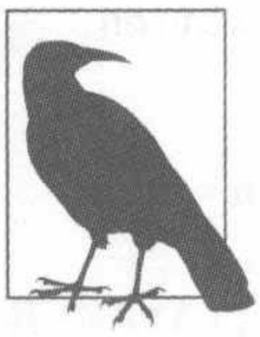
cv::idct()只是离散余弦逆变换的一个方便的简写。对cv::idct()的调用实际上相当于调用带参数的cv::dct()：

```
cv::dct( src, dst, flags | cv::DCT_INVERSE );
```

```
cv::idft()
void cv::idft(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             flags       = 0,        // for row-by-row, etc.
  int             nonzeroRows = 0         // only this many entries are nonzero
);
```

cv::idft()只是离散傅里叶逆变换的一个方便的简写。对cv::idft()的调用实际上相当于调用带参数的cv::dft()：

```
cv::dft( src, dst, flags | cv::DCT_INVERSE, outputRows );
```



默认情况下，cv::dft()和cv::idft()都不会缩放输出，所以你可能想要调用带cv::DFT_SCALE参数的cv::idft()；在这种情况下，变换和它的“逆”变换将是真正的逆运算。

cv::inRange()

```
void cv::inRange(
  cv::InputArray  src,                    // Input array
  cv::InputArray  upperb,                 // Array of upper bounds (inclusive)
  cv::InputArray  lowerb,                 // Array of lower bounds (inclusive)
  cv::OutputArray dst                     // Result array, cv::U8C1 type
);
```

$$dst_i = lowerb_i \le src_i \le upperb_i$$

当应用于一维矩阵时，src的每个元素都与upperb和lowerb中的对应元素进行校验。如果src中的元素在由upperb和lowerb给出的值之间，则dst的相应元素设置为255；否则设置为0。

然而，当src，upperb和lowerb是多通道矩阵时，输出仍然是单通道的。当且仅当src中的相应条目的值都位于upperb和lowerb中相应通道所对应的值之间时，元素*i*的输出值将被设置为255。从这个意义上来说，upperb和lowerb为每个像素定义了一个*n*维超立方体，如果src中的像素位于该超立方体内，dst中的对应值只能设置为true（255）。

cv::insertImageCOI()

```
void cv::insertImageCOI(
  cv::InputArray  img,                  // Input array, single channel
  CvArr*          arr,                  // Legacy (pre v2.1) output array
  int             coi = -1              // Target channel
);
```

像我们在本章前面遇到的cv::extractImageCOI()函数一样，cv::insertImageCOI()函数旨在帮助我们使用像IplImage和CvMat这样的旧版（v2.1之前）矩阵。其目的是允许我们从新式C++ cv::Mat对象中获取数据，并将该数据写入旧式矩阵的一个特定通道中。输入img是单通道cv::Mat对象，而输出arr是多通道旧式对象。两者的尺寸必须相同。img中的数据将被复制到arr的coi通道上。



如果想将特定的COI从旧式矩阵提取到单通道C++矩阵，可以使用函数cv::extractImageCOI()。另一方面，cv::insertImageCOI()用于将单通道C++矩阵的内容写入旧式矩阵的特定通道中。如果不使用旧式矩阵，并且只想将一个C++矩阵插入到另一个C++矩阵中，则可以使用cv::merge()来执行此操作。

cv::invert()

```
double cv::invert(                      // Return 0 if 'src' is singular
  cv::InputArray  src,                  // Input Array, m-by-n
  cv::OutputArray dst                   // Result array, n-by-m
  int             method = cv::DECOMP_LU  // Method for (pseudo) inverse
);
```

cv::invert()求矩阵src的逆，并将结果放在dst中。输入矩阵必须是浮点类型，结果矩阵与输入矩阵的类型相同。因为cv::invert()包括计算伪逆的可能性，所以输入矩阵不需要是方阵。如果输入矩阵尺寸为$n \times m$，则结果矩阵尺寸为$m \times n$。该函数支持计算逆矩阵的几种方法（表5-5），但默认为高斯消除。返回值取决于所使用的方法。

表5-5：cv::invert()方法参数的可能值

方法的参数值	意义
cv::DECOMP_LU	高斯清除法（LU分解）
cv::DECOMP_SVD	奇异值分解（SVD）
cv::DECOMP_CHOLESKY	只用于对称正定矩阵

在高斯消元（cv::DECOMP_LU）的情况下，当函数完成时，返回src的行列式。如果行列式为0，则求逆失败，矩阵dst的所有元素被设为0。

在cv::DECOMP_SVD的情况下，返回值是矩阵的逆条件数（最小特征值跟最大特征值的比例）。如果矩阵src是奇异的，则cv::invert()在SVD模式中将进行伪逆计算。另外两种方法（LU和Cholesky分解）要求源矩阵是方阵，非奇异和正定的。

cv::log()

```
void cv::log(
  cv::InputArray  src,
  cv::OutputArray dst
);
```

$$dst_i = \begin{cases} \log src_i & src_i \neq 0 \\ -C & else \end{cases}$$

cv::log()计算src1中的元素的自然对数，并将结果放在dst中。当源像素小于或等于零时，目标对应像素上会被标记为一个大的负数。

cv::LUT()

```
void cv::LUT(
  cv::InputArray  src,
  cv::InputArray  lut,
  cv::OutputArray dst
);
```

$$dst_i = \text{lut}(src_i)$$

函数cv::LUT()对输入的src进行“查找表变换”。cv::LUT()要求源矩阵src为8位索引值。lut矩阵保存查找表。该查找表矩阵应该具有256个元素，并且可以具有单个通道，或者在多通道src矩阵的情况下，与源矩阵具有相同数量的通道。然后函数cv::LUT()使用src中的相应值作为索引，从查找表lut中找到对应数值填充到目标矩阵dst中。

在src中的值为8位有符号数的情况下，它们会自动偏移+128，以使它们的范围对查找表索引来说有意义。如果查找表是多通道（并且索引也是多通道的），则src中的值被用作lut的多维索引，结果矩阵dst将是单通道的。如果lut是一维的，则结果矩阵将是多通道的，每个通道由src的相应索引和一维查找表分别计算。

```
cv::magnitude()
void cv::magnitude(
  cv::InputArray  x,
  cv::InputArray  y,
  cv::OutputArray dst
);
```

$$dst_i = \sqrt{x_i^2 + y_i^2}$$

cv::magnitude()主要计算在二维向量场上笛卡尔坐标系到极坐标系转换的径向部分。在cv::magnitude()的情况下，该向量场应该表示为两个单独的单通道矩阵的形式。这两个输入矩阵应该具有相同的尺寸。（如果有一个双通道矩阵，则cv::split()将给你分离后的单通道。）dst中的每个元素都从x和y的相应元素中计算两者的欧几里得范数（即，对应值的平方和的平方根）得到。

cv::Mahalanobis()

```
cv::Size cv::mahalanobis(
    cv::InputArray  vec1,
    cv::InputArray  vec2,
    cv::OutputArray icovar
);
```

cv::Mahalanobis()计算值：

$$r_{mahalonobis} = \sqrt{(\vec{x} - \vec{\mu})^T \Sigma^{-1} (\vec{x} - \vec{\mu})}$$

马氏距离被定义为一点和高斯分布中心之间的向量距离；该距离使用该分布的逆协方差作为度量来计算（图5-2）。直观上，这类似于基础统计学中的标准分数（z-score），某一点到分布中心的距离是以该分布的方差为单位来衡量的。马氏距离则是该思路在高维空间中的推广。

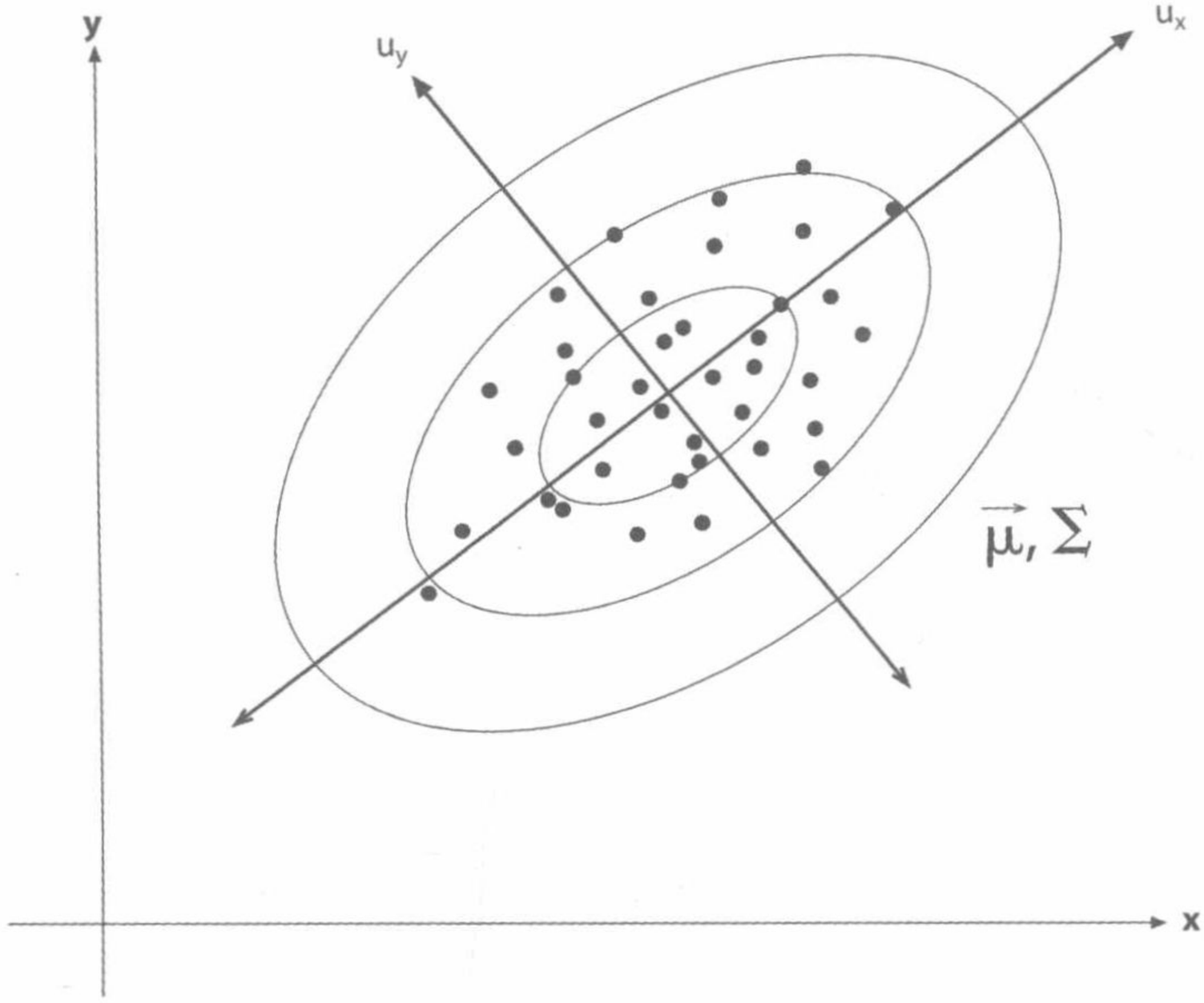


图5-2. 点在二维空间分布，叠加在一起的椭圆分别到分布中心的马氏距离为1.0,2.0,3.0

假设向量vec1对应x点，向量vec2是分布的均值。（Mahalanobis距离（马氏距离）。[注5] 矩阵icovar是协方差矩阵的逆。



这个协方差矩阵通常使用cv::calcCovarMatrix()（前面所述），然后用cv::invert()求逆来进行计算。使用cv::DECOMP_SVD方法求逆是良好的编程习惯，因为有一天你会遇到一个分布，该分布的特征值之一是0!

cv::max()

```
cv::MatExpr cv::max(
  const cv::Mat& src1,                    // First input array (first position)
  const cv::Mat& src2                     // Second input array
);

MatExpr cv::max(                          // A matrix expression, not a matrix
  const cv::Mat&  src1,                   // First input array (first position)
  double          value                   // Scalar in second position
);
```

注5: 实际上，马氏距离通常被定义为任何两个向量之间的距离；在任何情况下，矢量vec2从向量vec1中减去。cvMahalanobiscv::Mahalanobis()中的mat与逆协方差之间也没有任何基本的连接，任何度量都可以在这里酌情施加。

```
MatExpr cv::max(                          // A matrix expression, not a matrix
  double          value,                  // Scalar in first position
  const cv::Mat&  src1                    // Input array (second position)
);

void cv::max(
  cv::InputArray  src1,                   // First input array
  cv::InputArray  src2,                   // Second input array
  cv::OutputArray dst                     // Result array
);

void cv::max(
  const Mat&      src1,                   // First input array
  const Mat&      src2,                   // Second input array
  Mat&            dst                     // Result array
);

void cv::max(
  const Mat&      src1,                   // Input array
  double          value,                  // Scalar input
  Mat&            dst                     // Result array
);
```

$$dst_i = \max(src_{1,i}, src_{2,i})$$

cv::max()计算矩阵src1和src2中每个对应的像素对的最大值。它有两种基本形式：一种是返回一个矩阵表达式，另一种是计算结果并将其放在你指定的地方。在三参数形式中，在其中一个参数是cv::Scalar的情况下，与多通道矩阵的比较是以每个通道为基础与cv::Scalar的相应元素进行比较。

cv::mean()

```
cv::Scalar cv::mean(
  cv::InputArray  src,
  cv::InputArray  mask = cv::noArray(),    // Optional, do only where nonzero
);
```

$$N = \sum_{i, mask_i \neq 0} 1$$

$$mean_c = \frac{1}{N} \sum_{i, mask_i \neq 0} src_i$$

函数cv::mean()计算输入矩阵src中未被屏蔽的所有像素的平均值。如果src是多通道，则以每个通道为基础计算结果。

cv::meanStdDev()

```
void cv::meanStdDev(
  cv::InputArray  src,
  cv::OutputArray mean,
  cv::OutputArray stddev,
  cv::InputArray  mask  = cv::noArray(),   // Optional, do only where nonzero
);
```

$$N = \sum_{i,mask_i \neq 0} 1$$

$$mean_c = \frac{1}{N} \sum_{i,mask_i \neq 0} src_i$$

$$stddev_c = \sqrt{\sum_{i,mask \neq 0} (src_{c,i} - mean_c)^2}$$

函数cv::meanStdDev()计算输入矩阵src中未被屏蔽的像素的平均值以及它们的标准差。如果src是多通道的，则以每个通道为基础计算平均值和标准差。



这里计算的标准差与协方差矩阵不相同。实际上，这里计算的标准差只是协方差矩阵的对角元素。如果要计算协方差矩阵，则必须使用cv::calcCovarMatrix()。

cv::merge()

```
void cv::merge(
  const cv::Mat*  mv,                    // C-style array of arrays
  size_t          count,                 // Number of arrays pointed to by 'mv'
  cv::OutputArray dst                    // Contains all channels in 'mv'
);

void merge(
  const vector<cv::Mat>& mv,             // STL-style array of arrays
  cv::OutputArray dst                    // Contains all channels in 'mv'
);
```

cv::merge()是cv::split()的逆运算。mv中包含的矩阵被合并到输出矩阵dst中。在mv是指向cv::Mat对象的C风格矩阵的指针的情况下，还必须提供附加的数量参数count。

cv::min()

```
cv::MatExpr cv::min(                     // A matrix expression, not a matrix
  const cv::Mat&  src1,                  // First input array
```

```
  const cv::Mat&  src2                  // Second input array
);

MatExpr cv::min(                        // A matrix expression, not a matrix
  const cv::Mat&  src1,                 // First input array (first position)
  double          value                 // Scalar in second position
);

MatExpr cv::min(                        // A matrix expression, not a matrix
  double          value,                // Scalar in first position
  const cv::Mat&  src1                  // Input array (second position)
);

void cv::min(
  cv::InputArray  src1,                 // First input array
  cv::InputArray  src2,                 // Second input array
  cv::OutputArray dst                   // Result array
);

void cv::min(
  const Mat&      src1,                 // First input array
  const Mat&      src2,                 // Second input array
  Mat&            dst                   // Result array
);

void cv::min(
  const Mat&      src1,                 // Input array
  double          value,                // Scalar input
  Mat&            dst                   // Result array
);
```

$dst_i = \min(\text{src1},i, \text{src2},i)$

cv::min()计算矩阵src1和src2（或一个源矩阵和单个值）中每个对应的像素对的最小值。请注意，cv::min()的变体：返回值或矩阵表达式，可以由OpenCV的矩阵表达式机制控制。

在三参数形式中，在其中一个参数是cv::Scalar的情况下，与多通道矩阵的比较是以每个通道为基础与cv::Scalar的相应元素进行比较。

cv::minMaxIdx()

```
void cv::minMaxIdx(
  cv::InputArray src,                   // Input array, single channel only
  double*        minVal,                // min value goes here (in not NULL)
  double*        maxVal,                // min value goes here (in not NULL)
  int*           minIdx,                // loc of min goes here (if not NULL)
  int*           maxIdx,                // loc of max goes here (if not NULL)
  cv::InputArray mask = cv::noArray()   // search only nonzero values
);
```

```
void cv::minMaxIdx(
  const cv::SparseMat& src,                    // Input sparse array
  double*        minVal,                       // min value goes here (in not NULL)
  double*        maxVal,                       // min value goes here (in not NULL)
  int*           minIdx,                       // C-style array, indices of min locs
  int*           maxIdx,                       // C-style array, indices of max locs
);
```

这些例程找到矩阵src中的最小值和最大值，并且（有选择性地）返回它们的位置。计算出的最小值和最大值赋值给minVal和maxVal。可选的方式是，这些最值的位置也可以返回，这适用于具有任意维数的矩阵。这些位置将写入minIdx和maxIdx所给的地址中（前提是这些参数不是NULL）。

cv::minMaxIdx()也可以用cv::SparseMat类型的src矩阵调用。在这种情况下，矩阵可以是任意维数，最小值和最大值将被计算并返回它们的位置，此时，最值的位置将返回并放置在C风格的矩阵minLoc和maxLoc中。这些矩阵（如果提供的话）应该有与src矩阵维数相同的元素数量。在cv::SparseMat的情况下，最小值和最大值仅针对非零元素进行计算；然而，这个术语有点误导，因为真正的意思是存在于内存当中的稀疏矩阵中的元素。事实上，由于稀疏矩阵如何形成及过去已经对它做了什么的问题，可能产生一些存在且为0的元素，这些元素将被包括在最小值和最大值的计算中。



当提供一个单维度矩阵时，存放位置的矩阵必须有分配给两个整数的内存。这是因为cv::minMaxLoc()使用了这样一个惯例：一个单通道的矩阵本质上是一个$N \times 1$的矩阵。在这种情况下，返回的第二个值始终为0。

cv::minMaxLoc()

```
void cv::minMaxLoc(
  cv::InputArray src,                          // Input array
  double*        minVal,                       // min value goes here (in not NULL)
  double*        maxVal,                       // min value goes here (in not NULL)
  cv::Point*     minLoc,                       // loc of min goes here (if not NULL)
  cv::Point*     maxLoc,                       // loc of max goes here (if not NULL)
  cv::InputArray mask = cv::noArray()          // search only nonzero values
);

void cv::minMaxLoc(
  const cv::SparseMat& src,                    // Input sparse array
  double*        minVal,                       // min value goes here (in not NULL)
  double*        maxVal,                       // min value goes here (in not NULL)
  cv::Point*     minLoc,                       // C-style array, indices of min locs
  cv::Point*     maxLoc,                       // C-style array, indices of max locs
);
```

此例程在矩阵src中找到最小值和最大值并（可选）返回其位置。计算的最小值和最大值分别置于minVal和maxVal中。可选的方式是，也可以返回这些最值的位置。这些位置将被写入minLoc和maxLoc给出的地址中（前提是这些参数不为NULL）。因为这些位置的类型是cv::Point，所以这种形式的函数只能用在二维矩阵（即矩阵或图像）上。

与cv::minMaxIdx()一样，在稀疏矩阵的情况下，在搜索最小值或最大值时，仅考虑有效元素。

在处理多通道矩阵时，有几个选项。cv::minMaxLoc()原本不支持多通道输入。主要是因为这个操作是不明确的。



如果希望得到所有通道的最小值和最大值，可以使用cv::reshape()将多通道矩阵重新整合成一个巨大的单通道矩阵。如果想分别为每个通道选择最小和最小值，则可以使用cv::split()或cv::mixChannels()分离通道并分别进行分析。

在两种形式的cv::minMaxLoc中，最小值或最大值或位置的参数可以设置为NULL，这时不计算该参数。

cv::mixChannels()

```
void cv::mixChannels(
  const cv::Mat*          srcv,          // C-style array of matrices
  int                     nsrc,          // Number of elements in 'srcv'
  cv::Mat*                dstv,          // C-style array of target matrices
  int                     ndst,          // Number of elements in 'dstv'
  const int*              fromTo,        // C-style array of pairs, ...from,to...
  size_t                  n_pairs        // Number of pairs in 'fromTo'
);

void cv::mixChannels(
  const vector<cv::Mat>& srcv,           // STL-style vector of matrices
  vector<cv::Mat>&       dstv,           // STL-style vector of target matrices
  const int*              fromTo,        // C-style array of pairs, ...from,to...
  size_t                  n_pairs        // Number of pairs in 'fromTo'
);
```

从输入的一个或多个图像中重新排列通道并在输出的一个或多个图像中将它们分到特定的通道，OpenCV中有许多操作是这个一般问题的特殊情况。像cv::split()，cv::merge()和（至少有些情况的）cv::cvtColor()这样的函数都使用这样的功能。这些方法通过调用更通用的cv::mixChannels()来做自己需要做的事情。此函数允许你将多个

矩阵作为输入，每个矩阵具有多个通道，输出也一样，并且以任何你选择的方式将输入矩阵中的通道映射到输出矩阵的通道中。

输入矩阵和输出矩阵可以被指定为由cv::Mat对象组成的C风格矩阵，并带有用来表明cv::Mats数量的附加整数。或者由cv::Mat对象组成的STL向量。输出矩阵必须预先分配，其尺寸和维数与输入矩阵的尺寸和维数相匹配。

映射是由C风格的整数矩阵fromTo控制的。该矩阵可以按顺序包含任意数量的整数对，每对的第一个值表示源通道，第二个值表示应复制的目标通道。对于第一个图像，通道顺序编号从1开始，然后通过第二个图像，以此类推（图5-3）。总数由参数n_pairs提供。

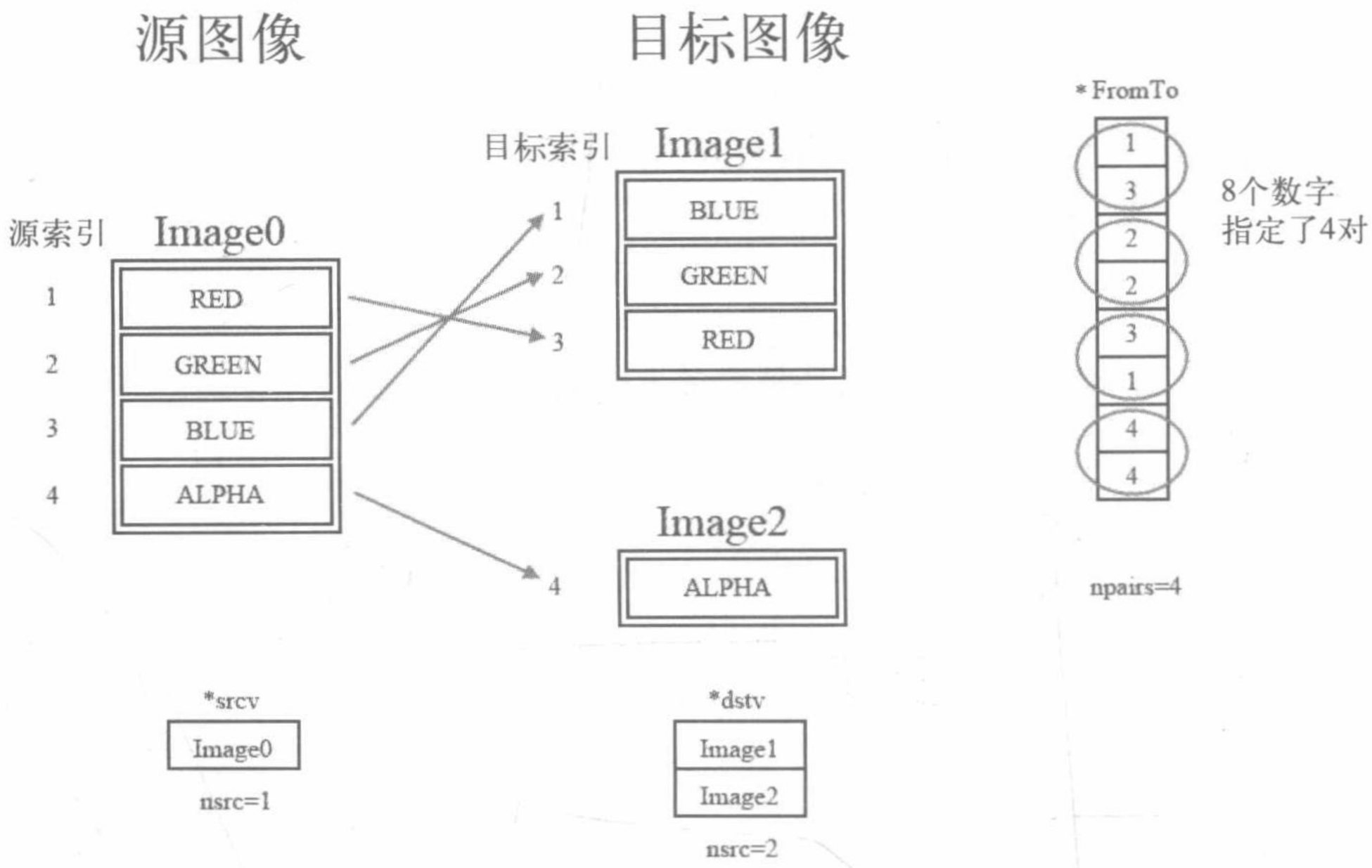
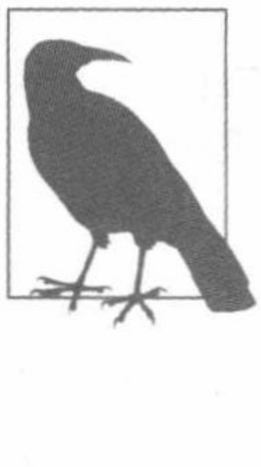


图5-3. 单个四通道RGBA图像转换为一个BGR图像和一个alphaonly图像



与2.1版以后的库中其他大多数函数不同，cv::mixChannels()不为输出矩阵分配内存。它们必须预先分配，并具有与输入矩阵相同的尺寸和维度。

cv::mulSpectrums()

```
doublevoid cv::mulSpectrums(
    cv::InputArray  arr1,                 // First input array
    cv::InputArray  arr2,                // Second input array, same size as 'arr1'
```

```
    cv::OutputArray dst,              // Result array, same size as 'arr1'
    int             flags,            // used to indicate independent rows
    bool            conj  = false     // If true, conjugate arr2 first
);
```

在涉及频谱的许多操作（即cv::dft()或cv::idft()的结果）中，人们希望做一个遵循频谱规则（实矩阵）的逐元素乘法，或者将它们作为复数变量。（更多详细信息，请参阅cv::dft()的说明。）输入矩阵可以是一维或二维的，第二个矩阵与第一个矩阵具有相同的尺寸和类型。如果输入矩阵是二维的，则可以将其视为一个真正的二维频谱或一维频谱矩阵（每行一个）。在后一种情况下，标志位应设置为cv::DFT_ROWS；否则可以设置为0。

当两个矩阵为复数时，它们简单地逐元素相乘，但是cv::mulSpectrums()提供了在乘法之前共轭第二个矩阵元素的选项。例如，你将使用此选项来执行相关性操作（使用傅里叶变换），但是对于卷积，使用conj = false。

cv::multiply()

```
void cv::multiply(
  cv::InputArray  src1,               // First input array
  cv::InputArray  src2,               // Second input array
  cv::OutputArray dst,                // Result array
  double          scale = 1.0,        // overall scale factor
  int             dtype = -1          // Output type for result array
);
```

$$dst_i = saturate(scale*src1_i*src2_i)$$

cv::multiply()是一个简单的乘法函数；它将src1中的元素乘以src2中的相应元素，并将结果放在dst中。

cv::mulTransposed()

```
void cv::mulTransposed(
  cv::InputArray  src1,               // Input matrix
  cv::OutputArray dst,                // Result array
  bool            aTa,                // If true, transpose then multiply
  cv::InputArray  delta = cv::noArray(), // subtract from 'src1' before multiply
  double          scale = 1.0,        // overall scale factor
  int             dtype = -1          // Output type for result array
);
```

$$dst = \begin{cases} scale*(src - delta)^T(src - delta) & aTa = true \\ scale*(src - delta)(src - delta)^T & aTa = false \end{cases}$$

cv::mulTransposed()用于计算矩阵与其转置的乘积，例如计算协方差。src应为二维单通道矩阵，但不同于cv::GEMM()，它不限于浮点类型。除非另有指定，否则结果矩阵将与源矩阵的类型相同。如果dtype不为负，输出类型应为CV_32F或cv::F64，输出矩阵dst将是指定类型。

如果提供了第二个输入矩阵delta，则在乘法之前将从src中减去该矩阵。如果没有提供该矩阵（即，delta = cv::noArray()），则不进行减法。矩阵delta不需要与src尺寸相同；如果delta尺寸小于src，则重复delta（也称为*tiling*；请参阅cv::repeat()），以生成一个尺寸与src尺寸相匹配的矩阵。在乘法完成之后，将参数scale应用于矩阵。最后，参数aTa用于选择乘法中src的转置在左侧（aTa = true）或在右侧（aTa = false）。

cv::norm()

```
double cv::norm(                                  // Return norm in double precision
  cv::InputArray src1,                            // Input matrix
  int            normType = cv::NORM_L2,          // Type of norm to compute
  cv::InputArray mask     = cv::noArray() // do for nonzero values (if present)
);

double cv::norm(                                  // Return computed norm of difference
  cv::InputArray src1,                            // Input matrix
  cv::InputArray src2,                            // Second input matrix
  int            normType = cv::NORM_L2,  // Type of norm to compute
  cv::InputArray mask     = cv::noArray() // do for nonzero values (if present)
);

double cv::norm(
  const cv::SparseMat& src,                       // Input sparse matrix
  int                  normType = cv::NORM_L2, // Type of norm to compute
);
```

$$\|src1\|_{\infty, L1, L2}$$

$$\|src1 - src2\|_{\infty, L1, L2}$$

函数cv::norm()用于计算一个矩阵的范数（表5-6），或者如果提供两个矩阵，该函数也可以计算两个矩阵间的各种距离范数（表5-7）。也可以计算cv::SparseMat的范数，在这种情况下，范数的计算中忽略零项。

表5-6：当src2=NULL时，对于不同的normType值，由cv::norm()计算范数的公式

normType	Result
cv::NORM_INF	$\|src1\|_{\infty} = \max_i abs(src1_i)$
cv::NORM_L1	$\|src1\|_{L1} = \sum_i abs(src1_i)$
cv::NORM_L2	$\|src1\|_{L2} = \sqrt{\sum_i src1_i^2}$

如果第二个矩阵参数src2非空，那么范数的计算将使用不同的公式，就像两个矩阵之间的距离[注6]。前三种情况（如表5-7所示），这些范数是绝对范数；在后三种情况下，将会根据第二个矩阵src2的幅度进行重新调整。

表5-7：当src2非空时，对于不同的normType值，由cv::norm()计算范数的公式

normType	Result
cv::NORM_INF	$\|src1 - src2\|_{\infty} = \max_i abs(src1_i - src2_i)$
cv::NORM_L1	$\|src1 - src2\|_{L1} = \sum_i abs(src1_i - src2_i)$
cv::NORM_L2	$\|src1 - src2\|_{L2} = \sum_i (src1_i - src2_i)^2$
cv::NORM_RELATIVE_INF	$\frac{\|src1 - src2\|_{\infty}}{\|src2\|_{\infty}}$
cv::NORM_RELATIVE_L1	$\frac{\|src1 - src2\|_{L1}}{\|src2\|_{L1}}$
cv::NORM_RELATIVE_L2	$\frac{\|src1 - src2\|_{L2}}{src2_{L2}}$

在所有情况下，src1和src2必须具有相同的尺寸和通道数。当通道数大于1时，将会对所有通道一起计算范数（即在表5-6和表5-7中，不仅是针对i，也针对通道数求和）。

cv::normalize()

```
void cv::normalize(
  cv::InputArray  src1,                    // Input matrix
```

注6：至少在计算L2范数的情况下，在维数与图像像素数相同的空间中，差分范数可直观理解为欧式距离。

```
  cv::OutputArray dst,                    // Result matrix
  double          alpha    = 1,           // first parameter (see Table 5-8
  double          beta     = 0,           // second parameter (see Table 5-8
  int             normType = cv::NORM_L2, // Type of norm to compute
  int             dtype = -1              // Output type for result array
  cv::InputArray  mask  = cv::noArray()   // do for nonzero values (if present)
);

void cv::normalize(
  const cv::SparseMat& src,               // Input sparse matrix
  cv::SparseMat&       dst,               // Result sparse matrix
  double               alpha    = 1,      // first parameter (see Table 5-8
  int                  normType = cv::NORM_L2, // Type of norm to compute
);
```

$$\| dst \|_{\infty, L1, L2} = \alpha$$

$$\min(dst) = \alpha, \max(dst) = \beta$$

与许多OpenCV函数一样，cv::normalize()的功能比表面看更多。根据normType的值，图像src被归一化，或者映射到一个特定范围内并放到dst中。除非使用dtype参数，否则矩阵dst与src具有相同的尺寸和数据类型。可选地，dtype可以设置为OpenCV基本类型之一（例如，CV_32F），并且输出矩阵将为该类型。此操作的确切含义取决于normType参数。normType的可能值如表5-8所示。

表5-8：cv::normalize()的参数normType可能的值

norm_type	结果值
cv::NORM_INF	$\| dst \|_{\infty} = \max_i abs(dst_i) = a$
cv::NORM_L1	$\| dst \|_{L1} = \sum_i abs(dst_i) = a$
cv::NORM_L2	$\| dst \|_{L2} = \sqrt{\sum_i dst_i^2} = a$
cv::NORM_MINMAX	映射到区间[α, β]

在无穷范数的情况下，矩阵src被重新缩放，使得最大元素的绝对值的大小等于α。在L1或L2范数的情况下，重新缩放矩阵，使得该范数的值等于α。如果normType设置为cv::MINMAX，则将矩阵的值进行重新缩放和转换，使其线性映射到alpha和beta（含）区间内。

如前所述，如果掩码非空；则只有对应掩膜图像的非零值的那些像素用于计算范数，并且只有那些像素将被cv::normalize()改变。请注意，如果dtype = cv::MIN MAX，源矩阵不能是cv::SparseMat。原因是cv::MIN_MAX操作可以用于整体偏移，这会影响矩阵的稀疏性（具体来说，由于这个操作，所有零元素都变为非0元素，所以稀疏矩阵将变为非稀疏矩阵）。

cv::perspectiveTransform()

```
void cv::perspectiveTransform(
  cv::InputArray  src,                    // Input array, 2 or 3 channels
  cv::OutputArray dst,                    // Result array, size, type, as src1
  cv::InputArray  mtx                     // 3-by-3 or 4-by-4 transoform matrix
);
```

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x'/w' \\ y'/w' \\ z'/w' \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} \rightarrow [mtx] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

cv::perspectiveTransform()函数执行一系列点（而不是像素）的平面投影变换。输入矩阵应为二通道或三通道矩阵，在这两种情况下，矩阵mtx尺寸分别为3×3与4×4。cv::perspectiveTransform()首先将src的每个元素转换为长度为src.channels() + 1的向量，其中附加维度（投影维度）初始化为1.0。这也称为齐次坐标。然后将每个扩展向量乘以mtx，并将结果重新排列为（新）投影坐标的值[注7]（然后将附加维度抛弃，因为在此操作以后始终为1.0）。



请再注意一点，此例程用于转换一系列点，而不是图像。如果要将透视变换应用于图像，你实际上不是转换单个像素，而是将它们从图像中的一个位置移动到另一个位置。这是cv::warpPerspective()的工作。

如果要解决逆问题，即给定很多对的对应点，找到最可能的透视变换，请使用cv::getPerspectiveTransform()或cv::findHomography()。

cv::phase()

```
void cv::phase(
  cv::InputArray  x,                      // Input array of x-components
  cv::InputArray  y,                      // Input array of y-components
  cv::OutputArray dst,                    // Output array of angles (radians)
```

注7：在技术上，乘以mtx后，w'的值将为0，对应于投影到无限远的点。在这种情况下，不是除以零，而是将值0分配给该比率。

```
  bool            angleInDegrees = false  // degrees (if true), radians (if false)
);
```

$$dst_i = \text{atan2}(y_i, x_i)$$

cv::phase()对二维矢量场计算笛卡尔－极坐标转换的方位角（角度）部分。该矢量场是由两个独立的单通道矩阵组成。当然这两个输入矩阵的尺寸相同。（如果你有一个二通道的矩阵，那么调用cv::split()将会做你所需要的。）然后，dst中的每一个元素都从x和y的相应元素中计算两者的反正切值得到。

cv::polarToCart()

```
void cv::polarToCart(
  cv::InputArray  magnitude,              // Input array of magnitudes
  cv::InputArray  angle,                  // Input array of angles
  cv::OutputArray x,                      // Output array of x-components
  cv::OutputArray y,                      // Output array of y-components
  bool            angleInDegrees = false  // degrees (if true) radians (if false)
);
```

$$x_i = magnitude_i * \cos(angle_i)$$

$$y_i = magnitude_i * \sin(angle_i)$$

cv::polarToCart()从向量场的极坐标中计算笛卡尔坐标（x, y）。输入具有相同尺寸和类型的两个矩阵：幅度和角度，指定每个点处向量的幅度和角度。输出类似的两个矩阵，它们与输入具有相同的尺寸和类型，并且将包含每个点处向量的x和y投影。附加标志angleInDegrees将使角度矩阵中的数值以度为单位，而不是弧度。

cv::pow()

```
void cv::pow(
  cv::InputArray  src,                    // Input array
  double          p,                      // power for exponentiation
  cv::OutputArray dst                     // Result array
);
```

$$dst_i = \begin{cases} src_i^{\,p} & p \in \mathbb{Z} \\ |src_i|^{\,p} & else \end{cases}$$

函数cv::pow()对矩阵逐元素取p（给定幂数）次幂。在p为整数的情况下，则直接进行幂运算。对于非整数p，首先计算源矩阵的绝对值，然后取p次幂（因此只返回实数值）。对于一些特殊的p值，例如整数值或±½，使用特殊算法来进行更快的计算。

cv::randu()

```
template<typename _Tp> _Tp randu(); // Return random number of specific type

void cv::randu(
  cv::InputOutArray mtx,            // All values will be randomized
  cv::InputArray    low,            // minimum, 1-by-1 (Nc=1,4), or 1-by-4 (Nc=1)
  cv::InputArray    high            // maximum, 1-by-1 (Nc=1,4), or 1-by-4 (Nc=1)
);
```

$mtx_i \in low_i, high_i)$

有两种方法调用cv::randu()。第一种方法是调用randu<>()的模板形式，它将返回适当类型的随机值。以这种方式生成的随机数是均匀分布的[注8]，对于整数，产生从0到该类型的最大值范围内的数，对于浮点类型，产生0.0到1.0（不包括1.0）范围内的数。此模板形式仅生成单个数字[注9]。

调用cv::randu()的第二种方法是提供一个你希望填充数值的矩阵mtx以及两个附加的矩阵，指定要为每个特定的矩阵元素生成随机数的范围的最小值和最大值。附加的low和high应为尺寸1×1，具有1个或4个通道的矩阵，或者尺寸为1×4的单通道矩阵；它们也可以是cv::Scalar类型。在任何情况下，它们都不是mtx的尺寸，而是mtx中单个条目的尺寸。

mtx既是输入矩阵也是输出矩阵，在某种意义上，必须为矩阵分配内存，以使cv::randu()知道你需要随机值的数量以及它们在行、列以及通道上如何排列。

cv::randn()

```
void cv::randn(
  cv::InputOutArray mtx,            // All values will be randomized
  cv::InputArray    mean,           // mean values, array is in channel space
  cv::InputArray    stddev          // standard deviations, channel space
);
```

$mtx_i \sim N\ (mean_i, stddev_i)$

函数cv::randn()使用随机正态分布的值填充矩阵mtx[注10]。这些值的生成参数来自两个附加的矩阵（mean和stddev），它们指定分布均值和标准差，从而为每个特定的矩阵元素生成随机数。

注8： 均匀分布随机数利用Multiply-With-Carry算法[Goresky03]生成。

注9： 具体而言，这意味着如果你使用向量参数调用模板形式，例如cv::randu<Vec4f>，则返回值虽然为向量类型，但除了第一个元素，其余元素全部为0。

注10： 使用Ziggurat算法[Marsaglia00]生成高斯分布随机数。

与cv::randu()的矩阵形式一样，mtx的每个元素都是单独计算的，矩阵mean和stddev在各个通道中对mtx的每个元素进行取值。因此，如果mtx是四通道矩阵，则mean和stddev将为尺寸为1×4的矩阵，或着为具有四个通道（或类似于cv::Scalar类型），尺寸为1×1的矩阵[注11]。

```
cv::randShuffle()
void cv::randShuffle(
  cv::InputOutArray mtx,                    // All values will be shuffled
  double            iterFactor = 1,         // Number of times to repeat shuffle
  cv::RNG*          rng        = NULL       // your own generator, if you like
);
```

cv::randshuffle()尝试通过选择随机元素对和交换其位置来随机化一维矩阵中的元素。交换的数量等于矩阵mtx的尺寸乘以可选因子iterFactor。可选的是，可以提供随机数生成器（更多信息，请参见第7章对随机数生成器（cv::RNG）的介绍）。如果没有提供，则自动使用默认的随机数生成器theRNG()。

```
cv::reduce()
void cv::reduce(
  cv::InputArray  src,                          // Input, n-by-m, 2-dimensional
  cv::OutputArray vec,                          // Output, 1-by-m or n-by-1
  int             dim,                          // Reduction direction 0=row, 1=col
  int             reduceOp = cv::REDUCE_SUM,    // Reduce operation (see #reduceop_
argument_in_cvreduceleft_parent)
  int             dtype = -1                    // Output type for result array
);
```

简化是指使用一些reduceOp所代表的组合规则，对输入矩阵src的每一行（或列）进行系统的转化，直到只剩一行（或列）为止（表 5-9）[注12]，使之成为向量vec。参数dim决定如何进行简化，总结如表5-10所示。

表5-9：参数reduceOp在cv::reduce()中所选择的简化操作

reduceOp的值	结果
cv::REDUCE_SUM	计算向量的总和
cv::REDUCE_AVG	计算向量的平均值
cv::REDUCE_MAX	计算向量中的最大值
cv::REDUCE_MIN	计算向量中的最小值

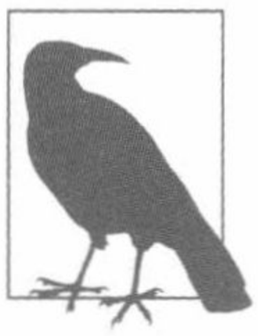
注11：注意，stddev不是方阵；cv::randn()不支持相关数生成。

注12：从严格意义上来讲，求平均并不是一个真正的简化方法。OpenCV从更实用的角度定义化简，因此在函数cvReduce中包括这一有用的函数。

表5-10：参数dim在cv::reduce()中控制简化的方向

dim的值	结果
0	合并成一行
1	合并成一列

cv::reduce()支持任意类型的多通道矩阵，通过使用dtype，我们可以为dst指定其他的类型。



使用dtype参数为dst指定一个高精度的数据类型，这对cv::REDUCE_SUM和cv::REDUCE_AVG来说尤为重要，否则可能出现溢出和累积问题。

cv::repeat()

```
void cv::repeat(
  cv::InputArray  src,                    // Input 2-dimensional array
  int             nx,                     // Copies in x-direction
  int             ny,                     // Copies in y-direction
  cv::OutputArray dst                     // Result array
);

cv::Mat cv::repeat(                       // Return result array
  cv::InputArray  src,                    // Input 2-dimensional array
  int             nx,                     // Copies in x-direction
  int             ny                      // Copies in y-direction
);
```

$$dst_{i,j} = src_{i\%src.rows,\ j\%src.cols}$$

本函数是将src的内容复制到dst中，根据需要重复多次以填充dst。具体而言，dst相对于src可以是任意尺寸。它可能比src大或比src小，它的维度与src对应的维度间不需要有任何的数值关系。

cv::repeat()有两个调用方式。第一个是输出矩阵作为引用传入cv::repeat()的旧式方式。第二个实际创建并返回一个cv::Mat，并且在使用矩阵表示时更方便。

cv::scaleAdd()

```
void cv::scaleAdd(
  cv::InputArray  src1,                   // First input array
  double          scale,                  // Scale factor applied to first array
  cv::InputArray  src2,                   // Second input array
```

```
  cv::OutputArray dst,                    // Result array
);
```

$$dst_i = scale * src1_i + src2_i$$

cv::scaleAdd()用于计算两个矩阵src1和src2的和，在求和之前，将比例因子scale应用于第一个矩阵。结果放在矩阵dst中。



使用矩阵代数运算可以得到相同的结果：

```
dst = scale * src1 + src2;
```

cv::setIdentity()

```
void cv::setIdentity(
  cv::InputOutputArray dst,                    // Array to reset values
  const cv::Scalar&    value = cv::Scalar(1.0)  // Apply to diagonal elements
);
```

$$dst_{i,j} = \begin{cases} value & i = j \\ 0 & else \end{cases}$$

cv::setIdentity()除了将行数和列数相等的元素设置为1（或其他给定数值）外，其他所有矩阵元素设置为0。cv::setIdentity()支持所有数据类型，不需要矩阵为方阵。



这也可以通过cv::Mat类的eye()成员函数来完成。当使用矩阵表达时，使用eye()通常更方便。

```
cv::Mat A( 3, 3, CV_32F );
cv::setIdentity( A, s );
C = A + B;
```

对于一些其他矩阵B和C以及某个标量s，这相当于：

```
C = s * cv::Mat::eye( 3, 3, CV_32F ) + B;
```

cv::solve()

```
int cv::solve(
  cv::InputArray  lhs,                    // Lefthand side of system, n-by-n
  cv::InputArray  rhs,                    // Righthand side of system, n-by-1
  cv::OutputArray dst,                    // Results array, will be n-by-1
```

```
    int             method = cv::DECOMP_LU  // Method for solver
  );
```

基于cv::invert()的函数cv::solve()为求解线性系统提供了一条捷径。具体的计算公式如下：

$$C = \text{argmin}_X \| A \cdot X - B \|$$

其中A是一个由lhs指定的方阵，B是向量rhs，C是由cv::solve()计算的结果，目标是寻找一个最优向量X。最优向量X将返回给dst。用于求解此系统的实际方法由method参数的值决定（参见表5-11）。只支持浮点数据类型。该函数将返回一个整数值，当返回的值是一个非零值的话，就表明找到了一个解。

表5-11：cv::solve()的method参数可能的值

method参数的值	意义
cv::DECOMP_LU	高斯消元法（LU分解）
cv::DECOMP_SVD	奇异值分解（SVD）
cv::DECOMP_CHOLESKY	对于对称正定矩阵
cv::DECOMP_EIG	特征值分解，只用于对称矩阵
cv::DECOMP_QR	QR因式分解
cv::DECOMP_NORMAL	可选附加标志；表示要求解标准方程

cv::DECOMP_LU和cv::DECOMP_CHOLESKY的方法不能用于奇异矩阵。如果提供一个奇异的lhs参数，两个方法都将退出并返回0（如果lhs是非奇异的，则返回1）。可以使用cv::solve()中的QR分解（cv::DECOMP_QR）或奇异值分解（cv::DECOMP_SVD）来求解给定方程组的最小二乘解，从而求解超定线性系统。这两种方法在矩阵lhs奇异的情况下也可以使用。

虽然表5-11中的前五个参数是互斥的，但最后一个参数cv::DECOMP_NORMAL可以与前五个中的任何一个组合（例如，通过逻辑OR：cv_DECOMP_LU | cv::DECOMP_NORMAL）。如果使用该参数，那么cv::solve()将尝试解决标准方程：$lhs^T \cdot lhs \cdot dst = lhs^T \cdot rhs$，而不是一般方程$lhs \cdot dst = rhs$。

cv::solveCubic()

```
int cv::solveCubic(
  cv::InputArray  coeffs,
  cv::OutputArray roots
);
```

给定由三或四个元素向量系数表示的三次多项式，cv::solveCubic()将计算该多项式的实根。如果coeffs有四个元素，则计算以下多项式的根：

$$\text{coeffs}_0 x^3 + \text{coeffs}_1 x^2 + \text{coeffs}_2 x + \text{coeffs}_3 = 0$$

如果coeffs只有三个元素，则计算以下多项式的根：

$$x^3 + \text{coeffs}_0 x^2 + \text{coeffs}_1 x + \text{coeffs}_2 = 0$$

结果存储在矩阵roots中，它将具有一个或三个元素，具体取决于多项式具有多少个实根。



关于cv::solveCubic()和cv::solvePoly()的提示：在两个例程中，输入矩阵系数的顺序是相反的。在cv::solve Cubic()中，最高阶系数在最后一个，而在cv::solvePoly()中，最高阶系数在第一个。

cv::solvePoly()

```
int cv::solvePoly (
  cv::InputArray  coeffs,
  cv::OutputArray roots                  // n complex roots (2-channels)
  int             maxIters = 300         // maximum iterations for solver
);
```

给定以系数向量表示的任意阶数的多项式，cv::solvePoly()将尝试计算该多项式的根。给定系数矩阵，计算以下多项式的根：

$$\text{coeffs}_n x^n + \text{coeffs}_{n-1} x^{n-1} + \cdots + \text{coeffs}_1 x + \text{coeffs}_0 = 0$$

不像cv::solveCubic()，这些根不能保证是实根。对于n阶多项式（即具有n+1个元素的coeffs），将存在n个根。因此，矩阵roots将返回双通道（实部，虚部）双精度矩阵。

cv::sort()

```
void cv::sort(
  cv::InputArray  src,
  cv::OutputArray dst,
  int             flags
);
```

OpenCV排序函数用于二维矩阵。只支持单通道源矩阵。你不要认为这样会像在电子表格中排序行或列；cv::sort()分别对每行或列进行排序。排序操作的结果将是一个新的矩阵dst，它与源矩阵具有相同的尺寸和类型。

可以通过使用cv::SORT_EVERY_ROW或cv::SORT_EVERY_COLUMN标志对每一行或每一列进行排序。排序可以是升序或者降序，分别由cv::SORT_ASCENDING或cv::SORT_DESCENDING标志指定。需要从每一组中各选一个标志。

cv::sortIdx()

```
void cv::sortIdx(
  cv::InputArray  src,
  cv::OutputArray dst,
  int             flags
);
```

与cv::sort()类似，cv::sortIdx()仅用于单通道二维矩阵。cv::sortIdx()分别对每行或每列进行排序。排序操作的结果是与源矩阵大小相同但包含排序元素整数索引的新矩阵dst。例如，给定一个矩阵A，调用cv::sortIdx(A, B, cv::SORT_EVERY_ROW | cv::SORT_DESCENDING)将产生：

$$A = \begin{bmatrix} 0.0 & 0.1 & 0.2 \\ 1.0 & 1.1 & 1.2 \\ 2.0 & 2.1 & 2.2 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 1 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 0 \end{bmatrix}$$

在这个简单的例子中，每排元素先前是从最低到最高排序的，排序操作将此顺序反转。

cv::split()

```
void cv::split(
  const cv::Mat&   mtx,
  cv::Mat*         mv
);

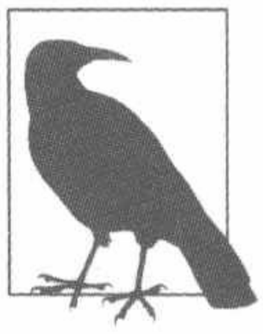
void cv::split(
  const cv::Mat&   mtx,
  vector<Mat>&     mv                  // STL-style vector of n 1-channel cv::Mat's
);
```

函数cv::split()是cv::mixChannels()的一个特别简单的例子。使用cv::split()将多通道矩阵中的通道分成多个单通道矩阵。这样做有两种方法。第一种方法是提供一个指向C风格矩阵的指针，该矩阵包含cv::split()将要保存分离操作结果的cv::Mat对象的指针。第二种方法是你提供一个由cv::Mat对象组成的STL向量。如果使用C风格的矩阵，务必确保可用的cv::Mat对象的数量（至少）等于mtx中的通道数。如果使用STL向量形式，cv::split()将为你分配结果矩阵的内存。

cv::sqrt()

```
void cv::sqrt(
  cv::InputArray  src,
  cv::OutputArray dst
);
```

作为cv::pow()的特例，cv::sqrt()计算矩阵中逐元素的平方根。多个通道分别进行处理。



（有时）存在矩阵的平方根；也就是说，矩阵B与某个矩阵A的关系满足$BB = A$。如果A是正定的方阵，则如果B存在，它就是唯一的。

如果A可以被对角化，那么就存在矩阵V（由A的特征向量按列组合），使得$A=VDV^{-1}$，其中D是一个对角矩阵。由于矩阵D的平方根恰好为矩阵D每个元素的平方根。所以如果想要计算我们只需要使用矩阵V并有：

$$A^{1/2} = VD^{1/2}V^{-1}$$

数学爱好者可以通过直接平方的方法来轻松验证此表达式是正确的：

$$\left(A^{1/2}\right)^2 = \left(VD^{1/2}V^{-1}\right)\left(VD^{1/2}V^{-1}\right) = VD^{1/2}V^{-1}VD^{1/2}V^{-1}$$
$$= VDV^{-1} = A$$

用代码表示是下面这样的[注13]：

```
void matrix_square_root( const cv::Mat& A, cv::Mat& sqrtA ) {
  cv::Mat U, V, Vi, E;
  cv::eigen( A, E, U );
  V = U.T();
  cv::transpose( V, Vi ); // inverse of the orthogonal V
  cv::sqrt(E, E);         // assume that A is positively
                          // defined, otherwise its
                          // square root will be
                          // complex-valued
  sqrtA = V * Mat::diag(E) * Vi;
}
```

注13：这就是说，如果你真的写出了负责任的代码，就做很多的检查，以确保你所传递的矩阵实际上是你所想的那样（即方阵）。你也可能要检查cv::eigen()和cv::invert()的返回值，更仔细地考虑用于分解和求逆的实际方法，在对它们调用sqrt()之前，务必保证特征值是正数。

cv::subtract()

```
void cv::subtract(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst,                     // Result array
  cv::InputArray  mask  = cv::noArray(),   // Optional, do only where nonzero
  int             dtype = -1               // Output type for result array
);
```

$$dst_i = saturate(src1_i - src2_i)$$

cv::subtract()是一个简单的减法函数：它从src1中减去src2的相应元素，并将结果放在dst中。



对于简单的情况，矩阵运算可以得到相同的结果：

```
dst = src1 - src2;
```

也支持累加：

```
dst -= src1;
```

cv::sum()

```
cv::Scalar cv::sum(
  cv::InputArray  arr
);
```

$$sum_c = \sum_{i,j} arr_{c\,i,j}$$

cv::sum()计算矩阵arr各个通道的所有像素的总和，返回值是cv::Scalar类型，因此cv::sum()可以进行多通道矩阵计算，但最多四个通道。每个通道的和被放在cv::scalar返回值的相应分量中。

cv::trace()

```
cv::Scalar cv::trace(
  cv::InputArray  mat
);
```

$$Tr(mat)_c = \sum_{i} mat_{c\,i,i}$$

矩阵的迹是对角线元素的总和。在OpenCV中，迹是在cv::diag()的基础上实现的，因此输入的矩阵不需要是方阵。支持多通道矩阵，但是迹被计算为标量，因此标量的每个分量是相应通道上的和，最多有四个通道。

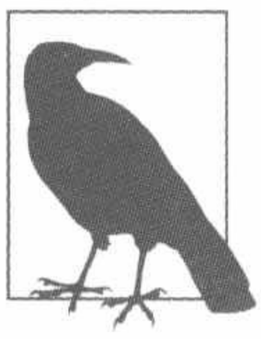
cv::transform()

```
void cv::transform(
  cv::InputArray  src,
  cv::OutputArray dst,
  cv::InputArray  mtx
);
```

$$dst_{c,i,j} = \sum_{c'} mtx_{c,c'} src_{c'i,j}$$

函数cv::transform()可用于计算任意线性图像变换。它将多通道输入矩阵src视为向量的集合，你可以将其视为“通道空间”，然后将这些向量乘以“小”矩阵mtx，以实现此通道空间中的转换。

mtx矩阵的行数必须与src的通道数相同，或者为通道数加一。在第二种情况下，src中的通道空间向量将自动扩展1维，并将扩展元素赋值为1.0。

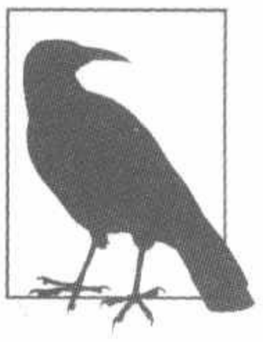


这种转换的确切含义取决于你使用不同通道的作用。如果你将它们用作颜色通道，则可以将此转换视为线性颜色空间转换。RGB和YUV颜色空间之间的转换就是这样一个变换的例子。如果使用通道来表示点的x,y或x,y,z坐标，则这些转换可以被认为是这些点的旋转（或其他几何变换）。

cv::transpose()

```
void cv::transpose(
  cv::InputArray  src,                    // Input array, 2-dimensional, n-by-m
  cv::OutputArray dst,                    // Result array, 2-dimensional, m-by-n
);
```

cv::transpose()将src中每一个元素复制到dst中行号与列数相调换的位置上。这个函数支持多通道矩阵；然而，如果你用多通道表示复数，那么记住cv::transpose()不执行复数共轭。



使用矩阵成员函数cv::Mat::t()可以实现相同的结果。成员函数的优点是可以在矩阵表达式中使用，例如：

```
A = B + B.t();
```

小结

在本章中，我们研究了利用OpenCV最重要的矩阵结构cv::Mat可以完成的大量基本操作，包括矩阵、图像和多维矩阵等。我们看到库提供了一些从非常简单的代数操作到比较复杂的特征操作的函数。其中一些操作旨在帮助我们将矩阵作为图像进行操作，而其他操作在矩阵表示其他数据类型时有用。在接下来的章节中，我们将讨论更复杂的算法从而实现有意义的计算机视觉算法。相对于这些算法以及你要编写的许多算法，本章中的操作将构成你要做的任何事情的基本构建模块。

练习

在下面的练习中，有关本章所述函数的详细信息，可能需要参考参考手册。

1. 这个练习将使你熟悉许多函数使用矩阵类型的概念。创建一个三通道二维矩阵，字节类型，尺寸为100×100，并设置所有数值为0。
   a. 在矩阵中使用void cv::circle(InputOutputArray img, cv::point center, intradius, cv::Scalar color, int thickness=1, int line_type=8, int shift=0)画一个圆。
   b. 使用第2章所讲的方法来显示这幅图像。
2. 创建一个有三个通道的二维字节类型矩阵，尺寸为100×100，并将所有值赋为0。使用cv::Mat元素访问函数修改像素。以(20, 5)与(40, 20)为顶点画一个绿色的长方形。
3. 创建一个尺寸为100×100的三通道RGB图像。将它的元素全部置0。使用指针算法以(20, 5)与(40, 20)为顶点绘制一个绿色平面。
4. 使用感兴趣区域（ROI）。创建一个210×210的单通道字节图像并将其归0。在图像中使用ROI和cv::Mat::setTo()建立一个增长如金字塔状的矩阵。也就是外部边界为0，下一个内部边界应该为20，再下一个内部边界为40，以此类推，直到最后内部值为200；所有的边界应该为10个像素的宽度。显示这个图像。
5. 为一个图像使用多个图像头。读取一个尺寸至少为100×100的图像。创建两个额外的宽度=20，高度=30的ROI的图像头。它们的起点分别为(5, 10)和(50, 60)。传递这两个新的图像头给cv::bitwise_not()。显示读取的图像，在这个大图像中应该有两个矩形，矩形内的值是原始值的求反值。
6. 使用cv::compare()创建一个掩码。加载一个真实的图像。使用cv::split()将图像分成红绿蓝图像。
   a. 找到并显示绿图。

b. 克隆这个绿图两次（分别命名为clone1和clone2）。

c. 求出这个绿平面的最小值和最大值。

d. 将clone1的所有元素赋值为thresh = (unsigned char)((最大值－最小值)/2.0)。

e. 将clone2所有元素赋值为0，然后用cv::compare (green_image, clone1, clone2, cv::CMP_GE)。现在，clone2将是一个标识绿图中的值超过thresh的掩码图像。

f. 最后，使用cv::subtract (green_image,thresh/2, green_image, clone2)并显示结果。

第6章

绘图和注释

绘图

我们常常会想要画一幅图片，或者在某个其他地方得来的图片上画些东西。为了实现这个目标，OpenCV提供了一大批可以供我们画直线、矩形、圆等图形的函数。

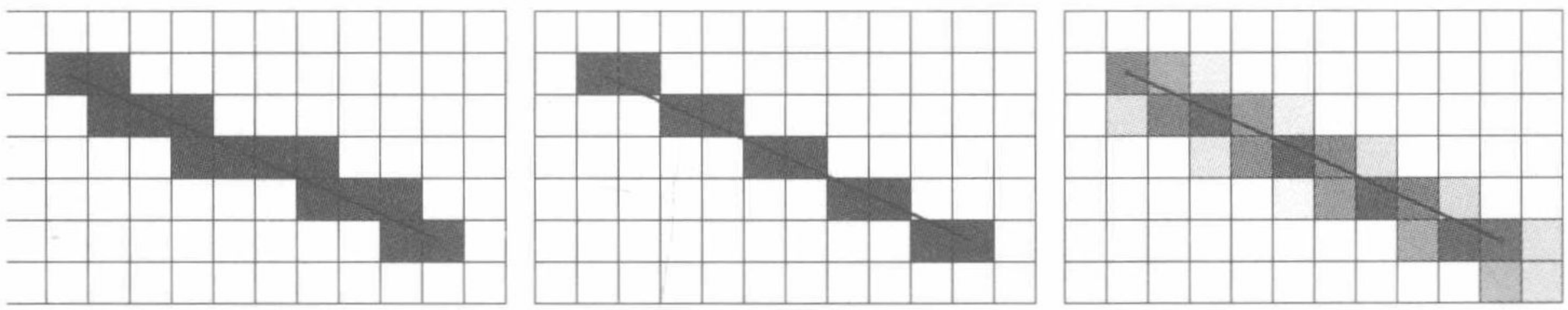
OpenCV的绘图函数可以在任意深度的图像上工作，但在大多数情况下，它们只对图像的前三个通道有影响，如果是单通道图像，则默认只影响第一个通道。大多数绘图函数都支持操作对象的颜色、宽度、线型（与直线平滑度有关）和亚像素对齐等参数。

想指定颜色的时候，通常的做法是使用cv::Scalar对象，即使大多数时候只用到最前面的三个值。（有些时候使用cv::Scalar里面第四个值来表示透明通道是很方便的，但是绘图函数目前还不支持透明混合处理）按照惯例，OpenCV在对多通道图像进行色彩渲染时使用了BGR的顺序[注1]（这就是在屏幕上进行绘图显示的函数imshow()所使用的顺序）。当然，也可以不按照这个惯例，特别是当你在使用含有其他OpenCV库文件的数据的时候这样做可能不很理想。无论如何，核心库函数都无法知道你对一个通道所赋予的任何意义。

注1：使用宏CV_RGB (r, g, b)会生成一个cv::Scalar对象s，它的值s.val[] = { b, g, r, 0 }。由于一般的OpenCV函数只能通过参数的顺序来判断哪个是红色、绿色或是蓝色，而对于图像数据，正如文中所说，这个顺序按照惯例是BGR，因此会有这样的结果。

艺术线条和填充多边形

画直线或者其他图形（虚线、圆、矩形等）的函数通常可以接受宽度（thickness）和线型（lineType）的参数。这两个参数都是整型的，但是第二个参数的值只能是4，8或者cv::LINE_AA。thickness参数是指线宽的像素数 对于圆、矩形和其他封闭图形，thickness参数可以被设置为cv::FILLED（相当于-1）。这样的话，整个图形就会被用边界一样的颜色填充满。lineType参数表示直线是“4邻域连接”或者“8邻域连接”或者是平滑处理。图6-1中的前两个例子，使用了Bresenham算法，而最后一个平滑处理则使用了高斯滤波器。宽直线总是被画成圆头的。



(A) 4邻域连接Bresenham算法　(B) 8邻域连接Bresenham算法　(C) 使用高斯滤波平滑处理的直线

图6-1. 同一条直线分别使用4邻域连接(A)8邻域连接(B)和平滑处理(C)的线型

在表6-1中所列出的绘图算法中，端点（直线）、中心点（圆）和角点（矩形）等通常都规定为整型值。然而这些算法都支持使用shift参数来进行亚像素对齐。当shift参数有效时，这个值将被用来指定整数参数的小数位数。例如，你要画一个圆，设置圆心为(5, 5)，但是又把shift参数设为1，这样会得到一个圆心在(2.5, 2.5)的圆[译注1]。这个对齐效果通常会很细微，取决于使用的线型。在平滑处理的直线中这个效果会比较明显。

表6-1：绘图函数

函数	描述
cv::circle()	画一个简单圆
cv::clipLine()	判断一条直线是否在给定的矩形内
cv::ellipse()	画一个椭圆（可以倾斜，或者只有部分圆弧）
cv::ellipse2Poly()	计算一个近似椭圆的多边形
cv::fillConvexPoly()	画一个填充的简单多边形
cv::fillPoly()	画一个填充的任意多边形
cv::line()	画一个简单直线
cv::rectangle()	画一个简单矩形
cv::polyLines()	画多重折线

译注1：这里也许有些难以理解，实际上在OpenCV的源码中，此处是用移位操作实现的，简单理解就是值会被除以2的shift次方。

接下来详细描述表6-1中的每个函数。

cv::circle()

```
void circle(
  cv::Mat&          img,           // Image to be drawn on
  cv::Point         center,        // Location of circle center
  int               radius,        // Radius of circle
  const cv::Scalar& color,         // Color, RGB form
  int               thickness = 1, // Thickness of line
  int               lineType  = 8, // Connectedness, 4 or 8
  Int               shift     = 0  // Bits of radius to treat as fraction
);
```

cv::circle()第一个参数就是你绘图的图像img。接下来是圆心center，一个二维坐标点，然后是半径radius。剩下的参数就是标准的color，thickness，lineType和shift。shift参数同时应用在半径和圆心上。

cv::clipLine()

```
bool clipLine(                     // True if any part of line in 'imgRect'
  cv::Rect        imgRect,         // Rectangle to clip to
  cv::Point&      pt1,             // First endpoint of line, overwritten
  cv::Point&      pt2              // Second endpoint of line, overwritten
);

bool clipLine(                     // True if any part of line in image size
  cv::Size        imgSize,         // Size of image, implies rectangle at 0,0
  cv::Point&      pt1,             // First endpoint of line, overwritten
  cv::Point&      pt2              // Second endpoint of line, overwritten
);
```

这个函数用来判断点pt1到点pt2的直线是否在一个矩形范围内。第一种函数的形式使用了一个cv::Rect，直线就是和这个矩形进行比较。只有当直线完全在指定的矩形范围之外时，函数cv::clipLine()才会返回False。第二种形式只有cv::Size参数和第一种形式不同。调用第二种形式相当于调用第一种形式时矩形范围是从(0，0)开始的。

cv::ellipse()

```
bool ellipse(
  cv::Mat&          img,           // Image to be drawn on
  cv::Point         center,        // Location of ellipse center
  cv::Size          axes,          // Length of major and minor axes
  double            angle,         // Tilt angle of major axis
  double            startAngle,    // Start angle for arc drawing
  double            endAngle,      // End angle for arc drawing
  const cv::Scalar& color,         // Color, BGR form
  int               thickness = 1, // Thickness of line
  int               lineType = 8,  // Connectedness, 4 or 8
```

```
  int                    shift = 0      // Bits of radius to treat as fraction
);

bool ellipse(
  cv::Mat&               img,           // Image to be drawn on
  const cv::RotatedRect& rect,          // Rotated rectangle bounds ellipse
  const cv::Scalar&      color,         // Color, BGR form
  Int                    thickness = 1, // Thickness of line
  Int                    lineType = 8,  // Connectedness, 4 or 8
  Int                    shift = 0      // Bits of radius to treat as fraction
);
```

cv::ellipse()函数和cv::circle()函数非常相似，主要的区别在于axes参数，这是一个cv::Size类型的值。其中的height和width参数分别表示椭圆长轴和短轴的长度。angle参数是主轴的角度（单位为度），计算方法是从水平方向（*x*轴正方向）开始逆时针旋转。类似的，startAngle和endAngle参数（单位也为度）表示了圆弧的起始和中止角度。因此，如果要得到一个完整的椭圆，这两个值要分别设置为0和360。另一种画指定椭圆的方法是使用一个边界框。这样的话，cv::RotatedRect类型的边界框就同时指定了椭圆的大小和朝向。这两种指定椭圆的方法如图6-2所示。

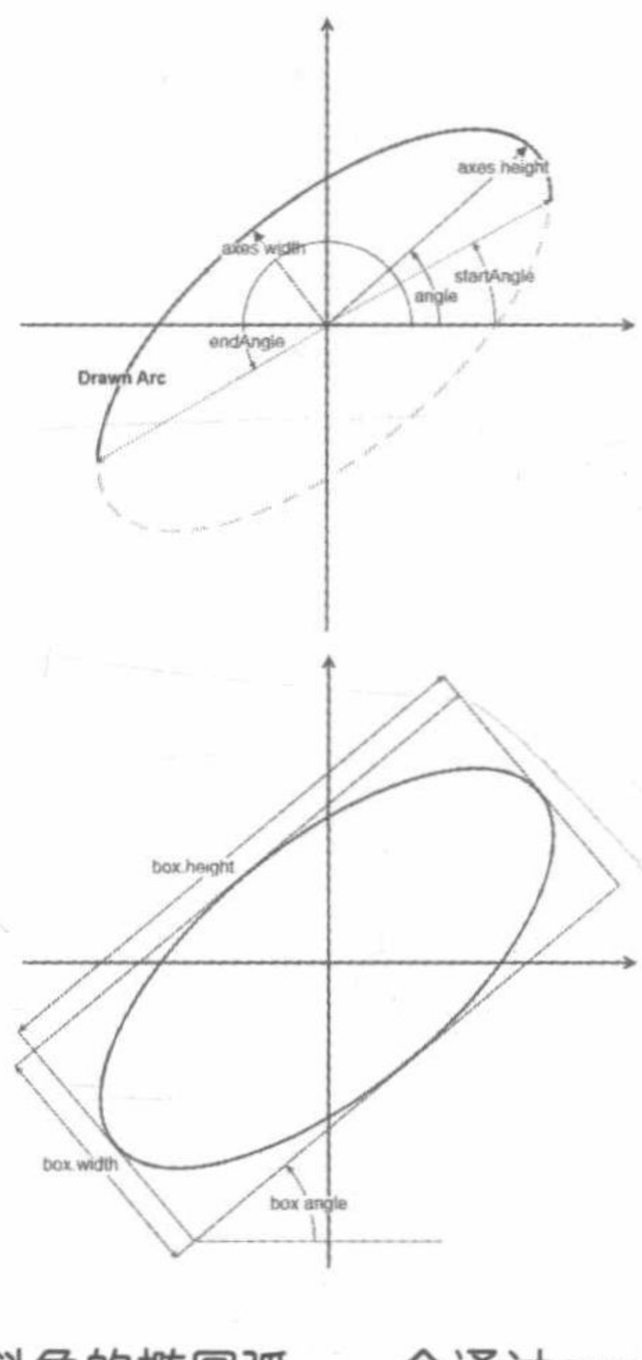


图6-2. 一个通过指定长短轴和倾斜角的椭圆弧；一个通过cv::RotatedRect指定的相似的椭圆

cv::ellipse2Poly()

```
void ellipse2Poly(
  cv::Point              center,        // Location of ellipse center
  cv::Size               axes,          // Length of major and minor axes
  double                 angle,         // Tilt angle of major axis
```

```
    double                  startAngle,     // Start angle for arc drawing
    double                  endAngle,       // End angle for arc drawing
    int                     delta,          // Angle between sequential vertices
    vector<cv::Point>&      pts             // Result, STL-vector of points
);
```

cv::ellipse2Poly()函数是在cv::ellipse()函数中内部使用的，用来计算椭圆弧，但是也可以单独调用。给定一个椭圆弧参数（和cv::ellipse()定义中相同的center, axes, angle, startAngle和endAngle）和用来指定下一个采样点的角度delta参数之后，cv::ellipse2Poly()函数就可以计算出一系列的点，用以这些点作为顶点的多边形来近似指定的椭圆弧。计算出的点使用vector<> pts返回。

cv::fillConvexPoly()

```
void fillConvexPoly(
  cv::Mat&                Img,            // Image to be drawn on
  const cv::Point*        pts,            // C-style array of points
  int                     npts,           // Number of points in 'pts'
  const cv::Scalar&       color,          // Color, BGR form
  int                     lineType = 8,   // Connectedness, 4 or 8
  int                     shift    = 0    // Bits of radius to treat as fraction
);
```

这个函数用来绘制一个填充的多边形。这个函数比cv::fillPoly()（接下来会讲到）速度快很多，因为它使用了更简单的算法。需要注意的是cv::fillConvexPoly()使用的算法在多边形有自交点[注2]时不能正确工作。pts中的点将被按顺序用直线段连接起来，第一个点和最后一个点之间也会连接起来（也就是说多边形被认为是封闭的）。

cv::fillPoly()

```
void fillPoly(
  cv::Mat&                img,            // Image to be drawn on
  const cv::Point*        pts,            // C-style array of arrays of points
  int                     npts,           // Number of points in 'pts[i]'
  int                     ncontours,      // Number of arrays in 'pts'
  const cv::Scalar&       color,          // Color, BGR form
  int                     lineType = 8,   // Connectedness, 4 or 8
  int                     shift = 0,      // Bits of radius to treat as fraction
  cv::Point               offset = Point()    // Uniform offset applied to all points
);
```

这个函数可以绘制任意数量的填充的多边形。与cv::fillConvexPoly()函数不同的是，这个函数可以处理自交点的情况。ncontours参数表示一共有多少个不同的多边形轮

注2：cv::fillConvexPoly()使用的算法实际上比这里讲述的更加通用。它将正确绘制外轮廓与每条水平线最多相交两次的任意多边形（并且允许多边形的顶部或底部相对于水平线平行）。这样的多边形被认为是“相对于水平线单调的”。

廓，npts参数是一个C风格的数组，表示在每一个多边形轮廓里有多少个点（也就是说，npts[i]表示在多边形i中有多少个点）。pts参数是一个C风格的数组的数组，包含了所有多边形中的所有点。（也就是说，pts[i][j]表示第i个多边形中的第j个点）。cv::fillPoly()还有一个额外的参数offset，这是在绘制多边形时加在所有顶点坐标上的像素偏移。多边形都被认为是封闭的（也就是说pts[i][]的最后一个点会和第一个点连起来）。当一个多边形的轮廓和任意水平线至多相交两次时，这个算法就可以正确画出这个多边形（包含多边形的上下两边是水平线的情况）。这种多边形称为“水平单调多边形”。

cv::line()

```
void line(
  cv::Mat&          img,                // Image to be drawn on
  cv::Point         pt1,                // First endpoint of line
  cv::Point         pt2                 // Second endpoint of line
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0        // Bits of radius to treat as fraction
);
```

cv::line()函数在图像img上绘制一条从pt1到pt2的直线。直线自动被图像边缘截断。

cv::rectangle()

```
void rectangle(
  cv::Mat&          img,                // Image to be drawn on
  cv::Point         pt1,                // First corner of rectangle
  cv::Point         pt2                 // Opposite corner of rectangle
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0        // Bits of radius to treat as fraction
);

void rectangle(
  cv::Mat&          img,                // Image to be drawn on
  cv::Rect          r,                  // Rectangle to draw
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0        // Bits of radius to treat as fraction
);
```

cv::rectangle()函数在图像img上绘制一个从角点pt1到pt2的矩形。这个函数的另一种形式可以使用一个cv::Rect类型的参数r指定矩形的坐标和大小。

cv::polyLines()

```
void polyLines(
cv::Mat&          img,                //  Image to be drawn on
const cv::Point*  pts,                //  C-style array of arrays of points
```

```
Int                 npts,                   //  Number of points in 'pts[i]'
Int                 ncontours,              //  Number of arrays in 'pts'
bool                isClosed,               //  If true, connect last and first pts
const cv::Scalar&   color,                  //  Color, BGR form
Int                 lineType = 8,           //  Connectedness, 4 or 8
Int                 shift = 0               //  Bits of radius to treat as fraction
);
```

这个函数可以绘制任意数量的未填充的多边形。它可以处理包括有自交点的多边形在内的普通多边形。ncontours参数表示一共有多少个不同的多边形轮廓，npts参数是一个C风格的数组，表示在每一个多边形轮廓里有多少个点（也就是说，npts[i]表示在多边形i中有多少个点）。pts参数是一个C风格的数组的数组，包含了所有多边形中的所有点。（也就是说，pts[i][j]表示第i个多边形中的第j个点）多边形被认为是不封闭的。如果参数isClosed为真，那么pts[i][]的最后一个点将和第一个点连起来。否则，轮廓将被认为是不封闭的，在npts[i]个点之间只有npts[i]-1条直线段。

cv::LineIterator

```
LineIterator::LineIterator(
cv::Mat&            img,                    // Image to be drawn on
cv::Point           pt1,                    // First endpoint of line
cv::Point           pt2                     // Second endpoint of line
int                 lineType = 8,           // Connectedness, 4 or 8
bool                leftToRight = false     // If true, always start steps on the
left
);
```

cv::LineIterator对象是一个用来顺序得到网格线每一个像素的迭代器。直线迭代器是OpenCV中第一个仿函数的例子。我们将在下一章看到更多这种“做这样事情的对象”。直线迭代器的构造函数需要直线的两个端点和直线的线型以及一个额外的用来表示处理方向的布尔值。

一旦初始化之后，直线的像素数量就保存在一个整型成员变量cv::LineIterator::count中。重载的取值算子cv::LineIterator::operator*()返回一个指向“当前”像素的uchar*类型的指针。当前像素从直线的一个端点开始，通过重载增量算子cv::LineIterator::operator++()移动当前像素。实际的遍历过程通过前面提到的Bresenham算法完成。

cv::LineIterator的目的是让你能够对直线的每一个像素进行某些特殊处理。如果要创建一些特殊效果，例如将一个像素的颜色从黑色变成白色或者从白色变成黑色（比如对二值图像进行异或操作），使用cv::LineIterator就特别方便。

当你访问一个独立的“像素”时，需要记住这个像素可能有一个或者多个通道，并且它可能是任意深度类型的图像。取值算子的返回值总是uchar*类型的，因此你需要将这个

指针转换成正确的类型。举个例子，如果处理的是一个三通道的32位浮点数的图像，你的迭代器叫做iter，然后你就可以通过(Vec3f*)*iter这样来转换返回的（指针）值。



重载的取值算子cv::Linelterator::operator*()和你熟悉的（例如STL库中的）算子有一些细微的差别。这个区别在于迭代器的返回值本身是一个指针，因此迭代器就表现得不像一个指针而像一个指针的指针。

字体和文字

绘图的一种额外形式就是画文字。当然，文字有它自己的复杂性，但是OpenCV总是更加致力于提供一种“简单粗暴”的解决方案来处理简单事物，而不是提供一个鲁棒的复杂的解决方案（对其他库提供兼容性无论如何都会变得冗余）。表6-2列出了OpenCV中两种文字的绘制函数。

表6-2：文字绘制函数

函数名称	描述
cv::putText()	在图像中绘制指定文字
cv::getTextSize()	获取一个文字的宽度和高度

cv::putText()

```
void cv::putText(
  cv::Mat&      img,                    // Image to be drawn on
  const string& text,                   // write this (often from cv::format)
  cv::Point     origin,                 // Upper-left corner of text box
  int           fontFace,               // Font (e.g., cv::FONT_HERSHEY_PLAIN)
  double        fontScale,              // Size (a multiplier, not "points"!)
  cv: :Scalar   color,                  // Color, RGB form
  int           thickness = 1,          // Thickness of line
  int           lineType = 8,           // Connectedness, 4 or 8
  bool          bottomLeftOrigin = false // true='origin at lower left'
);
```

这个函数是OpenCV的一个主要文字绘制方法；它可以简单地在图像上绘制一些文字。由text指定的文字将在以左上角为原点的文字框中以color指定的颜色绘制出来，除非bottomLeftOrigin标志被设为真，这种情况下文字框将以左下角为原点。使用的字体由fontFace参数决定，可以是表6-3列出的任何一个值。

表6-3：可以使用的字体（都是Hershey的变体）

标识符	描述
cv::FONT_HERSHEY_SIMPLEX	普通大小无衬线字体
cv::FONT_HERSHEY_PLAIN	小号无衬线字体
cv::FONT_HERSHEY_DUPLEX	普通大小无衬线字体；比cv::FONT_HERSHEY_SIMPLEX更复杂
cv::FONT_HERSHEY_COMPLEX	普通大小无衬线字体；比cv::FONT_HERSHEY_DUPLEX更复杂
cv::FONT_HERSHEY_TRIPLEX	普通大小无衬线字体；比cv::FONT_HERSHEY_COMPLEX更复杂
cv::FONT_HERSHEY_COMPLEX_SMALL	小号版本的cv:: FONT_HERSHEY_COMPLEX
cv::FONT_HERSHEY_SCRIPT_SIMPLEX	手写体
cv::FONT_HERSHEY_SCRIPT_COMPLEX	比cv::FONT_HERSHEY_SCRIPT_SIMPLEX更复杂的变体

表6-3中列出来的任何一种字体都可以和cv::FONT_HERSHEY_ITALIC组合使用（通过或操作）来得到斜体。每种字体都有一个“自然”大小。当fontScale不是1.0时，在文字绘制之前字体大小将由这个数缩放。图6-3是每种字体的示例。



图6-3. 表6-3中的8种字体，每一行的原点在垂直方向上间隔30像素

cv::getTextSize()

```
cv::Size cv::getTextSize(
  const string& text,
  cv::Point     origin,
  int           fontFace,
  double        fontScale,
```

```
    int             thickness,
    int*            baseLine
);
```

cv::getTextSize()函数回答了如果把文字绘制出来将有多大的问题（使用一组参数）而不用实际将文字绘制到图上。cv::getTextSize()唯一的新的参数就是baseLine，这实际上是一个输出参数。baseLine是和文字最低点相关的文字基线的y坐标值[注3]。

小结

在这章里，我们学习了一些可以用来绘制和注解图像的新函数。这些函数操作的都是之前章节我们使用过的cv::Mat图像类型。大多数函数都有相似的接口，可以让我们绘制不同宽度、颜色的直线和曲线。除了直线和曲线之外，我们还看到了OpenCV怎样在图像上绘制文字。所有这些函数在实际使用中都非常有用，比如在调试代码时使用，也可以在输入图像上显示计算结果。

练习

对于下面这些习题，请修改示例2-1中的代码来显示图像或者修改示例2-3中的代码，使其从视频或者摄像头中载入图像并显示。

1. 绘图练习：载入或创建一幅彩色图像并显示。把每一种OpenCV能够绘制的形状和线条都画一个示例。
2. 灰度练习：载入并显示一幅彩色图像。
   a. 将它转为三通道的灰度图（它依然是一幅BGR图像，但看起来是灰色的）。
   b. 在这幅图像上绘制彩色文字。
3. 动态文字练习：从一个视频文件或者摄像头中载入并显示一段视频。在图像的任意位置显示帧率（FPS）。
4. 编写一个绘图程序，载入或创建一个图像并显示。
   a. 允许用户绘制一个基面。
   b. 让这个面的组件可编辑（需要维护一个包含绘制对象的列表，当它们改变的时候，你可能需要擦除并重新绘制）。
5. 使用cv::LineIterator来计算一幅图像（例如300*300像素）中不同直线段的像素数。

注3： 例如字符*a*和*b*，基线是指它们底部对齐的直线。对于字符*y*和*g*，则是挂在基线下*f*方。

a. 在什么角度下，4邻域连接和8邻域连接直线能够得到相同的像素数？

b. 对于和上一题中不同的角度，哪一种连接有更多的像素，4领域还是8邻域？

c. 在4邻域连接和8邻域连接两种情况下，对于一条给定的直线段，直线长度和使用迭代器计算出的像素数之间的区别分别是什么？哪种连接下的像素数更接近真实的直线长度？

第7章

OpenCV中的函数子

操作对象

随着OpenCV库的发展，封装复杂功能的新对象越来越普遍地引进，这些功能不是单一的函数能够完成的，但是将它们实现为一组函数又会导致整个库的函数变得太杂乱。[注1]

因此，常常使用一个相关联的新的对象类型来表示一个新功能，这个对象类型可以看做是一个完成这个功能的“机器”。这些机器中的大多数都有一个重载的operator()，使其成为函数对象或者函数子。如果这种编程习惯不太熟悉，一个重要思想就是，和通常的函数不一样，函数对象是已经创建好的，并且能够在其内部维护状态信息。因此，它们能够和任何需要的数据或者配置一起被创建，并且它们被“要求”通过常规的成员函数或者自身作为函数被调用来提供服务（通常通过重载的operator()[注2]）。

主成分分析（cv::PCA）

主成分分析，如图7-1所示，是一个分析多维分布并且从中提取出带有最多信息量的维度子集的方法。在OpenCV库里有一个互相矛盾的习惯：使用重载的operator()来载入配置和使用一个成员来提供对象的基本服务这个习惯通常来说不是很标准，但是在OpenCV库中很常见。由PCA计算出的维度并不一定是分布规律最初指定的基础维度。

注1：　我们在第6章遇到过这些对象中的一个cv::LineIterator。

注2：　这里的“通常”是指“通常在使用函数对象时”，而不是指“通常使用OpenCV库时”。

事实上，PCA的一个重要功能就是能够按照轴的重要性排序来生成新的基础维度[注3]。这些基向量可以被证明是整体分布协方差矩阵的特征向量，相应的特征值就是那个维度的分布范围。

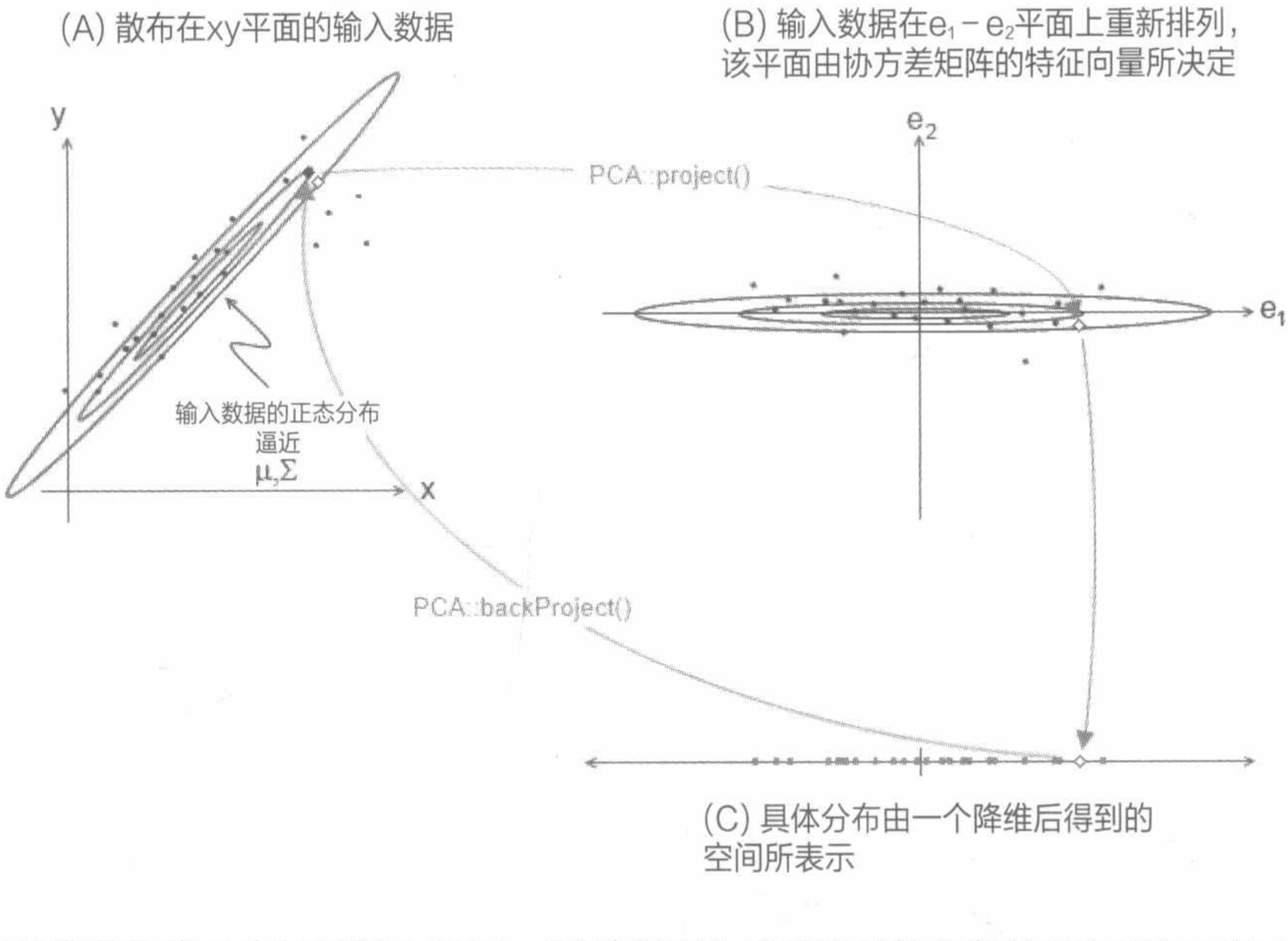


图7-1. (A)使用正态分布逼近输入数据。(B)映射到上述逼近的协方差的特征向量所构成的空间中。(C)过KLT映射到只由最“有用”的特征向量构成的空间，加上一个新的数据点（白色方块）通过cv::PCA::project()映射到降维空间中，同样的数据点通过cv::PCA::backProject()映射回原空间中（黑色方块）

现在我们来解释为什么PCA通过这些函数对象中的一个来操作。一旦给定一个分布，PCA对象可以计算并且记住这个新的基。这个新的基最大的优点在于大的特征向量对应的基向量带有分布中对象的最多的信息。因此，在不损失太多精度的前提下，我们可以扔掉较少信息量的维度。在现代计算机视觉中，机器学习成为日益增长的算法种类的发展方向。因此，例如PCA和SVD这样的组件功能越来越被看做“砖块”。这个降维的方法称为KLT变换。[注4]一旦载入一个样本分布，主成分就将被计算，你可能想用这些信息来做各种事情，例如将KLT变换应用在新向量上。当你把PCA功能实现为一个函数对象时，它能够“记住”你给的分布中它需要知道的东西，然后在需要时使用那些信息来提供变换新向量的“服务”。

注3：你可能会想：“这看起来不就是机器学习吗？这一章到底要讲什么？”这是个不错的问题。

注4：KLT表示“Karhunen-Loeve变换”，因此KLT变换的说法是有一点用词错误的，然而这两种说法也很常见。

cv::PCA::PCA()

```
PCA::PCA();
PCA::PCA(
  cv::InputArray data,                    // Data, as rows or cols in 2d array
  cv::InputArray mean,                    // average, if known, 1-by-n or n-by-1
  int            flags,                   // Are vectors rows or cols of 'data'
  int            maxComponents = 0        // Max dimensions to retain
);
```

PCA对象有一个默认的构造函数cv::PCA()，它简单地创建PCA对象并且初始化空结构。第二种形式执行默认的构造操作，然后立即将它的参数传递给PCA::operator()（接下来将会介绍）。

cv::PCA::operator()()

```
PCA::operator()(
  cv::InputArray data,                    // Data, as rows or cols in 2d array
  cv::InputArray mean,                    // average, if known, 1-by-n or n-by-1
  int            flags,                   // Are vectors rows or cols of 'data'
  int            maxComponents = 0        // Max dimensions to retain
);
```

为PCA重载的operator()生成PCA对象内部分布的模型。数据参数是一个包含所有构成分布的样本的矩阵。可选的mean参数是第二个包含每一维的平均值的矩阵（可以是$n \times 1$或者$1 \times n$）。数据能够被排成一个$n \times D$（n行采样，每行D维）或者$D \times n$（n列采样，每列d维）的矩阵。flags参数现在只被用来指定data和mean中数据的排布。具体而言，flags能够被设为cv::PCA_DATA_AS_ROW或者cv: :PCA_DATA_AS_COL来分别表示data是$n \times D$，mean是$n \times 1$或者data是$D \times n$，mean是$1 \times n$。最后一个参数maxComponents指定PCA最大保留的成分（维度）数。默认情况下，所有成分都将保留。

任何对cv::PCA::operator()的后续调用都将重写特征向量和特征值的内部表示，因此需要时你可以重用一个PCA对象，也就是说，对于每个新的分布，如果不再需要之前分布的信息时，可以直接重复使用当前对象而不用重新申请新的对象。

cv::PCA::project()

```
cv::Mat PCA::project(                     // Return results, as a 2d matrix
  cv::InputArray  vec                     // points to project, rows or cols, 2d
) const;

void PCA::project(
  cv::InputArray  vec                     // points to project, rows or cols, 2d
  cv::OutputArray result                  // Result of projection, reduced space
) const;
```

一旦使用cv::PCA::operator()()载入一个参考分布，你就可以开始让PCA对象

做一些有用的事情，例如计算一些向量集到主成分分析的基向量上的KLT映射。cv::PCA::project()函数有两种形式；第一种返回了一个包含映射结果的矩阵，第二种直接将结果写入你提供的矩阵。第一种形式的好处是你可以把它用在矩阵表达式中。

vec参数包含输入向量。当分布第一次被分析时，vec参数需要和传入PCA的数据矩阵有相同的维数和方向（也就是说，当你调用cv::PCA::operator()()时，数据是列向量，那么vec应该也是列向量）。

返回的矩阵将和相同方向的vec有相同数量的对象，但是每个对象的维度将是PCA对象第一次用cv::PCA::operator()()配置时传递给maxComponents的值。

cv::PCA::backProject()

```
cv::Mat PCA::backProject(                    // Return results, as a 2d matrix
  cv::InputArray  vec                        // Result of projection, reduced space
) const;

void PCA::backProject(
  cv::InputArray  vec                        // Result of projection, reduced space
  cv::OutputArray result                     // "reconstructed" vectors, full dimension
) const;
```

cv::PCA::backProject()函数执行和cv::PCA::project()相反的操作，对于输入和输出矩阵有相似的限制。vec参数包含来自映射后空间的输入向量。同样的，当分布第一次被分析时，vec参数的维数将会是你设置PCA对象时指定的maxComponents值，方向和传入PCA的数据矩阵相同（也就是说，当你调用cv::PCA::operator()时，数据是列向量，那么vec应该也是列向量）。

返回的矩阵将和相同方向的vec有相同数量的对象，但是每个对象的维度将会是PCA对象第一次用cv::PCA::operator()()配置时原始数据的维度。



如果在最开始配置PCA对象时，没有保留所有维度，那么逆映射结果向量（一些向量$\vec{x}$在原始数据空间中的自投影）将和原始向量不相等$\vec{x}$。当然，即使是保留的成分数比$\vec{x}$原始数据的维度小很多，最后结果的差别也会很少，这就是首先使用PCA的意义所在。

奇异值分解cv::SVD

cv::SVD类和cv::PCA很相似，它们都是相同类型的函数对象。但是它们的用途却非常不同。奇其他异值分解本质上是一个解决非方阵的，病态的（不适定的）或者你在解决欠定线性系统中遇到的不良矩阵的工具。

在数学上，奇异值分解（SVD）对于一个$m \times n$的矩阵A的分解形式为：

$$A = U \cdot W \cdot V T$$

其中W是一个对角矩阵，U和V分别是$m \times n$和$n \times n$的（单位）矩阵。当然，矩阵W也是一个$m \times n$的矩阵，因此这里“对角”的意思是任何行列下标值不相等的元素都为0。

cv::SVD()

```
SVD::SVD();
SVD::SVD(
  cv::InputArray A,                    // Linear system, array to be decomposed
  int            flags = 0             // what to construct, can A can scratch
);
```

SVD对象有一个默认的构造函数cv::SVD()，它简单创建SVD对象并且初始化空结构。第二种形式执行默认的构造操作，然后立即将它的参数传递给cv::SVD::operator()()，接下来会介绍。

cv::SVD::operator()()

```
SVD::& SVD::operator() (
  cv::InputArray A,                    // Linear system, array to be decomposed
  int            flags = 0             // what to construct, can A be scratch
);
```

cv::SVD::operator()()算子将需要分解的矩阵传递给cv::SVD对象。正如前面所描述的，矩阵A将被分解为一个矩阵U、一个矩阵V（实际上是V的转置，我们称之为Vt）和一组奇异值（矩阵W的对角元素）。

flags的值可以是cv::SVD::MODIFY_A，cv::SVD::NO_UV或者cv::SVD::FULL_UV。后面两个是互斥的，但是任意一个都可以与第一个组合。cv::SVD::MODIFY_A标志表示允许在计算时改变矩阵A的值。这样可以稍微提高运算速度，并且可以节约内存。当输入矩阵很大时，这就显得更加重要了。cv::SVD::NO_UV标志告诉cv::SVD不用显式的计算出矩阵U和Vt，而cv::SVD::FULL_UV标志表示不仅要计算出U和Vt，而且要让它们是全尺寸的正交方阵。

cv::SVD::compute()

```
void SVD::compute(
  cv::InputArray  A,                  // Linear system, array to be decomposed
  cv::OutputArray W,                  // Output array 'W', singular values
  cv::OutputArray U,                  // Output array 'U', left singular vectors
  cv::OutputArray Vt,                 // Output array 'Vt', right singular vectors
  int             flags = 0          // what to construct, and if A can be scratch
);
```

这个函数是使用cv::SVD::operator()()之外的另一种分解矩阵*A*的选择。主要的区别在于矩阵*W*，*U*和*Vt*存储在用户提供的矩阵里，而不是函数内部。支持的flags和cv::SVD::operator()()所支持的完全一样。

cv::SVD::solveZ()

```
void SVD::solveZ(
  cv::InputArray  A,                    // Linear system, array to be decomposed
  cv::OutputArray z                     // One possible solution (unit length)
);
```

$$\vec{z} = \text{argmin}_{\vec{x}:\|\vec{x}\|=1} \| A \cdot \vec{x} \|$$

给定一个欠定的（奇异）线性系统，cv::SVD::solveZ()将会（尝试）找出的一个单位长度的解，然后放在矩阵z中。然而，因为线性系统是奇异的，它有可能无解，或者有无穷多解，因此只要存在解，cv::SVD::solveZ()就会找到一个。如果无解，那么返回值将是使得值最小的向量。

cv::SVD::backSubst()

```
void SVD::backSubst(
  cv::InputArray  b,                    // Righthand side of linear system
  cv::OutputArray x                     // Found solution to linear system
);

void SVD::backSubst(
  cv::InputArray  W,                    // Output array 'W', singular values
  cv::InputArray  U,                    // Output array 'U', left singular vectors
  cv::InputArray  Vt,                   // Output array 'Vt', right singular vectors
  cv::InputArray  b,                    // Righthand side of linear system
  cv::OutputArray x                     // Found solution to linear system
);
```

假设矩阵*A*已经预先传递给cv::SVD对象（因此已经分解为*U*，*W*和*Vt*），cv::SVD::backSubst()的第一种形式尝试解如下系统：

$$(UWV^T) \cdot \vec{x} = \vec{b}$$

第二种形式完成同样的功能，但是需要传递矩阵*W*，*U*和*Vt*作为参数。实际计算dst的方法是如下表达式：

$$\vec{x} = V_t^T \cdot diag(W)^{-1} \cdot U^T \cdot \vec{b} \sim A^{-1} \cdot \vec{b}$$

这种方法生成一个超定系统的伪解，使得最小二乘法的误差最小。[注5]当然，这也会正确解出一个适定的线型系统。



在实际操作中，很少直接用到cv::SVD::backSubst()。因为你能够使用cv::solve()传递cv::DECOMP_SVD参数来做相同的事，而这样会更简单。只有在一些很少见的情况下，需要解很多有相同左手侧(x)的系统时，你才会需要直接使用cv::SVD::backSubst()，相反，如果需要解不同右手侧(b)的同一系统很多次，依然更适合使用cv::solve()。

随机数发生器cv::RNG

一个随机数对象（RNG）用来产生随机数的伪随机序列。这样做的好处是你可以方便地得到多重伪随机数流。



在写大型系统时，在不同的代码模块中使用不同的随机数流是个很好的习惯。这样的话，当你移除一个模块时，不会改变其他模块中的随机数流。

一旦随机数发生器创建，就会开始按需提供产生随机数的“服务”，无论是平均分布还是正态分布。发生器使用Multiply with Carry（MWC）算法[Goresky03]产生平均分布，使用Ziggurat算法[Marsaglia00]产生正态分布。

cv::theRNG()

```
cv::RNG& theRNG( void );                    // Return a random number generator
```

cv::theRNG()函数为调用它的线程返回一个默认的随机数生成器。OpenCV自动为每一个执行中的线程创建一个cv::RNG的实例。这和函数cv::randu()或者cv::randn()使用的是同样的随机数发生器。如果你只想要一个数或者只初始化一个数组，这些函数就很方便。然后，如果你有一个循环需要产生大量随机数，最好使用一个随机数生成器，在这个例子里，使用的是默认生成器，但是你也可以使用自己的来代替，通过使用RNG::operator T()来得到自己的随机数（比直接使用operator更多）。

注5：对象$diag(W)^{-1}$是一个对角元素由得到的，其中λ_I是W的对角元素，满足条件$\lambda_i \geq \varepsilon$。$\varepsilon$值是奇异阈值，一个很小的和$W$的对角元素和成比例的数 W (i.e., $\varepsilon 0 \sum_i \lambda_i$)。

cv::RNG()

```
cv::RNG::RNG( void );
cv::RNG::RNG( uint64 state );              // create using the seed 'state'
```

可以使用默认的构造函数来创建一个RNG对象，或者传递一个64位的无符号整型数，这个数将用来作为随机数序列的种子。如果调用默认的构造函数（或者第二个参数为0），这个生成器将使用一个定值来初始化。[注6]

cv::RNG::operator T()，其中T是数据类型

```
cv::RNG::operator uchar();
cv::RNG::operator schar();
cv::RNG::operator ushort();
cv::RNG::operator short int();
cv::RNG::operator int();
cv::RNG::operator unsigned();
cv::RNG::operator float();
cv::RNG::operator double();
```

cv::RNG::operator T()实际上是从特定类型cv::RNG中返回一个新的随机数的一组不同方法。这组方法中的每一个都是重载的类型转换操作符，因此你把RNG对象转换成任何你想要的类型都是有效的，如示例7-1所示。可以使用任意转换操作符的风格，这个例子里展示了int(x)和(int)x这两种形式。

示例7-1：使用默认的随机数生成器来产生一对整数和一对浮点数

```
cv::RNG rng = cv::theRNG();
cout << "An integer:       " << (int)rng  << endl;
cout << "Another integer: " << int(rng)  << endl;
cout << "A float:          " << (float)rng << endl;
cout << "Another float:    " << float(rng) << endl;
```

产生整型数的时候，它们将（使用前文描述的MWC算法，因此是均匀分布）覆盖整个可能取值的范围。当产生浮点数的时候，它们的范围始终是[0.0, 1.0)[注7]。

cv::RNG::operator()

```
unsigned int cv::RNG::operator()();        // Return random value from 0-UINT_MAX
unsigned int cv::RNG::operator()( unsigned int N );  // Return value from 0-(N-1)
```

注6：这个“定值”不是0，因为如果它是0，很多随机数生成器（包括RNG使用的）接下来都会返回0。一般来说，这个值是 $2^{32}-1$。

注7：解释一下这个写法以防有人可能不熟悉。一个范围使用方括号[表示这个界限值是包括在内的，使用圆括号(表示不包括这个界限值。因此，[0.0,1.0)表示范围从0.0到1.0，包括0.0但是不包括1.0。

当你要产生整型随机变量时，使用重载的operator()()会更加方便。事实上，调用my_rng()相当于调用(unsigned int)my_rng。cv::RNG::operator()()的一个更有意思的形式使用一个参数N。这种形式返回（使用前文描述的MWC算法，因此是均匀分布）一个模N的无符号随机整数。因此，my_rng(N)返回的整数范围就是从0到N-1的整数。

cv::RNG::uniform()

```
int    cv::RNG::uniform( int a,    int b    );     // Return value from a-(b-1)
float  cv::RNG::uniform( float a,  float b  );     // Return value in range [a,b)
double cv::RNG::uniform( double a, double b );     // Return value in range [a,b)
```

这个函数在[a, b)的范围内产生平均分布（使用MWC算法）的随机数。



C++编译器在决定多个相似形式使用哪个的时候只考虑参数，而不管返回值。因此，当你调用float x = my_rng. uniform(0,1)时，你只得到0.f，因为0和1是整型数，而[0, 1)范围内的整型数只有0。如果想得到浮点数，你应该使用类似my_rng. uniform(0.f, 1.f)这种写法，如果要双精度类型，使用my_rng. uniform(0. ,1.)。当然，对参数使用显式类型转换也是可以的。

cv::RNG::gaussian()

```
double  cv::RNG::gaussian( double sigma ); // Gaussian number, zero mean,
                                           // std-dev='sigma'
cv::RNG::fill()
void  cv::RNG::fill(
  InputOutputArray mat,              // Input array, values will be overwritten
  int              distType,         // Type of distribution (Gaussian or uniform)
  InputArray       a,                // min (uniform) or mean (Gaussian)
  InputArray       b                 // max (uniform) or std-deviation (Gaussian)
 );
```

cv::RNG::fill()算法可以使用特定分布随机数填充一个至多4通道的输入矩阵mat。分布类型使用distType参数指定，可以是cv::RNG::UNIFORM或者cv::RNG::NORMAL。当使用均匀分布时，mat的每个元素都会使用范围内的一个随机数。当使用正态分布(cv::RNG::NORMAL)时，每个元素都会由a作为均值，b作为标准差的正态分布生成。需要注意的是，矩阵a和b并不是mat的维度；它们是$n_c \times 1 \text{or} 1 \times n_c$的，其中nc是mat的通道数，也就是说，mat中的元素分布不是独立的，a和b决定了一个分布，而不是mat中每个元素一个分布。



如果有一个多通道数组，你可以在输入数组a和b中简单的为每个通道使用合适的均值和标准差，以便在“通道空间”中为一个多元分布产生独立的项目。然而，这个分布将会从协方差矩阵的非对角元素全为0的分布中提取得到（这是因为每个元素相对于其他元素是完全独立产生的）如果需要一个更加一般的分布，最简单的方法就是使用cv::RNG::fill()从一个0均值的单位共变矩阵中生成值，然后使用cv::transform()变换到原始的基上。

小结

在这一章里，我们介绍了函数子的概念，以及在OpenCV库中怎样使用它们。我们关注了一些常规功能的对象，看到了它们是怎样工作的。这些包括PCA和SVD对象以及非常有用的随机数生成器RNG。随后，我们深入研究了库中提供的更加高级的算法，我们可以看到库中许多更现代的插件使用同样的概念。

练习

1. 使用cv::RNG随机数生成器完成下列题目。
   a. 生成并显示三个浮点数，使用从0.0到1.0的均匀分布。
   b. 成并显示三个双精度浮点数，使用正态分布，均值0.0，标准差1.0。
   c. 生成并显示三个无符号字节数，使用从0到255的均匀分布。
2. 使用cv::RNG随机数生成器的fill()方法，生成如下数组。
   a. 从0.0到1.0的20个均匀分布的浮点数。
   b. 均值0.0标准差1.0的20个正态分布的浮点数。
   c. 从0到255的20个均匀分布的无符号字节数。
   d. 每个元素是从0到255的均匀分布的20个颜色三元组。
3. 使用cv::RNG随机数生成器，生成一个包含如下100个三字节对象的数组。
   a. 第一维和第二维是正态分布，均值分别为64和192，标准差都是10。
   b. 第三维是正态分布，均值128，标准差2。
   c. 使用cv::PCA对象，计算一个maxComponents=2的映射。
   d. 计算映射前后的维度均值，并解释结果。
4. 从如下矩阵开始。

$$A = \begin{matrix} 1 & 1 \\ 0 & 1 \\ -1 & 1 \end{matrix}$$

a. 首先手算矩阵$A^T A$。求出$A^T A$的特征值（e_1, e_2）和特征向量($\vec{v}_1$, $\vec{v}_2$)。通过特征值，计算出奇异值$(\sigma_1, \sigma_2) = (\sqrt{e_1}, \sqrt{e_2})$。

b. 计算矩阵和。其中，$V = [\vec{v}_1, \vec{v}_2]$和$U = [\vec{u}_1, \vec{u}_2, \vec{u}_3]$，是一个和，同时正交的向量。提示：两个向量的向量积总是和这两个向量正交。

c. 定义矩阵 Σ（考虑到这里A已有特定含义）如下：

$$\Sigma = \begin{matrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{matrix}$$

使用Σ的定义和之前的结果V和U，通过直接乘法验证$A = U \Sigma V^T$。

d. 使用`cv::SVD`对象计算之前的矩阵Σ，V和U，然后验证你笔算的结果是否正确。你是否得到了期望的结果？如果没有，请解释原因。

第8章

图像、视频与数据文件

HighGUI模块：一个可移植的图形工具包

OpenCV把用于操作系统、文件系统以及摄像机等硬件设备交互的函数纳入到HighGUI（High-level Graphical User Interface）模块中。有了HighGUI模块，我们可以方便地打开窗口、显示图像、读出或者写入图像相关的文件（包含图像和视频）、处理简单的鼠标点击、鼠标移动和键盘事件。我们还可以利用HighGUI模块创建一些有用的控件，比如滑动条，并将它们融入到我们所创建的窗口之中。如果对操作系统的图形界面编程很精通，也许会觉得HighGUI所提供的许多功能是没有必要的。但即便如此，HighGUI的跨平台特性对你也会有很大的帮助。

在本章中，我们将探讨HighGUI模块中静止图像的读/写以及视频流的捕获与存储相关的功能。在下一章，我们将学习如何利用HighGUI模块提供的跨平台工具实现与系统原生窗口工具以及其他相关跨平台窗口工具同等效果的图像显示功能。

基于最初的设计理念，OpenCV的HighGUI库可以划分成三部分：硬件相关的部分、文件系统部分以及图形用户界面部分。因此，在深入研究HighGUI模块之前，我们先花一些时间分别浏览一下每一部分的内容。



在OpenCV 3.0中，HighGUI模块分成三个子模块：imgcodecs（image encoding/decoding），vedioio（capturing and encoding video）以及一个现在被命名为highgui（用户界面部分）的部分。为了更好地向后兼容，highgui.hpp头文件包括了videoio.hpp和imgcodecs.hpp两个头文件。因此，大部分使用OpenCV 2.x版本编写的程序对OpenCV 3.x也是兼容的。将来，我们会使用*HighGUI*这个名字来引用本章所描述的所有的图像输入/输出、视频输入/输出以及图形用户界面相关的功能。代码示例可以同时兼容OpenCV 2.x版本与OpenCV 3.x版本。但是请记住，如果你当前使用的是OpenCV 3.0 并且只需要视频捕获功能或者只需要完成图片的读写读/写功能，那么请单独使用videoio或者imgcodecs 模块。

硬件部分最主要是对摄像机的操作。在大多数的操作系统下，与摄像机的交互是一件复杂而又痛苦的工作。HighGUI模块提供了一种简单的方法用于搜索相机设备并且从相机获取最新的图像数据。它将所有麻烦的部分都隐藏了起来，尽可能让每个使用它的人感到开心。

文件系统部分的主要工作是完成图片的载入与保存。HighGUI令人欣慰的一点就是它使得我们可以使用与读取摄像头数据同样的方法去读取视频文件。我们因此可以免于纠结于如何读取特定设备的数据而专注于完成我们真正感兴趣的代码。本着同样的精神，HighGUI为我们提供了一对函数用来完成图像的加载和保存操作。这两个函数根据文件名的后缀来确定文件的类型，并自动处理必要的编码和解码工作。OpenCV还提供了一系列以XML/YAML文件为基础的函数作为对此类函数的补充，用来帮助人们更加方便地将一些数据以一种简单，方便阅读的文本形式保存下来。

HighGUI的第三部分是窗口系统（或者称为GUI）。HighGUI为我们提供了几个简单的函数用来新建一个窗口，并将图片放入窗口中显示。同时，它也给我们提供了为窗口添加响应鼠标和按键事件功能的方法。这些函数为我们快速建立一个简单的应用程序提供了很大的帮助。仅使用HighGUI库中的滑动条，我们就可以完成很多实用的应用程序。如果我们愿意去链接Qt库，那么还可以得到更多酷炫的功能。[注1]在下一章中，我们将详细介绍以上这些与窗口工具相关的内容。

图像文件的处理

OpenCV为我们提供了许多特殊的函数用于图片加载与保存。它们显式地或者隐式地解决了与图像数据紧密相关的编码与解码问题。这些函数与第8章所描述的以XML/YAML

注1：Qt一个跨平台的窗口组件工具包，我们将在本章稍后再详细介绍一下它的工作原理。

为基础的函数有着许多方面的差别。最主要的区别在于，前者主要应用于图像的文件的处理，与普通的数组数据不同的是，图像数据的处理对于现有的压缩与解压缩算法有着较强的依赖性。将图像格式与对应的编码与解码算法相对应，使得OpenCV函数能够轻松应对各种格式的图片。一些图像压缩与解压缩的算法会以牺牲图像中的部分信息为代价，保留图像原有的视觉体验，对于图像来说这种方式是可以接受的。但对于别的数据，例如参数数据，数据的丢失显然是无法接受的。



有损压缩给图像带来的瑕疵可能会给视觉算法带来许多麻烦，很多时候，视觉算法会发现图像中肉眼无法察觉到的瑕疵，并被那些瑕疵所影响。

需要记住的是，我们首先将要讨论的内容，图像加载与保存的函数，才是与操作系统上现有的图像文件资源或者可用的库进行交互的真正的接口。相对而言，早些时候提及的XML/YAML数据存储，即我们本章后面部分将要讨论的内容，是完全属于OpenCV内部范围内使用的。

图像的载入与保存

从磁盘读取图像或者将图像保存到磁盘是我们完成的最基本的任务。最简单的实现方法是利用OpenCV的cv::imread()和cv::imwrite()函数。这些函数会帮你完成包括与文件系统进行交互、图像编码和解码在内的所有工作。

使用cv::imread()读取图片

为了进行后续操作，首先需要学习如何从磁盘载入图片，完成此项工作的函数是cv::imread()：

```
cv::Mat cv::imread(
  const string& filename,                    // Input filename
  int           flags = cv::IMREAD_COLOR     // Flags set how to interpret file
);
```

使用cv::imread()读取一张图片时，cv::imread()并不关心文件的拓展名是什么，而是分析文件中前几个字节（被称为文件的识别标识或者“魔法序列”）来确定图像的编码格式。第二个参数flags可以被设置为表8-1中的任意一个值。默认情况下，flags被设置为cv::IMREAD_COLOR。这个值表示图片将被以三通道8位的格式读取。在这种情况下，即便原始文件中的图像是灰度图像，读取到内存中的仍然有三通道，每个通道拥有相同数据的图像。如果flags被设置为cv::IMREAD_GRAYSCALE，那么不管文件内部图像是几通

道，图片都以灰度图像的格式加载。flags的另外一个值是cv::IMREAD_ANYCOLOR。在此种情况下，图片的载入方式取决于其内部图像的具体情况，如果是彩色图像，就以三通道的形式载入，如果是灰度图像，则按照单通道的形式载入。[注2]

表8-1：cv::imread()的标志

标志	含义	默认值
cv::IMREAD_COLOR	总是读取三通道图像	是
cv::IMREAD_GRAYSCALE	总是读取单通道图像	否
cv::IMREAD_ANYCOLOR	通道数由文件实际通道数（不超过3）	否
cv::IMREAD_ANYDEPTH	允许加载超过8bit深度。	否
cv::IMREAD_UNCHANGED	等于将cv::IMREAD_ANYCOLOR和CV::IMREAD_ANYDEPTH组合了起来。但也不是完全恰当[a]	否

a. IMREAD_UNCHANGED具有另一个独特的效果：读取图像时，它将保留图像中的alpha通道。请注意，即使IMREAD_ANYCOLOR也会将图像通道裁减为三个，而不一定保留alpha通道

除了与颜色相关的标志参数外，cv::imread()还支持cv::IMREAD_ANYDEPTH标志，表明输入图像的通道数据大于8位，使得图像载入过程中不会发生数据类型转换（分配用于存储数据的数组类型将会由其内部数据类型决定）。

cv::imread()在载入图像失败，它并不会抛出异常，而是仅返回一空的cv::Mat（可用cv::Mat::empty()==true来判断）。

使用cv::imwrite()保存图像

显然，与cv::imread()函数形成功能互补关系的肯定是cv::imwrite()，它有三个输入参数，如下所示：

```
bool cv::imwrite(
  const string&      filename,                    // Input filename
  cv::InputArray     image,                       // Image to write to file
  const vector<int>& params = vector<int>() // (Optional) for parameterized fmts
```

第一个参数给定了文件名，文件名的拓展名部分用来决定以何种格式保存图像，以下是一些OpenCV支持的常用拓展名：

- *.jpg*或者*.jpeg*：以baseline JPEG格式保存；8位数据；单通道或三通道输入；
- *.jp2*：JPEG2000；8位或者16位数据；单通道或三通道输入；

注2：在写本书期间，CV::IMREAD_ANYCOLOR尚不适用于那些支持通道的四通道文件，在加载时，图像的第四个通道会被忽略，图像文件会被当作三通道来处理。

- *.tif*或者*.tiff*：TIFF；8位或者16位数据；单通道、三通道或四通道输入
- *.png*：PNG；8位或者16位数据；单通道、三通道或四通道输入
- *.bmp*：BMP；8位数据；单通道、三通道或四通道输入
- *.ppm*，*.pgm*：NetPBM；8位数据； 单通道（PGM）或者三通道（PPM）

第二个参数是待存储的输入图像。第三个参数被用作特殊类型文件的写入操作时所需的数据。输入参数为内部为整型数据的一个STL vector，vector中的整型序列的具体内容为：一系列的参数ID，以及与该参数对应的参数值，每个参数ID之后跟着其对应的值。表8-2是OpenCV为这些参数ID起的别名以及取值范围。

表8-2：cv::imwrite()的标志

标志	含义	取值范围	默认值
cv::IMWRITE_JPG_QUALITY	JPEG的质量	0～100	95
cv::IMWRITE_PNG_COMPRESSION	PNG压缩值（更高的值意味着更多的压缩）	0～9	3
cv::IMWRITE_PXM_BINARY	对PPM，PGM或PBM文件是否使用二值格式	0或1	1

cv::imwrite()函数会将大多数格式的图像以8位单通道或多通道形式保存，但对于像PNG、TIFF或JPEG2000这些非常灵活的文件格式，则允许以16位甚至是浮点格式保存，甚至有些还可以保存为四通道(BGR外加α)。如果成功保存图像，则函数返回true，反之应该返回false。[注3]

关于codecs的一些注释

请记住，cv::imwrite()是为图像文件定制的，它非常依赖那些用于处理图像文件的库。这些库通常被叫做*codecs*（*co*-mpression and *dec*-compress librarie-*s*）。很多人用的操作系统应该已经拥有很多编码解码库，即使没有全部编解码库，但至少对于常见的格式，每一种类型的图片都应该有一个可以使用的编解码库。

OpenCV对一些图片格式（例如JPEG、PNG、TIFF等）自带用于编码解码库。对于这些编解码库，可以有以下三种选择：a)不使用这些编解码库；b)使用OpenCV所提供的编解码库（记得与其他OpenCV模块一起编译）；c)使用相应的拓展库（例如libjpeg、libpng等）。在Windows系统中，默认是*b*选项。在OS X/Linux系统上，默认选项是*c*；如果Cmake无法发现编解码库，就使用选项*b*。你也可以显式指定你所需要的选项。在

注3：这里用“应该”一词是因为在某些操作系统环境下，保存命令可能会导致系统抛出异常，然而在通常情况下，会返回0用于表明操作失败。

Linux系统中，如果想要使用*c*选项，请安装所需的编解码的库以及相关的开发文件（例如*libjpeg*和*libjpeg-dev*）。

图片的编码与解码

正如前文所提到的，cv::imread()和cv::imwrite()是你从磁盘读取图片或者将图像写入磁盘的高级工具，它们是由多个必要的部分整合而成的。在实际的应用中，这样既能够将每个子模块单独使用又可以将其作为一个整体使用的性质非常有用，特别是对于在内存中完成图像的压缩与解压缩工作（通过使用上述的编解码的库）。

使用cv::imencode()压缩图像

图像能够直接以OpenCV内部数组的形式进行编码。在这种情况下，函数的返回结果是一个简单的字符缓冲区而不是之前的数组类型。我们不必为此感到惊讶，因为生成数据的格式目前只对压缩它的编解码的库有意义，其大小也与原始图像不同。

```
void cv::imencode(
    const string&      ext,                    // Extension specifies codec
    cv::InputArray     img,                    // Image to be encoded
    vector<uchar>&     buf,                    // Encoded file bytes go here
    const vector<int>& params = vector<int>()
    );
```

cv::imencode()的第一个输入参数ext是以string表示的文件拓展名，用于函数与对应的编码压缩方式相关联。很显然，这里并不会有实际的文件写入发生。在这里，拓展名发挥着直观指明格式的作用，不仅如此，对于大多数操作系统，它还扮演着帮助系统索引对应编解码库的关键作用。输入的img参数表示要被编码压缩的图像数据，参数buf表示用来存储编码压缩后的图像数据的字符数组。最后一个参数params用于指明要使用编解码器的输入参数。参数params可以选用的值是表8-2中cv::imwrite()的输入参数。

使用cv::imdecode()解码文件

```
cv::Mat cv::imdecode(
    cv::InputArray buf,                    // Encoded file bytes are here
    int            flags = cv::IMREAD_COLOR  // Flags set how to interpret file
    );
```

正如cv::imencode()函数使我们能够将图像数据编码成为字节流，cv::imdecode()函数用于将字节流数据解码成数组形式的图像。cv::imdecode()函数只需要两个输入参数，第一个参数为字节流数据buf（一般情况下应该为std::vector<uchar>类型），[注4]

注4：这里用”应该”一词是因为在某些操作系统环境下，保存命令可能会导致系统抛出异常，然而在通常情况下，会返回0，表明操作失败。

第二个参数flags的取值与cv::imwrite()函数使用的flags有相同的选项（表8-1）。和cv::imread()函数一样，cv::imdecode()不需要文件拓展名作为输入参数，因为它可以从输入的字节流数据的前几个字节推断出应该使用哪种编解码器。

与cv::imread()函数类似，imdecode()在给定的字节流数据为空、包含无效数据或者包含无法使用的数据等情况下也会返回一个空的数组。

视频的处理

当我们处理视频的时候，必须注意一下这些事项，包括将视频文件写入磁盘和从磁盘读取视频文件。当然，一旦我们完成视频的读写读/写，便会想办法在显示屏播放它，以便显示程序的调试或者程序运行结果；接下来，我们将先为你介绍磁盘IO，之后为你介绍视频的播放。

使用cv::VideoCapture对象读取视频流

我们首先需要一个cv::VideoCapture对象，它是我们在第7章所遇到的函数对象（objects that do stuff）的一种，它包含我们从摄像机或者视频文件读取每一帧图像所必需的信息，根据数据源的不同，我们需要以不同的调用方式来构建一个cv::VideoCapture对象：

```
cv::VideoCapture::VideoCapture(
  const string&      filename,        // Input filename
);
cv::VideoCapture::VideoCapture(
  int device                          // Video capture device id
);
cv::VideoCapture::VideoCapture();
```

在第一个构造函数中，我们只需要给出视频文件的文件名即可（*.MPG*、*AVI*等），OpenCV将会打开对应的视频文件，并准备进行后续的读取操作。如果打开成功，那么便可以从中获取图像帧。成员函数cv::VideoCapture::isOpened()会返回true。

许多人在程序没有抛出异常的情况下并不会检查具体的返回结果。但在这儿，请不要这么做，因为在某些情况下，cv::VideoCapture::isOpened()将会返回false，如出于某些原因文件无法打开（例如如果文件不存在），这并非是函数返回false的唯一原因。如果视频被编码时所用的编解码器未知，同样会返回false。由于编解码器的种种问题（既有法律上的也有技术上的），函数无法正常工作比较常见。

同编解码器一样，你还需要对电脑中一些所需要的库进行注册，这样才能使你能够成功读取视频文件。这也是你需要检查cv::VideoCapture::isOpened()返回值的意义，因为

即便你的代码在一台电脑上（具备所需编解码器的静态或者动态链接库）可以使用，也有可能在另一台电脑上无法运行（由于缺少必备的编解码器）。有了一个成功打开的cv::VideoCapture对象之后，便可以从其中读取图像帧数据或者做别的工作。但在继续深入之前，让我们一起看一下如何从摄像机中获取图像数据。

输入参数为整型变量的cv::VideoCapture::VideoCapture()函数除了没有令人头疼的编解码器[注5]问题，在其他方面与我们刚刚讨论的string版本非常相似。在这种情况下，我们给出一个标识符来表明我们希望使用哪个摄像机以及我们希望操作系统以何种方式与该摄像机通信。对于摄像机来说，它只是一个标识码，如果在当前系统下只有一个摄像机时参数为0，当有多个摄像机同时处于当前系统之中时，参数随之递增即可。标识符的另一部分称为“相机的域”（the domain of the camera），表明我们使用的是何种相机，其值可以为表8-3中所定义的一些常量。

表8-3：相机“域”指示HighGUI应该如何匹配相机

相机捕捉常数	对应数值
cv::CAP_ANY	0
cv::CAP_MIL	100
cv::CAP_VFW	200
cv::CAP_V4L	200
cv::CAP_V4L2	200
cv::CAP_FIREWIRE	300
cv::CAP_IEEE1394	300
cv::CAP_DC1394	300
cv::CAP_CMU1394	300
cv::CAP_QT	500
cv::CAP_DSHOW	700
cv::CAP_PVAPI	800
cv::CAP_OPENNI	900
cv::CAP_ANDROID	1000
...	

当我们为cv::VideoCapture::VideoCapture()构造设备参数时，我们传入的标识符为相机的域索引与相机索引之和。例如：

注5：当然，我们必须承认虽然没有那些编解码器带来的问题，但我们仍然会为另一个问题感到头疼，一定要搞清楚我们的操作系统或者相机驱动程序是否支持该相机。

```
cv::VideoCapture capture( cv::CAP_FIREWIRE );
```

在这个例子中，cv::VideoCapture::VideoCapture()将试图打开第一个（例如序号0）对应的FireWire相机。在大多数情况下，如果只有一个相机，则值不是必须的，使用cv::CAP_ANY就够了（因为它默认为0，甚至不必我们手动输入）。在我们继续深入之前，还有最后一件事情需要讨论：在某些平台上，你可以给cv::VideoCapture::VideoCapture()函数传入-1，这会使OpenCV弹出一个选择窗口，让你能够手动选择希望使用的相机。

创建VideoCapture对象的最后一种方式是直接创建一个VideoCapture对象而不提供关于即将打开的硬件设备的任何信息。

```
cv::VideoCapture cap;
cap.open( "my_video.avi" );
```

在这种情况下，虽然会创建一个VideoCapture对象，但无法直接使用，直到显式打开你想要从中读取数据的视频源。可以像cv::VideoCapture的构造函数一样使用cv::VideoCapture::open()函数，并为其传入一个STL string类型或者一个设备编号作为参数。无论是这两种方式中的哪一种，cv::VideoCapture::open()函数最终产生的效果都与调用使用相同参数的cv::VideoCaptrue构造函数相同。

使用cv::VideoCapture::read()从视频流中读取图像

```
bool cv::VideoCapture::read(
  cv::OutputArray image                    // Image into which to read data
);
```

一旦有了一个cv::VideoCapture对象，便可以从中读取图像了。最简单的一种方式是调用cv::VideoCapture::read()函数，该函数从cv::VideoCapture表示的文件中读取下一帧数据，并将数据插入你提供的变量中。同时，这个行为将自动使VideoCapture对象更新，以便随后调用cv::VideoCapture::read()能够继续返回下一帧图像。 如果读取没有成功（例如已经读取到文件的最后一帧），那么此函数将返回false（成功则返回true）。与此同时，你为函数提供的序列对象也会为空。

使用cv::VideoCapture::operator>>()从视频流读取图像数据

```
cv::VideoCapture& cv::VideoCapture::operator>>(
  cv::Mat& image                                // Image into which to read data
);
```

作为对cv::VideoCaptrue::read()函数的补充，你还可以使用重载函数cv::VideoCapture::operator>>()（输入流操作符）从VideoCapture对象中读取下一帧

数据。在这种情况下，由于cv::VideoCapture::operator>>()是一个流操作符，所以无论是否读取成功，它都会返回一个对最初的cv::Capture对象的引用，除此之外，它的作用与cv::VideoCapture::read()完全相同。所以在这种情况下，请务必检查返回的图像序列是否为空。

使用cv::VideoCapture::grab()和cv::Video::retrieve()从视频流中读取图像数据

为了取代上述的一次从相机或者视频源只读取一张图像并完成解码的操作，你可以将其拆分为一个捕获操作（类似于内存的拷贝）和恢复操作（用于实际完成对已抓取的数据进行解码）。

```
bool cv::VideoCapture::grab( void );
bool cv::VideoCapture::retrieve(
  cv::OutputArray image,                    // Image into which to read data
  int      channel = 0                      // Used for multihead devices
);
```

cv::VideoCapture::grab()函数将当前可以使用的图像数据拷贝至用户看不到的一个内部缓存区。那么为什么我们希望OpenCV将图像帧放在一个我们无法触及的地方呢？原因是抓取的图像帧是未处理的，grab()函数只是设计用于将原始图像数据尽快获取到电脑（一般是从相机）。



之所以将捕获（grab）和恢复（retrieve）分开进行而不是像cv::VideoCaptrue::read()函数那样同时进行，是有许多原因的。多相机的数据读取是最常见的一种情况（例如立体成像）。在这种情况下，需要尽可能缩短多个相机获取的图像帧之间的时间差（对于立体成像的理想情况是完全同步获取）。因此，先完成数据的抓取，将数据安全放入缓存区之后再进行解码恢复，这样做非常有意义。

同cv::VideoCapture::read()函数一样，cv::VideoCapture::grab()只在抓取成功时返回true。

一旦成功抓取到图像帧，你便可以调用cv::VideoCapture::retrieve()函数用于处理必要的图像解码，内存分配与拷贝，最终以cv::Mat形式的序列将图像帧返回给你。cv::VideoCapture::retrrieve()函数与cv::VideoCapture::read()函数非常相似，不同之处在于，它处理的是内部缓存区数据而不是cv::VideoCapture::grab()所拷贝的图像帧。cv::VideoCapture::read()和cv::VideoCapture::retrieve()函数的另一个重要区别是附加的通道参数。当使用的设备本身就有多个传感头时（例如多个成像设备）。多传感头在被设计用于立体成像的设备中非常常见，同样还有Kinect这样的特殊设备。[注6]

注6：　目前支持的多传感头设备只有Kinect和Videre，其他的设备支持将逐渐完善。

在上述这些情况下，通道参数的值用来指示从设备获取的数据中恢复哪个图像。同样，在这些情况下，你可能只需要调用一次cv::VideoCapture::grab()函数，但需要多次调用cv::VideoCapture::retrieve()函数（每次传入不同的通道参数）。

相机属性cv::VideoCapture::get()和cv::VideoCapture::set()

视频文件通常不仅仅包含一帧一帧的图像，还包含许多重要的元数据（meta data），这对于正确处理文件有着重要的意义。当打开一个视频文件时，相关信息就会被拷贝到cv::VideoCapture对象内部数据区。通常，如果能够实现对该数据区参数的读取或者写入将会对我们非常有帮助。cv::VideoCapture::get()函数与cv::VideoCapture::set()为我们提供了实现上述功能的途径：

```
double cv::VideoCapture::get(
  int    propid                          // Property identifier (see Table 8-4)
);

bool cv::VideoCapture::set(
  int    propid                          // Property identifier (see Table 8-4)
  double value                           // Value to which to set the property
);
```

cv::VideoCapture::get()常用的属性ID在表8-4中列出。[注7]

表8-4：cv::VideoCapture::get() 和cv::VideoCapture::set()中的视频捕获属性

视频捕获属性	是否只有在摄像头模式下可以使用	含义
cv::CAP_PROP_POS_MSEC		视频文件中的当前位置（毫秒）或视频捕获时间戳
cv::CAP_PROP_POS_FRAMES		从零开始下一帧的索引
cv::CAP_PROP_POS_AVI_RATIO		视频中的相对位置（范围为0.0到1.0）
cv::CAP_PROP_FRAME_WIDTH		视频帧的像素宽度
cv::CAP_PROP_FRAME_HEIGHT		视频帧的像素高度
cv::CAP_PROP_FPS		录制视频的帧速率
cv::CAP_PROP_FOURCC		四个字符代码指示编解码
cv::CAP_PROP_FRAME_COUNT		视频文件中的帧总数
cv::CAP_PROP_FORMAT		返回的Mat对象的格式（例如CV_8UC3）

注7：我们应当理解，并不是被OpenCV所识别的属性都能够被VideoCapture的”后端”（backend）部分识别以及处理。例如，在安卓系统上内部，Linux系统中的Firewire相机（例如dc1394）驱动，QuickTime或者Kinect驱动所使用的捕获机制差别非常大，它们对这一长串的拓展参数选项并不是全部都支持的。希望这个列表能够尽快跟上新系统更新的速度。

表8-4：cv::VideoCapture::get() 和cv::VideoCapture::set()中的视频捕获属性（续）

视频捕获属性	是否只有在摄像头模式下可以使用	含义
cv::CAP_PROP_MODE		表示捕捉模式；值是特定于正在使用的视频后端（例如，DC1394）
cv::CAP_PROP_BRIGHTNESS	✔	相机的亮度设置（支持时）
cv::CAP_PROP_CONTRAST	✔	相机的对比度设置（支持时）
cv::CAP_PROP_SATURATION	✔	相机饱和度设定（支援时）
cv::CAP_PROP_HUE	✔	相机色调设置（支持时）
cv::CAP_PROP_GAIN	✔	相机的增益设置（支持时）
cv::CAP_PROP_EXPOSURE	✔	相机曝光设置（支持时）
cv::CAP_PROP_CONVERT_RGB	✔	如果非零，捕获的图像将被转换为具有三个通道
cv::CAP_PROP_WHITE_BALANCE	✔	相机的白平衡设置（支持时）
cv::CAP_PROP_RECTIFICATION	✔	立体相机整流标志（仅适用于DC1394-2.x）

大多数属性的含义都是显而易见的。POS_MSEC表示当前在视频文件中的位置，单位是毫秒。POS_FRAME表示当前所在位置的帧号。POS_AVI_RATIO是用一个0.0~1.0之间的值表示的位置（当你想使用滑动条去调节视频的浏览进度时，这个参数将非常有用）。FRAME_WIDTH和FRAME_HEIGHT两个参数表示当前视频中可以读取的独立帧的大小（或者表示当前相机的设置可以获取的图像帧的大小）。FPS参数仅限于视频文件，用于表明一秒钟可以获取多少帧图像。当你需要播放视频文件并且让其按照正确的速度播放时，需要这个参数。FOURCC表示四字符代码，供你当前正在读取的视频文件将要使用的编解码器使用。FRAME_COUNT理论上应该是视频中的总帧数，但这个表中的数据并不完全可靠。

这里所有参数都以double类型返回，这对大部分都适用，除了参数为FOURCC (FourCC) [FourCC85]的情况。此时需要转换参数的类型，具体见示例8-1。

示例 8-1：获取视频的编码

```
cv::VideoCapture cap( "my_video.avi" );
unsigned f = (unsigned)cap.get( cv::CAP_PROP_FOURCC);
char fourcc[] = {
                  (char) f,         // First character is lowest bits
                  (char)(f >> 8),   // Next character is bits 8-15
                  (char)(f >> 16),  // Next character is bits 16-23
                  (char)(f >> 24),  // Last character is bits 24-31
                  '\0'              // and don't forget to terminate
                };
```

对于前面这些视频捕获属性，有一个cv::VideoCapture::set()函数用来设定相关属性的值。这些参数并不是所有时候都是有效的，例如，读取视频文件时，不应该设定FOURCC

参数。通过设定当前读取位置参数来改变当前在视频中的位置是可行的，但只对一部分编码格式有效（视频的编码和解码将在后面讲解更多内容）。

使用cv::VideoWriter对象写入视频

我们对视频的另一个操作是将视频写入磁盘。OpenCV使这件事变得十分简单，除了一些细节外，大致与读取视频一致。

跟我们读取视频时需要创建一个cv::VideoCapture对象类似，在写入视频之前，我们必须首先创建一个cv::VideoWriter对象。VideoWriter类有两个构造函数，一个是默认构造函数，仅仅创建一个未初始化的VideoWriter对象用于之后的打开操作，另一种是拥有所有需要的参数，并初始化VideoWriter对象。

```
cv::VideoWriter::VideoWriter(
  const string& filename,                    // Input filename
  int           fourcc,                      // codec, use CV_FOURCC() macro
  double        fps,                         // Frame rate (stored in output file)
  cv::Size      frame_size,                  // Size of individual images
  bool          is_color = true              // if false, you can pass gray frames
);
```

你可能会注意到，相比较于视频的读取，视频的写入需要一些额外的参数，除了文件名称外，我们还要指定使用的编解码格式、帧率以及每一帧的大小。还可以选择性地指定图像是否彩色，如果将isColor参数设置为false，你可以传入灰度图，VideoWriter会正确处理它们。

如同视频读取一样，你也可以使用默认构造函数创建一个VideoWriter对象，然后使用cv::VideoWriter::open()方法对其进行配置，它将使用与完整的构造函数相同的参数。例如：

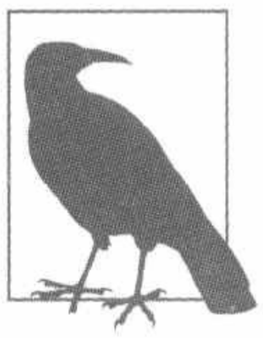
```
cv::VideoWriter out;

out.open(
  "my_video.mpg",
  CV_FOURCC('D','I','V','X'),   // MPEG-4 codec
  30.0,                         // Frame rate (FPS)
  cv::Size( 640, 480 ),         // Write out frames at 640x480 resolution
  true                          // Expect only color frames
);
```

在这里，使用四个字符标记来表示编码格式（对于不了解视频编码的读者来说，这里只需要了解每一种编码都会有唯一的四字符标识对应）。这里，cv::VideoWriter::VideoWriter()中名为fourcc的整型参数是由四个字符打包在一起构成的。由于经常使

用，所以OpenCV提供了一个很方便的宏CV_FOURCC(c0，c1，c2，c3)用于相关数据的打包。请记住，这里的c0，c1，c2，c3为字符（单引号）而不是字符串（双引号）。

一旦将所有VideoWriter需要的参数配置好，并且成功完成初始化，那么便可以询问它是否已经准备好开始工作了，使用cv::VideoWriter::isOpened()函数可以帮助你查询VideoWriter是否已经完全就绪，如果就绪，函数返回true，反之返回false。如果函数返回false，可能是因为你对指定的写入文件所在文件夹没有权限访问或者（最常见）你所指定的编解码器无效。



编解码器是否可用取决于你的系统类型以及是否完成相关附加库安装。能够较好地处理个别编解码器在某些机器上无法正常使用的问题，对于开发可移植的代码尤其重要。

使用cv::VideoWriter::write()函数完成图像帧的写入

一旦确定你所创建的VideoWriter对象已经准备好，可以进行写入工作，将想要写入的图像帧数组数据传入VideoWriter对象中即可：

```
cv::VideoWriter::write(
  const Mat& image                    // Image to write as next frame
);
```

图像必须和你所创建的VideoWriter具有相同的图像尺寸。如果写入彩色图像，那么必须是三通道图像。如果指定VideoWriter可以不是彩色图像，便可以使用单通道（灰度图）供写入。

使用cv::VideoWriter::operator<<()将图像写入视频

VideoWriter类也支持对输出流操作符的重载，在这种情况下，一旦VideoWriter成功打开，便可以像使用cout一样地使用<<()向视频写入图像帧：

```
my_video_writer << my_frame;
```

在实际应用中，可以根据自己的个人喜好选择哪种方式。

数据存储

作为对标准视频压缩方式的补充，OpenCV提供了一种序列化与反序列化的机制，用于将不同数据类型的数据以YAML或者XML格式写入磁盘或者从磁盘读取。这些方法可以用来加载或者保存任何OpenCV的数值变量（包括基本数据变量，像int和float等）到

一个文件中。这里的函数与之前所介绍的一些用于图像或者视频文件特定情景的函数完全独立。本小节将侧重于一般对象的存储（general object persistence）： 矩阵读/写、OpenCV结构体类型、配置文件以及日志文件等。

上述文件读写读/写的基本机制是建立在cv::FileStorage对象基础上的，cv::FileStorage实际上代表磁盘中的一个文件，但与普通方式不同的是，它让人们以一种简单而自然的方式访问文件中的数据。

cv::FileStorage的写入

```
FileStorage::FileStorage();
FileStorage::FileStorage( string fileName, int flag );
```

cv::FileStorage对象代表着一个XML或者YAML格式的数据文件。你可以给定文件名参数创建一个cv::FileStorage对象或者使用默认构造函数创建一个未打开的cv::FileStorage对象，稍后再使用cv::FileStorage::open()函数打开。flag参数可以为cv::FileStorage::WRITE或者cv::FileStorage::APPEND。

```
FileStorage::open( string fileName, int flag );
```

一旦成功打开了你想要写入的文件，便可以像对标准输出流输出数据一样使用操作符cv::FileStorage::operator<<()进行写入操作。你可以以这种简单的方式写入，是因为函数内部为你完成了许多复杂的工作。

cv::FileStorage内部数据的存储主要有两种形式，“mapping”（键/值对）和“sequence”（一系列未命名的条目）。在最顶层，你所写入的数据都在一个mapping中，在该mapping中，你可以放置其他的mappings或者sequences，甚至在mapping中继续放入mapping等，只要你愿意。

```
myFileStorage << "someInteger" << 27;                      // save an integer
myFileStorage << "anArray" << cv::Mat::eye(3,3,CV_32F);    // save an array
```

如果要创建一个序列条目，首先你得为它提供一个string类型的名字，接下来才是序列数据。条目内容可以是数字（整型或浮点型等），一个字符串或者别的OpenCV数据类型。

如果想要创建一个新的mapping或者sequence，可以使用特殊符号{（用于mapping)或者[（用于sequence）。一旦开始创建，就可以为其添加元素，最终以}或者]分别结束一个mapping或者sequence。

```
myFileStorage << "theCat" << "{";
myFileStorage << "fur" << "gray" << "eyes" << "green" << "weightLbs" << 16;
myFileStorage << "}";
```

一旦完成创建一个mapping，需要按顺序输入条目名以及对应的值，像你在最顶层的mapping完成的工作一样。如果创建的是sequence，只需要一个接一个地输入元素即可，直到sequence结束。

```
myFileStorage << "theTeam" << "[";
myFileStorage << "eddie" << "tom" << "scott";
myFileStorage << "]";
```

一旦完成写工作，便可以使用成员函数cv::FileStorage::release()关闭该文件。

示例8-2为OpenCV参考文档中的示例代码。

示例8-2：使用 cv::FileStorage创建一个*a.yml*参数文件

```
#include "opencv2/opencv.hpp"
#include <time.h>

int main(int, char** argv)
{

    cv::FileStorage fs("test.yml", cv::FileStorage::WRITE);

    fs << "frameCount" << 5;

    time_t rawtime; time(&rawtime);
    fs << "calibrationDate" << asctime(localtime(&rawtime));
    cv::Mat cameraMatrix = (
      cv::Mat_<double>(3,3)
      << 1000, 0, 320, 0, 1000, 240, 0, 0, 1
    );
    cv::Mat distCoeffs = (
      cv::Mat_<double>(5,1)
      << 0.1, 0.01, -0.001, 0, 0
    );
    fs << "cameraMatrix" << cameraMatrix << "distCoeffs" << distCoeffs;

    fs << "features" << "[";
    for( int i = 0; i < 3; i++ )
    {
        int x = rand() % 640;
        int y = rand() % 480;
        uchar lbp = rand() % 256;

        fs << "{:" << "x" << x << "y" << y << "lbp" << "[:";
        for( int j = 0; j < 8; j++ )
            fs << ((lbp >> j) & 1);
        fs << "]" << "}";
    }
    fs << "]";
    fs.release();

    return 0;

}
```

程序运行结果如下：

```
%YAML:1.0
frameCount: 5
calibrationDate: "Fri Jun 17 14:09:29 2011\n"
cameraMatrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1000., 0., 320., 0., 1000., 240., 0., 0., 1. ]
distCoeffs: !!opencv-matrix
   rows: 5
   cols: 1
   dt: d
   data: [ 1.00000000000000001e-01, 1.00000000000000000e-02,
       -1.00000000000000000e-03, 0., 0. ]
features:
   - { x:167, y:49,  lbp:[ 1, 0, 0, 1, 1, 0, 1, 1 ] }
   - { x:298, y:130, lbp:[ 0, 0, 0, 1, 0, 0, 1, 1 ] }
   - { x:344, y:158, lbp:[ 1, 1, 0, 0, 0, 0, 1, 0 ] }
```

在这个示例代码中，你会注意到有时候mapping或者sequence的所有数据存储在一行，有时候每个元素一行。这并不是自动格式化造成的。而是由于mapping的起始字符"{:"和"}"，sequence的起始字符"[:"和"]"的变化造成的。这个特点只对YAML格式的输出有意义；如果输出文件是XML格式，那么这些细微的差别将会被忽略，mapping与sequence的存储将不会有区别。

使用cv::FileStorage读取文件

```
FileStorage::FileStorage( string fileName, int flag );
```

cv::FileStorage对象在打开之后既可以用作写入，也同样可以用作读取。唯一的区别是参数flag的值为cv::FileStorage::READ。跟写入时一样，也可以用默认构造函数创建一个未打开的FileStorage对象，之后再通过调用cv::FileStorage::open()函数打开。

```
FileStorage::open( string fileName, int flag );
```

一旦文件被打开，接下来便可以对其中的数据进行读取，首先需要确定你要访问的数据名，即FileStorage最顶层的mapping中的关键字，可以通过重载操作符cv::FileStorage::operator获取自己需要的数据。然而，该操作符的返回值并不是你想要的数据，而是一个FileNode对象。

接下来，如果该值是基本数据类型，则可以通过强制类型转换直接获取，如果该数据为一个mapping对象，可以通过重载操作符[]，并给定所需关键字访问；如果该数据为sequence对象，可以通过重载的[]和一个整型参数访问其中数据，也可以

通过cv::FileNodeIterator迭代器进行访问。在完成读取之后，记得调用成员函数cv::FileStorage::release()关闭文件。关于cv::FileNode对象的更多操作，我们将在后面继续讨论。

cv::FileNode

当成功构建一个cv::FileNode对象之后，便可以利用它来完成许多工作。如果它直接表示一个实际的对象（或者一个数字或者字符串），你就可以直接使用重载操作符cv::FileNode::operator>>()，将它的值加载到对应类型的变量之中。

```
cv::Mat anArray;
myFileStorage["calibrationMatrix"] >> anArray;
```

cv::FileNode对象同样支持直接赋值给一些基本数据类型。

```
int aNumber;
myFileStorage["someInteger"] >> aNumber;
```

与下面这种方式等价：

```
int aNumber;
aNumber = (int)myFileStorage["someInteger"];
```

正如早些时候所提及的，使用迭代器可以完成对FileNode的遍历。给定一个cv::FileNode对象，成员函数cv::FileNode::begin()和cv::FileNode::end()可以分别返回该mapping或者sequence的首元素的迭代器和和末尾元素的迭代器。该迭代器对象可以通过cv::FileNodeIterator::operator*()操作符来进行解引用操作并返回迭代器对应的位置的FileNode对象。这些迭代器支持常见的自增以及自减操作符。如果该迭代器在对一个mapping进行遍历，那么返回的cv::FileNode对象将会有一个名字属性，并且可以通过cv::FileNode::name()函数进行获取。

在表8-5列出的一些函数中，有一个函数需要单独说明：cv::FileNode::type()函数。该函数的返回值为在类cv::FileNode中所定义的枚举类型。 可能的枚举值在表8-6中列出。

表8-5：cv::FileNode的成员函数

示例	说明
cv::FileNode fn	文件节点对象的默认构造函数
cv::FileNode fn1(fn0)	文件节点对象复制构造函数；从节点fn0创建节点fn1
cv::FileNode fn1(fs, node)	文件节点构造函数; 从C风格的CvFileStorage *指针fs和C风格的CvFileNode *指针节点创建一个C++风格的cv::FileNode对象

表8-5：cv::FileNode的成员函数（续）

示例	说明
fn[(string)key] fn[(const char*)key]	命名子串（映射节点）的STL字符串或C字符串存取器；将密钥转换为适当的子节点
fn[(int)id]	子编号（序列节点）的访问；将ID转换为相应的子节点
fn.type()	返回节点类型enum
fn.empty()	确定节点是否为空
fn.isNone()	确定节点是否具有值None
fn.isSeq()	确定节点是否是序列
fn.isMap()	确定节点是否是映射
fn.isInt() fn.isReal() fn.isString()	确定节点是否为整数、浮点数或字符串（分别）
fn.name()	如果节点是映射的子节点，则返回节点名称
size_t sz=fn.size()	返回序列或映射中的一些元素
(int)fn (float)fn (double)fn (string)fn	从包含整数、32位浮点数、64位浮点数或字符串（分别）的节点中提取值

表8-6：cv::FileNode::type()可能的返回值

示例	说明
cv::FileNode::NONE = 0	节点类型为None
cv::FileNode::INT = 1	节点包含一个整数
cv::FileNode::REAL = 2 cv::FileNode::FLOAT = 2	节点包含一个浮点数注释[a]
cv::FileNode::STR = 3 cv::FileNode::STRING = 3	节点包含一个字符串
cv::FileNode::REF = 4	节点包含参考（即复合对象）
cv::FileNode::SEQ = 5	节点本身是其他节点的序列
cv::FileNode::MAP = 6	节点本身就是其他节点的映射
cv::FileNode::FLOW = 8	节点是序列或映射的紧凑表示
cv::FileNode::USER = 16	节点是注册对象（例如矩阵）
cv::FileNode::EMPTY = 32	节点没有赋值给它
cv::FileNode::NAMED = 64	节点是映射的子节点（即它有一个名字）

a. 不区分浮点类型。这里有些玄机。回想一下，XML和YAML是ASCII文本文件格式。因此，所有浮点数都不具有特定的精度，直到转换为内部机器变量类型为止。所以，在解析的时候，所有的浮点数只能用抽象的浮点类型表示

说明：这里并不会区分浮点数为单精度或者双精度。回想一下，XML和YAML是以ASCII文本格式来表示数据的，所以所有浮点数据都将以抽象的浮点数据表示，直到它被转换为实际的变量才能确定最终为单精度浮点数还是双精度浮点数。

最后四个枚举值都是以2为底的幂（从8开始）。原因是一个FileNode除了拥有前面8种属性中的一种之外，还可以同时拥有这四个属性中的一个或者全部属性。

示例8-3（参见OpenCV官方文档）使用前面介绍的方法读取文件。

示例8-3：使用cv::FileStorage来读取a.yml文件

```
cv::FileStorage fs2("test.yml", cv::FileStorage::READ);

// first method: use (type) operator on FileNode.
int frameCount = (int)fs2["frameCount"];

// second method: use cv::FileNode::operator >>
//
std::string date;
fs2["calibrationDate"] >> date;

cv::Mat cameraMatrix2, distCoeffs2;
fs2["cameraMatrix"] >> cameraMatrix2;
fs2["distCoeffs"] >> distCoeffs2;

cout << "frameCount: "        << frameCount    << endl
     << "calibration date: "  << date          << endl
     << "camera matrix: "     << cameraMatrix2 << endl
     << "distortion coeffs: " << distCoeffs2   << endl;

cv::FileNode features   = fs2["features"];
cv::FileNodeIterator it = features.begin(), it_end = features.end();
int idx                 = 0;
std::vector<uchar> lbpval;

// iterate through a sequence using FileNodeIterator
for( ; it != it_end; ++it, idx++ )
{
    cout << "feature #" << idx << ": ";
    cout << "x=" << (int)(*it)["x"] << ", y=" << (int)(*it)["y"] << ", lbp: (";

    // ( Note: easily read numerical arrays using FileNode >> std::vector. )
    //
    (*it)["lbp"] >> lbpval;
    for( int i = 0; i < (int)lbpval.size(); i++ )
        cout << " " << (int)lbpval[i];
    cout << ")" << endl;

}
fs.release();
```

小结

在本章中，我们学习了几种与磁盘文件及硬件设备交互的方式。我们了解了OpenCV中的HighGUI模块为我们提供的一些从磁盘读/写图像文件的通用工具。这些工具会自动为我们完成对应文件格式的编码与解码工作。我们还学习到了用类似的工具去读/写磁盘中的视频文件以及完成从摄像头捕获图像。最后，我们了解到OpenCV为我们提供了非常强大的工具用于完成对XML和YAML格式文件的处理。这些文件能够以键/值对的形式存放数据，一旦读取到内存，便可以很方便地完成相关的解析工作。

练习

本章完成了对一些OpenCV中基本的数据I/O操作的介绍。下面这些习题可以帮助我们更好地理解学习的内容，还可以完成一些程序供之后的章节使用。

1. 编写程序完成：(1) 从视频读取图像帧；(2) 将结果转化成灰度图像；(3)对该图像完成Canny边缘检测。用三个窗口分别显示上述三个步骤的结果，并给窗口适当命名。

   a. 在同一个窗口中同时显示三个阶段的图像。（提示：创建一个图像，高度与原始图像相同，宽度是原始视频帧的三倍。将三个图像分别拷贝到该图像中，或者使用指针（更加机智的方法）创建三个图像头部，分别指向该图像数据的开头处、三分之一处和三分之二处，接下来使用Mat::copyTo()函数。）

   b. 使用文本标签分别描述三个过程。

2. 编写程序读取并显示一张图片。当用户鼠标点击图片时，读取当前鼠标对应位置的像素值（蓝，绿，红）并且以文本的形式在鼠标在屏幕中所在位置显示出来。对于练习2，鼠标单击显示的图像时，在图像中显示鼠标在单个图像中的坐标。

3. 编写程序读取并显示图片并完成下列要求。

   a. 允许当用户按下鼠标时在图像中选择一个矩形区域，并且在用户释放鼠标时，高亮该矩形区域。记得在内存中备份图像，防止破坏原始数据。下一次的鼠标点击可以继续完成上述工作。

   b. 在一个单独的窗口中，使用画图函数画出上述选择区域中蓝绿红每种颜色的值所对应的像素的个数，即该区域的颜色直方图。*x*轴为8位的像素值，分别落在0~31，32~63，……，223~255小区间内。*y*轴为像素值落在该区间的像素个数。对BGR每个通道都进行该操作。

4. 编写应用读取并播放视频，并通过滑动条控制播放位置，每次增量为10；另外使用一个二值的滑动条完成视频的暂停/播放操作。为两个滑动条添加合适的标签。

5. 创建一个属于你自己的画图程序。

   a. 编写程序，创建一个图像，所有值为0，并显示。让用户能够使用鼠标左键在上面前面画直线，圆，椭圆和多边形。创建一个擦出功能的函数供鼠标右键按下的时候使用。

   b. 开启“逻辑画图”功能，通过滑动条是用户能够选择“与”“或”“异或”模式。意思是，如果设置为“与”模式，那么只有当用户鼠标所在位置的值大于0时才能进行涂画（其他逻辑同理）。

6. 编写程序创建一个图像，值为0，并显示。当用户单击一个位置时，允许用户在该位置输入一个标签，允许用户通过Backspace进行编辑，并提供一个放弃按钮。当按下Enter键时，将该标签放置于输入时的位置。

7. 透视变换。

   a. 编写程序，读取一张图片并使用键盘上的按键1-9控制透视变换矩阵（参考第11章关对`cv::warpPerspective()`函数的讨论），按下按键会增加变换矩阵对应位置元素的值，使用Shift键与数字组合键可以减小变换矩阵对应位置的元素值（到零为止）。数值发生改变时，在两张图像中显示结果：一张变换前，一张变换后。

   b. 添加放大与缩小功能。

   c. 添加旋转功能。

8. 有趣的人脸：找到并编译示例代码*/samples/cpp*文件夹中的*facedetect.cpp*代码。画一个头骨图像（或者在网上找一个）并将其存储于磁盘，修改*facedetect*程序用于加载头骨图像。

   a. 当一个人脸被检测到并用矩形圈出时，在该矩形内画出一个头骨图像。提示：你可能需要`cv::resize()`函数调整图像大小与圈出人脸的矩形区域相同，接着使用`Mat::copyTo()`函数将调整过大小的头骨图像拷贝到该矩形区域。

   b. 添加一个滑动条，对应范围`0.0~1.0`内的十个值，使用该滑动条调节头骨的透明度，并使用`cv::addWeighted()`函数将头骨覆盖在该矩形区域。

9. 图像稳定：找到示例目录*/samples/cpp*目录下的*lkdemo*代码（运动跟踪与光流代码），并编译。创建一个比较大的窗口，并在其中显示视频，保证窗口比视频图像大得多。稍微移动相机，并利用光流算法输出的向量保证视频显示的位置不发生变化。这便是基本的图像稳定技术。

10. 创建一个内部有三种成员的结构体：分别是一个整数、一个`cv::Point`和一个`cv::Rect`，命名为`my_struct`。

a. 实现两个函数：void write_my_struct(FileStorage& fs, const string& name, const my_struct& ms) 和 void read_my_struct(const FileStorage& fs, const FileNode& ms_node, my_struct& ms) 用于读/写my_struct。

b. 完成对拥有10个my_struct数据的数组的读/写。

第9章

跨平台和Windows系统

基于Windows开发

前面介绍了HighGUI工具包。在本章，我们将详细了解怎样通过这个工具包来帮助我们完成一些文件和设备相关的任务。除了这些特性以外，HighGUI库中还提供了一些内建的用于创建窗体、显示图像、处理一些可能操作的特性。很久以前，OpenCV中原生的用户图形接口（GUI）相关函数就已经成为这个库的一部分，这些函数具有稳定、便捷的优势。[注1]

尽管HighGUI库使用起来很方便，但它的UI（用户接口）特性还是有不特别完整的缺陷。因此，我们正在尝试通过将初始的接口转化为Qt接口来完成现代化UI部分的工作，并将一些新的特性加到里面。Qt是一个跨平台的工具包，因此新特性的只需要在库中实现一次，而不是对每一个平台都需要实现。显而易见，这使得Qt接口的开发更有吸引力，所以它会具有更强的功能，并且在未来还会进一步增长，以前接口的代码作为静态的遗留代码保存下来。

最后，我们还将学习如何将OpenCV的数据结构和一些常用操作系统已经的平台专用工具包进行集成。

注1：可移植的原因是因为它们使用了各种平台上原始的窗体GUI。也就是说，在Linux系统中使用了X11，Mac OS X上使用了Cocoa，在微软的Windows系统上使用原始Win32 API。虽然，这种便捷性针对各个平台的库有具体实现的功能，但这里也存在没有实现的（比如安卓系统）。

HighGUI原生图形用户接口

本节将介绍核心的接口函数。这些接口函数是OpenCV的一部分，并不需要额外的工具包支持。如果将OpenCV编译成Qt支持的，其中一些函数可能会有一些不同或额外的参数选项；在接下来的一些小节中我们将逐步对此进行介绍。本节我们将主要集中介绍基本的 HighGUI UI工具。

HighGUI的图形用户输入工具只支持三个基本的交互，即按键、鼠标在图像区域点击、鼠标滚轮。这些基本的功能已经足够用于简单的模型和调试，但对于面向用户的应用还不太理想。因此，我们至少会使用基于Qt的接口或其他具有更完整特性的UI工具包。

原始库的工具包的最大优势是它们速度快，使用简便，并且不需要安装额外的库。

使用cv::namedWindow()创建一个窗口

首先我们想用HighGUI在屏幕上创建一个窗口并显示图像。cv::namedWindow()可以让我们完成这一步的功能，它需要一个窗口的名称和一个标识符flag作为参数输入。窗口的名字出现窗口的顶部，并且可以作为窗口的句柄传入HighGUI的其他函数。[注2]标识符flag用于表示窗口是否需要自动调整大小来适应我们输入的图像大小。完整的函数定义如下：

```
int cv::namedWindow(
  const string&  name,                  // Handle used to identify window
  int            flags = 0              // Used to tell window to autosize
);
```

现在，flag的合法值可以设置为0（默认值）表示用户可以调整窗口的大小，或设置为cv::WINDOW_AUTOSIZE。[注3]如果flag设置为cv::WINDOW_AUTOSIZE，那么加载不同的图像时，HighGUI会自动对窗口大小进行自适应调整，但是用户不能调整窗口。

一旦创建了窗口，我们通常想要放置一些东西在里面。但在完成这件事之前，首先看下当我们不需要一个窗口时，如何销毁一个窗口。为了消除一个不需要的窗口，我们使用函数cv::destroyWindow()，它只有一个参数，是一个字符串：创建窗口时给窗口指定的名称。

注2：在OpenCV中，窗口是通过窗口名字引用的，而不是通过一些不太友好的（可变的及与系统相关的）“句柄”引用。基于HighGUI自有的功能，可以实现句柄与名字之间的转换，因此，你不必为此担心。

注3：在接下来的章节里，我们会介绍一些基于Qt的作为后端引擎的HighGUI的函数。如果使用Qt为后端的函数，cv::namedWindow()和其他一些函数中将有更多的参数选项。

```
int cv::destroyWindow(
  const string&  name,                    // Handle used to identify window
);
```

通过cv::imshow()显示图像

现在我们已经准备好我们真正想做的：加载一张图像到窗口中，在窗体中我们可以查看和欣赏图像。我们可以通过一个简单的函数cv::imshow()来完成这些：

```
void cv::imshow(
  const string&  name,                    // Handle used to identify window
  cv::InputArray image                    // Image to display in window
);
```

第一个参数是我们将要用来画图的窗口的名称。第二个参数是将要显示的图像。需要注意的是，窗口会保存显示图像的一个副本，在需要的时候从缓存中重画。因此，图像显示后对输入原图的改变不会影响到显示图像的内容，除非重新调用cv::imshow()。

更新窗口和cv::waitKey()函数

cv::waitKey()函数是用于对键盘按键事件进行特定时长的等待，如果有按键输入，则返回按键值。

```
int cv::waitKey(
  int delay = 0                           // Milliseconds until giving up (0='never')
);
```

cv::waitKey()接受一个参数delay，这是在自动返回之前等待按键的时间量（以毫秒为单位）。如果延迟设置为0，cv::waitKey()将无限期地等待按键。如果在延迟毫秒时间之前没有按键输入，cv::waitKey()将返回-1。

cv::waitKey()的第二个不太明显的功能是让任何OpenCV窗口进行更新。这意味着如果你不调用cv::waitKey()，你的图像可能永远不会在你的窗口中绘制，或者窗口在移动、调整大小或从覆盖中显示出来时可能会表现得非常奇怪（甚至严重异常）。[注4]

一个展示图像的例子

现在我们将一个在窗口中显示图像的程序合并起来（具体可见示例9-1）。我们可以从命令行中读取文件名，通过15行代码（包括注释内容）创建一个窗口并将图像放在窗口中。这个程序可以展示一张图像，然后当我们按Esc（键值为27）时退出。

注4：这句话意味着cv::waitKey()是HighGUI中唯一可以获取和处理事件的函数。如果不定期调用，则不会发生正常的事件处理。作为推论，如果在处理事件的环境中使用HighGUI，则可能不需要调用cv::waitKey()。与此相关的更多内容，请参阅其他地方对页中的cv::startWindowThread()的介绍。

示例9-1：创建窗口并在窗口中展示图像

```
int main( int argc, char** argv ) {

  // Create a named window with the name of the file
  //
  cv::namedWindow( argv[1], 1 );

  // Load the image from the given filename
  //
  cv::Mat = cv::imread( argv[1] );

  // Show the image in the named window
  //
  cv::imshow( argv[1], img );

  // Idle until the user hits the Esc key
  //
  while( true ) {
    if( cv::waitKey( 100 /* milliseconds */ ) == 27 ) break;
  }

  // Clean up and don't be piggies
  //
  cv::destroyWindow( argv[1] );

  exit(0);
}
```

为了方便起见，我们将文件名作为窗口的名称。OpenCV会自动将窗口名称打印在左上角，所以这会非常方便地告诉我们正在浏览哪张图像（图9-1）。

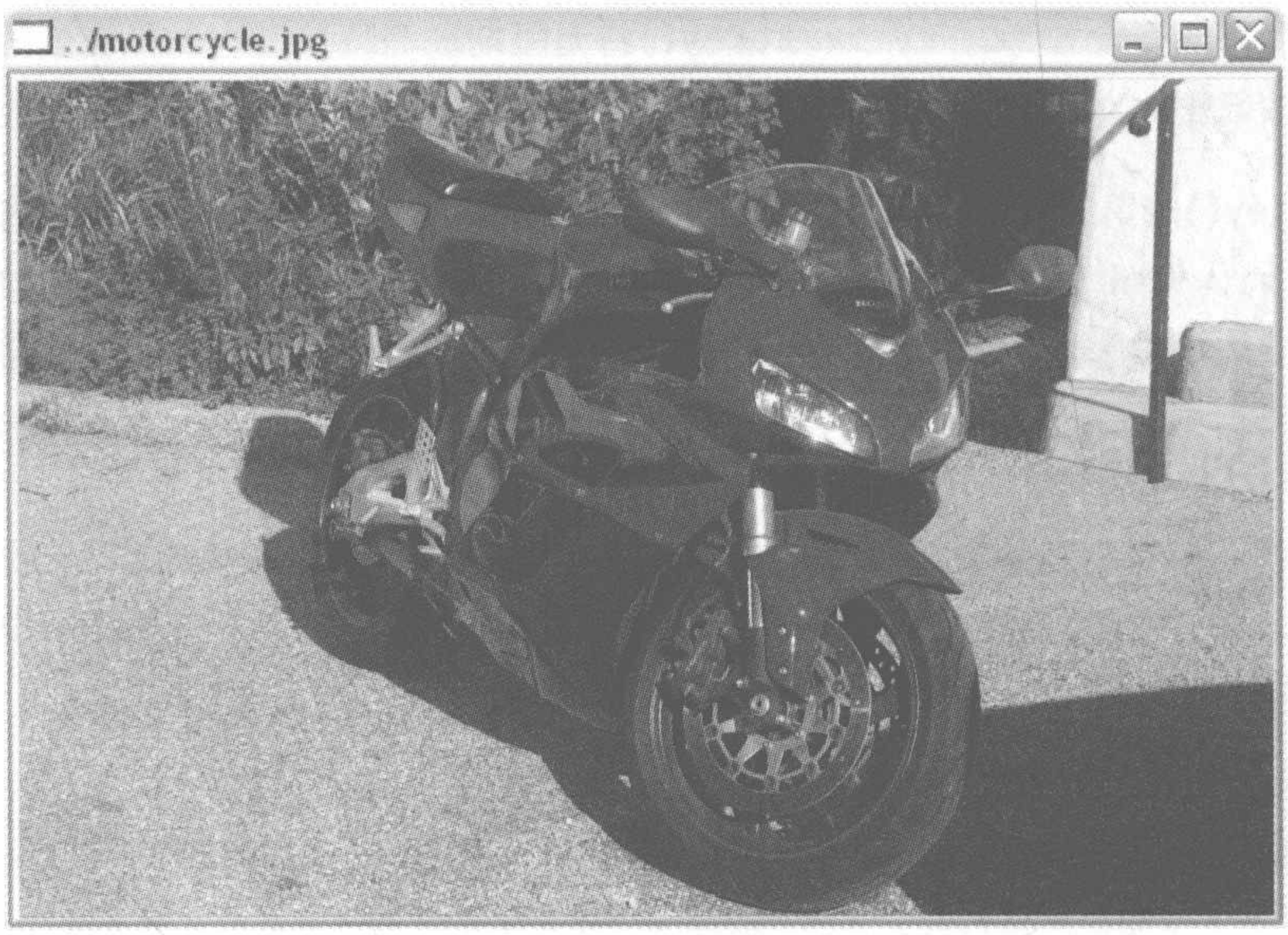


图9-1. 用cv::imshow()显示图像

在我们显示图像前，还有下面几个和窗口相关的函数我们需要了解：

```
void cv::moveWindow( const char* name, int x, int y );
void cv::destroyAllWindows( void );
int  cv::startWindowThread( void );
```

cv::moveWindow()是用于对窗口在屏幕上进行简单移动，使得窗口的左上角坐标位于(*x*,*y*)点。cv::destroyAllWindows()是一个有用的清理函数，它能够关闭所有窗口并释放相关的内存。

在Linux和MacOS X系统中，函数cv::startWindowThread()会尝试启动一个线程来自动更新窗口，并且处理改变大小以及其他一些操作。如果返回值为0，表示没有线程可以启动——比如说，你使用的OpenCV并不支持这一特性。记住，如果没有启动这样的窗口线程，OpenCV只可以在特别指定它和用户交互的时候执行交互（在你委托cv::waitKey()进行处理的时候）。

鼠标事件

现在我们可以为用户显示一张图像，接下来可能还想让用户与显示的图像有一定的交互性。我们是在窗口环境下进行的，并且已经知道如何利用cv::waitKey()进行键值捕捉，那么下一个逻辑事件就是如何对鼠标事件进行“监听”和做出响应。

与键盘事件不同的是，鼠标事件是通过更加传统的回调函数函数机制来完成的，这就意味着，我们必须先写好一个回调程序，使得OpenCV在发生任何鼠标事件时都可以调用这个回调程序。当我们完成回调程序以后，需要在OpenCV中注册这个函数，即告诉OpenCV这是一个正确的回调程序，用户使用鼠标在特定的窗口中完成一些操作后可以得到一定的响应。

让我们从回调说起。你们可能对事件驱动程序这种术语有一些陌生，回调其实就是任何以一个参数集作为输入并返回一个正确的类型的函数。这里需要明确告诉程序使用的回调函数发生的事件类型和发生地点。如果在用户按下鼠标同时还按下了Shift键或Alt键，也需要确切告诉回调函数。指向这样的函数的指针是cv::MouseCallback。以下是定义回调函数必须匹配的接口协议：

```
void your_mouse_callback(
  int   event,                    // Event type (see Table 9-1)
  int   x,                        // x-location of mouse event
  int   y,                        // y-location of mouse event
  int   flags,                    // More details on event (see Table 9-1)
  void* param                     // Parameters from cv::setMouseCallback()
);
```

现在，无论任何时候调用回调函数时，OpenCV都会将合适的参数值填进去。第一个参数，也就是事件，会取表9-1中的一个值。

表9-1：鼠标事件类型

事件	数值
cv::EVENT_MOUSEMOVE	0
cv::EVENT_LBUTTONDOWN	1
cv::EVENT_RBUTTONDOWN	2
cv::EVENT_MBUTTONDOWN	3
cv::EVENT_LBUTTONUP	4
cv::EVENT_RBUTTONUP	5
cv::EVENT_MBUTTONUP	6
cv::EVENT_LBUTTONDBLCLK	7
cv::EVENT_RBUTTONDBLCLK	8
cv::EVENT_MBUTTONDBLCLK	9

第二和第三个参数将被设置为鼠标事件的x坐标和y坐标。需要注意的是，这里的坐标是图像上的像素坐标，是独立于窗口中其他细节的。[注5]

第四个参数叫标志值，是一个bit位，每一个bit位都代表在事件发生时的不同条件。例如cv::EVENT_FLAG_SHIFTKEY的数值是16（即就是第四个bit，或者1<<4）；因此，如果我们想测试Shift键是否按下，我们可以计算按位与flags & cv::EVENT_FLAG_SHIFTKEY。表9-2展示了一个完整的标志列表。

表9-2：鼠标事件标志

标志	数值
cv::EVENT_FLAG_LBUTTON	1
cv::EVENT_FLAG_RBUTTON	2
cv::EVENT_FLAG_MBUTTON	4
cv::EVENT_FLAG_CTRLKEY	8
cv::EVENT_FLAG_SHIFTKEY	16
cv::EVENT_FLAG_ALTKEY	32

注5：通常来说，这个和系统返回的事件的像素坐标不相同，因为OpenCV更加关心鼠标事件在图像上发生的位置，而不是窗口（系统通常返回鼠标事件发生的位置参考）。

最后一个参数是一个void类型的指针，基于这种类型的指针，OpenCV可以用来传递额外的信息到任何类型的结构。[注6]

接下来，需要一个函数来注册这个回调函数，这个函数就是cv::setMouseCallback()，它需要三个参数。

```
void cv::setMouseCallback(
  const string&     windowName,         // Handle used to identify window
  cv::MouseCallback on_mouse,           // Callback function
  void*             param     = NULL    // Additional parameters for callback fn.
);
```

第一个参数是回调函数作用的窗口名称，只有事件在这个特定的窗口中时才能触发回调。第二个参数是你要注册的回调函数。第三个参数param，可以让我们在回调函数执行时给回调函数传递特定的信息。这个也就是我们刚刚讨论的回调函数接口协议的定义里面的param。

在示例9-2中，我们写了一个小程序在屏幕上画框。函数my_mouse_callback()对鼠标事件做出响应，并且当它被调用时，会根据事件来决定自己要完成什么任务。

示例9-2：使用鼠标回调完成绘制方框的小程序

```
#include <opencv2/opencv.hpp>

// Define our callback which we will install for
// mouse events
//
void my_mouse_callback(
   int event, int x, int y, int flags, void* param
);

Rect box;
bool drawing_box = false;

// A little subroutine to draw a box onto an image
//
void draw_box( cv::Mat& img, cv::Rect box ) {
  cv::rectangle(
    img,
    box.tl(),
    box.br(),
    cv::Scalar(0x00,0x00,0xff)    /* red */
  );
}

void help() {
  std::cout << "Call: ./ch4_ex4_1\n" <<
```

注6：这个参数param，通常的应用场景是这个回调函数本身是一个类的静态成员函数。在这种情况下，可以通过传递this指针来指示回调函数要作用于哪个类对象上。

```
    " shows how to use a mouse to draw regions in an image." << std::endl;
}

int main( int argc, char** argv ) {

  help();
  box = cv::Rect(-1,-1,0,0);
  cv::Mat image(200, 200, CV_8UC3), temp;
  image.copyTo(temp);

  box   = cv::Rect(-1,-1,0,0);
  image = cv::Scalar::all(0);

  cv::namedWindow( "Box Example" );

  // Here is the crucial moment where we actually install
  // the callback. Note that we set the value of 'params' to
  // be the image we are working with so that the callback
  // will have the image to edit.
  //
  cv::setMouseCallback(
    "Box Example",
    my_mouse_callback,
    (void*)&image
  );

  // The main program loop. Here we copy the working image
  // to the temp image, and if the user is drawing, then
  // put the currently contemplated box onto that temp image.
  // Display the temp image, and wait 15ms for a keystroke,
  // then repeat.
  //
  for(;;) {

    image.copyTo(temp);
    if( drawing_box ) draw_box( temp, box );
    cv::imshow( "Box Example", temp );

    if( cv::waitKey( 15 ) == 27 ) break;
  }

  return 0;
}

// This is our mouse callback. If the user
// presses the left button, we start a box.
// When the user releases that button, then we
// add the box to the current image. When the
// mouse is dragged (with the button down) we
// resize the box.
//
void my_mouse_callback(
   int event, int x, int y, int flags, void* param
) {
```

```
cv::Mat& image = *(cv::Mat*) param;

switch( event ) {

  case cv::EVENT_MOUSEMOVE: {
    if( drawing_box ) {
      box.width  = x-box.x;
      box.height = y-box.y;
    }
  }
  break;

case cv::EVENT_LBUTTONDOWN: {
  drawing_box = true;
  box = cv::Rect( x, y, 0, 0 );
}
break;

case cv::EVENT_LBUTTONUP: {
  drawing_box = false;
  if( box.width < 0  ) {
    box.x += box.width;
    box.width *= -1;
  }
  if( box.height < 0 ) {
    box.y += box.height;
    box.height *= -1;
  }
    draw_box( image, box );
  }
  break;
 }

}
```

滑动条、滚动条和开关

HighGUI中提供了一个方便的滑动条（slider）单元。在HighGUI中，滑动条也叫“滚动条”（trackbar），因为最初（历史上）滚动条设计的目的就是在视频播放帧中选择特定的帧。当然，当trackbar被添加到HighGUI中以后，人们便开始使用它来完成通常由slider来完成的一些常规任务和一些不常规的任务（我们将在讨论无button控件下的使用时讨论）。

在和父窗口使用时，需要给滑动条赋予一个特定的名称（通常是一个字符串），接下来将直接通过那个名称进行引用。HighGUI中创建一个滑动条的函数如下：

```
int cv::createTrackbar(
  const string&        trackbarName,      // Handle used to identify trackbar
  const string&        windowName,        // Handle used to identify window
  int*                 value,             // Slider position gets put here
  int                  count,             // Total counts for slider at far right
```

```
    cv::TrackbarCallback onChange    = NULL,// Callback function (optional)
    void*                param       = NULL // Additional params for callback fn.
  );
```

第一个参数是滑动条的名称，第二个参数是滑动条将要添加到的父窗口的名称。一旦滑动条创建好，它将被添加到窗口的顶部或底部。[注7]滑动条不会挡住任何已经在窗口中的图像，但它可能会让窗口稍稍变大一点。窗口的名将作为窗口的一个标记。至于滑动条上滑动钮的确切位置，也是由操作系统来决定。但大多数情况下都是在最左边，如图9-2所示。

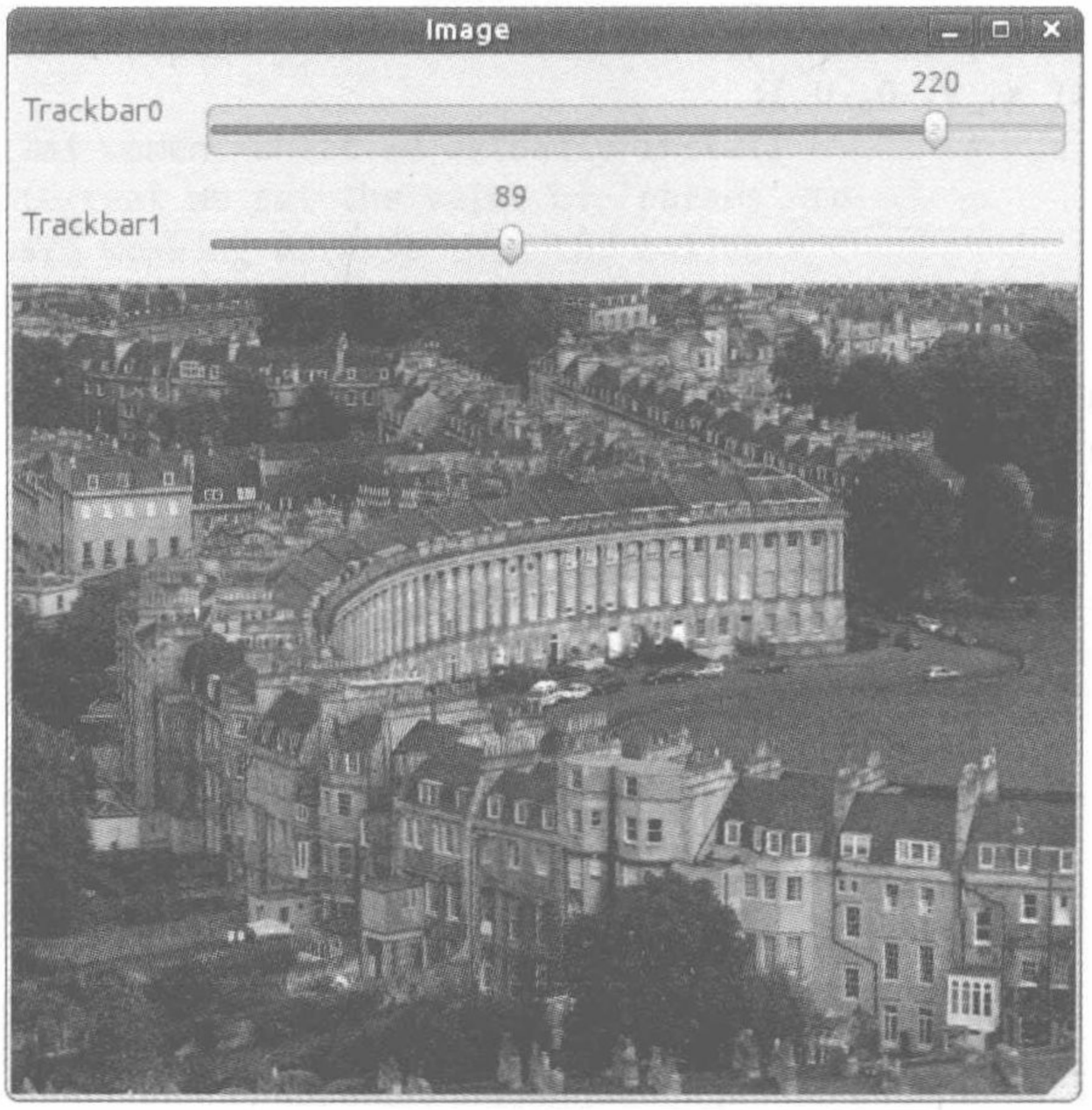


图9-2. 一个显示图像的简单例子

接下来有两个参数。一个是value，是一个指向整数的指针，这个整数值会随着滑动钮的移动自动变化。另一个Count是滑动滑动条可以滑动的最大值。

最后一个参数是一个指向回调函数的指针，当滑动钮移动时，回调函数就会自动调用。这是对鼠标事件的一种精确模拟。回调函数必须要有cv::TrackbarCallback的特定形式，也就是说它必须满足以下定义：

```
void your_trackbar_callback(
  int   pos,                          // Trackbar slider position
  void* param = NULL                  // Parameters from cv::setTrackbarCallback()
);
```

注7：滑动块添加的位置是顶部还是底部，由操作系统来决定，在同一个平台上，它通常只出现在一个位置。

这个回调函数不是必须的，因此，如果不想要这个回调函数，可以直接将这个参数设置为NULL。没有回调函数，移动滑动钮的唯一响应就是value指向的变量值的变化。（如果没有回调函数，可以通过获取这个值来对值的变化做出响应。）

cv::createTrackbar()最后一个参数是params，可以是一个任何类型的指针，一旦回调函数执行，这个参数可以传递给回调函数的params参数。这样做很有用，因为这样一来，不创建全局变量也可以处理滑动条事件。

最后，有两个函数可以用来获取和设置滑动条的位置值。

```
int cv::getTrackbarPos(
  const string& trackbarName,          // Handle used to identify trackbar, label
  const string& windowName,            // Handle used to identify window
);

void cv::setTrackbarPos(
  const string& trackbarName,          // Handle used to identify trackbar, label
  const string& windowName,            // Handle used to identify window
  int   pos                            // Trackbar slider position
);
```

通过这两个函数，你可以在程序的任何位置获取或设置滑动条的值。

不使用按键

不幸的是，基础的HighGUI并不提供对按键功能的支持。在日常使用中，有一种通用的小伎俩，就是将滑动条限制为只有两个位置。[注8]还有一种技巧，如.../*opencv/samples/c/*所使用的那样，使用键盘事件来代替按键（参阅在OpenCV源码中的*floodfill*例程）。

开关也是只有两个位置的滚动条。当它处于“开”状态的时候为1，处于“关”状态的时候为0（比如count变量被设置为0）。你可以发现，使用滚动条来模拟按键行为是一件非常轻松的事情。根据你期望的开关行为的不同，你可以使用滚动条的回调函数来将按键设置为0（如同在示例9-3中，这是和大多数GUI中的button有着相同的行为），或是自动设置其他开关为0（如同checkbox控件一样的行为，如果愿意做更多的工作，它可以像radio button那样）。

注8：另一种稍微复杂一些的操作是，将你需要显示的图像和一个假的“控制面板”相结合，并用鼠标回调事件进行相应的响应。当点击的(x, y)坐标处于你所绘制的按钮区域时，回调函数将做出按钮被按下那样的行为。通过这样的方法，所有的“按钮”将统一由鼠标回调函数处理，并且集成到显示窗口中。但是实际上，如果你希望实现类似的功能，我们更推荐你使用Qt进行实现。

示例9-3：使用滚动条来创建一个“开关”，使得用户一打开开关，程序就自动播放视频并且允许用户使用开关来播放和暂停

```
// An example program in which the user can draw boxes on the screen.
//
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

//
// Using a trackbar to create a "switch" that the user can turn on and off.
// We make this value global so everyone can see it.
//
int g_switch_value = 1;
void switch_off_function() { cout << "Pause\n"; }; //YOU COULD DO MORE
void switch_on_function()  { cout << "Run\n"; };

// This will be the callback that we give to the trackbar.
//
void switch_callback( int position, void* ) {
  if( position == 0 ) {
    switch_off_function();
  } else {
    switch_on_function();
  }
}

void help() {
    cout << "Call: my.avi" << endl;
    cout << "Shows putting a pause button in a video." << endl;
}

int main( int argc, char** argv ) {

  cv::Mat frame; // To hold movie images
  cv::VideoCapture g_capture;
  help();
  if( argc < 2 || !g_capture.open( argv[1] ) ){
    cout << "Failed to open " << argv[1] << " video file\n" << endl;
    return -1;
  }

  // Name the main window
  //
  cv::namedWindow( "Example", 1 );

  // Create the trackbar. We give it a name,
  // and tell it the name of the parent window.
  //
  cv::createTrackbar(
    "Switch",
    "Example",
    &g_switch_value,
    1,
```

```
  switch_callback
);

// This will cause OpenCV to idle until
// someone hits the Esc key.
//
for(;;) {
  if( g_switch_value ) {
        g_capture >> frame;
        if( frame.empty() ) break;
        cv::imshow( "Example", frame);
    }
    if( cv::waitKey(10)==27 ) break;
  }

  return 0;
}
```

通过Qt后端工作

如前所述，OpenCV中基于HighGUI部分开发任何GUI功能都要依赖独立的库。这样是有意义的，因为OpenCV并不想“造轮子”（重复底层的工作），而且这里已经有大量由优秀团队管理、维护和更新的GUI工具包。

我们目前所看到的基本的GUI工具包中的基础功能都是基于不同平台上最原始的库封装而来的，因此作为开发者，看不到这些封装前的库。这样原本是很方便，很好的，但在移动平台支持之前，这些库的维护开始变得有些困难。基于以上原因，向具有承载以前那些基本功能的独立跨平台库的转变日益增强，这个跨平台库就是Qt。

从OpenCV库的角度，使用独立的第三方库有很大的优势。我们不但可以有以前库的功能，还可以缩短开发的时间（否则这点很可能会从库的核心目标中去除）。

这一节中我们将学习使用基于Qt的HighGUI接口。将来HighGUI的更新演化将很有可能发生在这个组成部件中，因为使用基于Qt的HighGUI接口的开发比之前遗留接口的开发效率要高很多。

需要注意的是，使用Qt作为后端的HighGUI和直接使用Qt是不同的（本章最后将简要介绍）。HighGUI接口依然是HighGUI接口，只是使用Qt替代了其他各种各样的原始库。但这样做的缺点是使扩展Qt接口不再方便。如果需要获得（比HighGUI提供的）更多的功能，那么你可能更喜欢自己写窗口。好的一方面是，Qt的接口可以提供丰富的功能，这样一来，即使自己写窗口也不那么繁琐和麻烦。[注9]

注9：你会发现基于Qt的HighGUI接口主要是面向实验室开发需求和调试系统。如果你是开发面向用户的商业化代码，可能需要功能更强大、表现力更强的UI库。

开始

如果在编译OpenCV时选择了Qt支持，[注10]那么，当你打开一个窗口时，它会自动包括两个新的特性。这两个新特性分别是工具栏和状态栏（图9-3）。这些控件都是完整的，图中展示了所有的控件内容。具体而言，这个工具栏中包括一些按钮能够让你“平铺”（前四行的按钮），缩放图像（后四行按钮），保存当前图像（第9个按钮），弹出属性窗口（稍后详细介绍）。

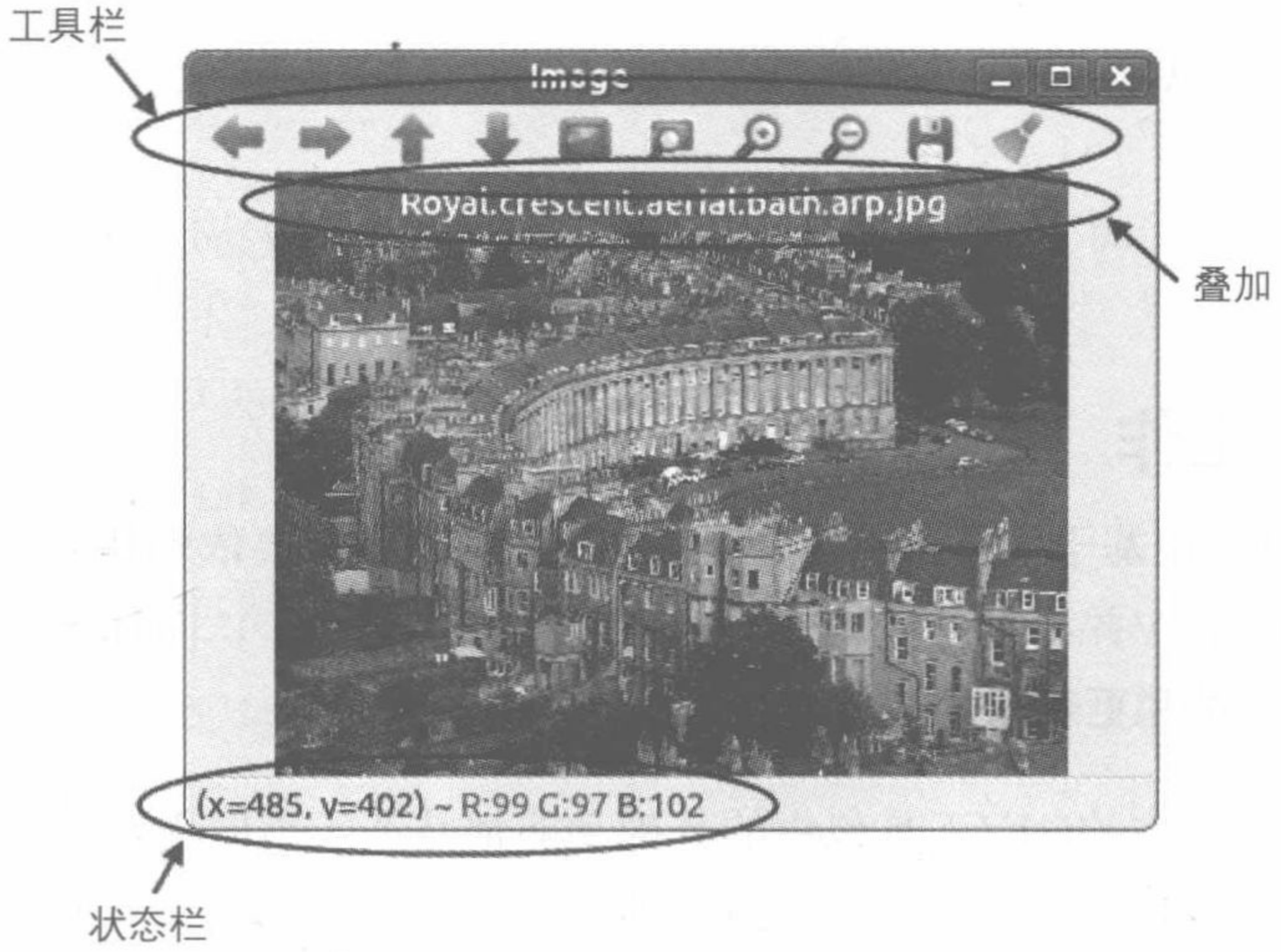


图9-3. 基于Qt支持的图像显示，其包含工具栏、状态栏以及包含图像名的文字蒙版

状态栏（图9-3）中包括鼠标在任何给定瞬间的信息。有x,y坐标，鼠标光标当前位置点的RGB值等。

一旦使用Qt接口支持编译代码，这些信息都会自动添加到窗口上。如果你已经使得Qt接口支持，但又不想要工具栏和状态栏等，你可以在调用cv::namedWindow()时增加一个输入标志cv::GUI_NORMAL。[注11]

操作菜单

可以看到，当你在使用cv::GUI_EXTENDED创建窗口时，工具栏上会有一系列按钮。弹出菜单栏可以作为工具栏的替代方案，并且可以简单通过使用cv::GUI_EXTENDED来设定。弹出菜单栏（图9-4）中的选项和工具栏相同，可以通过在图像上通过鼠标右击让它显示出来。

注10：意味着你在用cmake编译配置时使用了-D WITH_QT=ON选项。

注11：当然，这里也有一个CV_GUI_EXTENDED标志是用来产生状态栏和工具栏的，但这个标志的数值是0x00，是一个默认值。

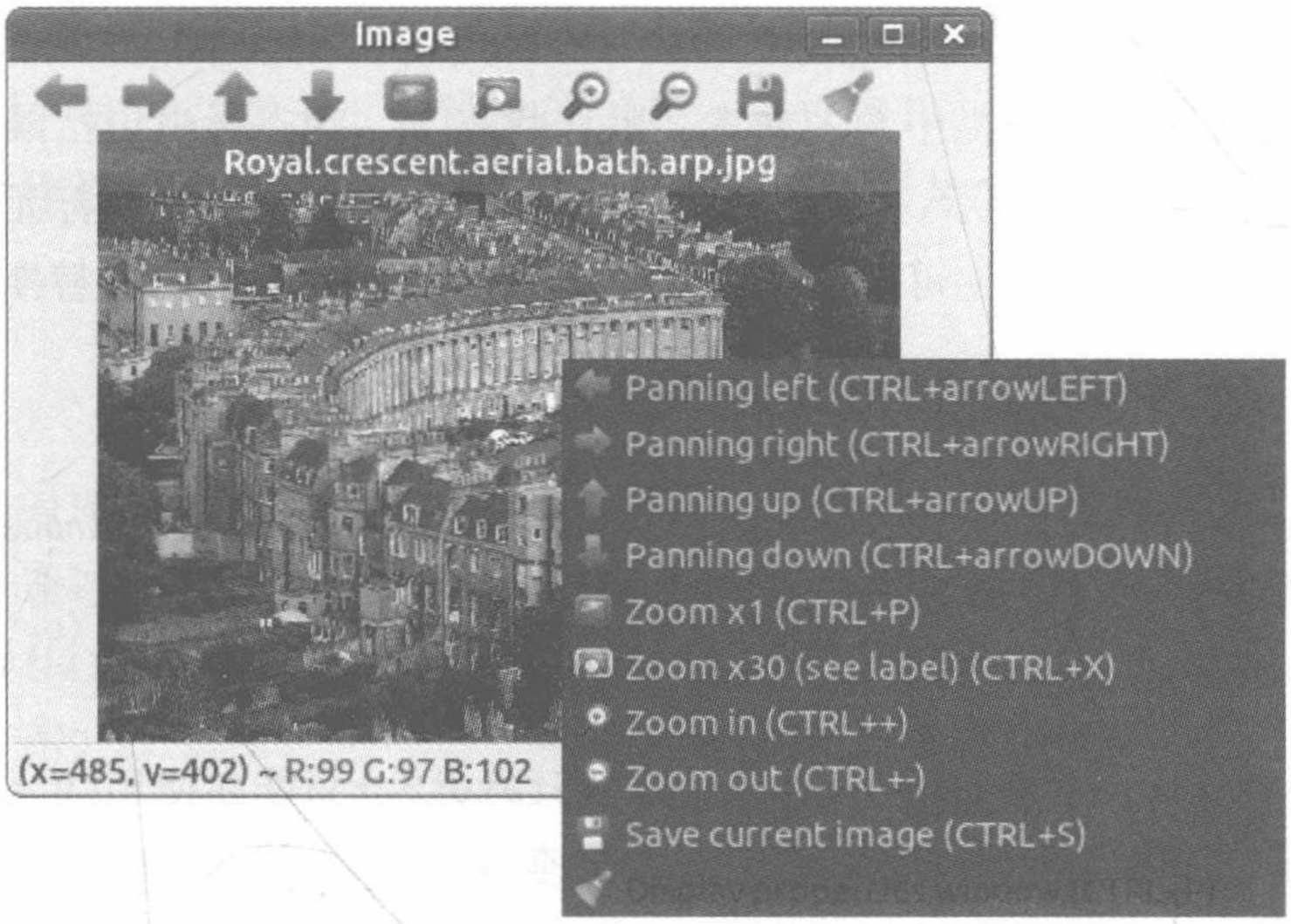


图9-4. 有弹出菜单栏的Qt扩展的UI窗口，提供的选项和工具栏相同（包括按钮的解释和快捷键）

文本蒙版

Qt的另一个可选项是能够放一个临时的条幅到展示的图像顶部。这个条幅也叫“蒙板”，在图像上展示时，它周围有一个便于阅读的带阴影的框。这个特性可以用于展示一些简单的信息，比如在视频上展示视频帧的编号和帧率或查看的图像名称。即使你使用了cv::GUI_NORMAL标志参数，也可以在窗口上展示蒙板。

```
int cv::displayOverlay(
  const string& name,                    // Handle used to identify window
  const string& text,                    // Text you want to display
  int           delay                    // Milliseconds to show text (0='forever')
);
```

cv::displayOverlay()有三个参数。第一个参数是添加蒙板的窗口之名称。第二个参数是需要展示的所有文本。需要注意的是，文本框的长度是固定的，因此如果一次性写入太多内容可能会出现文本溢出[注12]。默认情况下，文本居中放置。最后一个参数delay，是蒙板出现的窗口的时间。如果delay的值设置为0，那么蒙板将一直存在或直到你再次调用cv::displayOverlay()覆盖原来的蒙板。通常，在蒙板展示延迟结束前调用cv::displayOverlay()时，蒙板原来的文本内容将替换为新的，无论原来的蒙板延迟有没有结束，延迟时间都会被重置为开始。

注12：但是，可以插入新行，例如，如果给文本框的字符串是"Hello\nWorld"，*Hello*会出现在第一行，*World*紧随其后，出现在下一行。

状态栏中写入自己的内容

除了蒙板，你还可以在状态栏中写入文本。默认状态下，状态栏中包括鼠标光标所在位置的像素的信息。如图9-3所示，在展示一张图像时，状态栏中包括鼠标光标所在位置像素的坐标和像素的RGB值。你可以通过cv::displayStatusBar()函数替换状态栏中文本的值：

```
int cv::displayStatusBar(
  const string& name,                 // Handle used to identify window
  const string& text,                 // Text you want to display
  int           delay                 // Milliseconds to show text (0='forever')
);
```

与cv::displayOverlay不同的是，cv::displayStatusBar只能在使用cv::GUI_EXTENDED标志创建窗口时调用，并且在文本显示延迟时间到了后（如果delay没有被置为0），状态栏会自动恢复默认的鼠标光标位置和光标所在像素RGB值的显示。

属性窗口

到目前，我们没有讨论过菜单栏上的最后一个按钮（对应于弹出菜单的最后一个加暗的选项，如图9-5所示）。这个选项打开了一个新的窗口，叫“属性窗口”。如果你不想按钮和滑动条一直出现在眼前，属性窗口很适合用来放置滑动条和按钮（Qt GUI支持按钮）。需要注意的是，每一个应用只有一个属性窗口，因此，你不是真的创建它们而只是配置它们。

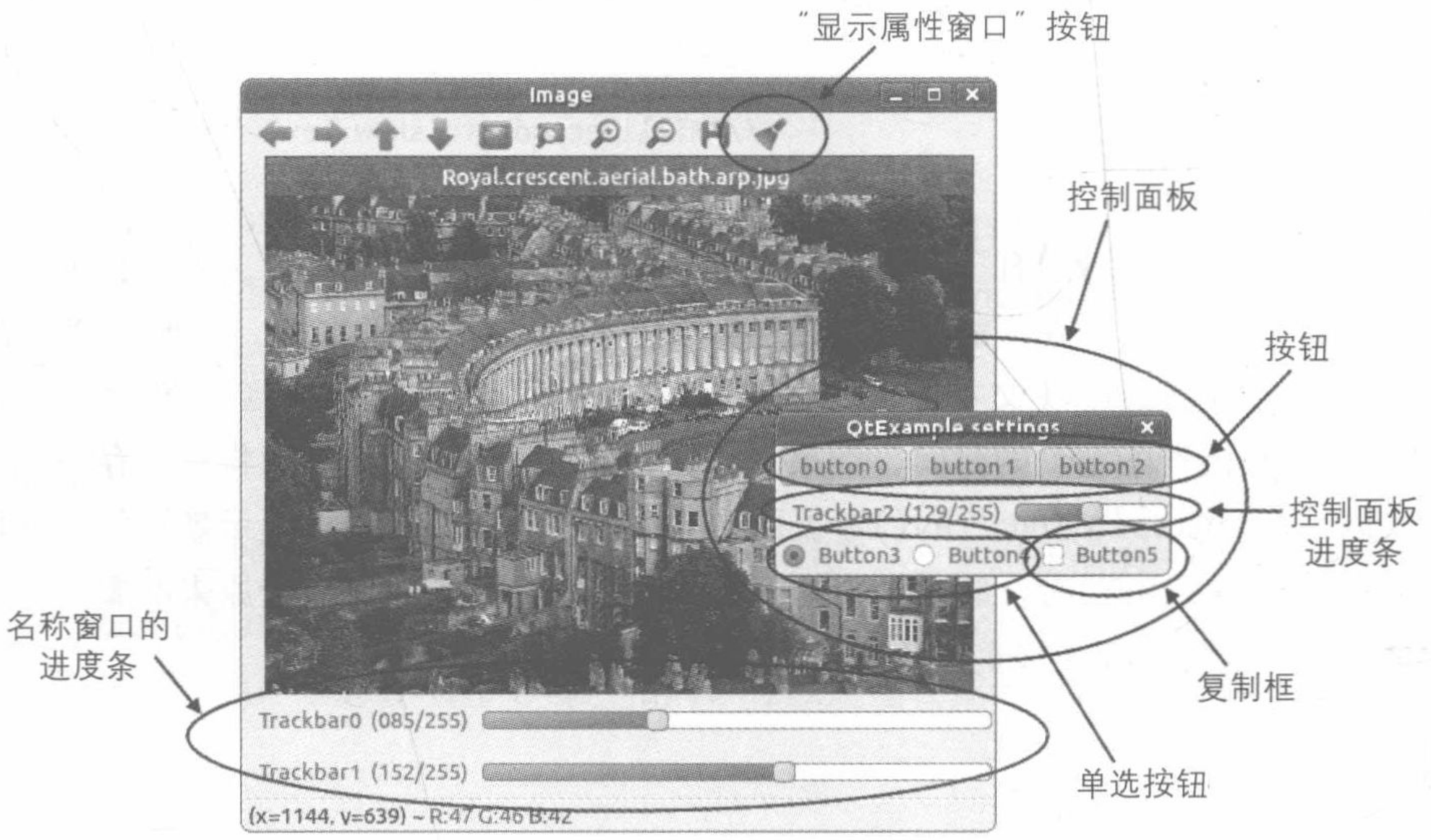


图9-5. 该窗口包含两个滑动条，还有包含三个按钮的控制面板、一个滚动条以及两个单选按钮和一个复选框

只有给属性窗口分配按钮和滑动条（而不仅仅只是显示出来），属性窗口才是可用的。当窗口可用时，你可以点击工具栏上的“显示属性窗口”按钮来显示它（最右边的按钮），或者操作栏上的相似按钮，再或者当鼠标停留在某个菜单上时用Ctrl+P快捷键组合调出。

再述滑动条

在前面的小节对HighGUI底层接口的叙述中，我们看到可以添加滑动条到窗口中。图9-5中的滑动条是通过我们之前看到的同一个函数命令cv::createTrackbar()创建的。相比我们之前使用非Qt接口创建的滑动条，图9-5中这个滑动条唯一的区别就是看起来更加漂亮（回忆一下图9-2）。

Qt接口中一个重要的概念是我们可以在属性窗口中创建滑动条。我们使用常规的方法创建一个滑动条，但将滑动条要添加的窗口的名称参数设置为一个空字符串。

```
int contrast = 128;
cv::createTrackbar( "Contrast:", "", &contrast, 255, on_change_contrast );
```

例如，以上程序片段将创建一个标签为“Contrast”的属性窗口。滑动条的值将从128开始，最大值是255。一旦滑动条发生改变，回调函数on_change_contrast()就会被调用。

通过cv::createButton()创建按钮

Qt接口提供的一个最有用的新功能是可以创建按钮。这些按钮包括正常的按钮、单选按钮和复选框。所有创建的按钮都位于控制面板上。

所有三种类型的按钮都通过同一个方法完成：

```
int cv::createButton(
  const string&      buttonName,          // Handle used to identify trackbar
  cv::ButtonCallback onChange     = NULL, // Callback for button event
  void*              params,              // (Optional) params for button event
  int                buttonType   = cv::PUSH_BUTTON,  // PUSH_BUTTON or RADIOBOX
  int                initialState = 0     // Start state of the button
);
```

按钮需要有名称buttonName，将出现在按钮上或按钮的旁边。也可以不指定名称，忽略这参数，只提供一个空串，则按钮名会按序出现（例如，“button0”，“button1”，……）。第二个参数onChange是一个回调函数，按钮点击时调用。这个回调函数的定义必须满足cv::ButtonCallback的定义：

```
void your_button_callback(
  int   state,                          // Identifies button event type
  void* params                          // Parameters from cv::createButton()
);
```

当点击按钮时，回调函数将被调用，state参数就＝会被赋值，state的值由点击按钮时产生。你传递给cv::createButton()的params指针也会被传给回调函数的params参数。

buttonType可以取如下三个值之一：cv::PUSH_BUTTON，cv::RADIOBOX或者cv::CHECKBOX。第一个是标准按钮，当你点击按钮便会调用回调函数。在复选框（checkbox）模式下，按钮值参数根据有没有选择复选框设置为1或者0。单选按钮radio模式下也与此相同，区别在于点击按钮时同时调用当前点击按钮的回调函数和当前没有被点击的按钮（收音机风格按钮的互斥状态）。在同一行的按钮称为按钮栏，接下来我们会介绍将按钮栏上的按钮归为同一互斥组的一部分。

按钮创建后会被自动组织成按钮栏。按钮栏就是属性窗口上的一行按钮。下面的代码创建了图9-5上的控制面板：

```
cv::namedWindow( "Image", cv::GUI_EXPANDED );
cv::displayOverlay( "Image", file_name, 0 );
cv::createTrackbar( "Trackbar0", "Image", &mybar0, 255 );
cv::createTrackbar( "Trackbar1", "Image", &mybar1, 255 );

cv::createButton( "", NULL, NULL, cv::PUSH_BUTTON );
cv::createButton( "", NULL, NULL, cv::PUSH_BUTTON );
cv::createButton( "", NULL, NULL, cv::PUSH_BUTTON );
cv::createTrackbar( "Trackbar2", "", &mybar1, 255 );
cv::createButton( "Button3", NULL, NULL, cv::RADIOBOX, 1 );
cv::createButton( "Button4", NULL, NULL, cv::RADIOBOX, 0 );
cv::createButton( "Button5", NULL, NULL, cv::CHECKBOX, 0 );
```

你可能注意到创建的Trackbar0和Trackbar1的名字是"Image"，Trackbar2创建在一个没有命名的窗口上（属性窗口）。前三个cv::createButton()调用没有给按钮命名，可以从图9-5上看到，自动命名的名称会在添加到按钮上。从图9-5可以注意到，前三个按钮是出现在一行，而另一组的三个按钮出现在另一行，因为有一个滑动条的缘故。

按钮一个接一个创建，从左到右依次成行排列，因为滑动条单独占据一行，所以一旦出现一个滑动条，接下来的按钮便会另起一行排列。[注13]

文本和字体

正如Qt接口可以提供更加美观的滑动条和其他一些元素，Qt还可以提供更美观和灵活的本文样式。使用Qt接口写文本必须首先创建一个CvFont对象。[注14]任何时候想在屏幕中添加文本时都可以使用这个对象。字体通过函数cv::fontQt()创建：

注13：遗憾的是，这里没有按钮排列的回车符。

注14：你可能注意到，这个对象的名称是CvFont而不是cv::Font。这是早期C++遗留接口的一种形式。CvFont是一个结构体并且不在cv命名空间中。

```
CvFont fontQt(                                  // Return font characterization struct
  const string& fontName,                       // e.g., "Times"
  int           pointSize,                      // Size of font, using "point" system.
  cv::Scalar    color    = cv::Scalar::all(0), // BGR color as scalar (no alpha)
  int           weight   = cv::FONT_NORMAL,    // Font weight, 1-100 (Table 9-3)
  int           spacing  = 0                   // Space between individual characters
);
```

fontQt的第一个参数是系统的字体名称，例如“Times”，如果系统中没有这个名称，系统会给你分配一个默认的字体。第二个参数是字体的大小（如12 ="12 point," 14 ="14 point"等）。可以将第二个参数设置为0，系统会自动将字体大小设置为默认值12。

显示文本时，字体颜色参数可以被设置为任何cv::Scalar（BGR）值。默认颜色是黑色。如表9-3所示，字体粗细可以设置为任何提前预定义的值或者1到100之间的任何整数。预定义值名和它们的值如Qt字体粗细预定值和定义名所示。

表9-3：Qt字体粗细的预定义名称及与其关联的值

Camera capture constant	数值
cv::FONT_LIGHT	25
cv::FONT_NORMAL	50
cv::FONT_DEMIBOLD	63
cv::FONT_BOLD	75
cv::FONT_BLACK	87

最后一个参数spacing控制字符间距，取值正负都可以。

定义好了字体以后，就可以在图像上（也就是屏幕上）放置文本了。[注15]

```
void cv::addText(
  cv::Mat&      image,                  // Image onto which to write
  const string& text,                   // Text to write
  cv::Point     location,               // Location of lower-left corner of text
  CvFont*       font                    // OpenCV font characerization struct
);
```

cv::addText的参数有：写上文本的图像image、文本内容text、写文本的位置 location和文本的字体font。font可以通过cv::fontQt函数定义。文本的位置为首字符左下角的位置（更精确来讲，是文本基准线的起点。）

注15：注意，cv::addText()和Qt其余的接口都不一样，它和非Qt接口的类似函数cv::putText()没有保持一致性。具体来说，cv::addText()并没有将文本放在窗口上而是放在图像上的。这意味着改变了图像上像素的值，例如，这一点和使用cv::displayOverlay()产生的效果不同。

设置和获取窗口属性

大部分基于Qt后端创建的窗口的状态属性都是可以获取的，并且大部分都能够在创建窗口后更改。

```
void   cv::setWindowProperty(
  const string& name,                    // Handle used to identify window
  int           prop_id,                 // Identifies window property (Table 9-4)
  double        prop_value               // Value to which to set property
);
```

```
double cv::getWindowProperty(
  const string& name,                    // Handle used to identify window
  int           prop_id                  // Identifies window property (Table 9-4));
```

要获取窗口的属性，调用cv::setWindowProperty()函数并提供窗口名和属性ID（prop_id）就可以了。同样，可以使用cv::setWindowProperty()函数根据属性ID设置属性（表9-4）。

表9-4：可获取和设置的窗口属性

属性名称	描述
cv::WIND_PROP_FULL_SIZE	设置为cv::WINDOW_FULLSCREEN，可将窗口显示为全屏；设置为cv::WINDOW_NORMAL，窗口显示为正常尺寸
cv::WIND_PROP_AUTOSIZE	设置为v::WINDOW_AUTOSIZE，可以使窗口大小自动调整为显示图像的大小；设置为cv::WINDOW_NORMAL，将显示图像大小调整为窗口大小
cv::WIND_PROP_ASPECTRATIO	设置为cv::WINDOW_FREERATIO，可以使窗口有任意长宽比（用户可调整窗口尺寸）；设置为cv::WINDOW_KEEPRATIO，只能按照固定长宽比调整窗口大小（用户不能调整长宽比）

保存和恢复窗口的状态

Qt接口可以保存和恢复窗口的状态。这样做可以带来很多方便，因为窗口属性中包括了窗口的位置、大小和所有滑动条与按钮的状态。接口状态可以用cv::saveWindowParameters()函数进行保存，该函数只需要窗口的名称作为输入参数：

```
void cv::saveWindowParameters(
  const string& name                     // Handle used to identify Windows
);
```

一旦窗口属性保存好，就可以用与之互补的函数cv::loadWindowParameters()来恢复：

```
void cv::loadWindowParameters(
  const string& name                     // Handle used to identify window
);
```

比较奇特的一点是，即使你退出或重启程序，属性加载命令依然能够正常工作。加载命令能够正常工作的原理对我们来说不重要，但一个需要注意的细节是：状态信息无论保存到哪里，都是通过关键字来保存的，而这个关键字是通过一个可执行文件名构造的，因此，如果你改变了可执行文件名，状态信息就不会保存（然而，改变可执行文件名的位置并不会造成这个问题）。

和OpenGL交互

使用Qt接口的一个令人激动地方是，我们可以使用OpenGL生成图像和放置于图像顶部的蒙板。[注16]这样可以很高效地可视化调试机器人或者增强现实的应用或者你想通过一些图像创建一个三维模型和在原始基础上观测结果。图9-6展示了一个可以用OpenGL来完成的简单例子。

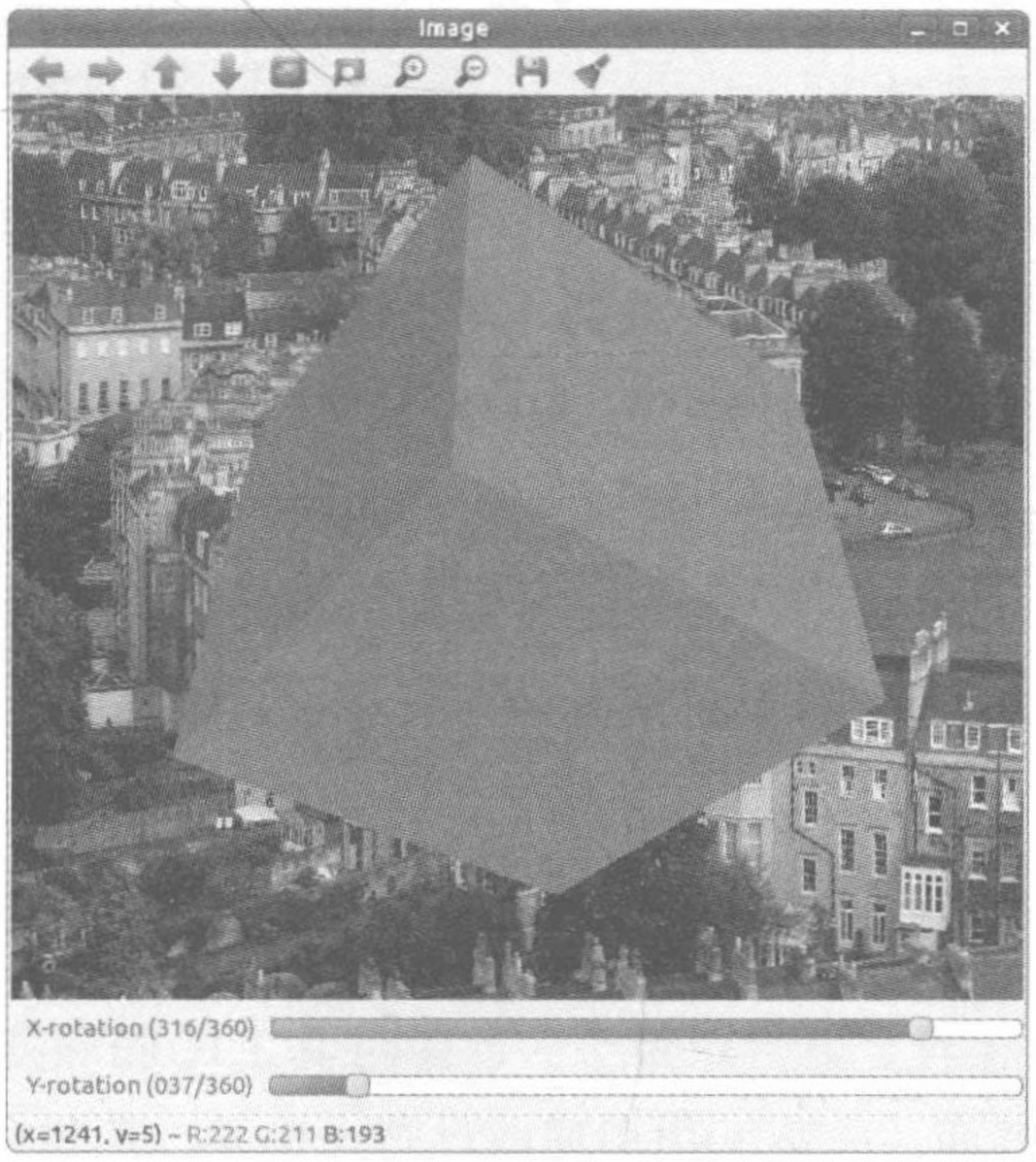


图9-6. 在以往图像的基础上通过OpenGL渲染立方体

基本概念非常简单，就是创建一个OpenGL画图的回调函数并通过接口注册它。接下来OpenCV来完成剩余的内容。每次窗口显示时都会调用这个回调函数（包括使用cv::imshow()时，在显示视频连续帧时）。你的回调函数需要匹配cv::OpenGLCallback()的定义，也就是说需要满足以下定义：

```
void your_opengl_callback(
  void* params                  // (Optional) Params from cv::createOpenGLCallback()
);
```

注16：为了能够使用这些命令，你需要在编译OpenCV时配置CMake选项-D WITH_QT_OPENGL=ON。

一旦你创建了回调函数，就可以通过cv::createOpenGLCallback()配置回调函数。

```
void cv::createOpenGLCallback(
  const string&      windowName,          // Handle used to identify window
  cv::OpenGLCallback callback,            // OpenGL callback routine
  void*              params    = NULL     // (Optional) parameters for callback
);
```

可以看到，我们不需要执行太多步骤就可以完成OpenGL的交互使用。除了需要用于画图时窗口的名称和回调函数外，还有一个params参数，这个参数可以在调用回调函数时给回调函数传递额外的参数。



值得注意的是，这里不会设置相机、光线或者OpenGL各项活动的其他方面。对OpenGL回调函数的内部封装可以通过调用gluPerspective()函数来设置射影矩阵。如果想获得不一样的效果（可能经常需要），需要在回调函数前清除和配置射影矩阵。

在示例9-4中，我们从OpenCV文档中找到一个用OpenGL画一个立方体的简单例子，但我们将固定的旋转角修改为变量（rotx，roty），并将它们作为之前例子中两个滑动条的值，修改版用两个全局变量rotx和roty与图9-6中两个滑动条发生关联，这样一来，用户就可以通过拖动滑动条来调整立方体的旋转角度。

示例9-4：简单修改OpenCV文档中用来画一个立方体的代码

```
void on_opengl( void* param ) {

  glMatrixModel( GL_MODELVIEW );
  glLoadIdentity();

  glTranslated( 0.0, 0.0, -1.0 );

  glRotatef( rotx, 1, 0, 0 );
  glRotatef( roty, 0, 1, 0 );
  glRotatef( 0, 0, 0, 1 );

  static const int coords[6][4][3] = {
    { { +1, -1, -1 }, { -1, -1, -1 }, { -1, +1, -1 }, { +1, +1, -1 } },
    { { +1, +1, -1 }, { -1, +1, -1 }, { -1, +1, +1 }, { +1, +1, +1 } },
    { { +1, -1, +1 }, { +1, -1, -1 }, { +1, +1, -1 }, { +1, +1, +1 } },
    { { -1, -1, -1 }, { -1, -1, +1 }, { -1, +1, +1 }, { -1, +1, -1 } },
    { { +1, -1, +1 }, { -1, -1, +1 }, { -1, -1, -1 }, { +1, -1, -1 } },
    { { -1, -1, +1 }, { +1, -1, +1 }, { +1, +1, +1 }, { -1, +1, +1 } }
  };

  for (int i = 0; i < 6; ++i) {
    glColor3ub( i*20, 100+i*10, i*42 );
    glBegin(GL_QUADS);
```

```
    for (int j = 0; j < 4; ++j) {
      glVertex3d(
        0.2 * coords[i][j][0],
        0.2 * coords[i][j][1],
        0.2 * coords[i][j][2]
      );
    }
    glEnd();
  }
}
```

综合OpenCV和全功能GUI工具包

尽管OpenCV内建的Qt接口对开发算法和开发的简单任务是一个方便的解决方案，但我们开发面向用户的代码时，原始的UI库和基于Qt的接口都无法满足开发要求。本节将非常简略地介绍使用OpenCV的一些问题和技巧以及三个工具包：Qt，wxWidgets和Windows Template Library（WTL）。

UI工具包有很多，我们不需要对每一个都进行详尽的探究。如前所述，探索OpenCV和特性全面的工具包使用中可能出现的问题会更加有用。我们这里探索的少量问题可以给你提供足够的启发，使你在遇到类似的问题不至于感到困惑。

首先，一个问题就如何将OpenCV图像转换为工具包对图形所期望的形式，并且知道工具包中哪个窗口小部件或组件能够用来完成显示工作。从那儿开始，你就不需要很多OpenCV特有的功能。值得注意的是，你将不需要或想要本章中已经介绍过的UI工具包的功能。

OpenCV和Qt的一个例子

这里我们将展示使用实际的Qt工具包编写一个读取视频文件并将其显示在屏幕上的程序示例。这里介绍几个细节，其中一些与如何使用Qt有关，而其他与OpenCV有关。当然，我们会重点关注后者，但还是值得花些时间来关注前者如何影响我们目前的目标。

示例9-5显示了程序的顶层代码，它创建了一个Qt应用程序，并添加了我们的QMoviePlayer小部件。该程序只有唯一一个指示视频文件的参数；视频文件将在我们定义的Qt对象内播放，称之为QMoviePlayer。一切有趣的事情都将发生在那个对象内。

示例9-5：ch4_qt.cpp的一个例子

```
#include <QApplication>
#include <QLabel>
#include <QMoviePlayer.hpp>
int main( int argc, char* argv[] ) {

  QApplication app( argc, argv );
```

```
  QMoviePlayer mp;
  mp.open( argv[1] );
  mp.show();

  return app.exec();
}
```

有意思的东西是这个QmoviePlayer对象。让我们看看示例9-6中定义该对象的头文件。

示例9-6：QMoviePlayer 对象头文件QMoviePlayer.hpp

```
#include "ui_QMoviePlayer.h"
#include <opencv2/opencv.hpp>
#include <string>

using namespace std;

class QMoviePlayer : public QWidget {

  Q_OBJECT;

  public:
  QMoviePlayer( QWidget *parent = NULL );
  virtual ~QMoviePlayer() {;}

  bool open( string file );

  private:
  Ui::QMoviePlayer  ui;
  cv::VideoCapture m_cap;

  QImage  m_qt_img;
  cv::Mat m_cv_img;
  QTimer* m_timer;

  void paintEvent( QPaintEvent* q );
  void _copyImage( void );

  public slots:
  void nextFrame();

};
```

这里有很多东西。第一个就是包含文件*ui_QMoviePlayer.h*，该文件由Qt Designer自动生成。重要的一点是，它只是一个QWidget，它只包含一个称为框架的QFrame。私有变量ui::QMoviePlayer是在*ui_QMoviePlayer.h*中定义的接口对象。

在这个文件中，还有一个名为m_qt_img的QImage和一个名为m_cv_img的cv::Mat。这些将包含我们从视频中获取的图像的Qt和OpenCV表示。最后，有一个QTimer，它将取代cv::waitKey()，让我们以正确的速度播放视频帧。看一看*QMoviePlayer.cpp*（示例9-7）中的实际定义，就会明白其他函数的意义。

示例9-7：QMoviePlayer对象源码文件QMoviePlayer.cpp

```
#include "QMoviePlayer.hpp"
#include <QTimer>
#include <QPainter>

QMoviePlayer::QMoviePlayer( QWidget *parent )
 : QWidget( parent )
{
  ui.setupUi( this );
}
```

QMoviePlayer的顶层构造函数只调用安装函数，这是为UI成员自动构建的。

```
bool QMoviePlayer::open( string file ) {

  if( !m_cap.open( file ) ) return false;

  // If we opened the file, set up everything now:
  //
  m_cap.read( m_cv_img );
  m_qt_img = QImage(
    QSize( m_cv_img.cols, m_cv_img.rows ),
    QImage::Format_RGB888
  );
  ui.frame->setMinimumSize( m_qt_img.width(), m_qt_img.height() );
  ui.frame->setMaximumSize( m_qt_img.width(), m_qt_img.height() );
  _copyImage();

  m_timer = new QTimer( this );
  connect(
    m_timer,
    SIGNAL( timeout() ),
    this,
    SLOT( nextFrame() )
  );
  m_timer->start( 1000. / m_cap.get( cv::CAP_PROP_FPS ) );

  return true;

}
```

在QMoviePlayer上调用open()时，会经过几个过程。第一个是需要打开cv::VideoCapture成员对象m_cap。如果打开失败，就会返回。接下来，我们将第一帧读入我们的OpenCV图像成员m_cv_img。一旦有了OpenCV图像，我们就可以设置Qt图像对象m_qt_img，使其与OpenCV图像具有相同的大小。之后，我们将UI元素中的图像帧对象调整为与输入图像相同的大小。

稍后我们将看看对QMoviePlayer::_copyImage()的调用，这将完成一个非常重要有过程——将捕获的图像m_cv_img转换为Qt图像m_qt_img，我们实际上会用Qt绘制图像到屏幕上。

我们在QMoviePlayer::open()中要做的最后一件事是设置一个QTimer，这样一来，当它“关闭”时，它将调用QMoviePlayer::nextFrame()函数（这不奇怪，这样可得到下一帧图像）。对m_timer->start()的调用是表明我们如何启动定时器运行，并指示它以cv::CAP_PROP_FPS所指代的正确速率（即1000毫秒除以帧速率）进行。

```
void QMoviePlayer::_copyImage( void ) {

  // Copy the image data into the Qt QImage
  //
  cv::Mat cv_header_to_qt_image(
    cv::Size(
      m_qt_img.width(),
      m_qt_img.height()
    ),
    CV_8UC3,
    m_qt_img.bits()
  );
  cv::cvtColor( m_cv_img, cv_header_to_qt_image, cv::BGR2RGB );

}
```

QMoviePlayer::_copyImage()函数负责将图像从缓冲区m_cv_img复制到Qt图像缓冲区m_qt_img中。我们这样做显示了cv::Mat图像的一个很好的功能。首先，我们定义一个名为cv_hreader_to_qt_image的cv::Mat对象。定义该对象时，我们实际上是告诉它要为其数据区使用哪一块内存区域，并将内存区传递给Qt的QImage对象m_qt_img.bits()。然后我们调用cv::cvtColor进行复制，对于彩色图像数据，OpenCV倾向于使用BGR排序，而Qt则喜欢RGB。

```
void QMoviePlayer::nextFrame() {

  // Nothing to do if capture object is not open
  //
  if( !m_cap.isOpened() ) return;

  m_cap.read( m_cv_img );
  _copyImage();

  this->update();

}
```

QMoviePlayer::nextFrame()函数用来处理后续帧的读取。回想一下，当QTimer结束后，这个程序将被调用。它将新图像读入OpenCV缓冲区，调用QMoviePlayer::_copyImage()将它复制到Qt缓冲区中；然后在QWidget上进行一个更新调用（以便让Qt知道某些事情发生了变化），这部分是所有都有的。

```
void QMoviePlayer::paintEvent( QPaintEvent* e ) {
```

```
    QPainter painter( this );
    painter.drawImage( QPoint( ui.frame->x(), ui.frame->y()), m_qt_img );
  }
```

最后一项重要的是QMoviePlayer::paintEvent()函数。只要需要绘制QMoviePlayer小部件，就会被Qt调用。该函数只创建一个QPainter，并指示它绘制当前的Qt图像m_qt_img（从屏幕的一角开始）。

OpenCV和wxWidgets的例子

在示例9-8中，我们将使用一个不同的跨平台工具包wxWidget。该程序只有唯一一个指示视频文件的参数，视频文件将在我们定义的wxWidgets对象中播放，叫WxMoviePlayer。wxWidget在GUI组件上和Qt有很多相似之处，但是在使用细节上难一些。与Qt示例一样，我们将有一个顶层文件，基本上将所有内容都放在一起，还有一对代码文件和头文件来定义一个对象以封装我们播放视频的示例任务。

示例9-8：ch4_wx.cpp示例程序

```
#include "wx/wx.h"
#include "WxMoviePlayer.hpp"

// Application class, the top level object in wxWidgets
//
class MyApp : public wxApp {
  public:
    virtual bool OnInit();
};

// hind the scenes stuff to create a main() function and attach MyApp
//
DECLARE_APP( MyApp );
IMPLEMENT_APP( MyApp );

// When MyApp is initialized, do these things.
//
bool MyApp::OnInit() {

wxFrame* frame = new wxFrame( NULL, wxID_ANY, wxT("ch4_wx") );
frame->Show( true );

WxMoviePlayer* mp = new WxMoviePlayer(
  frame,
  wxPoint( -1, -1 ),
  wxSize( 640, 480 )
);
mp->open( wxString(argv[1]) );
mp->Show( true );
```

```
  return true;
}
```

程序结构看起来比Qt例子中的稍微复杂一点，但是内容很相似。第一件事是要为我们的应用创建一个类定义，我们由wxAPP类库中派生出来这个类。我们这个类的唯一不同是它将重载MyApp::OnInit()函数为我们自己的内容。声明MyApp类后，将调用两个宏DECLARE_APP()和IMPLEMENT_APP()。简单来说，这两个宏用来创建主函数和MyApp实例配置成实例程序。我们在主程序要做的最后一件事就是实际填写MyApp::OnInit()函数，程序启动时调用。当MyApp::OnInit()调用时，它会创建一个窗口（也称为wxWidget中的帧）并生成一个WxMoviePlayer实例对象到这个帧中。接着，它调用WxMoviePlayer的open方法并将视频名传递给它。

当然，所有这些有趣的事都发生在WxMoviePlayer对象的内部。示例9-9展示了这个对象的头文件。

示例9-9：WxMoviePlayer对象头文件 WxMoviePlayer.hpp

```
#include "opencv2/opencv.hpp"

#include "wx/wx.h"
#include <string>

#define TIMER_ID 0

using namespace std;

class WxMoviePlayer : public wxWindow {

  public:
    WxMoviePlayer(
      wxWindow*      parent,
      const wxPoint& pos,
      const wxSize&  size
    );
    virtual ~WxMoviePlayer() {};
    bool open( wxString file );

  private:

    cv::VideoCapture m_cap;
    cv::Mat          m_cv_img;
    wxImage          m_wx_img;
    wxBitmap         m_wx_bmp;
    wxTimer*         m_timer;
    wxWindow*        m_parent;

    void _copyImage( void );
```

```
    void OnPaint( wxPaintEvent& e );
    void OnTimer( wxTimerEvent& e );
    void OnKey(   wxKeyEvent&   e );

  protected:
    DECLARE_EVENT_TABLE();
};
```

在这个声明中，有几件重要的事情需要注意。WxMoviePlayer对象是从wxWindow派生的，wxWindow是wxWidget使用的通用类，这个通用类可用于屏幕上任何可见的东西。我们有三个事件处理方法OnPaint()，onTimer()和OnKey()将分别处理绘图、从视频获取新图像以及分布使用Esc键关闭文件。最后，你将注意到除OpenCV的cv::Mat类型图像外，还有一个类型为wxImage的对象和类型为wxBitmap的对象。在wxWidget中，位图（与操作系统相关）与图像（图像数据的设备无关表示）是区分开的。在我们查看代码文件*WxMoviePlayer.cpp*（示例9-10）时，这两者的确切作用很清楚。

示例9-10：WxMoviePlayer对象源文件WxMoviePlayer.cpp

```
#include "WxMoviePlayer.hpp"

BEGIN_EVENT_TABLE( WxMoviePlayer, wxWindow )
  EVT_PAINT( WxMoviePlayer::OnPaint )
  EVT_TIMER( TIMER_ID, WxMoviePlayer::OnTimer )
  EVT_CHAR( WxMoviePlayer::OnKey )
END_EVENT_TABLE()
```

我们首先要做的是设置与个别事件相关联的回调。我们通过wxWidget框架提供的宏来完成这一点。[注17]

```
WxMoviePlayer::WxMoviePlayer(
  wxWindow*      parent,
  const wxPoint& pos,
  const wxSize&  size
) : wxWindow( parent, -1, pos, size, wxSIMPLE_BORDER ) {
  m_timer          = NULL;
  m_parent         = parent;
}
```

创建电影播放器时，其定时器元素为NULL（当我们实际打开视频时，会设置它）。但是，我们关注于播放器的上层对象。（在这种情况下，上层对象将是我们创建的wxFrame，并可以用来放置它。）回到Esc键关闭应用程序的时候，我们需要知道哪个框架是它的上层对象。

注17：精明的读者会注意到，键盘事件被“挂钩”到WxMoviePlayer窗口小部件上，而不是顶层的应用程序或框架（就像Qt的例子和HighGUI的情况一样）。有各种各样的方法来实现这一点，但wxWidgets更倾向于键盘事件被绑定到UI中的局部可见对象上而不是全局。既然这是一个简单的例子，我们就实现一些简单的功能，并将键盘事件直接绑定到电影播放器。

```
void WxMoviePlayer::OnPaint( wxPaintEvent& event ) {
  wxPaintDC dc( this );

  if( !dc.Ok() ) return;

  int x,y,w,h;
  dc.BeginDrawing();
    dc.GetClippingBox( &x, &y, &w, &h );
    dc.DrawBitmap( m_wx_bmp, x, y );
  dc.EndDrawing();

  return;
}
```

需要在屏幕上重新绘制窗口时，将调用WxMoviePlayer::OnPaint()。请注意，当我们执行WxMoviePlayer::OnPaint()时，实际需要绘制的信息假定是在m_wx_bpm中，即wxBitmap对象。因为wxBitmap是依赖于系统的表示，所以它已经准备好被复制到屏幕上了。接下来的两个方法WxMoviePlayer::_copyImage()和WxMoviePlayer::open()将先显示如何创建它。

```
void WxMoviePlayer::_copyImage( void ) {

  m_wx_bmp = wxBitmap( m_wx_img );

  Refresh( FALSE ); // indicate that the object is dirty
  Update();

}
```

每当从cv::VideoCapture对象读取新图像时，WxMoviePlayer::_copyImage()方法就会被调用。代码看起来并没有完成很多内容，但实际上这短短几行完成了很多内容。首先是从wxImage m_wx_img构建wxBitmap m_wx_bmp。构造函数正在处理由wxImage（看起来很像使用OpenCV的表示）所使用的抽象表示转换为特定机器使用的设备和特定系统的表示。一旦复制完成，便会调用Refresh()表示该窗口小部件“脏”并需要重新绘制，并且随后的Update()调用表示现在就进行重新绘制。

```
bool WxMoviePlayer::open( wxString file ) {
if( !m_cap.open( std::string( file.mb_str() ) )) {
  return false;
}

// If we opened the file, set up everything now:
//
m_cap.read( m_cv_img );

m_wx_img = wxImage(
  m_cv_img.cols,
  m_cv_img.rows,
```

```
    m_cv_img.data,
    TRUE  // static data, do not free on delete()
  );

  _copyImage();

  m_timer = new wxTimer( this, TIMER_ID );
  m_timer->Start( 1000. / m_cap.get( cv::CAP_PROP_FPS ) );

  return true;

}
```

WxMoviePlayer::open()方法也做了几件重要的事情。第一个是实际打开cv::VideoCapture对象，但还有很多事情要做。接下来，从播放器读取图像，并用于创建“指向”OpenCV cv::Mat图像的wxImage对象。这是我们在Qt示例中使用的相反理念：在这种情况下，首先创建cv::Mat并拥有数据，然后再创建GUI工具包图像对象，并且它只是现有数据的标题。接下来，我们调用WxMoviePlayer::_copyImage()，该函数将OpenCV图像m_cv_img转换为本地位图。

最后，我们需要创建一个wxTimer对象，并告诉它每隔几毫秒唤醒一次，该数字由cv::VideoCapture对象提供的FPS计算。每当定时器到计时时间，都会生成wxTimerEvent并将其传递给WxMoviePlayer::OnTimer()，你要记住这些事件的处理程序。

```
void WxMoviePlayer::OnTimer( wxTimerEvent& event ) {

  if( !m_cap.isOpened() ) return;

  m_cap.read( m_cv_img );
  cv::cvtColor( m_cv_img, m_cv_img, cv::BGR2RGB );
  _copyImage();

}
```

那个处理程序没有做太多的事情，主要是从视频中读取一个新帧，将该帧从BGR转换为RGB进行显示，然后调用WxMoviePlayer::_copyImage()，为我们产生下一个位图。

```
void WxMoviePlayer::OnKey( wxKeyEvent& e ) {

  if( e.GetKeyCode() == WXK_ESCAPE ) m_parent->Close();

}
```

最后，我们有对任何按键作出响应的处理程序。处理程序只是简单检查按下的健是否是Esc键，如果是，则关闭该程序。请注意，我们不关闭WxMoviePlayer对象，而是关闭上层框架。关闭框架与关闭窗口的结果相同，它也关闭应用程序。

OpenCV和Windows模板库（WTL）的一个示例

在这个示例中，我们将使用本地Windows GUI API.Special。[注18]Windows模板库（WTL）是一个非常精简的基于Win32 API的C++封装。WTL应用在结构上与MFC类似，因为其中有一个应用/文档视图的结构。为了这个示例，我们将运行Visual Studio中的WTL应用向导（图9-7），创建一个新的SDI 应用，并确保在用户界面功能中选择使用一个视图窗口（应该是默认选项）作为开始。

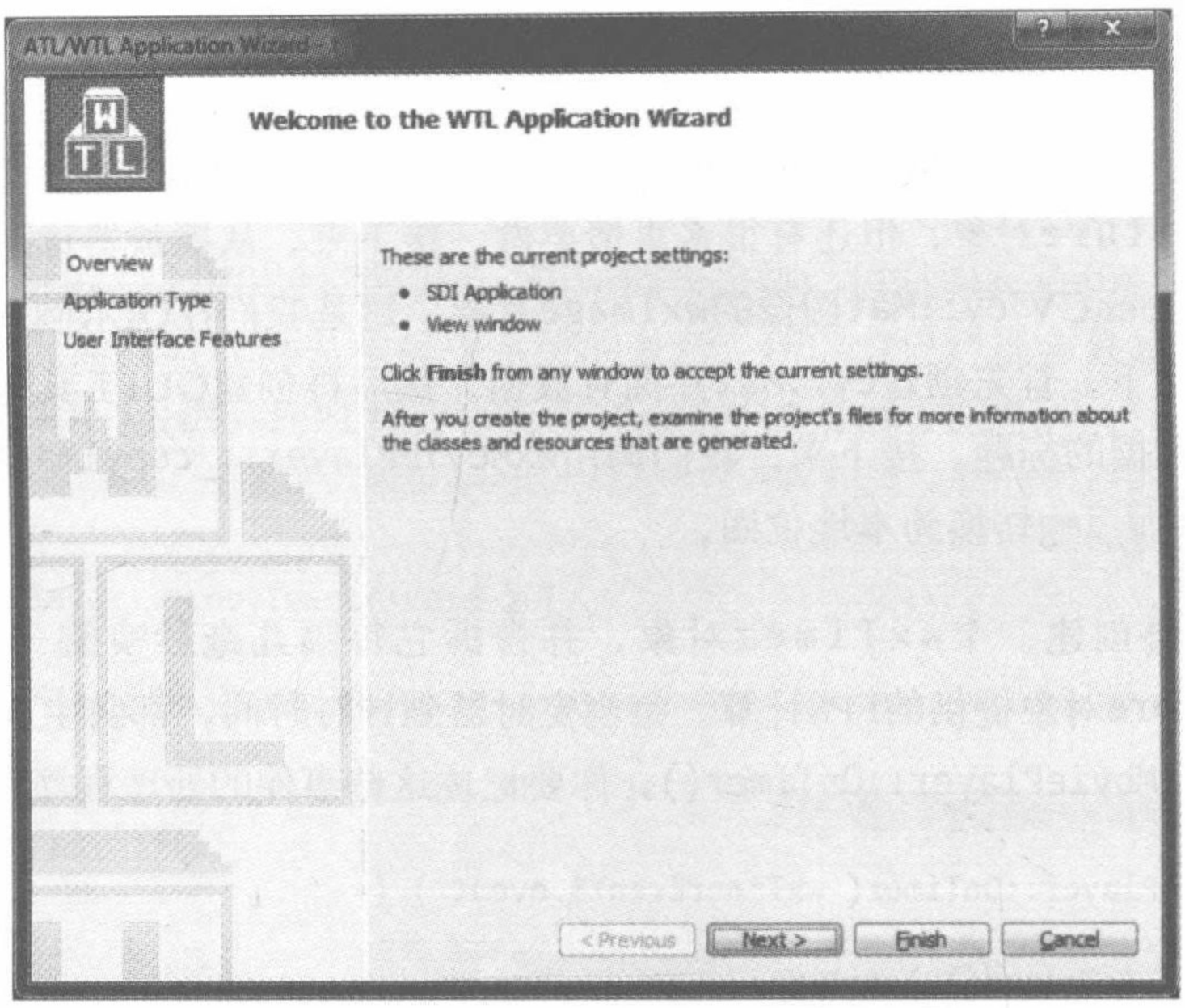


图9-7. WTL应用向导

向导生成的确切文件名取决于你为项目提供的名称。对于示例9-11，该项目名为*OpenCVTest*，我们将主要在COpenCVTestView类中工作。

示例9-11：视图类的头文件示例

```
class COpenCVTestView : public CWindowImpl<COpenCVTestView> {
public:
  DECLARE_WND_CLASS(NULL)

  bool OpenFile(std::string file);
  void _copyImage();


  BOOL PreTranslateMessage(MSG* pMsg);

  BEGIN_MSG_MAP(COpenCVTestView)
    MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBkgnd)
```

注18：在此特别感谢WTL示例代码的原作者Sam Leventer。

```
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_TIMER, OnTimer)
  END_MSG_MAP()

// Handler prototypes (uncomment arguments if needed):
//  LRESULT MessageHandler(
//    UINT    /*uMsg*/,
//    WPARAM  /*wParam*/,
//    LPARAM  /*lParam*/,
//    BOOL&   /*bHandled*/
//  );
//  LRESULT CommandHandler(
//    WORD    /*wNotifyCode*/,
//    WORD    /*wID*/,
//    HWND    /*hWndCtl*/,
//    BOOL&   /*bHandled*/
//  );
//  LRESULT NotifyHandler(
//    int     /*idCtrl*/,
//    LPNMHDR /*pnmh*/,
//    BOOL&   /*bHandled*/
//  );

  LRESULT OnPaint(
    UINT    /*uMsg*/,
    WPARAM  /*wParam*/,
    LPARAM  /*lParam*/,
    BOOL&   /*bHandled*/
  );
  LRESULT OnTimer(
    UINT    /*uMsg*/,
    WPARAM  /*wParam*/,
    LPARAM  /*lParam*/,
    BOOL&   /*bHandled*/
  );
  LRESULT OnEraseBkgnd(
    UINT    /*uMsg*/,
    WPARAM  /*wParam*/,
    LPARAM  /*lParam*/,
    BOOL&   /*bHandled*/
  );

private:
  cv::VideoCapture m_cap;
  cv::Mat          m_cv_img;

  RGBTRIPLE*       m_bitmapBits;
};
```

这里的结构非常类似于前面的wxWidget示例。视图代码之外的唯一变化是Open菜单项处理程序，在你的CMainFrame类中。需要调用视图类来打开视频：

```
LRESULT CMainFrame::OnFileOpen(
  WORD /*wNotifyCode*/,
```

```
  WORD /*wID*/,
  HWND /*hWndCtl*/,
  BOOL& /*bHandled*/
) {
  WTL::CFileDialog dlg(TRUE);
  if (IDOK == dlg.DoModal(m_hWnd)) {
    m_view.OpenFile(dlg.m_szFileName);
  }
  return 0;
}
```

```
bool COpenCVTestView::OpenFile(std::string file) {

  if( !m_cap.open( file ) ) return false;

  // If we opened the file, set up everything now:
  //
  m_cap.read( m_cv_img );

  // could create a DIBSection here, but let's just allocate memory for raw bits
  //
  m_bitmapBits = new RGBTRIPLE[m_cv_img.cols * m_cv_img.rows];

  _copyImage();

  SetTimer(0, 1000.0f / m_cap.get( cv::CAP_PROP_FPS ) );

  return true;
}
```

```
void COpenCVTestView::_copyImage() {

  // Copy the image data into the bitmap
  //
  cv::Mat cv_header_to_qt_image(
    cv::Size(
      m_cv_img.cols,
      m_cv_img.rows
    ),
    CV_8UC3,
    m_bitmapBits
  );
  cv::cvtColor( m_cv_img, cv_header_to_qt_image, cv::BGR2RGB );
}

LRESULT COpenCVTestView::OnPaint(
  UINT    /* uMsg     */,
  WPARAM /* wParam   */,
  LPARAM /* lParam   */,
  BOOL&  /* bHandled */
) {
  CPaintDC dc(m_hWnd);
```

```
WTL::CRect rect;
GetClientRect(&rect);

if( m_cap.isOpened() ) {

  BITMAPINFO bmi = {0};
  bmi.bmiHeader.biSize = sizeof(bmi.bmiHeader);
  bmi.bmiHeader.biCompression = BI_RGB;
  bmi.bmiHeader.biWidth       = m_cv_img.cols;

  // note that bitmaps default to bottom-up, use negative height to
  // resent top-down
  //
bmi.bmiHeader.biHeight = m_cv_img.rows * -1;

bmi.bmiHeader.biPlanes = 1;
bmi.bmiHeader.biBitCount = 24; // 32 if you use RGBQUADs for the bits

dc.StretchDIBits(
  0,                      0,
  rect.Width(),           rect.Height(),
  0,                      0,
  bmi.bmiHeader.biWidth, abs(bmi.bmiHeader.biHeight),
  m_bitmapBits,
  &bmi,
  DIB_RGB_COLORS,
  SRCCOPY
);

  } else {

    dc.FillRect(rect, COLOR_WINDOW);

  }

  return 0;
}

LRESULT COpenCVTestView::OnTimer(
  UINT   /* uMsg     */,
  WPARAM /* wParam   */,
  LPARAM /* lParam   */,
  BOOL&  /* bHandled */
) {
  // Nothing to do if capture object is not open
  //
  if( !m_cap.isOpened() ) return 0;

  m_cap.read( m_cv_img );
  _copyImage();

  Invalidate();

  return 0;
}
```

```
LRESULT COpenCVTestView::OnEraseBkgnd(
  UINT   /* uMsg     */,
  WPARAM /* wParam   */,
  LPARAM /* lParam   */,
  BOOL&  /* bHandled */
) {
  // since we completely paint our window in the OnPaint handler, use
  // an empty background handler
  return 0;
}
```

这段代码展示了如何在一个Windows的C++应用中使用基于位图的制图。这个方法比使用DirectShow来处理视频流更简单，但是效率较低。



如果正在使用.NET运行时（通过C#、VB.NET或者托管C++），可能想要观察一个完全封装OpenCV的包，例如Emgu。

小结

我们看到，OpenCV提供许多方法来将计算机视觉程序可视化。本地HighGUI工具方便易用，但对功能或最终效果不是很好。

为了更多的功能，基于Qt的HighGUI工具添加了按钮和一些漂亮的小工具，用于在屏幕上操纵图像，这对调试、参数调整以及研究程序的微小变化所带来的影响非常有帮助。由于这些方法缺乏可扩展性，所以可能不适用于专业应用程序，我们此外还查看了如何结合使用OpenCV与现有功能齐全的GUI工具包的几个示例。

练习

1. 仅使用HighGui，创建一个窗口，你可以一次加载和查看四个图像，每个窗口的大小至少为300 × 300。应该可以单击每个图像并打印点击的正确的（x, y）位置。打印输出你所点击的图像上的文字。
2. 使用Qt，创建一个窗口，你可以一次加载和查看四个图像。实现示例9-2的框图代码，以便可以在每个窗口中绘制框，但不允许框在要绘制的图像边界上绘制。
3. 使用Qt，创建一个足以容纳500 × 500的图像窗口。按下该窗口的按钮之后，会出现一个较小的100 × 100窗口，放大鼠标结束的第一张图像中的区域。滑块应允许放大1倍，2倍，3倍和4倍。处理鼠标周围超过500 × 500图像的边界的情况。黑色像素应显

示在放大窗口中。再次按下按钮时，小窗口消失，放大不起作用。按钮可以打开和关闭放大倍数。

4. 使用Qt，创建1000 × 1000窗口。按下按钮时，你可以在窗口中单击并键入并编辑文本。不允许文本超出窗口的边界。允许输入和回调。

5. 构建并运行示例9-12中所述的旋转立方体。修改它，使其包含按钮：向右，向左，向上和向下旋转。按下按钮时，立方体会旋转。

第10章

滤波与卷积

概览

至此，我们已经学过OpenCV中的基本结构体，利用这些结构体可以表示一幅图像，通过HighGUI接口可以编写一个程序来展示我们的运行结果。有了这些基础，接下来我们开始学习一些更加巧妙的图像操作方法。

后面将要学到一些比之前更加高级的图像处理方法，而不仅仅是对图像的某个像素的值进行简单的改变。在这章中，我们提到的“图像处理”，指的是在图像结构体上根据图像像素之间的联系对之进行高级处理。

预备知识

本章有两个重要的概念会经常出现，因而在深究具体的图像处理方法之前，我们很有必要花些时间来回顾一下。第一个是滤波器（也称核）以及在OpenCV中具体的运用，第二个是OpenCV在图像的边缘以及图像边界之外的区域如何调用滤波器或其他的方法。

滤波、核和卷积

这一章中，我们将讨论的函数大部分是关于图像滤波中的一些特殊情况。滤波器指的是一种由一幅图像$I(x, y)$根据像素点x, y附近的区域计算得到一幅新图像$I'(x, y)$的算法。其中，模板规定了滤波器的形状以及这个区域内像素的值的组成规律，也称“滤波器”或

者核[注1]。本章中出现的滤波器多数为线性核，也就是说I'(x, y)的像素的值由$I(x, y)$及其周围的像素的值加权相加得来。[注2]这个过程可以用下面的方程表示：

$$I'(x, y) = \sum_{i,j \in kernel} k_{i,j} \cdot I(x + i, y + j)$$

这个方程的含义是：对于一个任意形状的核（比如 5 × 5），我们需要对核内所有的点i, j做加权，然后相加，对于每一对i, j（通常表现为卷积核中的一个点），我们需要将其点乘原图像中相对于像素点x, y做一个偏移i, j之后的像素点的值后相加。矩阵I的形状称为“核的支集”。[注3]尽管“卷积”这个术语在计算机视觉行业中只是用来概括滤波在实体图像中的应用而被偶尔提及，也不能否认这种基于线性核的滤波就是我们所熟知的卷积。

用数组来可视化一个核（图10-1）是一种非常方便并且直观的方法。对于本书中所有的核，我们都以这种方式表示。

1	1	1	1	1
1	1	1	1	1
1	1	**1**	1	1
1	1	1	1	1
1	1	1	1	1

(A)

1/25 x

1	1	1	1	1
1	1	1	1	1
1	1	**1**	1	1
1	1	1	1	1
1	1	1	1	1

(B)

-1	0	1
-2	**0**	2
-1	0	1

(C)

1/273 x

1	4	7	4	1
4	16	26	16	4
7	26	**41**	26	7
4	16	26	16	4
1	4	7	4	1

(D)

图10-1. (A)5 × 5盒状核；(B)规范化的5 × 5盒状核；(C)3 × 3Sobel(sobel)核；(D)5 × 5规范化高斯（Gaussian）核，“锚点”均用粗体表示

锚点

图10-1中的每个核都有一个点的值加粗显示，这个点就称作“核的锚点”。它定义了核与源图像的对齐关系。例如图10-1(D)所示的核，数字41加粗显示，这意味着用于计算$I'(x, y)$的累加项中，一个项就是$^{41}/_{273}$与$I(x, y)$相乘的结果（同样，$I(x\text{-}1, y)$和$I(x\text{+}1, y)$与$^{26}/_{273}$相乘的结果是另外两个项）。

注1：一般认为这两种术语描述的是同一个意思，信号处理领域通常倾向于使用滤波，而数学领域更常用核。

注2：中值滤波器就是一个非线性核，这个滤波器将x, y处的值替换成核所覆盖区域的值之中位数。

注3：从技术上来说，核的支集由核内非零部分组成。

边界外推和边界处理

在探讨OpenCV如何进行图像处理的时候，我们会面临对边界的处理问题。与其他图像处理函数库不同的是，[注4]OpenCV中的滤波操作（如cv::blur()，cv::erode()，cv::dilate()等）得到的输出图像与源图像的形状是一样的。为了实现这种效果，OpenCV采用的一种方法是在源图像周围添加虚拟像素。cv::blur()函数实现了对图像每个像素与周围像素进行均值操作，显然，在cv::blur()函数的处理过程中为源图像添加虚拟像素是非常必要的。那么，如何对缺少相邻像素点的边缘像素点计算出一个有效的结果？实际上，在没有公认方法的情况下，我们一般通过自定义的方式在某一场景中处理问题。

自定义边框

在处理图像时，只要告诉要调用的函数添加虚拟像素的规则，库函数就会自动创建虚拟像素。[注4]同样，为了阐明一个操作的意图，就要注意函数在创建虚拟像素时所采用的方法了。

cv::copyMakeBorder()就是一个为图像创建边框的函数，通过指定两幅图像，第一幅是源图像，第二幅是扩充之后的图像，同时指明填充方法，这个函数就会将第一幅图像填补后的结果保存在第二幅图像中。

```
void cv::copyMakeBorder(
  cv::InputArray    src,                    // Input image
  cv::OutputArray   dst,                    // Result image
  int               top,                    // Top side padding (pixels)
  int               bottom,                 // Bottom side padding (pixels)
  int               left,                   // Left side padding (pixels)
  int               right,                  // Right side padding (pixels)
  int               borderType,             // Pixel extrapolation method
  const cv::Scalar& value = cv::Scalar()    // Used for constant borders
);
```

cv::copyMakeBorder()函数中，第一个参数是源图像；第二个参数是输出图像；后面的四个参数表示边框上下左右四个方向的尺寸；参数borderType指的是像素填充的方式（图10-2）。

为了更好地理解每幅图像中具体的操作，我们考虑如下几幅边缘处经过局部放大的图像（图10-3）。

注4：　例如MATLAB。

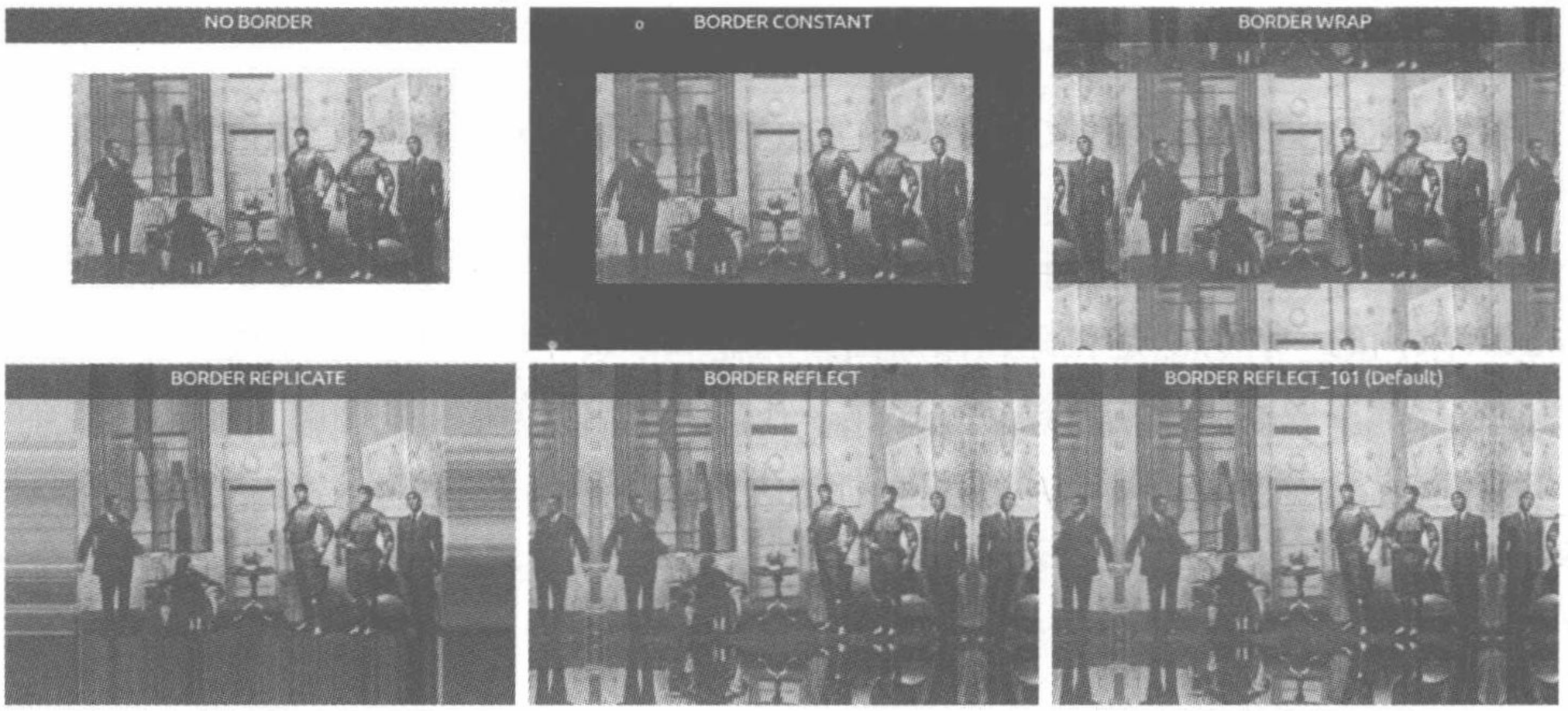


图10-2. 上面6幅图展示了cv::copyMakeBorder()函数通过不同的填充方法填充图像的结果（NO BORDER是用来对比的源图像）

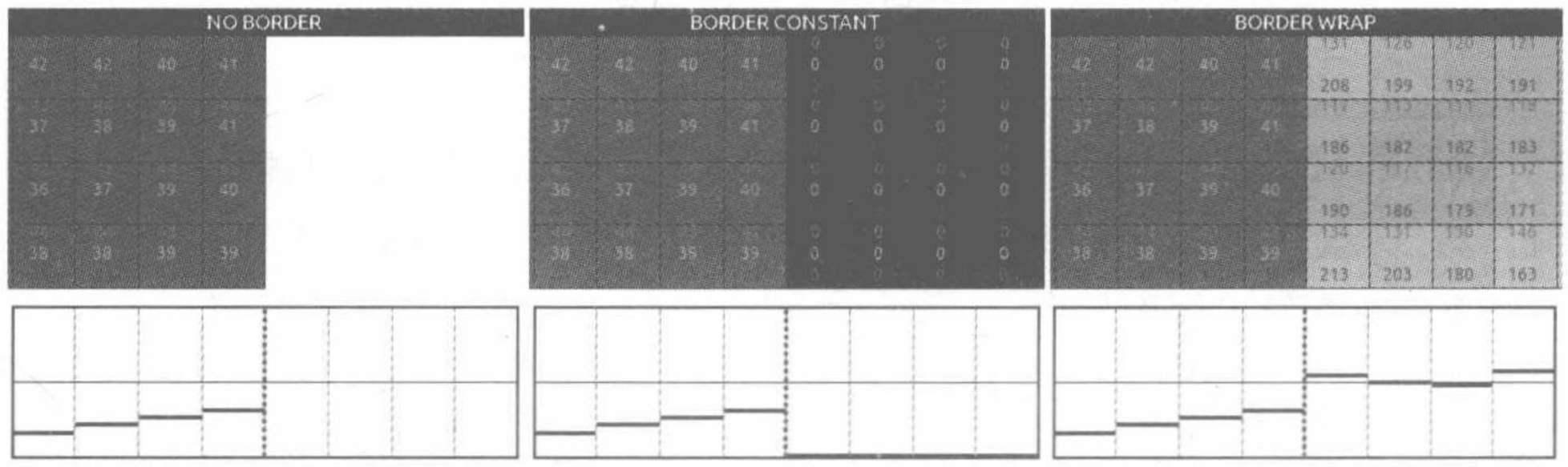


图10-3. 对每幅图像左边多次放大后的结果：对于每幅图像，显示它像素的数值，每幅图像下面是每种方法的原理图；竖直的点状线表示每幅图像的边界

可以看出，每种方法有其专门的效果。第一种方式（cv::BORDER_CONSTANT）为每个边框像素赋予一个相同的值，这个值通过传入cv::copyMakeBorder()的参数给定。（图10-2与图10-3显示了这个常量为cv::Scalar(0,0,0)时的情况。）第二种方式（cv::BORDER_WARP）的处理方式是为每个像素分配一个距离，这个距离是像素点到源图像边缘的距离，然后将距离对边相同距离的像素的值赋予对应的边框像素。第三种

方法是复制（cv::BORDER_REPLICATE），它将源图像以外的像素全部赋值为图像边缘像素的值。最后两种方法（cv::BORDER_REFLECT和cv::BORDER_REFLECT_101）效果相似，cv::BORDER_REFLECT为边框内每个像素复制了源图像内距离同一边界n对应的像素点的值。cv::BORDER_REFLECT_101为边框内每个像素复制了源图像内距离同一边界$n+1$对应的像素点的值。大多数情况下，cv::BORDER_REFLECT_101是OpenCV中默认的选项，cv::BORDER_DEFALT对应的是cv::BORDER_REFLECT_101。表10-1总结了这些操作。

表10-1：cv::copyMakeBorder()以及其他函数需要指明的边界类型

边框类型	效果
cv::BORDER_CONSTANT	复制指定的常量扩展边界
cv::BORDER_WRAP	复制对边的像素扩展边界
cv::BORDER_REPLICATE	复制边缘的像素扩展边界
cv::BORDER_REFLECT	通过镜像复制扩展边界
cv::BORDER_REFLECT_101	通过镜像复制扩展边界，边界像素除外
cv::BORDER_DEFAULT	cv::BORDER_REFLECT_101的别名

自定义外推

在某些情况下，我们需要计算某一特定像素所参考的像素的位置。比如，给定一幅宽为w、高为h的图像，我们需要知道用哪个像素为虚拟像素（$w+dx$, $h+dy$）赋值，尽管这一操作本质上是外推的，但计算这个结果的函数（有点令人震惊）叫cv::borderInterpolate()：

```
int cv::borderInterpolate(     // Returns coordinate of "donor" pixel
  int p,                       // 0-based coordinate of extrapolated pixel
  int len,                     // Length of array (on relevant axis)
  int borderType               // Pixel extrapolation method
);
```

函数cv::borderInterpolate()一次计算一个维度上的外推，这个函数的输入是一个坐标p、一个长度len（这是在关联方向上的图像的实际大小）以及一个边界类型。所以可以举个例子：我们可以在混合的边界条件下计算一个特定像素的值，在一维中使用BORDER_REFLECT_101，在另一个维度中使用BORDER_WRAP：

```
float val = img.at<float>(
 cv::borderInterpolate( 100, img.rows, BORDER_REFLECT_101 ),
 cv::borderInterpolate( -5, img.cols, BORDER_WRAP )
);
```

在OpenCV内部，这是一个经常使用的函数，比如cv::copyMakeBorder，也可以在自定义的算法中调用这个函数。borderType的可选值与cv::copyMakeBorder所使用的完全相

同。在这章中会遇到一些需要指明borderType参数的函数，对于所有这些，我们将使用相同的参数列表。

阈值化操作

图像处理过程中经常会遇见这种情况：我们已经完成了多层处理步骤并需要做出一个最终决定，或者将高于或低于某一值的像素置零同时其他的像素保持不变。OpenCV中的函数cv::threshold()实现了这些功能（参考[Sezgin04]），其原理是对于数组中每个值，根据其高于或低于这个阈值做出相应的处理，给定一个数组和阈值。根据个人喜好，也可以把阈值化操作理解成一个用1×1的核进行卷积，对每个像素进行一次非线性操作：[注6]

```
double cv::threshold(
  cv::InputArray    src,              // Input image
  cv::OutputArray   dst,              // Result image
  double            thresh,           // Threshold value
  double            maxValue,         // Max value for upward operations
  int               thresholdType     // Threshold type to use (Example 10-3)
);
```

如表10-2所示，每种阈值化操作类型对应于第i个源像素（src_i）和阈值thresh之间的比较运算方式。根据源像素和阈值之间的关系，目标像素dst_i可以被赋值为0、src_i或给定的最大值maxValue。

表10-2：cv::threshold()中thresholdType的可选项

阈值类型	操作
cv::THRESH_BINARY	$DST_I = (SRC_I > thresh)\ ?\ MAXVALUE : 0$
cv::THRESH_BINARY_INV	$DST_I = (SRC_I > thresh)\ ?\ 0 : MAXVALUE$
cv::THRESH_TRUNC	$DST_I = (SRC_I > thresh)\ ?\ THRESH : SRC_I$
cv::THRESH_TOZERO	$DST_I = (SRC_I > thresh)\ ?\ SRC_I : 0$
cv::THRESH_TOZERO_INV	$DST_I = (SRC_I > thresh)\ ?\ 0 : SRC_I$

图10-4显示了针对thresholdType每种选项对应的结果。

注6：随着我们继续学习本章，在接触了其他更复杂的卷积操作之后，这个观点的效果将非常清楚。计算机视觉领域中的很多方法都可以解释为一系列卷积运算，最后一次操作通常是阈值化处理。

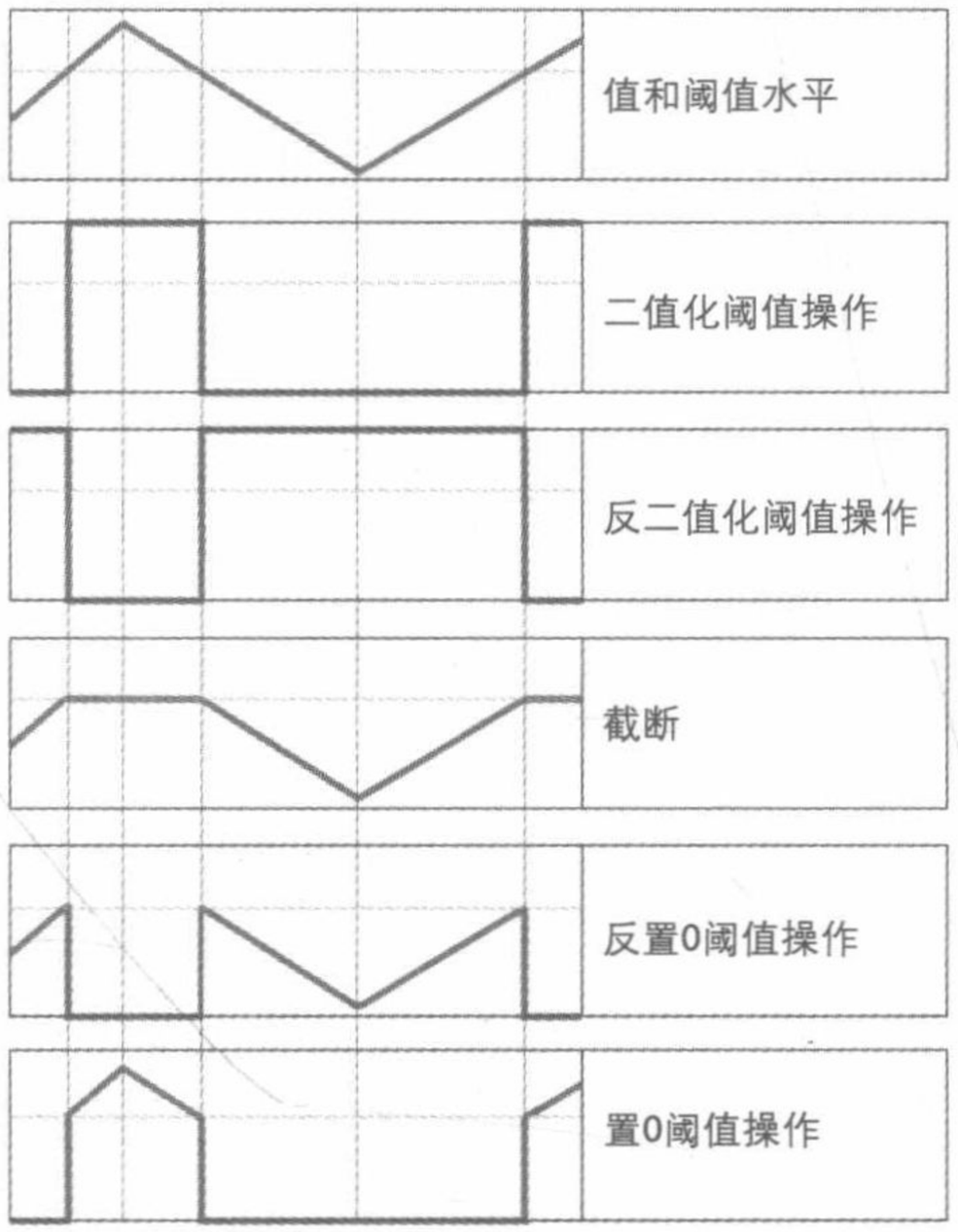


图10-4. 不同阈值类型对应的结果，在后面每幅图表示不同阈值化操作对应的效果，图中水平直线对应于第一幅图中特定的阈值。

接下来，我们看一个简单的示例，示例10-1将一幅图像的三个通道相加并将像素值限制在100以内。

示例10-1：使用cv::threshold()函数对三个通道求和

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;

void sum_rgb( const cv::Mat& src, cv::Mat& dst ) {

  // Split image onto the color planes.
  //
  vector< cv::Mat> planes;
  cv::split(src, planes);

  cv::Mat b = planes[0], g = planes[1], r = planes[2], s;

  // Add equally weighted rgb values.
  //
  cv::addWeighted( r, 1./3., g, 1./3., 0.0, s );
  cv::addWeighted( s, 1., b, 1./3., 0.0, s );

  // Truncate values above 100.
  //
  cv::threshold( s, dst, 100, 100, cv::THRESH_TRUNC );

}
```

```
void help() {
    cout << "Call: ./ch10_ex10_1 faceScene.jpg" << endl;
    cout << "Shows use of alpha blending (addWeighted) and threshold" << endl;
}

int main(int argc, char** argv) {

    help();

    if(argc < 2) { cout << "specify input image" << endl; return -1; }

    // Load the image from the given file name.
    //
cv::Mat src = cv::imread( argv[1] ), dst;
if( src.empty() ) {
cout << "can not load " << argv[1] << endl; return -1;
}
sum_rgb( src, dst);

// Create a named window with the name of the file and
// show the image in the window
//
cv::imshow( argv[1], dst );

// Idle until the user hits any key.
//
cv::waitKey(0);

return 0;
}
```

在这个示例中，有一些地方需要注意：一个是我们没有直接将数据相加并存入一个8位的数组，而是将三个通道取相同权值后进行相加，然后截断大于100的值，因为这可能会引起越界；另一个是如果我们在示例10-1中使用了一幅浮点型的图像，可以换用后面示例10-2中的代码。注意，cv::accumulate()可以将8位整型的图像累加到一幅浮点型的图像中。

示例10-2：组合和阈值图像平面的替代方法

```
void sum_rgb( const cv::Mat& src, cv::Mat& dst ) {

  // Split image onto the color planes.
  //
  vector planes;
  cv::split(src, planes);

  cv::Mat b = planes[0], g = planes[1], r = planes[2];

  // Accumulate separate planes, combine and threshold.
  //
  cv::Mat s = cv::Mat::zeros(b.size(), CV_32F);
  cv::accumulate(b, s);
  cv::accumulate(g, s); cv::accumulate(r, s);

  // Truncate values above 100 and rescale into dst.
  //
```

```
  cv::threshold( s, s, 100, 100, cv::THRESH_TRUNC );
  s.convertTo(dst, b.type());
}
```

Otsu算法

函数cv::threshold()也可以自动决定最优的阈值，你只需对参数thresh传递值cv::THRESH_OTSU即可。

简而言之，Otsu算法就是遍历所有可能的阈值，然后对每个阈值结果的两类像素计算方差σ_i^2（即低于阈值和高于阈值的两类像素）。Otsu算法计算方差使将下列表达式最小：

$$\sigma_w^2 \equiv w_1(t) \cdot \sigma_1^2 + w_2(t) \cdot \sigma_2^2$$

式中的$w_1(t)$和$w_2(t)$是根据两类像素的数量计算而来的权值，σ_1^2和σ_2^2表示两类像素的方差。实际情况中，这种方法不管是计算最小方差还是最大方差，耗费的时间都一样。原因是需要遍历所有可能的阈值，这不是一个相对高效的过程。

自适应阈值

有一种与之前不同的阈值化方法，这种方法中阈值在整个过程中自动产生变化。在OpenCV中，函数cv::adaptiveThreshold()[Jain86]实现了这种方法：

```
void cv::adaptiveThreshold(
  cv::InputArray    src,              // Input image
  cv::OutputArray   dst,              // Result image
  double            maxValue,         // Max value for upward operations
  int               adaptiveMethod,   // mean or Gaussian
  int               thresholdType,    // Threshold type to use (Example 10-3)
  int               blockSize,        // Block size
  double            C                 // Constant
);
```

cv::adaptiveThreshold()根据adaptiveMethod的设置，允许两种不同的自适应阈值方法。两种方法都是逐个像素地计算自适应阈值$T(x, y)$，方法是通过计算每个像素位置周围的$b \times b$区域的加权平均值然后减去常数C，其中b由blockSize给定。不同的是，如果选择的均值方法是cv::ADAPTIVE_THRESH_MEAN_C，那么均值时取得权值是相等的；如果选择的均值方法是cv::ADAPTIVE_THRESH_GAUSSIAN_C，(x, y)周围的像素的权值则根据其到中心点的距离通过高斯方程得到。

最后，参数thresholdType与cv::threshold()相同，可选项在表10-2中显示。

相对于一般的阈值化操作，当图像中出现较大的明暗差异时，自适应阈值时非常有效的。这个函数仅处理单通道8位或浮点型图像，并且要求源图像和目标图像不同。

示例10-3展示了比较cv::adaptiveThreshold()和cv::threshold()区别的源代码。图10-5展示了两个函数分别对一幅光照差异巨大的图像处理的结果。左下方显示的是通过cv::threshold()使用单个全局阈值的结果；右下角的部分显示了使用cv::adaptiveThreshold()的自适应局部阈值的结果。通过自适应阈值，我们获得了一个完整的棋盘，在单个全局阈值方法中，这是不存在的。此程序的启动方法在示例10-2中进行了说明，产生图10-5结果的调用指令如下：

```
./adaptThresh 15 1 1 71 15 ../Data/cal3-L.bmp
```

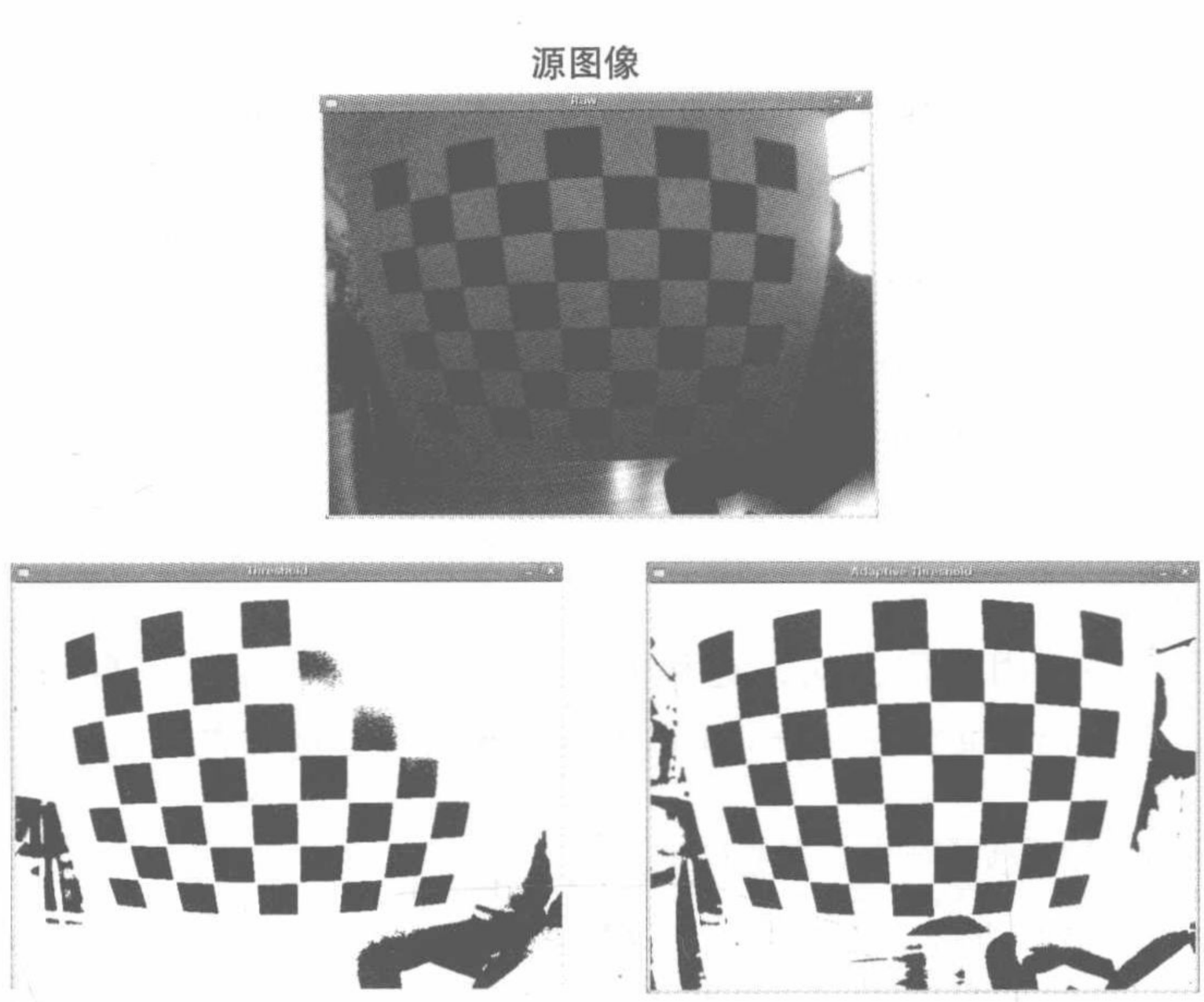


图10-5. 二值化阈值与自适应二值化阈值：使用全局阈值（左下）和自适应阈值（右下）将输入图像（顶部）转换为二值图像，源图像由Kurt Konolige提供

示例10-3：阈值与自适应阈值

```
#include <iostream>

using namespace std;

int main( int argc, char** argv )
{
  if(argc != 7) { cout <<
   "Usage: " <<argv[0] <<" fixed_threshold invert(0=off|1=on) "
   "adaptive_type(0=mean|1=gaussian) block_size offset image\n"
   "Example: " <<argv[0] <<" 100 1 0 15 10 fruits.jpg\n"; return -1; }

  // Command line
  //
  double fixed_threshold = (double)atof(argv[1]);
```

```
int threshold_type  = atoi(argv[2]) ? cv::THRESH_BINARY : cv::THRESH_BINARY_INV;
int adaptive_method = atoi(argv[3]) ? cv::ADAPTIVE_THRESH_MEAN_C
                                    : cv::ADAPTIVE_THRESH_GAUSSIAN_C;
int block_size = atoi(argv[4]);
double offset  = (double)atof(argv[5]);
cv::Mat Igray  = cv::imread(argv[6], cv::LOAD_IMAGE_GRAYSCALE);

// Read in gray image.
//
if( Igray.empty() ){ cout << "Can not load " << argv[6] << endl; return -1; }

// Declare the output images.
//
cv::Mat It, Iat;

// Thresholds.
//
cv::threshold(
  Igray,
  It,
  fixed_threshold,
  255,
  threshold_type);
cv::adaptiveThreshold(
  Igray,
  Iat,
  255,
  adaptive_method,
  threshold_type,
  block_size,
  offset
);

// Show the results.
//
cv::imshow("Raw",Igray);
cv::imshow("Threshold",It);
cv::imshow("Adaptive Threshold",Iat);
cv::waitKey(0);

return 0;
}
```

平滑

平滑在图10-6中也称“模糊”，是一种简单而又常用的图像处理操作。平滑图像的目的有很多，但通常都是为了减少噪声和伪影。在降低图像分辨率的时候，平滑也是十分重要的（这部分将在第11章对图像金字塔的介绍中详细讨论）。

OpenCV提供5种不同的平滑操作，每种操作都有对应的函数实现，这些操作平滑的结果有着细微的差别。这些函数中`src`和`dst`参数通常指源图像和目标图像，紧接其后的是不

同平滑操作所需指定的参数。其中唯一一个相同的参数是最后一个参数borderType，它指定边缘像素的平滑方式。

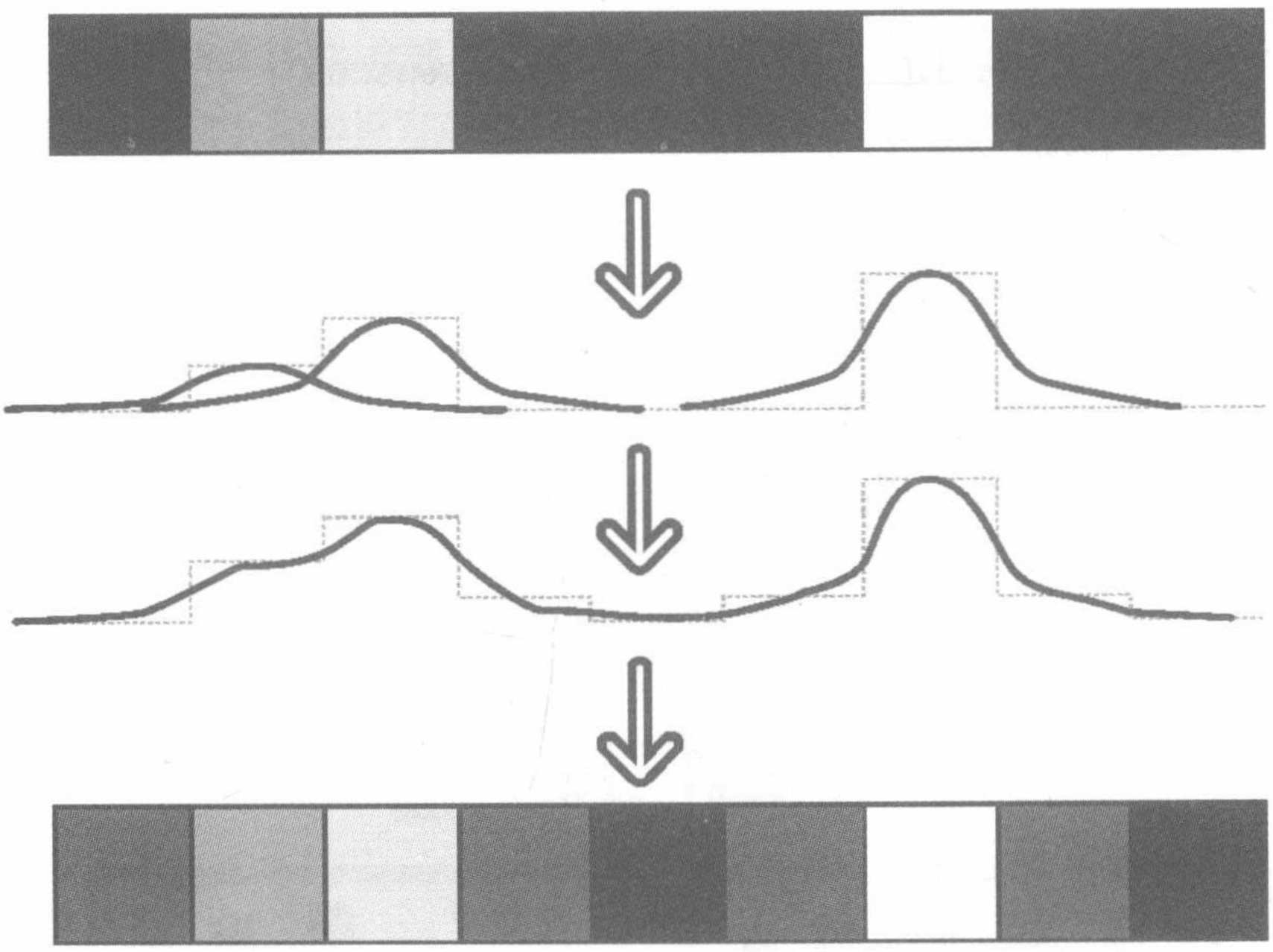


图10-6. 高斯模糊

简单模糊和方框型滤波器

```
void cv::blur(
  cv::InputArray  src,                                  // Input image
  cv::OutputArray dst,                                  // Result image
  cv::Size        ksize,                                // Kernel size
  cv::Point       anchor    = cv::Point(-1,-1),         // Location of anchor point
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

cv::blur()实现了简单模糊，目标图像中的每个值都是源图像中相应位置一个窗口（核）中像素的平均值。窗口的尺寸有ksize声明；anchor指定计算时核与源图像的对齐方式，默认情况下anchor为cv::Point(-1,-1)，表示核相对于滤波器居中。如果源图像是一幅多通道图像，则分别对每个通道进行计算。

图10-7所示的简单模糊是方框型滤波器（Box Filter）的一种特殊形式。方框型滤波器是一种矩形的并且滤波器中所有值$k_{i,j}$全部相等的滤波器。通常，所有的$k_{i,j}$为1或者1/A，其中A是滤波器的面积。后一种滤波器称为“归一化方框型滤波器”，其作用效果如图10-8所示。

$$\frac{1}{25} \underline{x} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \underline{1} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

图10-7. 一个5×5模糊滤波器，也称“归一化方框型滤波器”

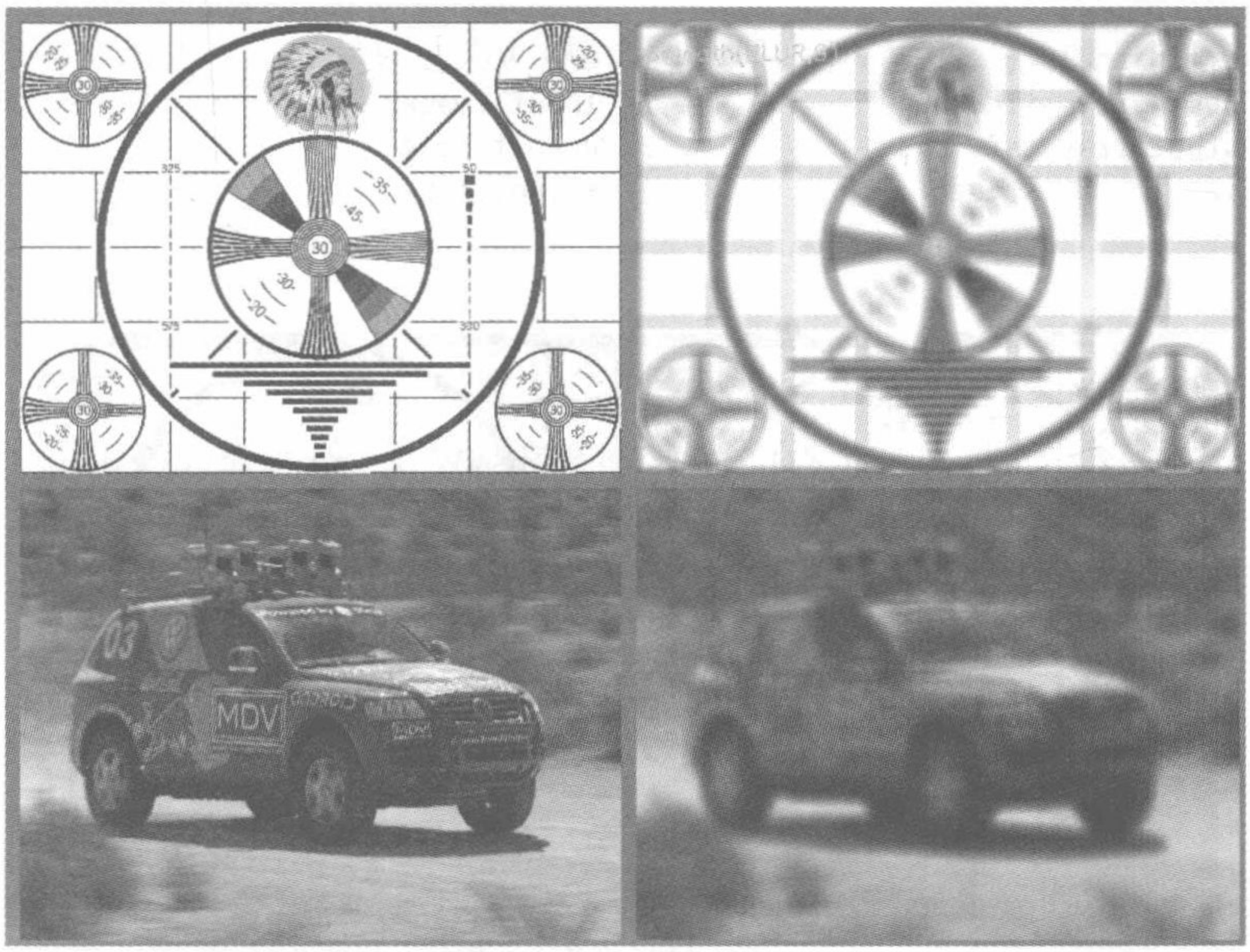


图10-8. 通过块均值平滑图像：左边是源图像，右边是输出结果

```
void cv::boxFilter(
  cv::InputArray  src,                              // Input image
  cv::OutputArray dst,                              // Result image
  int             ddepth,                           // Output depth (e.g., CV_8U)
  cv::Size        ksize,                            // Kernel size
  cv::Point       anchor     = cv::Point(-1,-1),    // Location of anchor point
  bool            normalize  = true,                // If true, divide by box area
```

```
    int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
  );
```

OpenCV中的函数cv::boxFilter()是一种一般化的形式，而cv::blur()是一种特殊化的形式。两者之间根本的区别主要是前者可以以非归一化形式调用，并且输出图像深度可以控制（cv::blur()输出图像的深度与源图像保持一致）。假如变量ddepth的值设为-1，目标图像的深度将于源图像保持一致；否则可以设置为其他任意一种常用别名（如CV_32F）。

中值滤波器

中值滤波器（Median Filter）[Bardyn84]将每个像素替换为围绕这个像素的矩形邻域内的中值或“中值”像素（相对于平均像素）。[注7]中值滤波的效果如图10-9所示。通过平均的简单模糊对噪声图像，尤其是有较大孤立的异常值（比如数字影像中的拍摄噪声）非常敏感。少量具有较大偏差的点也会严重影响到均值滤波。中值滤波可以采用取中间点的方式来消除异常值。

```
void cv::medianBlur(
  cv::InputArray  src,                  // Input image
  cv::OutputArray dst,                  // Result image
  cv::Size        ksize                 // Kernel size
);
```

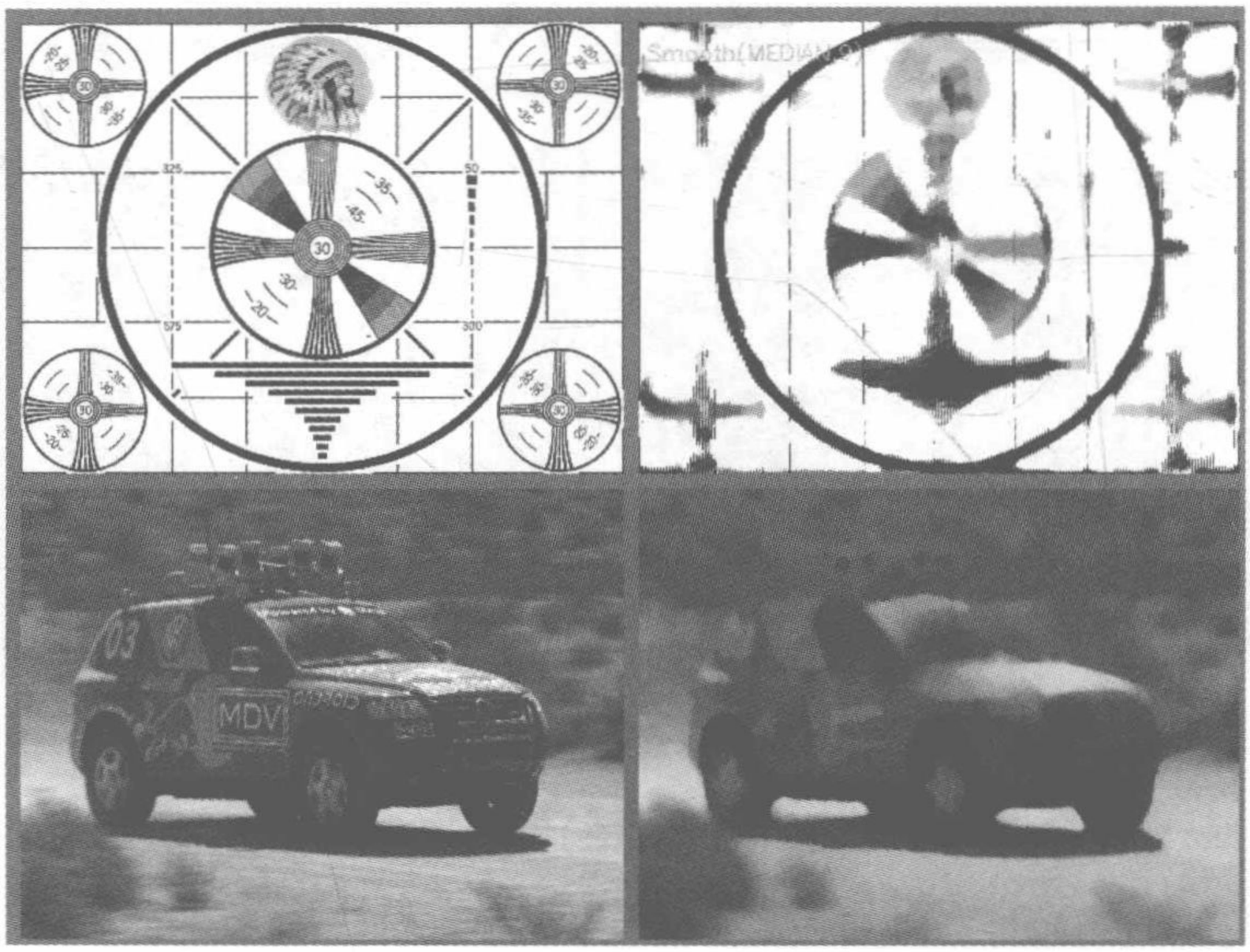


图10-9. 根据周围像素进行中值模糊

注7：　中值滤波器是一种非线性核的实例，无法以图10-1所示的图形形式表示。

cv::medianBlur的参数本质上与本章所了解的滤波器基本相同：源数组src，目标数组dst和内核大小ksize。cv::medianBlur()默认核的中心点为锚点。

高斯滤波器

接下来将要介绍的滤波器是高斯滤波器（Gaussian Filter），它应该是最有用的一种滤波器。高斯滤波对输入数组进行以此规范化的高斯核滤波，然后输出目标数组：

```
void cv::GaussianBlur(
  cv::InputArray  src,                    // Input image
  cv::OutputArray dst,                    // Result image
  cv::Size        ksize,                  // Kernel size
  double          sigmaX,                 // Gaussian half-width in x-direction
  double          sigmaY     = 0.0,       // Gaussian half-width in y-direction
    int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

对于高斯模糊（效果如图10-10所示），参数ksize指定滤波器窗口的宽度核高度；下一个参数表示高斯核在x方向上的sigma值（最大值的半宽）；第四个参数表示y方向上的sigma值。如果只给定了x，同时y设为0（默认），那么y将与x相等。如果两者都设为0，那么高斯参数将根据下列公式确定：

$$\sigma_x = \left(\frac{n_x - 1}{2}\right) \cdot 0.30 + 0.80,\ n_x = ksize.width - 1$$

$$\sigma_y = \left(\frac{n_y - 1}{2}\right) \cdot 0.30 + 0.80,\ n_y = ksize.height - 1$$

cv::GaussianBlur()的最后一个参数是常见的borderType参数。

$$^{1}/_{141} \times \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 7 & 26 & \underline{41} & 26 & 7 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

图10-10. 一个ksize=（5，3），sigmaX=1以及sigmaY=0.5的高斯核示例

OpenCV实现的高斯平滑还为几个常用的内核提供性能上的优化。3×3、5×5以及7×7的标准sigma核（sigmaX=0.0）相对其他核性能更优。高斯模糊支持单通道/三通道的8位/32位浮点型图像等。高斯模糊的效果如图10-11所示。



图10-11. 高斯滤波器（模糊）

双边滤波器

```
void cv::bilateralFilter(
  cv::InputArray  src,              // Input image
  cv::OutputArray dst,              // Result image
  int             d,                // Pixel neighborhood size (max distance)
  double          sigmaColor,       // Width param for color weight function
  double          sigmaSpace,       // Width param for spatial weight function
  int             borderType = cv::BORDER_DEFAULT  // Border extrapolation to use
);
```

OpenCV支持的第五种也是最后一种平滑方式是双边滤波器（Bilateral Filter）[Tomasi98]，其效果如图10-12所示。双边滤波器是一种比较大的图像分析算子，也就是边缘保持平滑。相对于高斯平滑，双边平滑更容易理解。高斯模糊的过程是减缓像素在空间上的变化，因此与邻域的关系紧密，而随机噪声在像素间的变化幅度又会非常的大（即噪声不是空间相关的）。基于这种前提高斯平滑很好地减弱了噪声并且保留了小信号。不幸的是，这种方式破坏了边缘信息，最终结果是高斯模糊把边缘也模糊了。

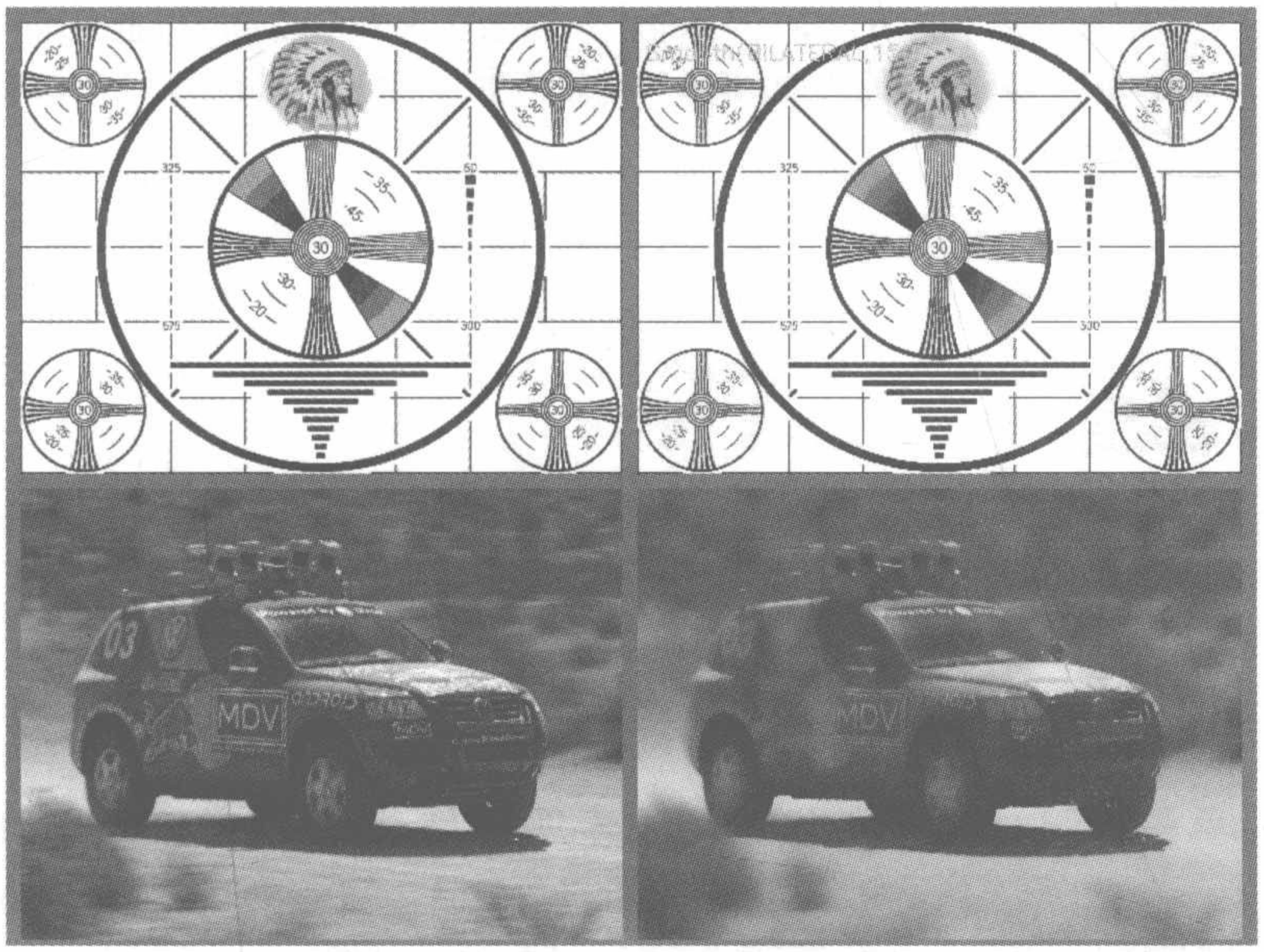


图10-12. 双边平滑

相似于高斯平滑，双边滤波对每个像素及其领域内的像素进行了加权平均。其权重由两部分组成，第一部分同高斯平滑；第二部分也是高斯权重，不同的是它不是基于空间距离而是色彩强度差[注8]计算而来，在多通道（彩色）图像上强度差由各分量的加权累加代替。[注9]可以把双边滤波当作是高斯平滑，只是相似程度更高的像素权值更高，边缘更明显，对比度更高。双边滤波的效果就是将源图像变成一幅水彩画，[注10]这种效果在多次迭代后更加显著，因此这种方法在图像分割领域十分有用。

双边滤波需要提供三个参数（源图像和目标图像除外）。第一个是像素邻域的直径d；第二个是颜色空间滤波器的sigma值`sigmaColor`，其与高斯滤波器中的参数segma相似；第三个是坐标空间中滤波器的sigma值`sigmaSpace`。第二个参数越大，平滑时所包括的强度（色彩）越大（因此图像的不连续性将会更显著）。

滤波器的大小d对算法的效率有很大影响，通常在视频处理时不大于5，但在非实时应用方面这个值可以放大到9。你也可以在调用这个函数时将其设为-1，函数将自动为图像计算`sigmaSpace`变量的值。

注8：图像为多通道（彩色）时，强度差用各色彩分量加权累加代替，在CIE实验室色彩空间上这个权值是欧几里得距离。

注9：技术上说，高斯分布函数并不是双边滤波所必须的，但OpenCV内部实现多用高斯权值，尽管还有很多其他方法可以使用。

注10：这种效应在双边滤波的多次迭代之后特别显著。



实际情况中，小的sigmaSpace值比如10会带来一个轻微的但也明显的效果；而大的sigmaSpace值比如150会对图像产生非常显著的影响，使图像有一种卡通的效果。

导数和梯度

卷积中最重要也是最基础的部分就是（近似）计算导数，方法有多种，但往往适合实际情况的却不多。

索贝尔导数

一般来说，用来表示微分的最常用的算子是索贝尔（Sobel）导数[Sobel73]算子（图10-13和图10-14）。Sobel算子可以实现任意阶导数和混合偏导数（例如∂2/∂x∂y）。

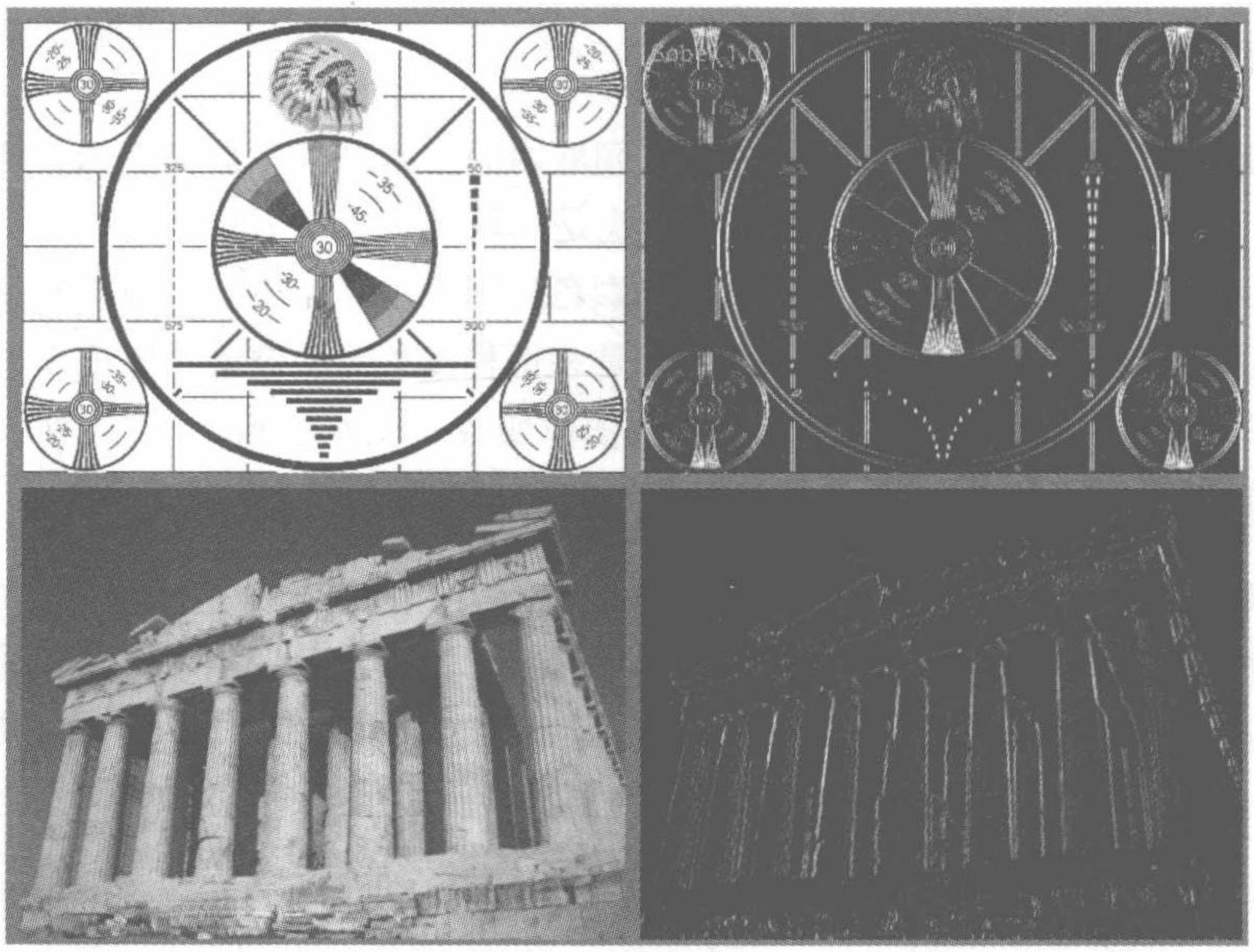


图10-13. 在*x*方向上用Sobel算子进行近似一阶求导的结果

```
void cv::Sobel(
  cv::InputArray  src,                // Input image
  cv::OutputArray dst,                // Result image
  int             ddepth,             // Pixel depth of output (e.g., CV_8U)
  int             xorder,             // order of corresponding derivative in x
  int             yorder,             // order of corresponding derivative in y
  cv::Size        ksize     = 3,      // Kernel size
```

```
    double          scale      = 1,      // Scale (applied before assignment)
    double          delta      = 0,      // Offset (applied before assignment)
    int             borderType = cv::BORDER_DEFAULT  // Border extrapolation
  );
```

其中的src和dst是源图像和目标图像；你可以通过指明参数ddepth来确定目标图像的深度或类型（如CV_32F）；举个例子，如果src是一幅8位图像，那么dst需要至少CV_16S的深度保证不出现溢出；xorder和yorder是求导顺序，其取值范围为0、1和2。0表示在这个方向上不进行求导[注11]；ksize是一个奇数，表示调用的滤波器的宽和高，目前最大支持到31；[注12]阈值和偏移将在把结果存入dst前调用，这有助于你将求导结果可视化：

$$dst_i = scale \cdot \left\{ \sum_{i,j \varepsilon\, sobel_kernel} k_{i,j} * I(x+i,\ y+j) \right\} + delta$$

borderType参数的功能与其他卷积操作完全一样。

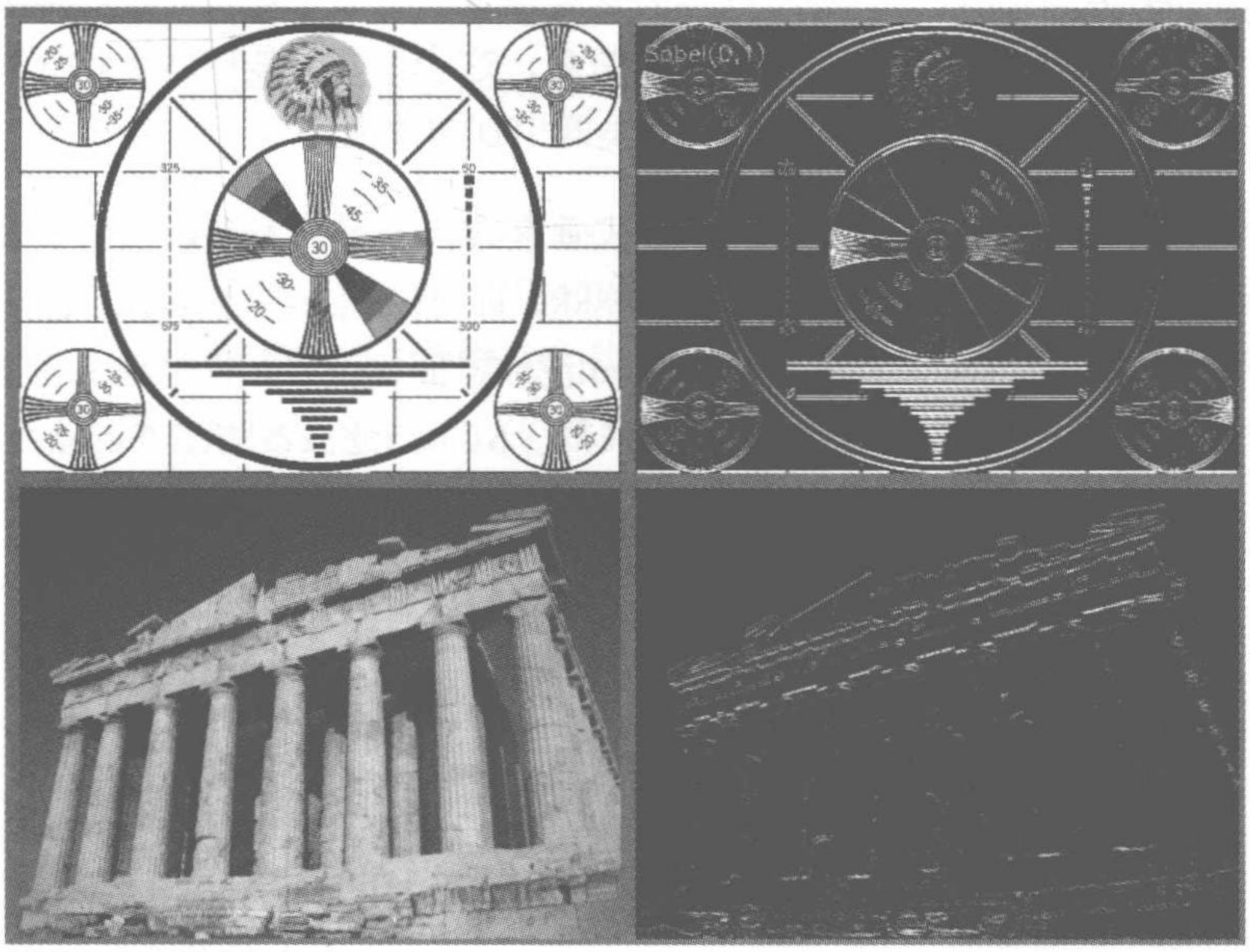


图10-14. Sobel算子效果，y方向近似一阶导数

Sobel算子有一个好处就是可以将核定义为各种大小，并且可以快速、迭代式地构造这些核。大的核可以更好地近似导数，因为可以消除噪声影响。不过，假如导数在空间上变化剧烈，核太大会使结果发生偏差。

注11：xorder和yorder不能同时取值为0。

注12：实际应用中，ksize≥3时才有意义，如果把ksize设为1，函数将自动使用3。

为了更好地理解Sobel算子，我们必须明确它不是真正的导数，因为它定义在离散空间上。Sobel算子实际上表示的是一个多项式，也就是说在*x*方向上进行二阶Sobel运算表示的并不是二阶导数，而是对抛物线函数的局部拟合。这也就说明了为什么要使用一个更大的核，更大的核拟合了更多的像素。

Scharr滤波器

在离散空间上有很多方法来近似导数，Sobel算子的缺点是核比较小的时候准确度不高。对于大型的核，近似过程中使用了较多的点，因此精度问题不太显著。cv::Sobel()中使用的*X*、*Y*滤波器不会出现这种偏差，因为它们准确地与x、y轴对齐。当你需要计算某一点的梯度时，问题就来了（图像梯度的方向通过计算两个滤波器响应比值*y*/*x*的反正切值得来）。[注13]

为了将图像内的信息联系起来，你可能需要这样测量一幅图像：在处理过程中，通过在目标附近组织一幅梯度直方图来收集其形状信息，这些直方图是许多形状分类器训练和使用的基础。因此，梯度角的误差会降低分类器识别的效果。

对于3×3的Sobel滤波器，梯度角距离水平或垂直方向越远，误差越明显。在OpenCV中，调用cv::Sobel()时设置ksize为cv::SCHARR，即可消除3×3这样小但是快的Sobel导数滤波器所带来的误差。Scharr滤波器和Sobel滤波器同样很快，但是前者精度更高。因此选择3×3的滤波器时，应当使用Scharr滤波器。Scharr滤波器的滤波器系数如图10-15所示[Scharr00]。

-3	0	+3
-10	0	+10
-3	0	+3

A

-3	-10	-3
0	0	0
+3	+10	+3

B

图10-15. 参数为cv::SCHARR时3×3的Scharr滤波器

注13：不知你是否还记得，函数cv::catToPolar()和cv::polarToCart()实现了这些转换。如果需要在*x*和*y*方向的导数上调用cv::cartToPolar()，则应该使用CV_SCHARR方法对这些图像进行计算。

拉普拉斯变换

OpenCV中的函数Laplacian（Marr[Marr82]第一次在视觉上使用了它）实现了对拉普拉斯（Laplacian）算子的离散近似：[注14]

$$Laplace\ (f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

由于的Laplacian算子可以通过二阶导数定义，因此可以把它的离散实现与二阶Sobel导数联系起来。确实如此，OpenCV在实现Laplacian算子时使用了Sobel算子：

```
void cv::Laplacian(
  cv::InputArray  src,                    // Input image
  cv::OutputArray dst,                    // Result image
  int             ddepth,                 // Depth of output image (e.g., CV_8U)
  cv::Size        ksize      = 3,         // Kernel size
  double          scale      = 1,         // Scale applied before assignment to dst
  double          delta      = 0,         // Offset applied before assignment to dst
  int             borderType = cv::BORDER_DEFAULT  // Border extrapolation to use
);
```

除了求导顺序，cv::Laplacian()函数与cv::Sobel()函数的参数相同。孔径ksize与之前在Sobel导数中提到的一样，就是在二阶求导时像素采样区域的大小。实际情况中，只要ksize不为1，Laplacian算子的实现就是直接计算Sobel算子响应之和；当ksize等于1时，Laplacian算子计算图像与图10-16所示的核的卷积。

0	1	0
1	-4	1
0	1	0

图10-16. ksize=1时cv::Laplacian()调用的核

Laplacian算子可以用于各种场景处理，一种常见的应用就是匹配“斑点”。回想一下，Laplacian算子就是图像在x、y轴方向上导数之和，这意味着一个被较大值包围的点或小斑点（比ksize小）处的值将会变得很大。相反，被较小值包围的点或小斑点处的值将在负方向上变得很大。

注14：Laplacian算子与Laplacian金字塔之间是有区别的，第11章将详细讨论。

有了这些，Laplacian算子同样可以用于边缘检测，想象一下，一个函数的一阶导数在原函数变化大的地方，值会相应变大，同样，图像中的边缘处也会有同样的变化，所以导数在这些地方将变得很大。因此，我们可以在二阶导为0的地方搜寻这么一个极大值，源图像中的边缘通过Laplacian算子运算后会变成0。不幸的是，真正的边和一些不是边的地方在Laplacian图像中都会变成0。不过这些问题能够解决，通过滤掉Sobel一阶导数中拥有较大值的点即可。图10-17显示了Laplacian算子对一幅图像运算的结果和其一阶导数和二阶导数在穿越*x*轴处的细节。

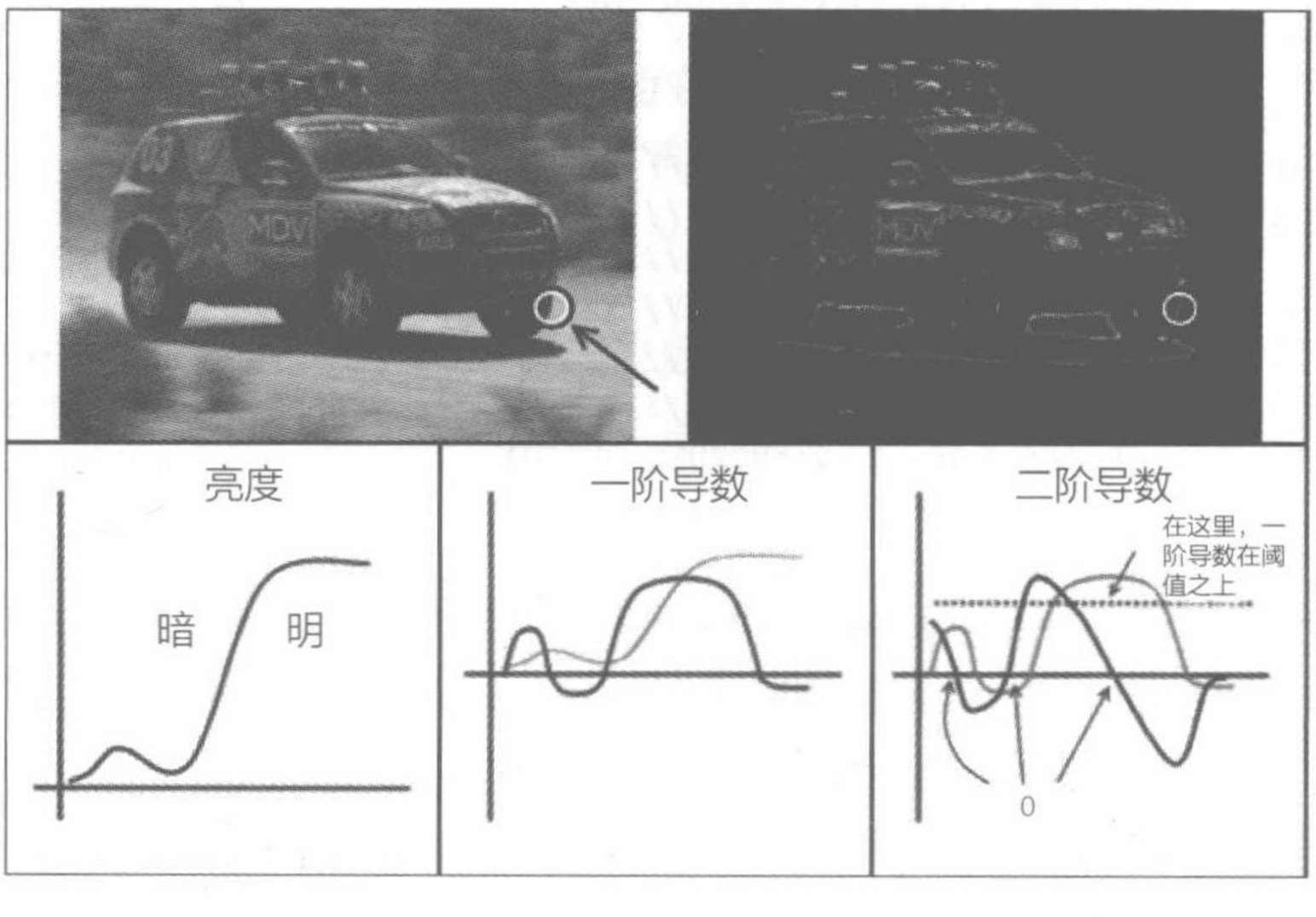


图10-17. 竞速赛车的拉普拉斯变换（右上角）：圆圈标记部分的局部放大（只考虑*x*轴方向），下面三幅图展示了亮度的一阶导数和二阶导数，二阶导中的0值表示边缘，对应一阶导中最大的0表示一条明显的边

图像形态学

OpenCV还提供了一种高效且易用的图像形态学变换接口。图10-18展示了几种常见的形态学处理。图像形态学有其特定的发展领域，特别是在计算机视觉发展早期，已经发展出了很多的形态学方法。大部分都是为某个特定目的而产生的，其中一些更是沿用了很长一段时间。基本上，所有的形态学操作都基于两种原始操作，接下来的讲述也将以这两点开始，循序渐进发展到更加复杂的操作，每个更加复杂的操作都将通过前面的方法来表示。

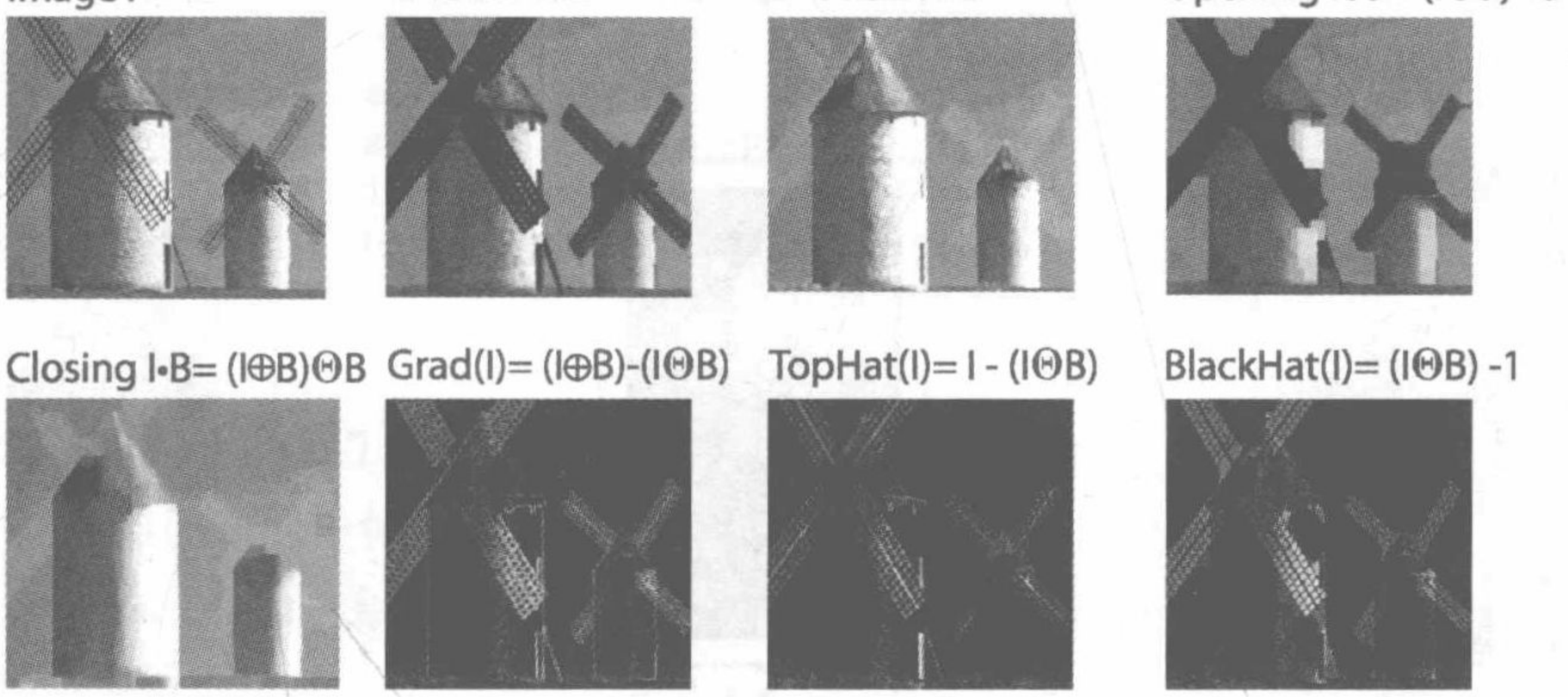


图10-18. 所有的形态学操作

膨胀和腐蚀

最基础的形态学变换是膨胀和腐蚀，它们在许多方面得到了应用，比如消除噪声、元素分割和连接等。基于这两种操作，可以实现更复杂的形态学操作，用来定位强度峰值或孔洞、另一种形式的图像梯度等。

膨胀是一种卷积操作，它将目标像素的值替换为卷积核覆盖区域的局部最大值。如前所述，这是一个非线性核的例子，因此不能像图10-1那样用图表表示。通常，膨胀采用的核是一个四边形或圆形的实心核，其锚点在中心。膨胀的作用是使图中填充区域[注15]像图10-19那样生长。

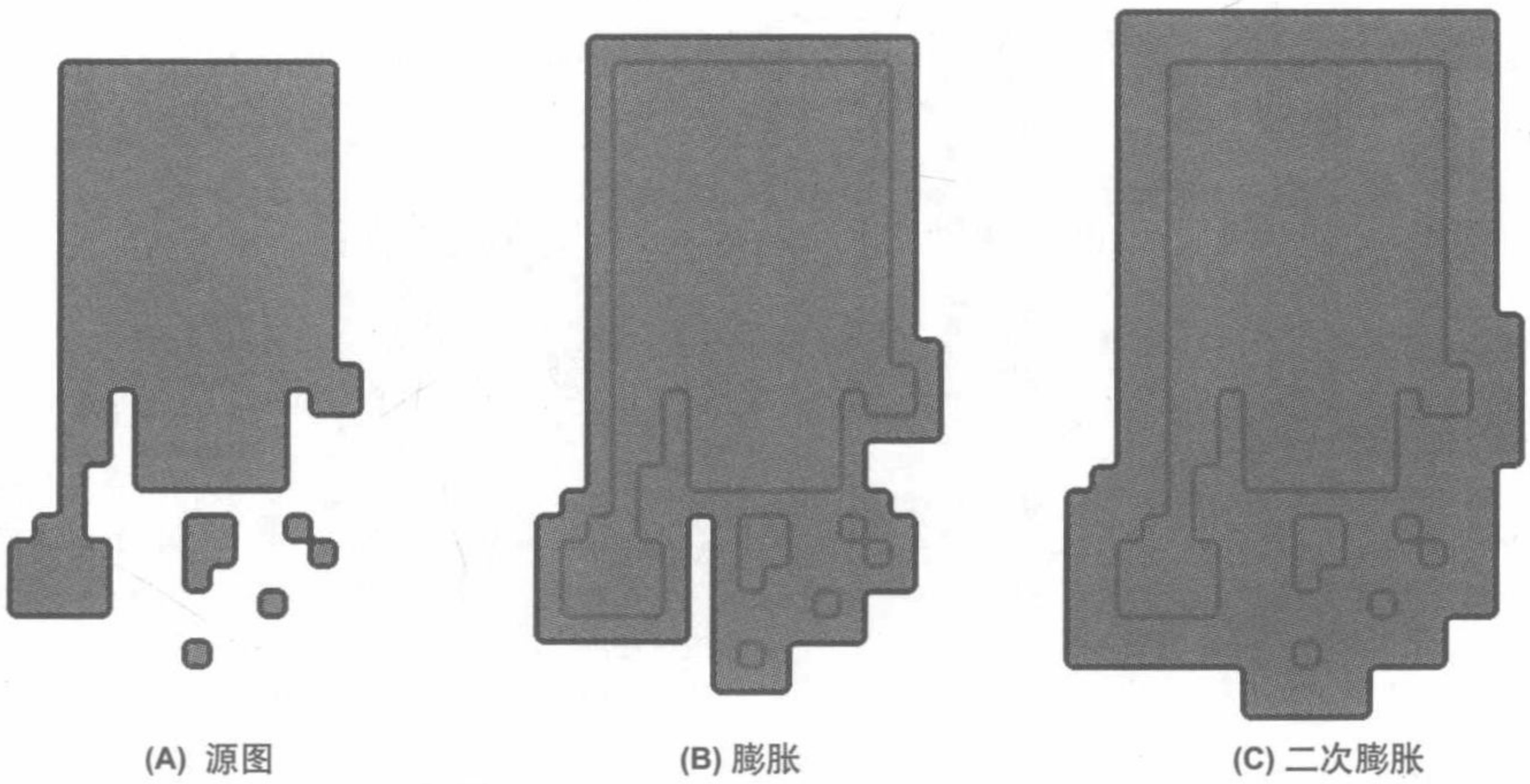
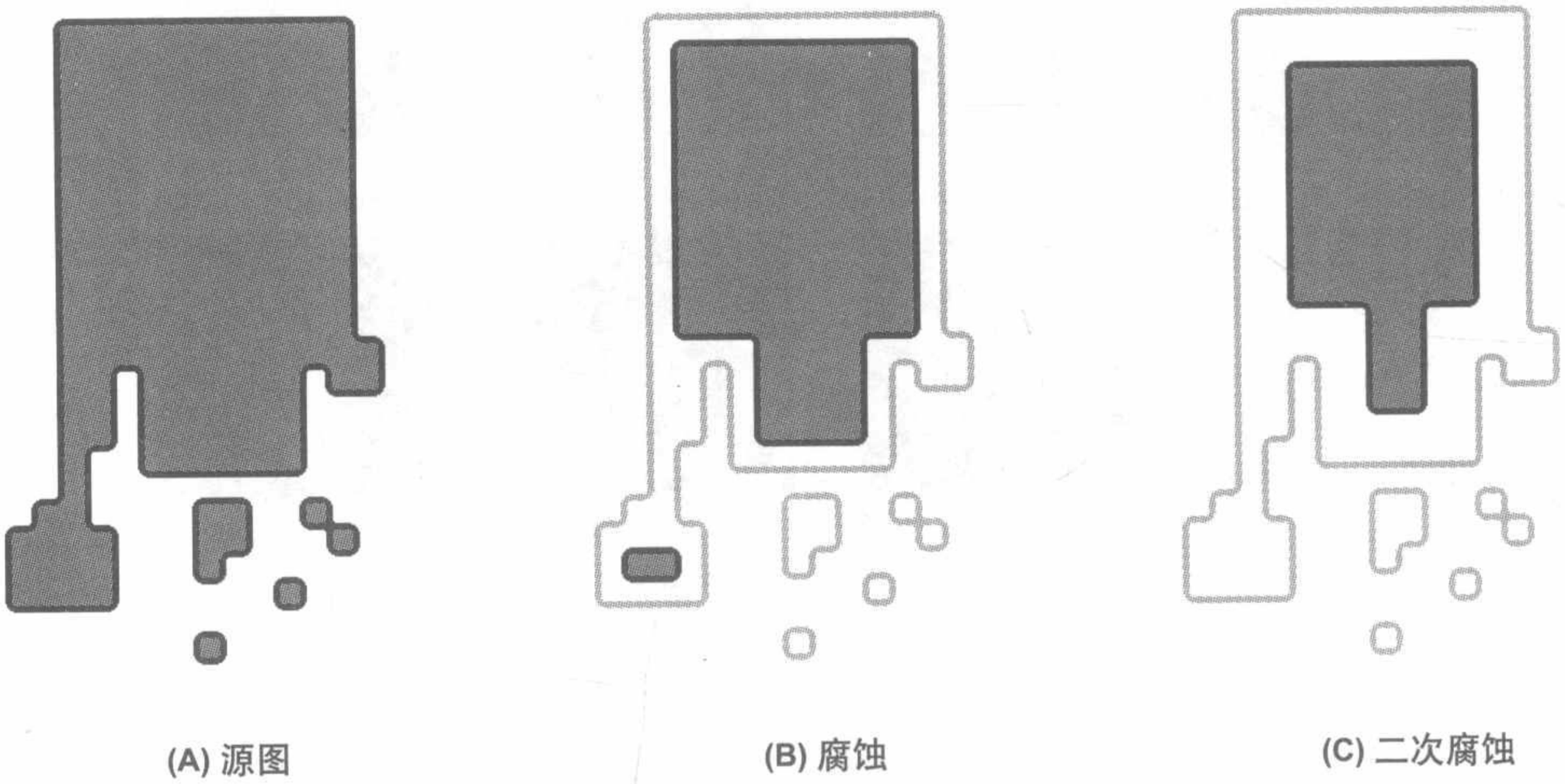


图10-19. 形态学膨胀：在四边形核覆盖范围内取最大值

注15：在这里，填充区域指的是那些非零的像素，也可以将其乘坐“亮点”，因为局部最大表示模版覆盖范围内强度最高的值。

与膨胀对应，腐蚀是与之相反的操作，腐蚀操作计算的是核覆盖范围内的局部最小值。[注16]图10-20解释了腐蚀的效果。



(A) 源图　　(B) 腐蚀　　(C) 二次腐蚀

图10-20. 形态学腐蚀：在四边形核覆盖范围内取最小值



图像的形态学操作通常在阈值化操作后的布尔[注17]图像上进行，不过由于膨胀和腐蚀只是最大和最小操作，因此形态学操作也可以在强度图像上进行。

总的来说，膨胀扩张了明亮区域，腐蚀缩减了明亮区域。另外，膨胀填充凹面，腐蚀消除突起。当然，具体的效果还跟核有关，不过只要和是凸并且实心，这些法则肯定就是有效的。

OpenCV中，我们用cv::erode()和cv::dilate()函数实现这些变换：

```
void cv::erode(
  cv::InputArray     src,                          // Input image
  cv::OutputArray    dst,                          // Result image
  cv::InputArray     element,                      // Structuring, a cv::Mat()
  cv::Point          anchor      = cv::Point(-1,-1),   // Location of anchor point
  int                iterations  = 1,                  // Number of times to apply
  int                borderType  = cv::BORDER_CONSTANT // Border extrapolation
  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
);
void cv::dilate(
```

注16：准确地说，是将目标图像中的值设为核覆盖范围内的最小值。

注17：需要说明的是，OpenCV中实际上并没有二值图像这种数据类型，最小规格的表示是8位字符。函数以0（假或者0）和非0（真或者1）的形式来对像素进行布尔分类。

```
    cv::InputArray    src,                  // Input image
    cv::OutputArray   dst,                  // esult image
    cv::InputArray    element,                              // Structuring, a cv::Mat()
    cv::Point         anchor      = cv::Point(-1,-1),   // Location of anchor point
    int               iterations  = 1,                  // Number of times to apply
    int               borderType  = cv::BORDER_CONSTANT // Border extrapolation
    const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
);
```

cv::erode()和cv::dilate()都需要传入源图像和目标图像，并且都支持就地调用（源图像和目标图像是同一幅图像）。第三个参数是核，你可以向它传递一个未被初始化的cv::Mat()，不过这会导致函数使用默认的锚点在中心的3×3核（后面将介绍如何自定义核）。第四个参数是迭代次数，如果不为1，将自动重复多次调用这个函数。参数borderType就是常用的边框类型，当borderType是cv::BORDER_CONSTANT时，边缘像素会填入borderValue。

图10-21展示了示例图像的腐蚀结果，图10-22展示了同一幅示例图像膨胀的结果。腐蚀操作通常用于消除图中斑点一样的噪声，原理是斑点经过腐蚀后会消失，而大的可见区域不会受到影响。膨胀操作通常用于发现连通分支（大的颜色或强度相似像素的离散区域）。膨胀出现的原因是，在许多情况下，由于噪声或者阴影等因素，大的区域可能被分解成多个分支。一些细微的膨胀会导致这些分支相遇并组合成一个整体。

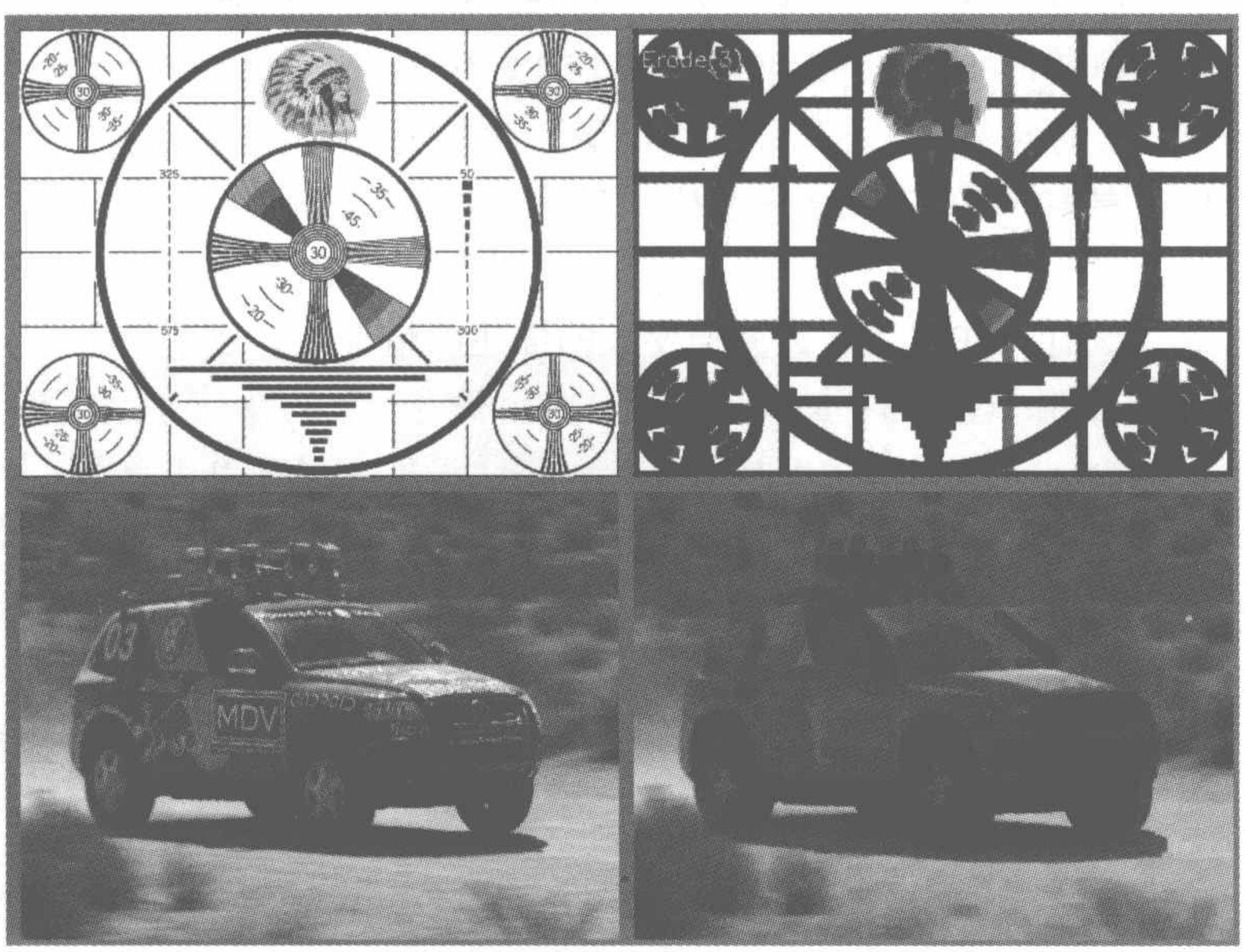


图10-21. 腐蚀结果，或者说“最小化”操作：明亮的区域被隔离并收缩

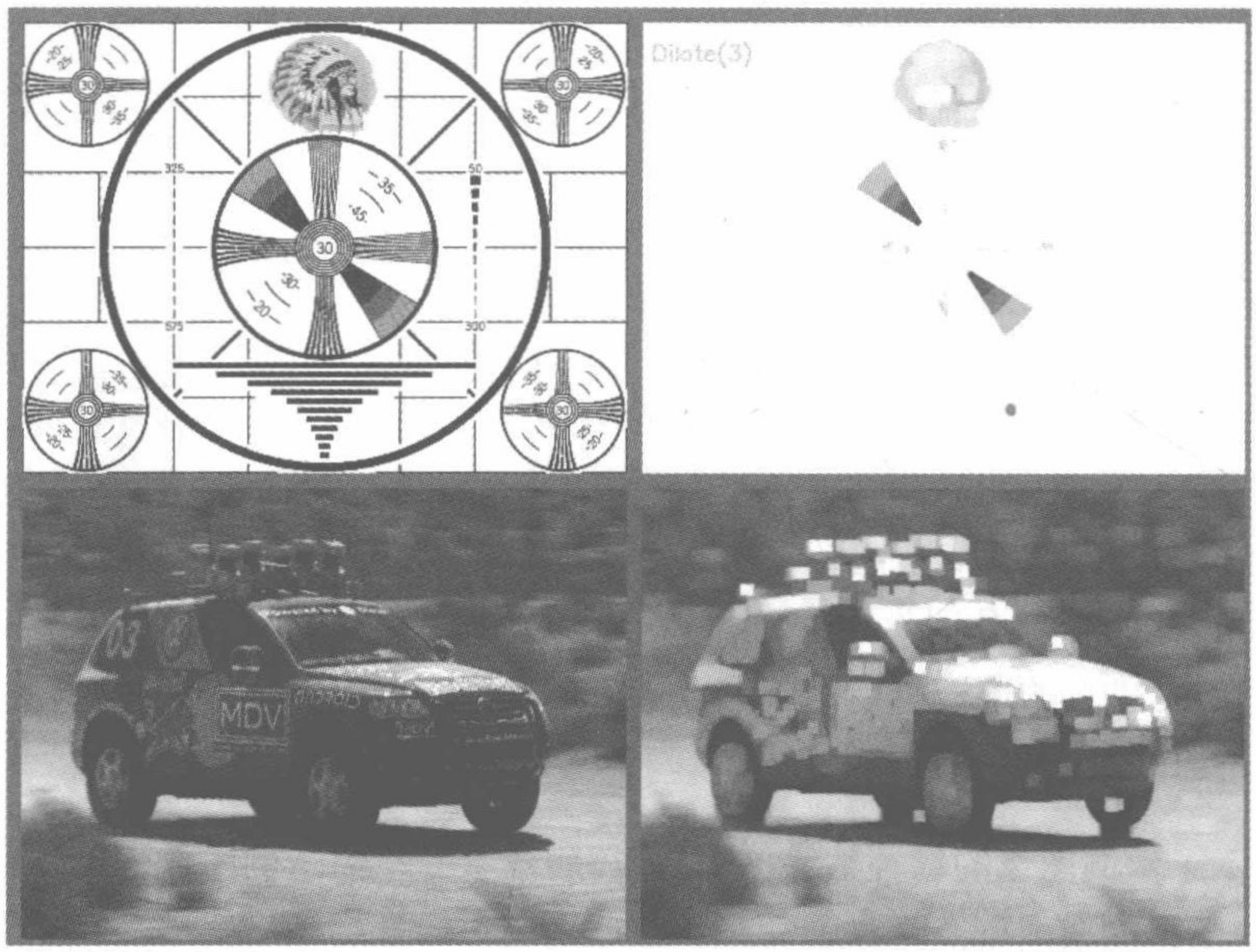


图10-22. 膨胀结果，或者说是“最大化”操作：明亮的区域扩张并连通

回顾一下：当OpenCV的函数cv::erode()运行时，函数内部发生的是将点p出的值设置成核与点p对齐后覆盖区域的最小值；对于cv::dilate()，是将最小值替换成最大值：

$$erode(x, y) = \min_{(i,j)\in kernel} src(x + i, y + j)$$

$$dilate(x, y) = \max_{(i,j)\in kernel} src(x + i, y + j)$$

你也许会好奇，为什么在有了这么多启发式的描述的情况下，还要给出这么复杂的式子。因为是一些读者比较倾向于阅读这样的公式，但更重要的是，这些式子表达的效果比定性描述更加一般化。注意，如果图像不是布尔型的，那么最小化和最大化操作的用处就不是那么有用。再回顾下图10-21和图10-22，分别显示了如何对一幅真实图像进行腐蚀和膨胀。

通用形态学函数

当处理的对象是二值图像和像素只可能是开（>0）或关（=0）的图像掩膜时，基本的腐蚀和膨胀操作就够用了。需要对灰度图或者彩色图进行处理时，一些其他的操作就非常有用，这些操作可以通过cv::morphologyEx()实现。

```
void cv::morphologyEx(
  cv::InputArray    src,                    // Input image
  cv::OutputArray   dst,                    // Result image
```

```
    int               op,                          // Operator (e.g. cv::MOP_OPEN)
    cv::InputArray    element,                     // Structuring element, cv::Mat()
    cv::Point         anchor      = cv::Point(-1,-1),  // Location of anchor point
    int               iterations  = 1,                 // Number of times to apply
    int               borderType  = cv::BORDER_DEFAULT // Border extrapolation
    const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()

);
```

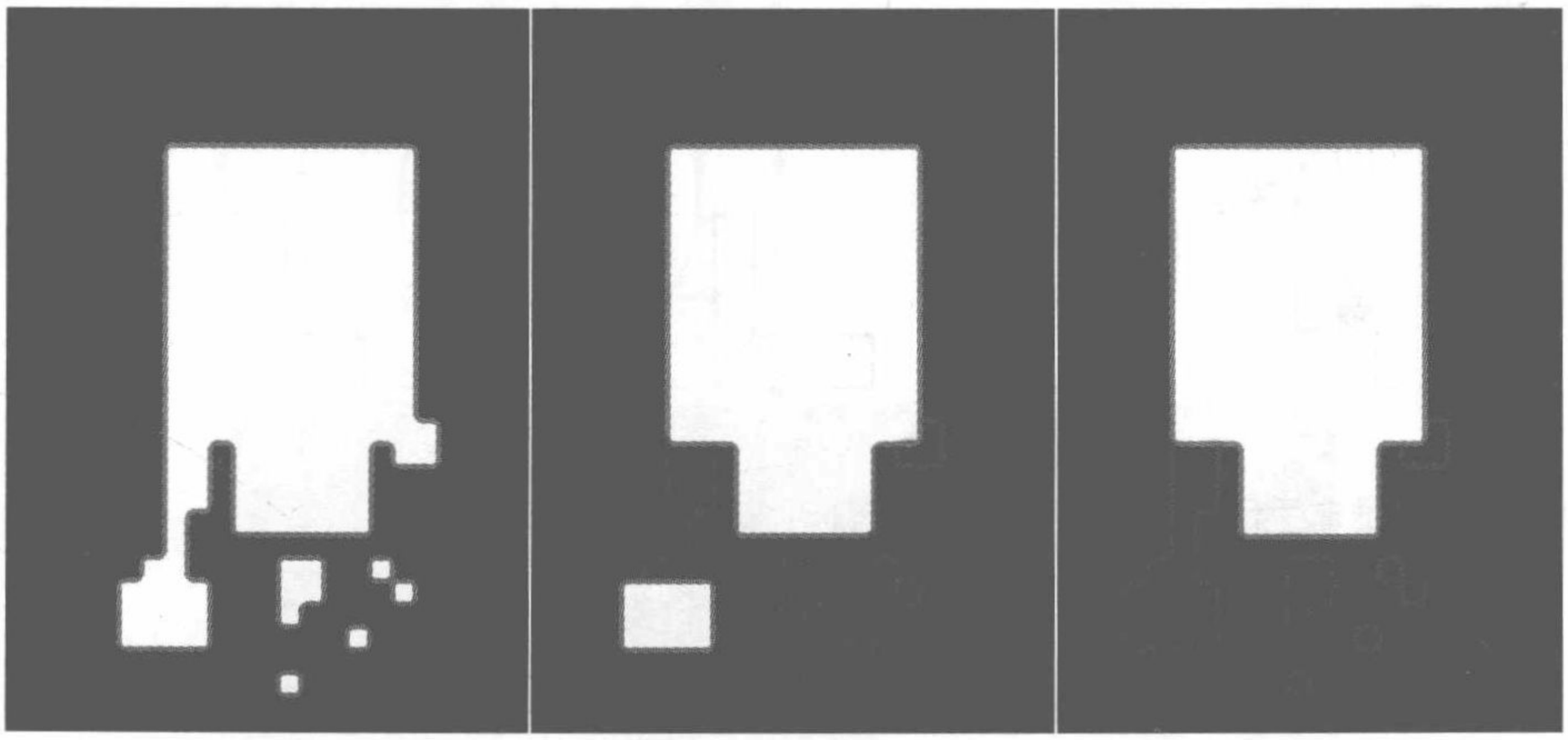
参数列表中除了我们在cv::dilate()和cv::erode()中看见的参数之外，还有一个新的但是十分重要的参数，也就是op。它用于制定函数将进行的操作，可选值如表10-3所示。

表10-3：cv::morphologyEx() 操作选项

操作值	形态学操作	是否需要临时图像
cv::MOP_OPEN	开操作	否
cv::MOP_CLOSE	闭操作	否
cv::MOP_GRADIENT	形态学梯度	总是需要
cv::MOP_TOPHAT	顶帽操作	就地调用需要 (src = dst)
cv::MOP_BLACKHAT	底帽操作	就地调用需要 (src = dst)

开操作和闭操作

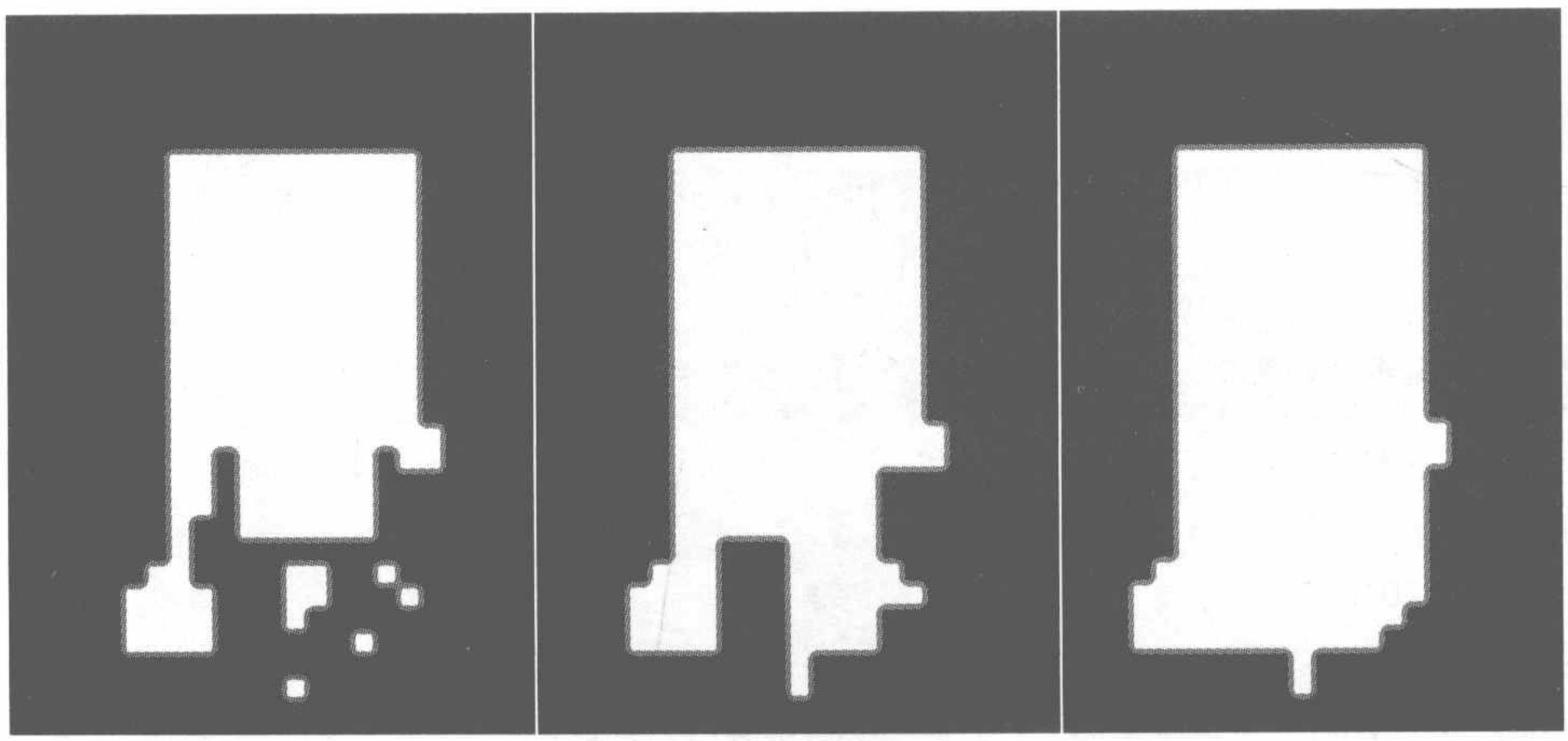
首先介绍的两种操作开操作和闭操作实际上是腐蚀和膨胀操作非常简单的组合。开操作先将图像腐蚀，然后对腐蚀的结果膨胀（图10-23）。开操作常用语对二值图像中的区域进行计数。举个例子，我们对显微镜载玻片上细胞的图像已经做了阈值化处理，在对区域进行计数之前，需要用开操作来将相互靠得很近的细胞分开。



(A) 源图像　(B) 开操作　(C) 二次开操作

图10-23. 对一幅简单的二值图像进行形态学开操作

闭操作先将图像进行膨胀，然后对膨胀的结果进行腐蚀（图10-24）。闭操作用于复杂连通分支算法中减少无用或噪声驱动的片段。对于连通分支，通常先进行腐蚀或闭操作消除噪声，然后通过开操作连接相互靠近的大型区域。（注意，尽管开和闭操作的效果与腐蚀和膨胀想近，但这些新操作在保持连通域的精度上更加可靠。）



(A) 源图像　　(B) 闭操作　　(C) 二次闭操作

图10-24. 对一幅简单的二值图像进行形态学闭操作

当对一幅非布尔型图像进行形态学操作时，闭操作最明显的效果是消除值小于邻域内的点的孤立异常值，而开操作消除的是大于邻域内点的孤立异常值。开操作的效果如图10-25和图10-26所示，闭操作效果如图10-27和图10-28所示。

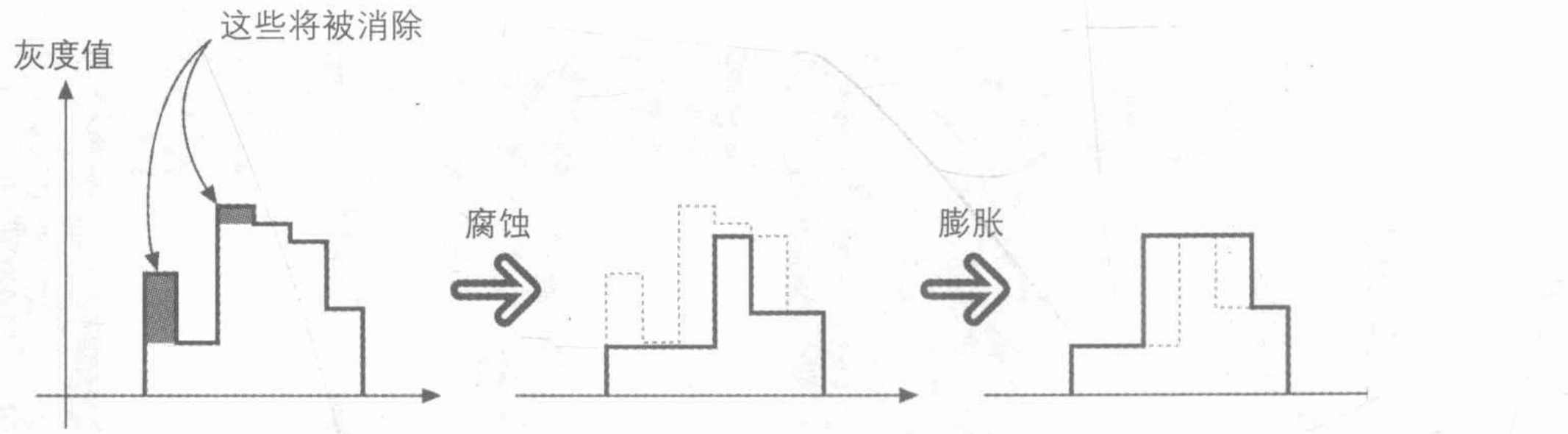


图10-25. 对一维非二值图像进行形态学开操作：较大的异常值被消除



图10-26. 对一幅图像进行形态学开操作：细微的明亮区域被消除，其余明亮区域被孤立，但大小保持不变

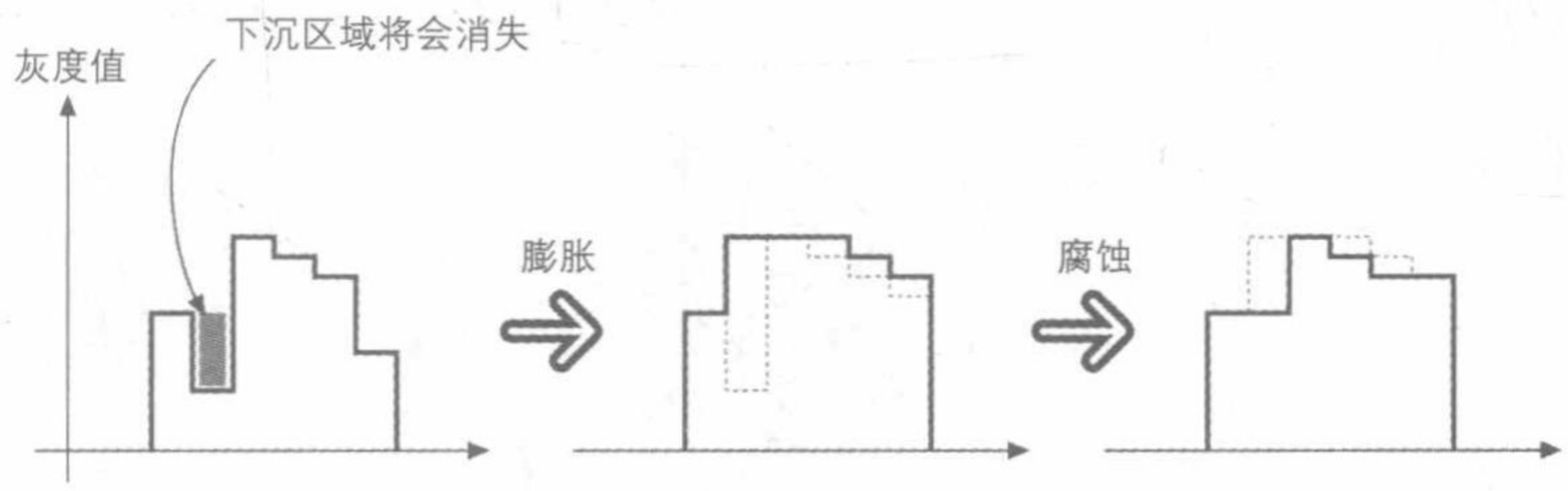


图10-27. 对一维非二值图像进行形态学闭操作：较小的异常值被消除

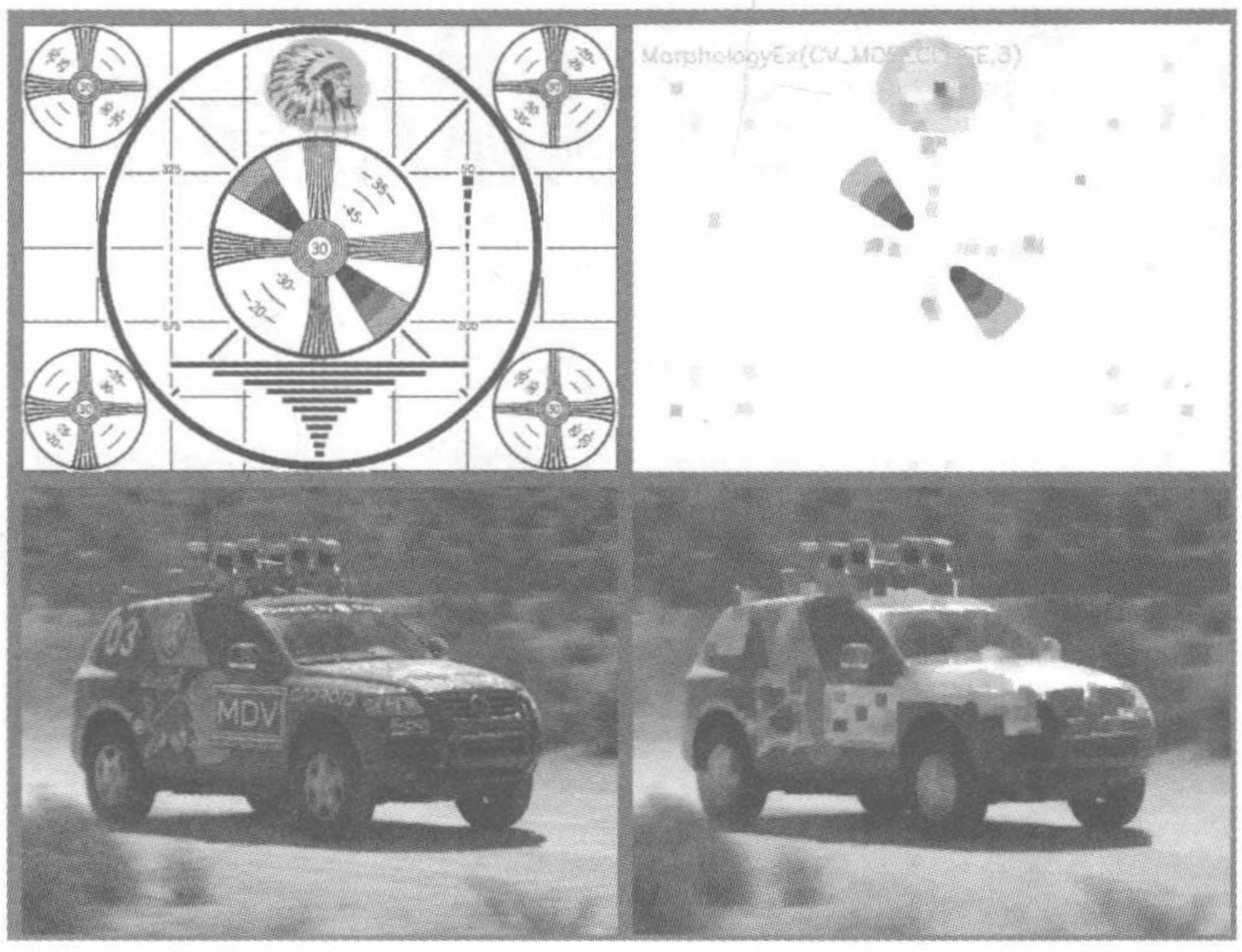


图10-28. 对一幅图像进行形态学闭操作：明亮区域被连通同时保持基本大小

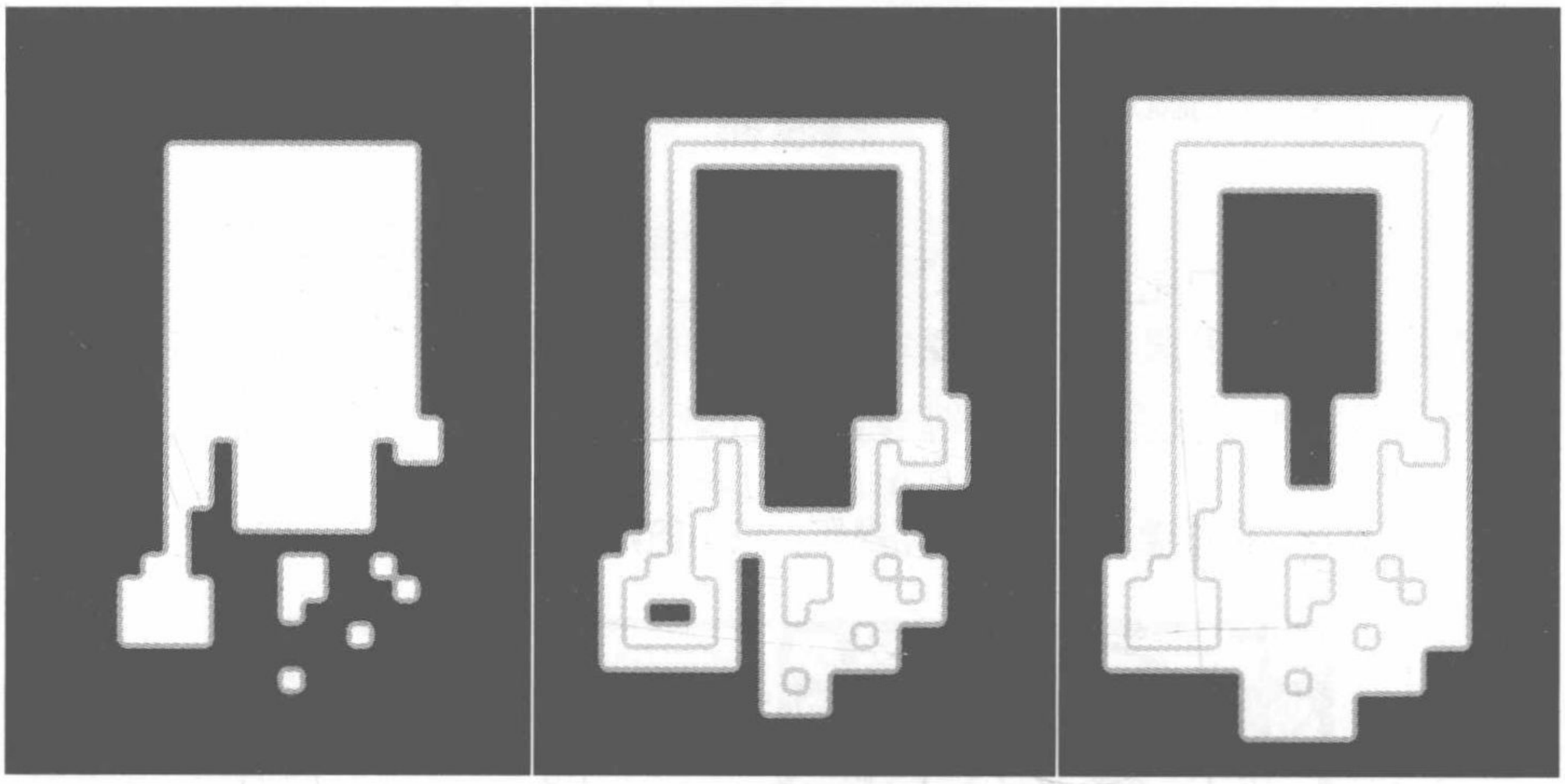
最后需要说明的是开操作和闭操作是iterations参数。可能你以为两次闭操作的迭代是膨胀－腐蚀－膨胀－腐蚀，但事实上这并没有什么用处。你真正需要的是膨胀－膨胀－腐蚀－腐蚀，并且实际上得到的也正是这个。这样做不但可以消除孤立的异常值，也能消除领域内异常值。图10-23(C)和图10-24(C)分别展示了迭代两次开和闭两种操作的结果。

形态学梯度

接下来要叙述的操作是形态学梯度（Morphological Gradient），可以用如下表达式描述这种方法：

$$gradient(\text{src}) = dilate(\text{src}) - erode(\text{src})$$

如图10-29所示，梯度操作的结果（扩张亮域）减腐蚀操作的结果（缩减亮域）产生了源图像中目标的边缘。



(A) 源图像　(B) 形态学梯度操作　(C) 二次操作

图10-29. 对简单二值图像进行形态学梯度操作

对于灰度图像（图10-30），其结果就是计算明暗变换的趋势，这说明将其命名为“形态学梯度”是有根据的。形态学梯度通常用于显示明亮区域的边界，然后便可以将他们看作目标或者目标的部分。用扩张的图像减去了收缩的图像，如此一来就找出了完整的边界。这与计算梯度不同，它不会关注某一个物体的周围。图10-31展示了形态学梯度操作的效果。

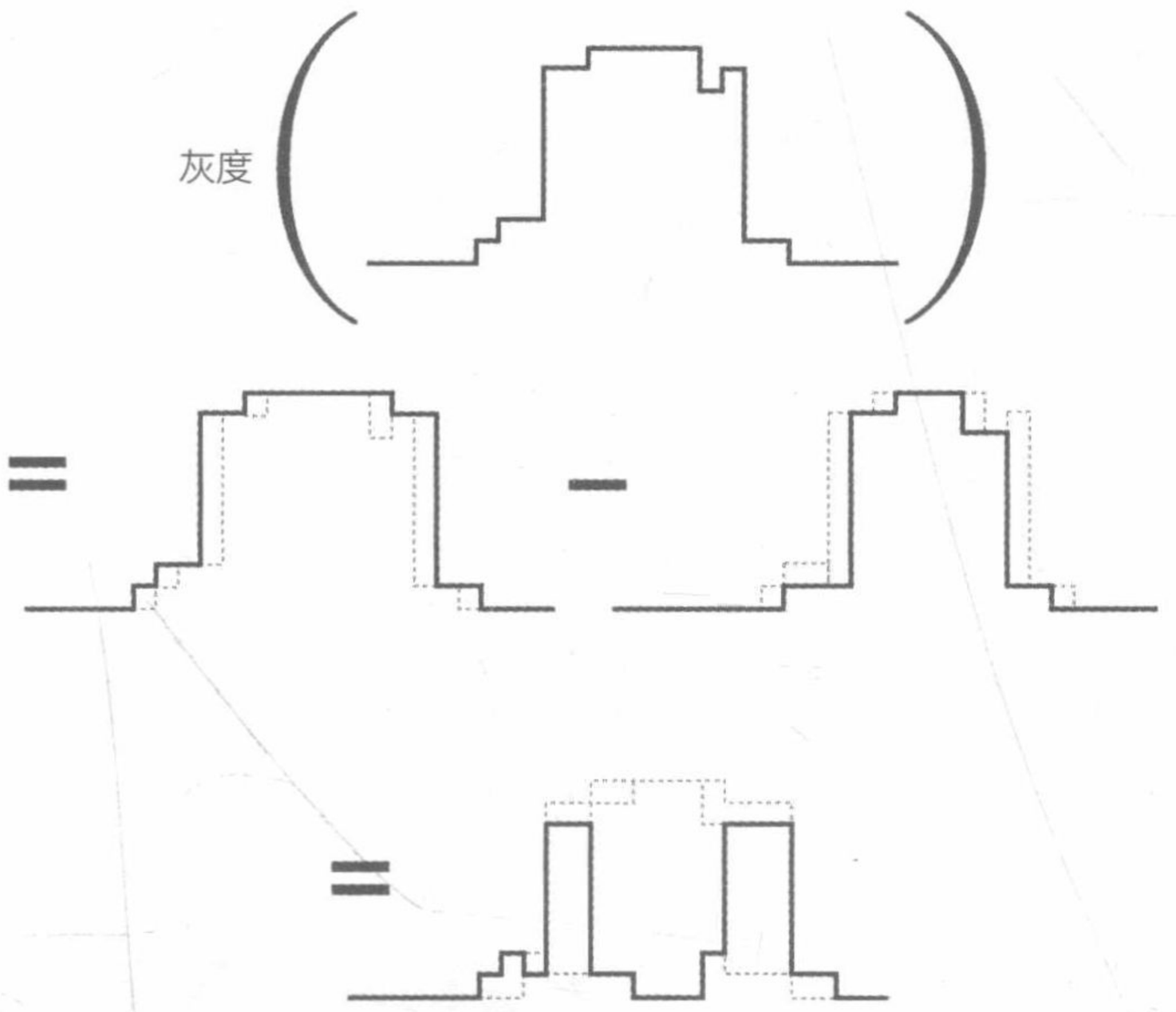


图10-30. 对以为非二值图像形态学梯度操作：果然，这个算子高亮了灰度图中变化剧烈的部分

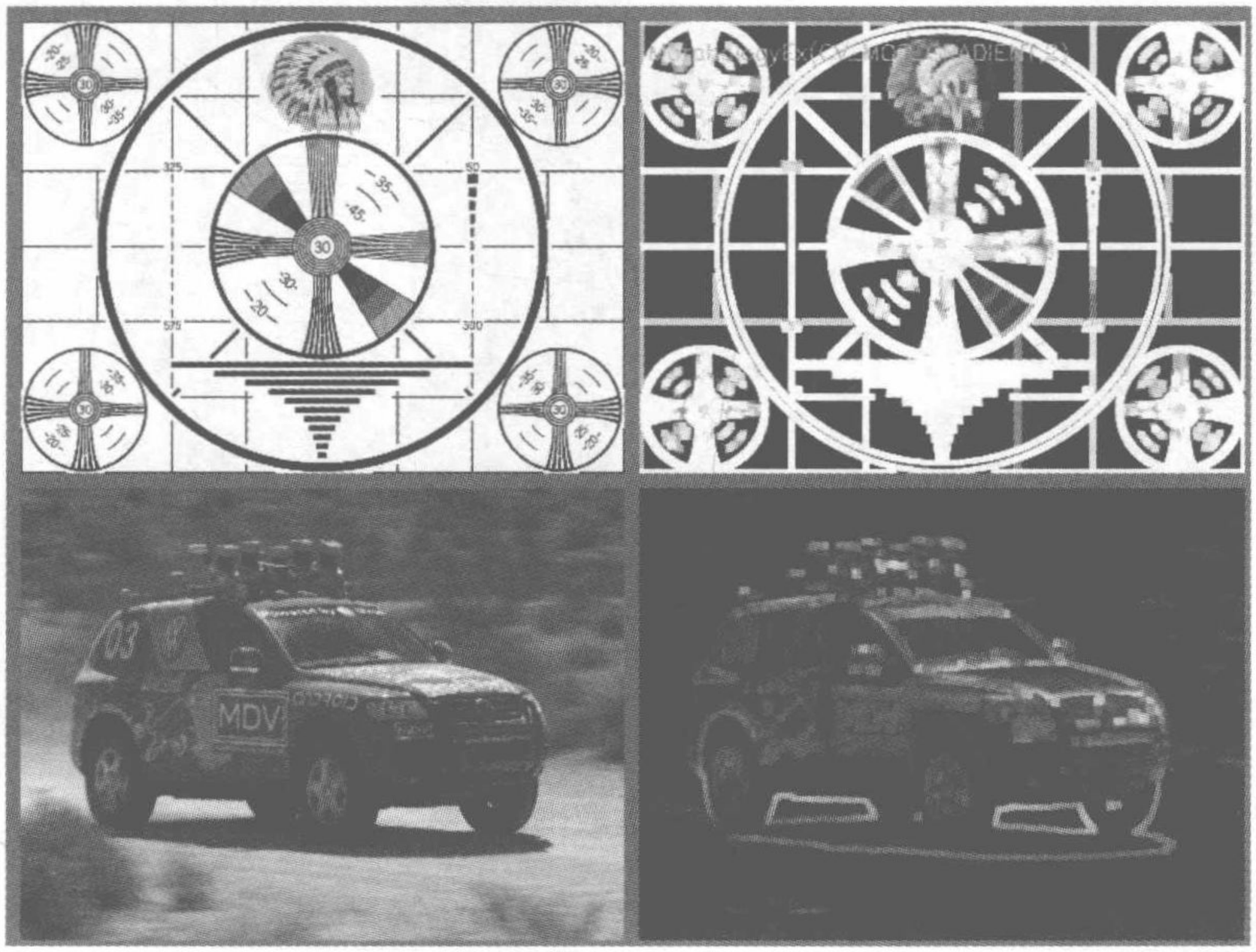


图10-31. 形态学梯度操作：明亮边缘被标注

顶帽和黑帽

最后两种操作是顶帽和黑帽[Meyer78]。这两种操作分别用于显示与其邻域相比更亮或更暗的部分。当试图根据物体的亮度变化分离依附于物体的某些部分时，就会用到这些方法。在生物体或细胞的显微镜图像上经常会用到这些方法。两种操作都是根据基础操作定义的，如下所示：

TopHat(src) = src – *open*(src) // Isolate brighter

BlackHat(src) = *close*(src) – src // Isolate dimmer

如你所见，顶帽操作用源图像减去对其开操作后的图像，回想一下，开操作的效果是放大裂缝和局部小洞。因此，用源图像减去对其开操作后的图像得到了比源图像更亮的环绕部分。相反，黑帽操作显示的是比源图像更暗的环绕部分。这章所讨论的形态学操作都在图10-18进行了总结。[注18]

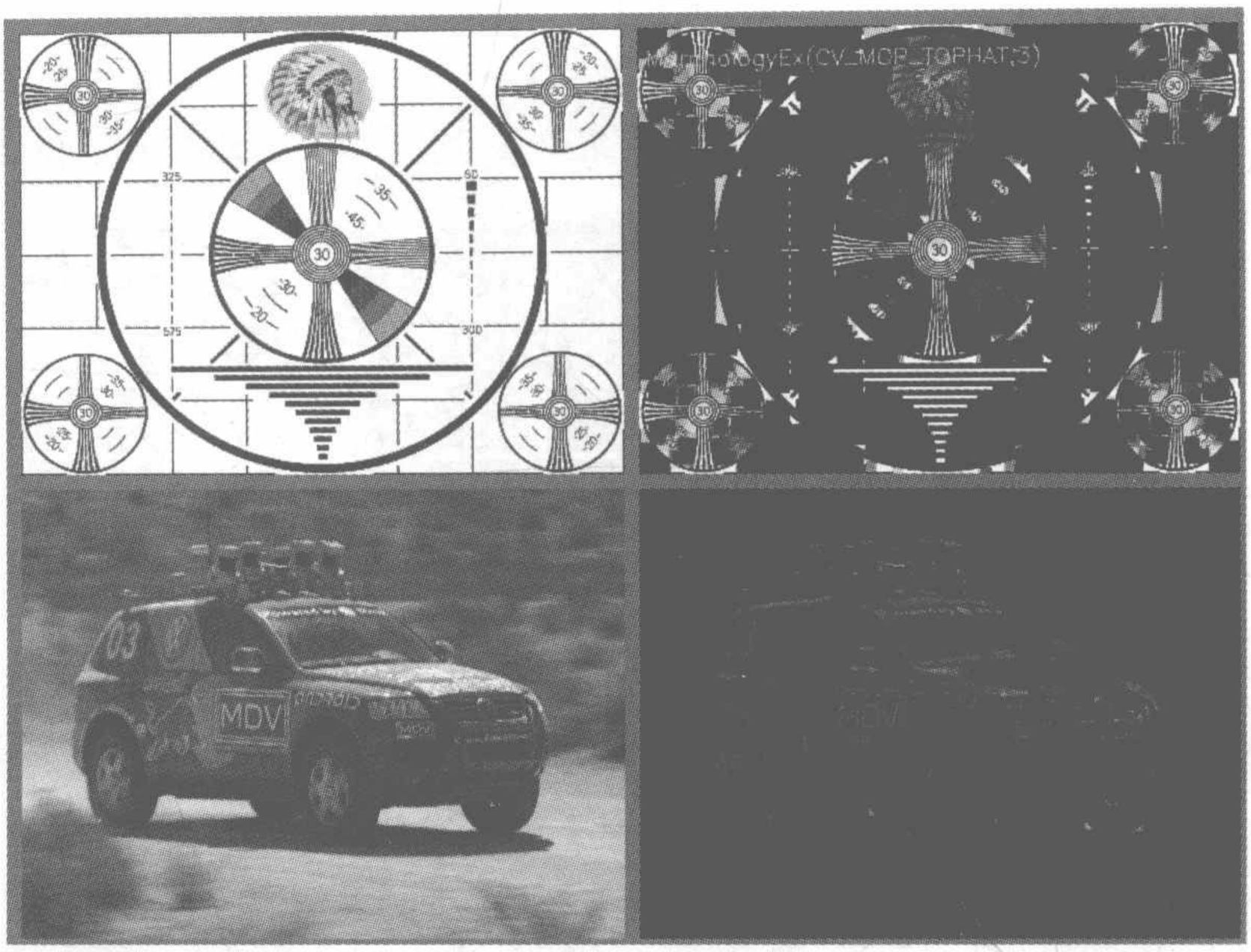
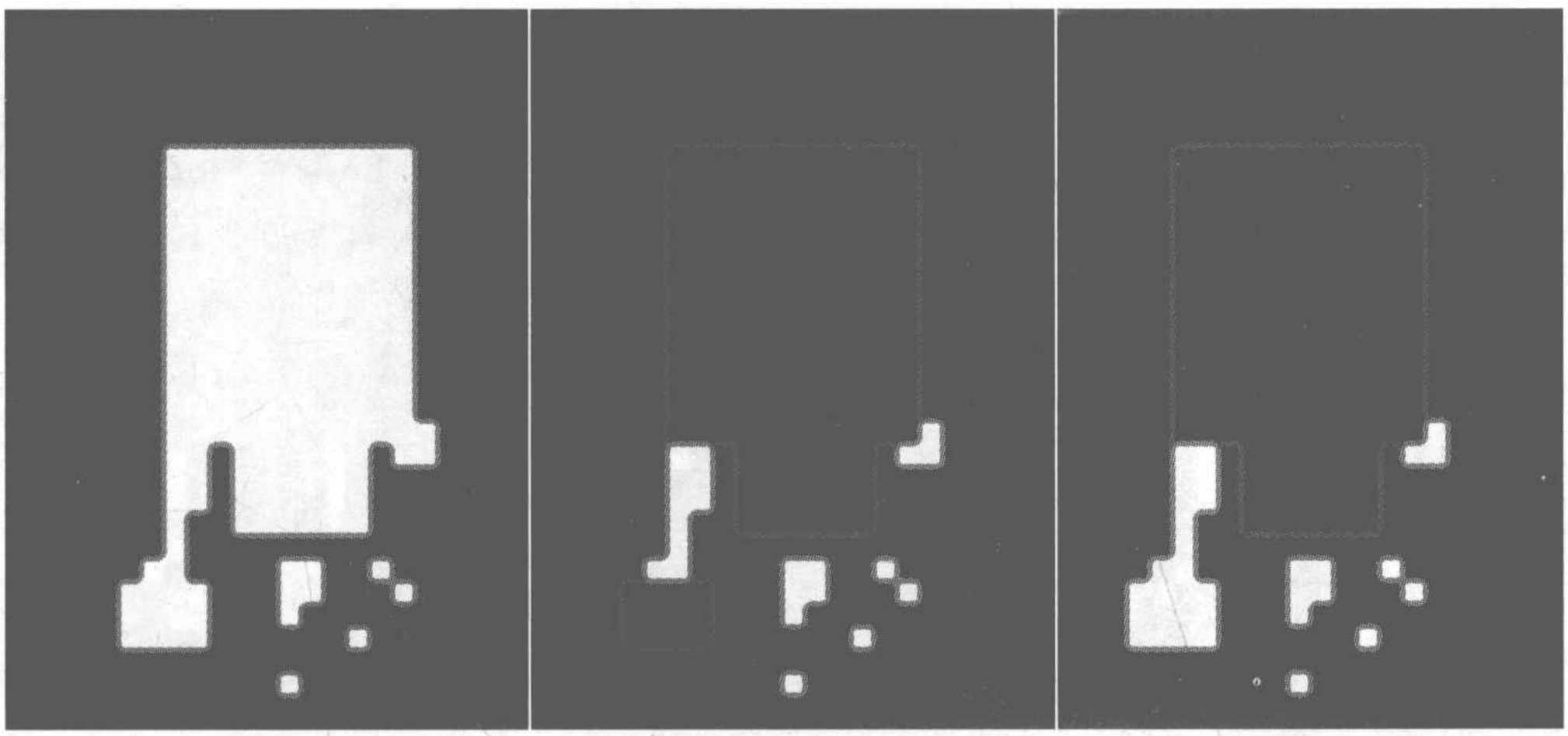


图10-32. 形态学顶帽操作：明亮的局部突起被提取出来

注18：顶帽操作和黑帽操作是灰度形态学中最有用的操作，其构造元素是一个实数矩阵而不是布尔掩模，并且在取最值之前，矩阵会加载当前像素点的邻域。虽然书中已经提到，但是，OpenCV中并没有实现这些功能。



(A) 源图像　　(B) 顶帽操作　　(C) 二次操作

图10-33. 对简单二值图像形态学顶帽操作

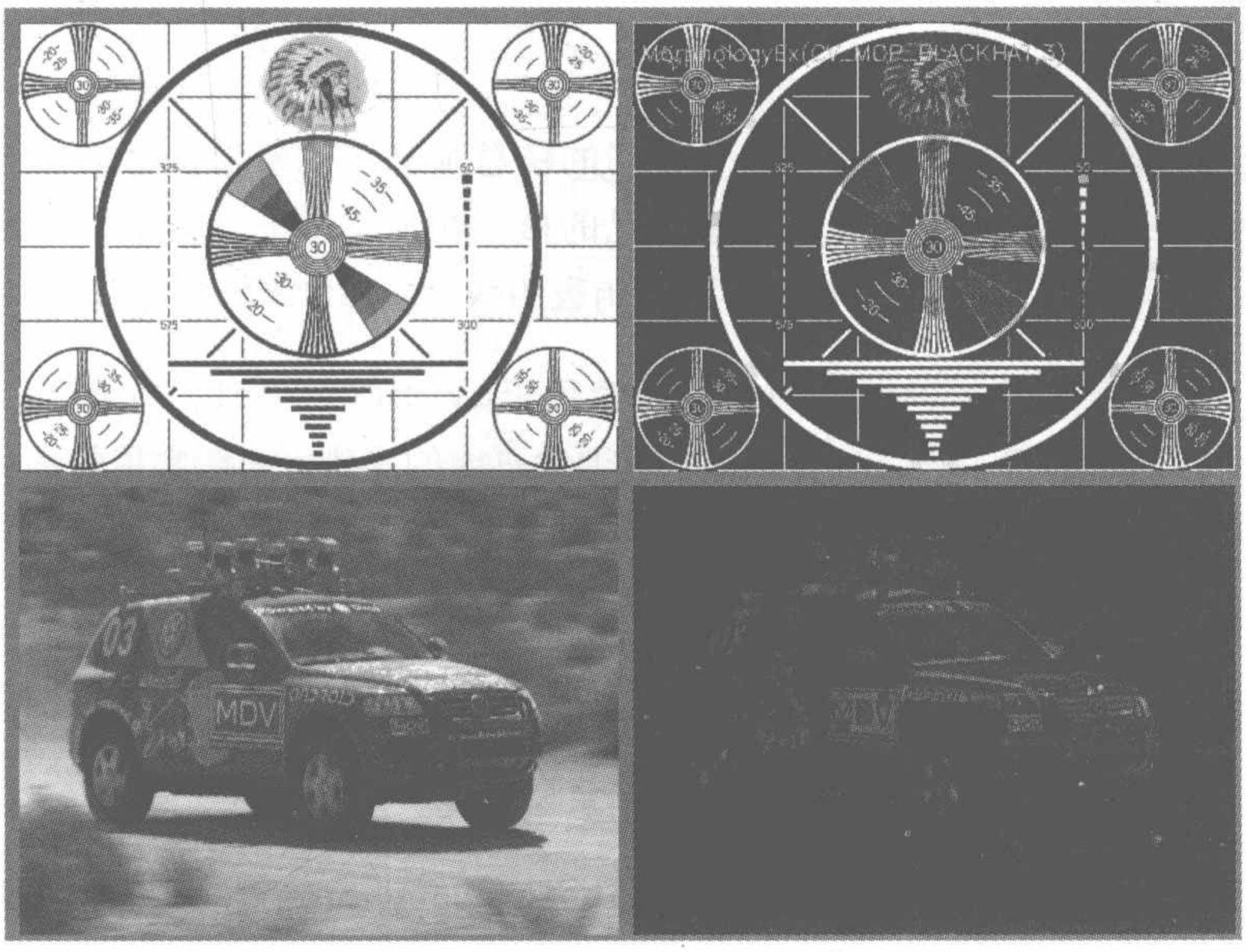
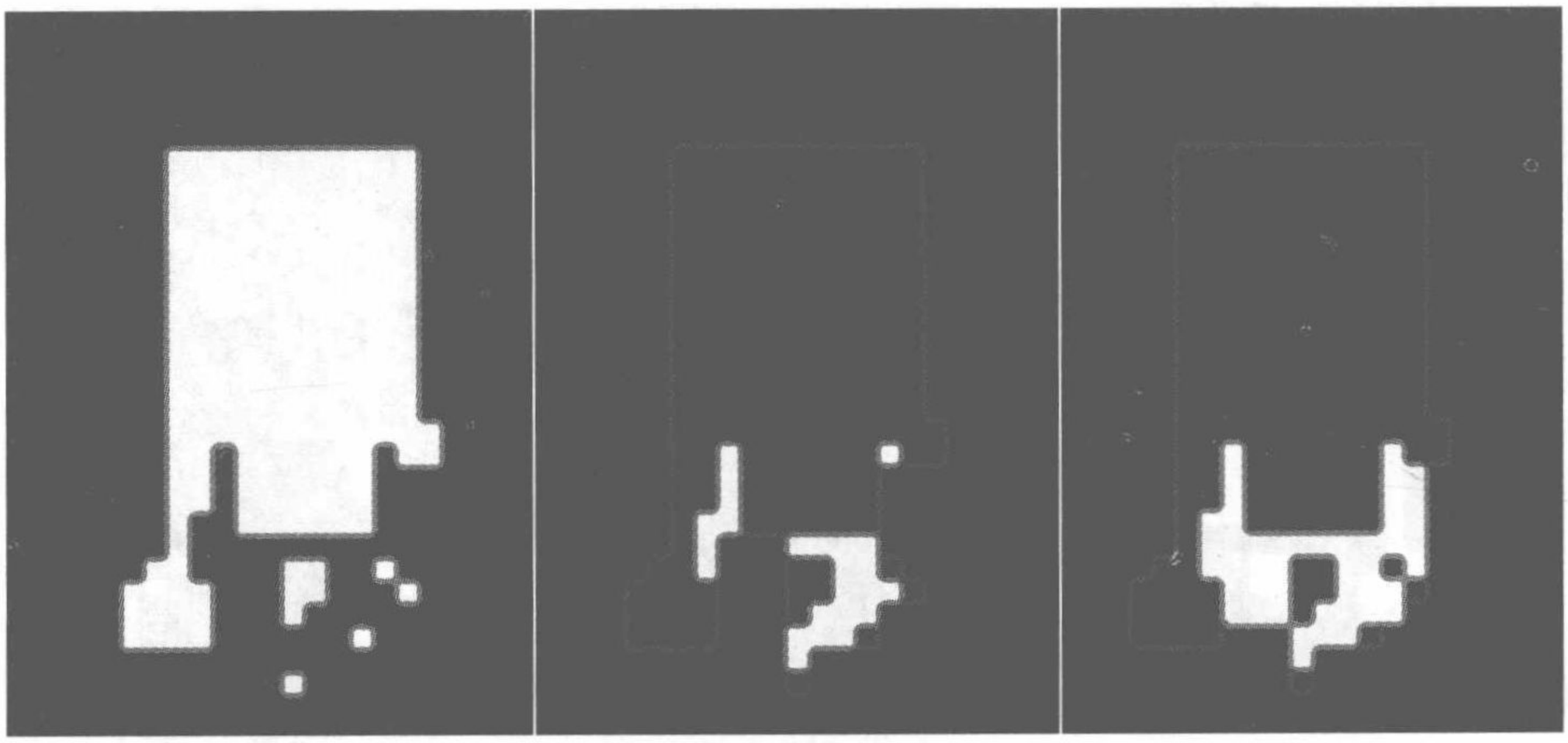


图10-34. 形态学黑帽操作：暗洞被提取出来



(A) 源图像　　(B) 黑帽操作　　(C) 二次操作

图10-35. 对简单二值图像形态学黑帽操作

自定义核

迄今为止我们所用到的形态学操作中，使用的核总是3×3的矩形。如果你需要一些比这更通用的核，OpenCV可以让你创建自己的核。在形态学上，核常常称为“构造元素”。因此供开发者创建自定义形态学核的函数叫cv::getStructuringElement()。

其实，你可以任意创建一种需要的构造元素用于cv::dilate()、cv::erode()或cv::morphologyEx()等，但这些行为通常需要做更多的操作，一般常用的是非矩形核是一种非常规形状。函数cv::getStructuringElement()如下所示：

```
cv::Mat cv::getStructuringElement(
  int       shape,                    // Element shape, e.g., cv::MORPH_RECT
  cv::Size  ksize,                    // Size of structuring element (odd num!)
  cv::Point anchor = cv::Point(-1,-1)  // Location of anchor point
);
```

第一个参数shape控制构造元素的基本形状；ksize和anchor确定的是元素的大小和锚点位置。通常，anchor默认为cv::Point(-1, -1)，这表示cv::getStructuringElement()默认锚点在元素中心。

表10-4：cv::getStructuringElement() 的元素形状

形状值	元素	描述
cv::MORPH_RECT	矩形	$E_{i,j} = 1, \forall\, i,j$
cv::MORPH_ELLIPSE	椭圆形	以 ksize.width 和ksize.height为两个半径做椭圆
cv::MORPH_CROSS	交叉	$E_{i,j} = 1$, 当i == *anchor.y* 或j == *anchor.x*



表10-4中的最后一项是为旧版接口留下的。在C API（v1.x）中，有一个单独的结构体用来表示卷积核。如今已经没有必要再用这个方式，因此如果cv::getSturcturingElement()不能满足需求，开发者可以将任意形式的cv::Mat作为形态学操作的构造元素。

用任意线性滤波器做卷积

到目前为止，我们所接触到的卷积都是在OpenCV中API内部实现了的。学习了一些卷积操作之后，就立即联系OpenCV中实现了相应功能的函数。在调用这些函数时，函数默认地选择了某一种核，我们做的只是向函数传递一些参数来调整这个核。在OpenCV中，实际是允许我们用一个真实存在的核进行卷积操作的。

理论上说，我们只要用一个数组表示一个核，然后放进一个函数，就可以用来卷积了。实际情况中，一些不起眼的地方却会在很大程度上影响到性能，可分解的矩阵通常会产生这种影响。

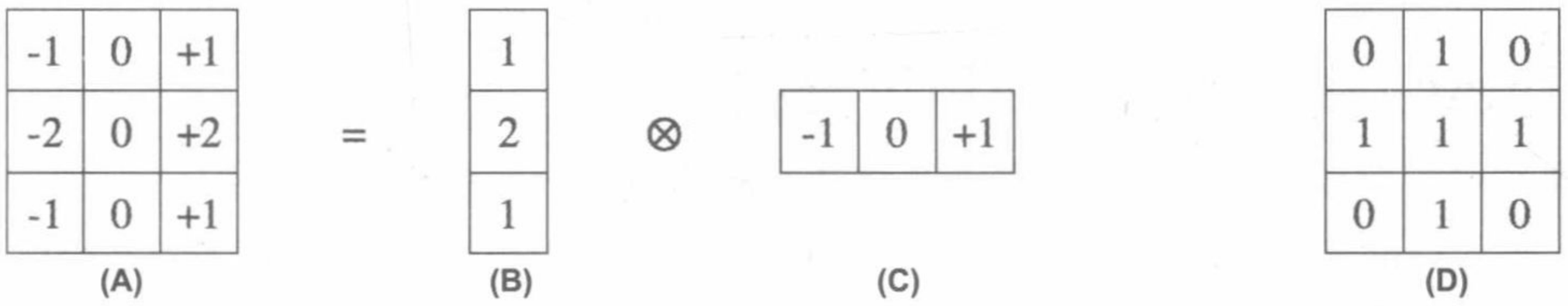


图10-36. Sobel核(A)可以分解成两个一维矩阵的卷积(B)和(C)，(D)是一个不能分解的核

一个可分核可以理解成两个一维核，在卷积时先调用x内核，然后再调用y内核。两个矩阵进行卷积所产生的消耗可以用两个矩阵的面积之积近似[注19]，如此一来，用$n \times n$的核对面积为A的图像进行卷积所需的时间时An^2，但如果分解成$n \times 1$和$1 \times n$的两个核，那么代价就是$An+An=2An$，因此分解卷积核可以提高提高卷积计算的效率。只要n不小于3，这种计算方式能提高效率，并且随着n的增大，这种收益愈发明显。

用cv::filter2D()进行卷积

对一幅图像卷积过程中的操作是十分巨大的，至少从第一印象来看[注20]操作数大概是图

注19：这个准则仅在空间域进行卷积时有效，这也是OpenCV在处理小型核的做法。

注20：我们使用“第一印象”是因为我们也可以在频域进行卷积操作。在这种情况下对于一个$n \times n$大小的图像以及$m \times m$大小的核（其中n远远大于m）在频域的计算复杂度将会接近$n^2\log(n)$而不是n^2m^2。因为频域的计算复杂度和卷积核的大小是无关的，所以其对于更大的核有更高的效率。OpenCV将会按照卷积核的大小自动决定是否要在频域进行卷积操作。

像中的像素数量乘以卷积核中的像素数，如此复杂的操作导致没有人愿意通过使用for循环和一堆指针去计算卷积。因此，有一个更好的选择是用OpenCV完成并优化这些操作。OpenCV中实现这个功能的是cv::filter2D()：

```
cv::filter2D(
  cv::InputArray  src,                                  // Input image
  cv::OutputArray dst,                                  // Result image
  int             ddepth,                               // Output depth (e.g., CV_8U)
  cv::InputArray  kernel,                               // Your own kernel
  cv::Point       anchor     = cv::Point(-1,-1),        // Location of anchor point
  double          delta      = 0,                       // Offset before assignment
  int             borderType = cv::BORDER_DEFAULT       // Border extrapolation to use
);
```

使用这个函数时，先创建一个适当大小的数组，然后设置好内部的系数，然后将它和源图像以及目标图像一起传入cv::filter2D()。像之前一样，可以通过参数anchor设置锚点位置，通过borderType设置边框外推方式。如果定义了锚点位置，那么核的大小可以为偶数，否则必须是奇数。如果还希望滤波的结果与源图像产生一个偏移，那么可以对delta参数指定一个数值。

通过cv::sepFilter2D使用可分核

当参与卷积的核是可分核时，那么先将其分解为两个一维核然后传递到OpenCV中计算，会获得最佳的计算性能（如用图10-36(B)核10-36(C)替代图10-36(A)）。OpenCV中函数cv::sepFilter2D()与cv::filter2D()相似，只是在核的接受上有些差异。

```
cv::sepFilter2D(
  cv::InputArray  src,                                  // Input image
  cv::OutputArray dst,                                  // Result image
  int             ddepth,                               // Output depth (e.g., CV_8U)
  cv::InputArray  rowKernel,                            // 1-by-N row kernel
  cv::InputArray  columnKernel,                         // M-by-1 column kernel
  cv::Point       anchor     = cv::Point(-1,-1),        // Location of anchor point
  double          delta      = 0,                       // Offset before assignment
  int             borderType = cv::BORDER_DEFAULT       // Border extrapolation to use
);
```

除了用rowKernel和columnKernel替换kernel参数外，cv::sepFilter2D()的所有参数与cv::filter2D()的参数相同。两个小核的大小应当是$n_1 \times 1$核$1 \times n_2$，n_1和n_2不一定相等。

生成卷积核

接下来介绍几个用于生成常用卷积核的函数，cv::getDerivKernel()用于生成Sobel核和Scharr核，cv::getGaussianKernel()用于生成高斯核。

cv::getDerivKernel()

导数滤波器的核实际上是通过cv::getDerivKernel()生成。

```
void cv::getDerivKernels(
  cv::OutputArray kx,
  cv::OutputArray ky,
  int             dx,                 // order of corresponding derivative in x
  int             dy,                 // order of corresponding derivative in y
  int             ksize,              // Kernel size
  bool            normalize = true,   // If true, divide by box area
  int             ktype     = CV_32F  // Type for filter coefficients
);
```

cv::getDerivKernel()的结果放在数组kx和ky中。你应该还有印象，导数核（Sobel和Scharr核）是可以分解的。这也是函数返回两个核的原因，返回的两个核中，一个大小为1×ksize的行向量，和ksize×1的列向量。这两个数组由x、y方向的求导顺序dx和dy计算而来。导数核永远都是正方形的，因此ksize是一个整数。Ksize可以是1、3、5、7或cv::SCHARR。Normalize参数决定cv::getDerivKernel()是否将核元素规范化。如果目标是一个浮点型图像，就应当将normalize设置为真；但如果目标是个整型数组，就不要将它设置成真，以免导致精度丢失。[注21]最后一个参数ktype表示滤波器的类型（默认kx和ky的类型）。ktype可以使CV_32F和CV_64F。

cv::getGaussianKernel()

高斯滤波器所使用的核由cv::getGaussianKernel()生成。

```
cv::Mat cv::getGaussianKernel(
  int             ksize,              // Kernel size
  double          sigma,              // Gaussian half-width
  int             ktype = CV_32F      // Type for filter coefficients
);
```

与导数核相同，高斯核也是可分的。因此，cv::getGaussianKernel()计算一个ksize×1的数组作为结果。ksize是一个奇数，sigma是近似高斯分布的标准差。高斯分布矩阵的系数通过下式计算而来：

$$k_i = \alpha \cdot e^{-\frac{(i-(ksize-1)2)^2}{(2\sigma)^2}}$$

系数α在滤波器需要规范化的时候才起作用。sigma可以是-1，这样将自动计算ksize。[注22]

注21：如果真的要这么做，应当使用另一个规范化系数$2^{ksize*2-dx-dy-2}$。

注22：这个示例中，$\sigma = 0.3 \cdot \left(\frac{ksize-1}{2} - 1\right) + 0.8$。

小结

这一章中，我们学习了图像卷积，包括了图像卷积时边界处理的重要性、核以及线性核与非线性核的区别，OpenCV实现的一些基本图像滤波器及其区别等。

练习

1. 读取一幅带有纹理的图像，使用cv::GaussianBlur平滑。
   a. 使用对称3×3、5×5、9×9和11×11大小的窗口平滑并显示结果。
   b. 通过用5×5高斯滤波器平滑图像两次，输出结果是否和11×11滤波器平滑一次几乎相同？为什么？
2. 创建一个100×100单通道图像。将所有像素都设置为0。最后，将中心像素设置为255。
   a. 使用5×5高斯滤镜平滑该图像并显示结果，你发现了什么？
   b. 使用9×9高斯滤波器再次平滑。
   c. 通过用5×5高斯滤波器平滑图像两次，结果会怎样？结果是否和9×9滤波器次几乎相同？为什么？
3. 载入一幅图像，通过cv::GaussianBlur()用高斯滤波器模糊。
   a. 设置param1=param2=9。尝试几个不同的param3（例如1、4和6），显示结果。
   b. 在将param3设置为1、4和6之前，设置param1=param2=0，显示结果。它们有什么区别吗？为什么？
   c. 再次使用param1=param2=0，但这次设置param3=1和param4=9，平滑图片并显示结果。
   d. 使param3=9和param4=1，重复练习3c，显示结果。
   e. 平滑图像，一次使用练习3c的设置，另一次使用练习3d的设置，显示结果。
   f. 将练习3e中的结果与使用param3=param4=9和param3=param4=0（即9×9滤波器）的平滑结果进行比较。结果是一样的吗？为什么？
4. 用相机在同一个场景中拍摄两张照片，尽量不要移动相机。将两幅图像载入到src1和src2中。
   a. 取src1-src2的绝对值，赋值到diff12并显示。如果不出意外，diff12是黑色的吗？为什么不？

b. 对diff12进行cv::erode()和cv::dilate()并赋值到cleandiff，将结果显示出来。

c. 对diff12进行cv::dilate()和cv::erode()并赋值到dirtydiff，将结果显示出来。

d. 解释cleandiff和dirtydiff的差别。

5. 为一个物体创建轮廓。拍一张照，然后不要移动相机，在镜头中放一杯咖啡，在拍一张照片。加载这些图片并转换成8位灰度图。

a. 取两幅图像差的绝对值，显示结果，看起来应该是一个不清晰的咖啡杯。

b. 取一个噪声与咖啡杯像素之间的值作为阈值对图像二值化，保留大部分咖啡杯的像素同时去除部分噪声，显示结果。“on”设为255。

c. 对图像开操作进一步清除噪声。

d. 腐蚀图像，然后与源图像异或，获取咖啡杯的边缘轮廓（仅剩余边缘像素）。

6. 高动态范围：在一个有强光照射但桌子能够遮光的房间拍摄一张照片。大多数相机都会拍到这样的照片：光照的地方曝光良好，但阴影区太暗；或者阴影区曝光正常，但光照的地方曝光过度。创建一个自适应滤波器调节图像的平衡，也就是提升灰暗区域的强度，降低明亮区域的强度。

7. 天空滤波器：创建一个自适应“天空”的滤波器，使其只平滑天空或湖面，不平滑地面。

8. 在噪声中创建一个掩膜。完成练习5后，保留图像中最大的剩余形状，用一个指针指向图像的左上方然后遍历图像。记下值为255（“开”）的地方，用100填充这个位置。读取这个位置所在的连通分支，记录它的面积。如果图像中还有更大的区域，就用0填充这个区域，并删除记录的面积。如果新区域大于上一个区域，就用0填充前一个区域，并删除前一个位置。最后，用255填充最大区域，显示结果，就看到了一个实心咖啡杯的掩膜。

9. 载入一幅室外环境的照片，使用练习8或者你自己创建（画图，或者一个简单的正方形）的掩膜，调用掩膜的copyTo()将照片的杯子复制到场景中。

10. 创建一个低方差随机图像（平均值为0，偏差不超过3）。用画图软件比如PowerPoint打开这幅图像，在图上的点周围画圈。用双边滤波器处理图像，并解释结果。

11. 加载一幅图像并转换成灰度图。

a. 对图像形态学顶帽操作，显示图像。

b. 将图像转换成8位掩膜。

c. 将b的结果灰度值复制到源图像中，只复制顶帽掩膜不为0的位置。显示结果。

12. 载入一幅细节很多的图像。

    a. 用resize()将每个维度减小至原来的一半（图像变成原来的1/4）。重复三次然后显示结果。

    b. 用cv::pyrDown()缩小源图像三次，显示结果。

    c. a和b的结果有什么区别？这两种方法为什么不同？

13. 载入一副细节很多的图像，使用cv::threshold()，设置阈值为128，用图10-4中每种方法对图像进行处理并显示结果。这个功能你应该非常熟悉，它非常有用。

    a. 用cv::adaptiveThreshold()重复练习，设置param1=5。

    b. 用param1=0和param1=-5重复a。

14. 近似双边（边缘保持）平滑滤波器。找出主要的边并取出来。用cv::pyrMeanShiftFiltering()分割图像，对每个区域进行平滑，然后将这些平滑的区域与边缘图像组合起来，结果平滑了图像并且保持了边缘。

15. 用cv::filter2D()创建一个滤波器匹配一幅细节较多的图中60度的线，显示结果。

16. 可分核：用[(1/16, 2/16, 1/16), (2/16, 4/16, 2/16), (1/16, 2/16, 1/16)]创建锚点在中心的3×3高斯核。

    a. 在一幅图上使用这个核并显示图像。

    b. 创建两个锚点在中间的一维核：一个是行向量（1/4, 2/4, 1/4），一个是列向量（1/4, 2/4, 1/4）。载入同一幅源图像，用cv::filter2D()对图像分别依次与两个一维核卷积。描述一下结果。

    c. 请描述a和b中核的复杂度（操作次数）。因为卷积是一个线性运算，所以两者之间的差别是使用可分核或整个高斯滤波器——或其他线性可分核的好处。

17. 是否可以用图10-15所示的Scharr滤波器产生一个可分核？如果可以，请画出来。

18. 用PowerPoint一类的软件画一些形如牛眼的同心圆。

    a. 用一些线穿过牛眼，保存图片。

    b. 用3×3大小的窗口进行一阶x、y导并显示，然后逐渐将窗口增大到5×5，9×9和13×13，并描述结果。

19. 创建一幅背景为黑色，只有一条45度白线的图像。对于给定的一些列孔径，对图像求一阶导dx和dy。并对图像做如下测量：dx和dy组成图像的梯度，图像在位置（i, j）处的梯度$mag(i, j) = \sqrt{dx^2(i, j) + dy^2(i, j)}$，角度是$\Theta(i, j) = atan2(dy(i, j), dx(i, j))$。扫描图像并找到梯度最大的地方并记录下来，将角度平均并记为直线的角度。

a. 用3×3的Sobel滤波器重复一次。

b. 用5×5重复一次。

c. 用9×9重复一次。

d. 结果是否会发生变化？如果发生了变化，为什么？

第11章

常见的图像变换

概览

在前几章中，我们介绍了可以从卷积方面具体理解的一类图像转换。当然，有很多有用的操作不能以这种方式表达（比如，用一个小窗口在图像上进行扫描，并进行各种各样的操作）。通常，可以表示为卷积的变换是局部的，这意味着即使它们可能改变整个图像，对任何特定像素的影响也仅由其周围的少量像素确定。我们在本章中看到的变换一般不会有此属性。

一些非常有用的图像变换比你想象的要简单得多，并且你会经常用到它们，例如，调整大小。其他的有更专门的用途。我们将在本章中看到的图像变换将一个图像转换成另一个图像。通常来说，输出图像与输入图像相比，大小不同，或是在其他方面不同，但它本质仍然是“一张图片”，与输入图像的意义相同。在第12章中，我们将考虑将图像渲染成一些可能完全不同的表示形式的操作。

在计算机视觉中有许多有用的变换。OpenCV提供了一些更常见的实现以及构建块以帮助你实现自己的、更复杂的变换。

拉伸、收缩、扭曲和旋转

我们遇到的最简单的图像变换是调整图像大小，使其变大或变小。这些操作比你想象的稍微复杂一些，因为调整大小也带来了像素如何插值（放大）或合并（减少）的问题。

均匀调整

我们经常遇到一些尺寸的图像，我们想转换成其他尺寸。我们可能想要增大或缩小图像，这两个任务都是可以通过相同的函数实现的。

cv::resize()

cv::resize()函数处理所有这些调整大小的需求。我们提供输入图像和我们想要转换的大小，它将生成大小合意的新图像。

```
void cv::resize(
  cv::InputArray  src,                                // Input image
  cv::OutputArray dst,                                // Result image
  cv::Size        dsize,                              // New size
  double          fx              = 0,                // x-rescale
  double          fy              = 0,                // y-rescale
  int             interpolation = CV::INTER_LINEAR // interpolation method
);
```

我们可以通过两种方式指定输出图像的大小。一种方法是使用绝对尺寸；在这种情况下，dsize参数直接设置为我们想要的结果图像dst的大小。另一个选择是使用相对尺寸；在这种情况下，我们将dsize设置为cv::Size(0,0)，并分别将fx和fy设置为我们要应用于*x*轴和*y*轴的比例因子。[注1]最后一个参数是插值方法， 默认为线性插值。其他可用选项如表11-1所示。

表11-1：cv::resize()插值选项

插值	含义
cv::INTER_NEAREST	最近邻插值
cv::INTER_LINEAR	双线性插值
cv::INTER_AREA	像素区域重采样
cv::INTER_CUBIC	双三次插值
cv::INTER_LANCZOS4	插值（超过8 × 8个邻域）

插值是这里的一个重要问题。源图像中的像素位于一个整数网格上，例如，我们可以在位置(20,17)中引用一个像素。当这些整数位置映射到新图像时，可能存在间隙，因为整数源像素位置映射到目标图像中的浮点位置，并且必须舍去到最接近的整数像素位置，或者因为有一些位置没有像素映射（考虑通过拉伸来加倍图像尺寸；然后每个其他目标像素将留空）。这些问题通常称为“前向投影问题”。为了处理这样的舍去问题和目标间隙，我们实际上是向后解决问题：逐步浏览目标图像的每个像素，并询问：“填充这

注1： dsize必须是cv::Size(0,0)或fx和fy都必须为0。

个目标像素需要哪些像素？”这些源像素将几乎总是在分数像素位置，所以我们必须内插源像素，导出我们的目标值的正确值。默认方法是双线性插值，但你可以选择其他方法（如表11-1所示）。

最简单的方法是从源图像中最接近的像素获取调整大小的像素的值，这是选择内插值cv::INTER_NEAR EST的效果。或者，我们可以根据它们到目标像素的接近度来线性加权2×2的源像素值，这是cv::INTER_LINEAR所做的。我们还可以将新的调整大小的像素放置在旧像素上，然后平均覆盖像素值，如cv::INTER_AREA所做。[注2]为了更光滑的内插，我们可以选择在源图像中的4×4周围像素之间拟合三次样条，然后从拟合的样条中读出相应的目标值，这是选择cv::INTER_CUBIC插值方法的结果。最后，我们有Lanczos插值，它与三次方法类似，但是使用的是像素周围的8×8区域的信息。[注3]



注意，cv::resize()和cv::Mat类的类似命名的cv::Mat::resize()成员函数之间的区别很重要。cv::resize()创建一个不同大小的新图像，原始像素映射到该图像上。cv::Mat::resize()成员函数调整的是正在调用其成员的图像大小，并将图像裁剪为新的大小。在cv::Mat::resize()的情况下，像素不会插值（或外推）。

图像金字塔

图像金字塔[Adelson84]广泛应用于各种视觉应用中。图像金字塔是图像的集合，它由单个原始图像产生，连续降采样，直到达到一些期望的停止点。此停止点可能是单像素图像！

文献和应用中经常出现两种图像金字塔：高斯[Rosenfeld80]和拉普拉斯[Burt83]金字塔[Adelson84]。高斯金字塔用于降采样图像，当我们要从金字塔中较低的图像重构上采样图像时，需要拉普拉斯金字塔（稍后讨论）。

cv::pyrDown()

通常，我们通过首先将层G_i与高斯核卷积，然后去除每个偶数行和列，从而生成金字塔的层G_i生成高斯金字塔（我们表示该层G_{i+1}）中的层（$i + 1$）。当然，在这种情况

注2：至少这种情况会发生在cv::resize()缩小图像时。扩展图像时，cv::INTER_AREA与cv::INTER_NEAREST相同。

注3：Lanczos滤波器的技巧超出了本书的范围，但是这种滤波器通常用于处理数字图像，因为它具有增加图像感知清晰度的效果。

下，每个图像恰好是其前身的四分之一。在输入图像G_0上迭代该过程产生整个金字塔。OpenCV为我们提供了一种从其前身产生每个金字塔阶段的方法：

```
void cv::pyrDown(
  cv::InputArray src,                          // Input image
  cv::OutputArray dst,                         // Result image
  const cv::Size& dstsize = cv::Size()         // Output image size
);
```

如果将目标大小参数dstsize设置为默认值cv::Size()，cv::pyrDown()方法就会为我们做到这一点。更具体一点，输出图像的默认大小是((src.cols+1)/2,(src.rows+1)/2)。[注4]另外，我们可以提供一个dstsize，它将指示我们想要输出图像的大小，但是，dstsize必须有一些非常严格的限制。具体如下：

$$|\, dstsize.width*2 \quad src.cols \,| \leq 2$$

$$|\, dstsize.height*2 \quad src.rows \,| \leq 2$$

此限制意味着目标图像差不多是源图像大小的一半。dstsize参数仅用于处理一些复杂的、需要严格控制来构建金字塔的情况。

cv::buildPyramid()

这是一个相对常见的情况，你有一个图像，并希望构建一系列的新图像，每个都从其前身缩减。函数cv::buildPyramid()在单个调用中为你创建一个图像堆栈。

```
void cv::buildPyramid(
  cv::InputArray           src,            // Input image
  cv::OutputArrayOfArrays dst,             // Output images from pyramid
  int                      maxlevel        // Number of pyramid levels
);
```

参数src是源图像。参数dst是一个有点不寻常的cv::OutputArrayOfArrays类型，但你可以将其视为只是一个STL向量<>或cv::OutputArray类型的对象。最常见的例子是<cv::Mat>向量。参数maxlevel表示要构建多少个金字塔级别。

参数maxlevel是大于或等于0的任何整数，并且指示要生成的金字塔图像的数量。当cv::buildPyramid()运行时，它将返回dst中的长度为maxlevel+1的向量。dst中的第一个条目将与src相同。第二个将是它的一半大小，就像调用cv::pyrDown()的结果一样。第三个将是第二个的一半大小，等等（图11-1中的左图）。

注4：这里的+1s是确保奇数大小的图像可以得到正确处理。如果图像是从偶数大小开始，它们将不起作用。

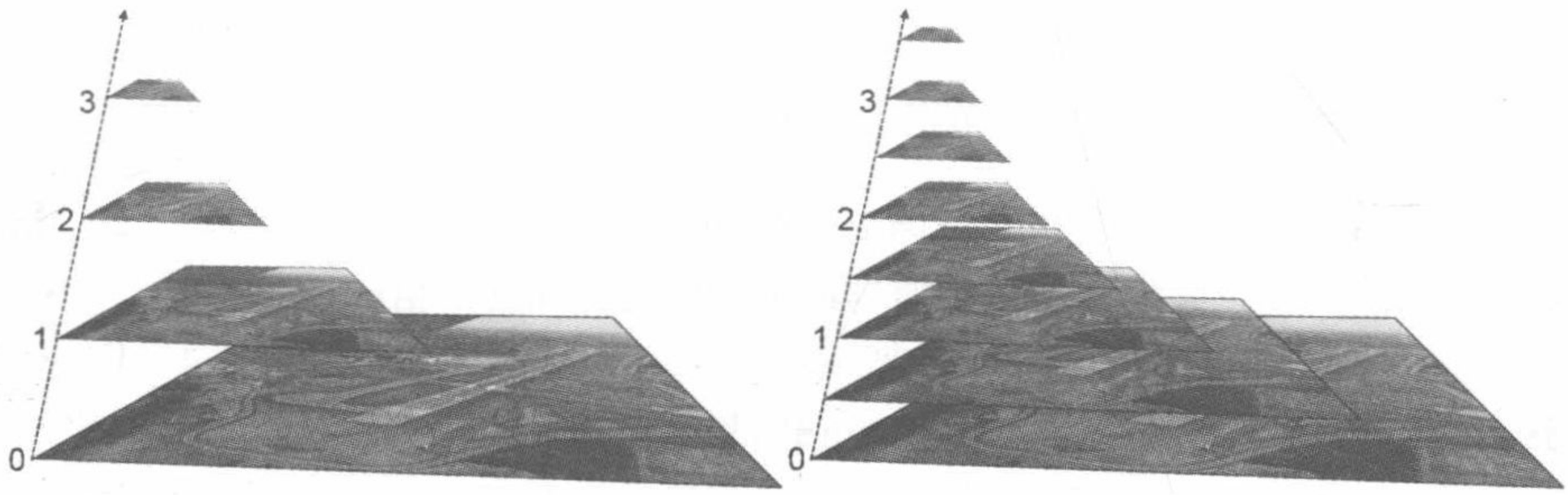


图11-1. 用maxlevel = 3生成的图像金字塔（左）；两个金字塔交错在一起创建一个金字塔（右）



在实践中，你经常会想要一个比两个因子更精细的对数缩放的金字塔。实现这一目标的一个方法是简单按你需要的次数调用cv::resize()，无论你想要使用什么比例因子，但这可能相当慢。一个替代方案（对于一些常见的比例因子）是对你想要的每个交错的图像集只调用cv::resize()一次，然后在这些调整大小的“基础”上逐次调用cv::buildPyramid()。然后，你可以将这些结果插入到一个大的、更细粒度的金字塔中。图11-1（右图）显示了生成两个金字塔的示例。原始图像首先通过因子重新缩放，然后在该图像上调用cv::buildPyramid()，以制作四个中间图像的第二个金字塔。一旦与原来的金字塔结合起来，就得到了一个更精致的金字塔，在整个金字塔中有一个比例因子。

cv::pyrUp()

类似地，我们可以通过以下类似（但不是反向！）操作将现有图像转换为每个方向两倍大的图像：

```
void cv::pyrUp(
  cv::InputArray  src,                        // Input image
  cv::OutputArray dst,                        // Result image
  const cv::Size& dstsize = cv::Size()        // Output image size
);
```

在这种情况下，首先将图像的大小增加到每个维度的原始值的两倍，新的（偶数）行填充0。此后，使用高斯滤波器[注5]执行卷积以近似“丢失”像素的值。

类似于cv::PyrDown()，如果dstsize设置为其默认值cv::Size()，结果图像将与src的大小（每个维度）的两倍大小相同。再次，我们可以提供一个dstsize，它将指示我们希望输出图像dstsize的大小，但是它必须再次遵循一些非常严格的限制。具体如下：

注5：该过滤器也被归一化为四个，而不是一个。这是合适的，因为插入的行在卷积之前的所有像素中都有0。通常，高斯核元素的和为1，但是在2×金字塔上采样的情况下，在2D情况下，所有内核元素乘以4，以恢复零行列插入之后的平均亮度。

$$|\,dstsize.width*2 - src.cols\,| \le (dstsize.width\%2)$$

$$|\,dstsize.height\,*2 - src.rows\,| \le (dstsize.height\,\%2)$$

这个限制意味着目标图像非常接近源图像大小的两倍。如前所述，dstsize参数仅用于处理有些复杂的情况，其中需要对如何构造金字塔进行非常严格的控制。

拉普拉斯金字塔

我们以前已经注意到，运算符cv::pyrUp()不是cv::pyrDown()的逆。这是很明显的，因为cv::pyrDown()是一个丢失信息的操作符。为了恢复原始（较高分辨率）的图像，我们需要访问下采样过程丢弃的信息。这个数据形成了拉普拉斯金字塔。拉普拉斯金字塔的第i层由以下关系定义：

$$L_i = G_i - UP(G_{i+1}) \otimes g_{5x5}$$

这里，运算符UP()通过将原始图像中的位置（x, y）中的每个像素映射到目标图像中的像素（$2x$ + 1, $2y$ + 1）来进行大小化；符号⊗表示卷积；而$g_{5\times5}$是5×5高斯核。当然，$UP(G_{i+1})\otimes g_{5\times5}$是由OpenCV提供的cv::pyrUp()运算符的定义。因此，我们可以使用OpenCV直接计算拉普拉斯算子：

$$L_i = G_i - pyrUp(G_{i+1})$$

高斯金字塔和拉普拉斯金字塔在图11-2中显示，这也显示了从子图像恢复原始图像的逆过程。请注意拉普拉斯算子是如何实际使用高斯差异的近似值的， 如之前的等式和图中示意图所示。

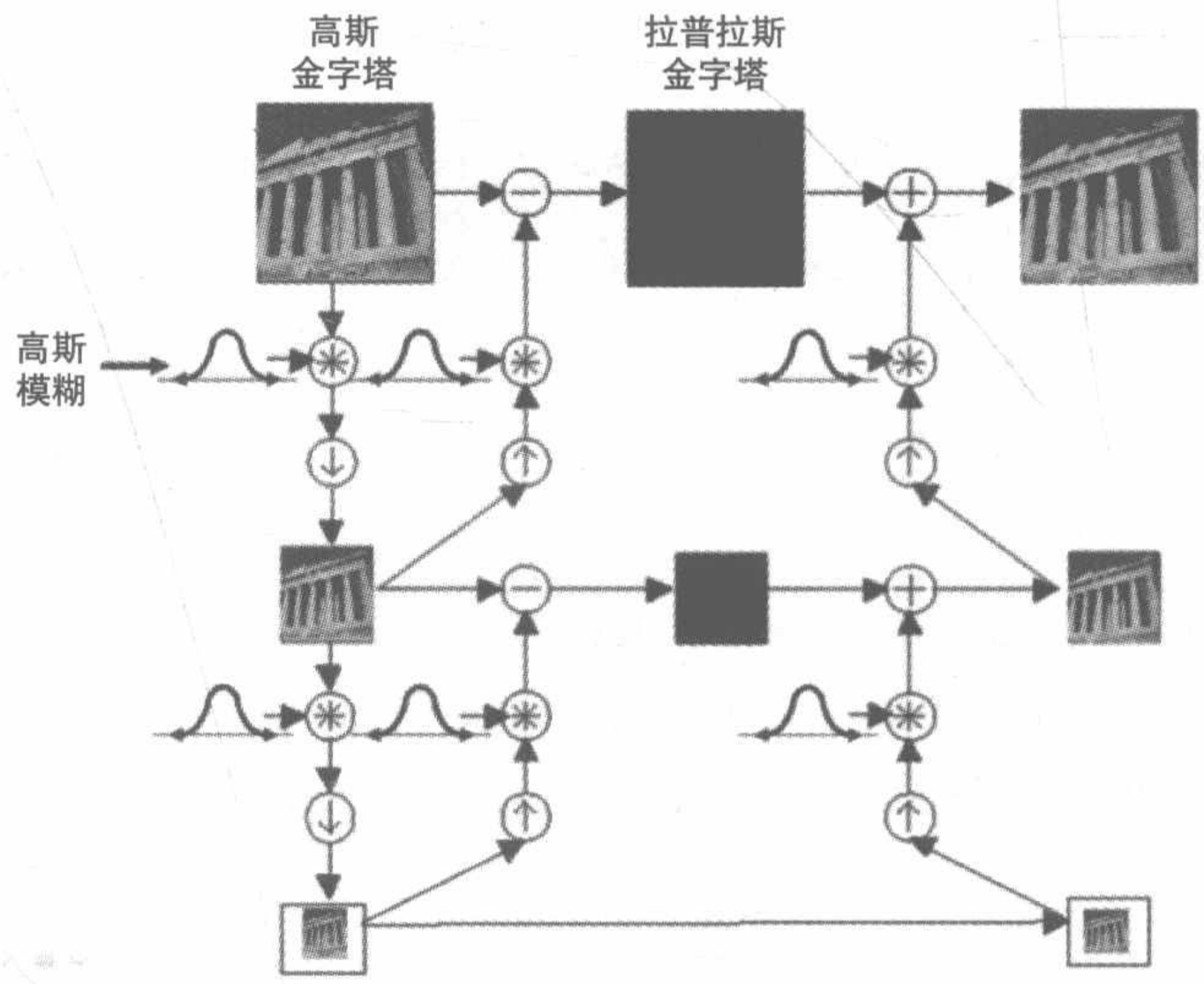


图11-2. 高斯金字塔和它的逆拉普拉斯金字塔

不均匀映射

在本节中，我们转向图像的几何操作，也就是说，这些变换起源于三维几何和投影几何的交叉点。[注6]这种操作包括均匀和不均匀的调整大小（后者称为“扭曲”）。执行这些操作有很多原因，例如，扭曲和旋转图像，使其可以叠加在现有场景的墙壁上，或人工放大用于目标识别的一组训练图像。[注7]可以拉伸、收缩、扭曲和/或旋转图像的功能称为“几何变换”（对于早期展示，请参见[Semple79]）。对于平面区域，有两种几何变换：使用2×3矩阵的变换，称为“仿射变换”；而基于3×3矩阵进行变换，称为“透视变换”或“同形”。你可以将后一种转换作为一种计算方法，用于计算一个特定观察者感觉三维平面的方法，而这些观察者可能不会直视平面。

仿射变换是可以以矩阵乘法后跟向量加法的形式表示的任何变换。在OpenCV中，代表这种转换的标准样式是2×3矩阵。定义如下：

$$A \equiv \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} B \equiv \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} T \equiv [A \ B] X \equiv \begin{bmatrix} x \\ y \end{bmatrix} X' \equiv \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

很容易看出，仿射变换$A \cdot X + B$的效果完全等同于将向量X扩展到向量X'，并且简单地将X'的转置左乘X。

仿射变换可以如下显示：平面中的任何平行四边形ABCD可以通过一些仿射变换映射到任何其他平行四边形$A'B'C'D'$。如果这些平行四边形的面积不是零，隐含的仿射变换就由两个平行四边形的（三个顶点）唯一定义。如果喜欢，你可以想象一个仿射变换，将自己的图像画成一个大的橡胶片，然后通过在角上的推或拉[注8]变形来制作不同样子的平行四边形。

当我们知道多个图像是同一个对象略有不同的视图时，可能需要计算与不同视图相关的实际变换。在这种情况下，通常使用仿射变换，而不是透视变换来模拟视图，因为它们

注6：我们将在这里详细介绍这些转换，并在第19章讨论如何在三维视觉技术的背景下使用它们。

注7：这个活动可能看起来有点诡异，毕竟，只用一种不变量的本地仿射失真识别方法不是更好吗？尽管如此，这种方法有着悠久的历史，在实践中非常有用。

注8：甚至可以采取这样的方式来反转平行四边形。

的参数更少，因此更容易解决。缺点是，真正的透视变换只能通过单应矩阵建模，[注9]所以仿射变换产生不能适应视图之间所有可能关系的表示。另一方面，对于视点的微小变化，所得到的变形是仿射的，因此在某些情况下，仿射变换可能够了。

仿射变换可以将矩形转换为平行四边形。它们可以挤压形状，但必须保持两边平行；它们可以旋转和/或缩放它。透视变换提供更多的灵活性；透视变换可以将矩形转换为任意四边形。图11-3显示了各种仿射变换和透视变换的示意图，图11-4展示了使用图像的例子。

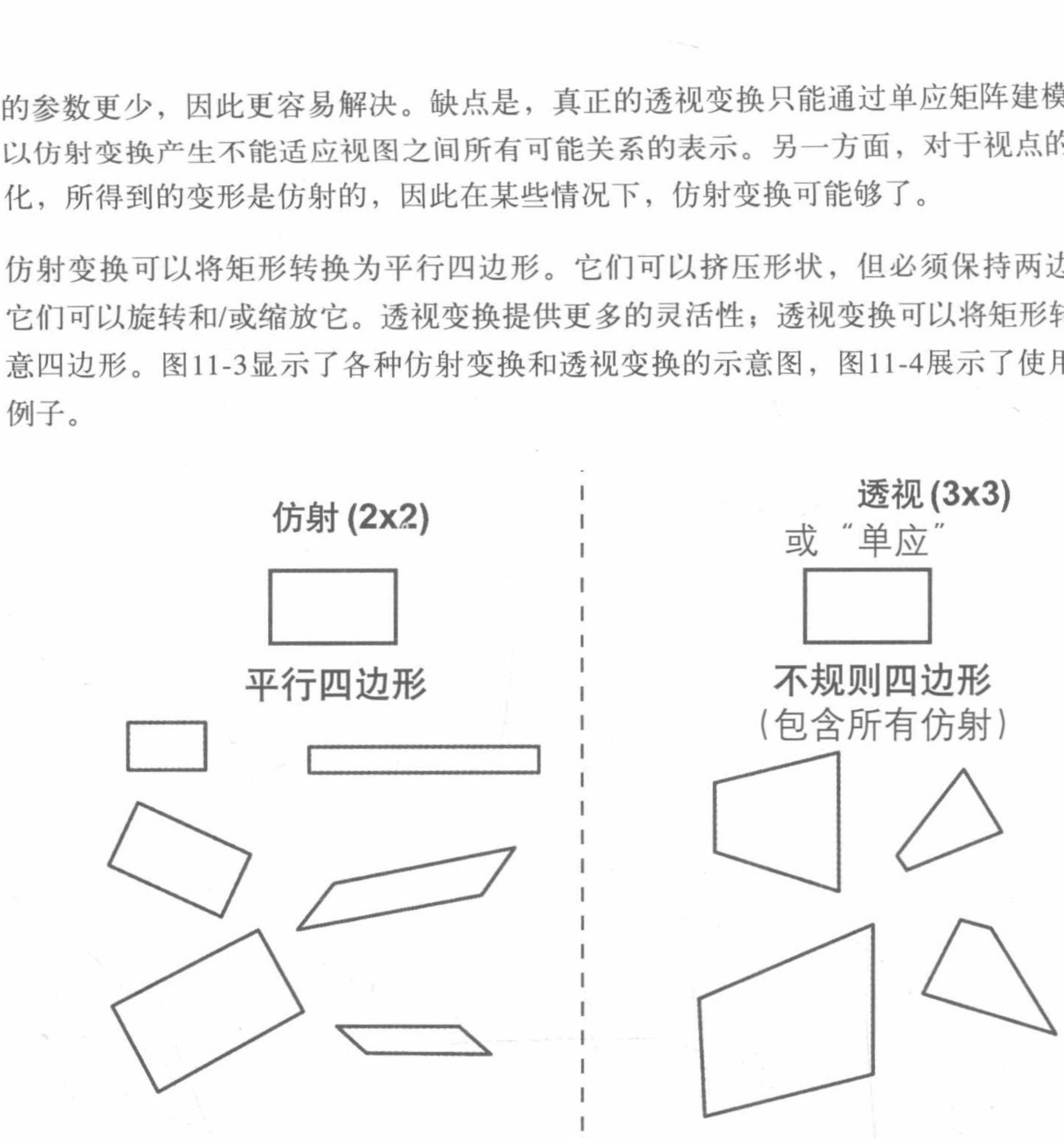


图11-3. 仿射变换和透视变换

仿射变换

仿射变换有两种情况。在第一种情况下，我们有一个想要转化的图像（或感兴趣的区域）；在第二种情况下，我们有一系列点，想要计算转换的结果。这些情况在概念上非常相似，但在实际执行方面却有很大的差异。因此，对于这些情况，OpenCV有两个不同的函数。

注9：同位素法是将一个表面上的点映射到另一个点上的数学术语。在这个意义上说，这个术语比这里更常用。在计算机视觉的上下文中，单应性几乎总是指在两个图像平面上对应于现实世界中平面物体上相同位置的点之间的映射。这样的映射可以由单个3×3正交矩阵表示（详情参见第19章）。

密集仿射变换函数cv::warpAffine()

在第一种情况下，明显的输入和输出格式是图像，隐含的要求是翘曲假设像素是底层图像的密集表示。这意味着图像扭曲必须处理内插，使得输出图像平滑而自然。OpenCV提供的用于密集变换的仿射变换函数是cv::warpAffine()：

```
void cv::warpAffine(
  cv::InputArray    src,                          // Input image
  cv::OutputArray   dst,                          // Result image
  cv::InputArray    M,                            // 2-by-3 transform mtx
  cv::Size          dsize,                        // Destination image size
  int               flags = cv::INTER_LINEAR,     // Interpolation, inverse
  int               borderMode = cv::BORDER_CONSTANT, // Pixel extrapolation
  const cv::Scalar& borderValue = cv::Scalar()    // For constant borders
);
```

这里src和dst分别是源和目标数组。输入M是我们前面介绍的量化所需变换的2×3矩阵。目标数组中的每个元素都是从源数组的元素中计算出来的：

$$dst(x, y) = src(M_{00}x + M_{01}y + M_{02}, M_{10}x + M_{11}y + M_{12})$$

然而，一般来说，该方程右边所示的位置可能不是整数像素。在这种情况下，需要使用插值来找到$dst(x, y)$的适当值。下一个参数，flags，选择插值方法。可用的插值方法是表11-1中的插值方法，与cv::resize()相同，另外还有一个附加选项cv::WARP_INVERSE_MAP（可以使用通常的布尔OR）。这个选项是一种方便，允许从dst到src而不是从src到dst的反向变换。最后两个参数用于边界外推，并且与图像卷积中的类似参数具有相同的含义（参见第10章）。

计算仿射映射矩阵的cv::getAffineTransform()

OpenCV提供了两个功能来帮助你生成映射矩阵M。第一个是当你已经有两个图像的时候你知道要用仿射变换来表示或者你想用那个方法来近似：

```
cv::Mat cv::getAffineTransform(          // Return 2-by-3 matrix
  const cv::Point2f* src,                // Coordinates *three* of vertices
  const cv::Point2f* dst                 // Target coords, three vertices
);
```

这里的src和dst是包含三个二维（x, y）点的数组。返回值是从这些点计算的仿射变换的数组。

本质上，src和dst中的点数组在cv::getAffineTransform()中定义了两个平行四边形。src中的点通过应用cv::warpAf fine()将使用结果矩阵M映射到dst中的对应点；所有其他点都将随之拖曳。一旦这三个独立的角被映射，所有其他点的映射可以完全确定。

示例11-1显示了一些使用这些功能的代码。在这个例子中，我们通过首先构造两个三分量的点阵列（我们的代表性平行四边形的角），然后使用cv::getAffineTransform()将其转换成实际的变换矩阵，得到cv::warpAffine()矩阵参数。然后我们做仿射变换，然后进行图像的旋转。对于我们在源图像中的代表点数组，称为srcTri[]，我们取三个点：(0,0)，(0, height-1)和(width-1,0)。然后，我们指定这些点映射到相应数组dstTri []中的位置。

示例11-1：仿射变换

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {
  if(argc != 2) {
    cout << "Warp affine\nUsage: " <<argv[0] <<" <imagename>\n" << endl;
    return -1;
  }

  cv::Mat src = cv::imread(argv[1],1);
  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }

  cv::Point3f srcTri[] = {
    cv::Point2f(0,0),           // src Top left
    cv::Point2f(src.cols-1, 0), // src Top right
    cv::Point2f(0, src.rows-1)  // src Bottom left
};

cv::Point2f dstTri[] = {
  cv::Point2f(src.cols*0.f, src.rows*0.33f),   // dst Top left
  cv::Point2f(src.cols*0.85f, src.rows*0.25f), // dst Top right
  cv::Point2f(src.cols*0.15f, src.rows*0.7f)   // dst Bottom left
};

// COMPUTE AFFINE MATRIX
//
cv::Mat warp_mat = cv::getAffineTransform(srcTri, dstTri);
cv::Mat dst, dst2;
cv::warpAffine(
  src,
  dst,
  warp_mat,
  src.size(),
  cv::INTER_LINEAR,
  cv::BORDER_CONSTANT,
  cv::Scalar()
);
for( int i = 0; i < 3; ++i )
  cv::circle(dst, dstTri[i], 5, cv::Scalar(255, 0, 255), -1, cv::AA);

cv::imshow("Affine Transform Test", dst);
cv::waitKey();
```

```
for(int frame=0;;++frame) {

  // COMPUTE ROTATION MATRIX
  cv::Point2f center(src.cols*0.5f, src.rows*0.5f);

double angle = frame*3 % 360, scale = (cos((angle - 60)* cv::PI/180) + 1.05)*0.8;

cv::Mat rot_mat = cv::getRotationMatrix2D(center, angle, scale);

cv::warpAffine(
  src,
  dst,
  rot_mat,
  src.size(),
  cv::INTER_LINEAR,
  cv::BORDER_CONSTANT,
  cv::Scalar()
);
  cv::imshow("Rotated Image", dst);
  if(cv::waitKey(30) >= 0 )
    break;

  }

  return 0;
}
```

计算映射矩阵M的第二种方法是使用cv::getRotationMatrix2D()，它计算围绕某个任意点的旋转的映射矩阵，并结合可选的重新缩放。这只是一种可能的仿射变换，所以它比cv::getAffineTransform()更不通用，但它代表一个重要的子集，它具有替代的（更直观的）更容易理解的表示：

```
cv::Mat cv::getRotationMatrix2D(     // Return 2-by-3 matrix
  cv::Point2f center                 // Center of rotation
  double angle,                      // Angle of rotation
  double scale                       // Rescale after rotation
);
```

第一个参数center，是旋转的中心点。接下来的两个参数给出旋转的大小和总体重新缩放。该函数返回映射矩阵M，（一如既往）是浮点数的2×3矩阵。

如果定义α= scale * cos（angle）和β= scale * sin（angle），则该函数计算矩阵M为：

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot center_x - \beta \cdot center_y \\ -\beta & \alpha & \beta \cdot center_x - (1-\alpha)\cdot center_y \end{bmatrix}$$

可以组合这些设置map_matrix的方法来获取，例如旋转、缩放和扭曲图像。

cv::transform()用于稀疏仿射变换

我们已经解释cv::warpAffine()是处理密集映射的正确方法。对于稀疏映射（即单个点的列表的映射），最好使用cv::transform()。可以回想一下第5章，变换方法有以下原型：

```
void cv::transform(
  cv::InputArray src,  // Input N-by-1 array (Ds channels)
  cv::OutputArray dst, // Output N-by-1 array (Dd channels)
  cv::InputArray mtx   // Transform matrix (Ds-by-Dd)
);
```

通常，src是具有D_s通道的$N \times 1$数组，其中的N是要变换的点数，D_s是这些源点的维数。输出数组dst将具有相同的大小，但可能具有不同数量的通道D_d。变换矩阵mtx是$D_s \times D_d$矩阵，然后将其应用于src的每个元素，之后将结果放入dst。



请注意，cv::transform()作用于数组中每个点的通道索引。对于当前的问题，我们假设数组本质上是这些多通道对象的大向量（$N \times 1$或$1 \times N$）。要记住一个重要事情，变换矩阵相对的是通道索引的索引，而不是大数组的“向量”索引。

在这个通道空间的简单旋转变换的情况下，我们的变换矩阵mtx只能是2×2的矩阵，可以直接应用于src的两个通道的索引。事实上，在一些简单的情况下，旋转、拉伸和扭曲也如此。然而，通常做一般的仿射变换（包括关于任意中心的变换和旋转等），有必要将src中的通道数量扩展到三个，使更常用的2×3仿射的动作 定义变换矩阵。在这种情况下，所有第三通道项必须设置为1（也就是说，点必须以齐次坐标提供）。当然，输出数组仍然是一个双通道阵列。

cv::invertAffineTransform()用于逆仿射变换

给定表示为2×3矩阵的仿射变换，通常希望能够计算逆变换，它可以用于将所有转换点“放回”它们原来的地方。这是用cv::invertAffineTransform()完成的：

```
void cv::invertAffineTransform(
  cv::InputArray M,                    // Input 2-by-3 matrix
  cv::OutputArray iM                   // Output also a 2-by-3 matrix
);
```

这个函数需要一个2×3的数组M，并返回另一个2×3的数组M的逆iM。注意，cv::invertAffineTransform()实际上不对任何图像起作用，它只提供逆变换。一旦你有iM，就可以像使用M一样使用它，使用cv::warpAffine()或cv::transform()。

透视变换

为了获得透视变换（也称为“单应性”）提供的更大的灵活性，我们需要一个新的功能，使我们能够表达更广泛的变换。首先我们指出，即使透视投影完全由单个矩阵指定，投影实际上也不是线性变换。这是因为变换需要通过最终维度（通常为Z；参见第19章）进行划分，从而在过程中失去一个维度。

与仿射变换一样，图像操作（密集变换）通过不同的函数来处理，而不是点集（稀疏变换）上的变换。

cv::warpPerspective()用于密集透视变换

密集透视变换使用类似于为密集仿射变换提供的OpenCV函数。具体来说，cv::warpPerspective()的参数与cv::warpAffine()相同，只不过小但至关重要的映射矩阵现在必须为3×3。

```
void cv::warpPerspective(
  cv::InputArray    src,                                // Input image
  cv::OutputArray   dst,                                // Result image
  cv::InputArray    M,                                  // 3-by-3 transform mtx
  cv::Size          dsize,                              // Destination image size
  int               flags = cv::INTER_LINEAR,           // Interpolation, inverse
  int               borderMode = cv::BORDER_CONSTANT, // Extrapolation method
  const cv::Scalar& borderValue = cv::Scalar()          // For constant borders
);
```

目标数组中的每个元素都是从源数组的元素中计算出来的：

$$dst(x, y) = src\left(\frac{M_{00}x + M_{01}y + M_{02}}{M_{20}x + M_{21}y + M_{22}}, \frac{M_{10}x + M_{11}y + M_{12}}{M_{20}x + M_{21}y + M_{22}}\right)$$

与仿射变换一样，由该方程右侧指示的位置不会（通常）为整数位置。再次使用flags参数来选择所需的插值方法，并且有与cv::warpAffine()相应参数相同的可能值。

cv::getPerspectiveTransform()用于计算透视映射矩阵

与仿射变换一样，为了填补上述代码中的map_matrix，我们有一个方便的函数，可以从点对应列表中计算变换矩阵：

```
cv::Mat cv::getPerspectiveTransform(        // Return 3-by-3 matrix
  const cv::Point2f* src,                   // Coordinates of *four* vertices
  const cv::Point2f* dst                    // Target coords, four vertices
);
```

src参数和dst参数现在是四个（不是三个）点的数组，所以我们可以独立控制src中的（典型）矩形的一个角如何映射到（通常）dst中的一些菱形。我们的变换完全由四个

源点的指定目的地定义。如前所述，对于透视变换，返回值将为3×3数组。有关示例代码，请参见示例11-2。除了3×3矩阵和从三到四个控制点的转变，透视变换与我们前面介绍的仿射变换完全相似。

示例11-2：透视变换代码

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Perspective Warp\nUsage: " <<argv[0] <<" <imagename>\n" << endl;
    return -1;
  }

  Mat src = cv::imread(argv[1],1);
  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }

  cv::Point2f srcQuad[] = {
    cv::Point2f(0, 0),                    // src Top left
    cv::Point2f(src.cols-1, 0),           // src Top right
    cv::Point2f(src.cols-1, src.rows-1), // src Bottom right
    cv::Point2f(0, src.rows-1)            // src Bottom left
  };

cv::Point2f dstQuad[] = {
  cv::Point2f(src.cols*0.05f, src.rows*0.33f),
  cv::Point2f(src.cols*0.9f, src.rows*0.25f),
  cv::Point2f(src.cols*0.8f, src.rows*0.9f),
  cv::Point2f(src.cols*0.2f, src.rows*0.7f)
};

// COMPUTE PERSPECTIVE MATRIX
//
cv::Mat warp_mat = cv::getPerspectiveTransform(srcQuad, dstQuad);
cv::Mat dst;
cv::warpPerspective(src, dst, warp_mat, src.size(), cv::INTER_LINEAR,
                    cv::BORDER_CONSTANT, cv::Scalar());

for( int i = 0; i < 4; i++ )
  cv::circle(dst, dstQuad[i], 5, cv::Scalar(255, 0, 255), -1, cv::AA);

  cv::imshow("Perspective Transform Test", dst);
  cv::waitKey();
  return 0;
}
```

cv::perspectiveTransform()用于稀疏透视变换

cv::perspectiveTransform()是一个特殊的函数，可以对点列表执行透视变换。因为cv::transform()被限定于线性运算，所以不能正确地处理透视变换。这是因为这种

变换需要由同质表示的第三个坐标（$x = f * x / Z$, $y = f * y / Z$）来划分。特殊函数cv::perspectiveTransform()为我们考虑了这个：

```
void cv::perspectiveTransform(
  cv::InputArray src,              // Input N-by-1 array (2 or 3 channels)
  cv::OutputArray dst,             // Output N-by-1 array (2 or 3 channels)
  cv::InputArray mtx               // Transform matrix (3-by-3 or 4-by-4)
);
```

像往常一样，src和dst参数分别是要变换的源点数组和由变换产生的目标点数组。这些数组应该是两通道或三通道数组。矩阵mat可以是3×3或4×4矩阵。如果是3×3，则投影从二维到二维；如果矩阵是4×4，则投影从三维到三维。

在当前的场景中，我们将图像中的一组点变换为图像中的另一组点，这听起来像从二维到二维的映射。然而，这不完全正确，因为透视变换实际上是将嵌入在三维空间中的二维平面上的点映射回到（不同的）二维子空间。可以认为这像是一个相机做的。（在后面的章节中讨论相机时，我们将回到这个主题进一步讨论。）相机采取三维点并将其映射到相机成像器的两个维度。这本质上意味着当源点被认为是“齐次坐标”时的意思。我们通过引入Z维并将所有Z值设置为1，向这些点添加一个维度。然后投影变换从这个空间回到我们输出的二维空间。这是相当繁琐的，你需要一个3×3的矩阵来解释如何将一个图像中的点映射到另一个图像中的点。

示例11-1和示例11-2中代码的输出如图11-4所示，用于仿射映射和透视变换。在这些例子中，我们变换实际的图像，可以将它们与图11-3中的简单图进行比较。

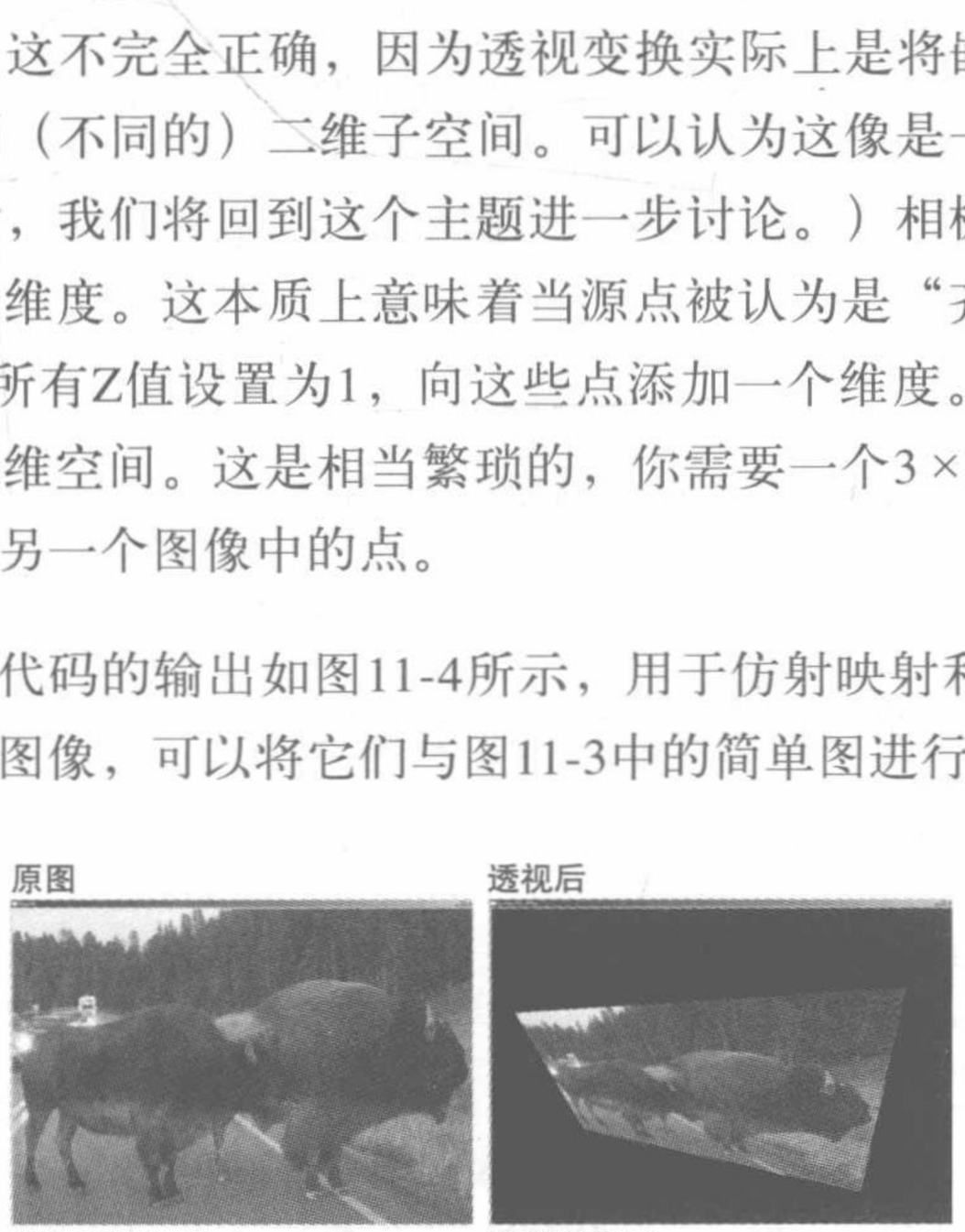


图11-4. 图像的透视变换和仿射映射

通用变换

我们目前所看到的仿射变换和透射变换是一些更为一般的处理过程中特殊的例子。本质上，这两种变换有着相似的特性：它们把源图像的像素从一个地方映射到目标图像的另一个地方。事实上，其他一些操作也有着相同的结构。在这个小节，我们学习一些类似的变换，而后学习如何让OpenCV实现自己的映射变换。

极坐标映射

在第5章讨论矩阵操作时，我们简略学习了两个函数，分别是cv::cartToPolar()和cv::polarToCart()，它们实现了点阵的直角坐标和极坐标之间的相互转换。



然而，在极坐标映射函数、透视变换函数以及仿射变换函数之间，有一些细微的差异。极坐标映射函数在表示二维向量的时候，只能用一系列一维矩阵，不能用多维矩阵。这个不同源于两个函数的传统用法，而非两者功能上的本质区别。

函数cv::cartToPolar()和cv::polarToCart()在更为复杂的例子中采用，像cv::logPolar()（后文会介绍），但也有它自己的用武之地。

cv::cartToPolar()用于将直角坐标转换为极坐标

使用函数cv::cartToPolar()将直角坐标映射到极坐标：

```
void cv::cartToPolar(
  cv::InputArray  x,                          // Input single channel x-array
  cv::InputArray  y,                          // Input single channel y-array
  cv::OutputArray magnitude,                  // Output single channel mag-array
  cv::OutputArray angle,                      // Output single channel angle-array
  bool            angleInDegrees = false      // Set true for degrees, else radians
);
```

前两个参数x,y是一维矩阵。从概念上讲，x,y表示的并不仅仅是一系列的点，它还代表着向量场，[注10]其中向量场中的元素x用矩阵x中的值表示，向量场中的元素y用矩阵y的值表示。类似地，该函数的返回结果保存在数组的大小和角度上，其中，每个点的的大小表示该点处x和y向量的长度，并且每个点的角度表示该向量的方向。angle中的每个点表示矩阵的方向。默认以弧度[0, 2π)来保存角度值。如果参数angleInDegrees设置为

注10：如果你对矢量场的概念还不够了解，在处理我们的问题的时候，可以认为它是每个点在图像的二维向量，这就足够了。

true，角度值以角度[0, 360)来保存。另外要注意，角度值使用atan2(y,x)计算（近似地），所以角度0代表着x方向的一个向量。

这里给一个假设要用到这个函数的例子。假设你已经求得一幅图像x和y方向的导数，不管是使用cv::Sobel()还是使用卷积函数cv::DFT()或cv::filter2D()。如果你储存x方向的导数在dx_img中，储存y方法的导数在dy_img中，你应当创建一个边缘—角度识别直方图，也就是说，你可以保存所有的角度值，只要边缘像素的大小和强度高于某个所需的阈值即可。为了计算此阈值，我们首先为方向导数创建两个目标图像（并且分别称为img_mag和img_angle），然后用函数cv::CartToPolar(dx_img, dy_img, img_mag, img_angle, 1)。这样，我们就可以从 图像img_angle中找到图像img_mag相应像素点高于我们所需的阈值的点，绘出一个直方图。



在第22章中，我们将讨论与图像识别和图像特征提取相关的知识。这个计算过程实际上是重要的图像特征HOG（梯度统计直方图）作用于目标识别的基础。

cv::polarToCart()用于将极坐标转换为直角坐标

函数cv::cartToPolar()用来进行极坐标到直角坐标的反向映射。

```
void cv::polarToCart(
  cv::InputArray  magnitude,                  // Output single channel mag-array
  cv::InputArray  angle,                      // Output single channel angle-array
  cv::OutputArray x,                          // Input single channel x-array
  cv::OutputArray y,                          // Input single channel y-array
  bool            angleInDegrees = false      // Set true for degrees, else radians
);
```

从极坐标反向映射为直角坐标也很常用。从本质上讲，它的参数与函数cv::cartToPolar相同，区别在于，向量的大小和角度值作为输入参数，而元素x和y作为结果返回。

LogPolar

对于二维图像，对数-极坐标变换是从直角坐标变换为对数极坐标，即 $(x, y) \leftrightarrow re^{i\theta}$，其中，$r = \sqrt{x^2 + y^2}$，$\theta = atan2(y, x)$。接下来，为将极坐标表示为a($\rho$, θ)形式，类似于点的表达形式（x_c, y_c），然后取对数，即ρ=log(), $\theta = atan2(y - y_c, x - x_c)$。当我们需要把一些有趣的东西填充到目标图像的空白区，通常的做法是将比例因子m应用于ρ。图11-5展示了左侧正方形图像及其在对数极坐标下的图像。

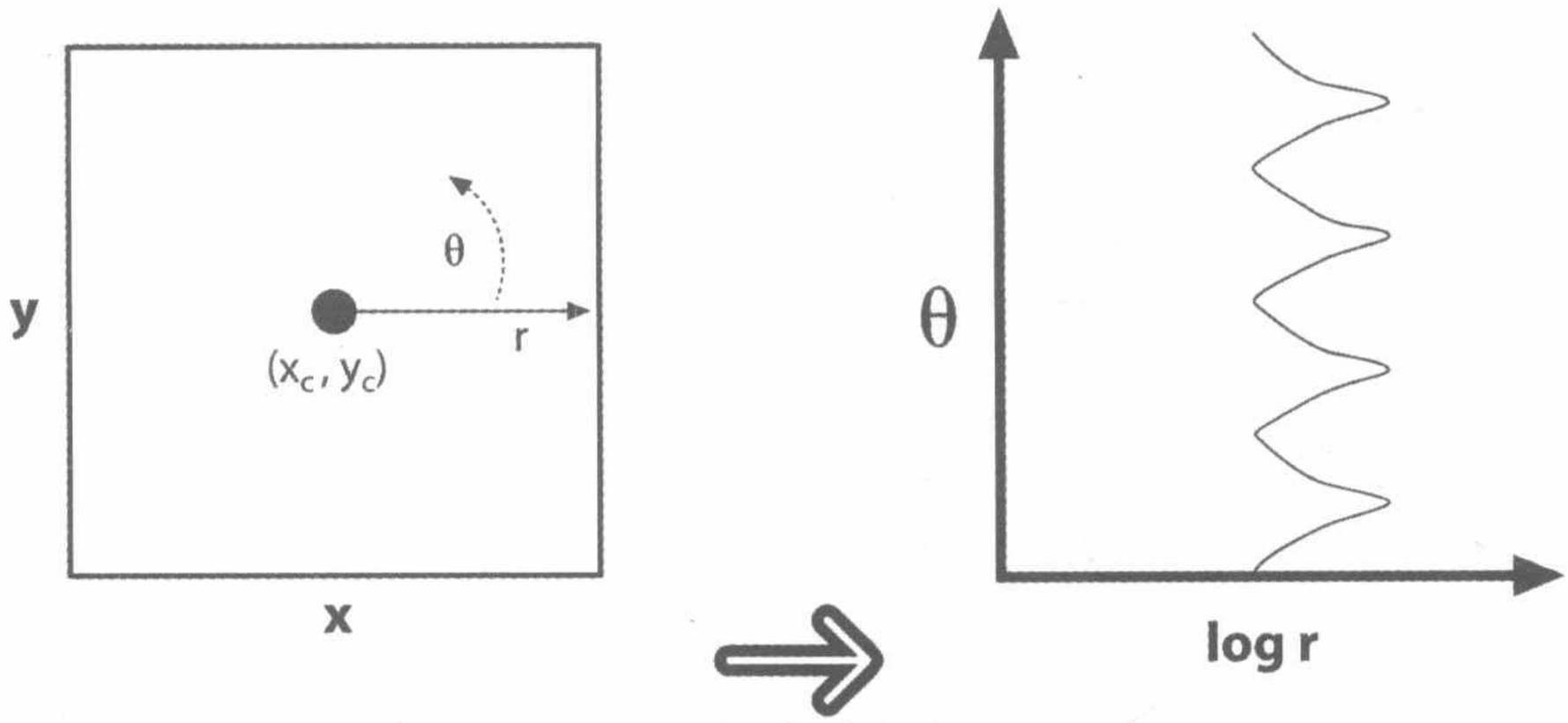


图11-5.用对数-极坐标转换函数将坐标（x,y）映射为（log(r), θ），在对数-极坐标中展示一个方形图像

你可能奇怪为什么人们都乐于此道。对数-极坐标变换的灵感来源于人类视觉系统。眼睛里面（中央凹）有一个小而密集的光感受器，并且感觉器官的密度从中心向四周快速降低（呈指数级）。试着盯着墙上的一个点，并且将你的手指固定在手臂长度的视线内。然后，手指慢慢远离脸部，但眼睛一直盯着墙上的那个点，注意手指在视网膜上的成像慢慢远离眼睛中心（中央凹）的细节。这种结构同时具有某些非常好的数学特性（超出本书范围），即保存直线交叉点的角度。

对我们来说，有一点更重要：对数-极坐标转换可以用来创建对象视图的固定的二维表示法，通过转换图像众多中心点为对数-极坐标平面的固定一个点；观察图11-6，左图是三个方形图。问题是，它们看起来不同。外面虚线的比其他两个大，另一个是旋转后的。右图是对数—极坐标转换后的图像。注意观察大小不同的方形图在对数-极坐标系中log(*r*)的值的不同，注意方向不同的方形图在对数-极坐标系中θ值的不同。如果我们把每个正方形的变换中心放在对数-极坐标系中，然后将那个点重新指向某个固定的点，这样，所有的正方形将在对数-极坐标系中有相同的表示方式。这就可以推出二维图像旋转缩放的不变性。[注11]

注11：在第22章中，我们将学习图像识别。目前来讲，我们只需要知道将整个目标图像进行对数-极坐标变换并不知道明智之举，因为这种变换对图像的中心点位置要求严格。更可能用于对象识别的是检测对象周围的关键点（像边角点、大量的其他点）的集合，截断这些视图的范围，然后把这些关键点的中心作为对数-极坐标系的中心。然后可以使用这些本地对数-极坐标变换创建本地的固定比例，固定方向的特性图像，并且可以与视觉对象相关联。

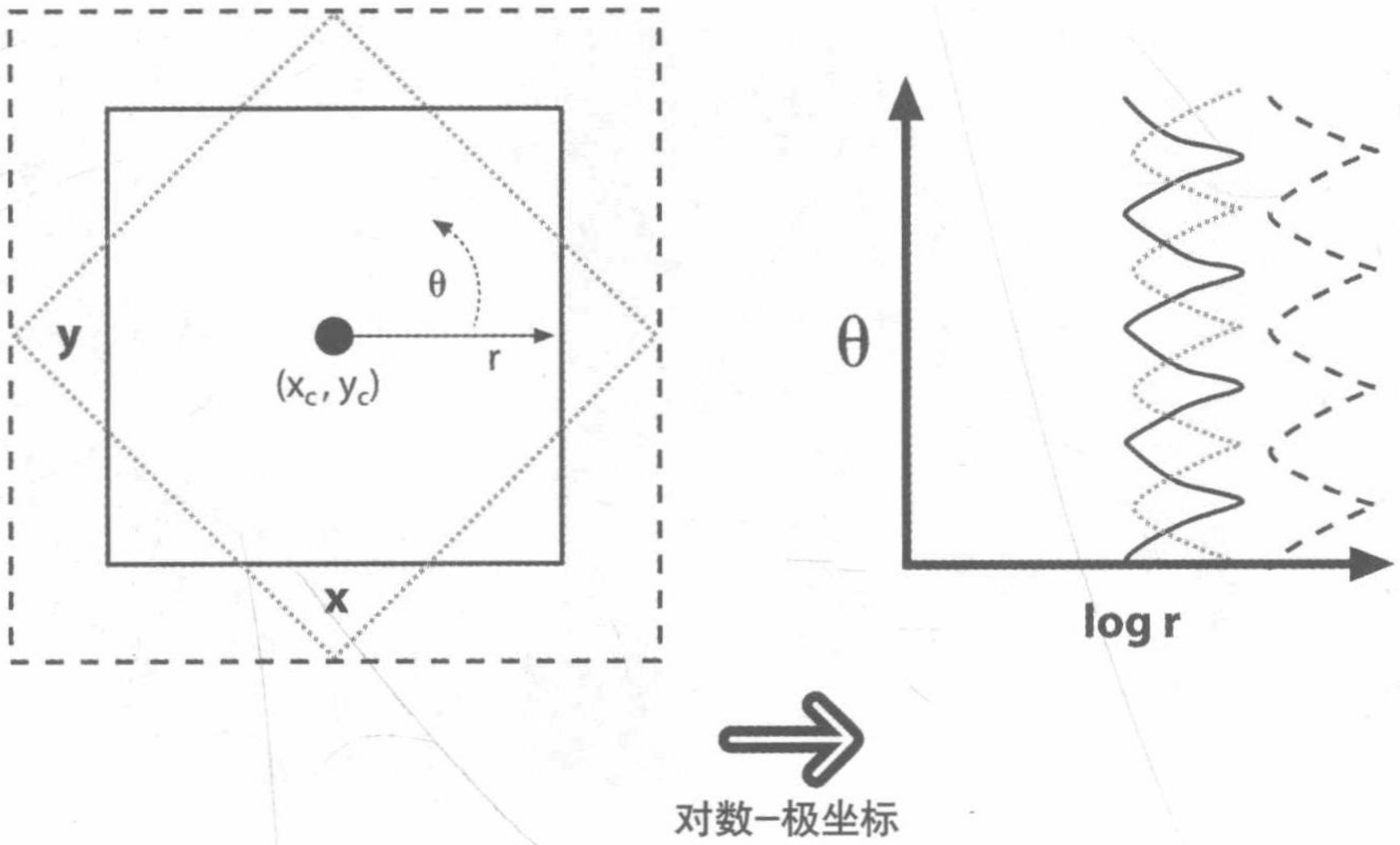


图11-6. 旋转方形和放大方形对数-极坐标变换：大小影响着变换的log(*r*)的值，旋转（方向）影响着变换后的θ的值

cv::logPolar()函数

OpenCV提供函数cv::logPolar()进行对数-极坐标转换。

```
void cv::logPolar(
  cv::InputArray  src,                          // Input image
  cv::OutputArray dst,                          // Output image
  cv::Point2f     center,                       // Center of transform
  double          m,                            // Scale factor
  int             flags = cv::INTER_LINEAR      // interpolation and fill modes
                        | cv::WARP_FILL_OUTLIERS

);
```

src和dst分别表示输入的原图和输出的目标图。参数center是对数-极坐标变换的中心点(x_c, y_c)；m是比例因子，这个比例因子的设置原则应该便于最感兴趣的特征能够存在于图像的大部分区域。参数flags支持不同的插入方式。OpenCV（参见表11-1）提供了一组标准的插值方法集。参数插入方式可以选择cv::WARP_FILL_OUTLIERS（填充未定的点）和cv::INTER_LINEAR（计算从对数-极坐标系转换为直角坐标系的反向映射）中的一种，或者二者都选。

示例11-3是对数-极坐标编码的简单实例，示范了进行对数-极坐标变换的前后图像。图11-7是转换后的高清图像。



图11-7. 左图是以麋鹿的白点为中心进行对数－极坐标变换，右图是变换结果

示例11-3：对数－极坐标变换实例

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 3) {
    cout << "LogPolar\nUsage: " <<argv[0] <<" <imagename> <M value>\n"
      <<"<M value>~30 is usually good enough\n";
    return -1;
  }

  cv::Mat src = cv::imread(argv[1],1);

  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }
  double M = atof(argv[2]);
  cv::Mat dst(src.size(), src.type()), src2(src.size(), src.type());

  cv::logPolar(
    src,
    dst,
    cv::Point2f(src.cols*0.5f, src.rows*0.5f),
    M,
    cv::INTER_LINEAR | cv::WARP_FILL_OUTLIERS
  );
  cv::logPolar(
    dst,
    src2,
    cv::Point2f(src.cols*0.5f, src.rows*0.5f),
    M,
    cv::INTER_LINEAR | cv::WARP_INVERSE_MAP
  );
```

```
  cv::imshow( "log-polar", dst );
  cv::imshow( "inverse log-polar", src2 );

  cv::waitKey();

  return 0;
}
```

任意映射

我们有时想要实现以编程方式插值。这意味着我们要使用一些已知的能够实现映射的算法。另一方面，我们又想自己实现这种映射。在研究这些能为我们计算（并应用）这些映射方法前，我们先看看其他方法依赖的能够实现这种映射的函数。

函数cv::remap()通常用来纠正校准的立体图像。在第18章和第19章中，我们将会学习能够将计算出的相机失真和排列转换为参数mapx和mapy的函数。

我们需要的函数是cv::remap()。

cv::remap()用于常规图像重绘

```
void cv::remap(
  cv::InputArray    src,                                  // Input image
  cv::OutputArray   dst,                                  // Output image
  cv::InputArray    map1,                                 // target x for src pix
  cv::InputArray    map2,                                 // target y for src pix
  int               interpolation = cv::INTER_LINEAR,     // Interpolation, inverse
  int               borderMode    = cv::BORDER_CONSTANT, // Extrapolation method
  const cv::Scalar& borderValue   = cv::Scalar()          // For constant borders
);
```

函数cv::remap()的前两个参数分别表示源图像和目标图像。接着的两个参数map1和map2分别表示源图像上需重新定位的任意一点的x和y的位置。这样你就可以自己指定一般映射。这些图应与源图像和目标图像保持相同尺寸，并且必须使用以下数据类型之一：cv::S16C2，cv::F32C1，cv::F32C2（允许使用非整型数据映射），函数cv::remap()将自动进行插值计算。

下一个参数interpolation告诉函数cv::remap()以哪种方式进行插值计算。表11-1列出的选项可以应用于cv::INTER_AREA，但函数cv::remap()不可以使用。

图像修复

图像经常因噪声造成破损。镜头上可能有灰尘或水渍，旧图像可能有划痕，或者图像的一部分被损坏。图像修复是消除这种损坏的一种方式，它通过摄取被损坏区域边缘的

色彩和纹理，然后传播混合至损坏区域的内部。图11-8展示了如果从写有字的图上消除字。



图11-8. 左图是被文字覆盖而损坏的图像，右图是经过修复后的图像

图像修复

修复作品提供的损坏区域不是很“密集”，并且损坏区域周边残留很多源图中的纹理和色彩。图11-9展示了损坏区域过大在修复后的结果。

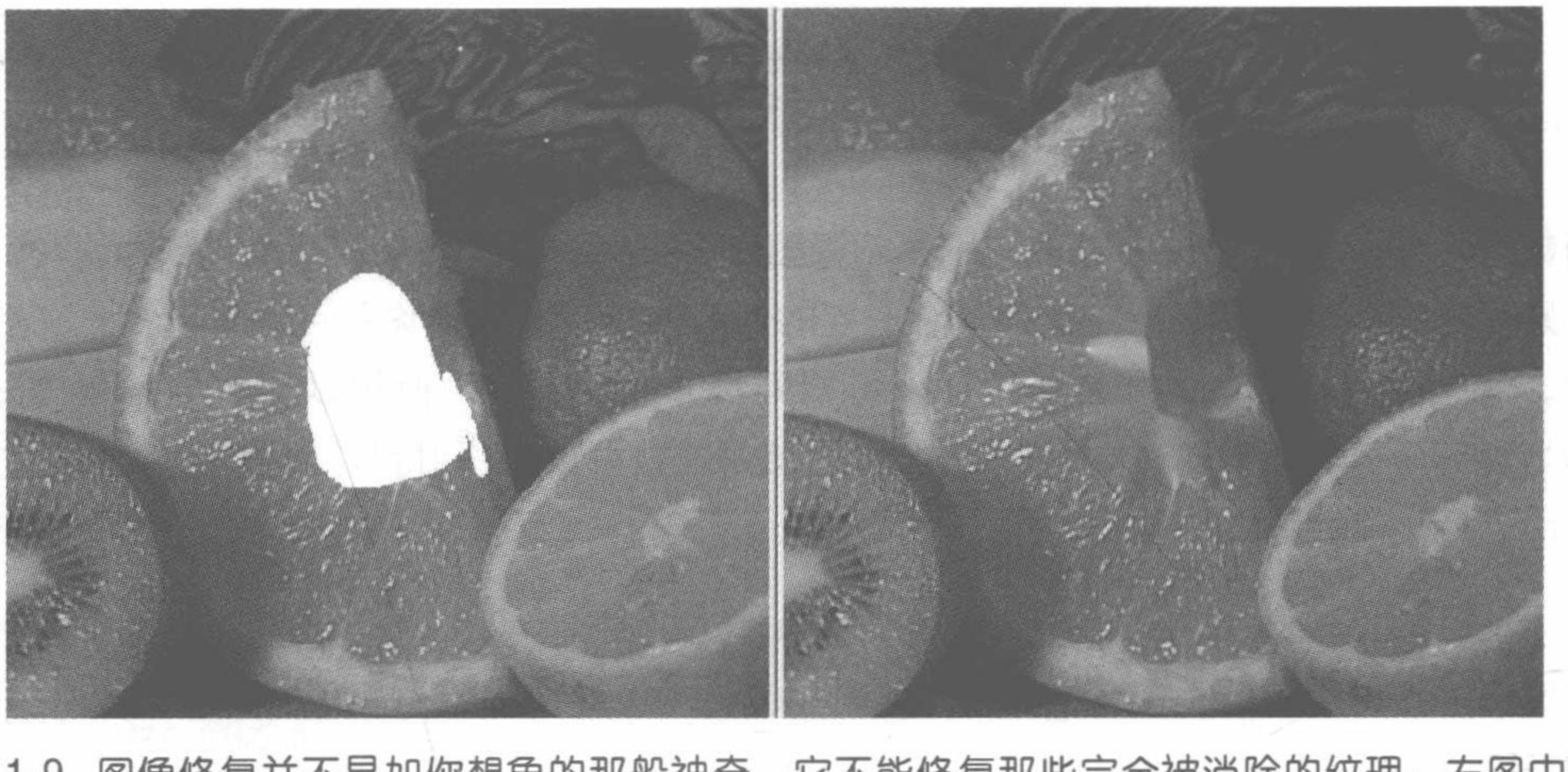


图11-9. 图像修复并不是如你想象的那般神奇，它不能修复那些完全被消除的纹理：左图中橙子中央被完全污染，右图中修复结果只是填充了近似橙子的颜色

函数cv::inpaint()的源码如下：

```
    void cv::inpaint(
      cv::InputArray  src,                    // Input image: 8-bit, 1 or 3 channels
```

```
    cv::InputArray  inpaintMask,        // 8-bit, 1 channel. Inpaint nonzeros
    cv::OutputArray dst,                // Result image
    double          inpaintRadius,      // Range to consider around pixel
    int             flags               // Select NS or TELEA
);
```

这里的src是一个8位，一维的灰度图像或者一个三维的彩色的待修复图像，inpaintMask是一个8位，与src大小相同的一维图像，且损坏区域被非零像素标记（例如，图11-8左图中写有字的图像），inpaintMask的其他像素均被设置为0。dst为输出图像，它与src有相同大小和数量的维度。inpaintRadius是每个已渲染像素周围的区域，这一区域将被分解成该像素的结果输出颜色。

如图11-9所示，在一个足够厚的凹凸区域内部的内部像素可以整个从其他更接近边界的未上色的像素中得到它们的颜色。使用小半径，如3，因为过大半径而造成明显模糊的情况时有发生。最后，参数flags支持两种不同的修复方法：cv::INPAINT_NS（Navier-Stokes方法）和cv::INPAINT_TELEA（A. Telea's方法）。

去噪

另一个重要的问题是图像中的噪声。在许多应用中，噪声的主要来源于低光条件的影响。在低光下，数字成像器的增益必须增加，结果呢，噪声也被放大。这种噪声的特征是随机孤立的像素，看起来太亮或太暗，但在彩色图像中也可能发生变色。

OpenCV中实现的去噪算法称为“快速非局部均值去噪”（FNLMD），基于Antoni Buades，Bartomeu Coll，Jean-Michel Morel的工作成果。虽然简单的去噪算法基本上依赖于对其周边的各个像素进行平均，但是FNLMD的中心概念是在图像中的其他地方寻找类似的像素，再对其取平均值。在这种情况下，像素被认为是相似的像素，不是因为它的颜色或强度相似，而是因为它在环境中是相似的。这里的关键是，许多图像包含重复的结构，因此即使像素被噪声破坏，也会有许多其他类似的像素。

基于以像素p和大小s为中心的窗口$B(p, s)$，进行类似像素的识别。在点的周围给定一个期望更新的窗口，我们可以将这个窗口与在其他像素q周围的窗口进行比较。我们如下定义$B(p, s)$和$B(q, s)$之间的平方距离：

$$d^2(B(p, s), B(q, s)) = \frac{1}{3(2s+1)} \sum_{c=1}^{3} \sum_{j \varepsilon B(0,s)} (I_c(p+j) - I_c(q+j))^2$$

其中，c是色彩索引，$I_c(P)$是图像上点p在通道c的色彩强度，并且j的总和超过了patch元素。从这个平方距离，可以将权重分配给相对于当前正在更新的像素的每隔一个像素。这个权重由下式给出：

$$w(p, q) = e^{-\frac{\max(d^2-2\sigma^2, 0.0)}{h^2}}$$

在这个权重函数中，σ是噪声中预期的标准偏差（强度单位），而h是一个通用的过滤参数，可以确定补丁的速度随着平均距离从我们更新的补丁增长而变得无关紧要。一般来说，增加h的值将增加去除的噪声，但是会牺牲一些图像细节。降低h的值会保留细节，同时增加噪音。

通常情况下，考虑到选择与要更新的补丁相距较远（以像素为单位）的点，返回结果会相应递减，因为在一定距离内，这样的补丁数量呈平方增加。因此，通常会定义一个称为“搜索窗口”的整体区域，只有搜索窗口中的修补程序有助于更新。然后使用给定的指数衰减权重函数，通过搜索窗口中所有其他像素的简单加权平均值来给出当前像素的更新。[注12]这就是为什么算法被称为“非本地”的原因。有助于修复给定像素的补丁仅与重新计算的像素的位置松散程度相关。

FNLMD的OpenCV实现包含几个不同的功能，每个功能适用于稍微不同的情况。

FNLMD基础算法cv::fastNlMeansDenoising()

```
void cv::fastNlMeansDenoising(
  cv::InputArray  src,                    // Input image
  cv::OutputArray dst,                    // Output image
  float           h                 = 3,  // Weight decay parameter
  int             templateWindowSize = 7, // Size of patches used for comparison
  int             searchWindowSize   = 21 // Maximum patch distance to consider
);
```

这四个函数中的第一个cv::fastNlMeansDenoising()，恰好实现了所描述的算法。我们使用templateWindowSize（用于比较的补丁尺寸）的补丁区域和衰减参数h从输入数组src计算出结果数组dst，并考虑searchWindowSize（允许范围内补丁的最大值）距离内的补丁。图像可以是一维、二维或三维，但必须是cv::U8类型。[注13]表11-2列出了算法作者提供的一些值，可以用来帮助设置衰减参数h。

注12：这里有一个不太容易掌握的地方，像素p在其自身重新计算中的贡献的重量是$w(p, p) = e^0 = 1$。通常，这会导致相对于其他相似像素的重量太高并且p值的变化很小。因此，p处的权重通常选择为区域$B(p, s)$内的像素的权重的最大值。

注13：请注意，虽然此方式下图像可以是一维、二维、三维，但它并不是处理彩色图像的最佳方式。对于彩色图像，最好使用cv::fastNlMeansDenoisingColored()。

表11-2：cv::fastNlMeansDenoising()的灰度图像的推荐值

噪声:σ	块大小:s	搜索窗口	衰减参数:h
$0 < \sigma \leq 15$	3 × 3	21 × 21	$0.40 \cdot \sigma$
$15 < \sigma \leq 30$	5 × 5	21 × 21	$0.40 \cdot \sigma$
$30 < \sigma \leq 45$	7 × 7	35 × 35	$0.35 \cdot \sigma$
$45 < \sigma \leq 75$	9 × 9	35 × 35	$0.35 \cdot \sigma$
$75 < \sigma \leq 100$	11 × 11	35 × 35	$0.30 \cdot \sigma$

FNLMD 彩色图像算法cv::fastNlMeansDenoisingColor()

```
void cv::fastNlMeansDenoisingColored(
  cv::InputArray  src,                    // Input image
  cv::OutputArray dst,                    // Output image
  float           h                = 3, // Luminosity weight decay parameter
  float           hColor           = 3, // Color weight decay parameter
  int             templateWindowSize = 7, // Size of patches used for comparison
  int             searchWindowSize   = 21 // Maximum patch distance to consider
);
```

FNLMD算法的第二个变体用于彩色图像。它只接受cv::U8C3类型的图像。虽然原则上可能直接将算法或多或少地应用于RGB图像，但实际上，最好将图像转换为不同的颜色空间进行计算。函数cv::fastNlMeansDenoisingColored()首先将图像转换为LAB颜色空间，然后应用FNLMD算法，然后将结果转换回RGB。其主要优点是在颜色方面实际上有三个衰减参数。然而，在RGB表示中，不太可能将其中任何一个设置为不同的值。但是在LAB空间中，为颜色分量指定不同的衰减参数对于亮度分量是很自然的。函数cv::fastNlMeansDenoisingColored()可以让你做到这一点。参数h用于亮度衰减参数，而新参数hColor用于颜色通道。一般来说，hColor的值会比h小很多。在大多数情况下，10是一个合适的值。表11-3列出了一些可用于帮助设置衰减参数h的值。

表11-3：彩色图像的cv::fastNlMeansDenoising()的推荐值

噪声:σ	块大小:s	搜索窗口	衰减参数:h
$0 < \sigma \leq 25$	3 × 3	21 × 21	$0.55 \cdot \sigma$
$25 < \sigma \leq 55$	5 × 5	35 × 35	$0.40 \cdot \sigma$
$55 < \sigma \leq 100$	7 × 7	35 × 35	$0.35 \cdot \sigma$

FNLMD视频图像算法cv::fastNlMeansDenoisingMulti() 和cv::fastNlMeansDenoisingColorMulti()

```
void cv::fastNlMeansDenoisingMulti(
cv::InputArrayOfArrays srcImgs,              // Sequence of several images
cv::OutputArray        dst,                  // Output image
int                    imgToDenoiseIndex,    // Index of image to denoise
```

```
  int                    temporalWindowSize,     // Num images to use (odd)
  float                  h = 3,                  // Weight decay parameter
  int                    templateWindowSize = 7, // Size of comparison patches
  int                    searchWindowSize = 21   // Maximum patch distance
);
void cv::fastNlMeansDenoisingColoredMulti(
  cv::InputArrayOfArrays srcImgs,                // Sequence of several images
  cv::OutputArray        dst,                    // Output image
  int                    imgToDenoiseIndex,      // Index of image to denoise
  int                    temporalWindowSize,     // Num images to use (odd)
  float                  h                = 3,   // Weight decay param
  float                  hColor           = 3,   // Weight decay param for color
  int                    templateWindowSize = 7, // Size of comparison patches
  int                    searchWindowSize   = 21 // Maximum patch distance
);
```

第三和第四变体用于顺序图像，例如可能从视频捕获的图像。在顺序图像的情况下，很自然地想象除了当前帧以外的帧可能包含去噪像素的有用信息。在大多数应用中，图像之间的噪声不会是恒定的，而信号可能会相似甚至相同。函数cv::fastNlMeansDenoisingMulti()和cv::fastNlMeansDenoisingColorMulti()期望一个图像数组srcImgs，而不是单个图像。另外，通过参数imgToDenoiseIndex告知函数序列中哪个图像是要被去噪的。最后，必须提供一个时间窗口，指示在去噪中使用的序列中的图像数量。该参数必须为奇数，隐含窗口始终以imgToDenoiseIndex为中心（因此，如果要将imgToDenoiseIndex设置为4并将temporalWindowSize设置为5，那么在去噪中使用的图像将为2，3，4，5和6。）

直方图均衡化

相机和图像传感器不仅可以适应场景中自然产生的对比度，还可以管理图像传感器在可用的光照水平下的曝光程度。在标准相机中，设置快门和镜头光圈以确保传感器既不太多也不太少。然而，对于传感器的可用动态范围，特定图像中的对比度范围往往太大。因此，捕获需要更长曝光时间的黑暗区域（例如阴影）和需要更短曝光的明亮区域之间存在权衡，以避免饱和“白化”。在许多情况下，在同一个图像中二者不可兼得。

拍摄完成后，传感器记录的东西对我们来说毫无用处，然而，我们可以采取一些措施尝试扩大图像的动态范围以增加对比度。最常用的技术是直方图均衡。[注14]在图11-10中，我们可以看到左侧的图像很差，因为值的范围没有太大的变化。右边的强度值的直方图

注14：直方图均衡是一种古老的数学技术；它在图像处理中的应用在各种教科书中有所描述[Jain86; Russ02; Acharya05]，会议论文[Schwarz78]，甚至生物学视觉[Laughlin81]。为什么直方图均衡不在直方图那一章（第13章）中介绍？因为直方图均衡不能直观地使用直方图数据类型。尽管在内部使用直方图，但功能（从用户的角度来看）根本不需要直方图。

体现了这一点。因为我们处理的一个8位图像，其强度值的范围可以是0～255，但直方图显示实际的强度值全部聚集在可用范围的中间附近。直方图均衡是拉伸该范围的方法。

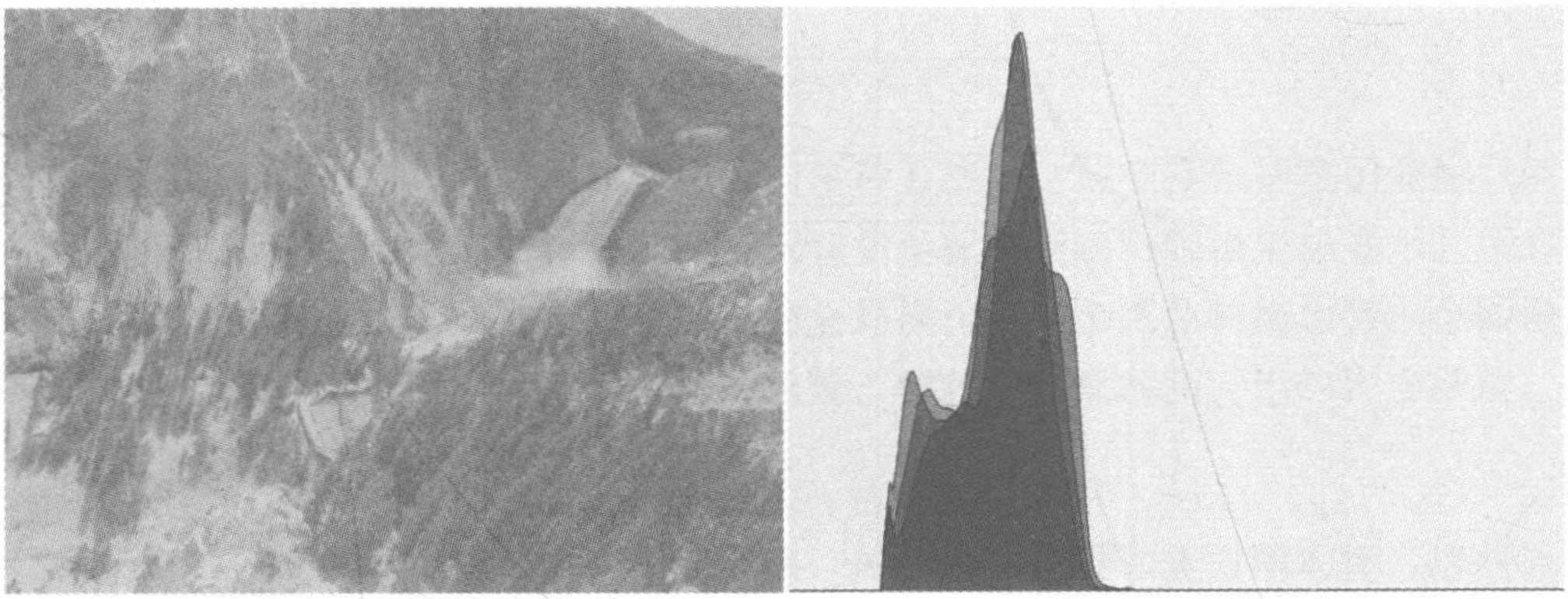


图11-10. 左侧图像对比度差，正如右侧强度值的直方图所证实的

直方图均衡数学背景是将一个分布（强度值的给定直方图）映射到另一个分布（强度值的更宽和理想的均匀分布）。也就是说，我们希望在新分配中尽可能均匀分布原始分布的y值。事实证明，解决扩展分布值的问题的一个好方法是：重映射函数应该是累积分布函数。如图11-11所示，是累积分布函数的一个例子，对于原始纯高斯的密度分布有些理想化的情况。然而，累积密度可以应用于任何分布；原始分布的运行总和从负到正的范围。

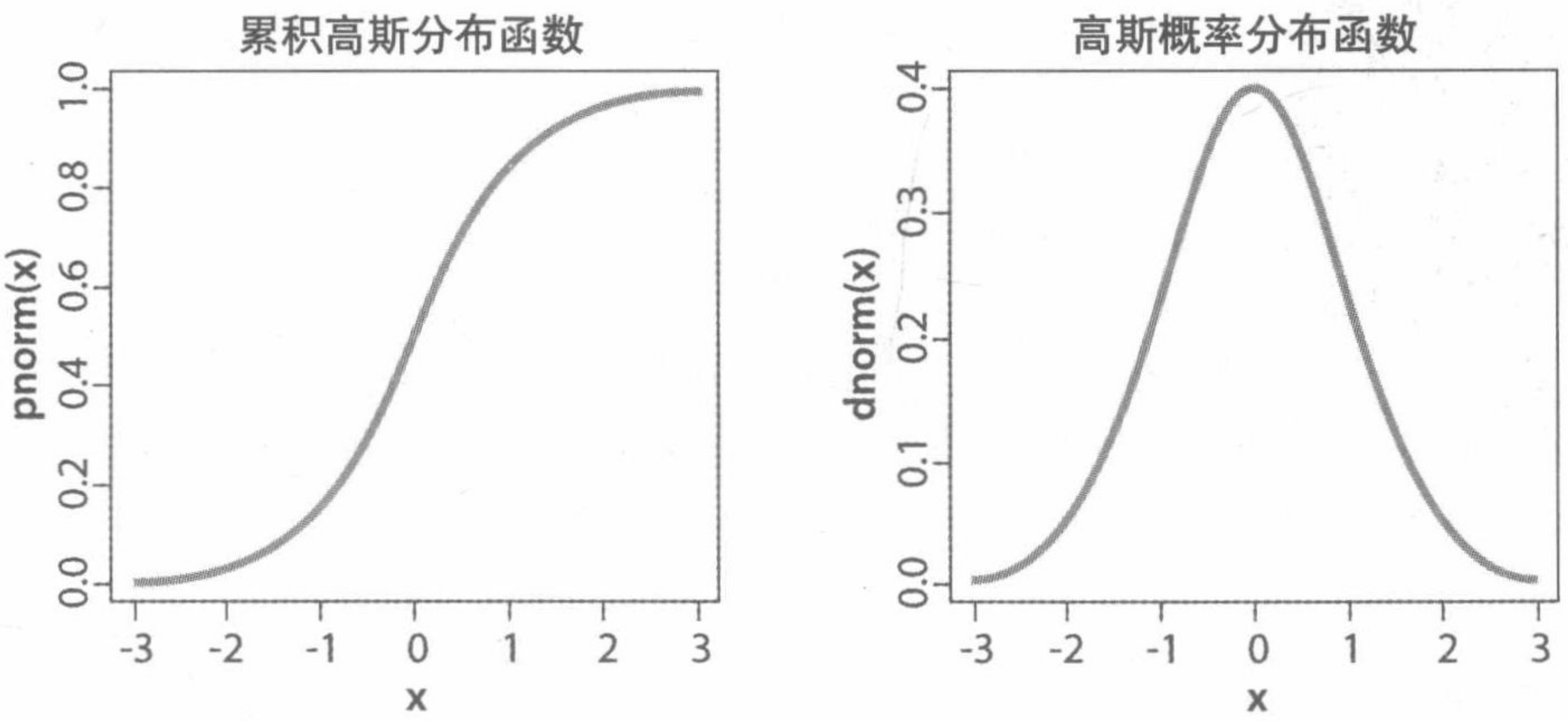


图11-11. 对于高斯分布计算的累积分布函数（左）的结果（右）

我们可以使用累积分布函数将原始分布重新映射到均匀分布（图11-12），只需查看原始分布中的每个y值，并查看在均衡分布中应该进行的位置。对于连续分布 结果将是一个精确的均衡，但是对于数字/离散分布，结果可能很不一致。

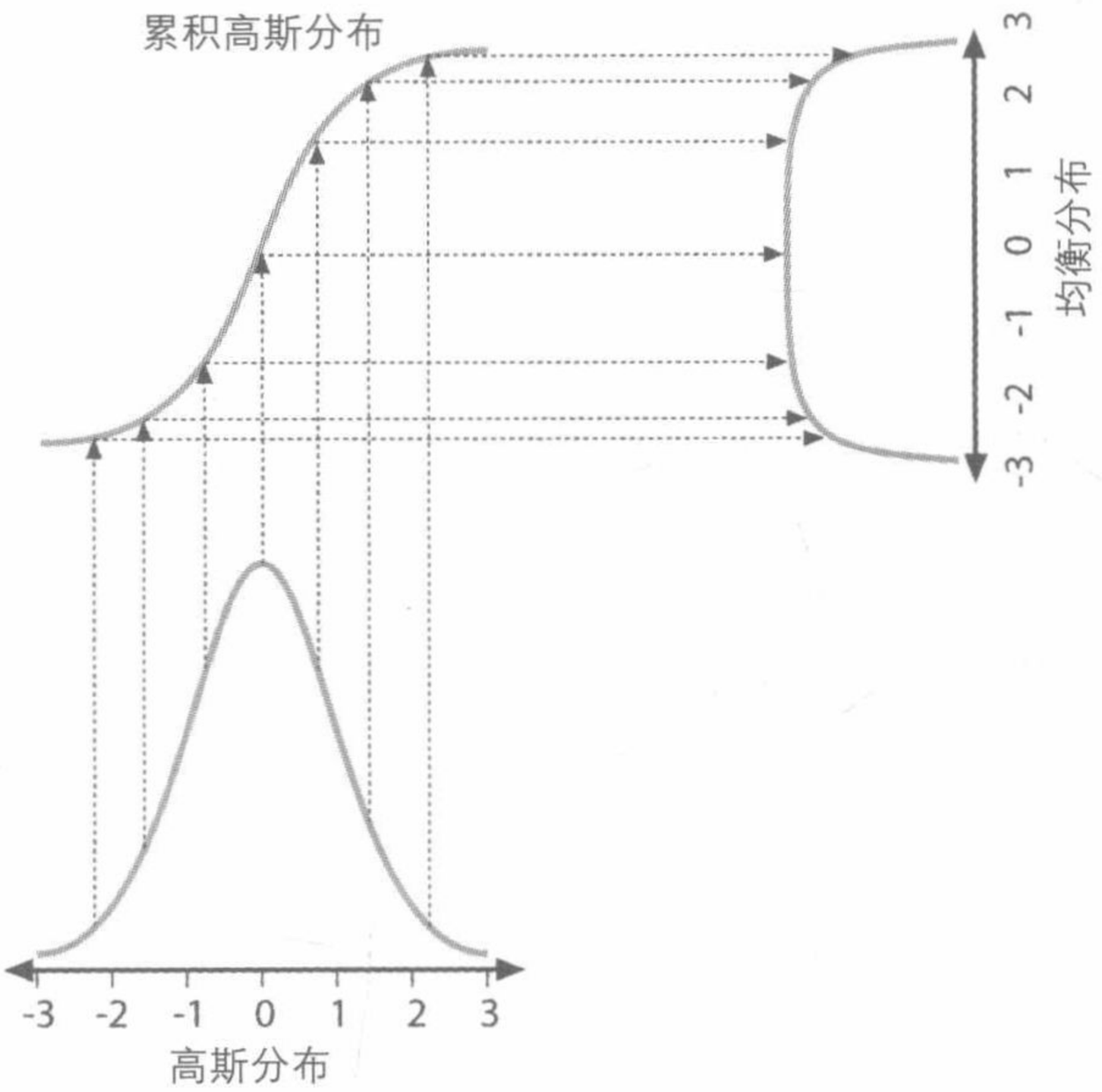


图11-12. 使用累积密度函数均衡高斯分布

将这种均衡过程应用于图11-10，得到均衡的强度分布直方如图11-13所示。

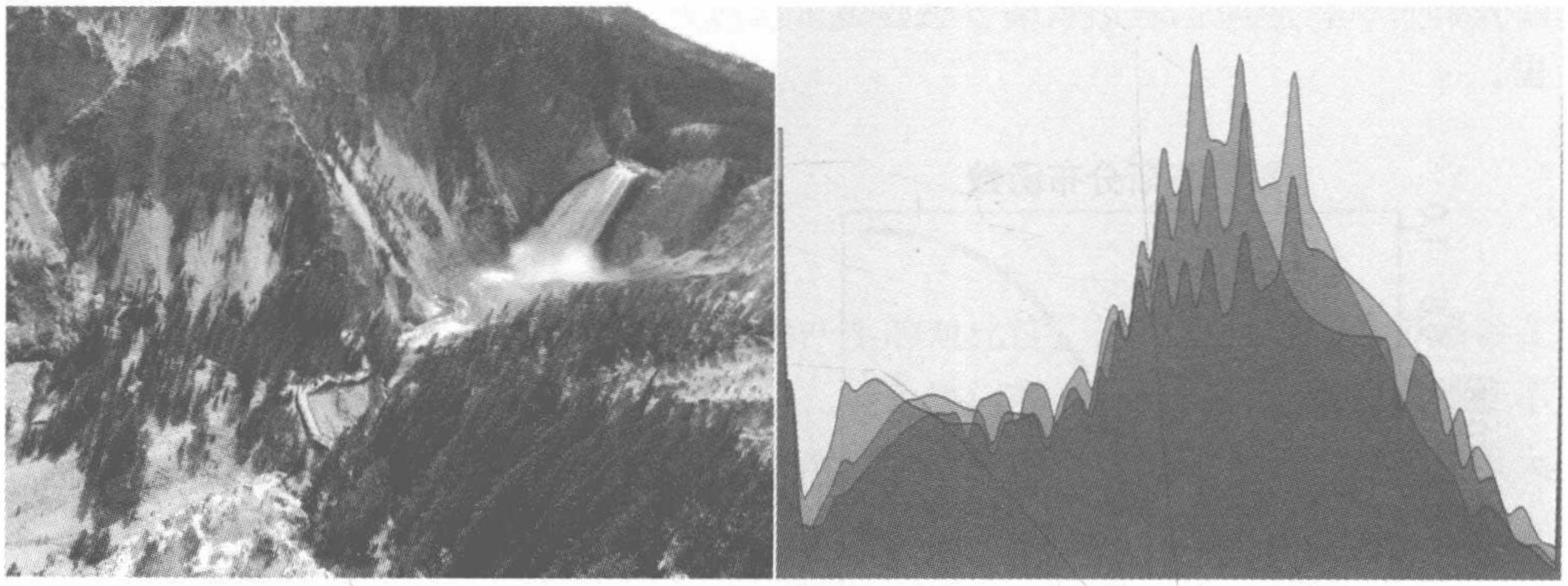


图11-13. 直方图均衡结果：频谱已经展开

cv::equalizeHist()用于对比均衡

OpenCV将整个过程整合在一个函数中。

```
void cv::equalizeHist(
  const cv::InputArray   src,                    // Input image
  cv::OutputArray dst                            // Result image
);
```

在函数cv::equalizeHist()中，源图src必须是8位一维图像。目标图像dst也是8位一维图像。对于彩色图像，必须分通道逐一处理。[注15]

小结

在本章中，我们学习了可以用来转换图像的各种方法。这些变换包括范围变换以及仿射和透视变换。我们学习了如何将矢量函数从直角坐标重新映射到极坐标表示。所有这些功能的共同之处在于它们通过整个图像上的全局操作将一个图像转换为另一个图像。我们看到一个可以处理最通用的重新映射的函数，相对于本章前面讨论的许多函数，可以看作是特殊情况。

我们还遇到了一些在计算摄影中有用的算法，如修复、去噪和直方图均衡。这些算法通常用于处理来自相机摄像头视频流的图像，并且非常适用于在粒状或其他质量差的视频数据之上实现其他计算机视觉技术时。

练习

1. 查找并加载一幅脸部图像，这幅图像是一张双眼睁开，正脸占据图像的大部分区域。编码找出眼睛里的瞳孔。



拉普拉斯算法“喜欢”中心明亮，周边黑暗。反转并与一个足够大的拉普拉斯算法相结合。将此图反转。

2. 看看对数极坐标函数如何将正方形转换成波浪线的图表。
   a. 对数极点中心点位于正方形的一个角上，绘制对数极坐标结果。
   b. 中心点在圆圈内并且靠近边缘，那么在极坐标变换中，圆圈会如何？
   c. 中心点坐在圆圈外面，绘制变换后的图像。
3. 对数极化变换将不同旋转和尺寸的形状转换成空间，其中这些对应于θ轴和$\log(r)$轴上的偏移。傅里叶变换是平移不变的。我们如何使用这些事实来强制不同尺寸和旋转的形状自动在对数极域中给出等效表示？

注15：在实践中，对RGB图像中的每个通道单独应用直方图均衡不太可能给出美观的结果。可能一种更好的做法是，将LAB转换为更合适的空间，然后将直方图均衡应用于亮度通道。

4. 绘制大、小、大旋转和小旋转正方形的单独图片。分别进行这些对数极坐标变换。对二维移位器进行编码，该移位器将所得到的对数极坐标域中的中心点移动，并使形状尽可能相同。

5. 加载图像，进行透视变换，然后旋转。这一转变是否可以一步完成?

6. 修复作品对纹理区域的文字工作相当不错。如果文字掩盖了图片中的真实对象边缘，会发生什么？ 尝试一下。

7. 对加载的图像进行直方图均衡，并报告结果。

8. 解释图像的直方图均衡和去噪图像之间的差异。

第12章

图像分析

概览

在前一章中，我们了解了OpenCV为我们实现的图像变换，这些变换本质上是从输入图像到输出图像的映射，即输出仍是一幅图像，就像输入那样。

本章中，我们会考虑一些操作，将图像转换成一种可能完全不同的表现方式。

这些新的表现形式仍然是一些数组，只是这些数组中的值在含义上将与原图像中的强度值大不相同。比如，马上就要提到的方法：离散傅里叶变换，它的输出图像仍然是个数组，只不过是输入图像的频域表示。在某些方法中，变换结果会是一系列成分的序列而不是一个数组，霍夫线变换就属于这种情况。

最后，本书还将介绍一些图像分割方法，用来表示图像中有意义的连通域。

离散傅里叶变换

对于任意以离散参数为索引的数值集合，都可以通过与连续傅里叶变换相似的方法来定义离散傅里叶变换（DFT）。[注1]对于N个复数x_0，x_1，x_2，...，x_{N-1}，一维DFT由如下公式定义（其中$i=\sqrt{-1}$）：

注1：维瑟夫·傅里叶（Joseph Fourier）[Fourier]最先发现一个函数可以分解成一系列其他函数的表示，这就是后来的傅里叶分析。提出把函数分解成傅里叶级数的方法的一些关键文章有Morse的“Fourier transforms”[Morse53]以及Papoulisd的“The Fourier Integral and Its Applications”[Papoulis62]等。尽管卡尔·高斯（Carl Gauss）于1805年[Johnson84]首次提出了快速傅里叶变换的关键步骤，但实际上Cooley和Tukey[Cooley65]于1965年才发明了这个方法。早期用于计算机视觉的方法是Ballard和Brown[Ballard82]所描述的方法。

$$g_k = \sum_{n=0}^{N-1} f_n e^{-\frac{2\pi i}{N} kn}$$

相似的，对于二维数值数组，也可以定义变换（当然，更高维的也行）：

$$g_{k_x,k_y} = \sum_{n_x=0}^{N_x-1} \sum_{n_y=0}^{N_y-1} f_{n_x,n_y} e^{-\frac{2\pi i}{N}(k_x n_x + k_y n_y)}$$

一般，计算项数为N的的变换预计需要$O(N^2)$次运算。实际上，有几种快速傅里叶变换（FFT）算法可以在的复杂度内计算这些值。

cv::dft()离散傅里叶变换

OpenCV的cv::dft()函数实现了一种这样的FFT算法。函数cv::dft()可以计算一维和二维输入数组的FFT。在后一种情况下，可以计算二维变换，或者如果需要，也可以只对每一行单独进行一维变换，这样做比多次调用cv::dft()快很多。

```
void cv::dft(
  cv::InputArray    src,                    // Input array (real or complex)
  cv::OutputArray   dst,                    // Output array
  int               flags        = 0,       // for inverse, or other options
  int               nonzeroRows = 0         // number of rows to not ignore
);
```

输入阵列必须是浮点型，可以是单通道或双通道。如果是单通道数组，那么数组必须是实数组，并且输出将以复数共轭对（CCS）的形式保存。[注2]如果输入是双通道的，那么两个通道分别代表实部和虚部。在这种情况下，不会出现特殊的结果，当然输入和输出数组的一些空间会被大量的0填充而浪费。[注3]

单通道CCS输出结果的值如下所示。

一维数组：

$$\left[Re\,Y_0 \quad Re\,Y_1 \quad Im\,Y_1 \quad Re\,Y_2 \quad Im\,Y_2 \quad \ldots \quad Re\,Y_{\left(\frac{N}{2}-1\right)} \quad Im\,Y_{\left(\frac{N}{2}-1\right)} \quad Re\,Y_{\left(\frac{N}{2}\right)} \right]$$

注2：用这种紧凑的形式保存，结果就是单通道图像的输出数组在大小上和输入数组一样，因为为0的元素都被忽略了；对于双通（复数）道数组，输出理所应当的与输入保持一致。

注3：在这种情况下，必须明确将双通道表示的虚部设为0。一种简单的方法是使用cv::Mat::zeros()创建全0数组，然后用cv::merge()将实部与之整合成为一个复数数组以调用cv::dft()。这样调用会获得一个完整的、未打包的复数频谱矩阵。

二维数组：

$$\begin{bmatrix} Re\,Y_{00} & Re\,Y_{01} & Im\,Y_{01} & Re\,Y_{02} & Im\,Y_{02} & \cdots & Re\,Y_{0,\frac{N_x}{2}-1} & Im\,Y_{0,\frac{N_x}{2}-1} & Re\,Y_{0,\frac{N_x}{2}} \\ Re\,Y_{10} & Re\,Y_{11} & Im\,Y_{11} & Re\,Y_{12} & Im\,Y_{12} & \cdots & Re\,Y_{1,\frac{N_x}{2}-1} & Im\,Y_{1,\frac{N_x}{2}-1} & Re\,Y_{1,\frac{N_x}{2}} \\ Im\,Y_{20} & Re\,Y_{21} & Im\,Y_{21} & Re\,Y_{22} & Im\,Y_{22} & \cdots & Re\,Y_{2,\frac{N_x}{2}-1} & Im\,Y_{2,\frac{N_x}{2}-1} & Im\,Y_{2,\frac{N_x}{2}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ Re\,Y_{\frac{N_y}{2}-1,0} & Re\,Y_{N_y-3,1} & Im\,Y_{N_y-3,1} & Re\,Y_{N_y-3,2} & Im\,Y_{N_y-3,2} & \cdots & Re\,Y_{N_y-3,\frac{N_x}{2}-1} & Im\,Y_{N_y-3,\frac{N_x}{2}-1} & Re\,Y_{N_y-3,\frac{N_x}{2}} \\ Im\,Y_{\frac{N_y}{2}-1,0} & Re\,Y_{N_y-2,1} & Im\,Y_{N_y-2,1} & Re\,Y_{N_y-2,2} & Im\,Y_{N_y-2,2} & \cdots & Re\,Y_{N_y-2,\frac{N_x}{2}-1} & Im\,Y_{N_y-2,\frac{N_x}{2}-1} & Im\,Y_{N_y-2,\frac{N_x}{2}} \\ Re\,Y_{\frac{N_y}{2},0} & Re\,Y_{N_y-1,1} & Im\,Y_{N_y-1,1} & Re\,Y_{N_y-1,2} & Im\,Y_{N_y-1,2} & \cdots & Re\,Y_{N_y-1,\frac{N_x}{2}-1} & Im\,Y_{N_y-1,\frac{N_x}{2}-1} & Re\,Y_{N_y-1,\frac{N_x}{2}} \end{bmatrix}$$

花点时间注意一下数组的下标，这是很有意义的，数组中某些数值为0，准确地说，一些值为实数。另外，表中最后一行仅当为偶数时存在，最后一列仅当为偶数时存在。在二维数组被视为个独立的一维数组而不是一个完整的二维变换的情况下（我们将看看如何做到这一点），有输出结果将类似于一维数组以单行的形式排列。

第三个参数flags正好说明了要完成的操作。通常把flags当作一个位数组，所以你可以将自己需要的任何标志与布尔OR组合。本书将从前向变换开始，这也是默认选项。除了指数和系数的符号出现变化，逆变换[注4]定义的形式和前向变换完全相同。要进行没有比例因子的逆变换，使用标识cv::DFT_INVERSE即可。尺度因子的标志为cv::DFT_SCALE，意味着输出的所有项尺度因子为N^{-1}，如果是二维变换则尺度因子是$(N_xN_y)^{-1}$。为了使正变换之后逆变换的结果与初始保持一致，尺度变换是必须的。由于人们通常将cv::DFT_INVERSE和cv::DFT_SCALE组合使用，所以这种操作有几种简写符号。除了组合这两个操作之外，可以使用cv::DFT_INV_SCALE（或者cv::DFT_INVERSE_SCALE，如果不想用简写的话）。最后一个可能需要的标识是cv::DFT_ROWS，这个标识告诉cv::dft()将二维数组当作一个一维数组的集合，每个数组应分别转换为长度为个不同的向量。当一次进行大量变换时，这样做可以显著减少开销。通过使用cv::DFT_ROWS，还可以实现三维或更高维的DFT。

尽管前向变换默认生成CCS格式的结果（输出数组的大小与输入数组完全相同），你仍然可以通过使用标识cv::DFT_COMPLEX_OUTPUT让OpenCV不这么做，获得一个复数组。相对而言，当对一个复数组进行逆变换时，结果也是一个复数组。如果原数组具有复共

注4：逆变换中，输入以之前表述的特殊形式打包。这样做是因为如果我们首先调用前向DFT，然后对结果进行逆DFT，结果将与原始数据保持一致，当然，一定要记得使用cv::DFT_SCALE标志。

轭对称性[注5]，你可以通过使用标识cv::DFT_REAL_OUTPUT让OpenCV产生一个比输入数组小的纯实数组。

为了理解最后一个参数nonzero_rows，我们还得先知道点儿别的东西。一般而言，DFT算法倾向于某些特定长度的输入向量，而不是其他长度的输入向量；类似的，对于数组来说，也倾向于某些特定大小的数组，而不是其他大小的数组。大多数DFT算法，这个大小最好是2的幂（比如大小为2^n的某个整数）。在OpenCV使用的算法中，优先考虑的向量长度或数组维数为$2^p3^q5^r$的整数 ，其中p、q和r是整数。因此常用的方法是创建一个更大的数组，然后将原数组复制至此，空白位置用0填充。为了方便起见，有一个方便的效用函数cv::getOptimalDFTSize()，它获取向量的（整数）长度后，会返回一个特定大小的值（这样的序列中，大于等于原向量长度的第一个值）。虽然这些填充是需要的，但仍然可以指定cv::dft()不进行这些行变换，已以保留真实数据。（或者，在进行逆变换时，也可以指定哪些行是你不在意的。）无论哪种情况，你都可以使用nonzero_rows指明有效数据的行数来节省计算时间。

cv::idft()用于离散傅里叶逆变换

如之前所见，函数cv::dft()不仅可以实现离散傅里叶变换，还可以实现逆变换（在提供正确的flags参数时）。但为了代码的可读性，应当使用有专门功能的函数进行逆变换。

```
void cv::idft(
  cv::InputArray  src,                          // Input array (real or complex)
  cv::OutputArray dst,                          // Output array
  int             flags        = 0,             // for variations
  int             nonzeroRows = 0               // number of rows to not ignore
);
```

调用cv::idft()等价于调用标志为cv::DFT_INVERSE的cv::dft()（当然前提是不为cv::idft()不提供任何标志）。

cv::mulSpectrums()频谱乘法

在大多数涉及到傅里叶变换的计算中，还必须计算两个频谱的逐元素乘积。由于傅里叶变换的结果一般都是复数，通常又是高密度的CCS格式，所以在这种情况下进行解析并进行矩阵乘法是很繁琐的。幸运的是，OpenCV提供了函数cv::mulSpectrums()来实现这个功能。

注5：这里不是说它是CCS格式，只是说它具有对称性，如果它是一个纯实数组的前向变换，就有对称性。另外请注意，如果你告诉OpenCV输入数组有这种对称性，OpenCV就会这么认为，不再检测这个对称性。

```
void cv::mulSpectrums(
  cv::InputArray  src1,                    // Input array (ccs or complex)
  cv::InputArray  src2,                    // Input array (ccs or complex)
  cv::OutputArray dst,                     // Result array
  int             flags,                   // for row-by-row computation
  bool            conj = false             // true to conjugate src2
);
```

前两个参数是数组，必须同时为CCS格式单通道频谱或双通道复数频谱，这可以通过cv::dft()获取。第三个参数是目标数组，大小和类型将与输入数组保持一致。最后一个参数*conj*告诉cv::mulSpectrum()你的具体目的。其中，设置为false实现了源数组的乘法；设置为真实现了前一个数组的元素与第二个数组的复共轭的乘法。[注6]

使用傅里叶变换进行卷积

可以通过卷积定理[Titchmarsh26]使用DFT大大提高卷积的速度，该卷积定理将空间域中的卷积与傅里叶域中的乘积相关联[Morse53; Bracewell65; Arfken85]。为了实现这一点，我们首先计算图像的傅里叶变换，然后计算卷积滤波器的傅里叶变换；一旦完成，我们在变换空间中完成卷积的时间就能缩减到与图像像素数成线性。下面是卷积的源代码，很值得一看，因为它提供了使用cv::dft()的很多很好的范例。代码如示例12-1所示，是从OpenCV的参考中直接引用的。

示例12-1：用cv::dft()和cv::idft()加速卷积计算

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Fourier Transform\nUsage: " <<argv[0] <<" <imagename>" << endl;
    return -1;
  }

  cv::Mat A = cv::imread(argv[1],0);

  if( A.empty() ) { cout << "Cannot load " << argv[1] << endl; return -1; }

  cv::Size patchSize( 100, 100 );
  cv::Point topleft( A.cols/2, A.rows/2 );
  cv::Rect roi( topleft.x, topleft.y, patchSize.width, patchSize.height );
  cv::Mat B = A( roi );
```

注6：此参数的主要用法是傅里叶空间中的相关性的实现。事实证明，卷积（我们将在下一节讨论）和相关性之间的唯一区别是频谱乘法中第二个阵列的共轭。

注7：只要数据大小合适，OpenCV的DFT算法实现的FFT就可以足够快。

```
int dft_M = cv::getOptimalDFTSize( A.rows+B.rows-1 );
int dft_N = cv::getOptimalDFTSize( A.cols+B.cols-1 );

cv::Mat dft_A = cv::Mat::zeros( dft_M, dft_N, CV::F32 );
cv::Mat dft_B = cv::Mat::zeros( dft_M, dft_N, CV::F32 );

cv::Mat dft_A_part = dft_A( Rect(0, 0, A.cols,A.rows) );
cv::Mat dft_B_part = dft_B( Rect(0, 0, B.cols,B.rows) );

A.convertTo( dft_A_part, dft_A_part.type(), 1, -mean(A)[0] );
B.convertTo( dft_B_part, dft_B_part.type(), 1, -mean(B)[0] );

cv::dft( dft_A, dft_A, 0, A.rows );
cv::dft( dft_B, dft_B, 0, B.rows );

// set the last parameter to false to compute convolution instead of correlation
//
cv::mulSpectrums( dft_A, dft_B, dft_A, 0, true );
cv::idft( dft_A, dft_A, DFT_SCALE, A.rows + B.rows - 1 );

cv::Mat corr = dft_A( Rect(0, 0, A.cols + B.cols - 1, A.rows + B.rows - 1) );
cv::normalize( corr, corr, 0, 1, NORM_MINMAX, corr.type() );
cv::pow( corr, 3., corr );

cv::B ^= cv::Scalar::all( 255 );

cv::imshow( "Image", A );
cv::imshow( "Correlation", corr );
  cv::waitKey();

  return 0;

}
```

在示例12-1中，我们可以看到，首先是创建和初始化输入数组。接下来，创建两个新的阵列，其尺寸对DFT算法是最佳的。将原始数组复制到这些新数组中，然后计算变换。最后，将频谱相乘，逆变换的结果就是最终的输出。变换是整个操作中最慢的部分[注8]，一幅N × N的图像需要为之耗费$O(N^2 \log N)$的时间，因此,将这个时间考虑成整个处理过程时间的近似（假设卷积核大小为$M \times M$，且$N > M$）。这个速度远比非离散傅里叶变换卷积所耗费的时间$O(N^2M^2)$快。

注8：“最慢”，我们的意思是“渐近的最慢”，换句话说，算法这一部分花最大的时间为非常大的N。这个区别非常重要。在实践中，正如我们前面关于卷积的部分所看到的，转换到傅里叶空间的开销并不总是最佳的。一般来说，在使用小内核进行卷积时，进行这种转换是不值得的。

cv::dct()离散余弦变换

对于实数数据，通常很容易计算其离散傅里叶变换。实际上这一过程实际需要的计算量只有表面看起来的一半。离散余弦变换（DCT）[Ahmed74; Jain77]与完全离散傅里叶变换定义类似，方程如下：

$$c_k = \left(\frac{1}{N}\right)^{\frac{1}{2}} x_0 + \sum_{n=1}^{N-1} \left(\frac{2}{N}\right)^{\frac{1}{2}} x_n \cos\left(\left(k + \frac{1}{2}\right)\frac{n}{N}\pi\right)$$

当然，更高维也有类似的变换。注意，按照惯例，余弦变换及其逆变换都应该采用归一化因子（而在离散傅里叶变换中则没有此要求）。

DFT的基本思想同样适用于DCT，只是所有系数都变成了实数。[注9]实际函数调用如下所示：

```
void cv::dct(
  cv::InputArray  src,                    // Input array (even size)
  cv::OutputArray dst,                    // Output array
  int             flags = 0               // for row-by-row or inverse
);
```

函数cv::dct()所需要的参数与cv::dft()相似，由于结果是实数值，因此没必要对结果数组或输入数组进行特别的打包。与cv::dft()不同的是，输入数组的大小必须是偶数。如果不是，可以在数组最后补0使其变成偶数。参数flags可以设置为cv::DCT_INVERSE以进行逆变换，并且可以与cv::DCT_ROWS组合使用，与cv::dft()具有相同的效果。由于归一化方式的不同，余弦变换的前向变换和逆变换都包含它们对变换的全局归一化的贡献；因此，cv::dct()没有cv::DFT_SCALE选项。

与cv::dft()一样，cv::dct()的性能很大程度上依赖数组大小。其实，如果深入分析，cv::dct()的实现实际上是在一个数组上调用cv::dft()，这个数组大小恰恰就是输入数组的一半。因此，cv::dct()所需数组的最佳大小刚好是cv::dft()所需的两倍。综合考虑，计算cv::dct()最佳大小的最好方式如下：

```
size_t optimal_dct_size = 2 * cv::getOptimalDFTSize( (N+1)/2 );
```

其中的N是要转换的数据的实际大小。

注9：精明的读者会发现，不能将余弦变换用于非偶函数的向量。不过，cv::dct()简单用镜像的方式将向量在负数上进行扩展，使其变成偶函数。

cv::idct()离散余弦逆变换

就像cv::idft()和cv::dft()一样，cv::dct()可以使用flags参数来计算逆余弦变换。像之前说的一样，为了更好的代码可读性，通常使用单独的逆变换函数代替：

```
void cv::idct(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Output array
  int             flags = 0,              // for row-by-row computation
);
```

调用cv::idct()等价于调用标志为cv::DCT_INVERSE的cv::dct()（前提是不向cv::idct()传递其他标志）。

积分图

通过OpenCV的cv::integral()函数，你可以轻松地计算积分图。积分图[Viola04] 是一种允许子区域快速求和的数据结构。[注10]这种求和在很多方面都很有用，值得一提的是Harr小波的计算，它用于人脸识别和类似的算法。

OpenCV支持积分图的三种变体，分别是总和、平方求和以及倾斜求和。每种情况的结果图像在图像的每个方向上都加1之后，与原始图像的大小相同。

计算标准积分图像总和形式如下：

$$sum(x, y) = \sum_{y'<y} \sum_{x'<x} image(x', y')$$

平方和图像是平方的和：

$$sum_{square}(x, y) = \sum_{y'<y} \sum_{x'<x} [image(x', y')]^2$$

倾斜求和与标准求和相似，只是累加方向旋转了45度：

$$sum_{tilted}(x, y) = \sum_{y'<y} \sum_{abs(x'-x)<y} image(x', y')$$

通过积分图，你可以对图像的任意直立或“倾斜”的矩形区域求和、平均以及标准差。举个简单的例子，如果要求由(x^1, y^1)和(x^2, y^2)描述的矩形的和，其中$x^2 > x^1$且$y^2 > y^1$，我们可以这样计算：

注10：引文进一步介绍了这种方法，但实际上，同名作者2001年才在论文“Robust Real-Time Object Detection”中向计算机视觉领域引入了该概念，这种方法早在1984年就被应用于计算机图形学中，其中，积分图被称为“总和区域表”。

$$\sum_{y_1 \le y < y_2} \sum_{x_1 \le x < x_2} image(x, y) = [sum(x_2, y_2) - sum(x_1, y_2) - sum(x_2, y_1) + sum(x_1, y_1)]$$

这样一来，即使目标区域大小不确定，也可以高效进行快速模糊、近似梯度、求均值以及标准差，即使对可变大小的窗口，也可执行快速块相关。

为了介绍更清楚，我们来看一幅如图12-1所示的图像。这个区域看起来有点像柱状图，其中柱体的高度表示所在像素的亮度。图12-2以矩阵的形式显示这些信息，左边是原图而右边是积分图。我们对它进行标准求和得到积分图$I'(x, y)$，这种方法采用逐行的方式计算，积分图中每个位置的值由之前计算过的值与当前像素计算而来。公式如下：

$$\sum_{y_1 \le y < y_2} \sum_{x_1 \le x < x_2} image(x, y) = [sum(x_2, y_2) - sum(x_1, y_2) - sum(x_2, y_1) + sum(x_1, y_1)]$$

最后一项要记得减去，因为在加第二项和第三项的时候，该值被重复计数。可以通过图12-2验证这一点。

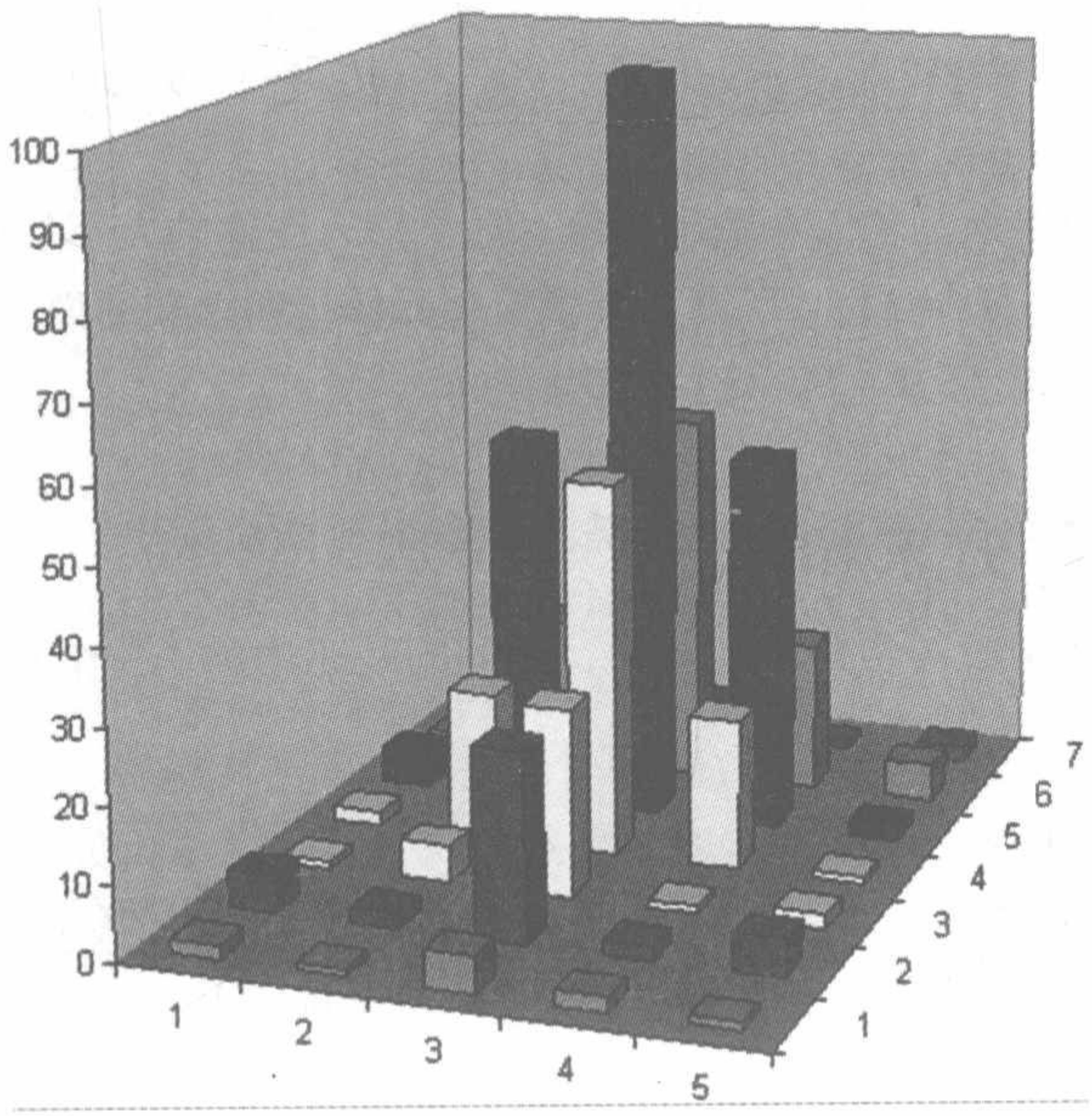


图12-1. 一幅用柱状图表示的简单的7×5的图像，柱体高度等于像素值

使用积分图计算一个区域时，比如图12-2，如果要计算中间被20包围的矩形区域，我们是可以这样计算的，即398－9－10 + 1=380。因此，使用四个量就可以对任意矩形区域计算并且将计算复杂度控制为$O(1)$。

1	2	5	1	2
2	**20**	**50**	**20**	5
5	**50**	**100**	**50**	2
2	**20**	**50**	**20**	1
1	5	25	1	2
5	2	25	2	5
2	1	5	2	1

0	0	0	0	0	0
0	1	3	8	9	11
0	3	**25**	**80**	**101**	108
0	8	**80**	**235**	**306**	315
0	10	**102**	**307**	**398**	408
0	11	108	338	430	442
0	16	115	370	464	481
0	18	118	378	474	492

图12-2. 左图是图12-1中7×5图的数值表示（原点假定在左上角），右图是8×6大小的积分图

cv::integral()标准求和积分

在C++的API中，不同形式的积分通过参数来区分。计算基本总和的形式只有三个。

```
void cv::integral(
  cv::InputArray  image,                    // Input array
  cv::OutputArray sum,                      // Output sum results
  int             sdepth = -1               // Results depth (e.g., cv::F32)
);
```

第一个参数和第二个参数是输入和输出图像，如果输入图像的大小是$W \times H$,，输出图像的大小就是$(W + 1) \times (H + 1)$。[注11]第三个参数sdepth指明求和图像（结果图像）需要的深度。sdepth可以是CV::S32，CV::F32或CV::F64。[注12]

cv::integral()平方求和积分

计算平方求和采用的函数与常规求和一样，不同的是还增加了一个额外的输出参数用于保存平方求和。

```
void cv::integral(
  cv::InputArray  image,                    // Input array
  cv::OutputArray sum,                      // Output sum results
  cv::OutputArray sqsum,                    // Output sum of squares results
  int             sdepth = -1               // Results depth (e.g., cv::F32)
);
```

cv::OutputArray类型的参数sqsum告诉cv::integral()除了标准求和还要计算平方和。与之前一样，sdepth指明目标图像的深度。sdepth可以是CV::S32，CV::F32或CV::F64。

注11：这允许全0行的存在，因为将0项相加结果还是o 0。

注12：值得注意的是，尽管sum和tilt_sum允许32位浮点型作为32位浮点型输入图像的输出，它仍然建议使用64位浮点型，特别是对于较大的图像。毕竟，现代图像的像素数量可以达到百万级。

cv::integral()倾斜求和积分

与平方求和类似，求倾斜求和积分有着同样的函数，还需要一个附加的参数来得到一个附加的结果。

```
void cv::integral(
  cv::InputArray  image,                    // Input array
  cv::OutputArray sum,                      // Output sum results
  cv::OutputArray sqsum,                    // Output sum of squares results
  cv::OutputArray tilted,                   // Output tilted sum results
  int             sdepth = -1               // Results depth (e.g., cv::F32)
);
```

除了其他的核，cv::OutputArray类型的附加参数titled作为一个附加的项由函数cv::integral()计算而来，除此之外，其他的参数全部相同。

Canny边缘检测

尽管已经可以用诸如Laplace滤波器之类的简单滤波器来曝光图像中的边缘，但这个方法还是可以改进的。1986年，简单的Laplace滤波器理论经过J. Canny于改进成为现在的Canny边缘检测器方法。Canny算法和上一章中描述的简单的基于Laplace算法的差异之一，在Canny算法中，先在*x*和*y*方向上求得一阶导，然后将它们组合成为四个方向的导数。其中方向导数是局部最大值的点是组成边缘的候选项。Canny算法最明显的创新点，就是将单个的边缘候选像素加入轮廓。[注13]

这个算法通过对像素应用滞后阈值来形成这些轮廓，也就是说算法采用了两个阈值，一个较大值和一个较小值。如果一个像素的梯度大于较大值，就接受；如果小于较小值，就抛弃；但如果介于两者之间，那么只有当它连接到一个高于阈值的像素时，才接受。Canny算法建议高阈值：低阈值介于2:1和3:1之间。图12-3和图12-4分别展示了cv::Canny()采用高：低为5:1和3:2是对一张测试图案和照片处理的结果。

注13：稍后我们将进一步讨论轮廓。在这之前，请注意，cv::Canny()实际上并不返回轮廓类型的对象，如果需要，必须使用cv::findContours()从cv::Canny()的输出结果构建它们。第14章将全面讲述细节你可能感兴趣的这些。

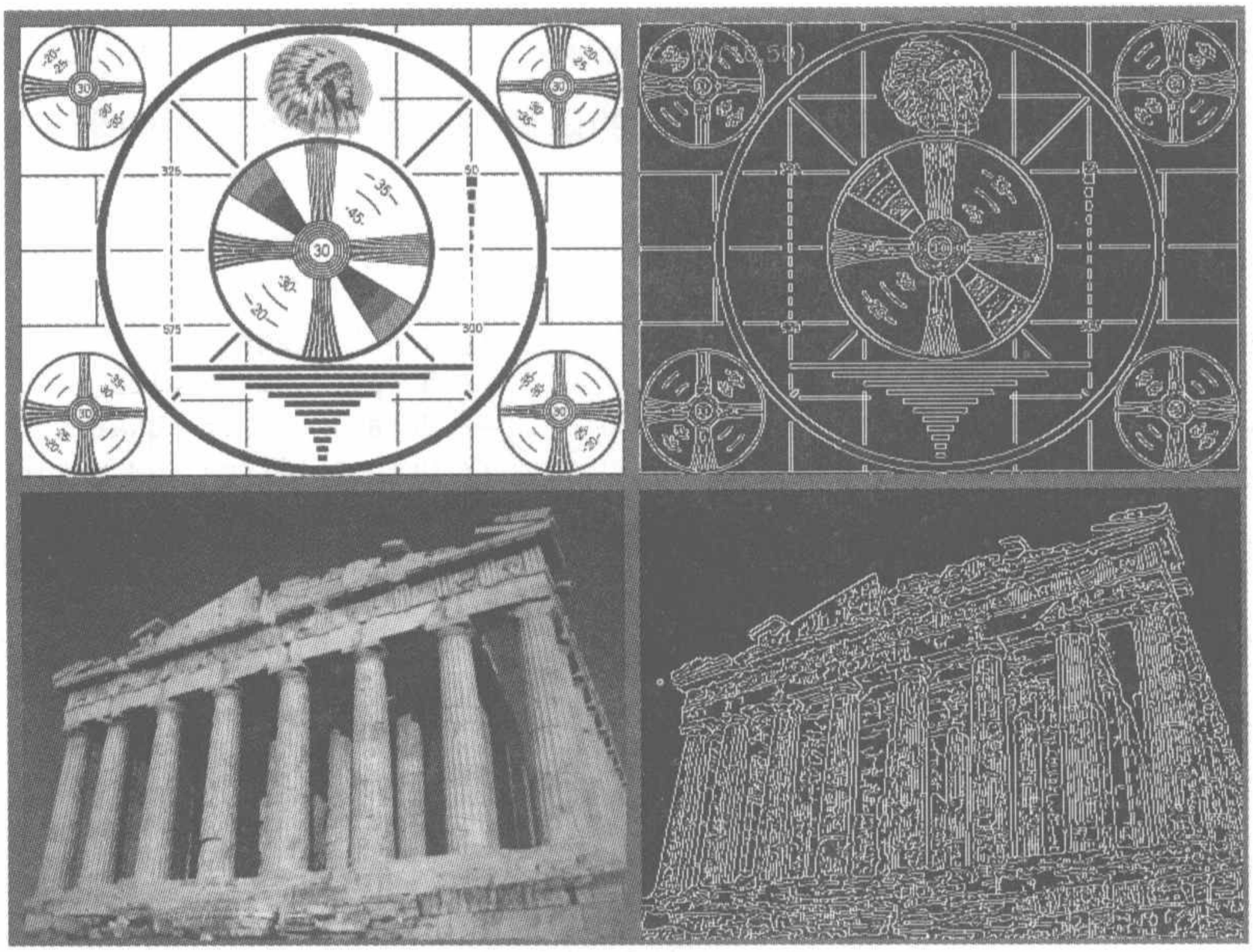


图12-3. 高低阈值分别为50和10时，两幅不同图像应用Canny边缘检测的效果

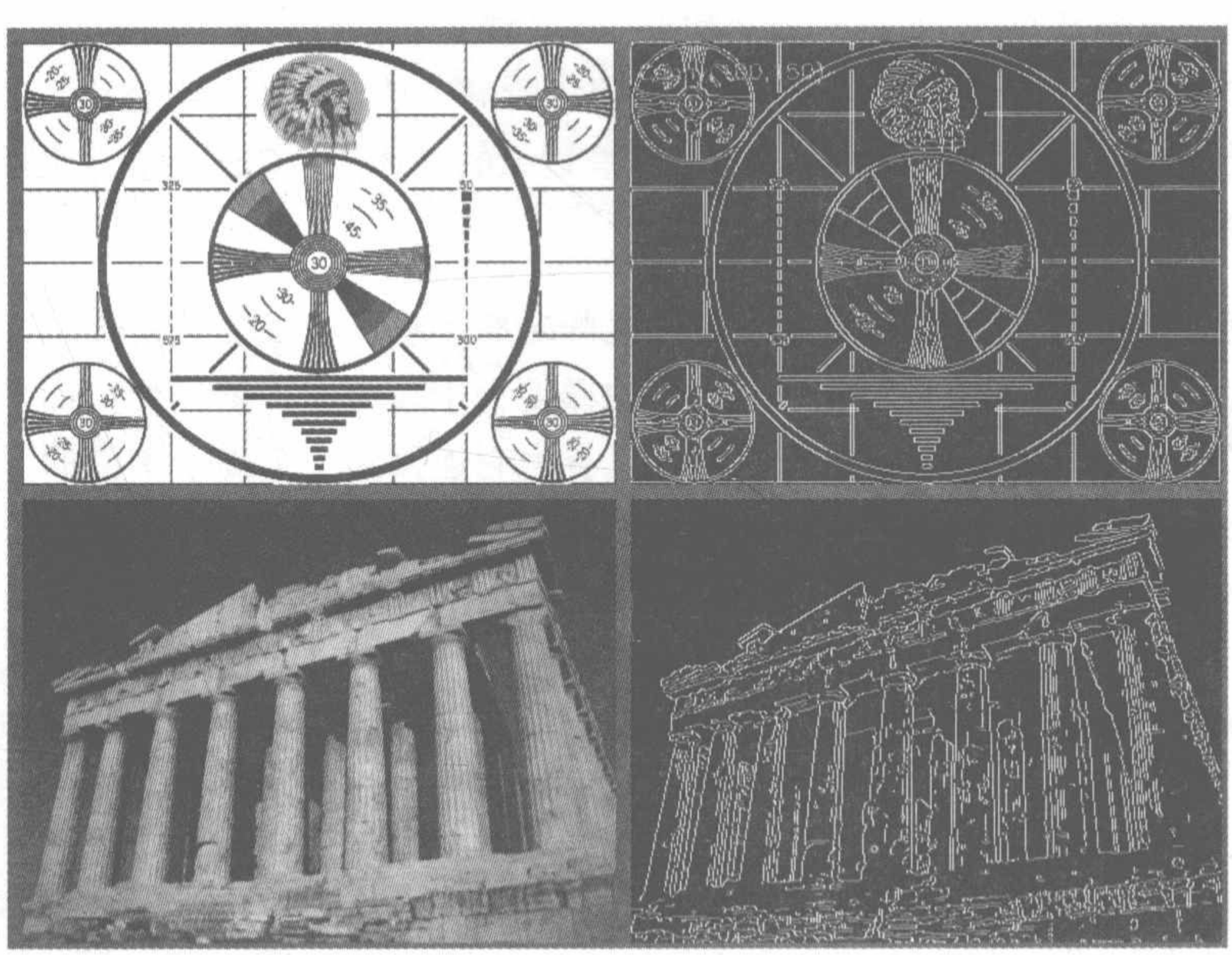


图12-4. 高低阈值分别为150和100时，Canny边缘检测对两幅不同图像的效果

cv::Canny()

Canny边缘检测算法的OpenCV实现将输入图像转换成“边缘图像”。

```
void cv::Canny(
  cv::InputArray  image,                    // Input single channel image
  cv::OutputArray edges,                    // Output edge image
  double          threshold1,               // "lower" threshold
  double          threshold2,               // "upper" threshold
  int             apertureSize = 3,         // Sobel aperture
  bool            L2gradient   = false      // true=L2-norm (more accurate)
);
```

cv::Canny()的参数中，输入图像必须是单通道图像，输出图像是灰度图（实际上是一幅布尔图）。接下来的两个参数是低阈值和高阈值。最后两个参数中，apertureSize是cv::Canny()内部调用的Sobel导数算子所用的aperture的大小；最后一个参数L2gradient用于选择是使用适当的L_2范数“正确”计算方向梯度，还是速度更快但精度更低的基于L_1范数的方法。如果L2gradient取true，就使用公式：

$$|grad(x, y)|_{L_2} = \sqrt{\left(\frac{dI}{dx}\right)^2 + \left(\frac{dI}{dy}\right)^2}$$

否则使用：

$$|grad(x, y)|_{L_1} = \left|\frac{dI}{dx}\right| + \left|\frac{dI}{dy}\right|$$

Hough变换

Hough（霍夫）变换[注14]是一种用于检测线、圆或者图像中其他简单形状的方法。最初Hough变换是一种线变换，这是一种相对较快的检测二值图像中直线的方法，可以进一步推广到除简单线之外的情况。

Hough线变换

Hough线变换的基本理论是：二进制图像中的任何点都可能属于某些可能的线。如果我们将每一条线参数化，如斜率为a，截距为b，原始图像中的点就可以转换为对应于通过该点的所有线在该平面(a, b)中的点的轨迹，当然可能只是一部分轨迹（图12-5）。如果我们将原图中每个非0像素都转换成输出图像中这样的一系列点，并将所有这些贡献相

注14：霍夫研究这种变换时，本来是打算用于物理实验的[Hough59]，Duda和Hart（杜达和哈特）把它引入了计算机视觉[Duda72]。

加，那么原图（(*x*, *y*)平面）中的线将显示为输出图像（(*a*, *b*)平面）中的局部最大值。由于我们对每个点的贡献进行求和，所以(*a*, *b*)平面通常称为“累加器平面”。

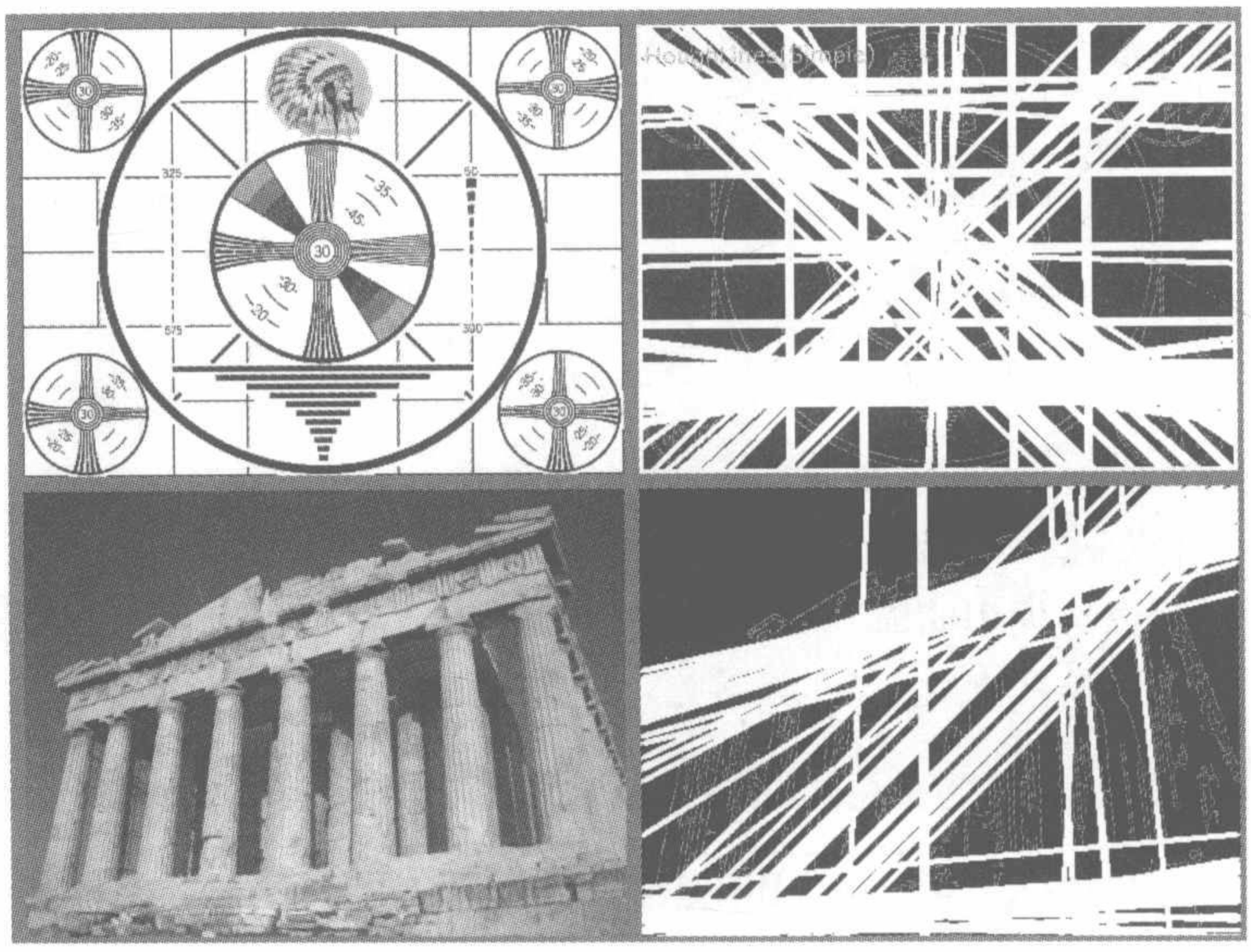


图12-5. Hough线变换在图中找出大量的线；有些线有用，但有些线无用

你可能会发现，斜截式方程并不是代表所有穿过某个点的线的最佳方式（因为图像中线的密度与斜率有很大的不同，还有一个相关事实是，斜率的区间可能从 $-\infty$ 到 ∞。正因如此，图像变换时数值计算实际用到的参数有所不同。最好的参数形式是用极坐标中的点(ρ, θ)表示线，每条直线经过这个点并且垂直于它与原点的连线（图12-6）。一条线的方程如下所示：

$$\rho = x\cos\theta + y\sin\theta$$

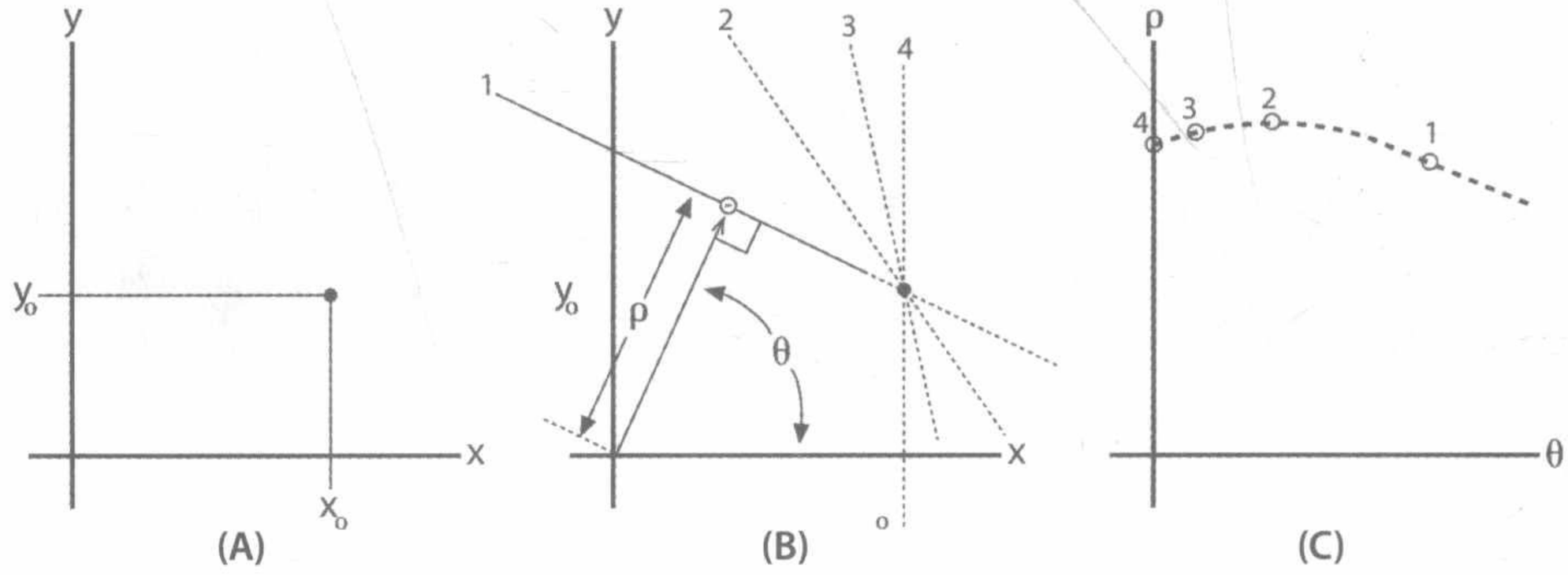


图12-6. 图像平面（A）中有一点可以表示很多直线，每条直线都可以由（B）中的ρ和θ参数化；在(ρ, θ)平面中，这些线组合在一起形成特殊形状（C）的曲线

OpenCV中Hough变换算法没有把这些计算显示给用户，它只是简单返回(ρ, θ)平面中的局部最大值。但为了理解OpenCV中Hough线变换函数的参数，你还是有必要把这个过程了解一遍的。

OpenCV支持三种Hough线变换：标准Hough变换（SHT）、多尺度Hough变换（MHT）和渐进概率Hough变换（PPHT）。[注15]SHT就是我们刚刚说过的变换算法。MHT算法是对上一种算法的细化，对线的匹配精度更高。PPHT是一种改进算法，这种算法除了计算一条独立直线的方向外还计算它的延长线（图12-7）。之所示说是“概率性的”，是因为它没有对平面中的每一个可能点都进行累积，它只累积其中的一小部分。这个方法的思想是：如果峰值足够高，那么只需要花很少的时间就足够获得结果了，这个猜想的结果可以大幅度减少计算时间。

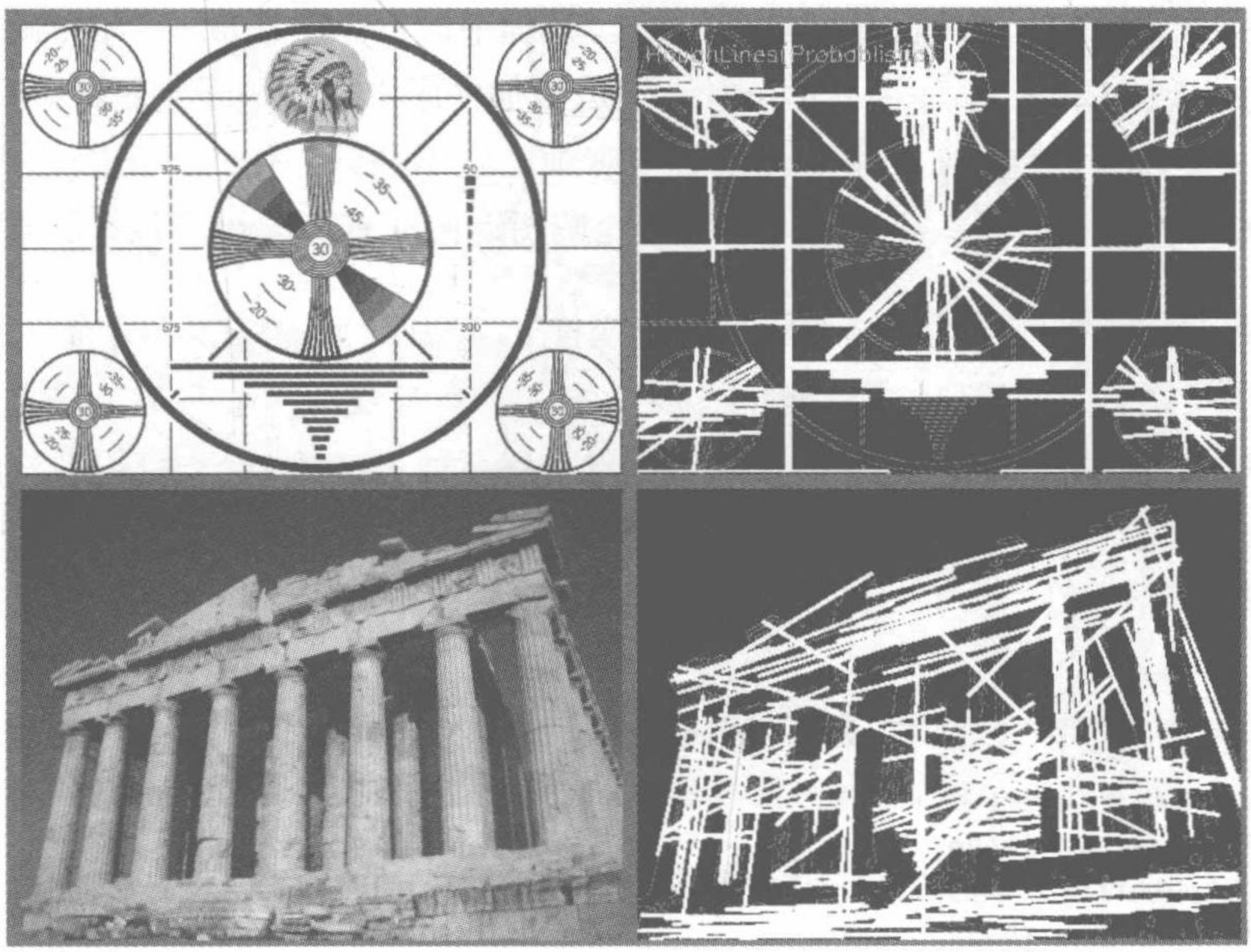


图12-7. Canny边缘检测器(param1=50, param2=150)的效果如第一幅图所示，线条为灰色；渐进概率Hough变换(param1=50, param2=150)的效果如第二幅图所示，线条为白色。可以看到Hough变换将明显的线条提取出来了

cv::HoughLines()：标准和多尺度Hough变换

标准和多尺度Hough变换都由函数cv::HoughLines()实现，两种不同的用法由两个可选参数区分。

注15：概率霍夫变换（PHT）由Kiryati，Eldar和Bruckshtein三人在1990年[Kiryati91]提出，PPHT由Matas，Galambos和Kittler三人于1999年[Matas00]提出。

```
void cv::HoughLines(
  cv::InputArray  image,                  // Input single channel image
  cv::OutputArray lines,                  // N-by-1 two-channel array
  double          rho,                    // rho resolution (pixels)
  double          theta,                  // theta resolution (radians)
  int             threshold,              // Unnormalized accumulator threshold
  double          srn      = 0,           // rho refinement (for MHT)
  double          stn      = 0            // theta refinement (for MHT)
);
```

第一个参数是输入图像，它必须是一幅8位图像，同时输入图像被当作是二值图（即非0像素含义全部一样）。第二个参数是结果存放位置，是一个$N \times 1$的双通道浮点型数组（列数N是返回的直线数目）。[注16]两个通道分别包含每条线的rho（ρ）和theta（θ）值。

接下来的两个参数rho和theta设置直线所需的分辨率（即累加平面的分辨率）。rho的单位是像素，而theta的单位是弧度，因而，累加平面可以考虑成一个由θ弧度大小为ρ像素的单元组成的二维直方图。threshold是累加平面中算法用于判断线条属于一条直线的阈值。最后一个参数在使用的时候有些麻烦，它不是归一化的，所以你应该用图像大小对它进行缩放。这个参数实际上指明了轮廓图返回的直线至少应包含的点的数量。

参数srn和stn用于控制称为“多尺度Hough变换”（MHT）的SHT算法的扩展，不用于标准Hough变换。在MHT中，这两个参数指明了每条直线所应达到的分辨率。MHT首先以rho和theta参数给出的精度计算线的位置，然后通过参数srn和stn来细化这些结果（rho的最终分辨率是ρ除以srn，theta的最终分辨率是θ除以stn）。将这些参数设为0时，函数将采用SHT算法。

用cv::HoughLinesP()渐进概率Hough变换

```
void cv::HoughLinesP(
  cv::InputArray  image,                  // Input single channel image
  cv::OutputArray lines,                  // N-by-1 4-channel array
  double          rho,                    // rho resolution (pixels)
  double          theta,                  // theta resolution (radians)
  int             threshold,              // Unnormalized accumulator threshold
  double          minLineLength = 0,      // required line length
  double          maxLineGap    = 0       // required line separation
);
```

函数cv::HoughLinesP()与cv::HoughLines()相似，但也有两个重要的区别。第一个是lines参数变成四通道（或者一个Vec4i类型的向量）。四通道分别是找出的线段两个端点的坐标(x_0,y_0)和(x_1,y_1)（按顺序）。第二个重要的区别是两个参数的含义。PPHT中，

注16：这通常由你传入的参数类型决定，可能是$N \times 1$的双通道数组，也可能是一个N项类型为Vec2f的std::vector<>向量。

minLineLength和maxLineGap参数设置了返回线段的最小长度，以及共线线段之间的最小间隔，防止算法把他们连接成一条。

Hough圆变换

Hough圆变换的方法与之前描述的线变换方法相似。粗略解释的话，就是如果你想尝试用完全类似的方法去做，就要从累加平面变成累加三维的体积，三维坐标分别是圆心的位置x、y和圆的半径r。但这样做意味着更大的内存需求和更慢的速度。OpenCV中圆变换的实现通过采用一种称为Hough梯度法的较为复杂的方法来避免了这个问题。

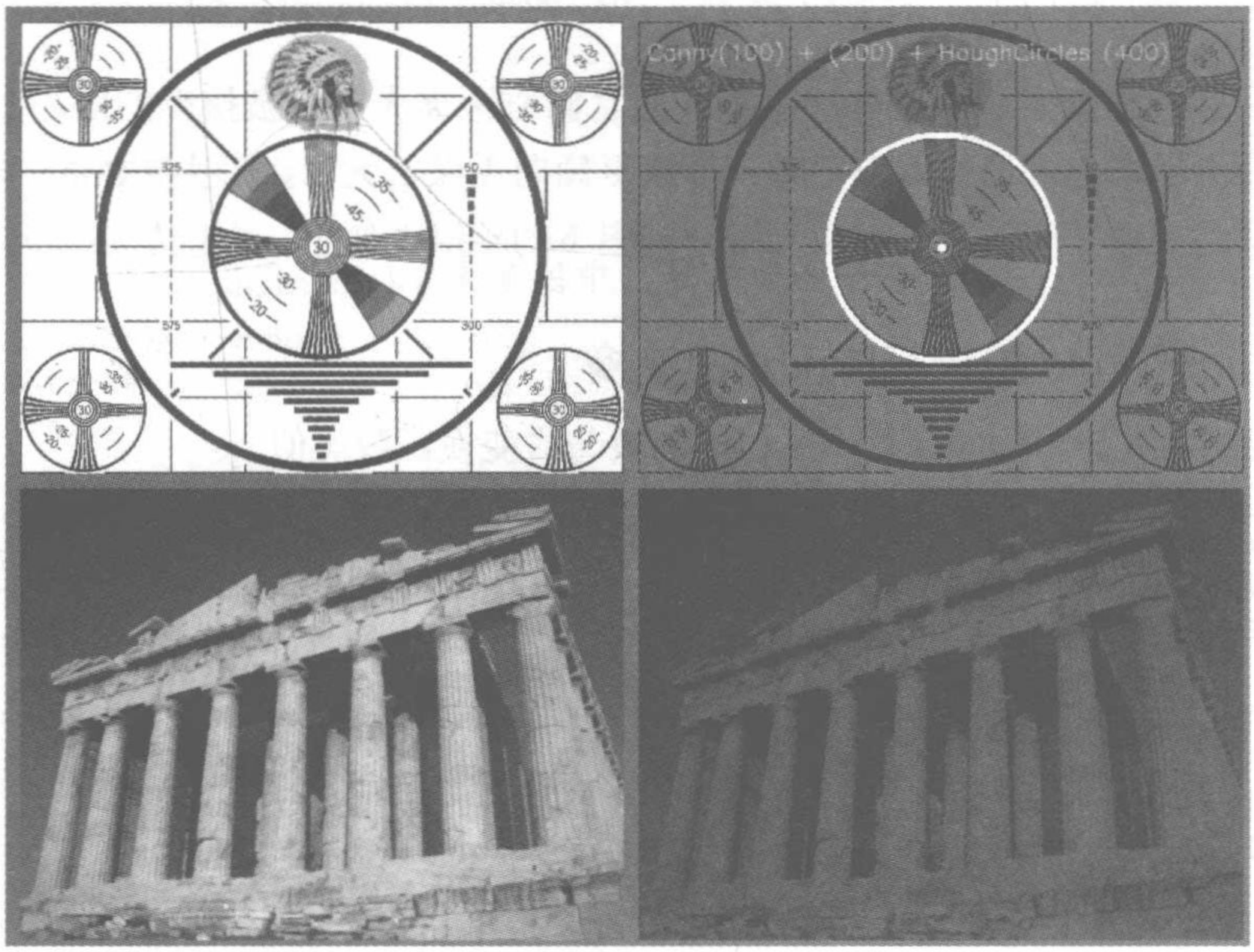


图12-8. Hough圆变换在测试图形中找到了圆，但在照片中没有找到圆

Hough梯度法工作过程如下。首先，对图像进行边缘检测（这里使用cv::Canny()）；然后，对每个轮廓图像中的非零点，考虑局部梯度（我们通过首先通过cv::Sobel()计算一阶Sobel x-导数和y-导数来计算梯度）。通过这个梯度，我们沿着这个斜率表示的线在累加器内从一个最小值到一个最大值遍历每个点，同时，记录轮廓图像中每个非零像素所在的位置。然后候选圆心就从这些（二维）累加器中分离出来，这些点都高于一个阈值且同时大于其所有直接相邻的点。这些点根据其累加器值的降序排列，使得最有可能是圆心的点排在前面。然后，对于每个圆心，考虑所有非零像素点（之前已经构建好该列表），将这些像素根据离圆心的距离排序。从最小距离到最大半径中选择一个最好的值作为圆的半径。如果有足够数量的点组合成一个圆并且其圆心与之前选中圆心的距离足够大，就保留这个圆心。

这种实现使得算法有足够的速度，更重要的是克服三维累加器的其他稀疏群体的问题，这将导致大量噪声并使结果不稳定。不过，这个算法也有一些需要注意的问题。

第一，用Sobel导数计算局部梯度——伴随的假设是这可以被认为相当于一个局部切线——不是一个数值稳定的命题，甚至“大部分时间”都是这样，但是你应该期望在输出中产生一些噪音。

第二，对每个候选圆心进行判断时要考虑轮廓图像中所有非零像素。因此，如果累加器阈值过低，算法就会很慢。

第三，因为对每个圆心都只能选择一个圆，所以如果出现同心圆，最终将只能得到一个。

最后，由于圆心根据累加器值升序排列，并且圆心距离先前被被接受的圆心太近时会被舍去，因而出现同心或近似同心圆时，算法更倾向于保留大圆。（由于Sobel导数产生的噪声，它只是一种“倾向”，但在无限分辨率下的平滑图像中，它就是必然的了。）

用cv::HoughCircles()进行Hough圆变换

Hough圆变换函数cv::HoughCircles()的参数与线变换函数类似。

```
void cv::HoughCircles(
  cv::InputArray  image,                  // Input single channel image
  cv::OutputArray circles,                // N-by-1 3-channel or vector of Vec3f
  int             method,                 // Always cv::HOUGH_GRADIENT
  double          dp,                     // Accumulator resolution (ratio)
  double          minDist,                // Required separation (between lines)
  double          param1    = 100,        // Upper Canny threshold
  double          param2    = 100,        // Unnormalized accumulator threshold
  int             minRadius = 0,          // Smallest radius to consider
  int             maxRadius = 0           // Largest radius to consider
);
```

输入图像是一幅8位图像，cv::HoughCircles()和cv::HoughLines()最大的区别就是后者需要的是二值图像。cv::HoughCircles()内部调用的是cv::Sobel()内部调用cv::Sobel()而不是cv::Canny()，其原因是cv::HoughCircles()需要估计每个像素的梯度方向，这在二值轮廓图中很难做到，因此可以使用更普遍的灰度图。

结果数组circles是一个矩阵数组或向量，取决于传入cv::HoughCircles()的内容。如果是个矩阵，它将是CV::F32C3类型的一维数组；三个通道分别是圆的坐标及半径。如果是个向量，那么它的类型是std::vector<Vec3f>。method参数的值实际设置为cv::HOUGHGRADIENT。

参数dp是所使用的累加图像的分辨率。这个参数允许我们创建一幅分辨率比输入图像更低的累加器。这样做是有意义的，因为我们并不能一直保证图中的圆会自然落入与图像

本身的宽度或高度相同数量的分区。如果dp设置成1，那么分辨率不发生变化，如果设置成一个较大的数比如2，累加器的分辨率会减小一半。dp的值不能小于1。

参数minDist是是两个圆之间必须存在的最小距离，以便算法可以判断出它们是两个圆。

cv::HOUGH_GRADIENT后面的两个参数param1和param2分别是边阈值和累加器阈值。你可能还记得Canny边检测器需要两个阈值，在内部调用cv::Canny()时，第一个（较高）阈值设置成传入cv::HoughCircles()的param1，第二个（较低）阈值设置成它的一半。参数param2用于阈值化累加器，和cv::HoughLines()的threshold相似。

最后两个参数是可以找到的圆的最小、最大半径，这意味着它们是累加器能够表示的圆的半径。

示例12-2是一个完整的程序。

示例12-2：用cv::HoughCircles()从灰度图中获取一组圆的序列

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <math.h>

using namespace cv;
using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Hough Circle detect\nUsage: " <<argv[0] <<" <imagename>\n" << endl;
    return -1;
  }

  cv::Mat src, image;

  src  = cv::imread( argv[1], 1 );
  if( src.empty() ) { cout << "Cannot load " << argv[1] << endl; return -1; }

  cv::cvtColor(src, image, cv::BGR2GRAY);
  cv::GaussianBlur(image, image, Size(5,5), 0, 0);

  vector<cv::Vec3f> circles;
  cv::HoughCircles(image, circles, cv::HOUGH_GRADIENT, 2, image.cols/10);

  for( size_t i = 0; i < circles.size(); ++i ) {
    cv::circle(
      src,
      cv::Point(cvRound(circles[i][0]), cvRound(circles[i][1])),
      cvRound(circles[i][2]),
      cv::Scalar(0,0,255),
      2,
      cv::AA
```

```
    );
  }

  cv::imshow( "Hough Circles", src);
  cv::waitKey(0);

  return 0;

}
```

值得反思的是，无论我们采用什么技巧，直线都只需要两个自由度，圆都要用三个自由度(x, y和r)表示。这必然会导致圆的搜索算法相比之前的线搜索算法需要更多内存和计算时间。考虑到这一点，就可以根据情况允许选择半径参数来控制系统开销。[注18] Hough变换在1981年由Ballard扩展到任意形状[Ballard81]，基本上都是将对象作为渐变边缘的集合来考虑。

距离变换

图像的距离变换定义为一幅新图像，其中每个输出像素的值被设为输入图像中与最近的零像素的距离——当然得根据某个特定的距离度量。不难看出，距离变换生成的是某种边缘图像。在大多数应用中，距离变换的输入是边缘检测器的输出，就像把Canny边缘检测器反过来（使得边缘变成0，其他非边缘变成非0）。

计算距离变换的方式有两种，第一种方法是使用通常为3×3或5×5阵列掩膜， 数组中的每个点都定义了与掩膜中心相对的特定位置的点的“距离”。更大的距离是作为掩膜成员定义的“移动”序列被构建的（并且因此是近似的）。使用这个方法时，给定一个特定的距离度量，OpenCV就会自动从集合中选择一种近似掩膜。这是Borgefors在1986年[Borgefors86]提出的“原始”方法。第二种方法计算精确的距离，由Felzenszwalb [Felzenszwalb04]提出。两种方法运算复杂度与像素数都是线性关系，但精确的算法会稍慢一点。

距离度量可以有多种不同的类型，包括经典的L2距离度量。图12-9展示了在图案和图像上进行距离变换的结果。

注18：尽管cv::HoughCircles()可以很好地找到圆心，但有时找不到正确的半径。因此，在只要求找到圆心（或者可以通过其他技术来找到实际半径）的程序中，cv::HoughCircles()返回的半径可以被忽略。

用户自定义掩膜3×3
(a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1	0	1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

用户自定义掩膜3×3
(a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1	0	1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

图12-9. 首先使用参数是param1=100，param2=200时运行Canny边缘检测器，然后进行距离变换，输出按比例缩放5倍以提高可视性

cv::distanceTransform()无标记距离变换

调用OpenCV距离变换函数，输出图像将为32位浮点图像（即cv::F32）。

```
void cv::distanceTransform(
  cv::InputArray  src,                // Input image
  cv::OutputArray dst,                // Result image
  int             distanceType,       // Distance metric to use
  int             maskSize            // Mask to use (3, 5, or see below)
);
```

cv::distanceTranform()有两个参数，第一个是distanceType，它表示要使用的距离度量，它可以选cv::DIST_C，cv::DIST_L1或cv::DIST_L2。这些方法根据网格上的整数步长计算到最接近的零的距离。它们的区别是：cv::DIST_C计算四连通距离（即不能沿对角线移动），cv::DIST_L1计算八连通距离（即能沿对角线移动）。当distanceType取cv::DIST_L2时，cv::distanceTransform()会尝试计算精确的欧式距离。

距离类型之后的参数是maskSize，可以是3、5或cv::DIST_MASK_PRECISE。取3或5时，该参数表示用Borgefors方法使用3×3或5×5的掩膜。如果使用cv::DIST_L1或cv::DIST_C，你可以随时使用3×3掩膜，并且可以得到精准结果。如果使用cv::DIST_L2，Borgefors方法将总是近似的，用5×5大掩膜可以更好地逼近L2距离，其代价是计算速度稍慢。或者，可以用cv::DIST_MASK_PRECISE调用Felzenszwalb算法（使用cv::DIST_L2时）。

cv::distanceTransform()有标记距离变换

调用距离变换算法时，不仅仅可以计算距离，也可以报告最小距离对应的对象。这些

“对象”称为连接组件，我们将在第14章中加以着重介绍，但是现在，可以把它们想象成一个由连续的0像素组成的结构体。

```
void cv::distanceTransform(
  cv::InputArray  src,                                // Input image
  cv::OutputArray dst,                                // Result image
  cv::OutputArray labels,                             // Connected component ids
  int             distanceType,                       // Distance metric to use
  int             maskSize,                           // (3, 5, or see below)
  int             labelType = cv::DIST_LABEL_CCOMP // How to label
);
```

如果提供一个标签数组，在执行cv::distanceTransform()后，会与dst的大小保持一致。这种情况下，将自动计算连接的组件，并将与最近的这种组件相关联的标签放置在标签的每个像素中，输出的“标签”数组基本上是离散Voronoi图又称“泰森多边形”或“诺图”。



如果想了解如何区分标签，请考虑src中为0的任何像素，则相应的距离也必须为0。另外，像素的标签必须是其所属连接组件的标签。因此，如果你想知道什么标签被赋予了任何特定的0像素，查找标签中的该像素即可。

参数labelType可以设置成cv::DIST_LABEL_CCOMP或cv::DIST_LABEL_PIXEL。在前一种情况下，该函数自动发现输入图像中零像素的连接分量，并给出每个独立的标签。在后一种情况下，所有零像素都被赋予不同的标签。

分割

图像分割是个很大的话题，我们已经在很多地方触及到了，在本书后面的内容中，将回到形态学的范畴内讨论它。这里，我们重点研究OpenCV中的几种专门实现分割方法的技术实现或者后面要用到的形态学策略。请注意，图像分割现在还没有一种“百试百灵”的解决方案，在计算机视觉研究中，它仍是一个非常活跃的领域。尽管如此，已经开发出许多可靠的技术，至少在某些特定的领域是可靠的，并且在实践中可以产生非常好的结果。

漫水填充

漫水填充[Heckbert90; Shaw04; Vandevenne04]是一种非常有用的方法，常用于标记或隔离图像中的部分以做进一步的处理或分析。漫水填充也可用于从输入图像中导出可

以由后续例程使用的掩膜，以加速处理过程或限制为仅由掩膜指示的那些像素。函数cv::floodFill()提供了一个可选参数mask用于控制填充区域，例如，对于同一图像的多个填充。

相比你可能已经接触过的经典计算机绘图程序，OpenCV中的漫水填充是一种更一般的填充方法。算法首先选取一个种子点，然后将所有与它相似的点包括本身上一种特定的颜色，区别是相邻像素不一定全部上同一种颜色。[注19]漫水填充的结果是一个单连通区域，如果像素灰度值与任一当前像素差异在指定范围（loDiff至upDiff）内或在指定原始种子像素灰度值的指定范围内，cv::floodFill()函数都将对它进行上色。漫水填充也可以通过可选的掩膜参数进行约束。cv::floodFill()有两种不同的原型，一种接受一个显式的mask参数，另一个不需要。

```
int cv::floodFill(
  cv::InputOutputArray image,                    // Input image, 1 or 3 channels
  cv::Point            seed,                     // Start point for flood
  cv::Scalar           newVal,                   // Value for painted pixels
  cv::Rect*            rect,                     // Output bounds painted domain
  cv::Scalar           lowDiff  = cv::Scalar(),// Maximum down color distance
  cv::Scalar           highDiff = cv::Scalar(),// Maximum up color distance
  int                  flags                    // Local/global, and mask-only
);

int cv::floodFill(
  cv::InputOutputArray image,                     // Input w-by-h, 1 or 3 channels
  cv::InputOutputArray mask,                      // 8-bit, w+2-by-h+2 (Nc=1)
  cv::Point            seed,                      // Start point for flood
  cv::Scalar           newVal,                    // Value for painted pixels
  cv::Rect*            rect,                      // Output bounds painted domain
  cv::Scalar           lowDiff  = cv::Scalar(), // Maximum down color distance
  cv::Scalar           highDiff = cv::Scalar(), // Maximum up color distance
  int                  flags                     // Local/global, and mask-only
);
```

参数image是输入图像，可以是8位或浮点型图像，通道数必须是1或3。cv::floodFill()函数会修改image数组。漫水填充将从种子点开始，种子点将设置成值newVal，然后算法就会对这些像素进行上色。如果一个像素的强度不小于一个着色相邻像素的强度减去loDiff，并且不大于着色相邻像素的强度加上upDiff，则该像素将被着色。如果flags参数包含了cv::FLOODFILL_FIXE_RANGE，那么比较时将会选用种子点而不是相邻像素。通常，flags参数控制填充的连接性，填充相对于何处填充，是否只填充一个掩膜区域，以及填充掩膜的值。图12-10显示了第一个漫水填充的例子。

注19：使用现代绘画和绘图程序的用户应该能注意到，那些程序现在大部分都使用了一种非常像cv::floodFill()的填充算法。

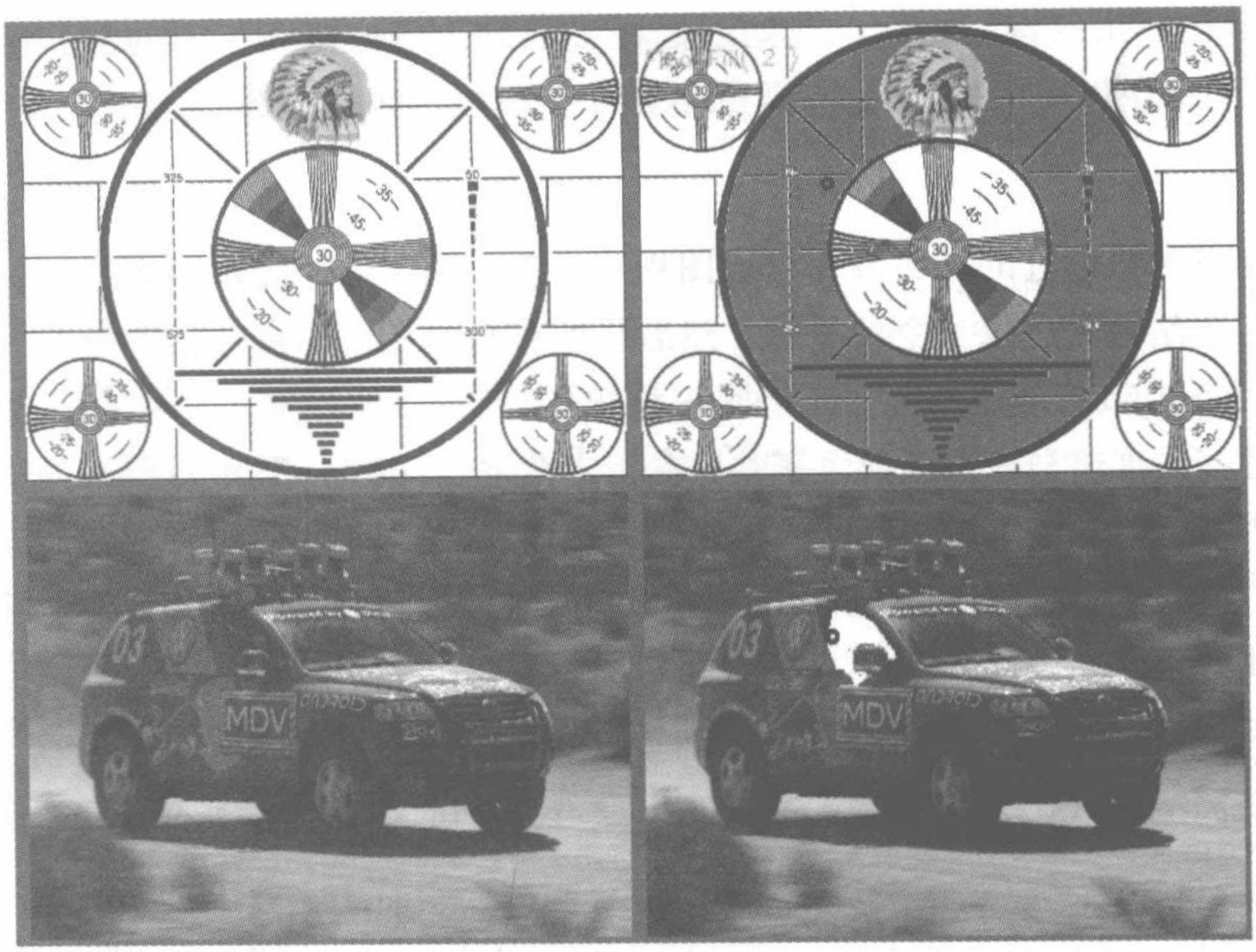


图12-10. 漫水填充结果（上图用灰色填充，下图用白色填充），从位于两个图像中心的黑色圆圈开始填充，upDiff和loDiff参数均为7.0

mask参数表示一个可以同时用作cv::floodFill()的输入的掩膜（在这种情况下，它限制可以被填充的区域），并且作为cv::floodFill()的输出（在这种情况下，它将指示实际填写的区域）。mask必须是单通道8位图像，大小比原图像宽高分别大2像素。[注20]

mask作为cv::floodFill()的输入时，算法不会填充掩膜中非零区域。因此，如果不想屏蔽掩膜操作，应该在使用前将其置零。

mask存在时，也将用于输出。当算法运行后，掩膜中每个“填充”像素将被设置为掩膜中的非零值。你可以向flags添加cv::FLOODFILL_MASK_ONLY（通过布尔OR操作）执行此选项。这样，输入图像就不会被修改。相反，只有掩膜会被修改。



如果使用漫水填充掩膜，填充过的像素就会被设为1。不要疑惑为什么填充后的掩膜里什么也没有却只有一片黑，填充值是确实存在的，但需要对掩膜图像的灰度值进行归一化才能显示。毕竟，0和1之间的灰度差相对0和255实在太小了。

注20：这样做可以使内部算法的处理变得更加容易和快捷。注意，由于掩膜比原始图像大，图像中的像素（x,y）对应于掩膜中的像素（$x+1,y+1$）。因此，你可能会发现这是使用cv::Mat::getSubRect()的绝佳机会。

flags可取的值有：cv::FLOODFILL_FIXED_RANGE和cv::FLOODFILL_MASK_ONLY。除了这些，你还可以为它加数值4或8。[注21]这样，你就声明了连通方式采用4连通还是8连通。前一种情况，4连通数组指的是与当前点距离最近的四个邻点（左、右、上和下）。而8连通情况下，对角线上连接的邻点也算在内。

flags参数有一点麻烦，因为它有三个部分表达的不是非常直观。如之前所见，为了控制漫水算法的连接性，低8位（0～7）可以设置成4或8。高8位（16～23）是包含了诸如cv::FLOODFILL_FIXED_RANGE和cv::FLOODFILL_MASK_ONLY标识的部分。中八位（8～15）的用法与实际展示的数值有一定区别，为填充时想填入的数值。如果flags中间的位全是0，就会向掩膜填充1（默认），其他的值会被认为是8位无符号整数。所有的标识可以用OR连接起来。比如你的漫水要求为八连通填充、只填充限定区域、只填充掩膜（而不是图像），同时用值47填充，那么应传入的参数为：

```
flags = 8
      | cv::FLOODFILL_MASK_ONLY
      | cv::FLOODFILL_FIXED_RANGE
      | (47<<8);
```

图12-11展示了在示例图像上进行漫水填充的结果。它在图像中心开始，使用范围广泛的cv::FLOODFILL_FIXED_RANGE，导致大部分图像区域都会被填充。需要注意，newVal、loDiff和upDiff的数据类型是cv::Scalar，因此可以同时对它们的三个通道赋值。比如，loDIff = cv::Scalar(20, 30, 40)将设置loDIff阈值为红色通道20，绿色通道30，蓝色通道40。

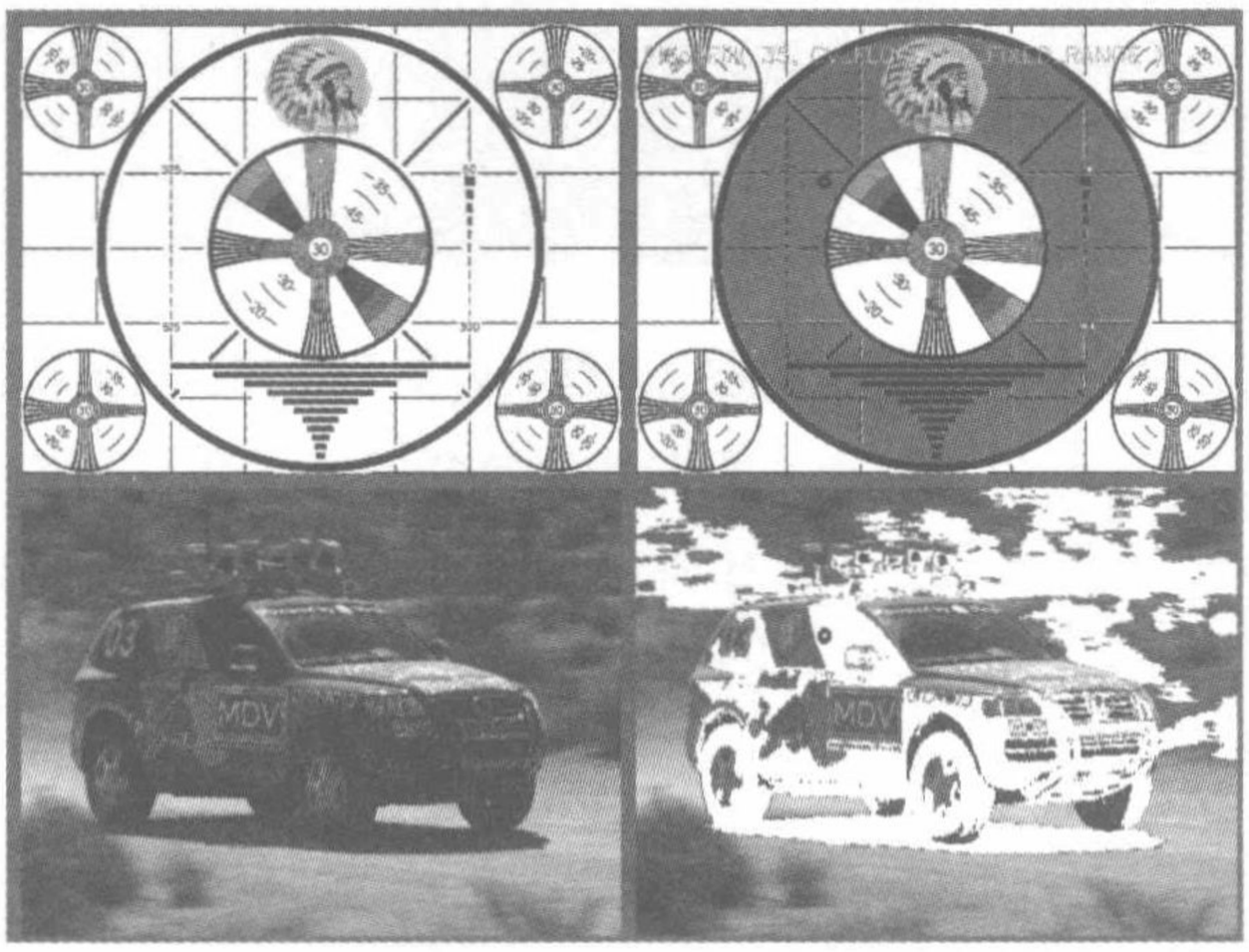


图12-11. 从中心位置开始漫水填充，上图填充灰色，下图填充白色；本例中，以固定范围的方式进行填充，高低差都为25.0

注21：这里写起来是“加”，但实际上，flags是一个位字段的参数。然而，方便的是，4和8是单个位，所以你可以使用“add”或“OR”，例如，flags = 8 | cv::FLOODFILL_MASK_ONLY）。

分水岭算法

在许多实际情况中，我们想分割一幅图像，但没有任何一种单独的背景蒙版的能用。这种情况下一种有效的技巧就是分水岭算法。这种算法将一幅图像中的线转化成“山”，平坦区域转换成“谷”以用于辅助物体分割。分水岭算法首先计算强度图像的梯度，这有助于在原图中形成没有纹理的谷或盆地（较低点），同时形成线条丰富的山脉或ranges（对应边缘的高脊）。然后从指定的点开始向这些盆地灌水。当图像被“填满”时，所有有标记的区域就被分割开了。这样一来，连通到标记点的盆地就属于这个标记点了，然后就把相应的标记区域从图像中分割出来。

更具体地说，分水岭算法允许使用者（或其他算法）标记已知是对象或背景一部分的对象或背景部分。或者，调用者可以画一条或几条简单的线，这些线条有效地告诉分水岭算法“将这些点组合在一起”。分水岭算法然后通过让标记区域“获取”梯度图中与片段连接的边界确定的峡谷来分割图像。图12-12说明了这一过程。

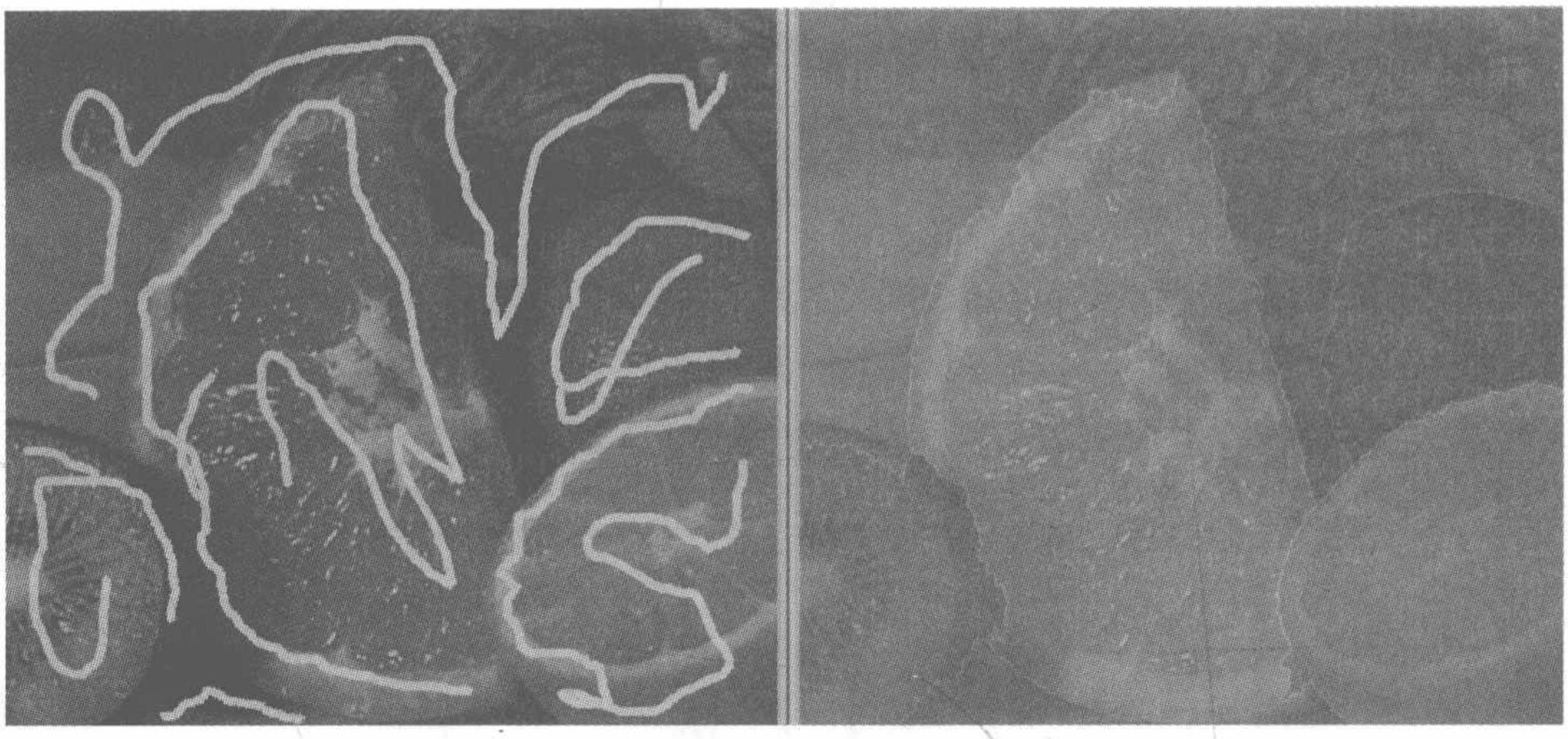


图12-12. 分水岭算法：用户标记共同区域后（左图），算法合并这些区域的结果（右图）

分水岭算法的函数声明如下：

```
void cv::watershed(
  cv::InputArray       image,                // Input 8-bit, three channels
  cv::InputOutputArray markers               // 32-bit float, single channel
);
```

这里，image必须是8位三通道（彩色）图，同时markers是单通道整型（CV::S32）图像。输入中，markers的值为0，除非调用者用正数指定了某些区域是连通的。比如，图12-12的左边，橘子用一个“1”标记，柠檬用一个“2”标记，青柠用“3”标记，大的北京背景区域用“4”标记，以此类推。

算法执行至此后，除了区域间的边界像素设为-1，markers中所有0值像素都会被设置为给定标记中的一个（即所有的橘子区域变成“1”，柠檬区域变成“2”，等等）。图12-12（右）显示了这样分割的一个例子。



所有的区域被标记为-1的像素分割，这是非常理想的。不过，事实上并非如此。值得注意的是，如果两个相邻像素输入时就是非零但两个区别明显的值，那么它们仍会保留，不会被-1分隔开。

Grabcuts算法

由Rother，Kolmogorov和Blake [Rother04]引入的Grabcuts算法扩展了Graphcuts算法[Boykov01]，用于用户导向的图像分割。Grabcuts算法可以得到良好的分割效果，通常不超过待分割的前景对象周围的边界矩形。

原始Graphcuts算法使用用户标记的前景和用户标记的背景区域来为两类图像区域建立分布直方图。然后将未标记的前景和符合类似分布的背景主张组合在一起，认为它们之间很平滑并且是连通的（即一堆斑点）。这些主张组成一个能量函数，它给出了一个符合这些主张的低能量（即开销）的解决方案，同时给出一个不符合这些主张的高能量解决方案。[注22]

Grabcuts算法以几种重要方法扩展Graphcuts，第一个是它用不同的模型（高斯混合）替代了原来的直方图模型，使得算法可以用于彩色图像。另外，它通过迭代方法解决了能量函数最小化的问题，这提供了一个更高的全局结果，并且赋予了用户提供的标记更大的灵活性。值得一提的是，后面这一点使得单面标签也是可行的，就是说只标记背景或者前景中的一个（Graphcuts需要两者）。

OpenCV的实现允许调用者只在待分割对象周围提供一个矩形框，矩形框的边界下方的像素（即它的外部）属于背景，并且没有指定前景。或者，调用者可以指定一个全局掩膜，将像素分类为前景、背景、疑似前景或疑似背景。[注23]在这种情况中，确定性区域将用于分类其他区域，算法将把后者分类成确定性的。

OpenCV中cv::Grabcuts()实现了Grabcuts算法：

注22：该算法通过最小化该能量函数获得最终结果。这个最小化是一个非常重要的问题。在实践中，它通过一种称为Mincut的技术来执行，这也是Graphcuts和Grabcuts算法之所以这样命名的原因。

注23：也许无意中，实现的算法不允许标记“不知道”，只能标记疑似信息。

```
void cv::grabCut(
  cv::InputArray        img,
  cv::InputOutputArray mask,
  cv::Rect              rect,
  cv::InputOutputArray bgdModel,
  cv::InputOutputArray fgdModel,
  int                   iterCount,
  int                   mode     = cv::GC_EVAL
);
```

给定一幅输入图像img，cv::grabCut()就会计算得到的标签并把它保存在输出数组mask里。mask数组也可以当作输入，用变量mode说明。如果mode包含标签cv::GC_INIT_WITH_MASK，[注24]那么在mask里的值将被用于初始化图像的标签。掩膜应当是单通道的cv::U8类型的图像，掩膜里的每个像素都应按下表赋值。

枚举值	数值	含义
cv::GC_BGD	0	确定性背景
cv::GC_FGD	1	确定性前景
cv::PR_GC_BGD	2	疑似背景
cv::PR_GC_FGD	3	疑似前景

参数rect只适用于你不用掩膜初始化的时候。当模式包含标志cv::GC_INIT_WITH_RECT时，矩形区域之外的整个区域就被当作是“确定性背景”，而剩下的区域则默认为“疑似前景”。

后面两个数组本质上是临时缓冲区。当你第一次调用cv::grabCut()时，它们可以是空数组，但如果需要迭代多次并且中间需要重启算法（可能在允许用户提供额外的“确定”像素来指导算法之后），你就需要传入由上一次运行所填充的相同（未修改）缓冲区（除了使用从上一次运行中返回的掩码作为下一次运行的输入）。

Grabcuts算法需要在内部运行几次Graphcuts算法（之前提到的次要扩展）。在每个这样的运行之间，都要重新计算混合模型。参数itercount一般是10或12，但其大小可能取决于图像的大小和性质。

注24：实际上，cv::grabCut()中已经实现了，不需要显式提供cv::GC_INIT_WITH_MASK标志，因为掩膜初始化实际上是默认行为。如此一来，只要你不提供cv::GC_INIT_WITH_RECT标志，就会进行掩膜初始化。但是，这不是作为默认参数实现的，而是函数的过程逻辑中的默认值，因此不能保证在库的将来版本中这一点会保持不变。所以最好明确使用cv::GC_INIT_WITH_MASK标志或cv::GC_INIT_WITH_RECT标志，这样做不仅能为以后增加一个校对标识，也能在平时增强代码的清晰度。

Mean-Shift分割算法

Mean-Shift分割算法找到空间上颜色分布的峰值[Comaniciu99]，它与mean-shift算法有关，后面第17章中讨论追踪和运动的时候会讨论到它。两者主要的差别是前者关注颜色的空间分布（并且因此与我们当前的分割主题相关），而后者在连续帧中通过时间追踪这些分布。基于颜色分布峰值进行这种分割的函数是cv::pyrMeanShiftFiltering()。

给定一个维度为(*x*, *y*, blue, green, red)的多维数据点集，mean-shift可以通过一个滑动窗口在空间上找到该空间中数据的最高密度“块”。但是请注意，空间变量(*x*, *y*)的范围可以和彩色分量范围（蓝、绿、红）非常不同。在这种情况下，我们应该至少具有空间变量（spatialRadius）的一个半径和颜色幅度（colorRadius）的一个半径。随着mean-shift窗口的移动，在数据中收敛于峰值的窗口遍历的所有点都连接或“拥有”该峰值。这种所有权，从最密集的峰值部分辐射出来，形成了图像的分割。分割实际上是通过尺度金字塔cv::pyrUp()和cv::pyrDown()完成的（第11章中已经介绍过），因此金字塔中强度较高的彩色聚类（缩放图像）就会在较低层的金字塔修正边界。

Mean-Shift分割算法的输出是一个“色调分离”的新图像，意味着去除了精细纹理、颜色梯度大部分变得平坦了。然后，可以使用适合自己需要的任何算法，例如，如果实际上是最终需要轮廓分割，就可以用cv::Canny()结合cv::findContours()进一步细分此类图像。

cv::pyrMeanShiftFiltering()函数原型如下：

```
void cv::pyrMeanShiftFiltering(
  cv::InputArray   src,                          // 8-bit, Nc=3 image
  cv::OutputArray  dst,                          // 8-bit, Nc=3, same size as src
  cv::double       sp,                           // Spatial window radius
  cv::double       sr,                           // Color window radius
  int              maxLevel = 1,                 // Max pyramid level
  cv::TermCriteria termcrit = TermCriteria(
    cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS,
    5,
    1
  )
);
```

在cv::pyrMeanShiftFiltering()中，我们有一个输入图像src和输出图像dst。两个都必须是8位三通道彩色图像，并且宽高相同。spatialRadius和colorRadius定义了mean-shift算法如何平均颜色和空间在一起进行分割。对于640×480的彩色图像，spatialRadius为2，colorRadius为40时效果较好。下一个参数是max_level，它描述了用于分割的金字塔缩放的层数。max_level为2或3时对640×480的彩色图像效果良好。

最后一个参数是cv::TermCriteria，我们在以前的一些算法中见到过。cv::TermCriteria

用于OpenCV中所有的迭代算法。如果只想将此参数留空，则mean-shift分割函数有理想的默认值。

图12-13展示了Mean-Shift分割算法用以下参数的结果：

```
cv::pyrMeanShiftFiltering( src, dst, 20, 40, 2);
```

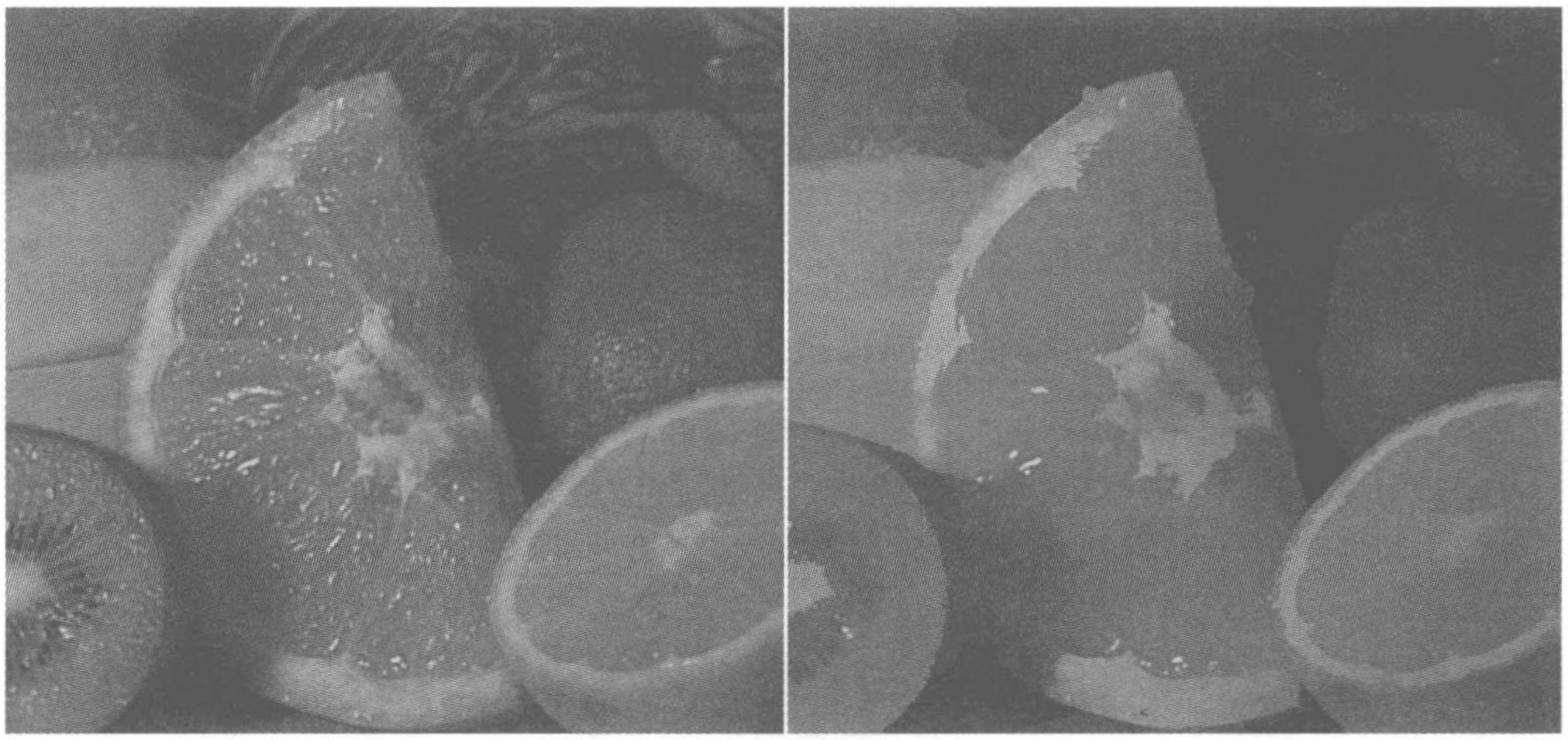


图12-13. 使用参数max_level=2，spatialRadius=20和colorRadius=40的cv::pyrMeanShiftFiltering()进行mean-shift分割；类似的区域现在有相似的值，因此可以视为超像素（较大的统计学意义上相似的区域），这可以显著加速后续处理

小结

在本章中，我们扩展了图像分析技术的所有内容。基于第11章的一般图像分析知识，我们学习了新的方法来进一步理解我们正在操作的图像，我们将看到，这将形成更复杂的算法之基础。距离变换、积分图像和分割技术等工具将成为OpenCV中其他算法以及你自己在图像分析时的重要组成部分。

练习

1. 这个练习中，我们通过在cv::Canny()中设置好的lowThresh和highThresh值来学习参数实验。使用程序读取一幅趣味适中的线结构的图像，我们将使用1.5：1、2.75：1和4：1三种不同高：低的阈值设置。

   a. 高值低于50的时候，你看到了什么？

   b. 高值介于50到100之间的时候，你看到了什么？

   c. 高值介于100到150之间的时候，你看到了什么？

d. 高值介于150到200之间的时候，你看到了什么？

e. 高值介于200到250之间的时候，你看到了什么？

f. 总结结果，并尽量解释发生了什么。

2. 读取一幅包含清晰线条和圆形的图像，比如自行车的侧视图。使用Hough线变换和Hough圆变换，查看图像发生了什么变化。

3. 你能想到一种使用霍夫变换来识别具有不同外围的任何形状的方法吗？解释一下具体怎么做。

4. 取一个小的高斯分布傅里叶变换和图像的傅里叶变换，乘以它们并对结果进行傅里叶逆变换，你得到了什么？随着滤波器越来越大，你会发现在傅里叶空间中滤波比几何空间快得多。

5. 读取一幅图像，转换成灰度图，然后求积分图。用积分图的特性求原图中的竖直和水平边缘。



用很长的矩形，减去然后加上那个位置。

6. 写一个函数来计算旋转45度的积分图像，然后，你可以使用该图像从四个点找到45度旋转矩形的和。

7. 说明当尺度已知并保持固定时，如何使用距离变换自动将已知形状与测试形状对齐。在多尺度的情况下如何进行？

8. 编写函数使用大小为（50, 50）的核调用cv::GaussianBlur()来模糊图像并计时，然后用的DFT重新进行一次模糊，比比看哪次更快。

9. 编写一个图像处理函数，以交互的方式从图像中移除人，用cv::grabCut()分割，然后用cv::inpaint()填充产生的漏洞（回想一下，我们在第11章中学习了cv::inpaint()）。

10. 拍摄一张足够有趣的照片，用cv::pyrMeanShiftFiltering()分割它，用cv::floodFill()掩盖其中两个分割结果，然后用这些掩膜来进行模糊。

11. 创建一个包含20 × 20的正方形的图像，将其旋转到任意角度，对此图像进行距离变换；创建一个20 × 20的正方形，使用算法将距离变换后的正方形图像叠加到自己做的图像中旋转的正方形上。

12. 在2005的DARPA大挑战机器人竞赛中，斯坦福团队的作者用一种颜色聚类算法将道路与非公路部分分开。这些颜色是从汽车前面的激光探测到并用梯形标记出来的道路斑点上采样的。在场景中与这个斑点接近的的其他颜色（其连接组件与原始梯形路连接）被标记为“道路”。参见图12-14，在道路内部使用梯形标记，并在道路外使用倒U形标记之后，使用分水岭算法分割道路。假设我们可以自动生成这些标记。这种分割道路的方法可能存在什么问题？

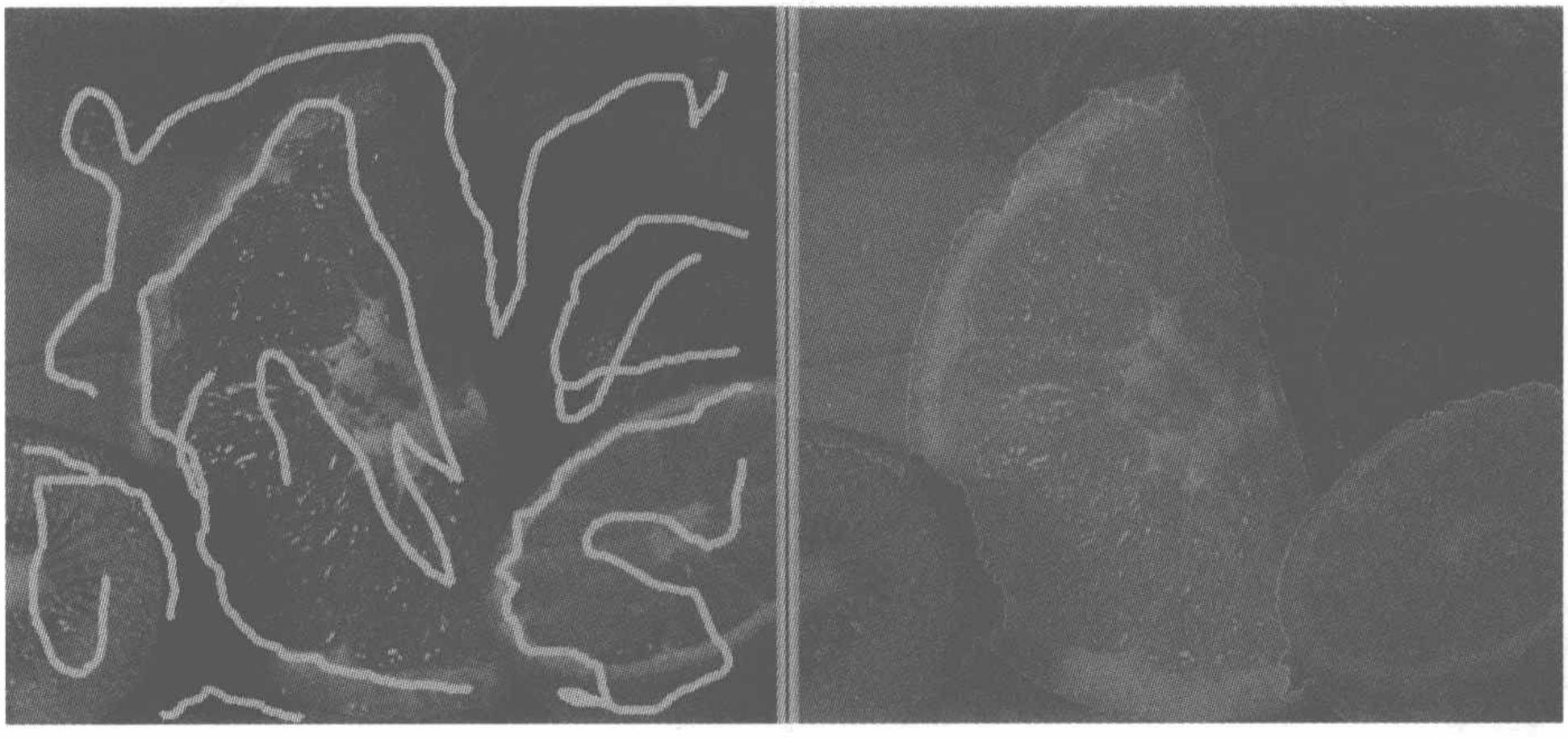


图12-14. 利用分水岭算法识别道路：激光标记的原图（左图）与视觉算法分割道路效果图（右图）

第13章

直方图和模板

在分析图像、物体和视频信息的时候，我们经常用直方图来表达我们关注的信息。直方图可以用来表达很多不同的信息，例如物体的颜色分布，物体的边缘梯度模板[Freeman95]或是以概率分布的形式表达当前对物体位置的估计。图13-1展示了使用直方图来进行进行手势的快速识别。我们从表示“上”“右”“左”“停”和“OK”的手势中提取边缘梯度信息，借助网络摄像头，一个人可以通过上述手势控制网络视频的播放。在每一帧中，首先从视频中检测感兴趣的颜色区域，然后在感兴趣区域周边计算边缘梯度的方向，并使用以梯度方向为组的直方图对它们进行统计，然后将计算所得的直方图和不同手势对应的直方图模型做匹配，进而识别出目标的手势。图13-1中垂直的细长矩形的长度展示了手势的匹配程度。灰色的水平线表示接受某个垂直长矩形作为当前手势的阈值。

直方图在计算机视觉中应用广泛。例如，通过判断帧与帧之间边缘和颜色的统计量是否出现巨大变化，来检测视频中场景的变换。通过使用兴趣点邻域内的特征组成的直方图，来辨识兴趣点。若将边缘、颜色、角点等等的直方图作为特征，可以使用分类器来进行目标识别。提取视频中的颜色或边缘直方图序列，可以用来判断视频是否拷贝自网络。这样的应用数不胜数，直方图可以说是计算机视觉领域中的经典工具之一。

直方图只是简单地将数据归入预定义的组，并在每个组内进行计数。也可以选择对数据提取特征，再对特征进行计数，这里的特征可以是梯度的长度、梯度的方向、颜色或者其他任何可以反映数据特点的特征。也就是说，直方图是一种用来揭示数据分布的统计特性的工具。直方图的维度通常比原始数据的维度低，图13-2中给出了一种典型情况。该图给出了一些分布在二维平面上的点（左上图）；我们在平面上引入网格（右上图）并对落入每个网格单元内的点进行计数，从而得到一个一维的直方图（右下）。

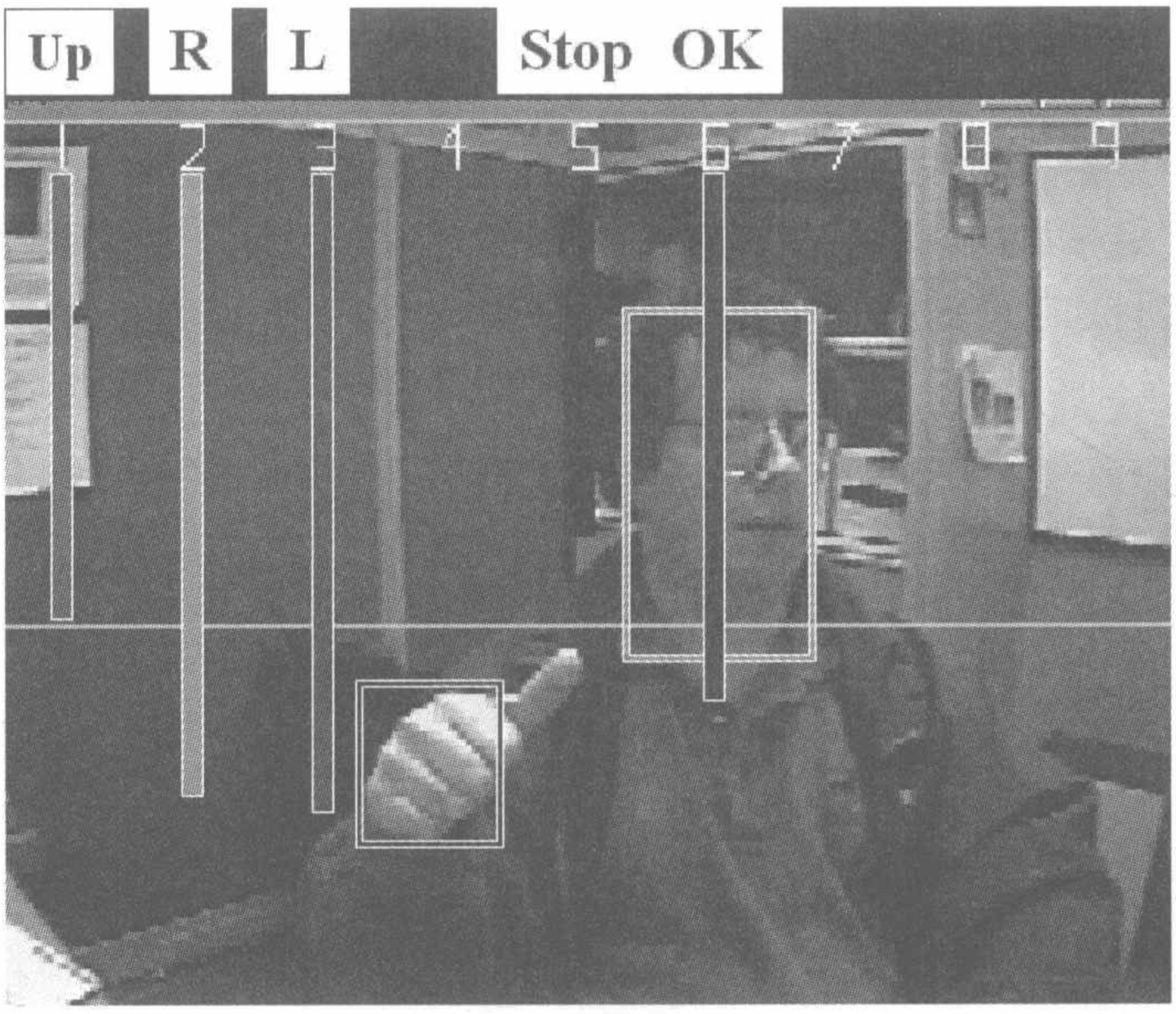


图13-1. 局部梯度方向的直方图被用来寻找手和辨别手势，这里识别出的手势（最长的垂直长矩形）是正确的“左”（向左移动）

原始数据可以是任意东西，因此直方图可以作为一种方便的工具来表达图片中的信息。

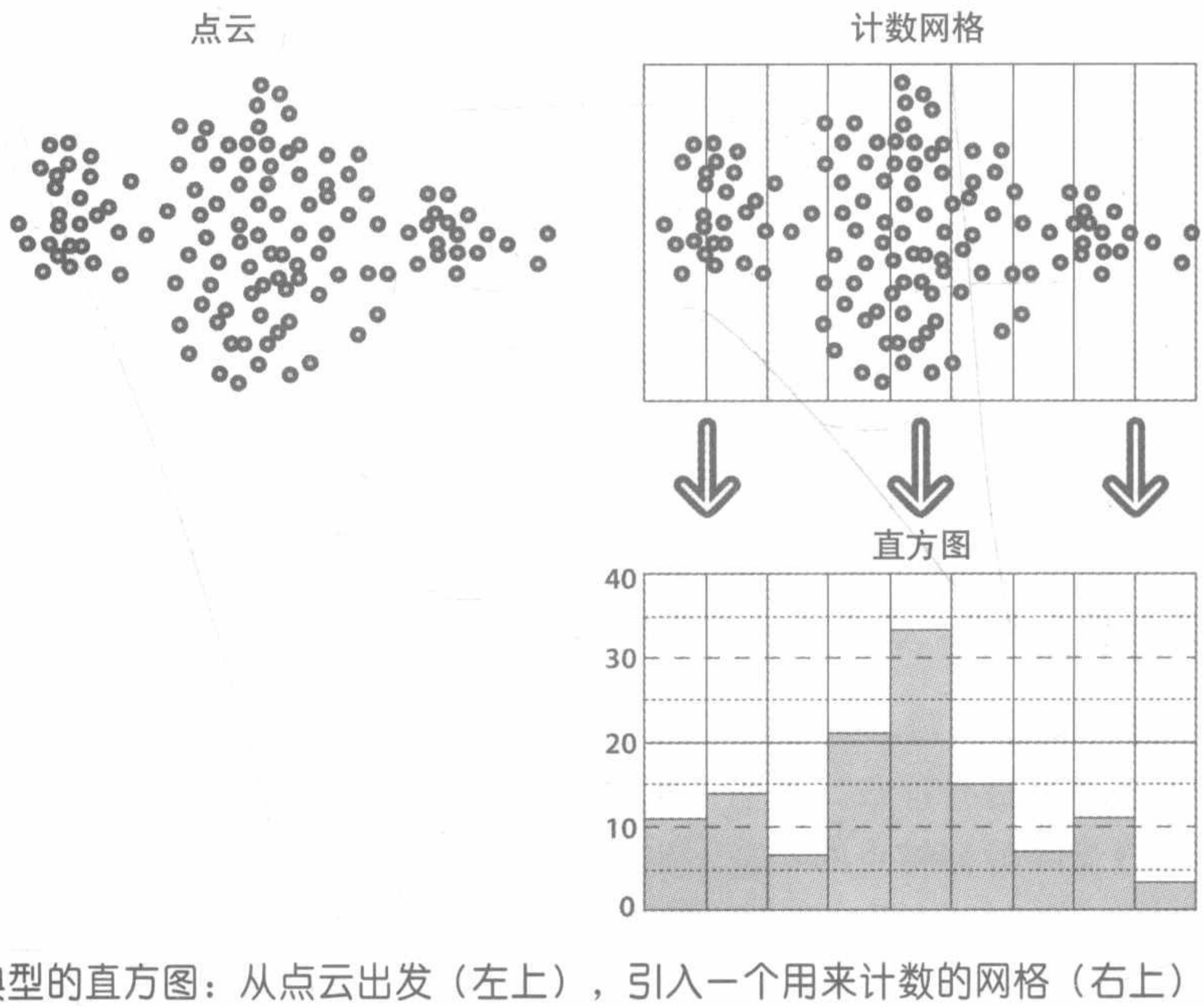


图13-2. 典型的直方图：从点云出发（左上），引入一个用来计数的网格（右上），进而通过对点计数产生一个一维直方图（右下）

如果原始数据是连续分布的，可以通过离散量化的方式来生成直方图[注1]，但这种方法会引发一些问题，正如图13-3所示。如果用来量化的网格选得过宽（左上），量化的结果会过于粗糙，我们会丢失一些数据分布的结构信息。如果网格过窄（右上），每组中没有足够的数据点来准确估计数据分布，直方图里就会出现一些小的、细且尖的矩形。

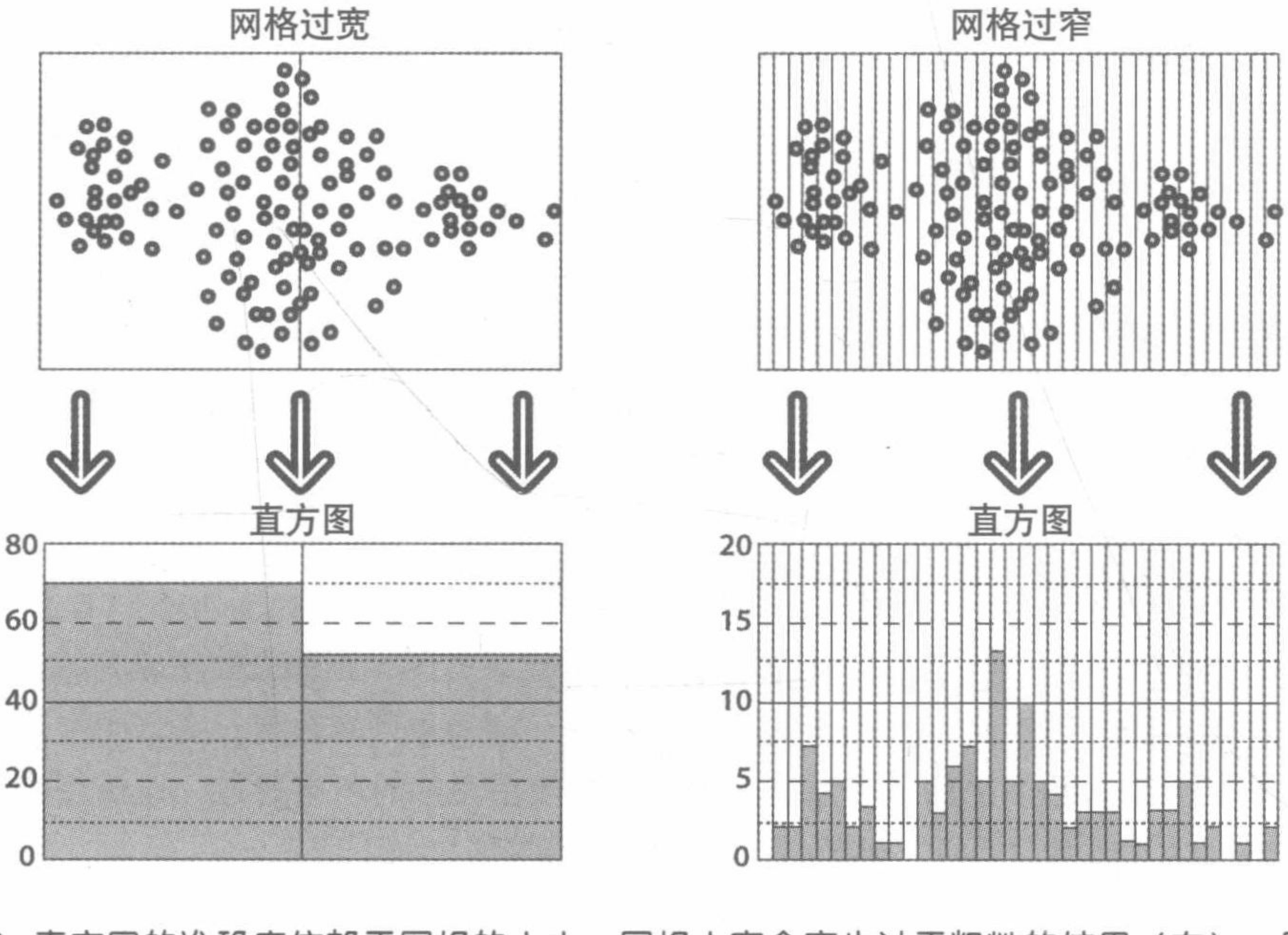


图13-3. 直方图的准确度依赖于网格的大小：网格太宽会产生过于粗糙的结果（左）；网格太窄会由于没有足够样本而产生细而尖的结果

OpenCV提供一种数据类型来表达直方图，这个数据类型可以表达一维至多维的直方图，并包括必要的数据以支持均匀或非均匀的组宽。并且如你所想，它配备各种有用的函数来方便你对直方图进行各种操作。

OpenCV中直方图的表示

OpenCV使用数组来表示直方图，这里的数组和用来表示其他数据的数组没有任何区别。[注2]这意味着对于一维或二维数组，你可以使用cv::Mat（用$N \times 1$或者$1 \times N$来表示一

注1：同样的事也发生在离散数据自然成组而使用的直方图组数少于原始数据自然形成的组数的情况下。例如，使用分成10组的直方图来表达8位亮度值：直方图中的每个组中大约有25种不同的亮度值，它们被（错误的）给予了相同的地位。

注2：其实C++ API相比C API有细微的改动。后者使用一个名为CvHistogram的结构体表示直方图。但出于简洁和统一的考虑，C++接口中删去了这个结构体。

维数组），也可以使用vector<>或稀疏矩阵类型。当我们把一个数组视为直方图时，它就具有和普通数组完全不同的含义。例如，若一个*n*维数组被视为直方图，数组中的每个元素表达的是该*n*维直方图中对应组中的计数结果。而直方图中组的编号就是数组某维与其该组对应的元素的下标，它们通常都是整数。但直方图中组的实际含义不同于该组的编号，要注意区分。当你使用直方图时，可能需要在组的实际含义和组的编号之间转换。例如我们有一个表达人群体重的直方图，其中的组可能被设计为20～40，40～60，60～80和80～100千克，而这时这些组的编号仍然是0，1，2和3。很多OpenCV中的函数会帮你完成（或是一部分）这种转换的工作。

在处理一个高维直方图时，经常会出现直方图中大部分元素都是0的情况，这时可以选择使用cv::SparseMat类来表示直方图。事实上，cv::SpareseMat类存在的主要原因正是用来表示直方图的。大多数在稠密数组上有效的基本函数在稀疏数组上同样有效，但在下一节我们会指出几个重要的反例。

cv::calcHist()：从数据创建直方图

函数cv::calcHist()可以从一个或多个数组中创建直方图。正如先前所提到的，直方图的维度和输入数组的维度或大小无关，而是取决于输入数组的数量。直方图中的每个维度对应于某个输入数组上的某个通道。若是不想使用图片的所有通道，选择想使用的通道传入cv::calcHist()即可。函数cv::calcHist()的接口如下所示：

```
void cv::calcHist(
  const cv::Mat*  images,            // C-style array of images, 8U or 32F
  int             nimages,           // number of images in 'images' array
  const int*      channels,          // C-style list of int's, lists channels
  cv::InputArray  mask,              // in 'images' count, iff 'mask' nonzero
  cv::OutputArray hist,              // output histogram array
  int             dims,              // hist dimensionality < cv::MAX_DIMS (32)
  const int*      histSize,          // C-style array, hist sizes in each dim
  const float**   ranges,            // C-style array, 'dims' pairs set bin sizes
  bool            uniform    = true, // true for uniform binning
  bool            accumulate = false // true, add to 'hist' else replace
);

void cv::calcHist(
  const cv::Mat*  images,            // C-style array of images, 8U or 32F
  int             nimages,           // number of images in 'images' array
  const int*      channels,          // C-style list of int's, lists channels
  cv::InputArray  mask,              // in 'images' count, iff 'mask' nonzero
  cv::SparseMat&  hist,              // output histogram (sparse) array
  int             dims,              // hist dimensionality < cv::MAX_DIMS (32)
  const int*      histSize,          // C-style array, hist sizes in each dim
  const float**   ranges,            // C-style array, 'dims' pairs set bin sizes
  bool            uniform    = true, // true for uniform binning
  bool            accumulate = false // if true, add to 'hist', else replace
);
```

cv::calcHist()总共有三种形式，前两种使用“老式的”C风格数组，而第三种使用现在流行的STL vector模板类型的参数。前两种之间的主要区别在于，计算结果是被保存在稠密的还是稀疏的数组中。

第一个参数是数组数据，images要么是一个指向C风格数组列表的指针，要么是一个更现代化的cv::InputArrayOfArrays。不管是哪种情况，直方图都是从images中一个或更多数组中计算的。尽管images中数组的通道数可以不同，但它们的大小必须是相同的。数组中可以是8位整数或是32位浮点数，但所有数组的数据类型必须统一。如果以C风格作为输入，必须同时指定nimages作为images中包含的数组个数。可以通过参数channels来控制在直方图计算时，哪些通道需要被考虑。同样，channels也可以是C风格数组或是存放整数的STL vector。channels中的整数表示在计算直方图时，使用输入数据中的哪些通道。通道是按顺序进行序列编号的，image[0]中的前$N_c^{(0)}$个通道被编号为$0 \sim N_c^{(0)}$,而接下来的images[1]中的$N_c^{(1)}$个通道被编号为$N_c^{(0)}+N_c^{(1)}-1$，以此类推。显而易见，channels中的元素个数正是计算得出的直方图的维度。

数组mask是一个可选的掩膜，它可以用来选择images中每个数组中哪些像素参与直方图的计算。mask必须是一个8-bit数组并且和images中的输入数组大小相同。计算直方图时会考虑每个在mask中有非零值的像素。如果不想使用掩膜，只需传给它一个cv::noArray()。

hist参数是直方图计算的输出值。dims参数是输出直方图的维度。正如前面提到的，该值恰好也是channels中元素的个数，而每个维度的来源便由channels指定。histSize参数可以是C风格数组或是STL风格整数vector，表示直方图中每个维度需要分为多少个区间。histSize中元素的个数也应该等于dims。

每个维度中的区间的个数由histSize来指定，而这里面每个区间对应的值由ranges来指定。ranges同样可以使用C风格数组或STL整数vector。如果以C风格传参，其每个元素range[i]应该是另一个数组，同时ranges的长度必须等于dims。这时，ranges[i]表示第i维中区间的结构。而ranges[i]的具体解释依赖于另一个参数uniform。如果uniform为true，那么第i维中所有的区间都等长，此时需要指定最矮区间的下界（包含），和最高区间的上界（不包含）（例如，ranges[i]={0,100.0}）。如果uniform的值为false，假如第i维中有N_i个bins，那么ranges[i]中必须有N_i个元素。这时，第j个元素是第j个区间的下界（包含）和第$j-1$个区间的上界（不包含）。在ranges使用vector时，它的含义和使用C风格时相同，只不过将C风格中的内容摊平成一维数组（即在非规范化的情况下时，vector中按照维度的顺序每个维度对应两个元素，在非规范化的按照维度的顺序每个维度对应N_i+1个元素）。图13-4展示了这些情况。

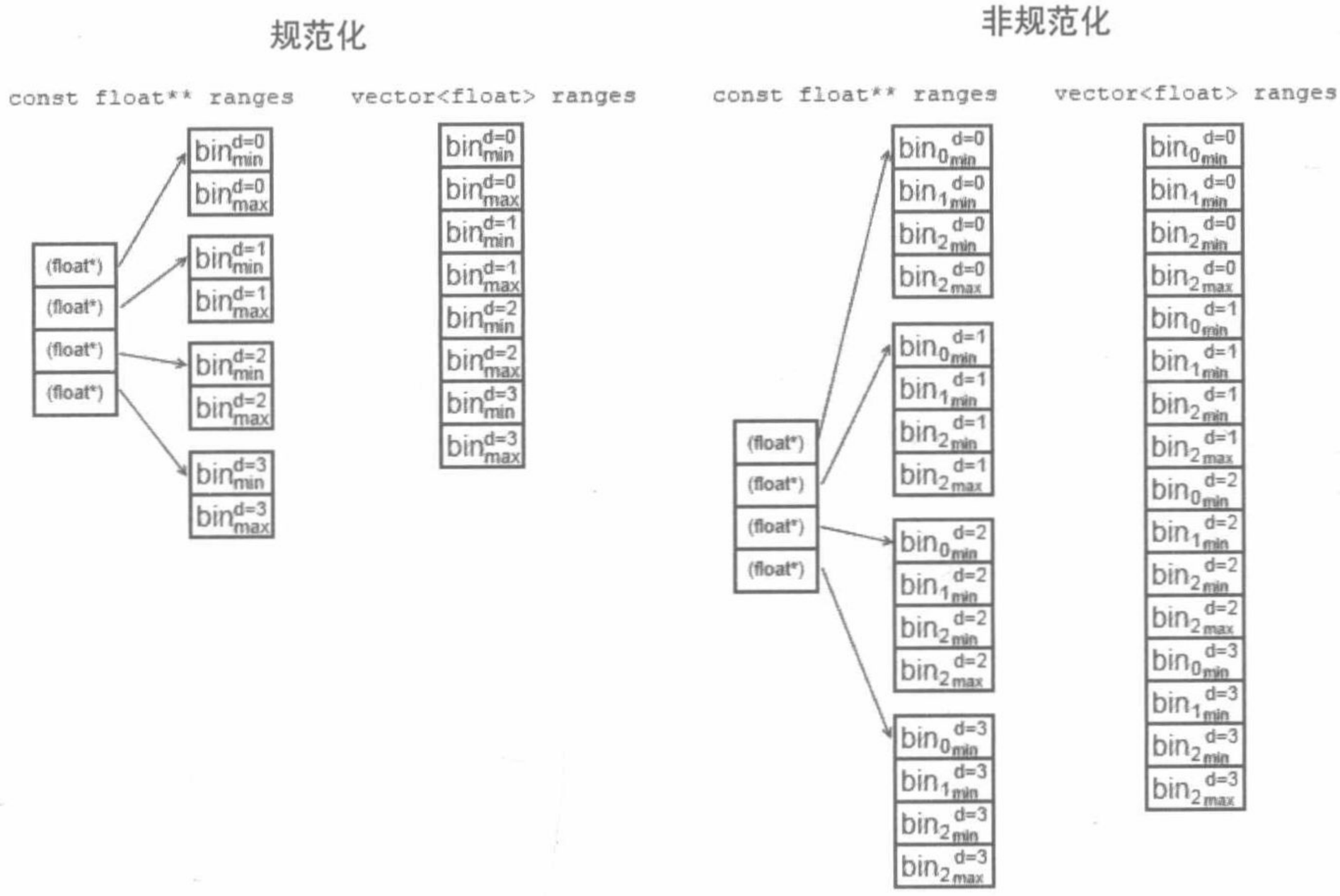


图13-4. ranges参数要么是一个C风格数组的数组，要么是一个STL风格的浮点数vector。当直方图是均匀直方图时，只需提供每个区间的最小边界和最大边界。在其为非均匀直方图时，需要提供每一维度中的每个区间的最小边界和最大边界

最后一个参数accumulate告诉OpenCV在从images得到的数据被累加进入hist之前，不要将其中的元素删除，重新分配或是设置为0。

基本直方图操作

尽管用来表示直方图的数据结构和用来表示矩阵或是图像的数据结构相同，由于对这种数据结构的解释有所不同，所以直方图配有一系列用来完成特定任务的新操作。在这一节中，我会设计一些与直方图相关的简单操作，并且讲解如何用先前讲述的矩阵操作来完成一些重要的直方图操作。

直方图归一化

当构造直方图时，我们首先需要将信息放入在各个区间。然而，一旦完成这些，我们有时还会希望将直方图变为归一化的形式，这时每个区间恰好表示的是该区间内的信息占总体的百分比。在C++ API中，我们可以简单使用数组的代数算子和操作来完成直方图归一化：

```
cv::Mat normalized = my_hist / sum(my_hist)[0];
```

或是：

```
cv::normalize( my_hist, my_hist, 1, 0, NORM_L1 );
```

直方图二值化

另一个很常见的情况是，你希望将一个直方图二值化并（例如）丢弃所有其中元素个数少于给定值的区间。和归一化相同，二值化也可以不用任何特定的直方图方法来完成。你可以使用标准数组二值化函数：

```
cv::threshold(
  my_hist,                    // input histogram
  my_thresholded_hist,        // result, all values<threshold set to zero
  threshold,                  // cutoff value
  0,                          // value does not matter in this case
  cv::THRESH_TOZERO           // threshold type
);
```

找出最显著的区间

有时你希望能找出所有元素个数高于某个给定阈值的区间，有时你只是希望能找出有最多元素的区间。这种情况多发生在使用直方图来表示概率分布的时候。这时你可以选择使用cv::minMaxLoc()。

当处理二维数组时，你可以使用cv::InputArray形式的cv::minMaxLoc()：

```
void cv::minMaxLoc(
  cv::InputArray src,                    // Input array
  double*        minVal,                 // put minimum value (if not NULL)
  double*        maxVal = 0,             // put maximum value (if not NULL)
  cv::Point*     minLoc = 0,             // put minimum location (if not NULL)
  cv::Point*     maxLoc = 0,             // put maximum location (if not NULL)
  cv::InputArray mask   = cv::noArray() // ignore points for which mask is zero
);
```

参数minVal和maxVal需要传入要用来存放计算出来的最小值最大值的变量的指针。同样，minLoc和maxLoc是用来存放计算出的最大值最小值出现位置（在这种情况下类型为：cv::Point）上的变量的指针。如果你不希望计算某个结果，只需向其对应的变量传入NULL：

```
double    max_val;
cv::Point max_pt;

cv::minMaxLoc(
  my_hist,    // input histogram
  NULL,       // don't care about the min value
```

```
  &max_val,   // place to put the maximum value
  NULL,       // don't care about the location of the min value
  &max_pt     // place to put the maximum value location (a cv::Point)
);
```

然而，在这个例子中需要一个二维直方图。[注3]你也可以使用一个稀疏数组型的直方图，注意，到cv::minMaxLoc()有一个专为稀疏数组设计的版本：

```
void cv::minMaxLoc(
  const cv::SparseMat& src,                    // Input (sparse) array
  double*              minVal,                 // put min value (if not NULL)
  double*              maxVal = 0,             // put max value (if not NULL)
  cv::Point*           minLoc = 0,             // put min location (if not NULL)
  cv::Point*           maxLoc = 0,             // put max location (if not NULL)
  cv::InputArray       mask   = cv::noArray() // ignore points if mask is zero
);
```

注意，这种形式的cv::minMaxLoc()实际上和普通形式的有本质上的不同。除了使用稀疏矩阵作为输入，还要求传入的minIdx和maxIdx为int*型，而不是minLoc和maxLoc所采用的cv::Point*型。这是因为稀疏矩阵形式的cv::minMaxLoc()支持任意维的矩阵。因此，需要手动分配存放位置的变量并确保其可存放n维稀疏直方图对应的原图中的一个（n维）点：

```
double maxval;
int   max_pt[CV_MAX_DIM];

cv::minMaxLoc(
  my_hist,                  // input sparse histogram
  NULL,                     // don't care about the min value
  &max_val,                 // place to put the maximum value
  NULL,                     // don't care about the location of the min value
  max_pt                    // place to put the maximum value location (a cv::Point)
);
```

如果只是想找出一个n维非稀疏数组中的最大值或是最小值，则需要使用另一个函数。这个函数和cv::minMaxLoc()的工作方式相同，且有相近的函数名称，但不是同一个函数：

```
void cv::minMaxIdx(
  cv::InputArray src,
  double*        minVal,                  // put min value (if not NULL)
  double*        maxVal = 0,              // put max value (if not NULL)
  int*           minLoc = 0,              // put min location indices (if not NULL)
  int*           maxLoc = 0,              // put max location indices (if not NULL)
  cv::InputArray mask   = cv::noArray() // ignore points if mask is zero
);
```

注3：如果有一个一维vector<>数组，则可以使用cv::Mat(vec).reshape(1)来将其转为一个$N \times 1$的二维数组。

这时，上述函数的参数和cv::minMaxLoc()函数具有相同的含义。你必须手动分配minIdx和maxIdx为正确大小（正如先前提到的）的C风格数组。不过这里要提醒你注意：如果输入的src是一维的，你应该将变量minIdx和maxIdx置为二维的。因为cv::minMaxIdx()在内部将一维数组视为二维数组。因此，如果最大值是一维数组的第k位，maxIdx的返回值对于单列矩阵为（k,0），对于单行矩阵为（0,k）。

比较两个直方图

另一个不可或缺的工具是通过某些特殊的判据对两个直方图的相似度进行度量，这种工具首先由Swain和Ballard[Swain91]引入，后又由Schiele和Crowley[Schiele96]加以推广。函数cv::compareHist()正是为此而来。

```
double cv::compareHist(
  cv::InputArray       H1,       // First histogram to be compared
  cv::InputArray       H2,       // Second histogram to be compared
  int                  method    // comparison method (see options below)
);

double cv::compareHist(
  const cv::SparseMat& H1,       // First histogram to be compared
  const cv::SparseMat& H2,       // Second histogram to be compared
  int                  method    // comparison method (see options below)
);
```

前两个参数是待比较的直方图，它们应该具有相同的大小。第三个参数允许我们选择需要的距离度量。当然了，我们可以通过对两幅图像取直方图，并通过下面给出的直方图比较方法来比较原来的两幅图像。我们也可以借助直方图比较方法来在图像中识别物体，方法是将物体的直方图和图片中不同的子区域的直方图进行对比，通过直方图相配的程度来判断该子区域是否有物体。[注4]

四个可用的方法如下。

相关性方法（cv::COMP_CORRL）

第一种比较方法使用基于统计相关性的方法，其中实现了皮尔逊(Pearson)相关性系数方法。这里H_1和H_2应该解释为概率分布：

$$d_{correl}(H_1, H_2) = \frac{\sum_i H_1'(i) \cdot H_2'(i)}{\sqrt{\sum_i H_1'^2(i) \cdot \sum_i H_2'^2(i)}}$$

注4：在OpenCV 2.4中，有一个函数cv::calcBackProjectPath()用来自动进行这种子区域匹配，但在OpenCV 3.0中，由于速度的原因而移除了这个函数。

其中 $H_k'(i) = H_k(i) - N^{-1}\sum_j h_k(j)$，$N$为直方图中区间的个数。

大相关性系数要比小相关性系数匹配的好。完全匹配的相关性系数为1，而完全不匹配会得到-1，而0对应着两个分布没有关系（随机组合）。

卡方方法（cv::COMP_CHISQR_ALT）

该距离度量是基于卡方（Chi-square）检验[注5]统计量，是一种检验两个分布是否相关的替换检验方法。

$$d_{chi\text{-}square}(H_1, H_2) = \sum_i \frac{(H_1(i) - H_2(i))^2}{H_1(i) + H_2(i)}$$

对于该检验方法，低分值表示直方图匹配得很好。完全匹配的分值为0，而完全不匹配的分值无下限（依赖于直方图的大小）。

交集法（CV::COMP_INTERSECT）

直方图交集法简单基于两个直方图的交集。该方法对两个直方图中的共同部分求和。

$$d_{intersection}(H_1, H_2) = \sum_i \min(H_1(i), H_2(i))$$

对于该度量方法，高分值意味着匹配得比较好。如果两个直方图都进行了归一化，完全匹配对应的是1，完全不匹配对应的是0。

巴氏距离（cv::COMP_BHATTACHARYYA）

最后一个选项称为巴氏距离（Bhattacharyya distance method）[Bhattacharyya43]，也是对两个分布重叠程度的一种度量方法。

$$d_{correl}(H_1, H_2) = \sqrt{1 - \frac{\sum_i H_1(i) \cdot H_2(i)}{\sqrt{\sum_i H_1(i) \sum_i H_2(i)}}}$$

这时，匹配的效果越好，值越小，效果越差，值越大。完全匹配是0，而完全误匹配是1。

cv::COMP_BHATTACHARYYA的代码中有一个特殊的因子来对输入的直方图进行归一化。然而，一般的来说，在对两个直方图进行比较之前应该对其进行归一化，因为类似直方图交集这样的概念只在归一化后有才意义（虽然不归一化也是允许的）。

图13-5给出了一些简单的情况。事实上，它给出的是能想出的最简单的情况：一个一维只有两个区间的直方图。标准模型直方图指的是左侧区间为1.0右侧区间为0.0的直方

注5：　卡方检验是Karl Pearson[Pearson]发明的，同时他还开拓了数理统计领域。

图。后三行给出了与模型比较的三个直方图和各种距离度量方法（EMD度量很快将在后文介绍）下的比较值。

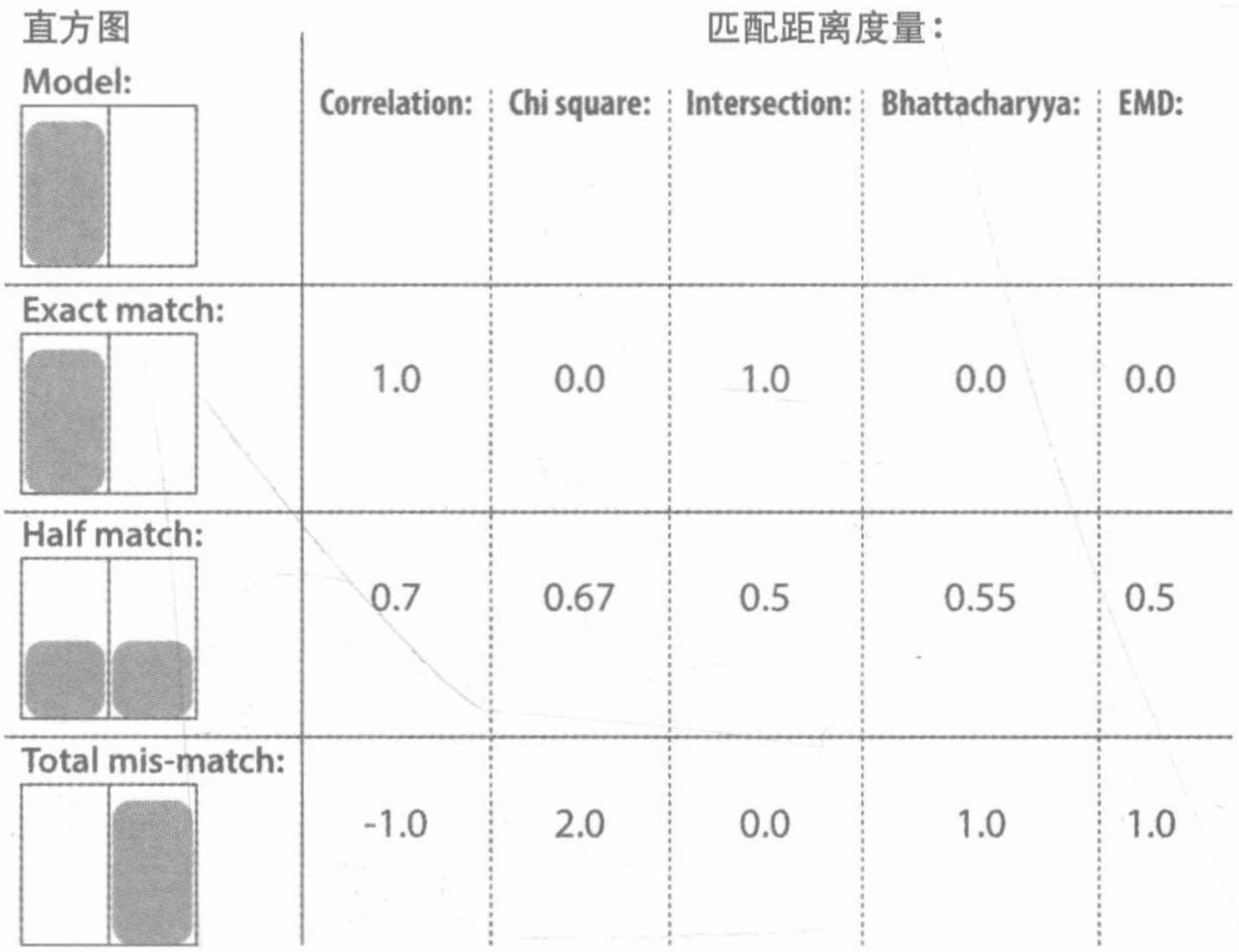


图13-5. 直方图匹配距离度量

图13-5给出了不同匹配方法行为的一个快速参考。仔细观察图中这些匹配方法给出的结果，我们得到一个令人不安的发现：如果直方图的区间仅仅平移了一个槽位，正如表格中第一个和第三个比较的直方图，那么所有的匹配方法（除了EMD）都产生了最大的误匹配值，即使这两个直方图有相同的“形状”。图13-5中给最右侧给出的值来自EMD算法，它也是一种距离度量方法。在将第三个直方图和模型直方图进行对比时，EMD度量准确表示了该情况：第三个直方图是模型向右移动了一个单位。我们会在后面介绍“EMD距离”时进一步讨论这个度量方法。

在作者的经验中，交集法在快速而粗略的匹配中效果很好，而卡方法和Bhattacharyya法效果更好，但速度更慢。EMD给出了最符合直觉的匹配效果，但是计算速度更慢。

直方图用法示例

是时候给出一些有用的示例了。示例13-1中给出的程序（改编自 OpenCV 代码集）展示了如何使用先前给出的函数。在这个程序中，我们先从图片中计算色调（hue）-饱和度（saturation）直方图，然后将这些直方图绘制在一个网格中。

示例 13-1：直方图的计算和绘制

```
#include <opencv2/opencv.hpp>
#include <iostream>
```

```
using namespace std;

int main( int argc, char** argv ){

  if(argc != 2) {
    cout << "Computer Color Histogram\nUsage: " <<argv[0] <<" <imagename>" << endl;
    return -1;
  }

  cv::Mat src = cv::imread( argv[1],1 );
  if( src.empty() ) { cout << "Cannot load " << argv[1] << endl; return -1; }

  // Compute the HSV image, and decompose it into separate planes.
  //
  cv::Mat hsv;
  cv::cvtColor(src, hsv, cv::COLOR_BGR2HSV);

  float h_ranges[]       = {0, 180}; // hue is [0, 180]
  float s_ranges[]       = {0, 256};
  const float* ranges[] = {h_ranges, s_ranges};
  int histSize[]         = {30, 32}, ch[] = {0, 1};

  cv::Mat hist;

  // Compute the histogram
  //
  cv::calcHist(&hsv, 1, ch, cv::noArray(), hist, 2, histSize, ranges, true);
  cv::normalize(hist, hist, 0, 255, cv::NORM_MINMAX);

  int scale = 10;
  cv::Mat hist_img(histSize[0]*scale, histSize[1]*scale, CV_8UC3);

  // Draw our histogram.
  //
  for( int h = 0; h < histSize[0]; h++ ) {
    for( int s = 0; s < histSize[1]; s++ ){
      float hval = hist.at<float>(h, s);
      cv::rectangle(
        hist_img,
        cv::Rect(h*scale,s*scale,scale,scale),
        cv::Scalar::all(hval),
        -1
      );
    }
  }

  cv::imshow("image", src);
  cv::imshow("H-S histogram", hist_img);
  cv::waitKey();

  return 0;

}
```

在这个例子中，我们花了相当的时间来为cv::clacHist()准备参数，这种情况很常见。

在很多实际应用中会考虑和人类肤色相关的颜色直方图。以图13-6为例，它包含了不同光照条件下人手的颜色直方图。左侧一列展示了在室内环境下、室外有阴影环境下和室外阳光下的人手图片。中间一列是手的肤色在蓝绿红（BGR）颜色下的直方图。右侧一列是HSV颜色空间下的直方图，其中竖轴是纯度（Value），半径是饱和度（Saturation），角度是色调（Hue）。注意，室内环境是最黑的，室外阴影环境稍微亮一点，而室外阳光下是最亮的。还要注意到，由于灯光颜色的变化，手的颜色会发生一些变化。

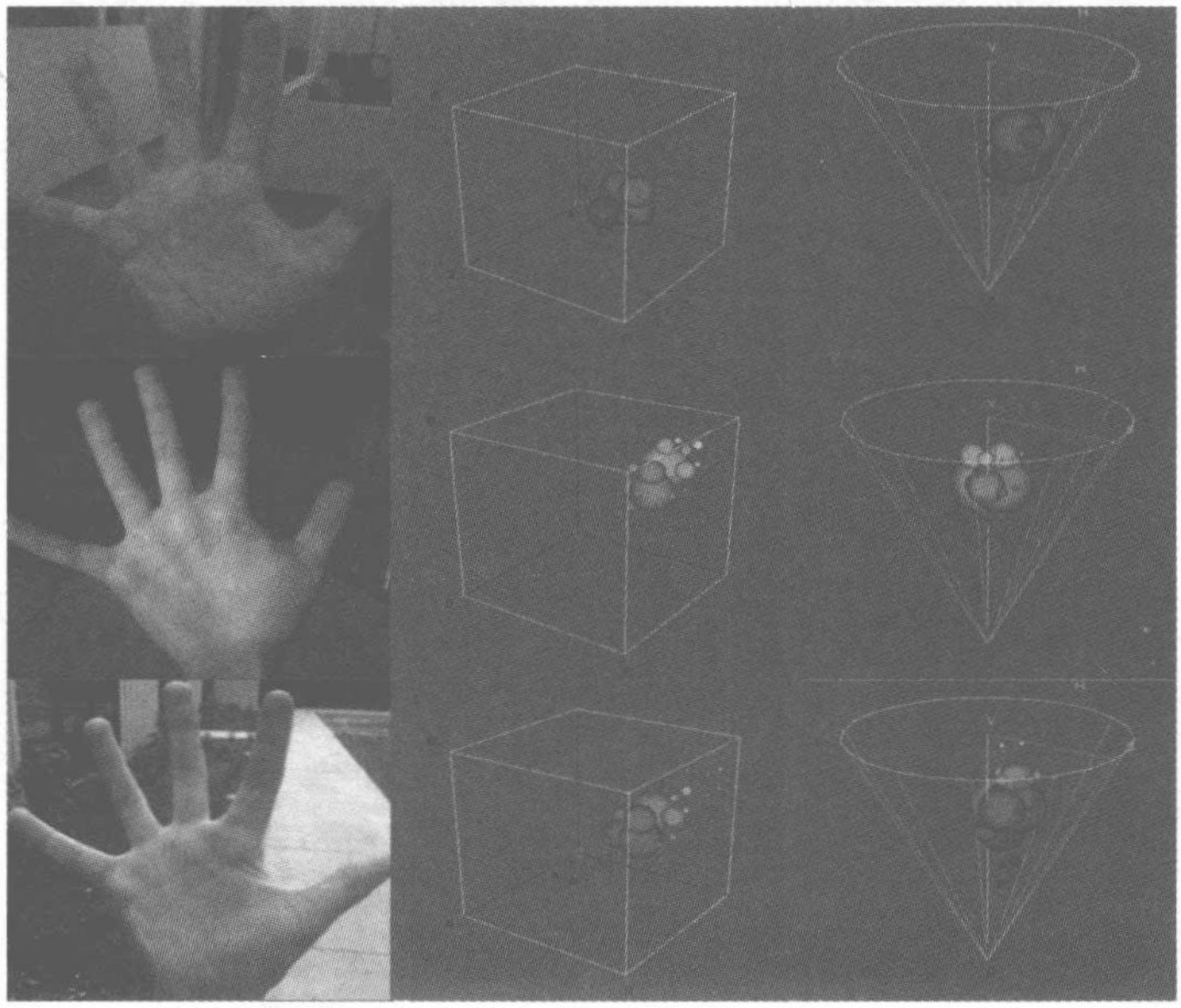


图13-6. 在室内环境下（上左）、室外阴影环境下（中左）和室外阳光直射下（下左）的光照条件下肤色的直方图；中间列和右侧列分别表示BGR和HSV颜色空间下的直方图

为了测试直方图匹配方法，我们从手掌图片（例如室内光照条件下手掌的左上部分）中取出一部分，计算其颜色直方图，并将其与该图片的区域部分的直方图或是其他光照条件下的手掌的直方图进行比较。在颜色空间转为HSV后，很容易选出肤色。在这里，我们只关注色调和饱和度平面，这样做不仅足以进行实验，而且可以跨种族进行肤色识别。

将上述实验付诸实践（示例13-2），我们使用在三种不同光照条件下拍摄的人手相片（图13-6）。首先，我们从最上面（最黑的光照条件）图片中的手上取一小部分作为参考。接下来，我们对该图片的下半面上的手计算直方图，同时也对其余两种光照条件下的（整张）图片的中的手计算直方图。第一张图片拍摄于室内，其余两张拍摄于室外。匹配的结果放在表13-1中。注意，其中一些距离度量在直方图间距离小时给出小值，在

直方图间距离大时给出大值，而其他的距离度量则恰好相反。这和我们对图13-5中展示的距离度量做的简单分析结果很相符。同时注意到，室内环境下，来自下半幅图片的部分皮肤匹配得很好，而在室外环境的光的强度和颜色的影响下，这两部分皮肤匹配得并不好。

表13-1：室内环境下的上半幅图上的手的肤色和列出的各种不同光照下手的肤色的直方图比较，其中使用了四种直方图比较方法。作为参考，完全匹配的分值在第一行给出

比较	CORREL	CHISQR	INTERSECT	BHATTACHARYYA
（完全匹配）	(1.0)	(0.0)	(1.0)	(0.0)
室内下半幅图	0.96	0.14	0.82	0.2
室外阴影	0.09	1.57	0.13	0.8
室外阳光	0.0	1.98	0.01	0.99

一些复杂的直方图方法

我们目前所讨论的都比较基本。每个函数都带有明显的目的性。总的来说，它们为在计算机视觉中（或是其他领域）使用直方图这个工具奠定了良好的基础。在这里，我们将给出一些在 OpenCV 中可用的更复杂的方法，它们在特定的应用中十分有用。我们会介绍一个用来比较两个直方图的高级方法以及计算和/或可视化图像的哪些部分对直方图给定部分所做贡献大小的工具。

EMD距离

先前，我们发现光照的变化会使颜色值产生大量的偏移（图13-6），虽然这种偏移倾向于并不改变颜色直方图的形状，但目前我们见到的距离度量方法都对直方图颜色位置的移动束手无策。核心的困难是对于两个形状相同、但只是相对平移的两个直方图，距离度量会给出一个很大的值。我们希望能找到一个对这种平移不敏感的距离度量方法。EMD（Earth mover's distance）[Rubner00]就是这样一种距离度量。它的基本思路是，通过将一部分（或全部）直方图搬到一个新位置，度量花多大的功夫才能把一个直方图“搬到”另一个直方图里。EMD距离可以在任意维度下工作。

再来看一下图13-5。我们可以从最右一列中看出EMD距离的“搬运”本质。精确匹配的两个量会得到0距离，而半匹配是将半个直方图移动到下一槽，需要消耗半铲子。最后，将整个直方图向右移动一步会消耗一个单位距离（就是说将模型直方图搬运为“完全不匹配”的直方图）。

EMD算法本身是一个很广义的算法，它允许用户设置自己的距离度量或是自己的移动花

费矩阵。即记下“材料”如何从一个直方图流向另一个的，并用从数据的先验信息导出的非线性距离对其度量。OpenCV中的EMD函数为cv::EMD()：

```
float cv::EMD(
  cv::InputArray  signature1,              // sz1-by-(dims+1) float array
  cv::InputArray  signature2,              // sz2-by-(dims+1) float array
  int             distType,                // distance type (e.g., 'cv::DIST_L1')
  cv::InputArray  cost       = noArray(), // sz1-by-sz2 array (if cv::DIST_USER)
  float*          lowerBound = 0,         // input/output low bound on distance
  cv::OutputArray flow       = noArray()  // output, sz1-by-sz2
);
```

虽然我们正在直方图上使用EMD方法，但该方法的接口要求我们使用一种称为签名（signatures）的方式传入前两个数组参数。签名要求是float类型的数组，每行包括直方图区间的计数值，接下来是该区间的坐标。对于图13-5中的一维直方图，左一列直方图（跳过标准模型）的签名（以数组中行的方式列出）为：上，[[1,0],[0,1]]；中，[[0.5,0],[0.5,1]]；下，[0,0],[1,1]]。如果我们在某个三维直方图中的(x,y,z)位置上有区间计数537，那么该区间的一行签名为[537,7,43,11]。一般来说，在调用cv::EMD()之前必要的一步就是将直方图转为签名的形式（在示例13-2中，我们会涉及更多的细节）。

参数distType应该是以下之一：曼哈顿距离（cv::DIST_L1）、欧几里得距离（cv::DIST_L2）、棋盘距离（cv::DIST_C）或用户自定义的距离（cv::DIST_USER）。当使用用户定义的距离度量时，用户通过cost参数传进一个（预计算好的）费用矩阵（这时费用矩阵是一个$n_1 \times n_2$的矩阵，n_1和n_2是signature1和signature2的大小）。

参数lowerBound有两个功能（一个是作为输入，另一个是作为输出）。作为返回值时，它是两个直方图重心距离的下界。为了计算这个下界，必须提供一个标准的距离度量方法（不可以是cv::DIST_USER），同时两个签名的总权重必须相同（正如直方图归一化后的情况）。如果选择提供一个下界，你必须将该值初始化为一个有意义的值。低于这个下界的，EMD距离才会被计算[注6]。例如，如果你想无论何种情况都计算EMD距离，将lowerBound初始化为0即可。

下一个参数flow是一个可选的$n_1 \times n_2$矩阵，用来记录从signature1的第i个点流向signature2的第j个点的质量。本质上，这给出的是在计算整个EMD中质量的具体安排情况。

注6：这样的特性很重要。因为通常对直方图计算这个下界要比计算EMD快许多。在很多实际情况下，如果EMD高于某些值，你可能并不关心它到底是多少，仅仅是觉得它“太大了”（例如你想比较的实物“不相似”）。这种情况之下，一旦EMD大到你不在乎它的具体值，就立刻停止，这样的机制很有用处。

举一个例子，我们有两个直方图hist1和hist2，我们希望将转化为两个签名，sig1和sig2。为了使它看起来更困难一点。假定它们是二维的直方图（正如之前例子中的代码），维度为h_bins*s_bins。示例13-2展示了如何将这两个直方图转化为签名。

示例13-2：从直方图出发为EMD算法构造签名。注意，代码是表13-1中数据的数据源，是在不同的光照条件下对手部的直方图进行对比

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;

void help( char** argv ){
  cout << "\nCall is:\n"
  << argv[0] <<" modelImage0 testImage1 testImage2 badImage3\n\n"
  << "for example: "
  << "  ./ch7_ex7_3_expanded HandIndoorColor.jpg HandOutdoorColor.jpg "
  << "HandOutdoorSunColor.jpg fruits.jpg\n"
  << "\n";
}

// Compare 3 images' histograms
int main( int argc, char** argv ) {

  if( argc != 5 ) { help( argv ); return -1; }

vector<cv::Mat> src(5);
cv::Mat         tmp;
int             i;

tmp = cv::imread( argv[1],  1);
if( tmp.empty() ) {
  cerr << "Error on reading image 1," << argv[1] << "\n" << endl;
  help();
  return(-1);
}

// Parse the first image into two image halves divided halfway on y
//
cv::Size size  = tmp.size();
int width      = size.width;
int height     = size.height;
int halfheight = height >> 1;

cout <<"Getting size [["   <<tmp.cols <<"] [" <<tmp.rows <<"]]\n" <<endl;
cout <<"Got size (w,h): (" <<size.width <<"," <<size.height <<")" <<endl;

src[0] = cv::Mat(cv::Size(width,halfheight), CV_8UC3);
src[1] = cv::Mat(cv::Size(width,halfheight), CV_8UC3);

// Divide the first image into top and bottom halves into src[0] and src[1]
//
cv::Mat_<cv::Vec3b>::iterator tmpit = tmp.begin<cv::Vec3b>();
```

```
// top half
//
cv::Mat_<cv::Vec3b>::iterator s0it = src[0].begin<cv::Vec3b>();
for(i = 0; i < width*halfheight; ++i, ++tmpit, ++s0it) *s0it = *tmpit;

// Bottom half
//
cv::Mat_<cv::Vec3b>::iterator s1it = src[1].begin<cv::Vec3b>();
for(i = 0; i < width*halfheight; ++i, ++tmpit, ++s1it) *s1it = *tmpit;

// Load the other three images
//
for(i = 2; i<5; ++i){
  src[i] = cv::imread(argv[i], 1);
  if(src[i].empty()) {
    cerr << "Error on reading image " << i << ": " << argv[i] << "\n" << endl;
    help();
    return(-1);
  }
}

// Compute the HSV image, and decompose it into separate planes.
//
vector<cv::Mat> hsv(5), hist(5), hist_img(5);
int          h_bins      = 8;
int          s_bins      = 8;
int          hist_size[] = { h_bins, s_bins }, ch[] = {0, 1};
float        h_ranges[]  = { 0, 180 };                    // hue range is [0,180]
float        s_ranges[]  = { 0, 255 };
const float* ranges[]    = { h_ranges, s_ranges };
int          scale       = 10;

for(i = 0; i<5; ++i) {
  cv::cvtColor( src[i], hsv[i], cv::BGR2HSV );
  cv::calcHist( &hsv[i], 1, ch, noArray(), hist[i], 2, hist_size, ranges, true );
  cv::normalize( hist[i], hist[i], 0, 255, cv::NORM_MINMAX );
  hist_img[i] = cv::Mat::zeros( hist_size[0]*scale, hist_size[1]*scale, CV_8UC3 );

  // Draw our histogram For the 5 images
  //
  for( int h = 0; h < hist_size[0]; h++ )
    for( int s = 0; s < hist_size[1]; s++ ){
      float hval = hist[i].at<float>(h, s);
      cv::rectangle(
        hist_img[i],
        cv::Rect(h*scale, s*scale, scale, scale),
        cv::Scalar::all(hval),
        -1
      );
    }
}

// Display
//
```

```
cv::namedWindow( "Source0", 1 );cv::imshow( "Source0", src[0] );
cv::namedWindow( "HS Histogram0", 1 );cv::imshow( "HS Histogram0", hist_img[0] );

cv::namedWindow( "Source1", 1 );cv::imshow( "Source1", src[1] );
cv::namedWindow( "HS Histogram1", 1 ); cv::imshow( "HS Histogram1", hist_img[1] );

cv::namedWindow( "Source2", 1 ); cv::imshow( "Source2", src[2] );
cv::namedWindow( "HS Histogram2", 1 ); cv::imshow( "HS Histogram2", hist_img[2] );

cv::namedWindow( "Source3", 1 ); cv::imshow( "Source3", src[3] );
cv::namedWindow( "HS Histogram3", 1 ); cv::imshow( "HS Histogram3", hist_img[3] );

cv::namedWindow( "Source4", 1 ); cv::imshow( "Source4", src[4] );
cv::namedWindow( "HS Histogram4", 1 ); cv::imshow( "HS Histogram4", hist_img[4] );

// Compare the histogram src0 vs 1, vs 2, vs 3, vs 4
cout << "Comparison:\n"
  << "Corr                 Chi                  Intersect          Bhat\n"
  << endl;

for(i=1; i<5; ++i) {  // For each histogram
  cout << "Hist[0] vs Hist[" << i << "]: " << endl;;
  for(int j=0; j<4; ++j) { // For each comparison type
    cout << "method[" << j << "]: " << cv::compareHist(hist[0],hist[i],j) << "  ";
  }
  cout << endl;
}

//Do EMD and report
//
vector<cv::Mat> sig(5);
cout << "\nEMD: " << endl;

// Oi Vey, parse histograms to earth movers signatures
//
for( i=0; i<5; ++i) {

  vector<cv::Vec3f> sigv;

  // (re)normalize histogram to make the bin weights sum to 1.
  //
  cv::normalize(hist[i], hist[i], 1, 0, cv::NORM_L1);
  for( int h = 0; h < h_bins; h++ )
    for( int s = 0; s < s_bins; s++ ) {
      float bin_val = hist[i].at<float>(h, s);
      if( bin_val != 0 )
        sigv.push_back( cv::Vec3f(bin_val, (float)h, (float)s));
    }

// make Nx3 32fC1 matrix, where N is the number of nonzero histogram bins
//
sig[i] = cv::Mat(sigv).clone().reshape(1);
if( i > 0 )
  cout << "Hist[0] vs Hist[" << i << "]: "
```

```
            << EMD(sig[0], sig[i], cv::DIST_L2) << endl;
  }

  cv::waitKey(0);

}
```

反向投影

反向投影（Back Projection）是计算像素和直方图模型中像素吻合度的一种方法。例如，如果我们有肤色的直方图，我们可以使用反向投影来在图像中寻找肤色区域。用来做这种寻找操作的函数有两种形式，一个是为稠密数组而设计的，另一个为稀疏数组设计。

基础的反向投影：cv::calcBackProject()

和cv::clacHist()类似，反向投影从输入图像的指定通道中计算出一个向量，但不同于前者在向直方图中记录累计值，反向投影从输入的直方图中读取当前像素对应的计数值作为结果。从统计学的视角来看，如果将输入的直方图视为某个物体上特定的向量（颜色）的一个（先验）概率分布，那么反向投影就是计算图片上某个特定部分来自该先验分布（即属于物体的一部分）的概率。

```
  void cv::calcBackProject(
    const cv::Mat*  images,          // C-style array of images, 8U or 32F
    int             nimages,         // number of images in 'images' array
    const int*      channels,        // C-style list, ints identifying channels
    cv::InputArray  hist,            // input histogram array
    cv::OutputArray backProject,     // output single channel array
    const float**   ranges,          // C-style array, 'dims' pairs of bin sizes
    double          scale      = 1,  // Optional scale factor for output
    bool            uniform    = true // true for uniform binning
  );

  void cv::calcBackProject(
    const cv::Mat*  images,          // C-style array of images, 8U or 32F
    int             nimages,         // number of images in 'images' array
    const int*      channels,        // C-style list, ints identifying channels
    const cv::SparseMat& hist,       // input (sparse) histogram array
    cv::OutputArray backProject,     // output single channel array
    const float**   ranges,          // C-style array, 'dims' pairs of bin sizes
    double          scale      = 1,  // Optional scale factor for output
    bool            uniform    = true // true for uniform binning
  );

  void cv::calcBackProject(
    cv::InputArrayOfArrays images,          // STL-vector of images, 8U or 32F
    const vector<int>&     channels,        // STL-vector, channels indices
    cv::InputArray         hist,            // input histogram array
    cv::OutputArray        backProject,     // output single channel array
    const vector<float>&   ranges,          // STL-style vector, range boundaries
```

```
    double                  scale      = 1,    // Optional scale factor for output
    bool                    uniform    = true  // true for uniform binning
);
```

有三个版本的cv::calcBackProject()。前两个使用C风格数组作为输入，一个支持稠密直方图，另一个支持稀疏直方图。第三个使用基于模板的输入而不是C风格的指针[注7]。不管是哪种情况，都应以单通道或是多通道数组（即images变量）集合的形式提供图像，同时以cv::calcHist()输出的直方图的格式传入直方图（即hist变量）。单通道数组集合恰和调用cv::calcHist()时所传入的第一个参数的具有相同的形式，只是此时它表示要和直方图做比较的图片集合。如果参数images是一个C风格数组（类型为cv::Mat*），你需要告诉cv::calcBackProject()数组中元素的个数，这正是参数nimages的功能。

参数channels是在反向投影中会发挥作用的通道的列表。同样的，该参数的形式和调用cv::calcHist()时对应的参数形式相同。通过在channels中按顺序枚举通道的编号，可以将channels中的整数和arrays中的通道相关联，具体地说，从第一个数组开始（arrays[0]），对第二个数组（images[1]）等等（例如如果images指向三个矩阵，每个三个通道，则第一个矩阵的通道对应编号0、1、2，第二个为3、4、5，第三个为6、7、8）。由此可见，虽然channels中元素的个数等于直方图hist的维数，但该数组并不要求等于images中的数组数（或者说images中通道的总数）。

反向投影的计算结果会保存在数组backProject中，它和images[0]的大小类型相同，只有一个通道。

由于直方图数据和其他数据一样，存储在简单的数组结构中，其中并没有空间记录直方图中各个区间的信息。在这种意义上来说，为了真正完全的理解直方图，需要与其相关的cv::Mat(或是cv::SparseMat或其他)和在通过cv::calcHist()构造直方图时使用的原始ranges数据结构[注8]。这正是cv::calcBackProject()需要在参数ranges中传入区间信息的原因。

最后，有两个可选参数scale和uniform。scale是一个可选的应用于backProject的放缩因子（如果你想对结果可视化，这个参数十分有用）。uniform表示输入的直方图是不是一个均匀的直方图（和cv::calcHist()中意义相同）。由于uniform默认为true，所以这个参数只需要在直方图非均匀的时候才设置。

注7：对于这三个而言，第三个一般在现代代码里更受欢迎，就是说，在很多现代OpenCV代码里使用C风格数组作为输入被认为是“过时的”。

注8：一个替代的方法是为直方图设计继承自cv::Mat的另一个数据结构，其中包含区间信息。库的作者在2.0（及以后）版本中之所以没有这么做，是出于对简化的考虑。

示例13-1展示了如何将一幅图分出各个单通道并构造一个它们的数组。如前所述，backProject中的各个像素被设置为与其hist中相关区间的计数值。如果直方图被归一化，那么这个值就和条件概率值有关（也就是说，图像中的像素值是hist对应的直方图所表示的物体中的概率）[注9]。在图13-7中，我们使用一个肤色直方图来产生一个图中像素为肤色的概率图。

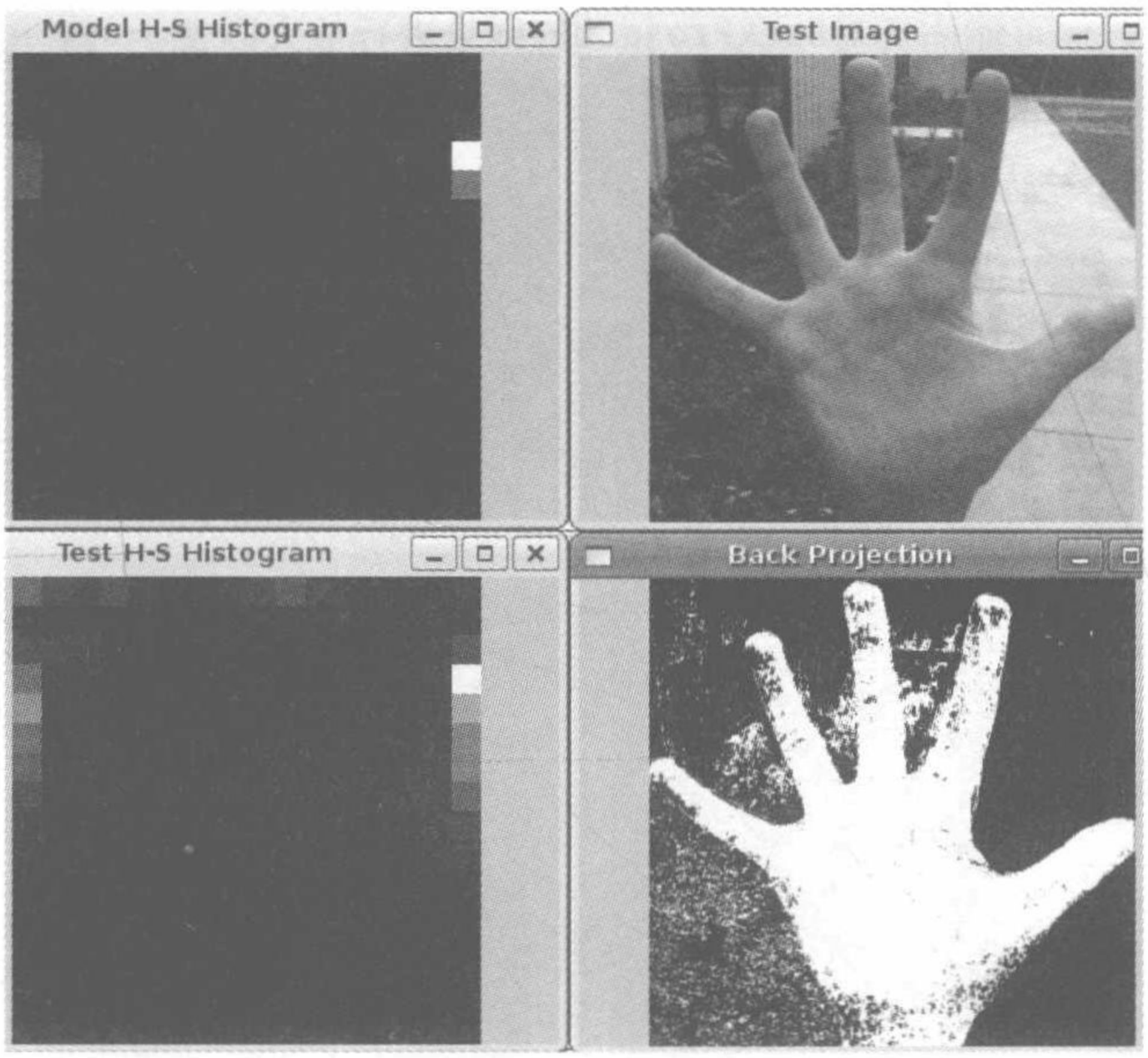


图13-7. 基于像素颜色用直方图进行反向投影：用HS（取图像HSV表示中的色调和饱和度部分）肤色直方图（左上）将手部图（右上）转为肤色概率图（右下）；左下图是手部图的直方图

在新图片中找寻一个物体或是一个需要的区域的一个方法如下。

1. 创建要寻找的物体或区域的直方图。
2. 为了在新图片中找到它，使用上述直方图对新图片进行cv::clacBackProject()以创建一个反向投影图（在反向投影图中，具有最大值的区域很可能包含感兴趣的物体或区域）。

注9：具体地说，在肤色使用HS分量进行直方图统计的时候，如果C是像素的颜色值，F是该像素为皮肤的概率，则可以得到概率映射$p(C|F)$，该映射描述了在真的为皮肤的情况下看到这种颜色的概率。这个概率和$p(F|C)$表示的意思不同，后者表示此种颜色属于皮肤的概率。但这两种概率可以通过贝叶斯定律[Bayes1763]联系起来，所以，如果我们知道场景中出现肤色目标的概率$p(F)$以及颜色中出现肤色的概率$p(C)$，我们就可以从$p(C|F)$计算$p(F|C)$。具体来说，贝叶斯定律给出了如下关系：

$$p(F|C) = \frac{p(F)}{p(C)} p(C \mid F)$$

3. 反向投影图中的高值区域取局部直方图并使用cv::compareHist()将它与要寻找的物体或区域的直方图进行比较。



在对一个以字节表示的图像而不是浮点表示的图像进行反向投影时，要小心，要么不对直方图进行归一化，要么在使用前对其进行比例放大。因为在归一化后的直方图中最大值为1，因此在8位图中小于1的都会被取整为0。你可能需要对backProject的结果进行放大以便于可视化，而放大多少取决于直方图中的值的大小。

模板匹配

通过cv::matchTemplate()进行模板匹配并不基于直方图，而是使用一个图像块在输入图像上进行“滑动”，并使用下文要介绍的匹配方法来进行比较。图13-8给出了一个例子。

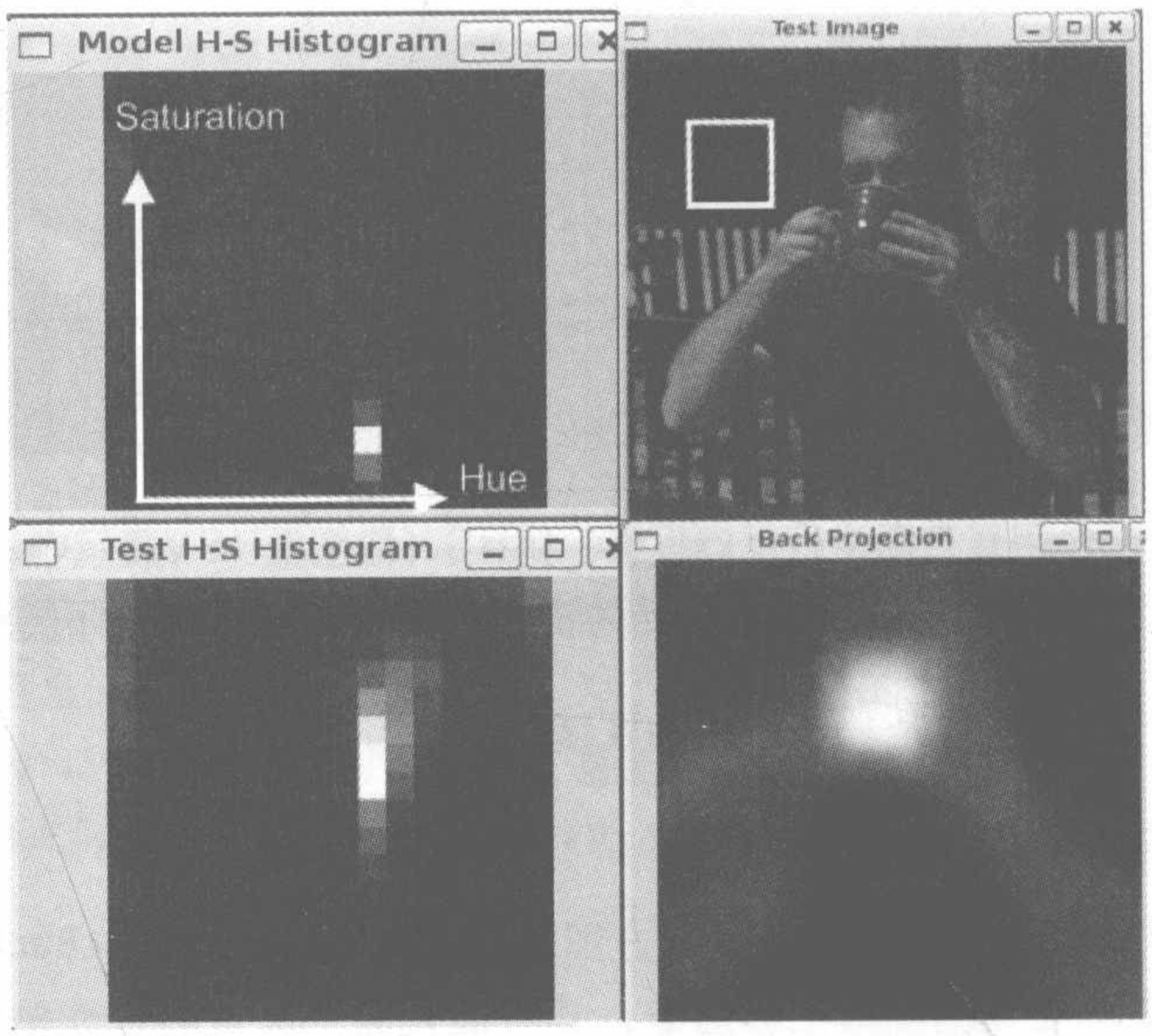


图13-8. 使用cv::matchTemplate()并结合cv::TM_CCOEFF_NORMED来定位一个大小相当于图像块（右上图中的白色方框）的物体（这里是一个咖啡杯）：要寻找的物体使用色调饱和度直方图进行建模（左上），可以和整个图像的HS直方图（左下）进行比较；从cv::matchTemplate()的结果（右下）可以看出，很容易从场景中辨识物体

在图13-9中，如果我们选取一个包含人脸的图像块，就可以在新图像中滑动这个图像块以寻找另一个人脸出现的位置。

```
void cv::matchTemplate(
  cv::InputArray  image,    // Input image to be searched, 8U or 32F, size W-by-H
  cv::InputArray  templ,    // Template to use, same type as 'image', size w-by-h
  cv::OutputArray result,   // Result image, type 32F, size (W-w+1)-by(H-h+1)
  int             method    // Comparison method to use
);
```



图13-9. cv::matchTemplate()使用图像块在另一张图片上扫描以寻找匹配的位置

cv::matchTemplate()的输入参数以一个单字节8位或浮点型灰度或彩色图片开始。templ为一个包含给定物体的（和当前图片相似的）另一张图片上取的图片块。计算结果存放在result中，它是一个单通道大小为（image.width - templ.width + 1, image.height - templ.height + 1）的单通道以整数字节或浮点存储的图片。匹配方法可以从下面列出的各种方法（其中*I*表示输入图像，*T*表示模板，*R*表示输出的结果图像）里选择。对于列出的每一个方法，都有一个归一化的版本。[注10]

方差匹配方法（cv::TM_SQDIFF）

这种方法以方差为依据进行匹配，因此完全匹配结果为0，而不匹配会得到一个很大的值。

$$R_{sq_diff} = \sum_{x',y'} [T(x',y') - I(x+x',y+y')]^2$$

注10：从Rodgers的[Rodgers88]中可以看出，归一化的版本最初是由Galton[Galton]开发的。归一化方法非常有用，因为它可以降低模板和图片之间由于光照不同引起的效果。这两种情况下归一化后的系数都相同。

归一化方差匹配方法（cv::TM_SQDIFF_NORMED）

正如cv::TM_SQDIFF, cv::TM_SQDIFF_NORMED中完全匹配会返回0。

$$R_{sq_diff_normed} = \frac{\sum_{x',y'} \left[T(x',y') - I(x+x',y+y') \right]^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

相关性匹配方法（cv::TM_CCORR）

这些方法以相乘的方式进行模板匹配，因此完全匹配会得到一个很大的值，而不匹配会得到一个很小的值或0。

$$R_{ccorr} = \sum_{x',y'} T(x',y') \cdot I(x+x',y+y')$$

归一化的互相关匹配方法（cv::TM_CCORR_NORMED）

正如cv::TM_CCORR，如果图片和模板极度不匹配，cv::TM_CCORR_NORMED会返回0。

$$R_{ccorr_normed} = \frac{\sum_{x',y'} T(x',y') \cdot I(x+x',y+y')}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

相关系数匹配方法（cv::TM_CCOEFF）

该方法对模板相对于其平均值的变化和图片相对于其平均值的变化进行匹配。因此，完全匹配会得到1，而完全误匹配会得到-1，如果用随机图片进行匹配会得到0，意思是两张图片毫无关联。

$$R_{ccoeff} = \sum_{x',y'} T'(x',y') \cdot I'(x+x',y+y')$$

$$T'(x',y') = T(x',y') - \frac{\sum_{x'',y''} T(x'',y'')}{(w-h)}$$

$$I'(x+x',y+y') = I(x+x',y+y') - \frac{\sum_{x'',y''} I(x'',y'')}{(w-h)}$$

归一化的相关系数匹配方法（cv::TM_CCOEFF_NORMED）

正如cv::TM_CCOEFF, cv::TM_CCOEFF_NORMED中相对好的匹配会得到正值，而误匹配会得到负值：

$$R_{ccoeff_normed} = \frac{\sum_{x',y'} T'(x',y') \cdot I'(x+x',y+y')}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

其中T'和I'和cv::TM_CCOEFF中的含义相同。

通常情况下，我们从一个相对简单的度量方法（方差）换成一个更复杂的度量方法（相关系数）时，会得到更精确的结果（以更长的计算时间为代价）。最好尝试所有的方法，选择一个对你来说可以兼顾精确度和计算时间的最好的方法。



要小心解释结果，方差方法以最小值表示最佳匹配，而相关性方法和相关系数方法使用最大值表示最佳匹配。

一旦使用cv::matchTemplate()获得result，我们就可以使用cv::minMaxLoc()或是cv::minMaxIdx()找到最优匹配出现的位置。同样，为了避免随机性引起的该位置恰好匹配得很好，我们希望确保在找到的最优匹配的邻域内也有不错的匹配结果。好的匹配点附近也应该有很好的匹配值，因为在进行匹配时，对模板位置进行轻微的扰动，并不会引起结果的剧烈变化。你可以在寻找最大匹配值（对于选用相关性判据或是相关系数判据）或者最小值（对于选取方差判据）前，先对结果进行轻微的平滑。这时，形态学算子可以帮你的忙。

示例13-3可以让你了解不同的模板匹配技巧是怎么工作的。该程序首先读取要匹配的模板和图片，然后使用本节讨论的方法对它们进行匹配。

示例13-3：模板匹配

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

void help( argv ){

cout << "\n"
  <<"Example of using matchTemplate(). The call is:\n"
  <<"\n"
  <<argv[0] <<" template image_to_be_searched\n"
  <<"\n"
  <<"   This routine will search using all methods:\n"
  <<"         cv::TM_SQDIFF        0\n"
  <<"         cv::TM_SQDIFF_NORMED 1\n"
  <<"         cv::TM_CCORR         2\n"
  <<"         cv::TM_CCORR_NORMED  3\n"
  <<"         cv::TM_CCOEFF        4\n"
```

```
    <<"          cv::TM_CCOEFF_NORMED 5\n"
    <<"\n";
}


// Display the results of the matches
//
int main( int argc, char** argv ) {

  if( argc != 3) {
    help( argv );
    return -1;
  }

cv::Mat src, templ, ftmp[6];                  // ftmp is what to display on

// Read in the template to be used for matching:
//
if((templ=cv::imread(argv[1], 1)).empty()) {
  cout << "Error on reading template " << argv[1] << endl;
  help( argv );return -1;
}

// Read in the source image to be searched:
//
if((src=cv::imread(argv[2], 1)).empty()) {
  cout << "Error on reading src image " << argv[2] << endl;
  help( argv );return -1;
}

// Do the matching of the template with the image
for(int i=0; i<6; ++i){
  cv::matchTemplate( src, templ, ftmp[i], i);
  cv::normalize(ftmp[i],ftmp[i],1,0,cv::MINMAX);
}

// Display
//
cv::imshow( "Template", templ );
cv::imshow( "Image", src );
cv::imshow( "SQDIFF", ftmp[0] );
cv::imshow( "SQDIFF_NORMED", ftmp[1] );
cv::imshow( "CCORR", ftmp[2] );
cv::imshow( "CCORR_NORMED", ftmp[3] );
cv::imshow( "CCOEFF", ftmp[4] );
cv::imshow( "CCOEFF_NORMED", ftmp[5] );

  // Let user view results:
  //
  cv::waitKey(0);
}
```

注意，代码中使用了cv::normalize()，这可以使我们用同样的代码来绘制不同方法产生的结果（回想一下，有些方法会产生带负值的结果）。在归一化时，我们使用了

cv::NORM_MINMAX标记，使这归一化函数会对匹配的结果进行平移并缩放到0，1之间。图13-10展示了使用cv::matTemplate()每个可用匹配方法来对人脸模板进行匹配的结果。使用归一化的匹配方法在绝大数情况下会有更好的结果，特别是在室外环境下的图像。在其中，相关系数方法给出了最清晰明确的匹配结果，当然是以较大的计算时间为代价。对于具体的应用而言，例如通过摄像头进行自动部件检测或者特征跟踪，应该尝试使用所有的方法，从中找出一个能同时兼顾效率和准确率的方法。

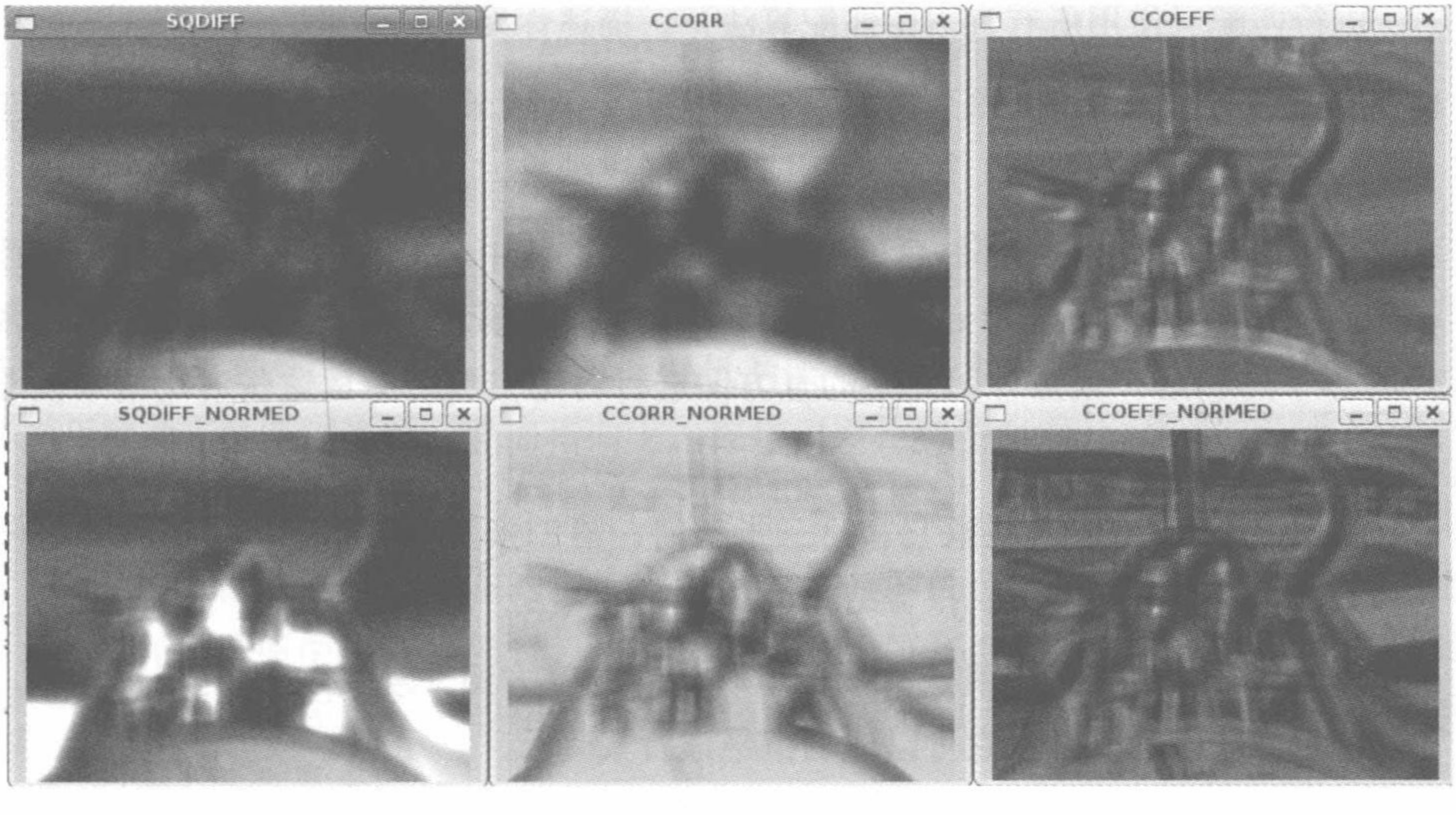


图13-10. 使用6种模板匹配方法对图13-9进行模板匹配：对于方差方法，最优匹配为0，对于其他方法，为最大值；因此，左列最优匹配点为黑色区域，而其他两列最优匹配点为亮色区域

小结

在这一章中，我们了解了OpenCV是如何将直方图表示为稠密或者稀疏矩阵对象的。在实际使用中，直方图通常用来表示一个n维数组每个元素概率的概率密度函数。我们也讨论了怎么用这种概率密度函数来做物体或区域的识别。我们了解了针对表示直方图的数组进行的一些基本操作，在这些数组被解释为概率分布时，这些操作用处很大，例如将归一化后的直方图与其他分布进行比较。我们最终以模板匹配结束本章。模板匹配在高度结构化的图片中非常有用。

练习

1. 在0到1之间生成1000个随机数。设计区间的大小，并针对$1/r_i$计算一个直方图。
   a. 每个区间内的数字的个数相等（在±10的范围内）吗？

b. 提出一个处理高度非线性分布的方法，使得每个区间内的数据个数大致相等（在−10～10的范围内）。

2. 在文中给出的三种光照条件下各拍摄三张手部的照片。使用cv::calcHist()对室内环境下其中一张照片上的肤色计算BGR颜色直方图。

   a. 尝试使用大区间（例如每个维度两个区间）、中等数量的区间（每个维度16个）和很多的区间（每个维度256个）。然后将其与室内光照条件下的其他照片进行匹配（使用所有直方图匹配算法）。描述你发现的结果。

   b. 现在给每个维度增加8个，然后32个区间，再试着在不同光照条件下进行匹配（在室内光照下训练，在室外测试）。描述结果。

3. 和练习2中一样，收集手部肤色的BGR直方图。取其中一个室内环境下的直方图作为模型，用EMD距离（earth mover's distance）将其与第二张室内环境下的、第一张室外阴影下的和第一张室外阳光下的直方图进行对比。然后应用这些对比结果设置阈值。

   a. 使用该EMD阈值，观察它和第三张室内、第二张室外阴影下和第二张室外阳光下的直方图对比情况。报告结果。

   b. 使用从非肤色背景随机选取的图像块计算的直方图来观察EMD的判别性能。在将它与肤色对比时，它可以成功拒绝该背景吗？

4. 使用收集的手部图片，设计一个可以判别该图片是在何种光照条件下拍摄的图片的直方图。为此，你可以构造一些特征，或许从整个场景中的某些部分进行采样，对亮度值进行采样，和/或对亮度值进行采样（例如使用从上到下的图像块）或从中心到边缘的梯度。

5. 将三种光照条件（室内、室外阴影和室外阳光下）下的肤色直方图模型组合起来。

   a. 在每种光照条件下使用三种直方图进行匹配。报告每种直方图在它自己的光照条件下的表现，在其他两种光照条件下的表现。

   b. 用我们所学的知识创建一个“直方图切换”模型。首先使用场景检测器检测应当使用何种直方图：室内、室外阴影还是室外阳光直射。接下来使用对应的肤色模型来对图像块进行判别。这种切换模型效果怎么样？

6. 构造一个肤色感兴趣（或“关注”）区域检测器。

   a. 首先只考虑室内的情况，使用手部和脸部的采样样本构造一个BGR直方图。

   b. 在三个新的室内场景中获取图片，其中两张带有肤色而一张没有。使用cv::calcBackProject()来创建一个反投影图。

   c. 在图片上取16等距格点，使用cv::meanShift()来寻找反投影图的峰值，mean-

shift窗的大小为¼图像的长和宽。

d. 在每个峰值附近的⅛ × ⅛区域计算直方图。

e. 尝试使用cv::compareHist()的所有比较方法来在三张新收集的图片中寻找肤色区域。

f. 报告结果。哪个比较方法比较准确?

7. 构造一个肤色感兴趣（或“关注”）区域检测器.

a. 首先只考虑室内的情况，使用手部和脸部的采样样本构造一个RGB直方图。

b. 使用cv::calcBackProject()寻找肤色区域。

c. 使用第10章介绍的cv::erode()来对噪音进行清理，再使用cv::floodFill()（第12章）来寻找图像中最大的区域。这些便是“关注”的区域。

8. 尝试进行手部姿态识别。在距离摄像机两英寸的地方对手部进行拍照来构造一些（非移动）的手部姿势：竖拇指向上，拇指向左，拇指向右。

a. 使用练习7中的检测器进行肤色检测，在检测到的区域附近计算图像的梯度直方图从而构造这三个姿态的梯度直方图模板。同样创建一个脸部的梯度直方图模板（如果图中没有脸部），这样你就有一个（不含姿态的）大面积肤色区域模型。你可能还需要一些相似的但不含特定姿态的手部直方图，以便将它们和真实姿态区分开来。

b. 使用网络摄像头进行识别测试：使用肤色检测器检测“潜在的手”；在每个肤色区域附近计算梯度并将它与前面的模型进行匹配，设计一个阈值来检测姿势。如果有两个模型都高于阈值，取匹配更好的作为结果。

c. 将你的手远离相机一或两英寸，观察梯度直方图是否还能识别出收的姿态。报告结果。

9. 使用EMD重复练习8的实验。将手远离时EMD效果如何?

第14章

轮廓

虽然Canny之类的边缘检测算法可以根据像素间的差异检测出轮廓边界的像素，但是它并没有将轮廓作为一个整体进行处理。下一步是要将这些边缘像素合成轮廓。现在你也许希望OpenCV中提供方便的函数能直接实现这一步，事实上，这个函数就是cv::findContours()。为了演示如何使用这一函数，本章将从一些基本的问题入手。了解这些基本的概念后，我们就可以深入讨论轮廓检测的一些细节。最后，我们将讨论轮廓检测的一些实际应用。

轮廓查找

一个轮廓对应一系列点，这些点以某种方式表示图像中的一条曲线。在不同情况下，这种表示方式也有所不同。有多种方式可以表示一条曲线。在OpenCV中，轮廓用标准模板库（STL）向量vector<>表示，向量中的每一个值都包含轮廓上下一个点的位置信息。表示轮廓有许多方式，而用一系列二维顶点（vector<cv::Point>或vector<cv::Point2f>）来表示只是其中最常见的一种。例如*Freeman chain*方法，就是将每个点表示为相对起始点在某个特定方向上发生的位移。在之后遇到这些不同时，我们将进一步详细讨论。现在，重要的是了解轮廓几乎总是以标准模板库中的向量来表达，但并不一定局限于常见的cv::Point对象。

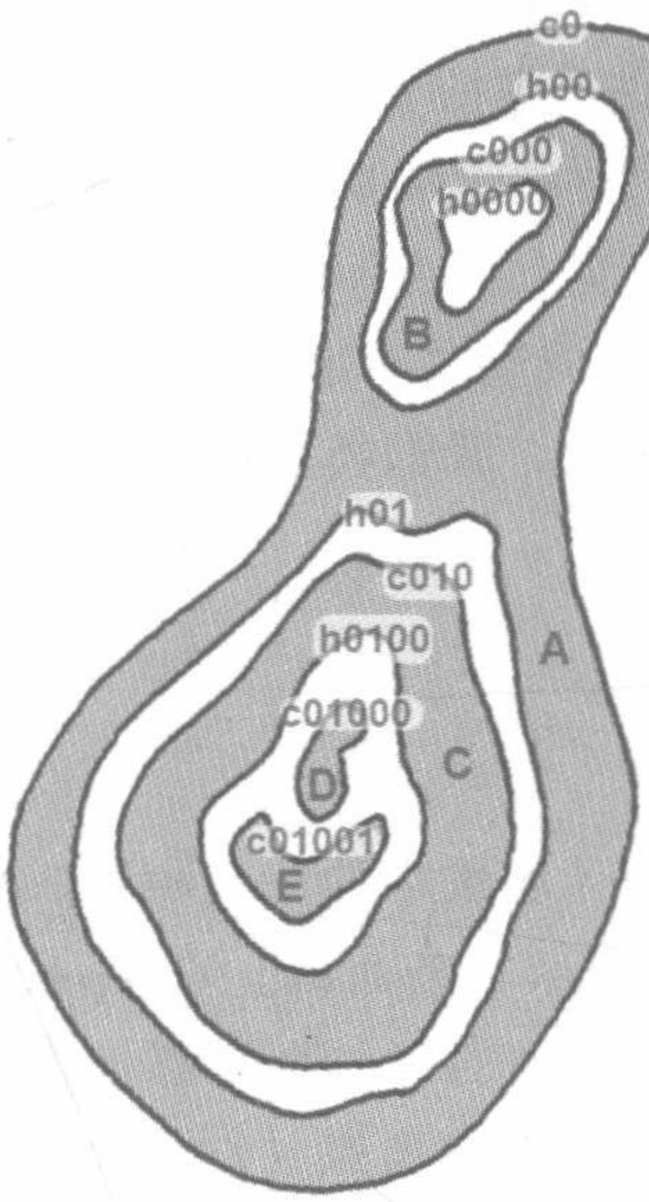
函数cv::findContours()从二维图像中计算轮廓。它处理的图像可以是从cv::Canny()函数得到的有边缘像素的图像，或是从cv::threshold()及cv::adaptiveThreshold()函数得到的图像，这时边缘是正负区域之间的边界。[注1]

注1：　采用cvFindContours()处理边缘图像和二值图像的方式有细微差别，我们稍后将讨论。

轮廓层次

在了解到底如何提取轮廓之前，有必要花一些时间来理解轮廓到底是什么以及一组轮廓之间如何互相关联。特别要注意轮廓树的概念，这对于理解其中一种最有效的方法cv::findContours()非常重要。[注2]

花点时间看看图14-1，它展示了cv::findContours()的函数功能。图像的上半部分是一张白色背景上的测试图像，包含一系列彩色区域（标记A到E）。[注3]图像中还展示了将由cv::findContours()函数定位的轮廓。这些轮廓被标记为*cX*或*hX*，其中，*c*代表“轮廓”（contour），*h*代表“孔”（hole），而X表示数字。其中一些轮廓用虚划线表示，这代表它们是白色区域的外部边界（例如，非零区域）。点线则表示白色区域的内部边界或孔的外部边界（例如，非零区域）。OpenCV的cv::findContours()函数区分内部和外部边界。



idx	contour
0	c0
1	h00
2	h01
3	c000
4	c010
5	h0000
6	h0100
7	c01000
8	c01001

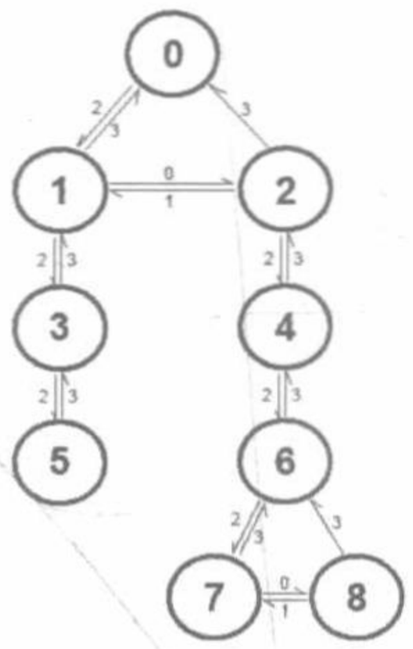


图14-1. 一张输入cv::findContours()函数的测试图像（左图）。图中有五块颜色区域（分别标记为A，B，C，D，E），每块区域的外部边界和内部边界都各自组成轮廓。因此共有9条轮廓。每条轮廓都由一组输出列表表示（右上角图——轮廓参数）。也可以选择生成一组层次表达（右下角图——层次参数）。在右下角的图中（对应构筑的轮廓树），每一个节点就是一条轮廓。根据每个节点在层次队列中的四元数组索引，图中的链接都做了相应标记

注2：该方法请参考[Suzuki85]。

注3：为了清晰展示，图中原本的黑色区域被画为灰色，可以简单假设这幅图像进行过阈值操作，因此传递给cv::findContours()前灰色区域会被设为黑色。

包含的概念在许多应用中都很重要。因此，OpenCV允许将找到的轮廓聚合成一棵轮廓树，在其结构中添加包含的关系。[注4]这张测试图像的轮廓树在根节点的轮廓叫c0，孔h00和h01是它的子树。这些轮廓中直接包含的轮廓成为它们的子节点，以此类推。

有多种方式可以表示这样的一棵轮廓树。OpenCV用数组（通常为vector）来表示轮廓树，其中每个值都代表一条特定轮廓。在该数组中，每个值都包含一个四整数组，通常像四通道数组中一样表示为cv::Vec4i型的元素。然而，在这种情况下，节点的向量表示中的每个部分都附加了一层特殊的含义。层次列表中的每个节点都由四个整数组成。每个数都代表在层次列表中与当前节点有特定关系的另一个节点。而当节点之间不存在这样的联系时，数组中的相应元素将被设成-1。例如根节点的3号元素（代表父节点）的值将被设为-1，因为根节点没有父节点。

以图14-1中的轮廓为例。算上每块区域的外部边界和内部边界，五块颜色区域共含有九条轮廓。从这九条轮廓搭建一棵轮廓树，则每个节点都将以其包含的轮廓作为自己的子节点。所构成的轮廓树如图14-1中的右下角图所示。对于每一个节点，所有有效链接也同样展示在图中。每条链接都根据其在该节点的四元素数组中的位置进行了标记。如表14-1所示。

表14-1：轮廓层次列表中四元素数组里每个元素代表的含义

索引	含义
0	同级的下一条轮廓
1	同级的前一条轮廓
2	下级的第一个子节点
3	上级的父节点



值得注意的是，在经过cv::canny()或类似的边缘检测算子处理过的图像上使用cv::findContours()函数，和在如测试图像#test_image_left_parenthesisleft_siderigh的二值图上使用该函数，结果是不同的。事实上，cv::findContours()函数并不理解边缘图像。这意味着对于cv::findContours()函数，一条“边缘”只是一块非常窄的“白色区域”。因此，对于每一条外部轮廓，总有一条几乎完全相同的孔轮廓。实际上，这个孔就是外部边界的内部区域。你可以把它当作从白色到黑色的过渡区域，同时也表示边缘曲线的内部边界。

注4：轮廓树最早由Reeb [Reeb46]提出，随后在[Bajaj97]，[Kreveld97]，[Pascucci02]和[Carr04]等论文中被改进。

使用cv::findContours()查找轮廓

了解了轮廓树的概念后，现在来看如何调用cv::findContours()函数以及如何理解函数返回值。

```
void cv::findContours(
  cv::InputOutputArray    image,          // Input "binary" 8-bit single channel
  cv::OutputArrayOfArrays contours,       // Vector of vectors or points
  cv::OutputArray         hierarchy,      // (optional) topology information
  int                     mode,           // Contour retrieval mode (#way_in_which_
the_tree_node_variables_are)
  int                     method,             // Approximation method
  cv::Point               offset = cv::Point() // (optional) Offset every point
);

void cv::findContours(
  cv::InputOutputArray    image,          // Input "binary" 8-bit single channel
  cv::OutputArrayOfArrays contours,       // Vector of vectors or points
  int                     mode,           // Contour retrieval mode (#way_in_which_
the_tree_node_variables_are)
  int                     method,         // Approximation method
  cv::Point               offset = cv::Point() // (optional) Offset every point
);
```

第一个参数是输入图像，图像必须是8位单通道图像，并且应该被转化成二值的（例如，所有非0像素都是一个定值）。cv::findContours()运行时，这个图像会被直接涂改，因此如果是将来还有用的图像，应该复制之后再传给cv::findContours()。第二个参数是一组数组，大部分情况下是一个或多个标准模板库vector。数组中储存的是找到的轮廓（例如，在一个轮廓vector中，contours[i]是一条特定轮廓，而contours[i][j]则是contour[i]中的一个点。

下一个参数hierarchy是可选项，如上述的两种函数形式所示。如果给定该参数，则hierarchy将输出所有轮廓的树结构。这一输出将是一个数组（通常仍是标准模板库向量），每条轮廓对应数组中的一个值。数组中的每个值都是一个四元数组，每个元素代表一个与当前节点有特定链接的节点。如表14-1所示。

参数mode告诉OpenCV你期望的轮廓提取方式。mode有4种可能值。

cv::RETR_EXTERNAL

只检索最外层轮廓。在图14-1中，只存在一条最外层轮廓，因此，图14-2中的第一条轮廓指向最外面的序列，并且该轮廓不与其他轮廓连接。

cv::RETR_LIST

检索所有轮廓并保存到表（List）中。图14-2展示了图14-1中测试图像的轮廓层级。

这种情况下，共找到了9条轮廓，它们通过hierarchy[i][0]和hierarchy[i][1]互相连接，（此处并未用到hierarchy[i][2]和hierarchy[i][3]）。[注5]

cv::RETR_CCOMP

检索所有的轮廓，并将它们组织成双层结构（two-level hierarchy），顶层边界是所有成分的外部边界，第二层边界是孔的边界。图14-2中，我们能看到5条外部边界，其中三条包含孔。孔被hierarchy[i][2]和hierarchy[i][3]与它们的外部边界连接起来。最外层的边界c0中包含两个孔。因为hierarchy[i][2]可以只包括一个值，节点可以只有一个子节点。c0之内的所有孔相互间由hierarchy[i][0]和hierarchy[i][1]指针连接。

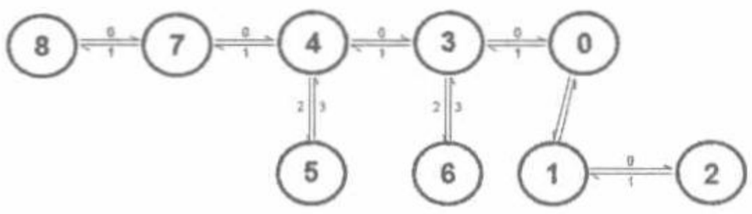
cv::RETR_TREE

检索所有轮廓并重新建立网状轮廓结构。在我们给出的例子（图14-1和图14-2）中，这意味着根节点是最外层的轮廓c0。c0之下是孔h00，在同一层中与另一个孔h01相连。同理，每个孔都有子节点（相对应的是c000和c010），这些子节点分别与父节点垂直相连。这个步骤一直持续到图像最内层的轮廓，这些轮廓会成为树的叶子节点。

cv::RETR_EXTERNAL ⓪

idx	contour
0	c0
1	h00
2	h01
3	c000
4	c010
5	h0000
6	h0100
7	c01000
8	c01001

cv::RETR_CCOMP



cv::RETR_LIST



cv::RETR_TREE

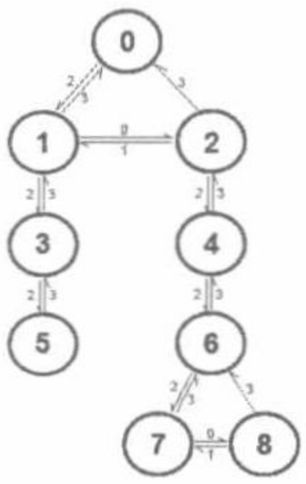


图14-2. 被cv::findContours()找到的轮廓通过树节点变量相互连接。轮廓节点与图14-1中相同

注5：一般不采用cv::RETR_LIST选项。在之前的OpenCV库中，轮廓的返回值并未自动排序为一个列表，因此这一选项是有意义的，但现在，vector<>类型已经能够实现这一功能。

以下的值与method有关（例如，轮廓如何被表达）。

cv::CHAIN_APPROX_NONE
: 将轮廓编码中的所有点转换为点。这一操作会产生大量的点，每个点都将成为前一个点的8个邻点之一，不会减少返回的点数，

cv::CHAIN_APPROX_SIMPLE
: 压缩水平、垂直、斜的部分，只保留最后一个点。在许多特殊情况下，这一操作将大大减少返回的点数。极端例子是，对于一个沿着x-y方向的矩形（任意大小），只返回4个点。

cv::CHAIN_APPROX_TC89_L1 or cv::CHAIN_APPROX_TC89_KCOS
: 使用Teh-Chin链逼近算法中的一个。[注6]Teh-Chin（T-C）算法是一种更复杂且计算密集型的算法，用于减少返回的点数。运行T-C不需要额外的参数。

cv::findContours()的最后一个参数是offset。该参数是可选项。给出这一参数，返回的轮廓中所有点都会根据参数值发生偏移。这在两种情况下非常有用。一是当你希望从兴趣区域（ROI, region of interest）中提取的轮廓用原图坐标系表达时，二是与之相反，当你希望从原图中提取的轮廓用图像子区域坐标系表达时。

绘制轮廓

获得一列轮廓后，一个最常用的功能是在屏幕上绘制检测到的轮廓。绘制轮廓可以用cv::drawContours()函数完成。

```
void  cv::drawContours(
  cv::InputOutputArray   image,                  // Will draw on input image
  cv::InputArrayOfArrays contours,               // Vector of vectors or points
  int                    contourIdx,             // Contour to draw (-1 is "all")
  const cv::Scalar&      color,                  // Color for contours
  int                    thickness = 1,          // Thickness for contour lines
  int                    lineType  = 8,          // Connectedness ('4' or '8')
  cv::InputArray         hierarchy = noArray(),  // optional (from findContours)
  int                    maxLevel  = INT_MAX,    // Max descent in hierarchy
  cv::Point              offset    = cv::Point() // (optional) Offset all points
)
```

第一个参数image很简单：待绘制轮廓的图像。下一个参数contour是要绘制的轮廓。该参数的类型与cv::findContours()的输出contour相同，都是储存在列表中的点。contourIdx用于告诉cv::drawContours()需要绘制的是contours参数中的某一条轮廓还

注6：若对该算法的实现细节感兴趣，请参考C. H. Teh and R. T. Chin的论文“On the Detection of Dominant Points on Digital Curve,” PAMI 11, no.8 (1989): 859–872。但是，由于该方法不需要调节参数，所以不需要了解算法实现细节，也能用得很好。

是全部轮廓。假如contourIdx是一个正数，则对应的轮廓将被绘制。假如contourIdx为负（通常设为-1），则所有轮廓都将被绘制。

参数color、thickness、lineType的功能与其他用于绘制的函数（如cv::Line()）中的对应参数功能相同。一般来说，参数color是一个四元组cv::Scalar；参数thickness是一个整数，表示绘制的线的粗细；参数lineType可以是4或8，代表绘制的线将是四联通线（不美观）、八联通线（较美观）或是cv::AA线（美观）。

参数hierarchy对应cv::findContours()函数输出的层次。参数hierarchy和参数maxLevel共同起作用。maxLevel限制将在图上绘制的轮廓层次深度。将maxLevel设为0表示只绘制“第0层”（最高层次）的轮廓；设为其他非0正数，则表示绘制最高层以下相同数量层级的轮廓。如图14-2所示，这对轮廓树很有用。同样，当你希望在连接成分（cv::RETR_CCOMP）时希望只显示最外层轮廓（而不显示内部孔轮廓）时，这也很有帮助。

最后，绘制轮廓时可以设置offset参数，这样，轮廓就能画在与原定义的绝对坐标系不同的地方。当轮廓坐标系被转换成质心坐标系或其他局部坐标系的时候，这个特性格外有用。如果在图像上的不同感兴趣区域（ROI）多次执行cv::findContours()，然后又想将所有结果在原来的大图像上显示出来，offset也很有用。相反，也可以先从大图提出一个轮廓，然后再用offset和填充，在小图像上形成和轮廓对应的蒙版（mask）。

轮廓实例

示例14-1来自OpenCV安装包。首先创建一个窗口用于显示图像，滑动条（trackbar）用于设置阈值，然后对采二值化后的图像提取轮廓并绘制。当控制参数的滑动条变化时，图像被更新。

示例14-1：根据滑动条参数检测轮廓，在滑动条变化时重新检测

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

cv::Mat g_gray, g_binary;
int g_thresh = 100;

void on_trackbar( int, void* ) {

  cv::threshold( g_gray, g_binary, g_thresh, 255, cv::THRESH_BINARY );
  vector< vector< cv::Point> > contours;
  cv::findContours(
    g_binary,
    contours,
    cv::noArray(),
    cv::RETR_LIST,
```

```
    cv::CHAIN_APPROX_SIMPLE
  );
  g_binary = cv::Scalar::all(0);

  cv::drawContours( g_binary, contours, -1, cv::Scalar::all(255));
  cv::imshow( "Contours", g_binary );

}

int main( int argc, char** argv ) {

if( argc != 2 || ( g_gray = cv::imread(argv[1], 0)).empty() ) {
  cout << "Find threshold dependent contours\nUsage: " <<argv[0]
    <<"fruits.jpg" << endl;
  return -1;
}
cv::namedWindow( "Contours", 1 );

cv::createTrackbar(
  "Threshold",
  "Contours",
  &g_thresh,
  255,
  on_trackbar
);
on_trackbar(0, 0);

  cv::waitKey();

  return 0;

}
```

我们关注的重点在on_trackbar()函数内。图像g_gray被阈值化，好让所有值小于g_thresh的像素置0。然后在这张阈值化的图像上使用cv::findContours()函数。最后，调用cv::drawContours()函数，在灰度图像上绘制白色轮廓。

另一个轮廓实例

在示例14-2中，我们检测出输入图像的轮廓，然后逐个绘制每条轮廓。从这个例子中，我们可以了解到轮廓检测方法（如代码中是cv::RETR_LIST）以及max_depth（代码中是0）等参数的用法。如果设置max_depth为一个较大的值，你会发现cv::findContours()返回的轮廓是通过hierarchy[i][1]连接遍历的。因此，对于其他一些拓扑结构（cv::RETR_TREE，cv::RETR_CCOMP等），你会发现有的轮廓画过不止一次。

示例14-2：在输入图像上寻找并绘制轮廓

```
#include <opencv2/opencv.hpp>
#include <algorithm>
#include <iostream>
```

```
using namespace std;

struct AreaCmp {
    AreaCmp(const vector<float>& _areas) : areas(&_areas) {}
    bool operator()(int a, int b) const { return (*areas)[a] > (*areas)[b]; }
    const vector<float>* areas;
};


int main(int argc, char* argv[]) {

  cv::Mat img, img_edge, img_color;

// load image or show help if no image was provided
//
if( argc != 2 || (img = cv::imread(argv[1],cv::LOAD_IMAGE_GRAYSCALE)).empty() ) {
  cout << "\nExample 8_2 Drawing Contours\nCall is:\n./ch8_ex8_2 image\n\n";
  return -1;
}

cv::threshold(img, img_edge, 128, 255, cv::THRESH_BINARY);
cv::imshow("Image after threshold", img_edge);
vector< vector< cv::Point > > contours;
vector< cv::Vec4i > hierarchy;

cv::findContours(
  img_edge,
  contours,
  hierarchy,
  cv::RETR_LIST,
  cv::CHAIN_APPROX_SIMPLE
);
cout << "\n\nHit any key to draw the next contour, ESC to quit\n\n";
cout << "Total Contours Detected: " << contours.size() << endl;

vector<int> sortIdx(contours.size());
vector<float> areas(contours.size());
for( int n = 0; n < (int)contours.size(); n++ ) {
  sortIdx[n] = n;
  areas[n] = contourArea(contours[n], false);
}

// sort contours so that the largest contours go first
//
std::sort( sortIdx.begin(), sortIdx.end(), AreaCmp(areas ));

for( int n = 0; n < (int)sortIdx.size(); n++ ) {
  int idx = sortIdx[n];
  cv::cvtColor( img, img_color, cv::GRAY2BGR );
  cv::drawContours(
    img_color, contours, idx,
    cv::Scalar(0,0,255), 2, 8, hierarchy,
    0                  // Try different values of max_level, and see what happens
  );
  cout << "Contour #" << idx << ": area=" << areas[idx] <<
```

```
        ", nvertices=" << contours[idx].size() << endl;
      cv::imshow(argv[0], img_color);
      int k;
      if( (k = cv::waitKey()&255) == 27 )
        break;
    }
    cout << "Finished all contours\n";

    return 0;

}
```

快速连通区域分析

与轮廓分析紧密相关的另一种方法是连通区域分析。采用阈值化等方法分割一张图像后，我们可以采用连通区域分析来有效地对返回图像逐张分离和处理。OpenCV中的连通区域分析算法，输入要求是一张二值（黑白）图像，输出是一张像素标记图，其中属于同一连通区域的非零像素都是同一定值。例如，在图14-1中有五块连通区域，其中最大的一块包含两个孔，较小的两块各自包含一个孔，最小的两块区域内不含孔。在背景分割算法中，连通区域分析常常被用作后处理滤波器，用于去掉小噪声块。另外，在例如OCR这一类含有已知前景待提取的问题中，这一方法也很常用。我们当然希望能够快速运行这一基本操作。一种更慢的“手动”运行方式是先调用cv::findContours()函数（传入cv::RETR_CCOMP标志），随后在得到的连通区域上循环调用cv::drawContours()，参数设置为color=component_label，thickness=-1。这种方法速度慢的原因如下。

- cv::findContours()函数首先为每条轮廓分配一个独立的标准模板库向量（vector），而图像中可能存在上百、甚至上千条轮廓。
- 之后当你想填充一块由一条或多条轮廓包围的非凸区域时，cv::drawContours()也很慢，而且需要收集所有包围该区域的小线段并排序。
- 最后，收集一块连通区域的基本信息（例如一块区域或一个包围框）需要更多、更耗时的函数调用。

幸好，在OpenCV 3中，函数cv::connectedComponents()和函数cv::connectedComponentsWithStats()代替了所有上述复杂操作。

```
int  cv::connectedComponents (
  cv::InputArrayn image,                // input 8-bit single-channel (binary)
  cv::OutputArray labels,               // output label map
  int             connectivity = 8,     // 4- or 8-connected components
  int             ltype        = CV_32S // Output label type (CV_32S or CV_16U)
);

int  cv::connectedComponentsWithStats (
  cv::InputArrayn image,                // input 8-bit single-channel (binary)
```

```
    cv::OutputArray labels,                 // output label map
    cv::OutputArray stats,                  // Nx5 matrix (CV_32S) of statistics:
                                            // [x0, y0, width0, height0, area0;
                                            //  ... ; x(N-1), y(N-1), width(N-1),
                                            // height(N-1), area(N-1)]
    cv::OutputArray centroids,              // Nx2 CV_64F matrix of centroids:
                                            // [ cx0, cy0; ... ; cx(N-1), cy(N-1)]
    int             connectivity = 8,       // 4- or 8-connected components
    int             ltype        = CV_32S // Output label type (CV_32S or CV_16U)
  );
```

cv::connectedComponents()函数简单生成了标记图。cv::connectedComponentsWithStats()也生成标记图，但同时也返回关于每块连通区域的一些重要信息，例如包围框（bounding box）、面积、质心（centroid）等。如果不需要质心信息，就将cv::noArray()传给OutputArray centroids参数。两个函数都返回找到的连通区域数量。这两个函数不需要用到cv::findContours()和cv::drawContours()；相反，它们直接实现“Two Strategies to Speed Up Connected Component Labeling Algorithms”论文（参考[Wu08]）中的高效算法。

考虑一个简单的例子，去掉所有较小的连通区域，并绘制出剩余的区域，如示例14-3所示。[译注1]

示例14-3：绘制标记连通区域

```
#include <opencv2/opencv.hpp>
#include <algorithm>
#include <iostream>

using namespace std;
int main(int argc, char* argv[]) {

  cv::Mat img, img_edge, labels, img_color, stats;

  // load image or show help if no image was provided
  if( argc != 2
    || (img = cv::imread( argv[1], cv::LOAD_IMAGE_GRAYSCALE )).empty()
  ) {
    cout << "\nExample 8_3 Drawing Connected componnents\n" \
      << "Call is:\n" <<argv[0] <<" image\n\n";
    return -1;
  }

  cv::threshold(img, img_edge, 128, 255, cv::THRESH_BINARY);
  cv::imshow("Image after threshold", img_edge);

  int i, nccomps = cv::connectedComponentsWithStats (
    img_edge, labels,
```

译注1：本段代码关于connectedComponentsWithStats函数的调用，最后一个参数输入noArray可能会导致程序崩溃，建议传入一个矩阵以保证程序的顺利运行。

```
  stats, cv::noArray()
);
cout << "Total Connected Components Detected: " << nccomps << endl;

vector<cv::Vec3b> colors(nccomps+1);
colors[0] = Vec3b(0,0,0); // background pixels remain black.
for( i = 1; i <= nccomps; i++ ) {
  colors[i] = Vec3b(rand()%256, rand()%256, rand()%256);
  if( stats.at<int>(i-1, cv::CC_STAT_AREA) < 100 )
    colors[i] = Vec3b(0,0,0); // small regions are painted with black too.
}
img_color = Mat::zeros(img.size(), CV_8UC3);
  for( int y = 0; y < img_color.rows; y++ )
    for( int x = 0; x < img_color.cols; x++ )
    {
      int label = labels.at<int>(y, x);
      CV_Assert(0 <= label && label <= nccomps);
      img_color.at<cv::Vec3b>(y, x) = colors[label];
     }
  cv::imshow("Labeled map", img_color);
  cv::waitKey();
  return 0;
}
```

深入分析轮廓

当分析一张图像的时候，针对轮廓，我们也许有很多事情要做。毕竟，所有轮廓都是或即将是我们想要进行识别或操作的。另外相关的还有多种对轮廓的处理，如描述轮廓，简化或拟合轮廓，匹配轮廓到模板，等等。

在这一节中，我们将介绍一些OpenCV函数，这些函数或可以完成我们的任务，或可以使工作变得容易。

多边形逼近

当我们绘制一个多边形或进行形状分析时，通常需要使用多边形逼近一个轮廓，使顶点数变少。有多种方法可以实现这个功能，OpenCV实现了其中的两种逼近方法。

函数cv::approxPolyDP()进行多边形逼近

cv::approxPolyDP()实现的是下列两种算法中的一种：[注7]

注7：对于熟悉的读者，我们这里讨论的是Douglas-Peucker (DP)逼近算法[Douglas73]。其他常用的方法包括Rosenfeld-Johnson算法[Rosenfeld73]和Teh-Chin算法[Teh89]。在后两者中，Teh-Chin算法在OpenCV中并不用于缩减顶点，而是在提取多边形时使用，详情参见“Finding contours with cv::findContours()”。

```
void cv::approxPolyDP(
  cv::InputArray  curve,       // Array or vector of 2-dimensional points
  cv::OutputArray approxCurve, // Result, type is same as 'curve'
  double          epsilon,     // Max distance from 'curve' to 'approxCurve'
  bool            closed       // If true, assume link from last to first vertex
);
```

cv::approxPolyDP()一次处理curve输入的一个多边形。cv::approxPolyDP()的输出储存在approxCurve数组中。和之前相同，这些多边形既可以表示成标准模板库向量cv::Point，也可以表示为OpenCV中的大小为$N \times 1$的cv::Mat数组（双通道）。不管你选择哪种表示方法，curve和approxCurve应该属于同一类型。

参数epsilon是你要求的逼近准确度。epsilon的含义是，你允许在原多边形和最终拟合的多边形之间存在的最大偏差。最后一个参数closed指明curve中的一系列点是否是一个闭合的多边形。若设为真true，则认为曲线是闭合的（例如，最后一个点应该与第一个点相连）。

理解Douglas-Peucker算法

为了更好地了解如何设置epsilon参数以及领会cv::approxPolyDP()函数，有必要花一些时间理解该算法具体如何运行。如图14-3所示，该算法首先从轮廓（图B）中挑出两个最远的点，将两点相连（图C）。然后在原来的轮廓上寻找一个离线段距离最远的点，将该点加入逼近后的新轮廓中。

如(D)所示，算法反复迭代，不断将最远的点添加到结果中，直到所有点到多边形的最短距离小于parameter参数指定的精度（图F）。从这里可以看出，将该精度设置为轮廓周长或外包矩形周长等表示轮廓总长度的值的几分之一比较合适。

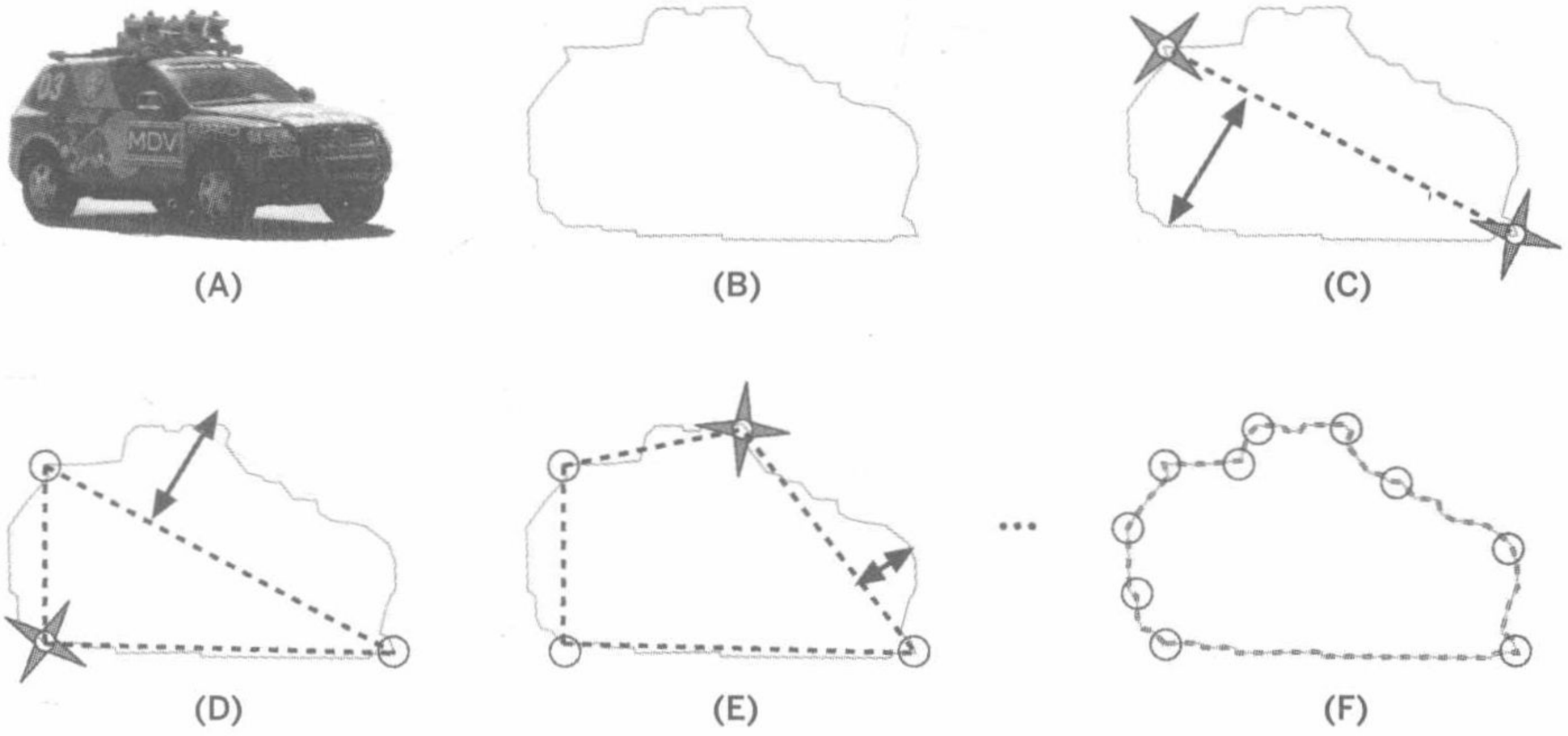


图14-3. cv::approxPolyDP()实现的DP算法的示意图：（A）为原始图像；（B）为提取的轮廓；（C）表示从最远的两个点开始；（D～F）表示其他点的选择过程

几何及特性概括

轮廓处理中经常遇到的另一个任务是计算一些轮廓变化的概括特性。这可能包括长度或其他一些反应轮廓整体大小的量度。另一个有用的特性是轮廓矩（contour moment），可以用来概括轮廓的总形状特性，这部分我们在下一节讨论。以下一些方法对任何形式的点集都适用（包括那些并不代表轮廓的点集）。我们会指出哪些方法只适用于轮廓（如计算弧长），而哪些方法对任何点集都适用（如外包矩形）。

用cv::arcLength()获得长度

子函数cv::arcLength()提取输入轮廓的长度。

```
double  cv::arcLength(
  cv::InputArray  points,      // Array or vector of 2-dimensional points
  bool            closed       // If true, assume link from last to first vertex
);
```

cv::arcLength()的第一个参数代表是轮廓，其形式可以是任何常见的轮廓表示方法（如标准模板库的点向量或二通道数组）。第二个参数closed表示该轮廓是否是闭合的。假如轮廓是闭合的，则参数points中的最后一个点到第一个的距离也算入总弧长中。

cv::arcLength()函数中的points参数必须代表一条曲线，因此它不适用于不代表轮廓的点集。

用cv::boundingRect()获得矩形包围框

当然，长度和面积只是轮廓的简单特性。描述轮廓的一种最简易的方法是为它加上一个外包围框。最简单的途径是直接计算外包围矩形。这正是cv::boundingRect()函数做的：

```
cv::Rect cv::boundingRect(     // Return upright rectangle bounding the points
  cv::InputArray  points,      // Array or vector of 2-dimensional points
);
```

cv::boundingRect()函数只有一个参数，就是需要计算包围框的曲线。该函数返回一个cv::Rect类型的值，这就是所需的外包围框。

计算外包围框对任何点集都有用，不管这些点是否代表一条曲线。

用cv::minAreaRect()获得最小矩形框

cv::boundingRect()得到的矩形框存在一个问题，cv::Rect只能表现一个四边水平和垂直的矩形。而函数cv::minAreaRect()可以返回一个包围轮廓最小的矩形，这个矩形很可能是倾斜的，如图14-4所示。该函数的参数和cv::boundingRect()的相似。OpenCV的数据类型cv::RotatedRect就是用来表述这样的矩形的。回忆一下该函数的定义：

```
class cv::RotatedRect {
  cv::Point2f center;     // Exact center point (around which to rotate)
  cv::Size2f  size;       // Size of rectangle (centered on 'center')
  float       angle;      // degrees
};
```

因此，若想得到更小的外包围框，可以使用cv::minAreaRect()函数：

```
cv::RotatedRect cv::minAreaRect( // Return rectangle bounding the points
  cv::InputArray  points,        // Array or vector of 2-dimensional points
);
```

同样，参数points可以是任何标准形式的点集，且该函数无论对曲线还是普通点集，都是有用的。

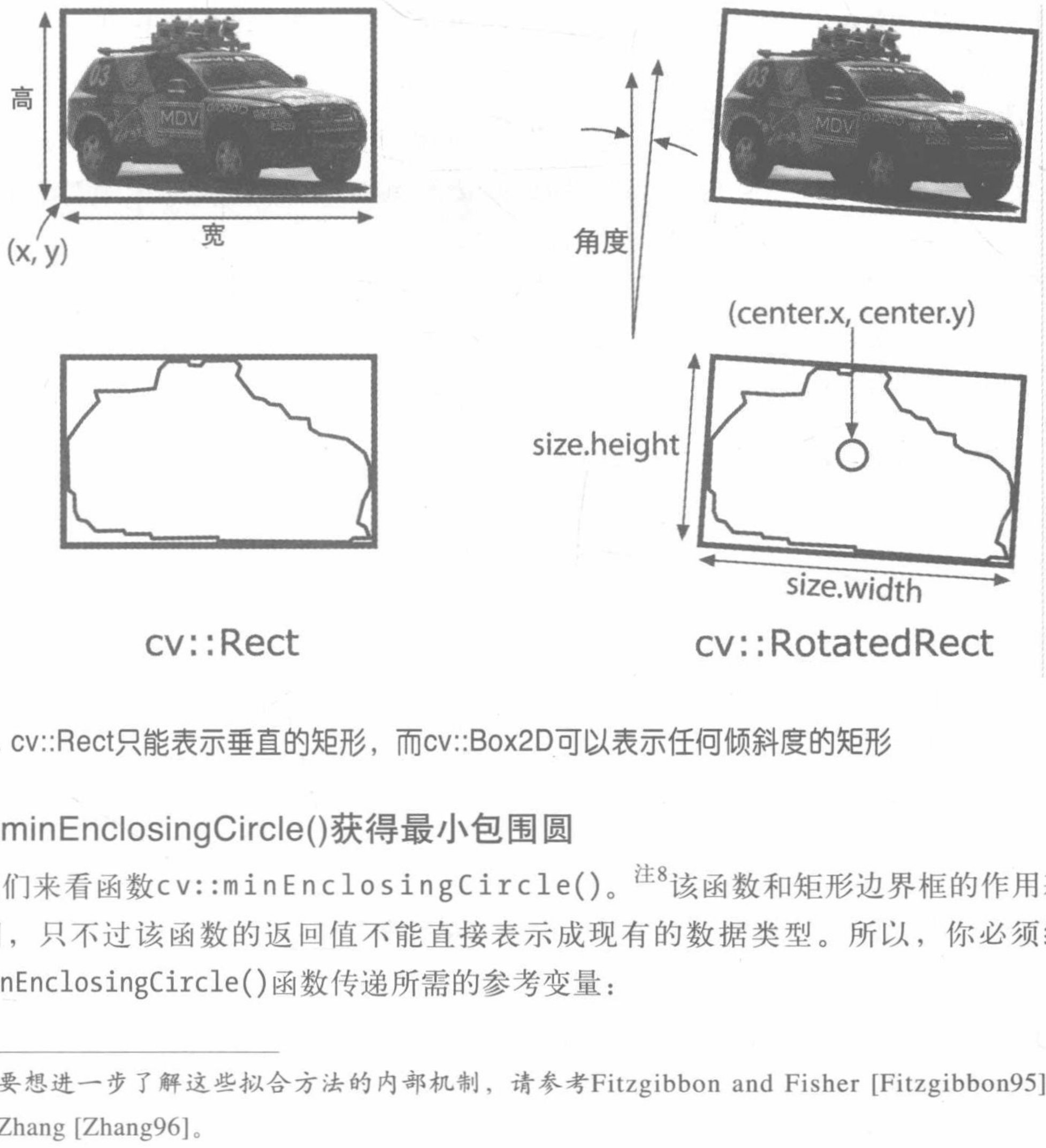


图14-4. cv::Rect只能表示垂直的矩形，而cv::Box2D可以表示任何倾斜度的矩形

用cv::minEnclosingCircle()获得最小包围圆

接着我们来看函数cv::minEnclosingCircle()。[注8]该函数和矩形边界框的作用基本相同，只不过该函数的返回值不能直接表示成现有的数据类型。所以，你必须给cv::minEnclosingCircle()函数传递所需的参考变量：

注8：要想进一步了解这些拟合方法的内部机制，请参考Fitzgibbon and Fisher [Fitzgibbon95]和Zhang [Zhang96]。

```
void  cv::minEnclosingCircle(
   cv::InputArray points,        // Array or vector of 2-dimensional points
   cv::Point2f&   center,        // Result location of circle center
   float&         radius         // Result radius of circle
);
```

输入曲线仍然是一系列点。center和radius参数是你需要传递的，cv::minEnclosing-Circle()函数将返回这两个值。

cv::minEnclosingCircle函数对曲线和其他点集都有用。

用cv::fitEllipse()获得椭圆边框

与最小包围圆一样，OpenCV提供函数来获得一组点集的最佳拟合椭圆边框。

```
cv::RotatedRect cv::fitEllipse(  // Return rect bounding ellipse (#ten-point_contour_
with_the_minimal_enclo)
  cv::InputArray  points         // Array or vector of 2-dimensional points
);
```

cv::fitEllipse()只有一个参数，即一组点集。

初看上去，cv::fitEllipse()函数只是cv::minEnclosingCircle()函数的椭圆化版本。然而，cv::minEnclosingCircle()函数与cv::fitEllipse()存在一些细微差别，前者只简单计算完全包围已有轮廓的最小圆，而后者使用拟合函数返回一个与轮廓最相似的椭圆。这意味着并不是轮廓中的所有点都会被抱在cv::fitEllipse()返回的椭圆中。[注9]该拟合由最小二乘拟合方法算出。

椭圆的拟合结果由cv::RotatedRect结构体返回。给出的矩形正好完全包围椭圆，如图14-5所示。

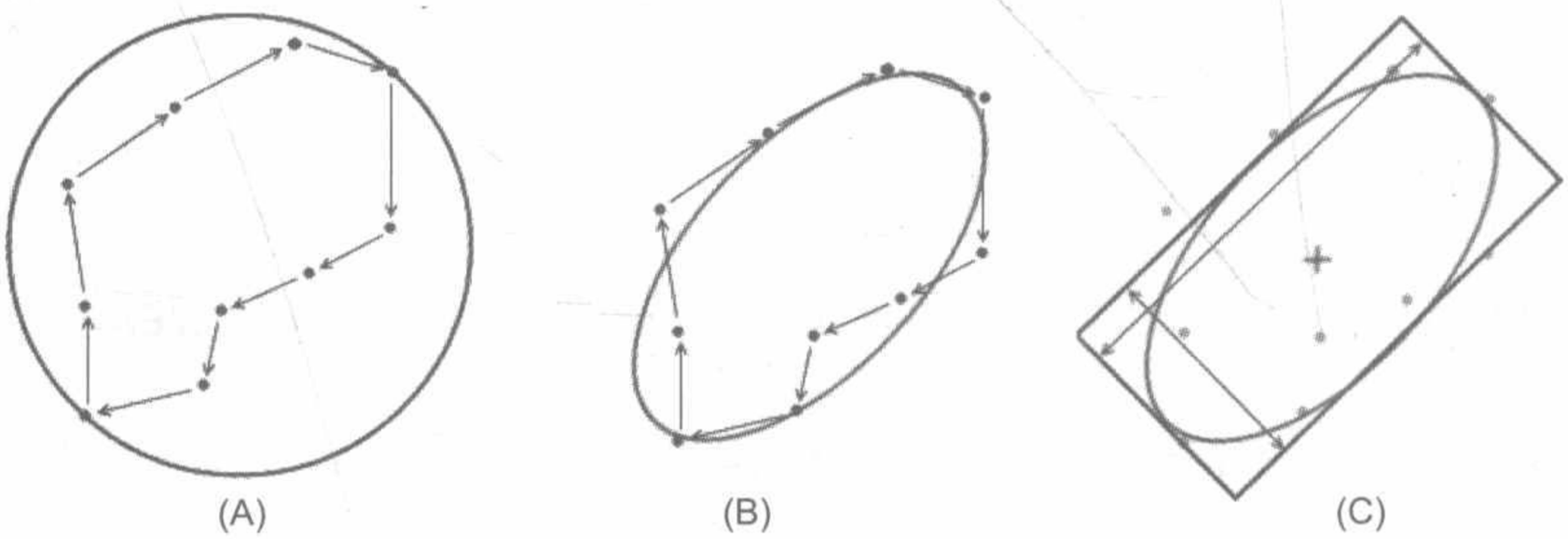


图14-5. (A)10个点的轮廓与最小包围圆；(B)最佳拟合椭圆；(C)OpenCV给出的表示该椭圆的矩形

注9：当然，如果点数很少（或者一些更简单的情况，如所有点共线），有可能所有点都落在椭圆上。但一般而言，一些点会落在椭圆内部，一些点会落在椭圆外部，还有更少的一些点可能落在椭圆上。

用cv::fitLine()获得轮廓的最佳拟合线

很多情况下，你的轮廓确实近似于一条直线，或者更准确地说，是一组带噪声的对直线的采样点。这时，需要解决的问题是，为观察到的点集选择一条最佳拟合线。事实上，需要获得一条最佳拟合直线的理由有很多，如何获得这条直线的方法也多种多样。

我们通过最小化代价函数的方式来拟合，定义如下：

$$cost(\vec{\theta}) = \Sigma_{points:i}\rho(r_i) \text{ , where } r_i = r(\vec{\theta}, \vec{x}_i)$$

其中，$\vec{\theta}$表示定义直线的参数；$\vec{x}_i$表示轮廓中的第i个点；r_i表示由$\vec{\theta}$表示的直线与该点之间的距离。因此，从根本上说，是函数$\rho(r_i)$将不同的可行拟合方法区分开。当$\rho(r_i) = \frac{1}{2}r_i^2$时，代价函数就变成了最小二乘拟合函数，了解基本统计方法的读者对此应该都不陌生。在需要更稳定的拟合方法时（例如，能更恰当地处理异常数据点的拟合方法），那些更为复杂的距离函数很有用。表14-2展示了有效的形式以及OpenCV中相应的用于cv::fitLine()的枚举值。

表14-2：cv::fitLine()函数中di stType参数的有效距离度量

distType	距离度量	
cv::DIST_L2	$\rho(r_i) = \frac{1}{2}r_i^2$	Least-squares方法
cv::DIST_L1	$\rho(r_i) = r_i$	
cv::DIST_L12	$\rho(r_i) = 2 \cdot \left(\sqrt{1 + \frac{1}{2}r^2} - 1\right)$	
cv::DIST_FAIR	$\rho(r_i) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right)$	$C = 1.3998$
cv::DIST_WELSCH	$\rho(r_i) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right)$	$C = 2.9846$
cv::DIST_HUBER	$\rho(r_i) = \begin{cases} \frac{1}{2}r_i^2 & r < C \\ C \cdot \left(r - \frac{C}{2}\right) & r \geq C \end{cases}$	$C = 1.345$

OpenCV中的cv::fitLine()函数原型如下：

```
void cv::fitLine(
  cv::InputArray  points,        // Array or vector of 2-dimensional points
  cv::OutputArray line,          // Vector of Vec4f (2d), or Vec6f (3d)
  int             distType,      // Distance type (表14-2)
  double          param,         // Parameter for distance metric (表14-2)
  double          reps,          // Radius accuracy parameter
  double          aeps           // Angle accuracy parameter
);
```

正如你预料到的，参数points是最常用的，它表示了一组cv::Mat数组或标准模板库向量形式的点集。然而，与本节中讨论过的其他函数相比，cv::fitLine()函数最重要的不同是它既可以处理二维点，也可以处理三维点。这点不同也体现在输出line上。这个值应该是cv::Vec4f类型（对于二维直线）或cv::Vec6f类型（对于三维直线）。

若是后者，则数值中的前一半代表直线方向，后一半代表直线上的一个点。第三个参数distType允许我们选择所用的距离度量。distType的有效值如表14-2所示。param用于为其中一些距离度量提供参数值（见表14-2中的C变量）。该参数可以置为0，这样cv::fitLine()函数将自动为选中的距离度量提供最优值。参数reps和aeps代表你对拟合直线的原点精度要求（x, y, z部分）和角度精度要求（v_x, v_y, v_z部分）这两个参数的常用值是1e-2。

用cv::convexHull()获得轮廓的凸包

很多时候，我们需要通过寻找一个多边形的凸包来简化它。一个多边形或一条轮廓的凸包，指的是一个完全包含原轮廓且只由原轮廓上的点组成的多边形，此多边形的每一处都必须是凸的（例如，多边形上的任意连续三个点内角小于180度）。图14-6为一个凸包的例子。计算凸包的用处有很多。其中特别常见的一项用处是，检测一个点是否落在一个凸多边形内的过程很快，因此在检测点是否落在一个复杂多边形内部之前，往往有必要先检测它是否落在该多边形的凸包内部。

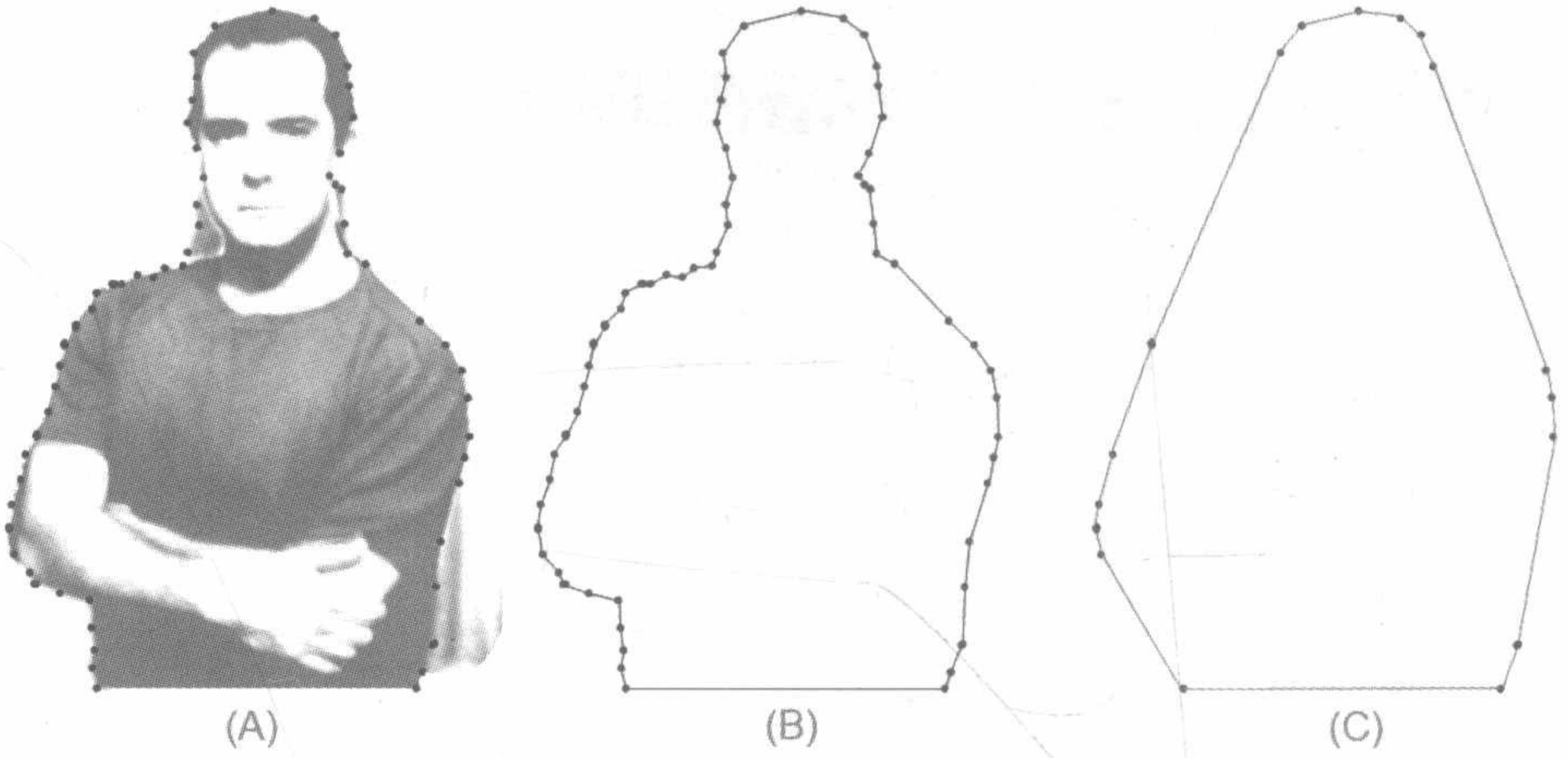


图14-6. 图像（A）转换成轮廓（B）。该轮廓凸包（C）上点更少，几何形状也更简单

OpenCV提供cv::convexHull()函数用于计算轮廓的凸包。

```
void cv::convexHull(
  cv::InputArray  points,                 // Array or vector of 2-d points
  cv::OutputArray hull,                   // Array of points or integer indices
  bool            clockwise    = false, // true='output points will be clockwise'
  bool            returnPoints = true   // true='points in hull', else indices
);
```

cv::convexHull()函数的输入points可以是任何形式的轮廓。参数hull给出返回的凸包。对于这个参数，你有两个选择：可以设置为常规的轮廓类型结构体，

cv::convexHull()会以该结构返回计算得到的凸包上的点。除了提供一组点，另一个选择是（如上述讨论的参数returnPoints）提供一组整数。这时，cv::convexHull()会把每一个属于凸包的点分别与一个索引对应，并在hull中返回这些索引。这样，索引将从0开始，索引值i代表点points[i]。

参数clockwise指明你希望cv::convexHull()如何呈现得到的凸包。假如将clockwise设为真（true），则凸包将按顺时针方向排列；若设为假（false），则将以逆时针方向排列。最后一个参数returnPoints与参数hull中的选择有关。假如points是一个标准模板库向量对象，则此参数会被忽略，因为向量模板中的类型（int对应cv::Point）足以满足需求了。但假如points是cv::Mat类型的数组，那么需要函数返回的是点时，returnPoints必须设为true，而需要函数返回索引时，returnPoints必须设为false。

几何学测试

当你在处理外包围框或其他类型的轮廓多边形时，常常需要做一些简单的几何学检测，如多边形重叠检测或外包围框重叠快速检测。OpenCV提供一些易用的小程序，用于进行这一类几何学检测。

矩形类型中，提供接口用于进行一些重要的矩形几何学测试。例如，cv::Rect类型中的contains()方法可以判断一个输入点是否落在矩形内部。

类似的，包含两个矩形的最小矩形框可以用逻辑或（OR）操作进行（如rect1 | rect2），而计算两个矩形的重叠部分可以用逻辑与（AND）进行（如rect1 & rect2）。可惜，这里并没有适用于cv::RotatedRect的操作符。

对于一般的曲线操作，可以使用库函数。

用cv::pointPolygonTest()检测点是否落在多边形内

第一个几何体工具函数是cv::pointPolygonTest()，可用来检测一个点是否落在一个多边形内（由contour数组表示）。具体而言，假如参数measureDist设为true，函数将返回该点距离最近的轮廓边缘的距离。若点在轮廓内，距离为0；若点在轮廓外，距离将是大于0的整数。如果该参数设为假，返回值将只会是+1、-1、0三种值，分别对应点落在轮廓（或顶点）内部、外部和正好在轮廓（或顶点）上。一如往常，轮廓既可以是标准模板库向量，也可以是大小为$n \times 1$的二通道点数组。

```
double cv::pointPolygonTest(    // Return distance to boundary (or just side)
  cv::InputArray contour,       // Array or vector of 2-dimensional points
  cv::Point2f    pt,            // Test point
  bool           measureDist    // true 'return distance', else {0,+1,-1} only
);
```

用cv::isContourConvex()测试轮廓是否为凸

判断一条轮廓是否为凸轮廓是常见的需求。这样做的理由很多，其中最常见的是许多算法只能用于凸多边形，还有许多算法在多边形为凸时可以大大简化。将任何常见形式的轮廓传给cv::isContourConvex()函数，就能简单测试多边形是否为凸。通常假定传给函数的轮廓是闭合的多边形（例如，认为轮廓中的最后一个点与第一个点是相连的）：

```
bool cv::isContourConvex(       // Return true if contour is convex
  cv::InputArray  contour       // Array or vector of 2-dimensional points
);
```

基于函数实现的性质，传递给cv::isContourConvex()轮廓需要是一个简单的多边形。这意味着轮廓内不能存在自相交。

匹配轮廓与图像

现在，你已经很了解轮廓是什么，也明白如何在OpenCV中处理轮廓对象，接下来我们将谈谈如何发挥它们的实际用途。一个跟轮廓相关的最常用到的功能是如何匹配多条轮廓。我们或许需要比较两条计算好的轮廓，或者比较一条轮廓和一个抽象模板。这两种情况随后都会讨论。

矩

比较两条轮廓最简洁的方法之一是比较它们的轮廓矩。轮廓矩代表了一条轮廓、一幅图像、一组点集的某些高级特征。下面的所有讨论对轮廓、图像、点集都同样适用，简便起见，将它们统称为对象。矩的数值定义如下式：

$$m_{p,q} = \sum_{i=1}^{N} I(x_i, y_i) x^p y^q$$

在上式中，$m_{p,q}$代表对象中所有像素的总和，其中每个像素x,y的像素值都乘以因子$x^p y^q$。在m_{00}时，这个因子等于1。因此若图像为二值图（例如，所有像素都等于0或者1），则m_{00}代表图像上所有值非零的区域。当处理轮廓时，结果是轮廓的长度。[注10]当处理点集

注10：纯数学论者可能不同意m_{00}为轮廓边界长度，他们认为应该是面积。但此处我们处理的是一个轮廓，而不是一个被填充的多边形，因此多边形的边长和边界上的像素在离散的像素空间里是相同的（至少在像素空间里是跟距离相关的）。同样，也有计算cv::Array图像的矩的函数。对于图像，m_{00}一般对应非0像素的面积。然而事实上，这个区别并不只有学术意义。假如一条轮廓由一系列顶点组成，那么直接采用公式计算轮廓长度得到的面积，与先把轮廓栅格化（比如使用cv::drawContours()函数）再计算得到的面积并不完全相同。尽管两者在分辨率无限大的情况下应该趋近于同一个值。

时，它代表的仅是点的数量。经过思考，你应该能理解对于同一张二值图，将m_{10}和m_{01}相加再除以m_{00}，能得到整个对象的平均x值和y值。“矩”这个名称与数值中的矩有关，而更高阶的矩则与数值分布（如面积、均值、方差等）中的矩有关。从这个意义上来说，可以将非二值图的矩看作是特殊情况下的二值图的矩，只不过其中每个像素点上存在多个值。

用cv::moments()计算矩

用于计算上述矩的函数如下：

```
cv::Moments cv::moments(                // Return structure contains moments
  cv::InputArray points,                // 2-dimensional points or an "image"
  bool           binaryImage = false // false='interpret image values as "mass"'
)
```

第一个参数points代表我们需要计算的轮廓。第二个参数binaryImage告诉OpenCV是否将输入图像理解为二值图。参数points既可以是二维数组（函数将把它视作图像），也可以是$N \times 1$或$1 \times N$的二通道数组点集，或是标准模板库cv::Point对象。当points是点集时，cv::moments函数将把这些点看成轮廓中的顶点，将整个点集当做一条轮廓，而非把它们作为各自离散的点。[注11]参数points的意义在于，当设为真时，所有的非零像素的像素值都会被视为1，而非它们原来的真值。图像经过阈值操作后，所有的非零值可能都是255，这个时候上述操作特别有用。cv::moments()函数返回的是cv::Moments对象。该对象定义如下：

```
class Moments {
public:
    double m00;                    //   zero order moment                  (x1)
    double m10, m01;               //  first order moments                 (x2)
    double m20, m11, m02;          // second order moments                 (x3)
    double m30, m21, m12, m03;     //  third order moments                 (x4)
    double mu20, mu11, mu02;       // second order central moments         (x3)
    double mu30, mu21, mu12, mu03; //  third order central moments         (x4)
    double nu20, nu11, nu02;       // second order Hu invariant moments    (x3)
    double nu30, nu21, nu12, nu03; //  third order Hu invariant moments    (x4)
  Moments();
  Moments(
    double m00,
    double m10, double m01,
    double m20, double m11, double m02,
    double m30, double m21, double m12, double m03
  );
  Moments( const CvMoments& moments ); // convert v1.x struct to C++ object
  operator CvMoments() const;          // convert C++ object to v1.x struct
}
```

注11：当你需要处理一个点集而非一条轮廓时，创建一幅包含这些点的图像是最简便的方法。

直接调用cv::moments()计算所有三阶的矩（例如，$p + q \leq 3$的矩），同时也会计算中心距和归一化中心距。这些我们稍后会讨论。

再论矩

刚刚描述的矩计算给出了一些轮廓的简单属性，可以用来比较两个轮廓。但是在很多实际使用中，刚才这种方法计算得到的矩并不是做比较时最好的参数。具体说来，目前为止我们所讨论的矩，对于形状相同但大小不同，或者存在相互位移或旋转的两条轮廓，会有不同的值。

中心距的位移不变性

给定一条轮廓或一幅图像，不管轮廓出现在图像中的哪个地方，m_{00}矩显然都是相同的，那些更高阶的矩则会改变。考虑之前我们用对象中像素x坐标均值定义过的m_{10}矩显然，当同样的物体出现在不同的地方时，对象的x坐标均值也会不同。尽管粗看不明显，但体现了物体铺开范围的二阶中心矩同样具有平移不变性。[注12]在许多情况下，我们需要利用这些矩，将可能出现在图像中任意位置的物体与参考图像中可能出现在另外位置的参考物体进行比较。

这一问题的解决方法是计算中心矩。中心矩常用$\mu_{p,q}$标注，定义如下：

$$\mu_{p,q} = \sum_{i=1}^{N} I(x_i, y_i)(x - \bar{x})^p (y - \bar{y})^q$$

其中：

$$\bar{x} = \frac{m_{10}}{m_{00}}$$

且：

$$\bar{y} = \frac{m_{01}}{m_{00}}$$

当然，很明显$\mu_{00}=m_{00}$（因为含有p和q的项合并消失了），并且μ_{10}和μ_{01}中心矩都等于0。更高阶的矩与非中心矩相同，只不过是相对物体质心（或基于物体质心坐标系）测量得到的。由于这些值是相对质心得到的，所以它们也不随物体在图像中的绝对位置改变而改变。

注12：若你不太理解这些数学术语，“平移不变性”指的是物体的某些计算量不随物体在图像中的移动（例如平移）而变化。类似的，“旋转不变性”指的是物体的某些计算量不随物体在图像中的旋转而变化。



你可能注意到了，在cv::Moments对象中并不含mu00、mu10、mu01元素。这只是因为这些值是冗余的（例如，mu00 = m00，而mu10 = mu01 = 0）。归一化中心矩也相同（除了nu00 = 1，且nu10和nu01都为0）。存储它们将耗费多余的内存空间，因此它们并不出现在结构体定义中。

归一化中心矩的缩放不变性

中心矩已经能够帮助我们比较图像中不同位置的两个物体。同样，比较图像中两个除了尺度大小其他全部相同的物体也有重要意义。有时候，这样做是为了在图像中寻找本来就尺度不一的物体（例如熊）。但大部分时候还是因为我们不知道物体距离开始拍摄图像的相机有多远。

中心矩通过减去均值而获得平移不变性，类似的，归一化中心矩通过除以物体的总尺寸而获得缩放不变性。归一化中心矩定义式如下：

$$\nu_{p,q} = \frac{\mu_{p,q}}{m_{00}^{(\frac{p+q}{2}+1)}}$$

这个略显复杂的公式表明，归一化中心矩就是中心矩加上一个归一化因子，这个因子是物体面积的幂（对于更高阶的矩，这个幂会更大）。

在OpenCV中，并没有特定的函数用于计算归一化矩，因为cv::Moments()在计算标准矩和中心矩时会自动计算归一化矩。

Hu不变矩的旋转不变性

*Hu*不变矩是归一化中心矩的线性组合。通过组合不同的归一化中心矩，我们可以得到一个反应图像不同特征的不变函数，这个函数不随尺度、旋转、镜面映射（除了h_1）变化而变化。

为了完整性，这里我们给出Hu矩的实际定义：

$$h_1 = \nu_{20} + \nu_{02}$$

$$h_2 = (\nu_{20} + \nu_{02})^2 + 4\nu_{11}^2$$

$$h_3 = (\nu_{30} - 3\nu_{12})^2 + (3\nu_{21} - \nu_{03})^2$$

$$h_4 = (\nu_{30} + \nu_{12})^2 + (\nu_{21} + \nu_{03})^2$$

$$h_5 = (\nu_{30} - 3\nu_{12})(\nu_{30} + \nu_{12})\left[(\nu_{30} + \nu_{12})^2 - 3(\nu_{21} + \nu_{03})^2\right]$$
$$+ (3\nu_{21} - \nu_{03})(\nu_{21} + \nu_{03})\left[3(\nu_{30} + \nu_{12}) - (\nu_{21} + \nu_{03})\right]$$

$$h_6 = (\nu_{20} - \nu_{02})\left[(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2\right] + 4\nu_{11}(\nu_{30} + \nu_{12})(\nu_{21} + \nu_{03})$$

$$h_7 = (3\nu_{21} - \nu_{03})(\nu_{30} + \nu_{12})\left[(\nu_{30} + \nu_{12})^2 - 3(\nu_{21} + \nu_{03})^2\right]$$
$$- (\nu_{30} - 3\nu_{12})(\nu_{21} + \nu_{03})\left[3(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2\right]$$

从图14-7和表14-3中，我们可以了解Hu矩是如何起作用的。首先，随着阶数增加，我们发现Hu矩变得越来越小。这并不意外，因为根据定义，更高阶的Hu矩由多个归一化因子的高阶幂计算得到。由于这些因子都小于1，所以指数越大，获得的数值就越小。

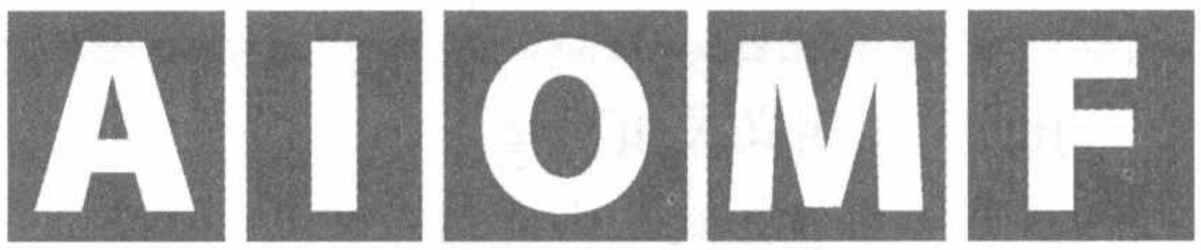


图14-7. 5个简单字母的图像。观察它们的Hu矩可以获得一些直观理解

表14-3：在图14-7中所示的5个字母的Hu矩的值

	h_1	h_2	h_3	h_4	h_5
A	2.837e–1	1.961e–3	1.484e–2	2.265e–4	–4.152e–7
I	4.578e–1	1.820e–1	0.000	0.000	0.000
O	3.791e–1	2.623e–4	4.501e–7	5.858e–7	1.529e–13
M	2.465e–1	4.775e–4	7.263e–5	2.617e–6	–3.607e–11
F	3.186e–1	2.914e–2	9.397e–3	8.221e–4	3.872e–8

需要特别注意的是*I*，它对于180度旋转和镜面反射都是对称的，它的h_3到h_7矩都是0；而*O*具有同样的对称性，所有的Hu矩都是非零的（但事实上，其中两个矩接近0）。我们把这个问题留给读者思考。你可以仔细查看字符图像，比较不同的Hu矩，以获得对这些矩的直观感受。

用cv::HuMoments()计算Hu不变矩

其他的矩都能用cv::moments()函数计算，Hu矩的计算用到了cv::HuMoments()函数。它接受cv::moments()函数返回的cv::Moments对象作为输入，并返回7个Hu矩的值。函数定义如下：

```
void cv::HuMoments(
  const cv::Moments& moments, // Input is result from cv::moments() function
  double*            hu       // Return is C-style array of 7 Hu moments
```

```
);
```

cv::HuMoments()函数接受cv::Moments对象作为输入，并返回一个指针，指向一个预分配好内存的C语言风格数组，数组中存储了7个Hu矩值。

使用Hu矩进行匹配

自然，我们想要使用Hu矩比较两个物体，并判定它们是否相似。当然，对“相似”的定义可能有很多。为了使比较过程变得简单，OpenCV的函数cv::matchShapes()允许我们简单提供两个物体，然后计算它们的矩，并根据我们提供的标准进行比较。

```
double  cv::MatchShapes(
  cv::InputArray object1,      // First array of 2D points or cv:U8C1 image
  cv::InputArray object2,      // Second array of 2D points or cv:U8C1 image
  int            method,       // Comparison method (表14-4)
  double         parameter = 0 // Method-specific parameter
);
```

这些对象既可以是灰度图，也可以是轮廓。不管是哪一类，cv::matchShapes()函数都会在比较之前把矩算好。cv::matchShapes()函数中使用的method如表14-4所示。

表14-4：cv::matchShapes()函数中使用的匹配方法

方法名称	cv::matchShapes()函数返回值
cv::CONTOURS_MATCH_I1	$\Delta_1 = \sum_{i=1..7} \left\| \frac{1}{\eta_i^A} - \frac{1}{\eta_i^B} \right\|$
cv::CONTOURS_MATCH_I2	$\Delta_2 = \sum_{i=1..7} \left\| \eta_i^A - \eta_i^B \right\|$
cv::CONTOURS_MATCH_I3	$\Delta_1 = \sum_{i=1..7} \left\| \frac{\eta_i^A - \eta_i^B}{\eta_i^A} \right\|$

在表中，η_i^A和η_i^B的定义如下：

$$\eta_i^A = \text{sign}(h_i^A) \cdot \log(h_i^A)$$

且：

$$\eta_i^B = \text{sign}(h_i^B) \cdot \log(h_i^B)$$

在式子中，h_i^A和h_i^B分别是图*A*和图*B*的Hu矩。

表14-3中定义的三个常量，每个都用了不同的对比度量标准计算方法。这个度量标准最终决定了cv::matchShapes()的返回值最后一个参数parameter现在不能使用，因此我们可以把它设为默认值0。今后出现的度量标准可能需要用户提供的参数，因此设置了这个参数。

利用形状场景方法比较轮廓

自20世纪80年代以来，用矩来比较形状一直是经典方法，但现在有了更好的现代算法。OpenCV 3中有一个专有模块shape，该模块可用于实现一些算法，特别是形状场景算法（Shape Context）[Belongie02]。

这一形状模块仍在开发中，所以我们这里只简单介绍它的高层次结构和一些已经可以使用的部分。

形状模块的结构

形状模块围绕一个抽象概念cv::ShapeDistanceExtractor构成。对于那些能比较两个（或多个）形状且返回用于衡量形状之间差值的“距离”度量标准的所有仿函数，这一抽象类型都是适用的。我们选择距离这个词，是因为许多情况下，形状之间的差值与距离的很多属性相似，如它们都永远非负，又如当两个形状一模一样时该值等于0。cv::ShapeDistanceExtractor中的重要定义如下：

```
class ShapeContextDistanceExtractor : public ShapeDistanceExtractor {
  public:
  ...
virtual float computeDistance( InputArray contour1, InputArray contour2 ) = 0;
};
```

从这个类衍生出不同的形状距离提取器。在我们讨论其中可用的两种提取器之前，我们需要先简要考虑一下另外两种抽象仿函数类型：cv::ShapeTransformer和cv::HistogramCostExtractor。

```
class ShapeTransformer : public Algorithm {

public:
  virtual void estimateTransformation(
    cv::InputArray       transformingShape,
    cv::InputArray       targetShape,
    vector<cv::DMatch>& matches
  ) = 0;

virtual float applyTransformation(
  cv::InputArray       input,
  cv::OutputArray      output       = noArray()
) = 0;

virtual void warpImage(
  cv::InputArray       transformingImage,
  cv::OutputArray      output,
  int                  flags        = INTER_LINEAR,
  int                  borderMode   = BORDER_CONSTANT,
```

```
    const cv::Scalar&   borderValue = cv::Scalar()
  ) const = 0;
};

class HistogramCostExtractor : public Algorithm {

public:
  virtual void  buildCostMatrix(
    cv::InputArray      descriptors1,
    cv::InputArray      descriptors2,
    cv::OutputArray     costMatrix
  )                                                    = 0;

  virtual void  setNDummies( int nDummies )          = 0;
  virtual int   getNDummies() const                  = 0;

  virtual void  setDefaultCost( float defaultCost ) = 0;
  virtual float getDefaultCost() const               = 0;
};
```

类*shape transformer*广泛用于表示一类算法。这些算法将一个点集重映射到另一个点集，或更通用地，将一张图像重映射到另一张图像上。放射变换和投射变换（我们已讨论过）也可以实现成形状变化器，正如一种重要的变化器，薄板样条变化器（thin plate spline transform）。后者得名于金属薄板物理模型，它从根本上解决了移动金属薄板上的一些“控制点”到不同位置时产生的映射问题。随着控制点的变化，金属薄板随之发生形变，由此产生的稠密映射就是算法得到的变换式。事实证明，这是一种广泛适用的结构，在图像配准和形状匹配中也有很多应用。在OpenCV中，该算法由函子cv::ThinPlateSplineShapeTransformer实现。

由直方图代价提取算子（histogram cost extractor）推广得到我们先前在陆地移动距离（EMD，Earth Mover Distance）中见过的结构体。在EMD中，我们想要得到从一个土堆向另一个土堆铲土的代价值。在一些情况下，这个代价值是常数，或是与铲土的距离线性相关。但在大部分情况下，我们希望代价值与转移的土的数量有关。EMD算法有自己的接口（遗留接口），但cv::HistogramCostExtractor基类及其派生类使我们能够解决更一般的问题。表14-5展示了一系列派生的代价提取算子。

表14-5：从cv::HistogramCostExtractor抽象类派生的代价提取算子

派生的提取类	提取算子采用的代价函数
cv::NormHistogramCostExtractor	从L2或其他范数计算的代价值
cv::ChiHistogramCostExtractor	用卡方距离进行比较
cv::EMDHistogramCostExtractor	如EMD代价矩阵一样，采用L2范数计算的代价
cv::EMDL1HistogramCostExtractor	如EMD代价矩阵一样，采用L1范数计算的代价

上述每一种提取算子和变换算子，都分别对应一个叫createX()的工厂方法，其中X代表仿函数名称，如cv::createChiHistogramCostExtractor()。掌握了这两种类型后，现在我们可以看看形状距离提取算子的一些特例。

形状场景距离提取

如本节开头提过的，OpenCV 3中有形状场景距离（shape context distance）[Belongie02]的具体实现，包含在cv::ShapeDistanceExtractor派生的仿函数中。cv::ShapeContextDistanceExtractor方法在函数实现中采用了形状变换和直方图代价提取仿函数。

```
namespace cv {

  class ShapeContextDistanceExtractor : public ShapeDistanceExtractor {

    public:
    ...
    virtual float computeDistance(
      InputArray contour1,
      InputArray contour2
    ) = 0;
  };

  Ptr<ShapeContextDistanceExtractor> createShapeContextDistanceExtractor(
    int   nAngularBins                          = 12,
    int   nRadialBins                           = 4,
    float  innerRadius                          = 0.2f,
    float  outerRadius                          = 2,
    int   iterations                            = 3,
    const Ptr<HistogramCostExtractor> &comparer
                                                = createChiHistogramCostExtractor(),
    const Ptr<ShapeTransformer>       &transformer
                                                = createThinPlateSplineShapeTransformer()
  );

}
```

在本质上，形状场景算法分别算出两个（或多个）待比较的形状的特征表达。每个特征表达式都考虑了形状边界上的一部分点集。对于每一个采样点，它都建立了特定的直方图，用于反应在极坐标中从该点观察到的形状轮廓。所有的直方图大小一致（nAngularBins * nRadialBins）。形状#1中的点p_i和形状#2中的点q_j都采用经典的卡方距离进行比较。接着，算法计算计算点与点之间（p和q）的最优等比联系，从而使总卡方距离和最小。该算法并不是速度最快的，计算代价矩阵需要N*N*nAngularBins*nRadialBins的复杂度，其中N是边界点集中的采样子集。但是它能给出较好的结果，如示例14-4所示。

示例14-4：使用形状场景距离提取算子

```
#include "opencv2/opencv.hpp"
#include <algorithm>
#include <iostream>
#include <string>

using namespace std;
using namespace cv;

static vector<Point> sampleContour( const Mat& image, int n=300 ) {

vector<vector<Point> > _contours;
vector<Point> all_points;
findContours(image, _contours, RETR_LIST, CHAIN_APPROX_NONE);
for (size_t i=0; i <_contours.size(); i++) {
  for (size_t j=0; j <_contours[i].size(); j++)
    all_points.push_back( _contours[i][j] );

// If too little points, replicate them
//
int dummy=0;
for (int add=(int)all_points.size(); add<n; add++)
  all_points.push_back(all_points[dummy++]);

// Sample uniformly
random_shuffle(all_points.begin(), all_points.end());
vector<Point> sampled;
  for (int i=0; i<n; i++)
    sampled.push_back(all_points[i]);
  return sampled;
}

int main(int argc, char** argv) {

  string path    = "../data/shape_sample/";
  int indexQuery = 1;

  Ptr<ShapeContextDistanceExtractor> mysc = createShapeContextDistanceExtractor();

  Size sz2Sh(300,300);
  Mat img1=imread(argv[1], IMREAD_GRAYSCALE);
  Mat img2=imread(argv[2], IMREAD_GRAYSCALE);
  vector<Point> c1 = sampleContour(img1);
  vector<Point> c2 = sampleContour(img2);
  float dis = mysc->computeDistance( c1, c2 );
  cout << "shape context distance between " <<
     argv[1] << " and " << argv[2] << " is: " << dis << endl;

  return 0;

}
```

若想学习比示例14-4更高级的例子，请参考OpenCV 3框架中的*...samples/cpp/shape_example.cpp*。

Hausdorff距离提取算子

与形状场景距离类似，Hausdorff距离通过cv::ShapeDistanceExtractor接口衡量形状相异度。在本书中，我们这样定义Hausdorff距离[Huttenlocher93]：对图像中的每一个点，寻找其离最近点的距离。这样的距离中最大的成为Hausdorff直接距离。Hausdorff距离是两个Hausdorff直接距离中更大的那个。（注意，Hausdorff直接距离本身是非对称的，而Hausdorff距离显然从结构上就是对称的。）在式子中，Hausdorff距离$H()$根据两个集合A和B之间的Hausdorff直接距离$h()$来定义，如下：[注13]

$$H(A, B) = maxx(h(A, B), h(B, a))$$

其中：

$$h(A, B) = \underset{a\varepsilon A}{maxmin} \| \ a - b \|$$

在这里，$\| \ \cdot \|$表示点A和点B的某种范数（通常是欧氏距离）。

本质上，Hausdorff距离衡量的是两个形状中最糟糕的一对点。Hausdorff距离提取算子可以由工厂方法cv::createHausdorffDistanceExtractor()创建：

```
cv::Ptr<cv::HausdorffDistanceExtractor> cv::createHausdorffDistanceExtractor(
  int   distanceFlag = cv::NORM_L2,
  float rankProp     = 0.6
);
```

记住，返回的cv::HausdorffDistanceExtractor和形状场景提取算子拥有一样的接口，因此它由cv::computeDistance()方法调用。

小结

在本章中，我们学习了二维点序列组成的轮廓。这些序列既可以表示成二维点对象的标准模板库向量（如cv::Vec2f），也可以表示成$N \times 1$维的双通道数组，还可以表示成$N \times 2$维的单通道数组。这些序列可以用于表示图像中的轮廓，并且OpenCV库中包含许多特征来帮助我们构造和操作这些轮廓。

轮廓在表达图像的空间划分或分割时通常很有用。在本书中，OpenCV库提供了许多用于比较不同空间、测试空间性质的工具，如凸性、矩、轮廓与点之间的关系等。最后，

注13：请原谅我们使用$H()$表示Hausdorff距离，用$h()$表示Hausdorff直接距离，尽管前几页中我们用h来表示Hu不变矩。

OpenCV还提供许多方法用于匹配轮廓和形状。我们考察了用于匹配的一些特征，既包括经典特征，也包括最新的基于形状距离提取算子接口的特征。

练习

1. 在一条含有N个点的封闭轮廓中，我们可以通过比较每个点与其他点的距离，找出最外层的点。

   a. 这样一个算法的复杂度是多少？

   b. 怎样用更快的速度完成这个任务？

2. 一张4×4的图像中能包含的最大闭合轮廓有多大？轮廓面积是多少？

3. 不使用`cv::isContourConvex()`函数，描述一个可以用来判断一条闭合轮廓是否为凸轮廓的算法。

4. 描述能够完成下列任务的算法：

   a. 判断一个点是否在一条线上。

   b. 判断一个点是否落在一个三角形内。

   c. 不使用`cv::pointPolygonTest()`函数，判断一个点是否在一个多边形内。

5. 用PowerPoint或类似程序，在黑色背景上画一个半径为20的白色的圆（圆的周长是$2\pi 20 \approx 125.7$）。将画的图保存为图像。

   a. 读入该图像，将其转化为灰度图，阈值化，然后寻找它的轮廓。轮廓长度是多少？上一题的结果是否与计算得到的长度相同（四舍五入）？

   b. 将125.7作为轮廓的基长度，分别用基长度的90%、66%、33%、10%作为参数运行`cv::approxPolyDP()`函数。寻找轮廓长度，并画出结果。

6. 假设我们正在开发一个瓶子检测算子，希望能创造一个“瓶子”特征。我们有许多瓶子的图像，在这些图像上进行分割和轮廓寻找都很容易，但图中的瓶子都是旋转过的，且大小不一。我们可以画出轮廓，找到Hu矩，以获得一个不变的瓶子特征向量。至此一切顺利，但我们是需要画出填充的轮廓，还是只需画出轮廓线即可？解释你的答案。

7. 在习题6中使用`cv::moments()`提取瓶子轮廓矩时，我们应该如何设置`isBinary`标志位？解释你的答案。

8. 使用在讨论Hu矩时用到的字母形状。通过对这些形状进行不同角度的旋转和尺度缩放，或二者组合，得到不同的图像。描述哪些Hu特征对应旋转、哪些对应缩放、哪些二者都对应。

9. 在Google图像中搜索“ArUco markers”，选择较大的一些图像。

   a. 矩适用于寻找ArUco图像吗？

   b. 矩或Hu特征适用于读取ArUco码吗？

   c. `cv::matchShapes()`函数适用于读取ArUco码吗？

10. 在PowerPoint（或其他绘图程序）中创造一个形状，并保存为图像。将该物体进行缩放、旋转、旋转缩放并分别储存为图像。用`cv::matchShapes()`函数比较它们。

11. 修改形状场景例子或修改OpenCV 3中的*shape_example.cpp*例子，用Hausdorff距离度量代替形状场景度量。

12. 获得5张含有不同手势的图像。在拍照时，穿上黑色外套或者彩色手套，以便选择算法能够找到手的轮廓。

    a. 尝试采用`cv::matchShapes()`识别手势。

    b. 尝试采用`cv::computeDistance()`识别手势。

    c. 哪个方法更好？为什么？

第15章

背景提取

背景提取概述

由于方法简单并且在很多时候相机位置是固定的，所以背景提取（又名“背景差分”）在很多应用中，特别是在视频安全保卫中，仍然是一个关键的图像处理操作。Toyama, Krumm, Brummitt和MEyers在[Toyama99]中对很多背景提取方法做了比较。为了实现背景提取，首先要“学习”得出一个背景的模型。

一旦背景的模型被学到，我们将用它和当前的画面进行对比，并且从其中将背景减除。减除后剩余的物体可以近似认为是前景物体。

当然，随着应用场景的变化，“背景”本身就是一个不是特别容易明白的概念。例如，当观察高速公路时，我们可能会将平均车流视为背景。通常情况下，背景被认为是场景中静态的或是周期性移动的并在整个观察过程中保持静态或是周期性移动的部分。背景可能包含随时间而变的，例如树枝在早晚挥舞，而在中午静止不动。我们可能会处理两类典型但明显不同的场景，即室内和室外场景。我们希望能找到合适的背景提取方法，在这两个场景下都工作良好。

在这一章中，我们首先讨论典型背景提取模型的缺点，进而引出高层场景模型的讨论。在其中，我们给出了一种方法，它在大多数光线变化不剧烈的室内场景中快速处理静态背景。接下来我们给出一个虽然稍慢但对室内和室外都能处理的“codebook”的方法，该方法既可以处理周期性的移动（例如随风飘舞的树枝），也可以处理光线缓慢或是周期性变化的场景。这个方法也可以在偶有前景物体移过时进行背景学习。接下来我们讨论如何利用连通分量（在第12章和第14章中首次出现）来对检测出的前景物体进行清

理。接下来我们会给出快速方法和codebook方法的对比。最终我们以OpenCV中实现的两个现代的背景提取算法的讨论来结束本章。这些算法使用了本章中给出的原则，但做出了一些扩展，并在实现细节上进行了调整使其更适合实际使用。

背景提取的缺点

虽然这里提到的背景提取方法对简单场景工作良好，但它们有一个经常难以满足的假设：图像中每个像素的行为和其他像素的行为在统计上是独立的。值得注意的是，我们这里讨论的方法是从单个独立的像素的变化中学习背景而不考虑其邻域内的像素。为了对其邻域内的像素加以考虑，我们可以学习一个“多部分模型”，例如我们可以对独立像素模型稍加扩展，使其包含对其相邻像素的亮度的临时信息。这时，我们使用邻域内像素的亮度来判别邻域像素是相对较亮还是相对较暗。接下来对每个单独的像素学习两个模型：一个邻域内像素较暗的模型，一个邻域内像素较亮的模型。这样，我们得到了一个考虑邻域场景的模型。但同时这也意味着要花两倍内存和两倍计算时间，因为我们需要分别对邻域内像素亮和像素暗存储不同的值。我们同样需要两倍的数据来拟合这个两状态模型。我们可以将“高”和“低”场景的概念推广到单个像素和周围像素强度的多维直方图中，并可能通过几个步骤来完成更复杂的操作。当然，这种更丰富的空间和时间模型还需要更多内存、更多数据样本和更多计算资源。[注1]

由于会引入额外的开销，所以要尽量避免使用更复杂的模型。在像素独立假设不满足时，我们经常选择那些可以更高效利用我们手里的资源的方法，即对结果中的*false-positive*像素进行清理。这种清理工作经常使用图像处理中的算子（经常是cv::erode()，cv::dilate()和cv::floodFill()）来删去杂点。我们先前在讨论如何从带噪音的数据中寻找最大的紧[注2]联通分量时（第10章）已经对这种操作进行了讨论。在这一章中，我们同样会使用联通分量的技巧，因此从现在开始，我们只考虑满足像素独立变化假设的情况。

场景建模

如何定义前景和背景？如果我们正在观察一个停车场，这时一辆车驶进一个停车位，那么这辆车此时显然是一个新的前景物体，但它应该始终被视为前景物体吗？考虑一个被移动的垃圾桶，它会在两个位置被视为前景物体：它的新位置和它在老位置留下的“空

注1：当我们希望计算机从数据中“学习”到东西时，实际上最先需要克服的困难是如何获取足够多的数据。使用的模型越复杂，模型的表达能力就越强，因此也就需要越多的数据进行拟合。我们会在第20章重新讨论这个问题。

注2：这里我们使用了数学意义上的“紧”，它和大小无关。

洞”，那么该如何区分这两个位置？而且，这个垃圾桶（和它留下的“空洞”）应该被视为前景多长时间呢？如果我们对一个黑暗的房间建模，这时有一个人突然打开房间里的灯，是否此时整个房间都应被视为前景物体？为了回答这些问题，我们需要一个高层的“场景”模型，在其中定义一个多层的从前景到背景的结构，同时需要一个根据时间将前景物体逐步降级为背景物体的方法。我们也需要检测何时场景发生全局性的变化并在此时创建新的场景模型。

一般来说，一个场景模型应该包括多层结构，以将“新的前景”物体降级为老的前景物体，进一步降级为背景。同时也可能需要使用一些移动监测方法来对移动物体的“正”位置（它的新位置）和它的“负”位置（它的老位置，即“空洞”）加以区分。

这样一个新的前景物体会被置于“新的前景”层同时被标记为“正”物体或是一个空洞。在没有前景物体的区域，我们可以持续更新背景模型。如果一个前景物体一段时间没有移动，它会被降级为“老的前景物体”，这时算法开始学习它的像素的一些统计量，直到最后学到的模型和背景模型结合，物体成为背景。

对于全局性的改变，例如点亮屋子里的灯，我们可能会使用全局帧间差分。例如，如果同时又有很多像素发生了变化，我们会将这次改变视为全局性的而非局部的变化，进而我们会开始为此学习一个全新的模型。

像素

在开始对像素变化进行建模之前，我们首先研究一下图片中的像素是怎样随时间变化的。考虑一台指向窗外随风摆动的树的相机，图15-1给出了相机拍摄的画面中一条给定的线上的像素在60帧是如何随着时间变化的。我们希望能对这种波动进行建模。但这之前我们跑个题来讨论一下怎么从图片中对这样一条线进行采样，因为这个技巧在创建特征和调试中很有用。

因为在很多时候我们都会对图片上的任意一条线上的像素进行采样，OpenCV提供了一个对象line iterator来简单完成这个操作，这在第6章有所提及。直线迭代器cv::LineIterator实例化后可以按顺序向我们提供一条直线上所有点的信息。

我们首先要做的是实例化一个直线迭代器对象，我们用cv::LineIterator构造函数来做：

```
cv::LineIterator::LineIterator(
  const cv::Mat& image,                // Image to iterate over
  cv::Point      pt1,                  // Start point for iterator
  cv::Point      pt2,                  // End point for iterator
  int            connectivity  = 8,    // Connectivity, either 4 or 8
  int            left_to_right = 0     // 1='fixed iteration direction'
);
```

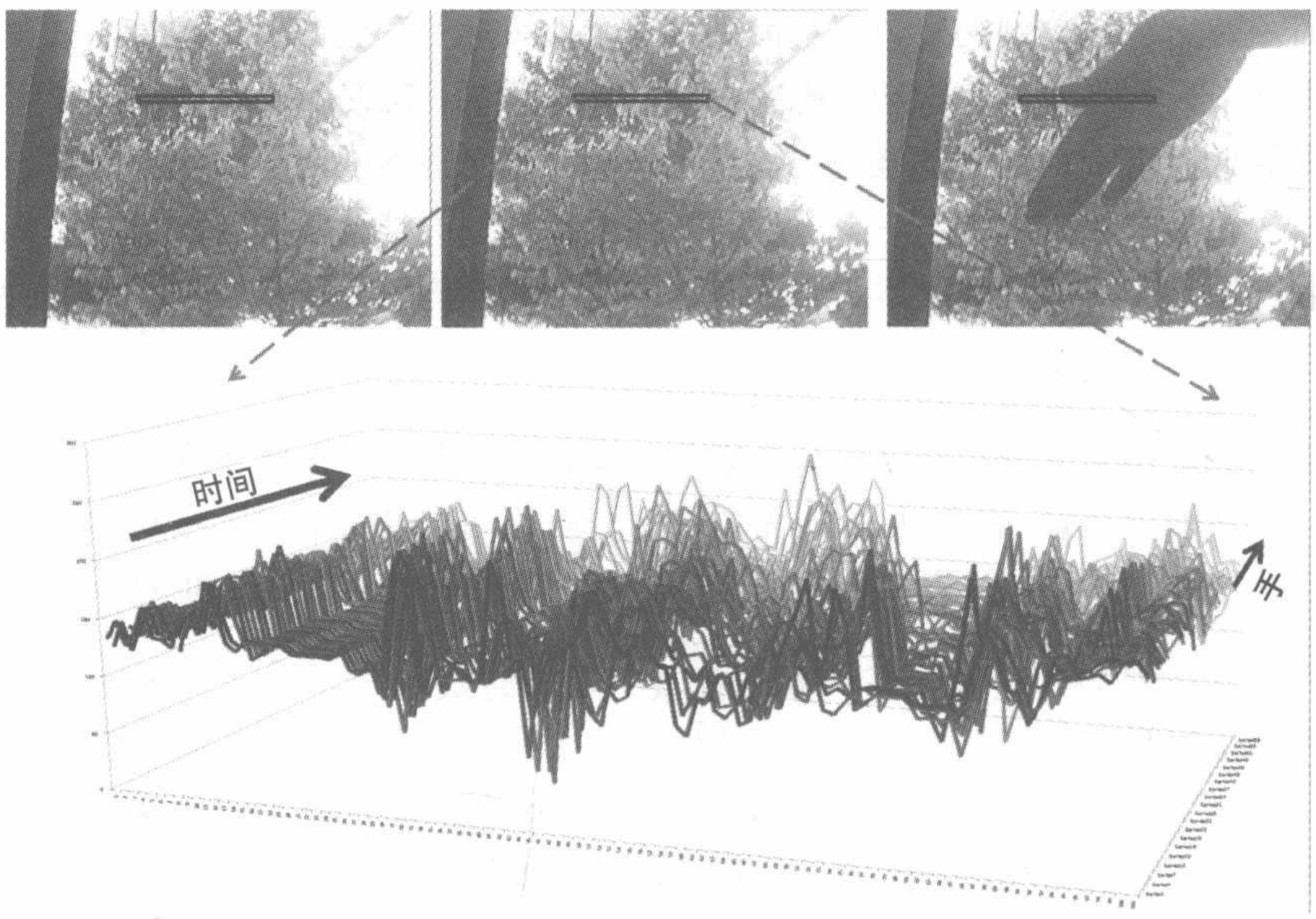


图15-1. 随风摆动的树这个视频中一条给定线上的像素在60帧如何随着时间变化：一些暗色区域（左上）比较稳定，而树枝移动的区域（中上）变化比较剧烈

这里的的输入image可以是任何类型拥有任意数量的通道。点pt1和pt2是直线的两个端点。联通度可以设置为4（直线只能右、左、上、下联通）或是8（直线还可以向对角线方向联通）。最后，如果left_to_right被置0（false），那么line_iterator从pt1到pt2扫过图片，否则从最左到最右扫过图片。[注3]

迭代器可以通过增量迭代的方式遍历给定端点之间的直线上的每个像素。我们使用cv::LineIterator::operator++()来增量迭代。在迭代的每一步，可以同时访问所有通道。例如一个支点迭代器line_iterator，我们可以通过解除引用（例如*line_iterator）来访问当前点。然而值得注意：cv::LineIterator::operator*()的返回值并不是一个指向OpenCV内置向量类型的指针，而是一个uchar*指针。这意味着你需要自己将这个值转为类似cv::Vec3f*这样的东西（或是其他任何适用于图像的数组）。[注4]

注3：之所以引入left_to_right这个标记是因为从pt1到pt2的直线并不总和从pt2到pt1的直线相同，因此通过这个标记可以使我们忽略传参数时pt1和pt2的顺序。

注4：在某些情况下，你可以在这里有点马虎一些。具体来说，当图像已经是无符号字符类型，你可以直接访问元素，使用（* line_iterator）[0]，（* line_iterator）[1]，等等。仔细观察，其实这些都是迭代器解引用获得的字符指针，然后使用内置的C带偏移的解引用括号操作符，而不是构造一个OpenCV中的向量类型，例如vec3f，再通过重载间接引用运算符来访问各个通道。最后，对vec3b特殊情况（或任何数量的通道）结果都一样。

有了这个方便的工具，我们可以从文件中提取一些数据。在示例15-1中，我们从一个视频文件中生成了如图15-1所示的数据。

示例15-1：从视频中的一行读取所有像素的RGB值并将其分别存入三个文件

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <fstream>

using namespace std;

void help( argv ) {
  cout << "\n"
    << "Read out RGB pixel values and store them to disk\nCall:\n"
    << argv[0] <<" avi_file\n"
    << "\n This will store to files blines.csv, glines.csv and rlines.csv\n\n"
    << endl;
}

int main( int argc, char** argv  ) {

  if(argc != 2) { help(); return -1; }
  cv::namedWindow( argv[0], CV_WINDOW_AUTOSIZE );

  cv::VideoCapture cap;
  if((argc < 2)|| !cap.open(argv[1]))
  {
    cerr << "Couldn't open video file" << endl;
    help();
    cap.open(0);
    return -1;
}

cv::Point pt1(10,10), pt2(30,30);
int max_buffer;
cv::Mat rawImage;
ofstream b,g,r;
b.open("blines.csv");
g.open("glines.csv");
r.open("rlines.csv");

// MAIN PROCESSING LOOP:
//
for(;;) {
  cap >> rawImage;
  if( !rawImage.data ) break;

  cv::LineIterator it( rawImage, pt1, pt2, 8);
  for( int j=0; j<it.count; ++j,++it ) {
    b << (int)(*it)[0] << ", ";
    g << (int)(*it)[1] << ", ";
    r << (int)(*it)[2] << ", ";
    (*it)[2] = 255;                // Mark this sample in red
  }
  cv::imshow( argv[0], rawImage );
```

```
    int c = cv::waitKey(10);
    b << "\n"; g << "\n"; r << "\n";
  }

  // CLEAN UP:
  //
  b << endl; g << endl; r << endl;
  b.close(); g.close(); r.close();
  cout << "\n"
    << "Data stored to files: blines.csv, glines.csv and rlines.csv\n\n"
    << endl;
}
```

在示例15-1中，我们一次一个地遍历这些点并分别对每个点进行处理。另一种常见而有用的方法是创建一个（具有合适类型的）缓存器，然后在处理前将整行存入缓存器。这里的缓存器中的副本是：

```
cv::LineIterator it( rawImage, pt1, pt2, 8 );

vector<cv::Vec3b> buf( it.count );

for( int i=0; i < it.count; i++, ++it )
buf[i] = &( (const cv::Vec3b*) it );
```

这种方法最主要的优点是如果图片rawImage不是一个无符号字符型，它就可以用一种更清爽的方法来完成向量类型的转换工作。

我们现在已经可以开始讨论一些用来对图15-1中出现的像素波动进行建模的方法了。在我们由简单到复杂地介绍这些方法时，会把目光关注在可以在合适内存限制下实时运行的模型。

帧间差分

最简单的背景提取方法就是一帧（或之后的几帧）减去另一帧，然后将“足够不同”的地方标为前景。这个过程会捕捉到移动物体的边缘。简单起见，我们考虑三个单通道图片frameTime1，frameTime2和frameForeground。图片frameTime1是过去的一张灰度图片，frameTime2是当前的灰度图片。我们用下面的代码来检测前景，将差分的值（绝对值）保存在frameForeground中。

```
cv::absdiff(
  frameTime1,                    // First input array
  frameTime2,                    // Second input array
  frameForeground                // Result array
);
```

由于像素总是存在噪音和波动，我们应该忽略（设为0）帧间微小的不同（例如小于15），并标记出（设为255）大的不同。

```
cv::threshold(
  frameForeground,                    // Input image
  frameForeground,                    // Result image
  15,                                 // Threshold value
  255,                                // Max value for upward operations
  cv::THRESH_BINARY                   // Threshold type to use
);
```

那么图片frameForeground就将候选的前景物体标记为255，而背景像素标记为0。我们需要像以前一样对其中的小噪音进行清理，可以使用cv::erode()或是连通分量方法。对于彩色图片，我们对每个通道都做帧间差分，然后使用cv::max()来合并各个通道的结果。如果你不只是想将移动区域标记出来，在实际应用中这种方法就太过简单。为了得到一个更有效地背景模型，我们需要记录场景中像素的一些统计量例如均值和平均差。你可以到后面对小测试的介绍看一下图15-6和图15-7展示的帧间差分的效果。

平均背景法

平均法只是简单学习得到背景像素的均值和方差（或者为了计算迅速，使用平均差）作为背景的模型。

考虑图15-1中的像素行。除了用每一帧中各个像素的值来表示（正如图15-1中那样），我们也可以使用在整个视频序列中各个像素的均值和平均差来表示它（图15-2）。该视频中，有一个前景物体（手）经过摄像机前。注意，这个前景物体的亮度和背景中天空与树的亮度有一定差别，图中也展示了手的亮度。

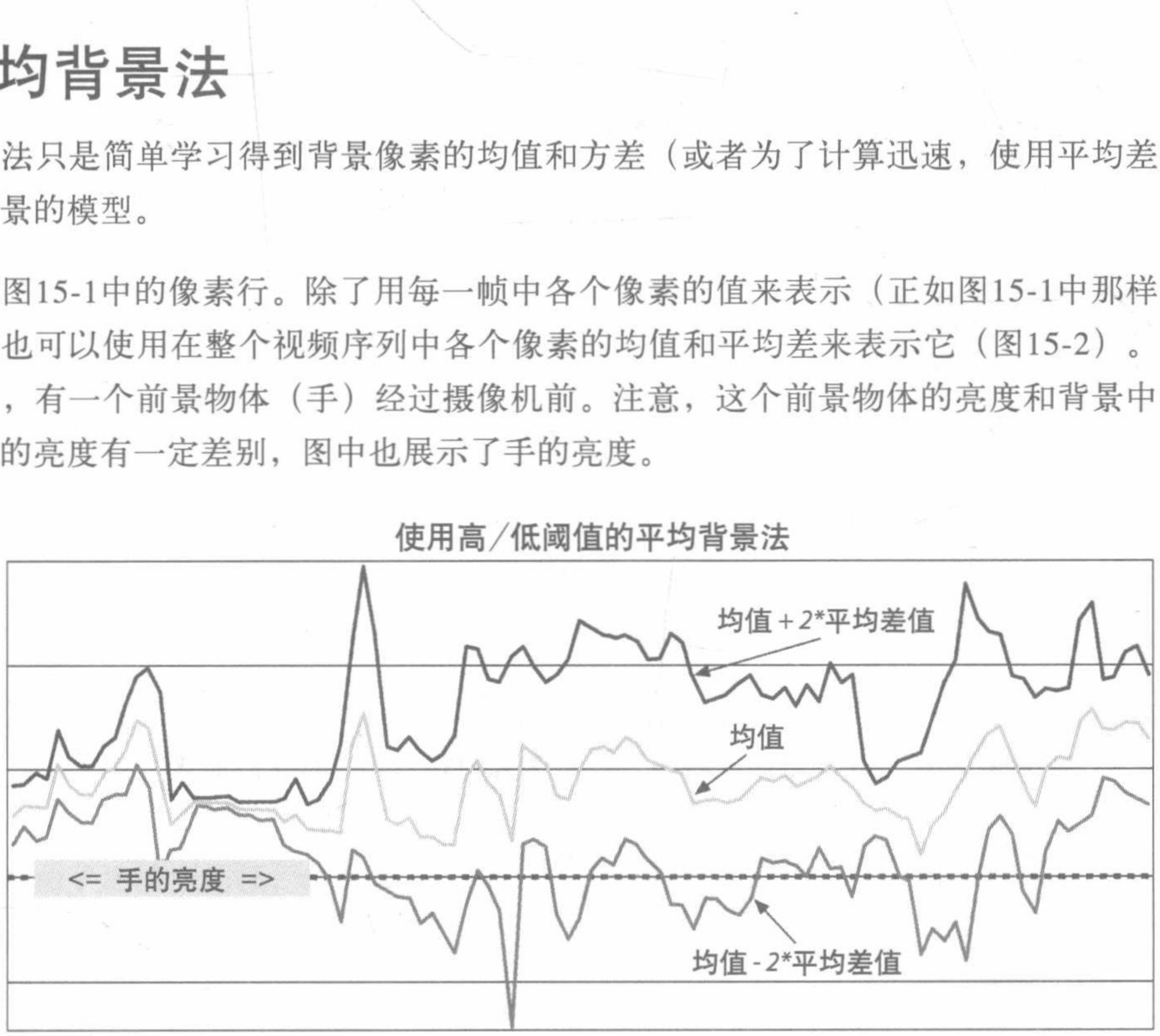


图15-2. 使用平均差的方式来表示图15-1中的数据：一个经过摄像机前的物体（手）比背景稍暗，该物体的亮度已经绘制在图中

平均法使用了四个OpenCV中的方法：cv::Mat::operator+=()来随着时间累加图片；cv::absdiff()来随着时间累加帧间图片的差分；cv::inRange()来对图片中的前景和背

景进行分割（需要学习好的背景模型）；而cv::max()用来将各个不同的颜色通道中分割的结果整合进一个掩膜图像中。由于示例的代码很长，我们一小块一小块地进行分析。

首先，我们创建一些指针，用来指向在整个计算过程中要用到的临时变量和存放统计值的图像的变量（见示例15-2）。对这些指针按其后所指向图片的类型进行排序，是很有用的。

示例15-2：学习一个背景模型来识别前景像素

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <fstream>

using namespace std;

// Global storage
//
// Float, 3-channel images
//
cv::Mat IavgF, IdiffF, IprevF, IhiF, IlowF;
cv::Mat tmp, tmp2;

// Float, 1-channel images
//
vector<cv::Mat> Igray(3);
vector<cv::Mat> Ilow(3);
vector<cv::Mat> Ihi(3);

// Byte, 1-channel image
//
cv::Mat Imaskt;

// Counts number of images learned for averaging later
//
    float Icount;
```

接下来，我们创建一个用来为所有必要中间临时图片申请内存的函数。[注5]为方便起见，我们传入一个图片参数（从视频中取得），这个图片参数只用于指定临时图片的大小：

```
    // I is just a sample image for allocation purposes
    // (passed in for sizing)
    //
    void AllocateImages( const cv::Mat& I ) {

      cv::Size sz = I.size();

      IavgF     = cv::Mat::zeros( sz, CV_32FC3 );
      IdiffF    = cv::Mat::zeros( sz, CV_32FC3 );
```

注5：在例子中，累加器图片是32位浮点型类型。这在帧数不多时不可以，但帧数很多时，可以使用64位浮点型。

```
  IprevF    = cv::Mat::zeros( sz, CV_32FC3 );
  IhiF      = cv::Mat::zeros( sz, CV_32FC3 );
  IlowF     = cv::Mat::zeros( sz, CV_32FC3 );
  Icount    = 0.00001; // Protect against divide by zero

  tmp       = cv::Mat::zeros( sz, CV_32FC3 );
  tmp2      = cv::Mat::zeros( sz, CV_32FC3 );
  Imaskt    = cv::Mat( sz, CV_32FC1 );

}
```

在下一段代码中，我们对累计背景图片和累计绝对值帧间差分（一种比学习图片标准差计算更快的替代方法[注6]）进行学习。通常需要对30到1000帧调用下面所示的函数，有时只是从每秒钟取几帧进行计算，而有时会利用所有的可用帧。需要使用一个三彩色通道8位图来调用下面的方法。

```
// Learn the background statistics for one more frame
// I is a color sample of the background, 3-channel, 8u
//
void accumulateBackground( cv::Mat& I ){

  static int first = 1;                  // nb. Not thread safe
  I.convertTo( tmp, CV_32F );             // convert to float
  if( !first ){
    IavgF += tmp;
    cv::absdiff( tmp, IprevF, tmp2 );
    IdiffF += tmp2;
    Icount += 1.0;
  }
  first = 0;
  IprevF = tmp;
}
```

我们首先使用cv::Mat::convertTo()来将原每通道8位，三通道的背景图转化为一个浮点型三通道图片。然后我们将浮点图累加紧IavgF。接下来，我们使用cv::absdiff()计算帧间绝对值差分图并将结果累加进idiffF。每次我们对图片进行累计时，我们还会对全局变量Icount进行加1，以便最后使用其根据累加值计算平均值。

一旦累加了足够多的帧，我们就可以将其转化为背景的一个统计模型，即计算每个像素的均值和偏移值（平均绝对差）。

注6：注意，我们用了“替代方法”这个词。平均差分在数学上并不等价于标准差，在这个应用中它们产生相似的质量结果。使用平均差的优点是它比标准差稍微快一些。通过对上面的代码稍加修改，你可以改用标准差而不是平均差，可以比较一下它们的区别，在后面我们会讨论这个问题。

```
void createModelsfromStats() {

  IavgF  *= (1.0/Icount);
  IdiffF *= (1.0/Icount);

  // Make sure diff is always something
  //
  IdiffF += cv::Scalar( 1.0, 1.0, 1.0 );
  setHighThreshold( 7.0 );
  setLowThreshold( 6.0 );
}
```

在这一部分我们使用cv::Mat::operator*=()对累计图片除以图片数来计算原图的均值和平均绝对差。以防万一，我们限制平均差图像中的值至少为1，以免出现上下两个阈值正好相等的情况，因此，我们在计算前景－背景阈值时需要用一个因子对进行缩放。

接下来的两个方法setHighThreshold()和setLowThreshold()是两个用来根据帧间平均绝对差（*frame-to-frame average absolute differences*，FFAAD）来设置阈值的实用函数。FFAAD可以视为用来判断两个数据是否差异明显的距离度量。例如，调用setHighThreshold(7.0)表示在高于平均值7倍FFAAD的像素被认为是前景，同理setLowThreshold(6.0)表示低于平均值6倍FFAAD。在这个区间中的像素被认为是背景。这两个函数如下：

```
void setHighThreshold( float scale ) {
  IhiF = IavgF + (IdiffF * scale);
  cv::split( IhiF, Ihi );
}
void setLowThreshold( float scale ) {
  IlowF = IavgF - (IdiffF * scale);
  cv::split( IlowF, Ilow );
}
```

在setLowThreshold()和setHightThreshold()中，我们首先对差分图按因子进行缩放，然后将它从平均图上加上或者减去。接下来通过cv::split()把IhiF或者IlowF应用于图像的每个通道。

一旦得到背景模型，完成高低阈值的设定，我们就可以使用它对图像进行前景（不是背景的部分）和背景（像素值位于背景模型的高低阈值之间的部分）的分割。我们通过下述函数进行分割：

```
// Create a binary: 0,255 mask where 255 means foreground pixel
// I      Input image, 3-channel, 8u
// Imask  Mask image to be created, 1-channel 8u
//
void backgroundDiff(
  cv::Mat& I,
  cv::Mat& Imask
```

```
) {
  I.convertTo( tmp, CV::F32 );    // To float
  cv::split( tmp, Igray );

  // Channel 1
  //
  cv::inRange( Igray[0], Ilow[0], Ihi[0], Imask );

  // Channel 2
  //
  cv::inRange( Igray[1], Ilow[1], Ihi[1], Imaskt );
  Imask = cv::min( Imask, Imaskt );

  // Channel 3
  //
  cv::inRange( Igray[2], Ilow[2], Ihi[2], Imaskt );
  Imask = cv::min( Imask, Imaskt );

  // Finally, invert the results
  //
  Imask = 255 - Imask;
}
```

这个函数首先通过cv::Mat::convertTo()将输入图片I（待分割图片）转化为一个浮点型图片。然后使用cv::split()将三通道图片分为三个单通道图片。接下来，我们对每个通道分别进行处理，使用cv::inRange()以判断每个像素是够位于背景的高低阈值之间，该函数将背景对应的8位灰度图Imaskt中的位置置max（255）而前景部分置为0。对于每个颜色通道，我们使用逻辑AND[注7]将其合称为“单个通道掩码Imask”，这意味着只有在任意颜色通道中都出现强烈差异的像素才会被标记为前景像素。最后，我们使用cv::operator-()来翻转Imask，这是因为前景为阈值之外的像素而不是之内的。这个掩码图像就作为输出图像。

将所有综合起来，我们在main()函数中读入一段视频并构建背景模型。在给出的例子里，我们在训练模式下读取视频直到用户按下空格键，然后程序将继续读入视频名将检测出的前景物体标红：

```
void help( argv ) {
  cout << "\n"
       << "Train a background model on incoming video, then run the model\n"
       << argv[0] <<" avi_file\n"
       << endl;
}

int main( int argc, char** argv  ) {

  if(argc != 2) { help( argv ); return -1; }
```

注7：也可以使用按位或运算符，按位或运算符产生的结果只有0x00和0xff。然而，cv::max()可以得到一个“模糊”的或的结果。

```
cv::namedWindow( argv[0], cv::WINDOW_AUTOSIZE );

cv::VideoCapture cap;
if((argc < 2)|| !cap.open(argv[1])) {
  cerr << "Couldn't open video file" << endl;
  help();
  cap.open(0);
  return -1;
}

// FIRST PROCESSING LOOP (TRAINING):
//
while(1) {
  cap >> image;
  if( !image.data ) exit(0);

  accumulateBackground( image );

  cv::imshow( argv[0], rawImage );
  if( cv::waitKey(7) == 0x20 ) break;
}

// We have all of our data, so create the models
//
createModelsfromStats();

// SECOND PROCESSING LOOP (TESTING):
//
cv::Mat mask;
while(1) {
  cap >> image;
  if( !image.data ) exit(0);

  backgroundDiff( image, mask );

  // A simple visualization is to write to the red channel
  //
  cv::split( image, Igray );
  Igray[2] = cv::max( mask, Igray[2] );
  cv::merge( Igray, image );

  cv::imshow( argv[0], image );
  if( cv::waitKey(7) == 0x20 ) break;
}

  exit(0);
}
```

前面展示的是一个很简单的学习场景中的背景进而分割出前景的方法。只有在场景中没有移动的背景物体的时候它才工作良好，在场景中存在飘动的窗帘或是树枝等具有双或多模态的特征时，该方法会失效。该方法也假设光线几乎是不同的，例如室内静态场景。可以查看图15-6，了解平均背景法的表现。

累计均值，方差和协方差

在平均背景法中描述了如何使用累加算子cv::operator+()来进行最简单的工作：将一些数据加起来进而在归一化后得到平均值。当然，平均值由于种种原因是一个很方便的统计量，但它有一个经常被忽略的优点，即事实上可以以增量的方式计算平均值。[注8]这意味着我们可以以增量的方式进行处理而无需分析前事先进行累加。在这一节中，我们会考虑一个稍微更复杂一点的模型，它以增量的方式进行计算。

我们下面这个模型使用Gaussian模型来表示一个像素亮度（或颜色）的变化。一个一维的Gaussian模型由一个均值和一个方差（描述被测数据和其均值的期望偏离程度）给出。而一个d维模型（例如三颜色通道）会由一个d维均值向量和一个d^2维协方差矩阵给出，这里的协方差矩阵不仅包括每一维的方差，还包括各维之间的协方差新信息。

正如前面所提到的，均值、方差和协方差，所有这些量都可以以增量的方式进行计算。如果输入图像流已给定，我们可以定义三个函数来累加必要的数据，并将其转为模型中的参数。

接下来的代码定义了几个全局变量：

```
cv::Mat sum;
cv::Mat sqsum;
int     image_count = 0;
```

使用cv::Mat::operator+=()计算均值

正如先前的例子，计算像素均值的好方法是使用cv::Mat::operator+=()来对像素进行累加，然后除以图片的总数进而得到均值：

```
void accumulateMean(
  cv::Mat& I
) {
  if( sum.empty ) {
    sum = cv::Mat::zeros( I.size(), CV_32FC(I.channels()) );
  }
  I.convertTo( scratch, sum.type() );
  sum += scratch;
  image_count++;
}
```

函数accumulateMean()将对每个输入的图像执行操作，当所有作为背景的图像被计算完

注8：纯粹来说，我们的实现不是一个纯的增量计算，因为我们在最后除以了样本数。然而，当出现新的数据点时，确实有一种纯增量的方法来更新平均值，但使用的“几乎增量”版本的计算效率更高。如果可以从数据的纯累积函数中计算且最后除以样本数的话，我们将在整个章节中继续把方法称为“增量”。

毕之后，你就可以调用下一个函数computeMean()来获得包含所有图像像素均值的“图像”。

```
cv::Mat& computeMean(
  cv::Mat& mean
) {
  mean = sum / image_count;
}
```

使用cv::accumulate()计算均值

OpenCV提供了另一个和使用cv::Mat::operator+=()相似的函数，但有两点主要的区别。第一是它会自动完成cv::Mat::convertTo()的功能（因此也就可以免去一个临时变量），第二是它可以使用掩码图像。这个函数就是cv::accmulate()。在计算背景模型时可以使用掩码图像是十分有用的，因为有时可以通过一些其他信息确认该部分一定不是背景。例如，你可能构建了一个高速公路上或者其他颜色单一区域上的背景模型。通过判断颜色，你可以立刻判断出某一区域一定不是背景。如果位于一个几乎没有前景物体的场景中，这种掩膜方法就十分有用了。

累加函数的原型如下：

```
void accumulate(
  cv::InputArray       src,                  // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                  // Result image, F32 or F64
  cv::InputArray       mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

累加发生在数组dst中，而数组src是等待累加的新图片。cv::accumulate()可以使用一个可选的掩膜。如果给定了掩膜，dst中掩膜对应位置非零的像素才会被更新。

通过使用cv::accumulate()，先前的accmulateMean()可以如下简化：

```
void accumulateMean(
  cv::Mat& I
) {
  if( sum.empty ) {
    sum = cv::Mat::zeros( I.size(), CV_32FC(I.channels()) );
  }
  cv::accumulate( I, sum );
  image_count++;
}
```

变种：使用cv::accumulateWeighted()计算均值

另一个可选且有用的方法是使用running average（移动平均），由以下公式给出：

$$acc(x,y) = (1 - \alpha) \cdot acc(x, y) + \alpha \cdot image(x, y)$$

对于一个给定的值alpha，移动平均法和使用cv::Mat::operator+=()或cv::accumulate()的求和法得到的结果并不相等。为了说明这一点，考虑在alpha为0.5时对三个数(2,3和4)相加。如果我们使用cv::accumulate()，则得到的累加结果为9，求平均是3。如果我们使用cv::accumulateWeighted()，那么第一个求和会得到0.5·2+0.5·3=2.5，然后加上第三项0.5·2.5+0.5·4=3.25。后者结果偏大的原因是移动平均法中，最近出现的值对结果的贡献比先前的要大。由于这个原因，移动平均法又被称为*tracker*。你可以认为参数alpha是控制先前帧产生的影响淡去的速率，它的值越小先前帧产生的影响淡去的速度越快。

为了在整个图像序列中进行累加，我们要用OpenCV提供的函数cv::accumulateWeighted()：

```
void accumulateWeighted(
  cv::InputArray       src,                  // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                  // Result image, F32 or F64
  double               alpha,                // Weight factor applied to src
  cv::InputArray       mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

累加发生在数组dst中，src是待累加的新图片。alpha值是权重参数。正如cv::accumulate()，cv::accumulateWeighted()可以使用一个可选的掩膜，如果给定了该掩膜，dst中只有在mask中对应为非0的元素才会被更新。

借助cv::accumulateSquare()计算方差

我们同样可以累加图像的平方，从而快速计算单个像素的方差。回忆一下统计课上学到的知识，一个有限总体上的方差定义如下：

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \bar{x})^2$$

其中是N个样本中x的均值。这个式子存在的问题是需要遍历数据两次来计算。稍微用一点代数知识我们得到下面的式子，同样是正确的：

$$\sigma^2 = \left(\frac{1}{N}\sum_{i=0}^{N-1}x_i^2\right) - \left(\frac{1}{N}\sum_{i=0}^{N-1}x_i\right)^2$$

使用上面这种形式，我们可以在一次遍历中同时累加像素的值和它们的平方。这样就可以通过像素平方的均值减去像素的均值来得到其方差。借助这个想法，我们可以和先前计算均值一样定义一个累加函数和一个计算函数。如同在

计算均值时，我们可以首先对输入图像做逐像素的平方，然后使用类似sqsum += I.mul(I)的方法进行累加。但这种方法有几个缺点，最重要的一个是I.mul(I)并不会做任何隐式的类型转化（就像cv::Mat::operator+=()算子也不会做）。这会导致在（例如）一个8位数组平方时候不可避免地发生溢出。然而和cv::accumulate()一样，OpenCV提供给我们一个恰好需要的函数cv::accumulateSquare()：

```
void accumulateSquare(
  cv::InputArray      src,                  // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                 // Result image, F32 or F64
  cv::InputArray      mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

有了cv::accumulateSquare()，我们可以将累加必要信息以计算方差这个操作写成以下函数：

```
void accumulateVariance(
  cv::Mat& I
) {
  if( sum.empty ) {
    sum   = cv::Mat::zeros( I.size(), CV_32FC(I.channels()) );
    sqsum = cv::Mat::zeros( I.size(), CV_32FC(I.channels()) );
  }
  cv::accumulate( I, sum );
  cv::accumulateSquare( I, sqsum );
  image_count++;
}
```

相关联的计算函数如下：

```
// note that 'variance' is sigma^2
//
void computeVariance(
  cv::Mat& variance
) {
  double one_by_N = 1.0 / image_count;
  variance        = one_by_N  * sqsum - (one_by_N * one_by_N) * sum.mul(sum);
}
```

使用cv::accumulateWeighted()计算协方差

在一个多通道图片中，每个单独通道的方差刻画了我们期望未来出现的背景像素和观察到的均值的相似程度。但这不管对于模型还是我们的“期望”都太过于简化了。这里引入一个重要的概念就是协方差。协方差刻画了不同通道间的相互关联性。

例如，我们的背景模型是一片海的场景，因此我们期望在红色通道上背景会有很小的方差，而在绿色和红色通道上的方差可以比较大。从我们的直觉来看，海的颜色应该只有一种，而其颜色反映在图片上的变化主要是光线产生的效果，我们可能会得到这样的结

论，如果在绿色通道上有亮度的增加或是减少，那么在蓝色通道上也应该出现相应的增加和减少。那么显然，如果一个像素在蓝色通道上右客观的增加但在绿色像素上却没有相应增加，我们认为这个像素不是背景。协方差这个概念就刻画了这种直觉。

在图15-3中，我们对海洋背景这个例子中某个像素的蓝色和绿色通道中的数据进行了可视化。左侧图中只计算了各个通道的方差，而右侧我们计算了两个通道的协方差，注意，协方差的模型对数据拟合得更好。

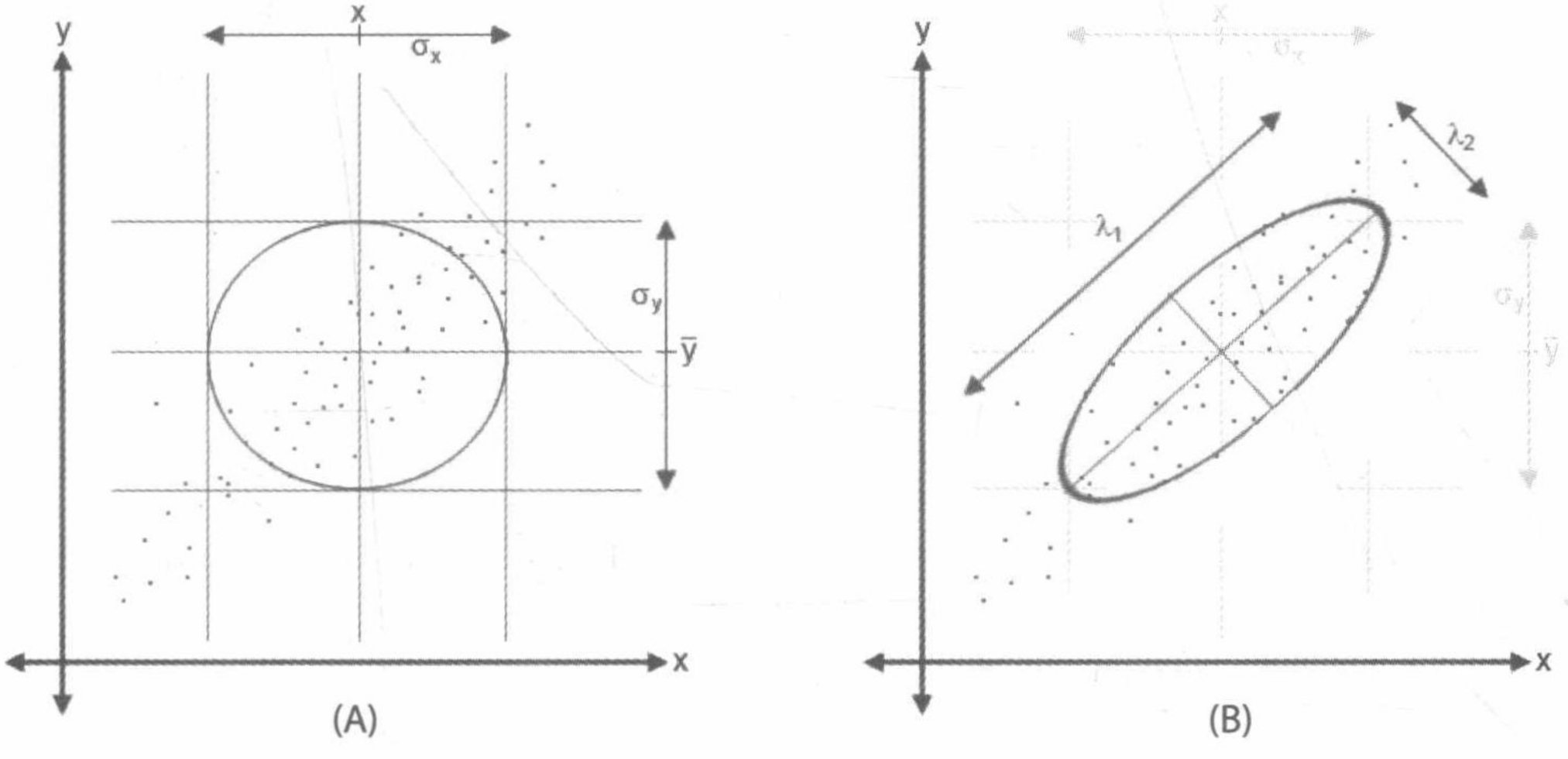


图15-3. 左右图为相同的数据集。在左侧（A）中，展示了数据在x和y维度上的方差（的平方根）和所定义的模型。在右侧（B）中，可视化数据的协方差。和方差时的模型相比，协方差时的模型变成了一侧宽一侧窄的椭圆

数学上来说，两个不同变量的协方差由下式给出：

$$Cov(x, y) = \left(\frac{1}{N}\sum_{i=0}^{N-1}(x_i \cdot y_i)\right) - \left(\frac{1}{N}\sum_{i=0}^{N-1}x_i\right)\left(\frac{1}{N}\sum_{i=0}^{N-1}y_i\right)$$

正如你看到的那样，任意一个变量x和它自己的协方差，$Cov(x,x)$，正是其方差σ_x^2。在一个d维空间中，比如一个像素的RGB值，$d = 3$，通过协方差矩阵$\Sigma_{x,y}$来讨论协方差十分方便，矩阵中每个元素是两个变量的协方差或是某个变量的方差。正如你从公式中看到的，协方差矩阵是对称的，也就是说$\Sigma_{x,y}=\Sigma_{y,x}$。

在第5章，我们遇到了一个可以在处理数据的单个向量时使用的函数（不是单个向量的数组）cv::calcCovarMatrix()。这个函数要求我们提供N个d为向量，它会计算出$d \times d$的协方差矩阵。现在我们遇到的问题是我们希望对一个数组中的每点计算出一个矩阵（至少，在三维RGB图片的情况，我们希望计算出对称矩阵中的6个系数）。

实际中，做这件事最好的方法就是简单使用已经写好的代码来计算方差，然后分别计算其余三个新对象（$\Sigma_{x,y}$中的非对角元素）。观察协方差的公式，我们注意到cv::accumulateSquare()不能用在这里，因为我们需要累加$(x_i \cdot y_i)$（也就是说，每张图中某个特定像素的两个不同通道的值的乘积）。这里，我们应该使用OpenCV中提供的函数cv::accumulateProduct()：

```
void accumulateProduct(
  cv::InputArray       src1,                 // Input, 1 or 3 channels, U8 or F32
  cv::InputArray       src2,                 // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                  // Result image, F32 or F64
  cv::InputArray       mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

该函数和cv::accumulateSquare()的工作方式一样，只不过它并不对src中的每个元素进行平方，而是对src1和src2中对应位置的元素进行相乘。这个函数（很不幸的）不允许我们从输入数组中提取单个通道。输入的src1和src2是多通道时，计算是以逐通道的形式进行的。

对于我们当前计算非对角线上元素的协方差这个需求，这并不是我们想要的，实际上我们需要的是同一个图片上不同的通道。为了实现这一点，我们必须用cv::split()将输入图片分离成单独的通道，正如示例15-3所示。

示例15-3：计算协方差模型中非对角线元素

```
vector<cv::Mat> planes(3);
vector<cv::Mat> sums(3);
vector<cv::Mat> xysums(6);

int image_count = 0;

void accumulateCovariance(
  cv::Mat& I
) {

  int i, j, n;
  if( sum.empty ) {
    for( i=0; i<3; i++ ) {          // the r, g, and b sums
      sums[i]   = cv::Mat::zeros( I.size(), CV::F32C1 );
    }
    for( n=0; n<6; n++ ) {          // the rr, rg, rb, gg, gb, and bb elements
      xysums[n] = cv::Mat::zeros( I.size(), CV::F32C1 );
    }
  }
  cv::split( I, rgb );
  for( i=0; i<3; i++ ) {
    cv::accumulate( rgb[i], sums[i] );
  }
```

```
  n = 0;
  for( i=0; i<3; i++ ) {            // "row" of Sigma
    for( j=i; j<3; j++ ) {          // "column" of Sigma
      n++;
      cv::accumulateProduct( rgb[i], rgb[j], xysums[n] );
    }
  }
  image_count++;

}
```

相应的计算函数也只是先前方差计算函数的做一点小改动。

```
// note that 'variance' is sigma^2
//
void computeVariance(
  cv::Mat& covariance          // a six-channel array, channels are the
                               // rr, rg, rb, gg, gb, and bb elements of Sigma_xy
) {
  double one_by_N = 1.0 / image_count;

  // reuse the xysum arrays as storage for individual entries
  //
  int n = 0;
  for( int i=0; i<3; i++ ) {       // "row" of Sigma
    for( int j=i; j<3; j++ ) {     // "column" of Sigma
      n++;
      xysums[n] = one_by_N  * xysums[n]
        - (one_by_N * one_by_N) * sums[i].mul(sums[j]);
    }
  }

  // reassemble the six individual elements into a six-channel array
  //
  cv::merge( xysums, covariance );
}
```

关于模型测试的简短注释和cv::Mahalanobis()

我们前面介绍了一些稍微复杂的模型，但没有讨论如何测试新图像中的特定像素是否位于背景模型的预测域中。只使用方差时（即对各通道间有统计上互相独立的Gaussian模型），由于各维度间的方差未必相同，因此会存在一些问题。这时通常使用一种对每个维度分别计算z值（z-score）（数据和均值的偏差除以标准差$(x-\bar{x}) / \sigma_x$）的方式来解决。z值告诉我们特定像素来自某个分布的概率。进而，我们通过对各个维度的z值的平方和开方值来表示多维数据的z值，例如：

$$\sqrt{z_{red}^2 + z_{green}^2 + z_{blue}^2}$$

在使用完整的协方差矩阵时，一个类似于z值的概念是马氏距离（Mahalanobis distance）。正如图15-3所示，这基本上是以恒定概率等高线测量的平均值到点的距离。观察图15-3，我们发现模型(A)中均值左上角的点倾向于有一个小的马氏距离。而模型(B)中相同位置的点会有一个大的马氏距离。值得注意的是，简化模型的z值正是图（A）中的马氏距离。OpenCV提供计算马氏距离的函数：

```
double cv::Mahalanobis(
  cv::InputArray vec1,                  // Return distance as F64
  cv::InputArray vec2,                  // First vector (1-dimensional, length n)
  cv::InputArray icovar                 // Second vector (1-dimensional, length n)
                                        // Inverse covariance matrix, n-by-n
);
```

该cv::Mahalanobis()函数要求向量vec1和vec2是d维的，还需要一个$d \times d$的协方差矩阵的逆icovar。（需要传入该逆的原因是，计算一个矩阵的逆很耗时，而一般都有很多向量需要使用同一个协方差矩阵进行比较，所以应该先将该矩阵求逆然后给cv::Mahalanobis()传入该矩阵逆多次即可。



在背景提取中，这很不方便，因为cv::Mahalanobis()需要以一种逐元素的方式调用。不幸的是，OpenCV中并不存在数组形式的调用。那你需要遍历每个像素创建协方差矩阵，对该矩阵求逆并存储。接下来如果你想进行比较，仍然需要遍历每个元素，使用求得的逆对每个像素调用cv::Mahalanobis()。

更复杂的背景提取方法

很多场景中的背景包括复杂的移动的物体，例如风中挥舞的树枝，转动的风扇，飘动的窗帘等。场景中也经常包括变化的光线，例如偶尔飘过的云朵或门窗开关所带来的光线变化。

一个处理这些场景的方法是对每个像素或每组像素使用一个时间序列的模型。这种模型可以很好地处理时间上的波动，但缺点是需要大量的内存[Toyama99]。如果我们使用30 Hz视频中的2秒，这意味着每个像素需要60个样本。然而，每个像素最终学习到的结果是60个不同的权值。通常情况下，我们需要使用远长于2秒的背景计算统计量，这意味着上述方法在当下硬件条件下是不切实际的。

为了达到和自适应滤波器相当的性能，我们从视频压缩的技巧中取得灵感，尝试使用

YUV[注9]码书codebook[注10]来表达背景中的关键状态[注11]。最简单的方法是将新的像素值和先前观察到的值进行比较。如果新值和先前的某个值很接近，我们就认为它就是先前的那个值加上微扰。如果它们并不接近，我们就认为该新值是该像素处背景的一类新颜色。可以将这种方式想象成RGB空间中漂浮着几团，每团都表示背景可能的颜色。

实际中，使用RGB并不是最好的选择。选择那些颜色空间坐标中有亮度的空间总会获得很好的效果，例如YUV颜色空间。（YUV是最常见的选择，但HSV效果也不错，其中*V*是指亮度）其中的原因是，根据经验，背景中大多数的颜色变化都发生在亮度轴上而不是其他颜色轴。

接下来要考虑的细节就是如何对这些“团”进行建模。我们可以和原来简单的模型使用一样的方法，例如我们可以将这些“团”建模为带有均值和协方差的Gaussian类簇。结果表明使用最简单的模型，仅仅使用沿颜色空间坐标轴的“盒子”模型，就足以达到很好的效果。这个模型在计算是否一个新像素在学习到的“盒子 ”中时，不仅在内存上，在计算时间上也都是最简单的。

让我们使用一个简单的例子具体解释一下什么是码书（codebook）（图15-4）。一个码书是由一些“盒子”组成，这些“盒子”会不断扩大以覆盖该点随时间变化所出现的常见的颜色。图15-4上方展示了一个随时间变化的波形，可以把它想象成是背景中某个像素的亮度变化。在下方，“盒子”生成以覆盖新出现的值并缓慢生长以覆盖其附近的值。如果新出现的值离得太远，那么会生成一个可以覆盖它的新“盒子”朝着新值出现的方向生长。

在我们的背景模型中，我们学习三维的码书盒子：三维是图像中每个像素处的三个通道。图15-5展示了从图15-1中6个不同像素学习出的（亮度）码书[注12]。码书法可以处理像素值变化剧烈的情况（例如一个表示随风飘舞的树的像素，不断在各种树叶的像素或是树后的蓝天之间切换）。通过使用这种更加精确的建模方法，我们可以检测出颜色在

注9：YUV是早期彩色电视机为了向前兼容黑白电视机而使用的一种颜色空间。第一个信号是零度或黑白电视机使用的“亮度”Y。为了节省空间，只需要传递两个色度信号U（蓝－亮度）和V（红－亮度）。使用一个公式可以从其中重建出RGB颜色。

注10：OpenCV中实现的方法来源于Kim，Chalidabhongse，Harwook和Davis[Kim05]，但不同于它使用的基于学习的在RGB空间中的圆柱体，为了速度，作者使用了和YUV颜色空间中坐标轴对齐的盒子。清理结果更快的方法可以参见Martins[Martins99]。

注11：虽然在背景建模和分割领域存在着大量的文献，OpenCV选用实现的主要想快速而稳定的使用其来收集前景物体，并最终使用其来训练分类器。在背景提取后期的工作可以允许任意的相机运动[Farin04,Colombari07]和使用均值漂移算法进行动态的背景建模[Liu07]。

注12：为了避免杂波，我们从直线上选了几个点。当然，每个点事实上都会学习出一个码书。

这些背景颜色之间的前景物体。和图15-2中无法区分出手的颜色（虚线所示）的情况对比，我们看出码书法比图15-8所示的平均法有更好的性能。

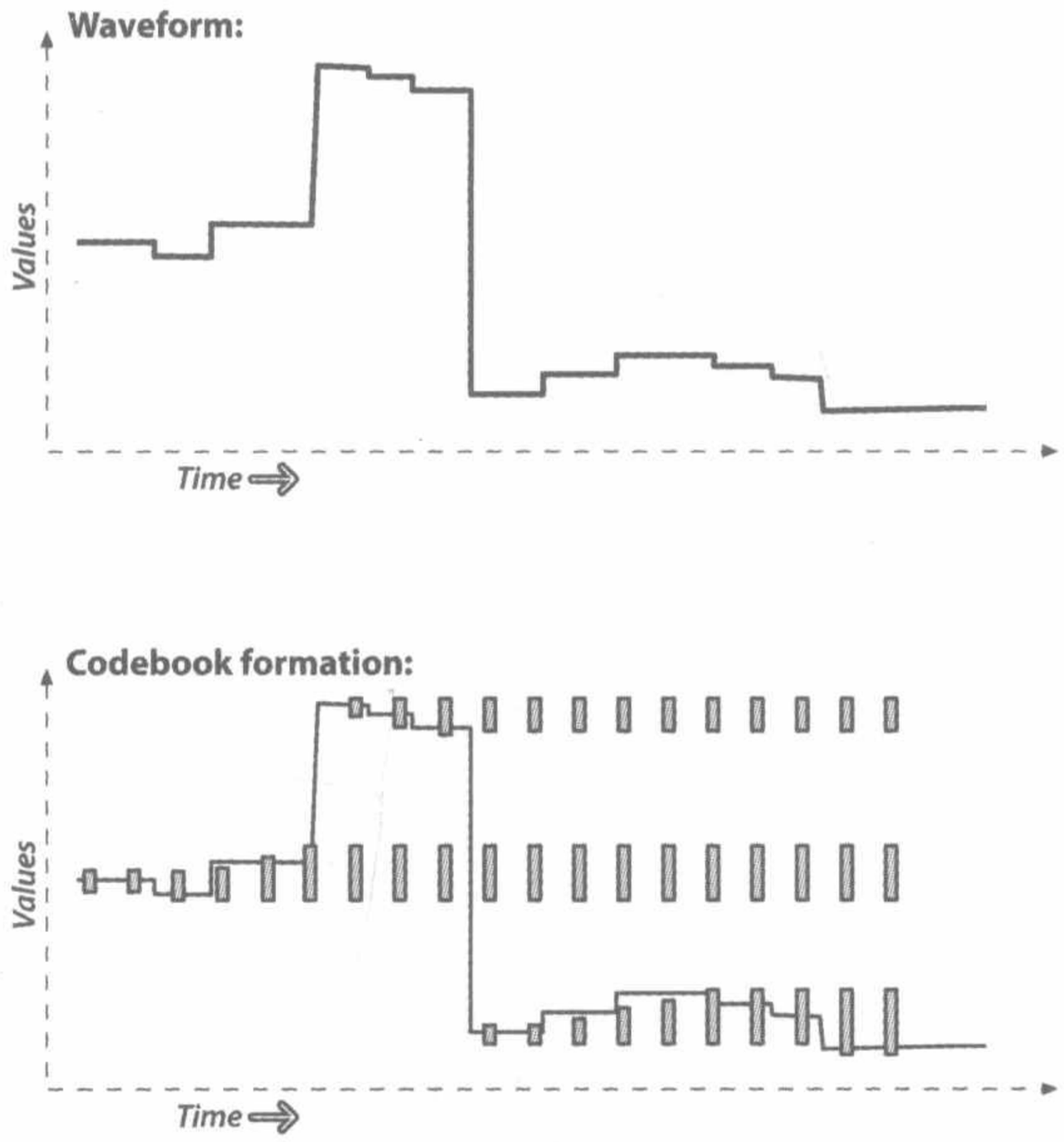


图15-4. 码书只是一些限制亮度值的“盒子”：“盒子”生成以覆盖新出现的值并缓慢生长以覆盖其附近的值。如果新出现的值离得太远，那么会生成一个可以覆盖它的新“盒子”

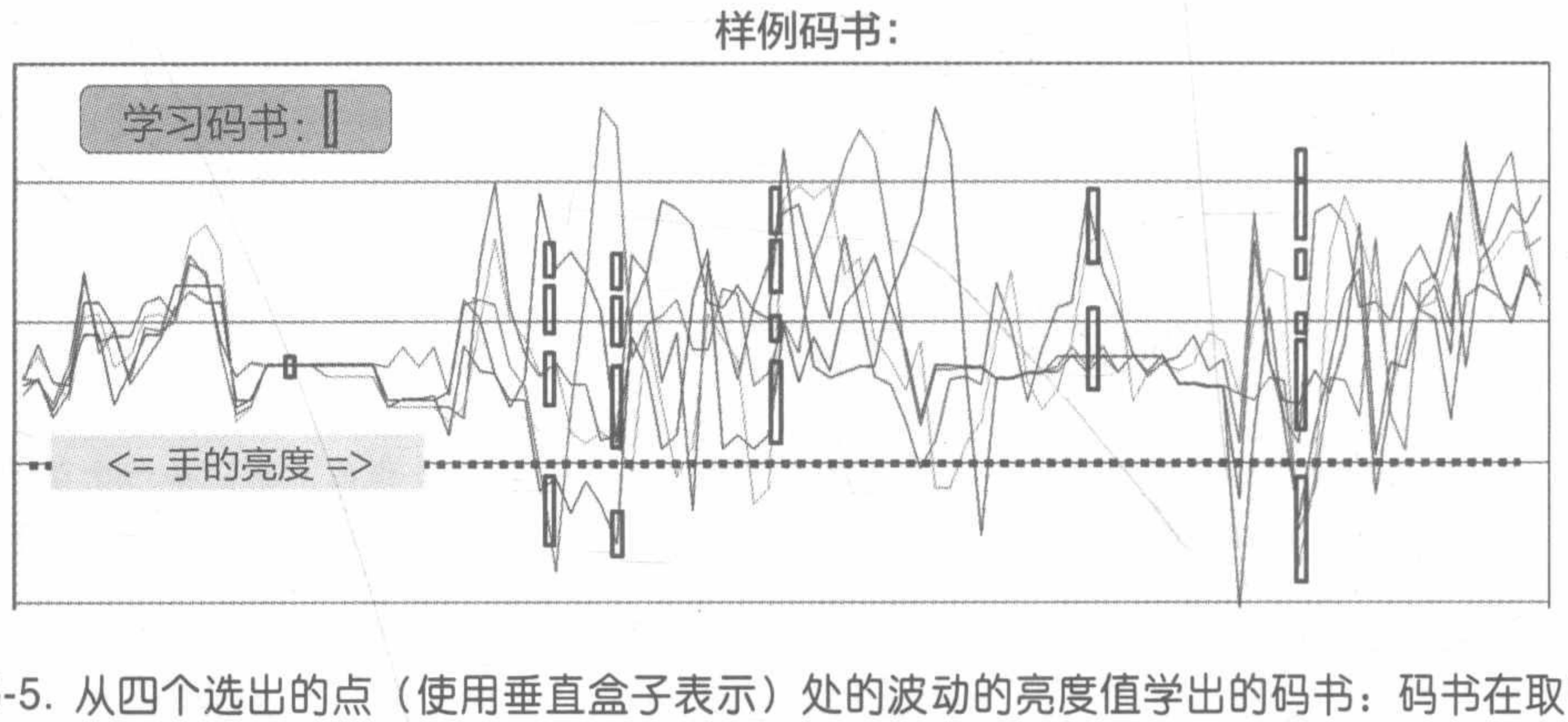


图15-5. 从四个选出的点（使用垂直盒子表示）处的波动的亮度值学出的码书：码书在取离散值得像素处进行累加计算，因此其可以更好地表示离散分布；因此它可以辨识出一个前景处的手（亮度值用虚线表示），手的亮度值在几个背景像素生成的盒子之间。此时学习出的码书是一维的，只能表示亮度的变化

在使用码书法学习背景模型时，每个盒子都是由两个阈值（max和min）在三种颜色上定义的。如果新背景像素落在学习域之内（learnHigh和learnLow)高于max或是低于min，

这些盒子的边界会扩张（max会变大，min会减小）。如果一个新的背景像素落在盒子边界和学习域之外，那么会生成一个可以覆盖它的新盒子。在背景差分模式下，会有一个接受域maxMod和minMod；通过接受域来判断一个新像素是不是“足够接近”max或是min，并据此判断改点是不是在盒子内。在运行时，“盒子”的边界可以和构造盒子时使用的边界不同，通常，这个容忍边界在三个通道中简单被设为0。



我们不讨论使用云台摄像机拍摄大场景的情况。此时应当同时考虑学习到的模型和云台摄像机的角度。

结构

是时候开始细节部分的讨论了，我们先来实现码书的算法（示例15-4）。首先需要定义码书的结构，它只是从YUV空间中的点到一系列盒子的映射。

示例15-4：*码书算法实现*

```
class CodeBook : public vector<CodeElement> {
public:

  int t;                                                  // count every access

  CodeBook() { t=0; }                                     // Default is an empty book
  CodeBook( int n ) : vector<CodeElement>(n) { t=0; } // Construct book of size n

};
```

码书继承自STL中的vector[注13]。变量t对从一开始或是上一次清零操作开始累加的点的个数。码书中实际元素如下：

```
    #define CHANNELS 3

    class CodeElement {
    public:

      uchar learnHigh[CHANNELS];  // High side threshold for learning
      uchar learnLow[CHANNELS];   // Low side threshold for learning
      uchar max[CHANNELS];        // High side of box boundary
```

注13：对每个像素使用STL中的vector来表示码书十分低效。在实际实现中，应该使用一种更加高效的表示方法。例如使用MAX_CODE来限制最大码书的大小，然后使用静态分配的数组CodeElement[MAX_CODES]。

```
uchar min[CHANNELS];        // Low side of box boundary
int   t_last_update;        // Allow us to kill stale entries
int   stale;                // max negative run (longest period of inactivity)

CodeElement() {
  for(i = 0; i < CHANNELS; i++)
    learnHigh[i] = learnLow[i] = max[i] = min[i] = 0;
  t_last_update = stale = 0;
}

CodeElement& operator=( const CodeElement& ce ) {
  for( i=0; i<CHANNELS; i++ ) {
    learnHigh[i] = ce.learnHigh[i];
    learnLow[i]  = ce.learnLow[i];
    min[i]       = ce.min[i];
    max[i]       = ce.max[i];
  }
  t_last_update = ce.t_last_update;
  stale         = ce.stale;
  return *this;
 }

 CodeElement( const CodeElement& ce ) { *this = ce; }

};
```

每个码书实体对每个通道占用4字节同时额外还占用两个整数，总共（4*通道数+4+4）字节(当使用三通道时为20字节)。我们可以把CHANNELS设置成等于或少于图片中颜色通道数的任意正数，但常用的时1（*Y*，或是只使用亮度）或3（YUV或是HSV图片）。在这个数据结构中，对于每个通道，max和min是码书中和盒子的边界。参数learnHigh[]和learnLow[]是出发新码书单元生成的开关。具体说来，如果一个新像素的值不在每个通道的min-learnLow和max+learnHigh之间，就会生成一个新的码书单元。最后一次更新的时间(t_last_update)和stale是在学习时用来对几乎没用过的码书单元进行删除。现在，我们可以着手研究使用这种数据结构来进行动态背景学习的函数了。

进行背景学习

对于每个像素而言，有伴随一个由码单元（CodeElement）组成的码书（CodeBook)。我们需要一个能容纳个数和图像中需要学习的像素相同的码书的数组。对于图像中的每个像素，需要调用足够对背景变化进行建模次数的updateCodebook()进行模型的学习。学习到的模型可能需要周期性重新训练，同时我们可以使用clearStaleEntries()来从存在（少量）移动着的前景物体的图像中进行背景学习。之所以可以这样学习的理由是由移动着的前景学习出的“陈旧”背景被clearStaleEntries()删去。updateCodebook()的接口如下：

```
// Updates the codebook entry with a new data point
// Note: cbBounds must be of length equal to numChannels
//
//
int updateCodebook(              // return CodeBook index
  const cv::Vec3b& p,            // incoming YUV pixel
  CodeBook&        c,            // CodeBook for the pixel
  unsigned*        cbBounds,     // Bounds for codebook (usually: {10,10,10})
  int              numChannels   // Number of color channels we're learning
) {
  unsigned int high[3], low[3], n;
  for( n=0; n<numChannels; n++ ) {
    high[n] = p[n] + *(cbBounds+n);     if( high[n] > 255 ) high[n] = 255;
    low[n]  = p[n] - *(cbBounds+n);     if( low[n]  < 0   ) low[n]  = 0;
  }
  // SEE IF THIS FITS AN EXISTING CODEWORD
  //
  int i;
  int matchChannel;
  for( i=0; i<c.size(); i++ ) {

    matchChannel = 0;
    for( n=0; n<numChannels; n++ ) {
      if( // Found an entry for this channel
       ( c[i].learnLow[n] <= p[n] ) && ( p[n] <= c[i].learnHigh[n])
      )
      matchChannel++;
    }

    if( matchChannel == numChannels ) {      // If an entry was found
      c[i].t_last_update = c.t;

      // adjust this codeword for the first channel
      //
      for( n=0; n<numChannels; n++ ) {
        if( c[i].max[n] < p[n] )        c[i].max[n] = p[n];
        else if( c[i].min[n] > p[n] )   c[i].min[n] = p[n];
      }
      break;
    }

  }
. . .continued below
```

该函数在像素p落在存在的码书盒子外时对盒子进行生长或是直接创建一个新的码书。当像素落在一个已经存在的盒子的cbBounds之内时，盒子会进行生长。此像素落在cvBounds外会直接创建一个新的码书。函数首先设定接下来要用到的high和low值。接下来遍历每个码书来检查像素p是否落在码书“盒子”的学习阈内。如果像素落在所有通道的学习阈内，那么max或min的值会被调整为包括该像素，同时表示上一次更新时间的变量也会被更新为当前时间戳，即c.t。接下来调用updateCodebook()维护每个码书命中次数这个统计量：

. . . continued from above

```
// OVERHEAD TO TRACK POTENTIAL STALE ENTRIES
//
for( int s=0; s<c.size(); s++ ) {

  // Track which codebook entries are going stale:
  //
  int negRun = c.t - c[s].t_last_update;
  if( c[s].stale < negRun ) c[s].stale = negRun;

}
```

. . . .continued below

这里变量stale存储的是最长未运行时间（即码书未被数据命中的最长时间段）。记录“陈旧”单元允许我们删去这些码书，因为它们通常是由噪音或是移动的前景物体学习来的，因此随着时间推移很少被命中而变得“陈旧”。在背景学习的下一阶段，updateCodebook()在需要的时候创建了一个新的码书：

. . . continued from above

```
// ENTER A NEW CODEWORD IF NEEDED
//
if( i == c.size() ) {          // if no existing codeword found, make one
  CodeElement ce;
  for( n=0; n<numChannels; n++ ) {
    ce.learnHigh[n] = high[n];
    ce.learnLow[n]  = low[n];
    ce.max[n]       = p[n];
    ce.min[n]       = p[n];
  }
  ce.t_last_update = c.t;
  ce.stale = 0;
  c.push_back( ce );

}
```

. . .continued below

最终，虽然像素落在盒子阈值外边但仍然在high和low的界内，updateCodebook()缓慢调整（通过加1）learnHigh和learnLow表示的学习阈：

. . . continued from above

```
// SLOWLY ADJUST LEARNING BOUNDS
//
for( n=0; n<numChannels; n++ ) {

  if( c[i].learnHigh[n] < high[n]) c[i].learnHigh[n] += 1;
  if( c[i].learnLow[n]  > low[n] ) c[i].learnLow[n]  -= 1;
```

```
    }
    return i;
  }
```

函数最终以返回修改后的码书的索引而结束。我们现在已经了解如何学习码书。为了能够在图片中存在移动的前景同时避免学习到杂乱的噪声，我们需要一种方法来删去学习过程中很少被命中的码书。

存在移动的前景物体时进行背景学习

下面的方法clearStaleEntries()允许我们在存在移动的前景物体时进行背景学习：

```
// During learning, after you've learned for some period of time,
// periodically call this to clear out stale codebook entries
//
int clearStaleEntries(     // return number of entries cleared
  CodeBook &c              // Codebook to clean up
){
  int staleThresh = c.t>>1;
  int *keep       = new int[c.size()];
  int keepCnt     = 0;

  // SEE WHICH CODEBOOK ENTRIES ARE TOO STALE
  //
  for( int i=0; i<c.size(); i++ ){
  if(c[i].stale > staleThresh)
    keep[i] = 0;  // Mark for destruction
  else
  {
    keep[i]  = 1; // Mark to keep
    keepCnt += 1;
  }
}

// move the entries we want to keep to the front of the vector and then
// truncate to the correct length once all of the good stuff is saved.
//
int k = 0;
int numCleared = 0
for( int ii=0; ii<c.size(); ii++ ) {
  if( keep[ii] ) {
    c[k] = c[ii];
    // We have to refresh these entries for next clearStale
    cc[k]->t_last_update = 0;
    k++;
  } else {
    numCleared++;
  }
}
c.resize( keepCnt );
delete[] keep;
```

```
    return numCleared;
  }
```

方法首先定义了参数staleThresh，这个参数硬编码为（凭经验）当前总共运行次数（c.t）的一半。这意味着在背景学习时，如果一个编号为i的码书在总共学习时间的一半时间内没有被命中，那么*i*就会被标记为待删除（keep[i]=0）。向量keep[]使我们可以对每个码书进行标记，因此它的尺寸是c.size()。变量keepCnt记录了需要保留多少个码书。在记录了要保留的码书后，我们对其进行遍历并将要保留的移到向量的前面。最后我们对向量的大小进行调整，删去所有待删除的码书。

背景差分：检测前景物体

我们已经了解了如何创建一个背景码书模型，同时了解了如何清除很少被命中的元素。接下来我们考虑backgroundDiff()，我们用它来根据先前学习到的模型对图片进行前景分割：

```
// Given a pixel and a codebook, determine whether the pixel is
// covered by the codebook
//
// NOTES:
// minMod and maxMod must have length numChannels,
// e.g. 3 channels => minMod[3], maxMod[3]. There is one min and
//      one max threshold per channel.
//
uchar backgroundDiff(      // return 0 => background, 255 => foreground
  const cv::Vec3b& p,      // Pixel (YUV)
  CodeBook&  c,            // Codebook
  int        numChannels,  // Number of channels we are testing
  int*       minMod,       // Add this (possibly negative) number onto max level
                           //   when determining whether new pixel is foreground
  int*       maxMod        // Subtract this (possibly negative) number from min
                           //   level when determining whether new pixel is
                           //   foreground
) {
  int matchChannel;

  // SEE IF THIS FITS AN EXISTING CODEWORD
  //
  for( int i=0; i<c.size(); i++ ) {
    matchChannel = 0;
    for( int n=0; n<numChannels; n++ ) {
      if(
        (c[i].min[n] - minMod[n] <= p[n] ) && (p[n] <= c[i].max[n] + maxMod[n])
      ) {
        matchChannel++; // Found an entry for this channel
      } else {
        break;
      }
    }
```

```
        if(matchChannel == numChannels) {
          break; // Found an entry that matched all channels
        }
      }

      if( i >= c.size() ) return 0;
      return 255;
    }
```

背景差分函数和学习方法updateCodebook()一样有一个内层循环，但前者对学习到每个码书的盒子的max和min阈值又加上了一个允许偏离的阈值maxMod和minMod。如果像素落在盒子阈值加maxMod或是减minMod之间，matchChannel计数器会增加。在matchChannel等于通道的个数时，我们已经遍历了所有维度以确认该点和背景相匹配。如果像素点没在学习到的盒子中，返回255（检测到前景）；否则返回0（检测到背景）。

三个函数updateCodebook()、clearStaleEntries()和backgroundDiff()构成的码书法用来使用学习到的背景模型从图片中分割前景。

使用码书法的背景模型

为了使用码书背景分割法，我们一般要执行下列步骤。

1. 在几秒或几分钟的视频上使用updateCodebook()学习基本的背景模型。
2. 使用clearStaleEntries()进行陈旧实体清除。
3. 调整minMod和maxMod使其已知的前景分割良好。
4. 维护一个高层场景模型（正如先前提到的）。
5. 通过backgroundDiff()和学习到的模型来对前景物体进行分割。
6. 周期性更新学习到的背景像素。
7. 以更低的频率使用clearStaleEntries()周期性地清理陈旧实体。

关于码书法的其他想法

一般来说，码书法在很多时候都可以取得比较好的效果，也可以相对较快地进行训练和运行。但是它无法处理光线模式的变化，例如早上，中午和晚上的日照，或是在室内突然开灯或关灯。我们可以使用多个不同的码书法来处理这种全局性的变化，每种情况一种，接下来视情况而定应该使用哪个模型。

使用连通分量进行前景清理

在开始对平均法和码书法的对比之前，我们先来讨论如何使用连通分量分析来对原始分割后的图片进行清理。这种分析非常适合有噪声的掩膜图片。

这种方法的想法是使用形态学开运算来将小的噪音区域收缩至0，接下来使用形态学闭运算重建开运算中丢失区域。接下来，我们在结果中寻找“足够大”的轮廓并计算这样区域的一些统计量。最后我们选取最大的轮廓或是所有在某个阈值之上的轮廓作为结果。在接下来的方法中，我们实现了大多数你想要的功能：

- 通过多边形或凸包来近似剩余的连通分量的轮廓；
- 设置需要保留多大的连通分量；
- 返回剩余联通分量的边界框；
- 返回剩余连通分量的中心。

实现上述操作的函数头如示例15-5所示：

示例15-5：使用连通分量进行清理

```
// This cleans up the foreground segmentation mask derived from calls
// to backgroundDiff
//
void findConnectedComponents(
  cv::Mat&   mask,                // Is a grayscale (8-bit depth) "raw" mask image
                                  // that will be cleaned up
  int        poly1_hull0 = 1,     // If set, approximate connected component by
                                  //  (DEFAULT) polygon, or else convex hull (0)
  float      perimScale  = 4,     // Len = (width+height)/perimScale. If contour
                                  //  len < this, delete that contour (DEFAULT: 4)
  vector<cv::Rect>&  bbs          // Ref to bounding box rectangle return vector
  vector<cv::Point>& centers      // Ref to contour centers return vector
);
```

函数体如下所示。首先我们按顺序进行形态学来运算和闭运算来清除小的像素噪音并重建在开运算中被腐蚀掉的区域。该方法依赖于两个使用#define硬编码的额外参数，你几乎不需要更改这两个硬编码的值。它们用来控制一个前景物体轮廓值得复杂程度（值越大复杂程度越低）和应该进行多少次形态学运算；运算的次数越多，开运算中腐蚀越严重[注14]。大腐蚀可以删去大斑点，但也会腐蚀大区域的边界。同样的示例代码中，参数工作良好，但你可也以试着调整它们：

```
// polygons will be simplified using DP algorithm with 'epsilon' a fixed
// fraction of the polygon's length. This number is that divisor.
```

注14：注意，CVCLOSE_ITR实际上依赖于图像的分辨率，如果图像分辨率非常高，使用1可能不会产生令人满意的效果。

```
//
#define DP_EPSILON_DENOMINATOR 20.0

// How many iterations of erosion and/or dilation there should be
//
#define CVCLOSE_ITR 1
```

我们现在来讨论连通分量算法。方法的第一部分是进行形态学开闭运算：

```
void findConnectedComponents(
  cv::Mat&   mask,
  int        poly1_hull0,
  float      perimScale,
  vector<cv::Rect>&  bbs,
  vector<cv::Point>& centers
) {

  // CLEAN UP RAW MASK
  //
  cv::morphologyEx(
    mask, mask, cv::MOP_OPEN,  cv::Mat(), cv::Point(-1,-1), CVCLOSE_ITR
  );
  cv::morphologyEx(
    mask, mask, cv::MOP_CLOSE, cv::Mat(), cv::Point(-1,-1), CVCLOSE_ITR
  );
```

现在噪音已被从掩膜中移去，我们来寻找所有的轮廓：

```
  // FIND CONTOURS AROUND ONLY BIGGER REGIONS
  //
  vector< vector<cv::Point> > contours_all; // all contours found
  vector< vector<cv::Point> > contours;     // just the ones we want to keep
  cv::findContours(
    mask,
    contours_all,
    CV_RETR_EXTERNAL,
    CV_CHAIN_APPROX_SIMPLE
  );
```

接下来，我们删去过小的轮廓并对剩余的使用多边形或是凸包来近似：

```
  for(
    vector< vector<cv::Point> >::iterator c = contours_all.begin();
    c != contours.end();
    ++c
  ) {

    // length of this contour
    //
    int len = cv::arcLength( *c, true );

    // length threshold a fraction of image perimeter
```

```
//
double q = (mask.rows + mask.cols) / DP_EPSILON_DENOMINATOR;

if( len >= q ) {        // If the contour is long enough to keep...

  vector<cv::Point> c_new;
  if( poly1_hull0 ) {   // If the caller wants results as reduced polygons...
    cv::approxPolyDP( *c, c_new, len/20.0, true );
  } else {              // Convex Hull of the segmentation
    cv::convexHull( *c, c_new );
  }
    contours.push_back(c_new );

  }

}
```

上面的代码中，我们使用Douglas-Peucker近似算法来减少多边形（如果用户没要求仅仅返回凸包）。做完这些会产生一个新的轮廓列表。在将轮廓绘制回掩膜前，我们定义一些简单的绘图颜色：

```
// Just some convenience variables
const cv::Scalar CVX_WHITE   = cv::RGB(0xff,0xff,0xff);
const cv::Scalar CVX_BLACK   = cv::RGB(0x00,0x00,0x00);
```

我们在接下来的代码中使用这些颜色，我们首先单独分析每个轮廓，接下来将掩膜置零并将清理后的轮廓集绘制到掩膜上。

```
// CALC CENTER OF MASS AND/OR BOUNDING RECTANGLES
//
int idx = 0;
cv::Moments moments;
cv::Mat scratch = mask.clone();
for(
  vector< vector<cv::Point> >::iterator c = contours.begin();
  c != contours.end;
  c++, idx++
) {

  cv::drawContours( scratch, contours, idx, CVX_WHITE, CV_FILLED );

  // Find the center of each contour
  //
  moments = cv::moments( scratch, true );
  cv::Point p;
  p.x = (int)( moments.m10 / moments.m00 );
  p.y = (int)( moments.m01 / moments.m00 );
  centers.push_back(p);

  bbs.push_back( cv::boundingRect(c) );

  Scratch.setTo( 0 );
```

```
    }

    // PAINT THE FOUND REGIONS BACK INTO THE IMAGE
    //
    mask.setTo( 0 );
    cv::drawContours( mask, contours, -1, CVX_WHITE );

  }
```

最终我们获得了一个从原始带噪音的掩膜中创建清理后的掩膜的方法。注意，OpenCV3中的新函数cv::connectedComponentsWithStats()可以在cv::findContours()之前使用，用来对小连通分量进行标记和移除。

小测试

这一节我们首先给出一个例子来说明在真实视频中方法的工作效果。让我们继续使用室外有树场景的那个视频。注意，图15-1中某些时刻一只手穿过了场景。使用一些类似帧间差分（先前介绍过）的技巧可以很容易找出这只手。帧间差分最基本的想法是将当前帧和某些“滞后”的帧相减并对结果进行二值化。

视频中连续的帧应该是相似的，所以可以认为，如果我们将当前帧和“滞后”的帧简单的做差，除了在场景中移动的前景物体[注15]外不会有其他大的差别。但是“没有太大的差别”究竟是什么意思呢？事实上，意味着“只有噪音”。因此在实际中，我们面临的问题是在前景真的存在时从带有噪音的图像中找出前景出现的信号。

为了更好地理解这种噪音，首先考虑从视频中提取的一对没有前景的图片。图15-6展示了视频中这样的常见的帧（左上）和其前一帧（右上）。该图也展示了以15为阈值对其进行帧间差分的结果（左下）。你可以看出从树叶的移动产生了很多噪音，然而使用连通分量法可以将这些杂散的噪音去掉[注16]（右下）。这并不令人惊奇，因为没理由相信这些杂散的噪音有任何空间上的关联，因此它们的信号应该分布在一群非常小的区域上。

现在让我们考虑有前景物体（我们无处不在的手）经过的情况。图15-7给出了和图15-6相似的两帧，区别是现在有一只从左向右移动的手。如往常一样，当前帧（左上）和上一帧（右上）和帧间差分的结果（左下）和经过连通分量清理后的还不错的结果（右下）都给出了。

注15：在使用背景差分方法时，一个物体被判别为“前景”主要是由于它的速度，在静态场景中或是前景物体相比背景物体更靠近相机时（因此相机的射影几何上来看其看起来移动的比较快），这是合理的。

注16：连通分量中的阈值被调整为对这样没有前景物体的帧给出零响应。真正的问题是，感兴趣的前景物体是否能在这样的阈值下被成功检测到。我们会发现效果还是不错的（图15-8）。

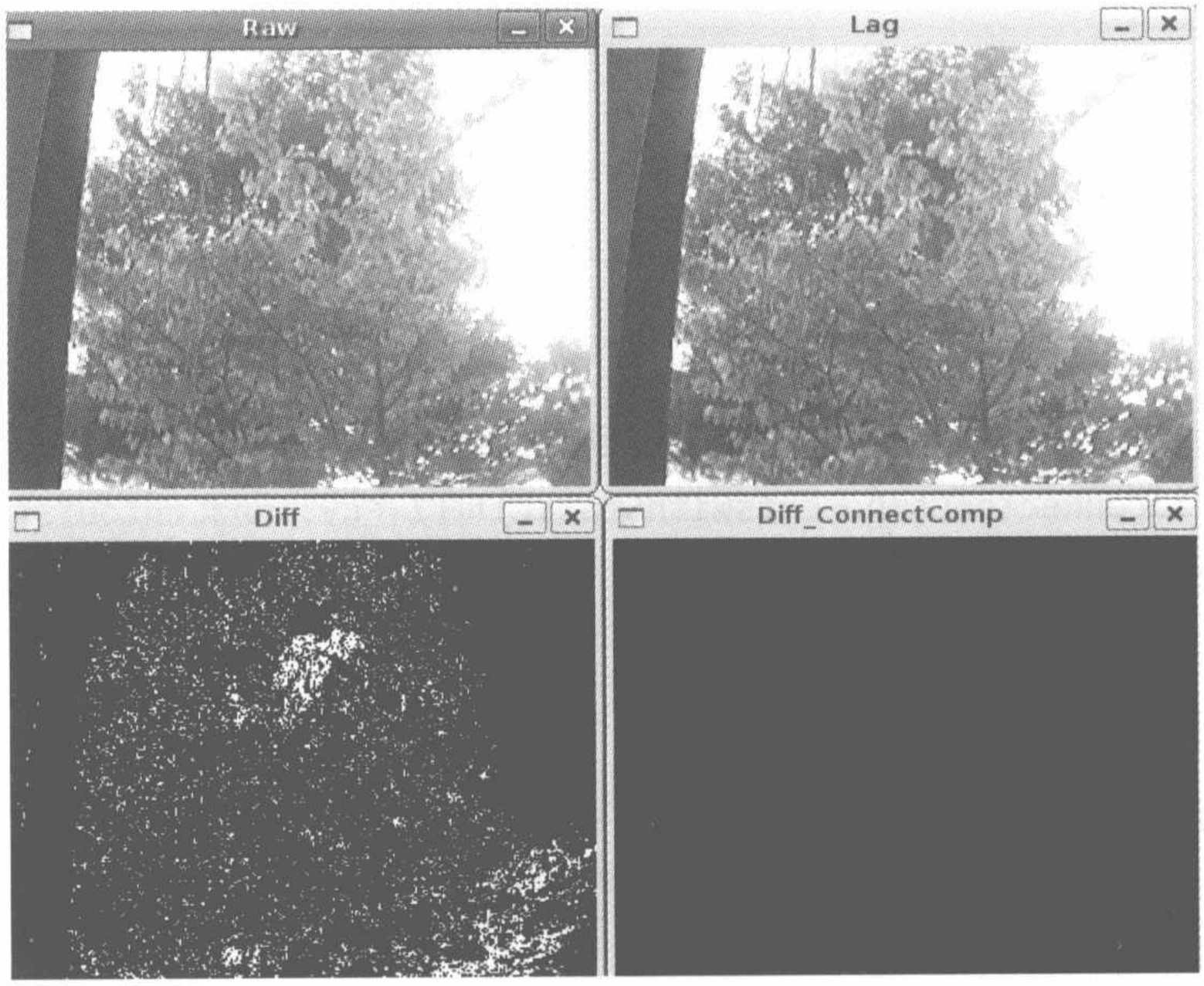


图15-6. 帧间差分：当前帧（左上）和上一帧（右上）的背景中有正在飘舞的树；图像的差分（左下）被连通分量法完全清理干净（右下）

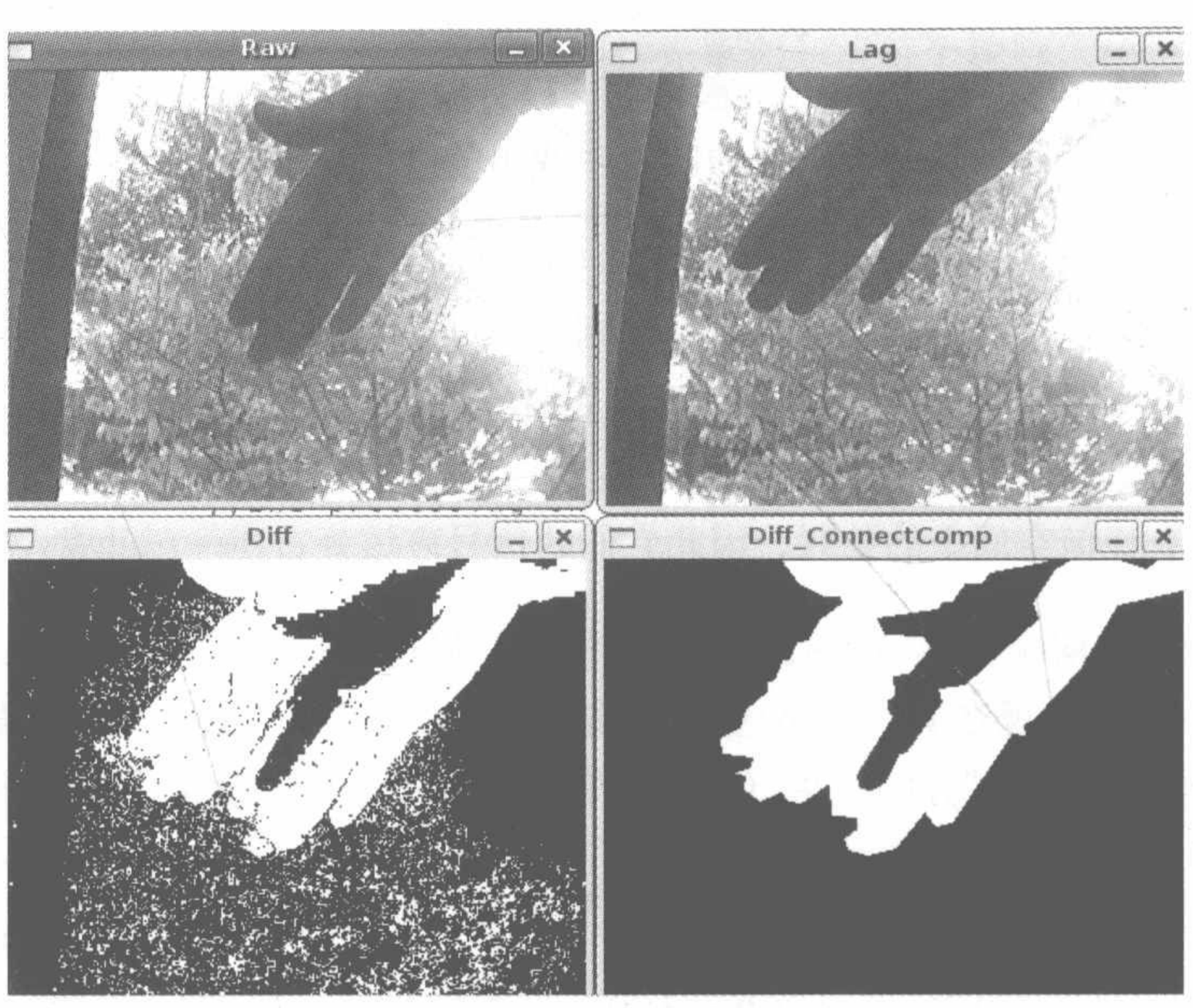


图15-7. 使用帧间差分法检测一只手，该手作为前景物体从左向右移动（上方两图）；差分图（左下）展示了左侧“空洞”（手原来所在的位置）和右侧它的当前位置，使用连通分量法清理的图像（右下）给出了清理后的差分结果

我们也可以清晰地看出帧间差分的一个缺点：它不能区分哪里的物体移走了（那个“空洞”）以及哪里是物体的新位置。而且在两个区域重叠的位置经常有一个缝隙存在，因为“肤色和肤色的差”是0（或是在阈值以下）。

由此可见，使用连通分量来进行背景提取中噪音的清理是一个很强大的技巧。同时我们也顺便了解了帧间差分方法的能力和缺点。

两种背景方法的对比

前面讨论了两类背景建模方法：平均距离法（和其变种）和码书法。你可能不清楚哪种方法更好一点，或是什么时候使用什么方法。在这种情况之下，最好的方法就是简单粗暴地都试一试[注17]。

我们会继续使用在这章中一直使用的带有树的视频。除了移动的树，这个视频中还有很多从建筑物左侧和右侧射出的光。这使得对该背景进行建模十分具有挑战性。

在图15-8中，我们对平均差法（上方）和码书法（下方）进行了对比；左侧是原始的前景分割图像而右侧是使用连通分量进行清理后的图像。可以看出，平均差法产生了一个很稀疏的掩膜并将手打散为两个连通分量。这并不太令人惊奇，在图15-2中，我们看出使用平均差法构造的背景中经常包含手部的颜色（图中虚线）。和图15-5对比，码书法对树叶树枝的波动有更精确的建模效果，可以很准确地检测出前景中手的像素（虚线）。图15-8不仅说明了背景模型有更少的噪音，还展示了连通分量法可以给出一个很准确的物体轮廓。

OpenCV中的背景提取方法的封装

目前为止，我们已经详细了解了如何实现自己的比较基本的背景提取算法。这样做的优点是对每件事情到底是怎么发生的会有一个清楚的认识。但缺点是随着算法的不断发展，有越来越新越来越好的算法被开发出来，虽然这些算法基于相同的基础想法，但它们十分复杂以至于你会更喜欢把它们视为一些“黑盒子”来使用它们而无需了解它们的细节。

为此，OpenCV提供了一种通用的、基于类的接口来进行背景提取。目前为止，已经有两种算法实现使用了这个接口，未来会有越来越多的算法使用这个接口。在这一节中，我们首先给出接口的一般形式，然后讨论两种可用的算法实现。这两个实现都是基于高

注17：对缺少经验的的人来说，都试一试是很好的，可以在一个预定义好的数据集上对多个算法进行比较。

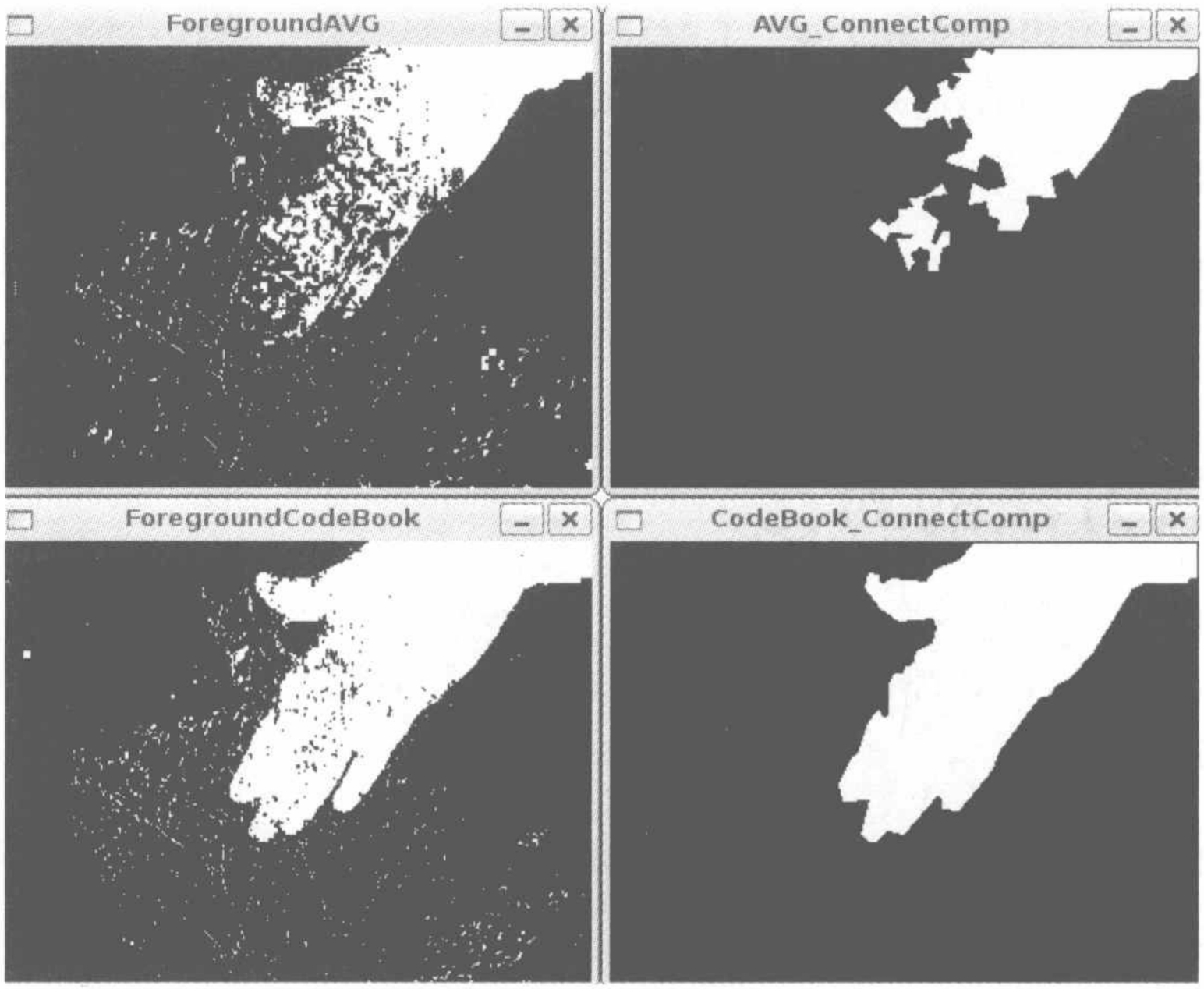


图15-8. 使用平均法（上一行），连通分量清理破坏了手指（上右）；使用码书法（下一行）在分割上和使用连通分量法清理（下右）上得到了更好的效果

斯混合模型（MOG）的，这种统计模型结合了我们先前介绍的最简单的背景模型和码书方法可以处理多模态数据的能力。这两种MOG方法都是21世纪提出的算法，普遍适用于很多日常实际情况。

cv::BackgroundSubstractor基类

cv::BackgroundSubstractor（抽象）基类[注18]只给出了最少的必要方法。它的定义如下：

```
class cv::BackgroundSubtractor {

public:
  virtual ~BackgroundSubtractor();
  virtual void apply()(
    cv::InputArray  image,
    cv::OutputArray fgmask,
    double          learningRate = -1
  );
```

注18：事实上，这个基类并不是一个抽象基类（就是说它不包含任何纯虚函数）。然而，在库中它总是作为一个抽象基类来使用的，这意味着即使编译器允许你对其进行实例化，你也不想，也不应该这么做。我们本来想使用“相对抽象”这个词，但后来改变了想法。

```
    virtual void getBackgroundImage(
      cv::OutputArray backgroundImage
    ) const;

};
```

如你所见，在构造函数之后只定义了两个函数。第一个是函数算子，用来进行处理新图像并针对该图像计算前景掩膜。第二个函数对背景模型生成一个图片描述。生成的这个图片主要是为了可视化和调试；注意在背景模型中，有很多信息每个像素相关联而不仅仅是一个颜色。因此，由getBackgroundImage()生成的结果图片仅仅是现有背景模型信息的部分表达。

看起来这些方法中没有通过累加背景图片进行训练的函数，这好像是一个明显的设计上的疏漏。其原因是，在学术界一个（相对）一致的观点是任何背景提取算法有应该能够连续进行训练。这样要求算法的原因有很多，最明显的是场景中的光线照明效果可能会逐渐变化（例如，室外太阳的升起和落下，阳光透过窗户照进室内）。而一个比较微妙的原因是在很多实际的场景中，不能找到一段较长的没有前景物体的时间段来训练算法。类似的，很多时候背景中的物体最终是会移动的（例如一个停车场），那么这些物体就会在之后的检测中永久地留下一个“空洞”（例如，一个曾经停在那里的车开走了）。出于这些原因，几乎所有的现代背景提取算法都不再区分训练模式和运行模式；它们会连续地训练并构造背景模型，不断从背景中删去很少出现的物体（再出现会被视为前景），保留大多数时候都出现的物体（这些被视为背景）。

KB方法

可用的算法中，第一个是KB方法（KaewTraKuPong and Bowden Method），它带给我们一些处理真实世界中很有挑战的背景建模问题的新能力：一个多模态模型，连续在线训练，两个用来提高初始化性能的训练模式和显式的阴影检测和去除[KaewTraKulPong2001]。几乎所有这些对用户来说都是不可见的，然而不出所料，算法中有一些需要根据特定应用进行微调的参数。它们是历史参数(history)、高斯混合数（number of Gaussian mixtures）、背景比率（background ratio）和噪音强度(noise strength)[注19]。

第一个参数历史参数，用来设定合适算法从初始化模式切换为正常运行模式。该参数的默认值为200帧。混合高斯数是指用来对每个像素进行背景建模的时候，混合高斯模型中高斯分量的数量，该参数的默认值是5。

注19：如果你正在看该算法的论文，这前三个参数——历史参数、高斯混合数和背景比率——在论文中对应着L，K和T。最后一个噪音强度可以认为是新建高斯分量初始化时的Θk的值。

对于模型中每个高斯分量，每个分量都配有一个权重，该权重表示像素以多大比例使用当前分量解释。这些分量不一定全是“背景”；一些可能是前景物体恰好在该点经过。通过将这些高斯分量按权重排序，背景可能会使用其前*b*个分量进行建模，其中*b*是满足权重加起来大于某个以百分数表示的界的最小分量数。这个百分数被背景比率，默认值为0.7（或70%）。举个例子，如果有五个分量，其权重为0.40，0.25，0.20，0.10和0.05，那么*b*应该为3，因为至少需要前三个分量0.40+0.25+0.20才能超过需要的0.7背景比率。

最后一个参数是噪音强度，设定的是新高斯分量构造时它的初始化不确定度。当一个未能解释的像素值出现时就会构造新高斯分量，这包括两种情况：要么还没分配完所有分量，要么该新像素值无法被任何存在的分量所解释（此时最没有价值的分量会被循环使用，以便为新信息腾出空间）。在实际使用中，增加噪音强度是希望使这些高斯分量能“解释”更多信息。当然，这其中的权衡便是他们可能会“解释”过多还没观察到的东西。该噪音强度的默认值是15（以0～255的像素亮度值为单位）。

cv::BackgroundSubstractorMOG

KB背景分割算法的对象实现和通用接口的主要区别是它的构造函数。你或许已经注意到了这个构造函数没有在基类中定义，因为每个算法实现都有其自己的参数需要设定，这是没有通用的原型的。该构造函数的原型是*opencv_contrib*库中的bgsegm模块中的cv::bgsegm::BackgroundSubstractorMOG类，如下所示：

```
cv::Ptr<cv::bgsegm::BackgroundSubtractorMOG>
    cv::bgsegm::createBackgroundSubtractorMOG(
        int history = 200,
        int nmixtures = 5,
        double backgroundRatio = 0.7,
        double noiseSigma = 0
    );
```

这个构造函数允许你设置算法需要的那四个参数，而同时，你也可以使用默认的构造函数，它将这四个值设置为先前提到的默认值（200，5，0.70和15）。

Zivkovic方法

第二个背景提取方法是Zivkovic方法，它和KB算法一样都使用高斯混合模型对某个特定像素的颜色进行建模。然而，一个特别要注意的区别是Zivkovic方法没有使用固定数量的高斯分量，而是使用了一个能够给观察到数据最好解释的动态变化的分量数量[Zivkovic04, Zivkovic06]。这样处理的缺点是高斯分量越多，进行模型比较和更新所需要的时间也就越多。另一方面来说，这也使得模型具有更高的保真度。

这个算法和KB算法相比，有一些相同的参数，但同时也引入了一些新参数。幸运的是，这其中只有两个参数比较重要，而其他参数在大多数时候我们都选择设为默认值。这两个特别重要的参数是历史参数（又称为“衰减参数”）和方差阈值。

第一个参数*history*，作用是设置每个像素的颜色能持续发挥作用的时间长度。从本质上讲，这代表了该像素的影响衰减到0持续的时间。这一衰减周期的默认值是500帧。500帧的时长大约就是一次量测被“遗忘”的时间。但是如果我们再深入一点，在这个算法内部，更准确的描述是把这个值看做一个指数衰减参数（decay parameter）$\alpha = 1 / 500 = 0.002$，即一次量测的影响按照$(1-\alpha)^t$衰减。

第二个参数variance threshold，对一个已知的高斯混合分量来说，用来设置对像素的一次新量测被接受为该分量的一部分的置信水平。方差阈值的单位是平方马氏距离（squared-Mahalanobis distance）。这意味着如果你想将一个像素包含进一个其中心距该像素3σ远的高斯混合分量中，你就要将方差阈值设置为`3 * 3=9`。[注20]而这个参数的默认值是`4*4=16`。

cv::BackgroundSubstractorMOG2

`cv::BackgroundSubstractorMOG2`的构造函数允许我们设置这两个最重要的参数。其他的参数可以在背景提取对象构造后再设置（这意味着它们是类中公开的成员）：

```
Ptr<BackgroundSubtractorMOG2> createBackgroundSubtractorMOG2(
  int    history       = 500,
  double varThreshold  = 16,
  bool   detectShadows = true
);
```

其中的历史参数`history`和方差阈值`varThreshold`如前所述。可选的新参数`bShadowDetection`允许我们检测图中的阴影并进行移除。当其发挥作用时，检测图中没被直接判为背景的部分是否只是暗化的背景。如果确实是暗化的背景，这些像素会被标记为一个特殊的值（通常该值和前景背景的值不同）。

除了这两个最重要的参数，还有其他参数，如果你觉得自己足够勇敢，可以试试对它们进行微调：

注20：注意，马氏距离本质上就是z值（即以分布的不确定度为单位，测量一个点距高斯分布重心的距离），它考虑了任意维度任意协方差矩阵σ的分布的复杂性。

$$r_M^2 = (\vec{x} - \vec{\mu})\Sigma^{-1}(\vec{x} - \vec{\mu})$$

你也可以从中看出为什么计算平方的马氏距离更加自然，这也是为什么要求传入$r_M{}^2$而不是r_M的理由。

```
class cv::BackgroundSubtractorMOG2 {

  ...

public:

  ...
  int    getNMixtures() const;           // Minimum number of mixture components
  void   setNMixtures( int nmixtures );

  double getBackgroundRatio() const;     // If component is significant enough
  void   setBackgroundRatio( double backgroundRatio );

  double getVarInit() const;             // Initial variance for new components
  void   setVarInit( double varInit ) const;

  double getVarMin() const;              // Smallest allowed variance
  void   setVarMin(double varMin);

  double getVarMax() const;              // Largest allowed variance
  void   setVarMax(double varMax);

  double getComplexityReductionThreshold() const; // Samples needed to prove that
                                                  //   the component exists
  void   setComplexityReductionThreshold(double CT);

  bool   getDetectShadows() const;       // true try to detect shadows
  void   setDetectShadows(bool detectShadows);

  int    getShadowValue() const;         // value of shadow pixels in
  void   setShadowValue(int value);      //   output mask

  double getShadowThreshold() const;     // Shadow threshold
  void   setShadowThreshold(double shadowThreshold);
  ...

};
```

各参数的含义如下：nmixtures表示每个像素模型最多可以拥有的高斯分量个数（默认值是5）。增加这个值可以提高模型的保真度，但同时计算时间增加。backgroundRatio和KB算法中的背景比率含义相同（在本算法中默认值为0.90）。varInit、varMin和varMax为特定高斯分量方差（σ^2）的最大和最小初始值（它们的默认值分别为15，4和75）。varInit和KB算法中noiseSigma含义相似。CT（Zivkovic等人称为“复杂度减缩先验”）和需要多少样本才能承认一个新高斯分量的个数有关。默认值为0.05，对于该值，一个可能最重要的事实是，如果已将其设置为0.00，那么整个算法会大幅简化[注21]（不仅从速度还从结果的质量）。shadowValue设置了计算出的前景图像中阴影

注21：一个更技术性的关于“大幅简化”的定义是，其实是Zivkovic的算法简化成一个和Stauffer与Grimson的算法十分相似的算法。我们不在这里讨论该算法的细节，但是该算法在Zivkovic的论文中被引用了，并在其中被作为一个证明Zivkovic算法性能提高的基准。

的值（假定构造函数中参数detectShadows被设置为true），其默认值为127。最后，shadowThreshold是用来决定一个像素是不是阴影的参数，对它的解释是其亮度相比较于模型中的亮度而言被判为阴影的阈值（例如：如果fTau为0.60，那么任何与现有高斯分量有相同颜色亮度在其0.60到1.0倍之间的像素都被视为阴影）。该参数的默认值为0.50。

小结

在这一章中，我们探讨了背景提取这个问题。该问题在实际的计算机视觉的应用中发挥着很重要的作用，从工业自动化到安保再到机器人都能看到它的身影。我们从背景提取的基本理论开始，基于简单的统计模型，导出了两个背景提取方法。同时我们也给出了如何使用连通分量分析来增强背景提取的实用性，并对这两种基本方法进行了比较。

最后以对一个OpenCV提供的一些更高级的背景提取方法结束本章。这些方法在核心思路上都和本章开始我们给出的简单方法相同，但包含一些提升使其更适用于真实世界的应用。

练习

1. 使用cv::accumulateWeighted()重新实现背景提取中的平均法。为此，从场景中学习出像素值的移动平均以获得均值，学习出绝对差的移动平均cv::absdiff()，作为图片标准差的一个替代。
2. 阴影在背景提取中是个问题，因为它总会被识别成前景。使用平均法或是码书法来学习背景提取中的背景。接下来，一个人走进前景，阴影出现在前景物体的脚下。
   a. 室外环境，阴影总是比其邻域内的物体黑而模糊，试着用这个删去它们。
   b. 室内环境，阴影总是比其邻域内的物体黑，试着用这个删去它们。
3. 本章给出的简单背景模型对它们的二值化阈值很敏感。在第17章中，我们会学习追踪动作，这可以用来检测背景和其阈值。你可以这样使用它，当一个已知的人在相机前做出“标定式行走”时：找出移动的物体并调节阈值直到前景物体和运动检测界一致。当我们知道一部分背景被遮挡时，也可以在校准对象本身（或背景上）使用不同的模式来进行检查和对参数调优进行指导。
   a. 修改代码，加入自动标定模式。学习出一个背景模型，然后将一个颜色鲜艳的物体放入场景。使用颜色来寻找该物体进而用此物体来自动设置背景分割的阈值。注意，你可以将此物体留在场景中以进行连续微调。
   b. 使用修改过的代码解决练习2中的阴影去除问题。

4. 使用背景分割方法来分割持有武器的人。探索在find_connect_components()中不同的参数和参数的默认值效果如何。对不同的配置给出结果：

   a. Poly1_hull0

   b. DP_EPSILON_DENOMINATOR

   c. PerimScale

   d. CVCLOSE_ITR

5. 以OpenCV安装目录下的*../samples/data/tree.avi*作为视频输入，比较cv::BackgroundSubtractorMOG2和cv::BackgroundSubtractorMOG2对移动的手的分割效果。使用视频的第一部分学习背景，使用其余部分进行分割。

6. 虽然可能很慢，但试一下在进行背景分割前先对图像使用bilateralFilter()进行滤波。这意味着背景在被平滑后再进行背景学习，通过码书背景分割方法进行前景分割测试。

   a. 将结果和不进行双边滤波进行对比

   b. 试着有计划地对双边滤波中的spatialSigma和colorSigma进行调节（即使用滑块来进行交互式的调节）。比较结果。

第16章

关键点和描述子

关键点和跟踪基础

本章将介绍图像中富含信息的特征点。我们将首先描述所谓的角点的概念，并在亚像素域中理解它们的定义。然后，我们将学习如何用光流跟踪这些角点。角点的跟踪随着研究发展逐渐演变成关键点的理论。我们将在本章的其余部分对它进行介绍，并对OpenCV库中实现的关键点特征检测器和描述符进行更广泛的讨论[注1]。

角点的概念以及关键点的概念是基于直观感受的。在许多应用中，它们能够在其他含有相同场景或目标的相似图像中以一种相同的或至少非常相似的不变形式表示图像或目标。角点和关键点的表示能力很强大。一个角点是一小块（含有丰富局部信息的）图像，因此能在另一幅图像中识别。而关键点，是这种概念的扩展，它对来自图像的小的局部像斑的信息进行编码，使得关键点具有高度可辨别性，并且至少在原理上基本是唯一的。关于关键点的描述性信息，被概括成其描述符的形式，其描述符的维度通常比形成关键点的像素像斑低得多。描述符表示该像斑，以便在另一个不同的图像中出现时更容易识别该像斑。

从直观的角度来看，你可以把一个关键点想象成一幅拼图中的一块。当你开始拼图的时候，一些片段很容易被辨别出来，比如：门的手柄，脸，教堂的尖顶。当你组装拼图时，可以立即将这些关键点与拼图盒上的图像关联起来，并立即知道该把它们放到哪

注1：在本章中，我们把重点放在主函数库中拥有授权的特征。xfeatures2D模块中还有更多的“非自由”特征(即提出者保留版权)、特征检测器和描述符（参见附录B）以及新的实验特征。我们将在本章中介绍SIFT和SURF这两个非自由特征，因为它们有很好的历史意义、实践意义和教学意义。

里。另外，如果你和一个朋友有两幅没有拼好的拼图，你想知道它们是否是来自美丽的新天鹅堡的两幅不同的图像区块，你可以组装两个拼图并比较，但是也可以选择每个拼图最突出的部分进行比较。在后一种情况下，你只需要比较几次就能确定它们是同一个拼图，还是由同一城堡的两幅不同图像制成的两张拼图。

在本章中，我们将首先讨论关键点理论的基础，讨论现代关键点的最早的起源：哈尔角点。从这儿开始我们将讨论这种角点的光流的概念，它包含了在视频序列中将这些特征从一帧跟踪到另一帧的基本思想。之后，我们将继续使用更现代的关键点及其描述符，并讨论OpenCV如何帮助我们找到它们以及OpenCV库如何帮助我们在帧之间进行匹配。最后，我们将借助一个简单的方法，将图像上检测到的关键点可视化。

角点检测

你可以跟踪很多局部特征。当然也很值得花点儿时间来考虑这样的特征到底是什么样子。显然，如果我们在一个白墙上选择一个白点进行跟踪，那么要在视频的下一帧中再找到这一个点，就很不容易。

如果墙上的所有点非常相似甚至相同，那么我们在后续的帧中跟踪这一点就会比较困难。另一方面，如果我们选择一个独一无二的点，那么我们再次跟踪到这一点的可能就很大。在实际操作中，选择的特征点应该是唯一的，或几乎是唯一的，并且应该是可参数化的，以至于它可以与其他图像中的其他点进行比较（参见图16-1）。

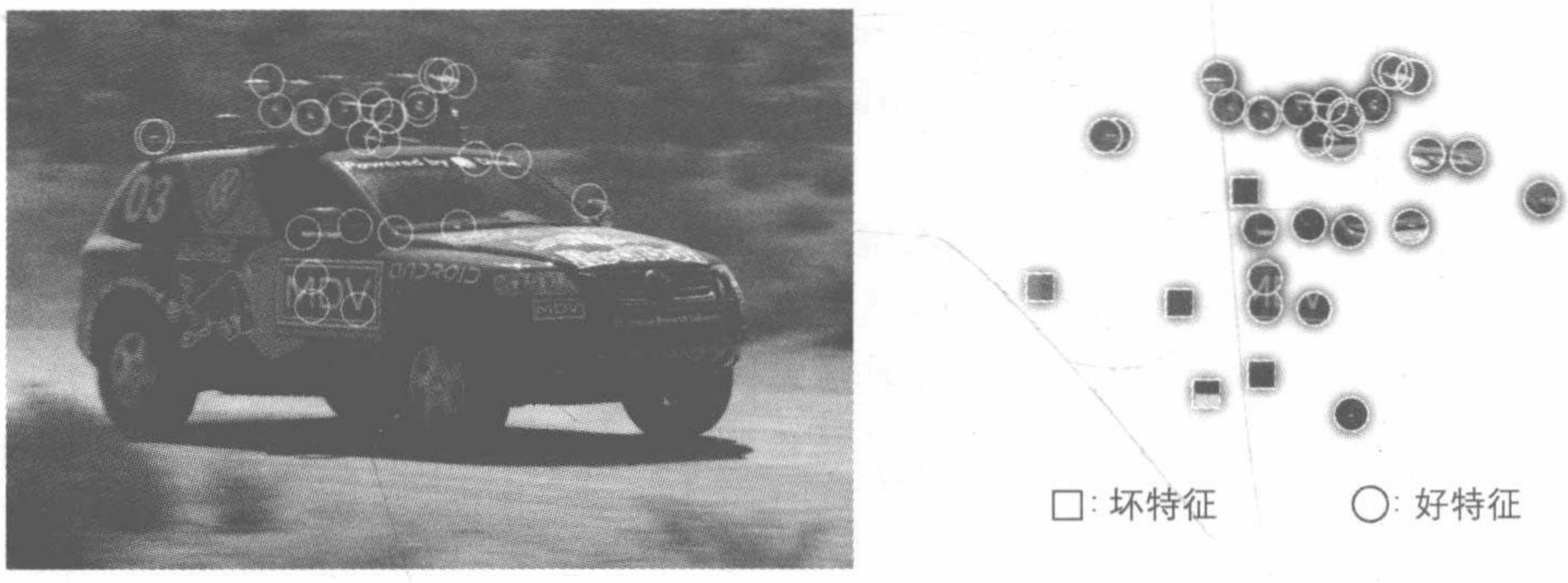


图16-1. 被圆圈标记的点是好的追踪点，而标有方框的点（即尖锐边界的点）都是差的选择

从白墙回到直觉，我们可能会试图寻找一些变化比较明显的点，例如响应很强的微分点。事实证明，这还不够，仅仅是一个开始。强微分点在某种边缘的可能性更大，但它仍然可能看起来像沿着同一边缘的所有其他点，参见图16-8中的孔径问题，后面介绍光流时也有相关讨论。

然而，如果在两个不同的方向附近观察到强微分，那么我们认为这一点就更有可能是独一无二的。因此，许多可追踪的特征才被称为角点。直观上看，角点（而不是边缘点）是从一帧到下一个帧之间包含足够信息的点。

哈尔[Harris88]提供了角点最普遍的定义。这个定义以数学的形式描述了前面所讲的直觉。我们稍后会看一下这个方法的细节，但是现在重要的是你可以用OpenCV在图像中找到被跟踪的好的候选点，然后用哈尔的方法确定它们。

使用cv::goodFeaturesToTrack()寻找角点

cv::goodFeaturesToTrack()例程实现了哈尔的方法，并在性能上略有改善（Shi和Tomasi [Shi94]）。该函数便于计算必要的导数运算符，对其进行分析，并返回符合我们定义的适合跟踪的点的列表：

```
void cv::goodFeaturesToTrack(
  cv::InputArray  image,                          // Input, CV_8UC1 or CV_32FC1
  cv::OutputArray corners,                        // Output vector of corners
  int             maxCorners,                     // Keep this many corners
  double          qualityLevel,                   // (fraction) rel to best
  double          minDistance,                    // Discard corner this close
  cv::InputArray  mask              = noArray(), // Ignore corners where mask=0
  int             blockSize         = 3,         // Neighborhood used
  bool            useHarrisDetector = false,     // false='Shi Tomasi metric'
  double          k                 = 0.04       // Used for Harris metric
);
```

输入图像可以是任何8位或32位（即8U或32F）的单通道图像。输出角点将是包含所有找到的角点的向量或数组（取决于你提供的内容）。如果它是向量vector<>形式的，它应该是cv::Point2f对象的vector。如果它是一个cv::Mat，它的一行将是角点，两列分别是点的*x*和*y*的位置。你可以使用maxCorners限制找到的角点数量，用quality Level来设置点的（通常介于0.10和0.01之间，绝对不超过1.0）的返回质量以及用minDistance设置相邻角之间的最小间距。

如果要设置mask参数，必须与图像的尺寸相同。角点将不会在mask为0的地方生成。blockSize参数表示计算角点时需要考虑的区域大小，默认值为3，但对于高分辨率图像，你可能想让它再稍微大一点。useHarrisDetector参数（如果设置为true）将设定cv::goodFeaturesToTrack()使用哈尔原始算法的精确角点强度公式；如果设置为false，将使用Shi和Tomasi的方法。参数k只能用于哈尔算法，并且最好保留其默认值。[注2]

注2：在本章的后面，我们将介绍许多计算关键点的方法，这些方法基本上是对角点的总结。在这一节中，我们将详细讨论许多关键点查找算法，包括哈尔算法。我们将对这些算法和这些参数进行更详细的描述。

亚像素角点

如果处理图像来提取几何测量值，而不是提取要识别的特征，那么通常需要（比cv::goodFeaturesToTrack()提供的简单像素值）更高的分辨率。另一种说法是这样的像素带有整数坐标，而我们有时需要实数值坐标——例如（8.25,117.16）——的像素位置。

如果你想找一幅相机图像中一个特定的小目标，比如一个遥远的星星，那么你总会感到沮丧，因为这个点的位置几乎不会恰好就在相机像素的中心。当然，在这种情况下，来自目标的一些光也将出现在相邻的像素中。为了克服这个问题，可以尝试将曲线拟合到图像像素值，然后使用一点数学运算来确定像素之间出现峰值的位置。亚像素角点检测技术都依赖于这种方法（对于相关评价和较新的技术，请参见Lucchese [Lucchese02]和Chen [Chen05]）。这种测量的常见用途包括跟踪三维重建、相机标定、扭曲场景的部分重叠视图以最自然的方式将它们拼接在一起，以及找到比如建筑物在卫星中的精确位置的外部信号图片。

用于亚像素细化的最常见的技巧之一是基于向量与其正交向量之间的点积为0的数学发现。这种情况发生在角点处，如图16-2所示。

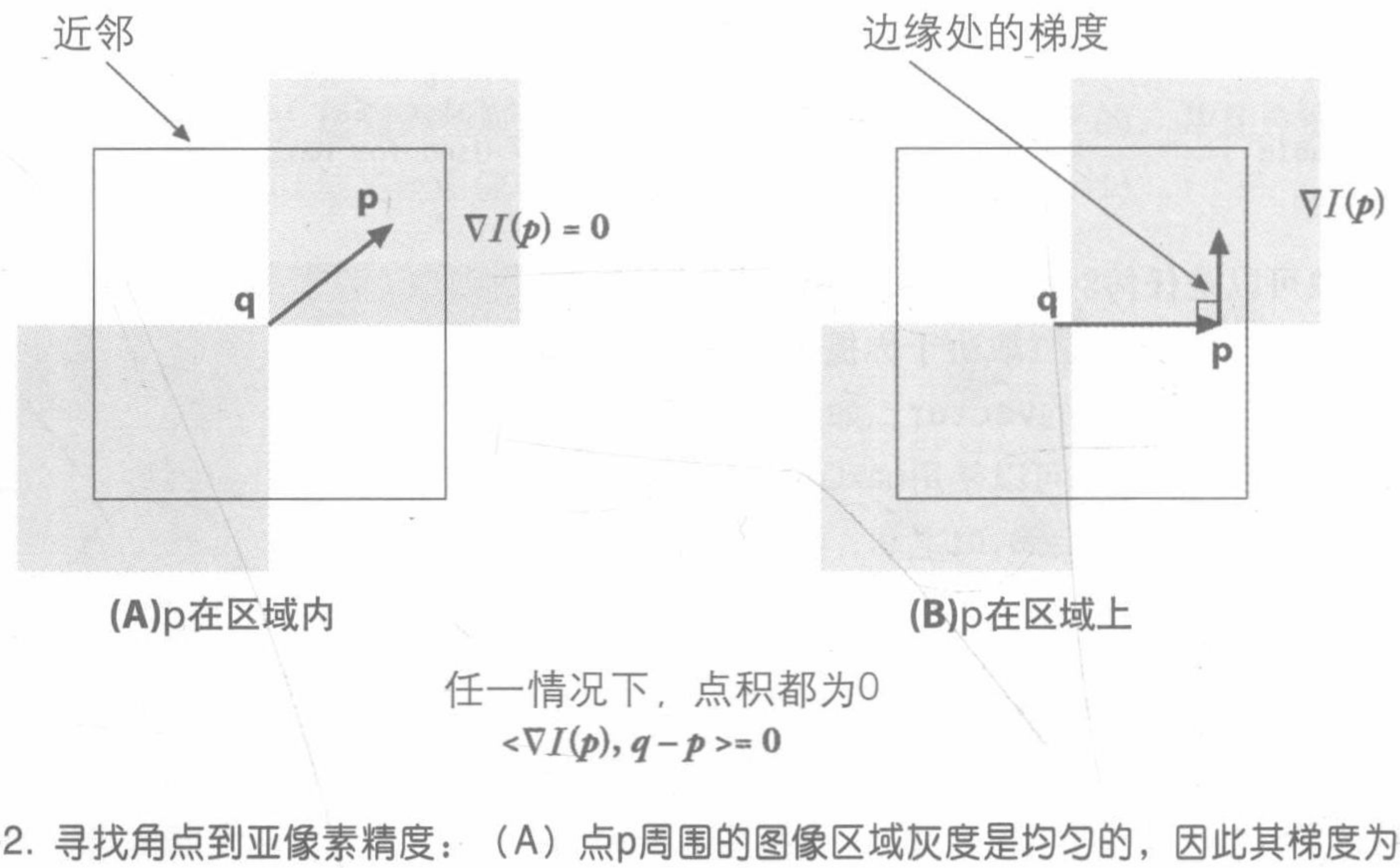


图16-2. 寻找角点到亚像素精度：（A）点p周围的图像区域灰度是均匀的，因此其梯度为0；（B）沿边缘的梯度与边缘的向量q-p正交。在任一情况下，p处的梯度和向量q-p之间的点积为0（参见正文）

在图16-2中，我们假设起始角点位置q在实际亚像素角点位置附近。我们检查点q开始以p结尾的向量。当p在附近的均匀或“平坦”区域时，梯度为0。另一方面，如果向量q-p与边缘对齐，则该边缘处的p处的梯度与向量q-p正交。在任一情况下，p处的梯度和向量q-p之间的点积为0。我们可以在附近的点p和相关向量q-p上组合许多这样的梯度对，将

它们的点积设置为0，并且求解这些组合的方程组；这个解将为q的准确位置产生更准确的亚像素位置。

寻找亚像素角点的函数是cv::cornerSubPix()：

```
void cv::cornerSubPix(
  cv::InputArray       image,                  // Input image
  cv::InputOutputArray corners,                // Guesses in, and results out
  cv::Size             winSize,                // Area is NXN; N=(winSize*2+1)
  cv::Size             zeroZone,               // Size(-1,-1) to ignore
  cv::TermCriteria     criteria                // When to stop refinement
);
```

输入image是计算角点的原始图像。corner数组包含整数形式的像素位置，例如从例程cv::goodFeaturesToTrack()获取的像素位置，它们被设为角点位置的初始值。

正如之前所说，亚像素位置的实际计算使用了点积表达式的系统，该系统表示一个和为零的组合（参见图16-2）。其中每个等式都考虑了p的周围区域的像素。参数winSize指定将生成这些方程的窗口的大小。该窗口以整数形式的原始角点位置为中心，并在每个方向向外扩展，扩展距离为通过winSize指定的像素数,例如，如果winSize.width = 4，则搜索区域实际上为4 + 1 + 4 = 9像素宽。这些方程形成一个可以通过一个自相关矩阵的逆求解的线性系统。[注3]在实践中，由于从非常接近p的像素产生的小特征值，该矩阵并不总是可逆的。为了防止这种情况，通常只考虑在p附近的像素。参数zeroZone定义了一个窗口（类似于winSize，但范围较小），这在约束方程系统中将不被考虑，因此也就不再考虑自相关矩阵。如果不需要这样的零区域，则该参数应设置为cv::Size(-1, -1)。

一旦为q找到一个新的位置，算法将使用该值作为起始点进行迭代，并直到达到用户指定的终止标准。回想一下，这个标准可以是cv::TermCriteria::MAX_ITER类型或cv::TermCriteria::EPS类型（或两者），通常用cv::TermCriteria()函数构造。使用cv::TermCriteria::EPS将有效表示你需要的亚像素值的准确性。因此，如果指定0.10，则要求亚像素精度在像素的1/10以内。

光流简介

光流问题涉及尝试找出一幅图像中的许多（可能全部）点在第二幅图像中移动的位置——通常这是以视频序列完成的，因此可以合理假定第一幅图像中的大部分点框架都可以在第二幅位置找到。光流可以用于场景中的物体的运动估计，甚至用于相机相对整个

注3：之后，我们将在哈尔角点内部运行的上下文中见到另一个自相关矩阵。然而，这两者并不相关。

场景的自运动估计。在诸如视频安全的许多应用中，运动本身表明场景的一部分具有特定的兴趣，或者有趣的事情正在发生。图16-3对光流进行了说明。

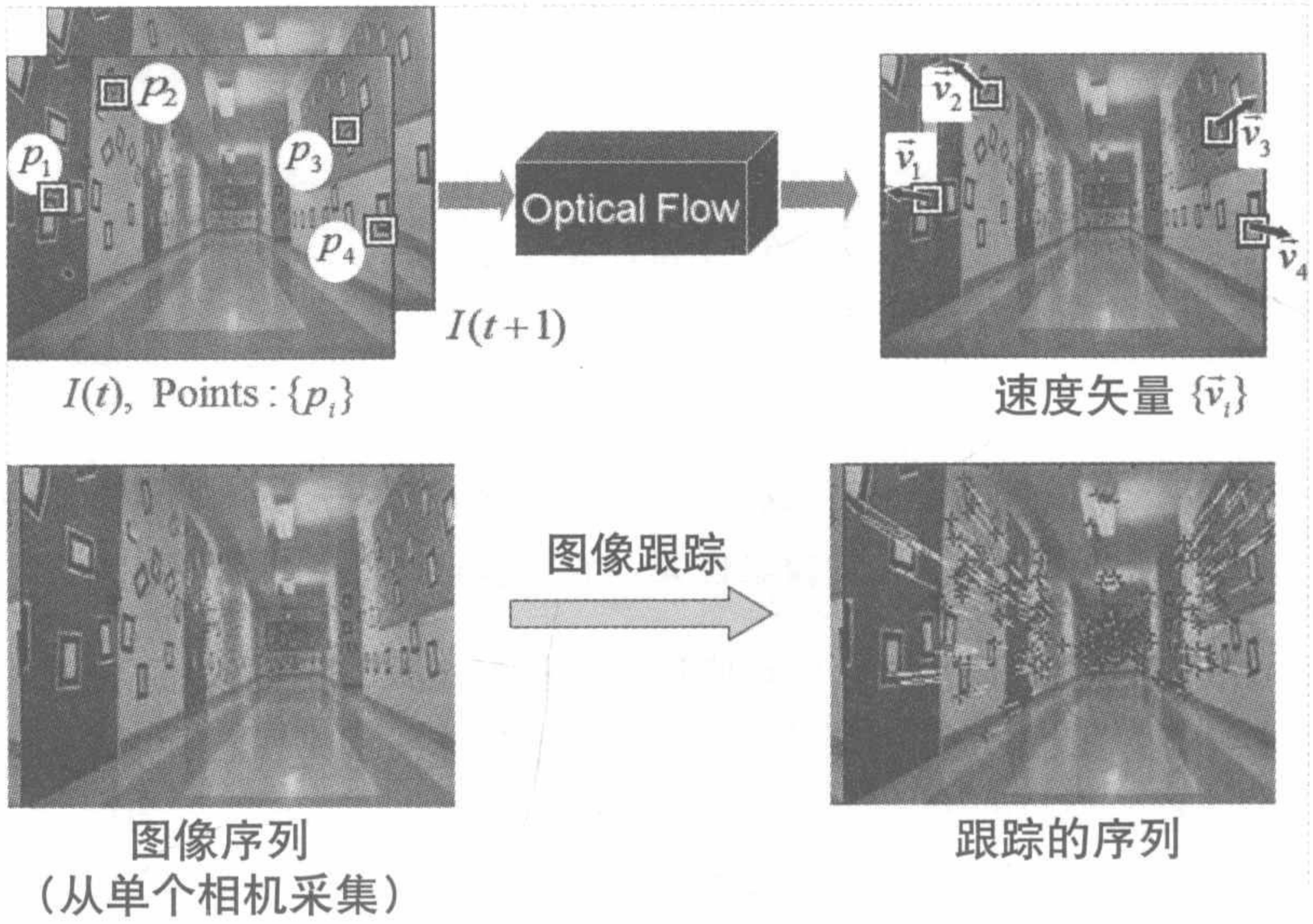


图16-3. 光流：目标特征（左）随时间跟踪，其运动被转换为速度矢量（右）（原图由Jean-Yves Bouguet提供）

光流算法的理想输出是两帧图像中每个像素的速度的估计关联，或者等效的，一幅图像中的每个像素的位移矢量，指示该像素在另一幅图像中的相对位置。图像中的每个像素都使用这种方法，则通常将其称为“稠密光流”。有一种替代类算法被称为“稀疏光流算法”，仅仅跟踪图像中某些点的子集。这些算法通常是快速和可靠的，因为它们将注意力限制在图像中更容易跟踪的特定点上。OpenCV有许多方法可以帮助我们识别非常适合跟踪的点，前面提到的角点只是列表中的一个。对于许多实际应用，稀疏跟踪的计算成本远远低于稠密跟踪，这导致后者的研究基本被限制在学术圈[注4]。在本节中，我们将介绍一种稀疏光流技术。之后，我们将看到更强大的稀疏光流工具，最后再开始介绍稠密光流。

Lucas-Kanade稀疏光流法

1981年第一次提出的Lucas-Kanade（LK）算法[Lucas81]试图计算稠密光流（即每个像素的光流）。然而，由于该方法容易应用到输入图像中的点的子集，所以反而成为了稀

注4：Black和Anandan创造了稠密光流技术[Black93; Black96]。该技术常用于电影制作，为了保证视觉质量，电影制片厂愿意花时间获取详细的光流信息。这些技术预计将包含在OpenCV的更高版本中（参见第23章）。

疏光流的重要技术。该算法可以应用在稀疏场景中，因为它仅依赖于围绕每个兴趣点的一些小窗口导出的局部信息。在Lucas-Kanade算法中使用小的局部窗口的缺点是运动幅度过大的话，像素点会运动到局部窗口之外，不能被算法找到。这个问题导致了“金字塔”LK算法的发展，该算法从图像金字塔（最低的细节）的最高级别开始跟踪，并逐渐跟踪至更低级别（更精细的细节）。跟踪图像金字塔允许大幅度运动被局部窗口捕获。[注5]

因为这是一个重要且有效的技术，我们需要了解一些数学细节，不想了解这些细节的读者可以跳到函数描述和代码。但是，建议你至少浏览一下Lucas-Kanade光流背后的假设的中间文字和数据，以便可以直观了解在跟踪不正常时该怎么做。

Lucas-Kanade算法的工作原理

LK算法的基本思想基于以下三个假设。

亮度恒定

场景中目标的图像像素看起来（可能）在帧到帧移动时不发生改变。对于灰度图像（LK算法也可以对彩色图像进行计算），这意味着我们假设像素的灰度不会随着帧的跟踪改变。

时间持续性或“微小移动”

图像上相斑的运动随时间变化缓慢。实际上，这意味着时间相对于图像运动的尺度来说足够快，从而目标两帧之间的运动幅度也不会太大。

空间一致性

场景中属于相同表面的相邻点，具有相似的运动，并且其投影到图像平面上的点距离也较近。

我们现在来看看这些假设（如图16-4所示，怎样推导出有效的跟踪算法。第一个假设亮度恒定，要求一个跟踪的相斑中的像素随着时间的推移看起来相同，定义如下：

$$f(x, t) \equiv I(x(t), t) = I(x(t + dt), t + dt)$$

注5：OpenCV中，金字塔框架下的Lucas-Kanade光流的确定性描述来自Bouguet [Bouguet04]未发表的论文。

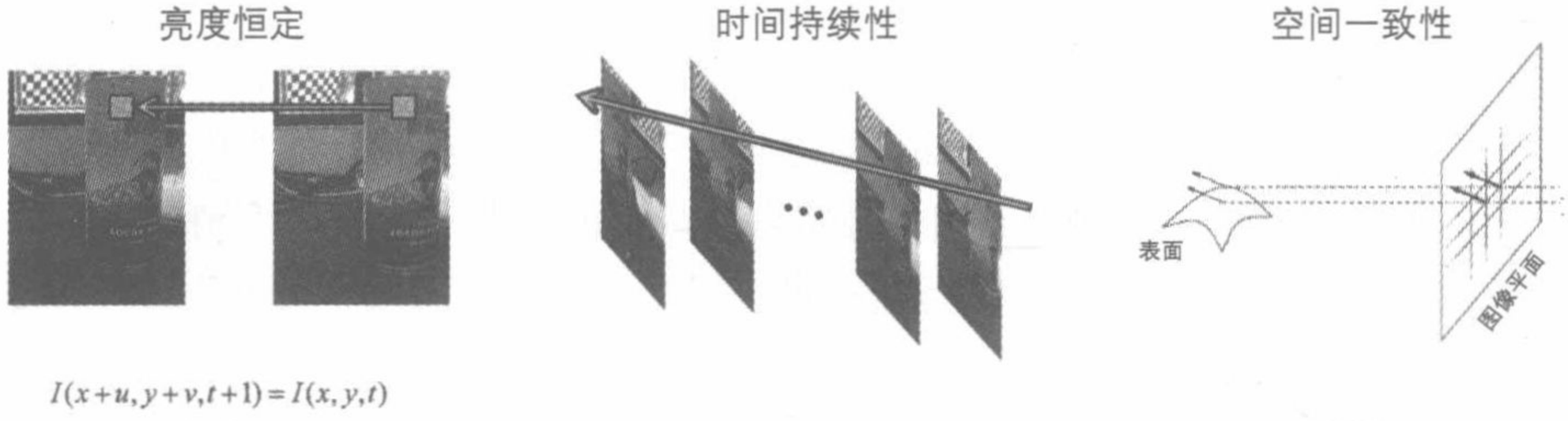


图16-4. Lucas-Kanade光流背后的假设：对于被追踪的场景目标上的像斑，其亮度不会改变（A）；运动相对于帧速率来说较慢（B）；相邻点仍旧保持相邻（C）；组件图像由Michael Black提供[Black92]

我们跟踪的像素强度随时间变化的假设可以简单表示如下：

$$\frac{\partial f(x)}{\partial t} = 0$$

第二个假设时间持续性，意味着目标在帧间的运动幅度非常小。换句话说，我们可以将该变化近似看做相对于时间相对于时间的导数（即我们认为图像序列中的一帧与下一帧之间的发生的变化差异很小）。要理解这个假设的含义，首先要考虑一个一维空间的情况。

在这种情况下，我们可以从亮度一致性方程开始，替代亮度$f(x, t)$的定义，同时考虑x对t，$I(x(t)t)$的隐含依赖性，然后应用求偏导的链式法则。即：

$$I_x \cdot v + I_t = \frac{\partial I}{\partial x}\Big|_t \left(\frac{\partial x}{\partial t}\right) + \frac{\partial I}{\partial t}\Big|_{x(t)} = 0$$

其中I_x是第一幅图像的空间导数，它是图像随时间的导数，v是我们正在寻找的速度。这样，我们就得到了光流速度在简单的一维情况下的方程：

$$v = -\frac{I_t}{I_x}$$

现在我们试着对这个一维跟踪问题进行一些比较直观的探索。考虑图16-5，它显示了一个“边缘”——左边高，右边低——正沿着x轴向右移动。我们的目标是确定边缘移动的速度v，如图16-5的第一幅图所示。在第二幅图中，我们可以看到，我们对这种速度的测量结果是“移动上升”的，其中“上升”是随时间进行的，“移动”是斜率（空间导数）。负号用于校正x的斜率。

图16-5揭示了我们所用的光流公式的另一面：我们的假设可能不完全正确。也就是说，图像亮度不是很稳定；我们的时间步长（由相机决定）通常也不如我们想要的那样快。因此，我们的速度求解方案并不精确。但是，如果我们已经足够接近了，就可以使用

迭代求解。迭代显示在图16-6中，我们使用我们的初始（不准确）速度估计作为下一次迭代的初值，然后重复。注意，由于假设亮度恒定，我们可以保持x方向上的空间导数与第一帧计算相同——在x方向上移动的像素不发生改变。空间导数计算结果的复用可以显著节省计算量。时间导数的求解仍然需要重新计算每次迭代和每一帧，但是如果我们在开始的时候就足够接近，那么这些迭代就可以在大约五次迭代中收敛到近似的精确度。这称为“牛顿法”。但如果我们的初始估计不够接近，那么牛顿法实际上就会产生偏离。

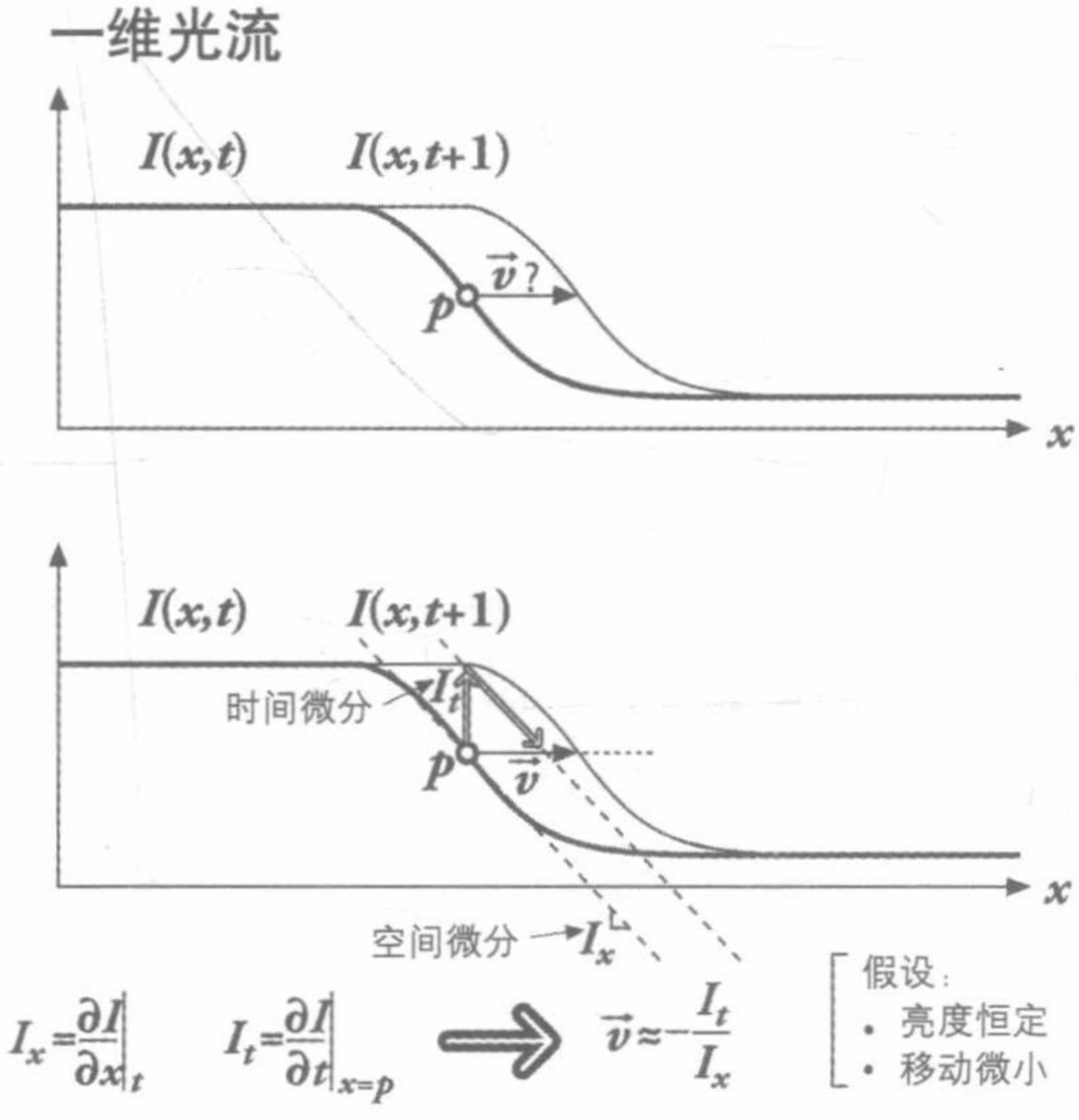


图16-5. 一维Lucas-Kanade光流：我们可以通过测量强度随时间的导数与由空间强度导数的比值来估计移动边缘的速度

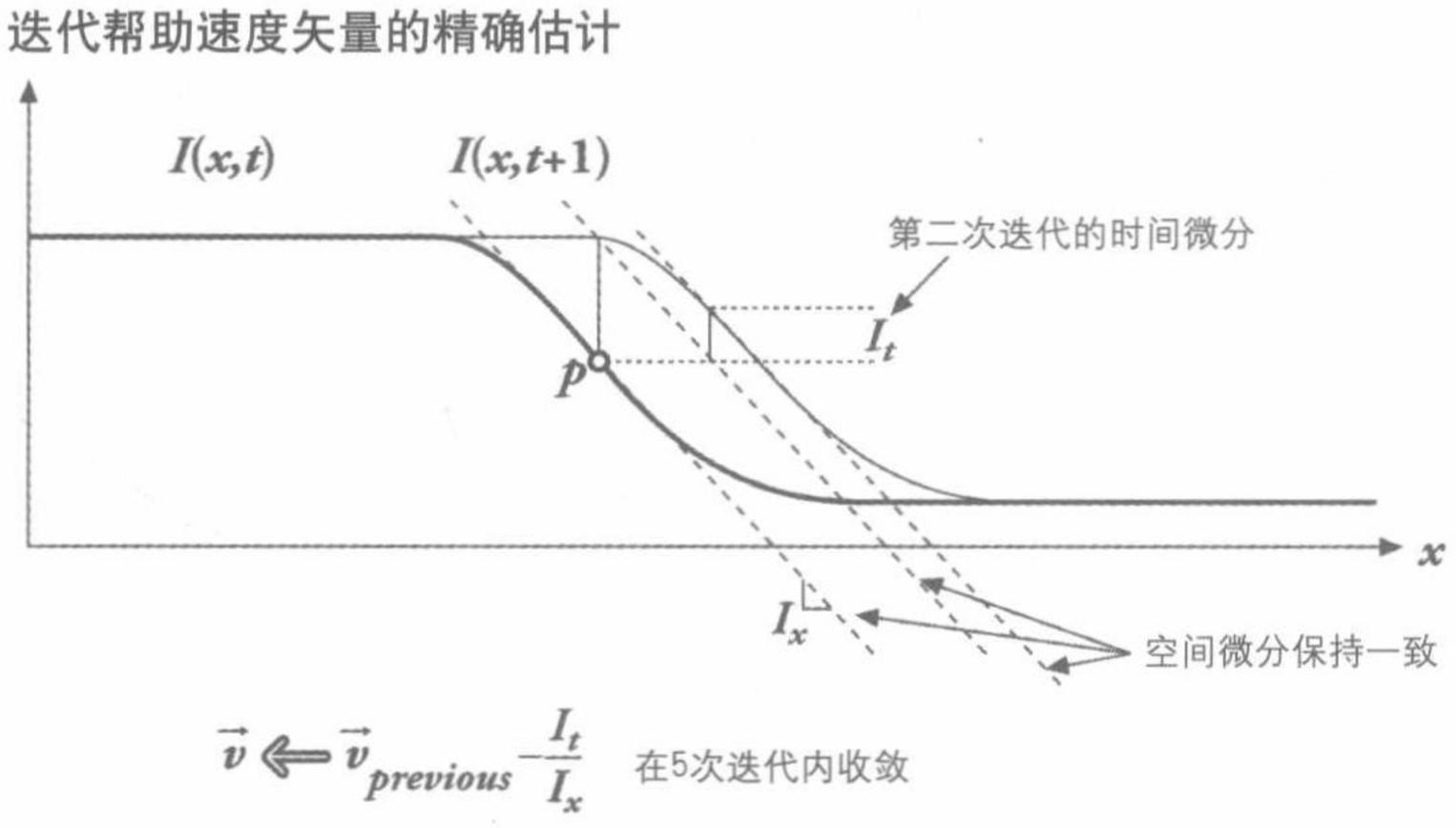


图16-6. 迭代细化光流解（牛顿法）：我们使用相同的两幅图像和相同的空间导数（斜率），再次求解时间导数。通常，几次迭代就会收敛到稳定解

现在我们已经看到了一维解决方案，接下来让我们把它推广到二维图像。乍一看，这似乎很简单：只需添加y坐标。稍微改变一下符号，我们将其称为速度v的y分量和速度u的x分量，于是有：

$$I_x u + I_y v + I_t = 0$$

这通常写成单向量方程：

$$\vec{\nabla} I \cdot \vec{u} = -I_t$$

其中：

$$\vec{\nabla} I = \begin{bmatrix} I_x \\ I_y \end{bmatrix}, and \vec{u} = \begin{bmatrix} u \\ v \end{bmatrix}$$

不幸的是，对于这个方程，任何给定像素都有两个未知数。这意味着在单像素级别的测量不受约束，并且不能获得该点二维运动的唯一解。相反，我们只能解决与我们的流量方程描述的线垂直或“法线方向”的运动分量。图16-7说明了几何原理。

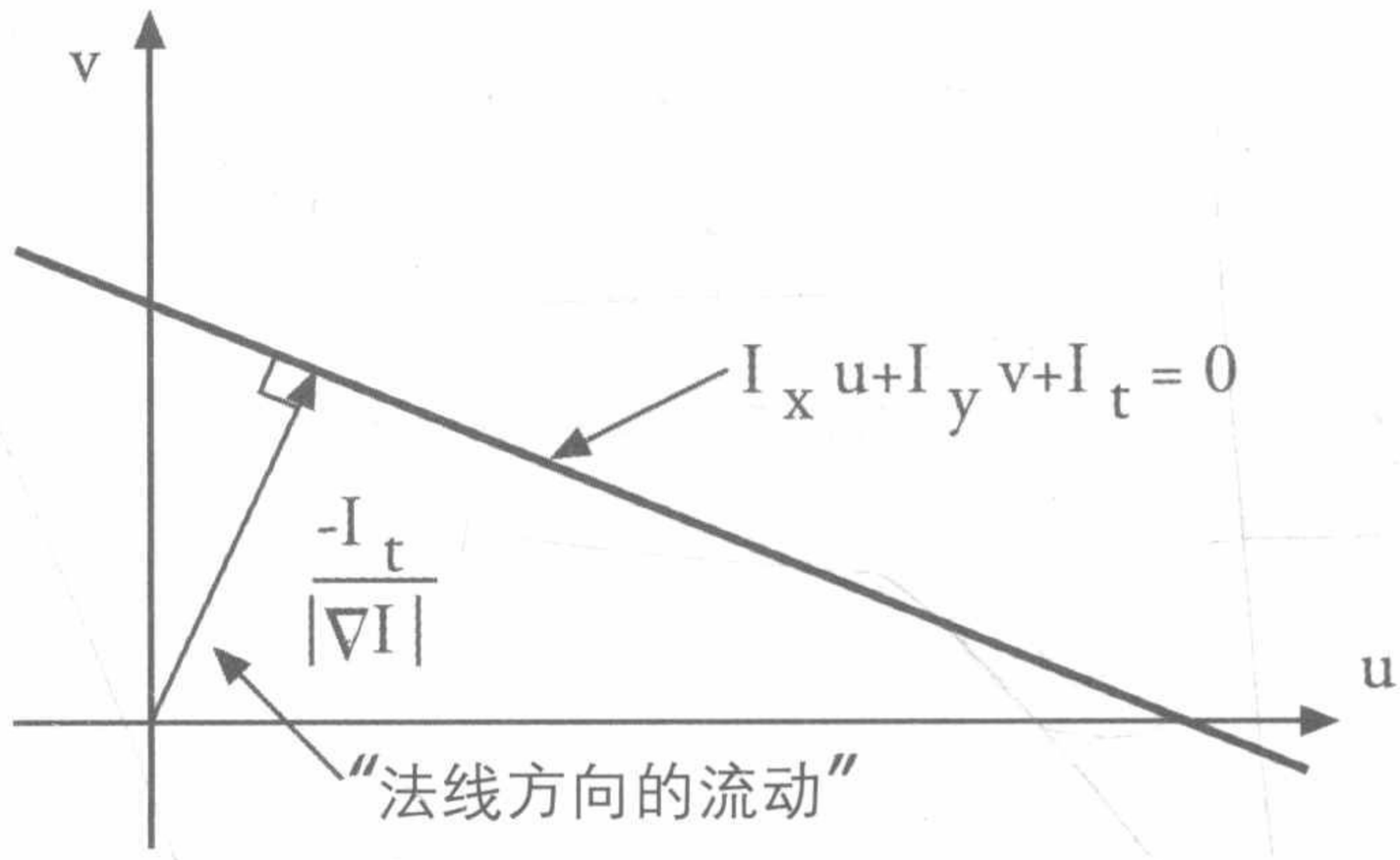


图16-7. 在一个像素处的二维光流：一个像素处的光流不足，因此最多只能产生与流动方程式描述的线垂直（“法线方向”）的运动（图片由Michael Black提供）

正常的光流结果来自孔径问题，这在通过一个小孔径或窗口来测量运动时就会出现。当用小孔径检测到运动时，经常都只能看到只有一个边缘，而不是一个角。但是，单独的边缘不足以确定整个对象正在移动的方式（即移动方向），详情请参见图16-8。

那么我们怎么解决这个问题呢，在一个像素上，无法解决全部的动作吗？ 我们转向最后一个光流假设来寻求帮助。如果局部像素斑点相干移动，那么我们可以通过使用周围像

素来轻松地求解中心像素的运动来建立方程组。例如，如果我们使用5×5窗口[注6]。在当前像素周围的亮度值窗口（如果是基于颜色的光流，可以简单将其重复三次）计算其运动，然后可以设置25个方程如下：

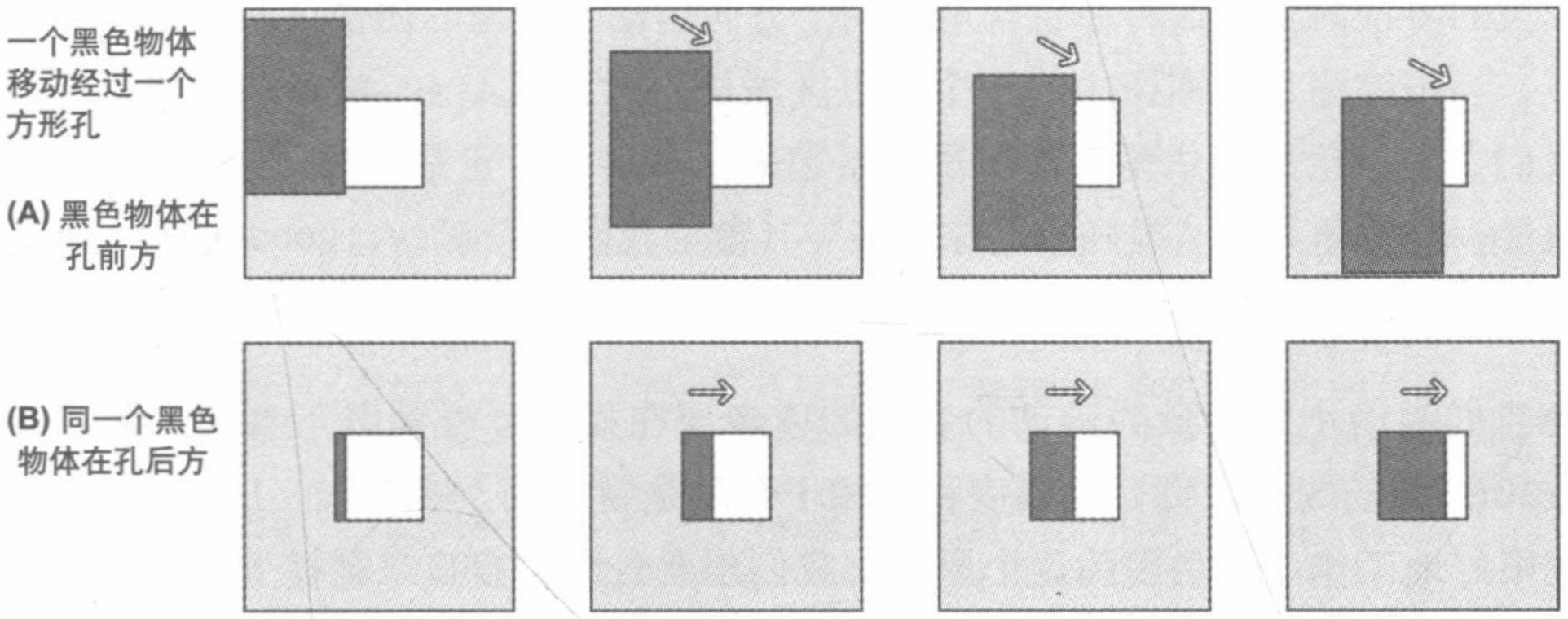


图16-8. 孔径问题：(A)物体正在向右移动；(B)通过一个小孔径，我们看到一个边缘向右移动但不能检测到运动的向下部分

$$A_{25\times 2}\cdot d_{2\times 1}=\begin{bmatrix} I_x(p_1) & I_y(p_1)\\ I_x(p_2) & I_y(p_2)\\ \vdots & \vdots\\ I_x(p_{25}) & I_y(p_{25})\end{bmatrix}\begin{bmatrix}u\\ v\end{bmatrix}=-\begin{bmatrix} I_t(p_1)\\ I_t(p_2)\\ \vdots\\ I_t(p_{25})\end{bmatrix}=b_{25\times 1}$$

我们现在有一个过度约束的系统，我们可以求解这个系统，只要它不仅仅是一个5×5窗口的边缘。为了求解这个系统，我们用最小二乘法对方程式进行最小化，其中$\min\| Ad-b\|^2$的解的标准形式为：

$$(A^TA)_{2\times 2}\cdot d_{2\times 1}=(A^Tb)_{2\times 2}$$

从这个关系，我们得到*u*和*v*运动分量。更详细的形式如下：

$$\left(A^TA\right)\cdot d=\begin{bmatrix}\sum I_xI_x & \sum I_xI_y\\ \sum I_yI_x & \sum I_yI_y\end{bmatrix}\begin{bmatrix}u\\ v\end{bmatrix}=-\begin{bmatrix}\sum I_xI_t\\ \sum I_yI_t\end{bmatrix}=\left(A^Tb\right)$$

那么这个方程式的解就是：

注6：当然，窗口可以是3×3，7×7或任何希望选择的尺寸。如果窗口太大，那么你最终会违反相干运动的假设，并且不能很好地跟踪。如果窗口太小，会再次遭遇孔径问题。

$$\begin{bmatrix} u \\ v \end{bmatrix} = \left(A^t A\right)^{-1} A^T b$$

什么时候可解？当（$A^T A$）是可逆时。（$A^T A$）满秩（rank=2）时是可逆的，此时它具有两个大的特征向量。这将在包含至少两个方向运动的纹理的图像区域中发生。在这种情况下，当跟踪窗口在图像中的一个角点区域居中时，（$A^T A$）将具有最佳属性。这就能与我们之前讨论的哈尔角点检测器联系起来。事实上，正是因为（$A^T A$）有两个大的特征向量，这些角点是“很好的追踪特征”（参见我们以前对cv::goodFeaturesToTrack()的讨论）。我们将很快看到cv::calcOpticalFlowPyrLK()函数怎样完成这些计算。

了解我们假设小而一致的运动的影响的读者现在或许会受到以下事实的困扰：对于大多数30Hz运动的摄像机，大幅度和非相干运动是常见的。事实上，L-K光流本身本身并不能很好地工作，正是因为这个原因：我们想要一个大窗口来捕捉大幅度的运动，但是一个大的窗口经常打破一致的运动假设！为了规避这个问题，我们可以使用图像金字塔首先跟踪较大的空间尺度，然后通过沿着图像金字塔的水平工作来完善初始运动速度假设，直到我们跟踪到原始图像像素。

因此，推荐的技术还是首先解决顶层的光流，然后使用最终的运动估计作为下一层的起始点。我们继续以这种方式下溯金字塔，直到我们达到最低层。因此，我们最大限度地减少违反我们的运动假设，因此可以追踪更快更长的运动。这个更精细的函数称为“金字塔Lucas-Kanade光流”，如图16-9所示。金字塔Lucas-Kanade光流算法在OpenCV中的函数实现是cv::calcOpticalFlowPyrLK()，接下来我们会详细介绍。

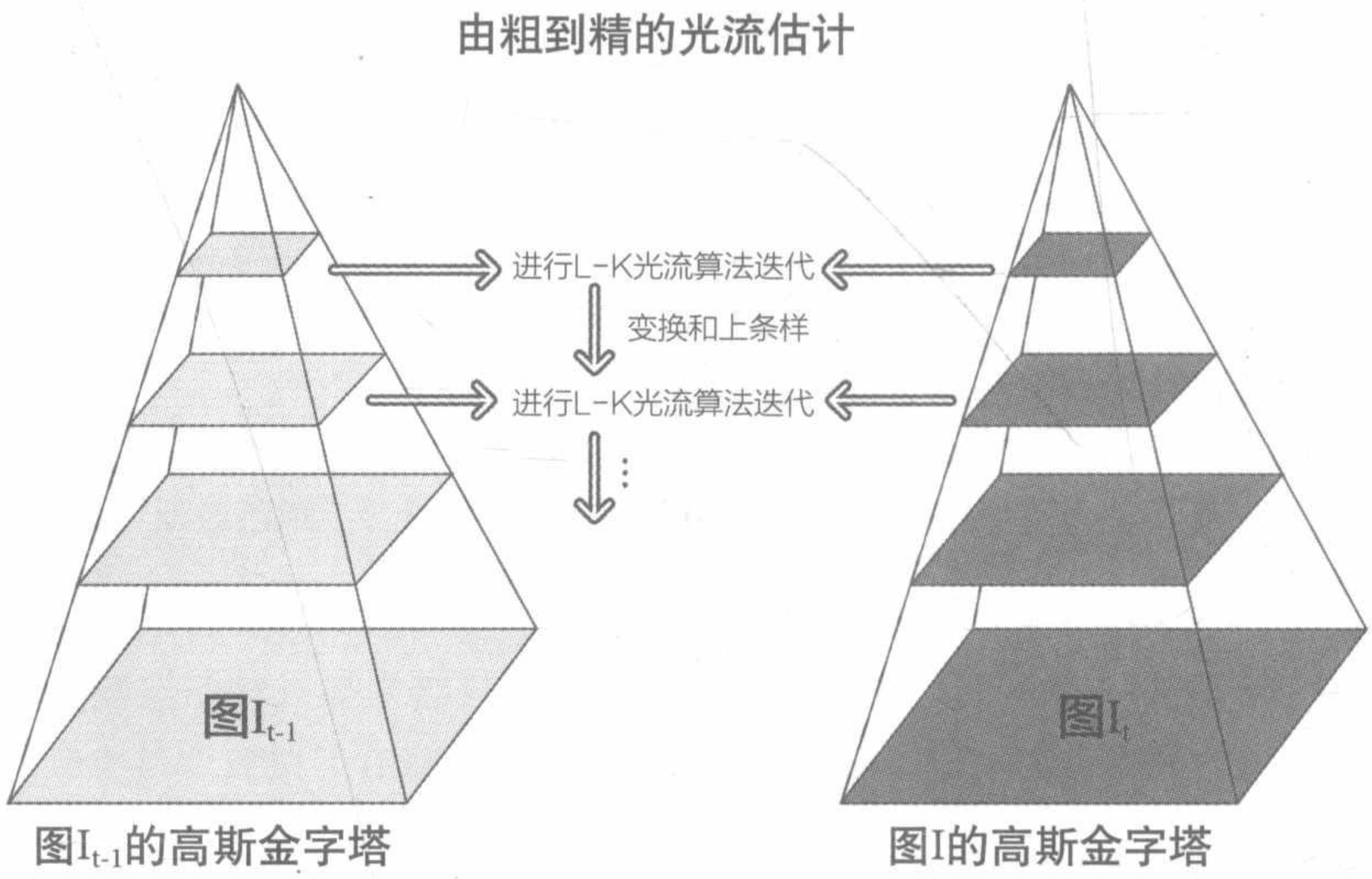


图16-9. 金字塔Lucas-Kanade光流：在金字塔顶部运行光流，可以减轻我们对小而连贯运动假设的违反问题；将来自前一层的运动估计值作为下一层估计运动的起始点

金字塔Lucas-Kanade代码：cv::calcOpticalFlowPyrLK()

我们现在来看OpenCV的算法，cv::calcOpticalFlowPyrLK()如何计算金字塔中的Lucas-Kanade光流。如我们将看到的，这种光流函数利用“良好的跟踪特征”，并且返回每个点的跟踪情况。

```
void cv::calcOpticalFlowPyrLK(
  cv::InputArray       prevImg,           // Prior image (t-1), CV_8UC1
  cv::InputArray       nextImg,           // Next image (t), CV_8UC1
  cv::InputArray       prevPts,           // Vector of 2d start points (CV_32F)
  cv::InputOutputArray nextPts,           // Results: 2d end points (CV_32F)
  cv::OutputArray      status,            // For each point, found=1, else=0
  cv::OutputArray      err,               // Error measure for found points
  cv::Size             winSize         = Size(15,15),   // size of search window
  int                  maxLevel        = 3,             // Pyramid layers to add
  cv::TermCriteria     criteria        = TermCriteria(  // How to end search
                         cv::TermCriteria::COUNT | cv::TermCriteria::EPS,
                         30,
                         0.01
                       ),
  int                  flags           = 0,    // use guesses, and/or eigenvalues
  double               minEigThreshold = 1e-4  // for spatial gradient matrix
);
```

这个函数有很多的输入，所以让我们花点时间来弄明白它所做的一切。一旦弄明白这个函数，我们就可以解决跟踪哪些点以及如何计算它们的问题。基本方案很简单：你可以提供图像，列出你要在prevPts中跟踪的点，并调用函数。当函数返回时，就可以检查状态数组以查看哪些点已成功跟踪，然后检查nextPts找到这些点的新位置。现在我们来详细说明一下。

cv::calcOpticalFlowPyrLK()的前两个参数，prevImg和nextImg是初始图像和最终图像。两者应该具有相同的大小并且具有相同数量的通道[注7]。接下来的两个参数prevPts和nextPts是来自第一幅图像的特征的输入列表以及第二幅图像中匹配点将被写入的输出列表。这些可以是$N \times 2$个数组或点的向量。数组status和err将填充匹配是否成功信息。特别是，status中的每个条目都会告知是否发现了prevPts中的相应特征（当且仅当在nextImg中找到prevPts[i]时，状态[i]将为非零。类似地，err[i]将指示在nextImg中发现的任何点prevPts[i]的错误度量,如果未找到i点，则err[i]不定义。

用于计算局部相干运动的窗口由winSize给出。因为我们正在构建图像金字塔，所以参数maxLevel用于设置图像堆栈的深度。如果maxLevel设置为0，则不使用金字塔。

注7：在旧版本的库中，只能使用单通道或三通道图像。新的实现（从v2.4开始）可以处理具有任意数量通道的图像。这使得可以使用纹理描述符或其他密集描述符来进行像素跟踪（只要描述符可以使用欧几里得范数进行比较）。

参数criteria用于告诉算法何时退出搜索匹配；回想一下，cv::TermCriteria是许多OpenCV算法都在使用的迭代求解的结构：

```
struct cv::TermCriteria(

public:
  enum {
    COUNT    = 1,
    MAX_ITER = COUNT,
    EPS      = 2
  };

  TermCriteria();
  TermCriteria( int _type, int_maxCount, double _epsilon );

  int    type,      // one of the enum types above
  int    max_iter,
  double epsilon
);
```

在大多数情况下，默认值都是令人满意的。然而，在通常情况下，如果图像非常大，你可能需要稍微增加允许的最大迭代次数。

参数flags可以具有以下值之一或两者。

CV::OPTFLOW_LK_GET_MIN_EIGENVALS

设置此标志来获得更详细的错误度量。error输出的默认错误度量是上一个角点周围窗口和新角点周围窗口之间平均每个像素强度的变化。将该标志设置为true时，该错误将替换为与角点相关联的哈尔矩阵的最小特征值。[注8]

CV::OPTFLOW_USE_INITIAL_FLOW

当数组nextPts在调用函数时已经包含特征坐标的初始估计时使用此参数。（如果没有设置这个标志，那么初始坐标的估计将只是在prevPts中的点位置。）

最后一个参数minEigThreshold被用作滤波器，用于去除点。事实上，这些点完全不适合跟踪。实际上，除了其确切的计算方法不同，它有点类似于cv::goodFeaturesToTrack的qualityLevel参数。默认值为10~4是一个不错的选择，也可以增大该参数以扔掉更多的点。

一个很好的例子

综上所述，我们现在知道了如何找到好的特征来进行跟踪，并且知道如何跟踪它们。我们可以通过使用cv::goodFeaturesToTrack()和cv::calcOpticalFlowPyrLK()的组合来获得良好的结果。当然，你也可以使用自己的标准来确定要跟踪的点。

注8：　当我们在本章稍后讨论哈尔算法的细节时，该标志的含义将变得更加明确。

现在我们来看一个使用cv::goodFeaturesToTrack()和cv::calcOpticalFlowPyrLK()的简单例子（示例16-1），另见图16-10。

示例16-1：金字塔Lucas-Kanade光流代码

```
// Pyramid L-K optical flow example
//
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

static const int MAX_CORNERS = 1000;

void help( argv ) {
    cout << "Call: " <<argv[0] <<" [image1] [image2]" << endl;
    cout << "Demonstrates Pyramid Lucas-Kanade optical flow." << endl;
}

int main(int argc, char** argv) {

  if( argc != 3 ) { help( argv ); exit( -1 ); }

  // Initialize, load two images from the file system, and
  // allocate the images and other structures we will need for
  // results.
  //
  cv::Mat imgA = cv::imread(argv[1], cv::LOAD_IMAGE_GRAYSCALE);
  cv::Mat imgB = cv::imread(argv[2], cv::LOAD_IMAGE_GRAYSCALE);
  cv::Size img_sz = imgA.size();
  int win_size = 10;
  cv::Mat imgC = cv::imread argv[2], cv::LOAD_IMAGE_UNCHANGED);

  // The first thing we need to do is get the features
  // we want to track.
  //
  vector< cv::Point2f > cornersA, cornersB;
  const int MAX_CORNERS = 500;
  cv::goodFeaturesToTrack(
    imgA,                           // Image to track
    cornersA,                       // Vector of detected corners (output)
    MAX_CORNERS,                    // Keep up to this many corners
    0.01,                           // Quality level (percent of maximum)
    5,                              // Min distance between corners
    cv::noArray(),                  // Mask
    3,                              // Block size
    false,                          // true: Harris, false: Shi-Tomasi
    0.04                            // method specific parameter
  );
  cv::cornerSubPix(
    imgA,                           // Input image
    cornersA,                       // Vector of corners (input and output)
    cv::Size(win_size, win_size),   // Half side length of search window
    cv::Size(-1,-1),                // Half side length of dead zone (-1=none)
    cv::TermCriteria(
```

```
    cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS,
    20,                                 // Maximum number of iterations
    0.03                                // Minimum change per iteration
  )
);

// Call the Lucas Kanade algorithm
//
vector<uchar> features_found;
cv::calcOpticalFlowPyrLK(
  imgA,                                 // Previous image
  imgB,                                 // Next image
  cornersA,                             // Previous set of corners (from imgA)
  cornersB,                             // Next set of corners (from imgB)
  features_found,                       // Output vector, elements are 1 for tracked
  noArray(),                            // Output vector, lists errors (optional)
  cv::Size( win_size*2+1, win_size*2+1 ), // Search window size
  5,                                    // Maximum pyramid level to construct
  cv::TermCriteria(
    cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS,
    20,                                 // Maximum number of iterations
    0.3                                 // Minimum change per iteration
  )
);

// Now make some image of what we are looking at:
// Note that if you want to track cornersB further, i.e.
// pass them as input to the next calcOpticalFlowPyrLK,
// you would need to "compress" the vector, i.e., exclude points for which
// features_found[i] == false.
for( int i = 0; i < (int)cornersA.size(); i++ ) {
  if( !features_found[i] )
    continue;
  line(imgC, cornersA[i], cornersB[i], Scalar(0,255,0), 2, cv::AA);
}
cv::imshow( "ImageA", imgA );
cv::imshow( "ImageB", imgB );
cv::imshow( "LK Optical Flow Example", imgC );
cv::waitKey(0);

return 0;
}
```

广义关键点和描述符

跟踪，目标检测和一些相关主题中最需要了解的两个基本概念是关键点和描述符。我们的第一个任务就是了解这两个概念以及它们之间的差异。

在最高抽象层次上，关键点是图像的一小部分，由于某种原因，它们非常独特，以至于让我们相信我们或许可以在另一个相关图像中定位它们。描述符是一些数学结构，通

常是（但不总是）浮点型的向量，它以某种方式描述了单个关键点，并且可以用于确定——在某些场景中——两个关键点是否“相同”。[注9]

在历史上，第一个重要的关键点类型就是哈尔角点，我们在本章开头曾经遇到过。回想一下，哈尔角点背后的基本概念是，在两个不同的坐标轴上，似乎强度变化强烈的图像中的任何一点像素都是另一个相关图像上对应点的匹配，例如，视频流连续帧中的图像。

看看图16-10中的图片；图像是一本书中的文本。[注10]你可以看到哈尔角点是在组成单个文本字符的行开始或结束的地方找到的，或者有交点的行（例如*h*或*b*的中间），你会注意到“角点”不会在字符中的长行边缘出现，只能在结尾处出现，这是因为在这样的边缘上发现的一个特征非常像在该边缘任何地方发现的任何其他特征，这是因为如果某些东西在当前图像中不是独一无二的，它在另一个图像中就不会是唯一的，所以这样的像素不符合特性。

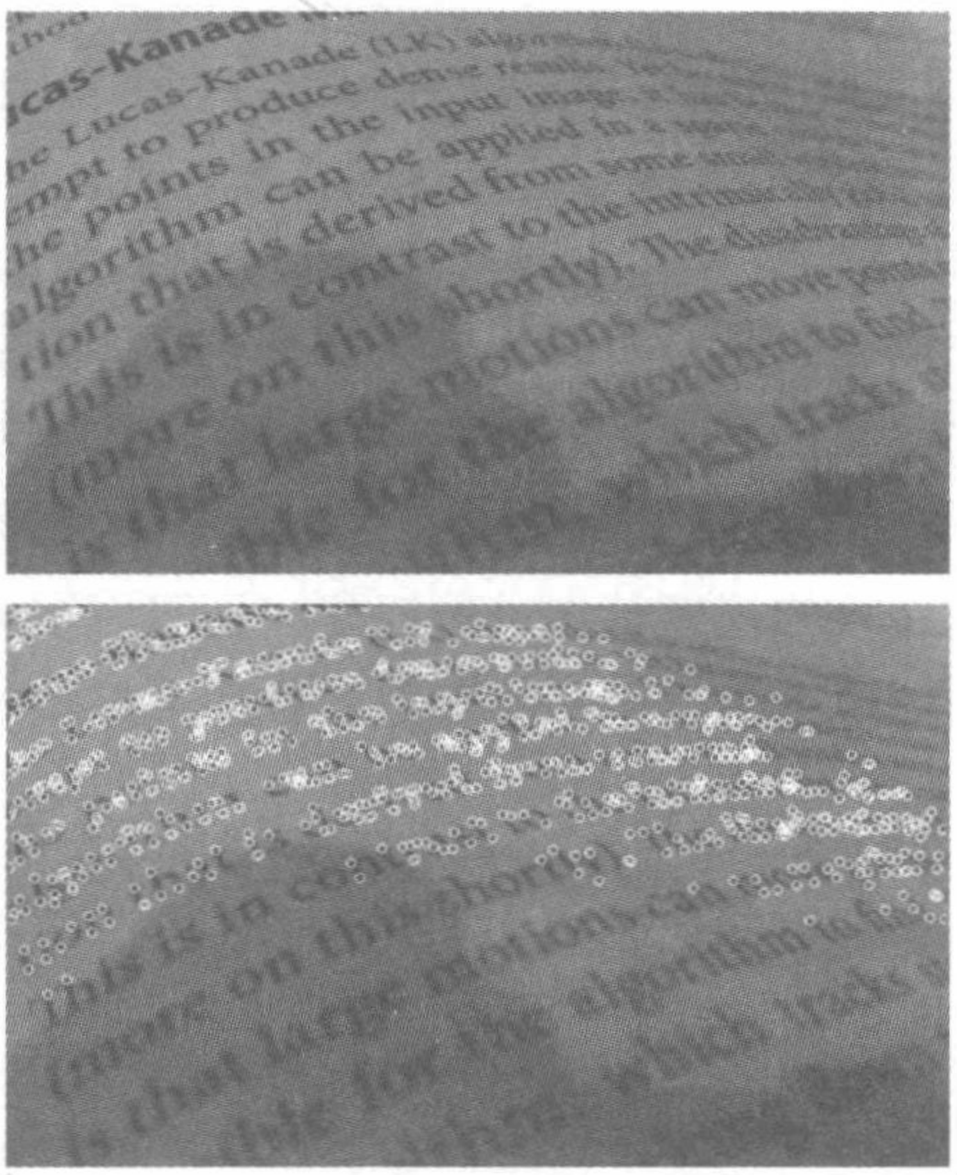


图16-10. 包含文字的两个图像，哈尔角点在右边显示为白色圆圈。请注意，在图像略有偏离焦点的部分，根本没有检测到任何角点

注9：在下一章中，我们将广泛地讨论跟踪（从一帧图像到另一帧图像的跟踪）以及关于目标识别（在图像中找到一个在先验数据库中有一些信息的东西）。第三个非常重要的情况，我们不会在这些章节中谈论很多，是稀疏的立体视觉。稀疏立体视觉的一个关注点是要在不同视角同时拍摄的两幅或多幅图像中定位关键点。直到我们在第19章中讨论立体视觉，唯一需要记住的重要事实是，从函数库的角度来看，稀疏立体视觉将使用与跟踪相同的方法和接口，也就是说，你需要输入两幅图像，并想要找到每个图像中的关键点之间的对应（匹配）。

注10：事实上，这是你正在阅读的这本书的第一版的文本。

对于我们人类的眼睛，每个特征看起来将会和其他任何特征都有所不同。特征描述符的问题解决了如何使计算机进行这种关联的问题。我们说过，出现在*h*或*b*中的“三路交叉路口”会产生良好的“特征”，但是电脑如何区分它们之间的区别呢？这就是特征描述符所做的事情。

我们可以用任何方式构造一个特征描述符——例如，我们可以从关键点周围的3 × 3区域的强度值中创建一个向量。这种处理的问题是如果从稍微不同的角度观察关键点，描述符就会具有不同的值。通常，旋转不变性[注11]是特征描述符的期望属性。当然，是否需要具有旋转不变性的描述符取决于你的应用程序。在检测和追踪人的时候，重力在创造一个不对称的世界中发挥了很大的作用，人们的头脑通常处于顶端，脚下通常处于底部。在这种应用中，缺乏旋转对称性的描述符或许不会有什么问题。相比之下，当飞机沿不同的方向行进时，地面的空中图像会“旋转”，因此图像可能看起来似乎是随机取向的位置。

光流，跟踪和识别

在上一节中，我们在讨论了Lucas Kanade算法在光流中的应用。在这种情况下，OpenCV为我们提供了一个单一的高级工具，可以获取关键点列表，并尝试找到下一帧图像中这些点的最佳匹配。这些关键点没有很多的结构或辨识度，仅仅够它们从两个非常相似的帧中的一帧到下一帧完成定位。但广义关键点可以具有非常强大的关联描述符，并且可以让这些点不仅能够在连续的视频帧中进行匹配，还可以在完全不同的图像中进行匹配。这可以让我们在新的环境中定位已知的对象，或通过复杂的变化场景来跟踪对象，以及完成许多其他应用。

关键点（及其描述符）主要用于三个任务，分别是跟踪、目标识别和立体视觉。

跟踪是随着场景在顺序图像流中演变而跟踪某个目标的运动的任务。跟踪出现在两个主要的子类别中：第一个是在静态场景中跟踪对象，另一个是跟踪场景本身，以估计摄像机的运动。通常，术语跟踪是指前者，后者称为“视觉测距”。当然，想同时做这两件事情也很常见。

第二个任务类别是目标识别。在这种情况下，程序观察场景并尝试识别一个或多个已知目标的存在。这里的想法是将某些关键点描述符与每个目标相关联，其原理是，如果发现了足够多的与特定目标相关联的关键点，就可以合理地推断出该目标存在于场景中。

注11：旋转不变性的确切含义实际上是旋转不变的特征。具体地说，这意味着当对下面的图像应用旋转操作时，特征描述符不变。当你选择或设计特征描述符时，这种不连续性（也称为对称性）至关重要。理想情况下，这不仅包括图像平面中的旋转，还包括（至少某种程度上）从成像器的角度来看，在目标的三维旋转中在图像平面中显示的仿射变换。

最后还有立体视觉。在这种情况下，我们有兴趣在相同场景或目标的两个或多个摄像机视图中定位相应的点。将这些位置与相机位置信息和摄像机本身的光学属性相结合，我们可以在三维中计算匹配出的各个点的位置。

OpenCV提供了处理多种类型的关键点和多种关键点描述符的方法。它还提供了在帧和帧之间（以稀疏光流、跟踪和视觉测距或立体视觉的方式）或帧与图像数据库（用于目标识别）之间进行匹配的方法。

OpenCV一般如何处理关键点和描述符

当进行跟踪时，或者其他类型的用到关键点及其描述符的分析时，通常需要做三件事情。第一个是根据一些关键点的定义搜索图像并查找该图像中的所有关键点。第二个是为发现的每个关键字创建一个描述符。第三个是通过将所找到的关键点的描述符与一些现有的描述符集进行比较，看看是否可以找到匹配项。在跟踪应用程序中，最后一步涉及查找序列的一帧图像中的特征，并尝试将其与前一帧中的特征进行匹配。在目标检测应用程序中，人们通常会在一些（潜在的广泛）数据库中搜索“已知”特征的“已知”与各个目标或目标类相关联的特征。

对于这些层中的每一个，OpenCV提供了一个遵循“做东西的类”（函子）模型的泛化机制。因此，对于这些阶段中的每一个，都有一个抽象基类，它为从其派生的一系列对象定义一个公共接口；每个派生类都实现一个特定的算法。

cv::KeyPoint对象

当然，如果要找到关键点，我们将需要一些表示它们的方法。回想一下，关键点不是特征描述符，所以当我们有一个关键点时，大多数时候都需要知道它所在的位置。之后，有些关键点会有次要特征，有些则没有。这是cv::KeyPoint类的实际定义：

```
class cv::KeyPoint {

public:

  cv::Point2f pt;       // coordinates of the keypoint
  float       size;     // diameter of the meaningful keypoint neighborhood
  float       angle;    // computed orientation of the keypoint (-1 if none)
  float       response; // response for which the keypoints was selected
  int          octave;  // octave (pyramid layer) keypoint was extracted from
  int          class_id; // object id, can be used to cluster keypoints by object

cv::KeyPoint(
  cv::Point2f _pt,
  float       _size,
  float       _angle=-1,
  float       _response=0,
```

```
  int         _octave=0,
  int         _class_id=-1
);
cv::KeyPoint(
  float       x,
  float       y,
  float       _size,
  float       _angle=-1,
  float       _response=0,
  int         _octave=0,
  int         _class_id=-1
);
...
};
```

你可以看到，每个关键点都有一个cv::Point2f成员，这告诉我们它位于哪里。[注12]size的概念告诉我们关于关键点周围区域的某些信息，或者在某种程度上包含关键点存在于第一位的判定，或者将在该关键点的描述符中发挥作用。关键点的angle只对某些关键点有意义。许多关键点能实现旋转对称，但实际上并不是在最严格的意义上不变，而是在比较两个描述符时可以考虑到的某种自然方向。这不是一个复杂的想法，如果你正在看铅笔的两幅图像，旋转显然很重要，但是如果你想比较它们，可以在进行比较之前，将它们旋转到同样的方向显示出来。

response用于能够对一个关键点比另一个关键点更强做出响应的检测器。在某些情况下，这可以被解释为该特征实际存在的概率。Octave用于在图像金字塔中寻找关键点的时候。在这些情况下，重要的是知道关键点可以在哪个尺度下被找到，因为在大多数情况下，我们都希望在新图像中以相同或相似的比例找到匹配点。最后，有一个类ID。在构造关键点数据库时，使用class_id来区分与一个对象相关联的关键点和与另一个对象相关联的关键点，下一小节中讨论关键点匹配接口时，我们将再次讨论这一点。

cv::KeyPoint对象有两个构造函数，它们基本相同，一的区别是使用两个浮点数还是单个cv::Point2f对象来设置关键点的位置。实际上，除非要写自己的关键点查找器，否则你不会倾向于使用这些函数。如果你正在使用OpenCV库中已有的检测，描述符构造和比较函数，通常甚至都不会查看关键点对象。

找到关键点的（抽象的）类且/或为它们计算描述符：cv::Feature2D

为了找到关键点或计算描述符（或同时执行这两个任务），OpenCV提供cv::Feature2D

注12：对于专家来说，这和它一开始出现的时候相比可能会有点微妙。大多数（如果不是全部）关键点实际上是一定数量的影响关键点的像素的扩展对象。然而，对于每个关键点类型，都存在根据该邻域定义的一个“中心”（或者更确切地说，邻域是根据该中心定义的）。在这个意义上，关键点的中心与滤波器或形态运算符的锚点相似。

类。有一些叫做cv::FeatureDetector和cv::DescriptorExtractor的类，它们是用于单纯的特征检测或描述符提取算法的单独的类，但是从OpenCV 3.x开始，它们都是cv::Feature2D的同义词。这个抽象类只有几种方法，然后描述。派生类添加了一些更多的方法来设置和检索各种属性，以及构造算法实例并返回智能指针的静态方法。实际上有两种方法需要cv::Feature2D类检测关键点，允许你从磁盘保存和恢复数据的函数，以及一个方便的（静态）函数，允许你通过提供名称来创建一个特征检测器派生类的检测器类型（作为字符串）。两种提供的检测方法（以及两种提供的“计算”方法）的区别只在于一种在单个图像上操作，而另一种在一组图像上操作。一次操作许多图像会有更高的效率（一些检测器比其他检测器更真实）。下面是cv::Feature2D的类描述中的相关代码：

```
class cv::Feature2D : public cv::Algorithm {

public:

  virtual void detect(
    cv::InputArray                    image,      // Image on which to detect
    vector< cv::KeyPoint >&           keypoints,  // Array of found keypoints
    cv::InputArray                    mask     = cv::noArray()
  ) const;

  virtual void detect(
    cv::InputArrayOfArrays            images,     // Images on which to detect
    vector<vector< cv::KeyPoint > >&  keypoints,  // keypoints for each image
    cv::InputArrayOfArrays            masks    = cv::noArray ()
  ) const;

  virtual void compute(
    cv::InputArray image,                  // Image where keypoints are located
    std::vector<cv::KeyPoint>& keypoints, // input/output vector of keypoints
    cv::OutputArray descriptors );         // computed descriptors, M x N matrix,
                                           // where M is the number of keypoints
                                           // and N is the descriptor size
  virtual void compute(
    cv::InputArrayOfArrays image,           // Images where keypoints are located
    std::vector<std::vector<cv::KeyPoint> >& keypoints, //I/O vec of keypnts
    cv::OutputArrayOfArrays descriptors ); // computed descriptors,
                                           // vector of (Mi x N) matrices, where
                                           // Mi is the number of keypoints in
                                           // the i-th image and N is the
                                           // descriptor size
  virtual void detectAndCompute(
    cv::InputArray image,                  // Image on which to detect
    cv::InputArray mask,                   // Optional region of interest mask
    std::vector<cv::KeyPoint>& keypoints, // found or provided keypoints
    cv::OutputArray descriptors,           // computed descriptors
    bool useProvidedKeypoints=false );     // if true,
                                           // the provided keypoints are used,
                                           // otherwise they are detected
```

```
    virtual int descriptorSize() const;     // size of each descriptor in elements
    virtual int descriptorType() const;     // type of descriptor elements
    virtual int defaultNorm() const;        // the recommended norm to be used
                                            // for comparing descriptors.
                                            // Usually, it's NORM_HAMMING for
                                            // binary descriptors and NORM_L2
                                            // for all others.

    virtual void read( const cv::FileNode& );
    virtual void write( cv::FileStorage& ) const;

    ...
};
```

如果它是一个单纯的关键点检测算法（如FAST），实际的实现可能只是执行cv::Feature2D::detect()；如果它是一个纯特征描述算法（如FREAK），实际的实现可能只是cv::Feature2D::compute()；在要求“完全解决”的算法的情况下，如SIFT，SURF，ORB，BRISK等，实际的实现就会是cv::Feature2D::detectAndCompute()，在这种情况下，detect()和compute()会被隐式调用：

- detect(image, keypoints, mask) ~ detectAndCompute(image, mask, keypoints, noArray(), false)
- compute(image, keypoints, descriptors) ~ detectAndCompute(image, noArray(), keypoints, descriptors, true)

cv::Feature2D::detect()方法直接或通过调用detectAndCompute()进行关键点的基本工作。第一个方法需要输入图像、关键点向量和可选掩码。然后搜索图像（或对应于掩码的部分，如果你提供的话）中的关键点，并将找到的任何内容放置在你提供的向量中。第二个方法做完全相同的事情，不同的是它合计一个图像的向量、一个掩码的向量（或根本没有）以及关键点向量的向量。图像的数量、掩码的数量（如果不是零）以及关键点向量的数量都必须相等。然后将搜索每个图像，并且关键点中的相应关键点向量将被填充找到的关键点。

查找关键点的实际方法当然对于cv::Feature2D的许多可用派生类中的每一个都是不同的。我们很快就会了解这些细节。同时，重要的是要记住，不同的检测器检测到的实际关键点可能不同（如果使用了一些内部参数，如何使用）。这意味着通过一种方法检测到的关键点可能不是对每种特征描述符都可用的。

一旦找到关键点，接下来要做的就是为它们计算描述符。如前所述，这些描述符可以对关键点进行相互比较（根据其外观而不是其位置）。这个功能可以作为跟踪或目标识别的基础。

描述符由compute()计算（也可以使用detecintt AndCompute()）。第一个compute()方法需要一个图像、一个关键点列表（可能由同一个类的detect()产生，但也可能不同）、cv::Mat（作为cv::OutputArray传递）格式的输出，这被compute()用于放置计算出的特征。由cv::Feature2D派生的对象生成的所有描述符都可以表示为固定长度的向量。因此，可以将它们全部排列成序列。一个使用的惯例是每行描述符都是单独的描述符，并且这些行的数量等于关键点中的元素的数量。

要一次处理几个图像时，将使用第二种compute()。在这种情况下，图像的参数需要一个其中包含你想要处理的所有图像的STL向量。关键点参数需要一个包含关键点向量的向量，描述符参数需要一个包含可用于存储所有生成的描述符的数组的向量。这种方法通常和多图像detect()同时出现，并且需要像给detect()方法那样输入的图像和从那种方法接收到的关键点。

根据算法何时实现，许多关键点特征检测器和提取器都设计在一起成为一个对象。在这种情况下，方法cv::Feature2D::detectAndCompute()被实现。每当某个算法提供detectAndCompute()时，强烈建议直接使用它，而不是在后续调用detect()，然后再使用compute()。原因很明显，就是性能更好：通常这种算法需要特殊的图像表示（称为“尺度空间表示”），并且计算可能相当昂贵。当你在两个单独的步骤中找到关键点和计算描述符时，基本上需要计算两次尺度空间表示。

除了主要的计算方法之外，还有方法descriptorSize()，descriptorType()和defaultNorm()。descriptorSize()告诉我们描述符的向量表示的长度。[注13] descriptorType()返回特定类型的向量描述符元素的信息（例如，32位浮点型的CV_32FC1，8位描述符的CV_8UC1或二进制描述符等）。[注14]defaultNorm()基本上告诉你如何比较描述符。在二进制描述符的情况下，你要使用的方法是NORM_HAMMING。若是SIFT或SURF这样的描述符，方法是NORM_L2（例如欧几里得距离），但是你可以通过使用NORM_L1获得同样好的或甚至更好的结果。

cv::DMatch对象

在深入了解匹配器工作原理之前，我们需要知道匹配是如何表达的。一般来说，匹配器将是一个尝试将一个图像中的关键点与其他单个图像或称为字典的其他图像的集合进行

注13：在二进制描述符的情况下（我们将在本章稍后讨论），该方法返回描述符中的字节数，而不是位数。通常，descriptorSize()返回cv::DescriptorExtractor::compute()将返回的描述符矩阵中的列数。

注14：在所有当前实现的情况下，描述符类型都是单通道。惯例是将描述符的任何类似通道的元素都“展平”成描述符的整体长度。结果，比如描述符可以是用于灰度图像的10维或彩色图像的30维，但永远不会是三个通道对象的10维阵列。

匹配的对象。当找到匹配时，OpenCV通过生成cv::DMatch对象的列表（STL向量）来描述它们。这是cv::DMatch对象的类定义：

```
class cv::DMatch {

public:

  DMatch();                  // sets this->distance
                             // to std::numeric_limits<float>::max()

  DMatch( int _queryIdx, int _trainIdx, float _distance );
  DMatch( int _queryIdx, int _trainIdx, int _imgIdx, float _distance );

  int   queryIdx;            // query descriptor index
  int   trainIdx;            // train descriptor index
  int   imgIdx;              // train image index
  float distance;

  bool operator<( const DMatch &m ) const; // Comparison operator
                                           // based on 'distance'

}
```

cv::DMatch的数据成员是queryIdx，trainIdx，imgIdx和distance。前两个辨别与每个图像中的关键点列表匹配的关键点。OpenCV库使用的惯例是将这两幅图像分别作为搜索图像（“新”图像）和训练图像（“旧”图像）。[注15]imgIdx用于辨别在图像和字典之间寻求匹配的情况下训练图像来自的特定图像。最后一个成员distance，用来表示匹配的质量。在许多情况下，这就像他们存在的多维向量空间中的两个关键点之间的欧氏距离。虽然这并不总是衡量标准，但是如果两个不同的匹配具有不同的距离值，则距离值越小，匹配越好。为了促进这种比较（特别是为了排序的目的），定义了运算符cv::DMatch::operator<()，这允许两个cv::DMatch对象直接在它们的距离成员的含义上进行实际比较。

（抽象）关键点匹配类：cv::DescriptorMatcher

检测描述匹配过程的第三阶段也通过一系列对象来实现，这些对象都是从它们共享接口的公共抽象基类派生的。第三层基于cv::DescriptorMatcher类。

在深入介绍接口如何工作之前，如本章前面所述，重要的是要了解有两种基本情况需要使用匹配器：目标识别和跟踪。在目标识别的情况下，首先将与各种对象相关联的关键点编译成称为“字典”的数据库。此后，当出现新场景时，提取该图像中的关键点并将

注15：这是一个有点尴尬的术语，并且有点混乱，词训练也用于指将处理目标识别情形中的所有图像的过程转换为随后可以查询的字典。我们将使用这个术语，因为它是OpenCV库中使用的，但是如果可能，我们总是会说“训练图像”来表明这个名词不同于动词列表。

其与字典进行比较，以估计新场景中可能存在的来自字典的对象。在跟踪情况下，目标是通常从视频流中查找某些图像中的所有关键点，然后查找另一图像中的所有关键点，通常是该视频流中的上一帧或下一帧图像。

因为有两种不同的情况，类中的匹配方法有两个相应的变化。对于目标识别的情况，我们需要首先用描述符的字典训练匹配器，然后才能向匹配器呈现单个描述符列表，并告诉我们匹配器存储的哪个（如果有的话）关键点是与我们提供的列表中的那些相匹配。对于跟踪的情形，我们需要提供两个描述符列表，然后匹配器告诉我们它们之间的匹配位置。cvDescriptorMatcher类接口提供了三个函数match()，knnMatch()和radiusMatch()；对于每个函数，有两个不同的变形——一个用于识别（需要一个特征列表以及经过训练的字典），另一个用于跟踪（需要两个特征列表）。

这是与通用描述符匹配器基类相关的类定义的一部分：

```
class cv::DescriptorMatcher {

public:

  virtual void add( InputArrayOfArrays descriptors );  // Add train descriptors
  virtual void clear();                                // Clear train descriptors
  virtual bool empty() const;                          // true if no descriptors
  void train();                                        // Train matcher
  virtual bool isMaskSupported() const = 0;            // true if supports masks
  const vector<cv::Mat>& getTrainDescriptors() const;  // Get train descriptors

  // methods to match descriptors from one list vs. "trained" set (recognition)
  //
  void match(
    InputArray                    queryDescriptors,
    vector<cv::DMatch>&           matches,
    InputArrayOfArrays            masks           = noArray ()
  );
  void knnMatch(
    InputArray                    queryDescriptors,
    vector< vector<cv::DMatch> >& matches,
    int                           k,
    InputArrayOfArrays            masks           = noArray (),
    bool                          compactResult   = false
  );
  void radiusMatch(
    InputArray                     queryDescriptors,
    vector< vector<cv::DMatch> >& matches,
    float                         maxDistance,
    InputArrayOfArrays             masks           = noArray (),
    bool                           compactResult   = false
  );

  // methods to match descriptors from two lists (tracking)
  //
```

```
// Find one best match for each query descriptor
void match(
  InputArray                    queryDescriptors,
  InputArray                    trainDescriptors,
  vector<cv::DMatch>&           matches,
  InputArray                    mask            = noArray ()
) const;
// Find k best matches for each query descriptor (in increasing
// order of distances)
void knnMatch(
  InputArray                    queryDescriptors,
  InputArray                    trainDescriptors,
  vector< vector<cv::DMatch> >& matches,
  int                           k,
  InputArray                    mask            = noArray(),
  bool                          compactResult   = false
) const;
// Find best matches for each query descriptor with distance less
// than maxDistance
void radiusMatch(
  InputArray                    queryDescriptors,
  InputArray                    trainDescriptors,
  vector< vector<cv::DMatch> >& matches,
  float                        maxDistance,
  InputArray                    mask            = noArray (),
  bool                          compactResult   = false
) const;

  virtual void read( const FileNode& );     // Reads matcher from a file node
  virtual void write( FileStorage& ) const; // Writes matcher to a file storage

  virtual cv::Ptr<cv::DescriptorMatcher> clone(
         bool emptyTrainData=false
  ) const = 0;
  static cv::Ptr<cv::DescriptorMatcher> create(
         const string& descriptorMatcherType
  );
...
};
```

第一组方法用于将图像与预先存储的描述集合匹配，每个图像一个阵列。目的是建立一个关键点字典，可以在提供新的关键点时被引用。第一种方法是add()，它需要一组描述符的STL向量，每一个都是cv::Mat格式的对象。每个cv::Mat对象应该有*N*行和*D*列，其中*N*是集合中描述符的数量，*D*是每个描述符的维度（即每个“行”是维度*D*的单独描述符）。add()接受阵列数组（通常表示为std::vector <cv::Mat>）的原因在于，在实际运用中，通常会从一组图像中的每个图像中计算一组描述符。[注16]这组图像可能被呈现给基于cv::Feature2D的类作为图像的向量，并作为关键点描述符的集合的向量返回。

注16：通常，这些图像包含单个对象，希望能够通过随后图像中出现的关键点来查找单个对象。因此，作为关键点参数的一部分提供的关键点应该将其class_id字段设置成对于要被识别的每个对象，关联关键点的class_id是不同的。

添加了一些关键点描述符集之后，如果想访问它们，可以使用（常量）方法getTrain-Descriptors()来返回描述符，就像你第一次提供它们的一样（一个描述符集合的向量，一个cv::mat类型的对象，每行都是一个描述符）。如果要清除添加的描述符，可以使用clear()方法，如果要测试匹配器中是否存储了描述符，可以使用empty()方法。

一旦加载了你想加载的所有关键点描述符，接下来可能需要调用train()。只有一些实现需要train()操作（即从cv::DescriptorMatcher派生的一些类）。该方法的目的是告诉匹配器你已经完成图像加载，并且可以继续处理预计算的任何内部信息，以便在提供的关键点上执行匹配。作为示例，如果匹配器仅使用提供的新关键点与现有字典中关键点之间的欧式距离来执行匹配，则构造四叉树或类似的数据结构来加速找到离提供的关键点最接近的字典关键点。这样的数据结构可能需要相当大的计算开销，并且在加载所有字典关键点之后仅计算一次。train()方法告诉匹配器花时间来计算这些附加的内部数据结构。通常，如果提供一个train()方法，则必须在调用使用内部字典的匹配方法之前调用它。

下一组方法是在目标识别中使用的一组匹配方法。它们每个都使用描述符列表，称为查询列表，并与训练好的字典中的描述符进行比较。在这组中，有三种方法：match()，knnMatch()和radiusMatch()。这些方法中的每一种以稍微不同的方式计算匹配。

match()方法需要通常的cv::Mat格式的单个关键点描述符列表queryDescriptors。在这种情况下，请记住每行代表单个描述符，每列是该描述符向量表示的一个维度。match()还需要一个cv::DMatch对象的STL向量，它可以填充各个检测到的匹配项。在match()方法的情况下，查询列表上的每个关键点将与列表中的“最佳匹配”匹配。

match()方法也支持可选的mask参数。与OpenCV中的大多数mask参数不同，该掩码不在像素空间中操作，而是在描述符的空间中。然而，mask的类型应该仍然是CV_8U。mask参数是cv::Mat对象的STL向量。该向量中的每个矩阵对应于字典中的训练图像之一[注17]。特定掩码中的每一行对应于queryDescriptors中的一行（即一个描述符）。掩码中的每个列对应于与字典图像相关联的一个描述符。因此，如果来自图像（对象）k的描述符j与查询图像中的描述符i进行比较，则masks[k].at<uchar>(i,j)应该是非零的。

下一个方法是knnMatch()函数，它需要与match()相同的列表描述符。然而，在这种情况下，对于查询列表中的每个描述符，它将从字典中找到特定数量的最佳匹配。该数由整型参数k指定。函数名中的“knn”代表*k*-最近邻。来自match()的cv::DMatch对象被替代成在knnMatch()方法中称为matches的cv::DMatch对象向量的向量。顶级向量的每个元素（例如，matches[i]）都与queryDescriptors中的一个描述符相关联。对于每个这样的

注17：请注意，这通常意味着“字典中的一个对象”。

元素，下一级元素（例如，matches[i][j]）是trainDescriptors中描述符的第j个最佳匹配。[注18]knnMatch()的掩码参数与match()的含义相同。knnMatch()的最后一个参数是布尔值compactResult。如果compactResult设置为默认值为false，则向量的匹配向量将包含一个向量条目，用于queryDescriptors中的每个条目——哪怕是那些没有匹配的条目（cv::DMatch对象的相应向量为空）。但是，如果将compactResult设置为true，那么这种非信息条目将简单地从匹配中移除。

第三种匹配方法是radiusMatch()。与搜索k个最佳匹配的k-最近邻匹配不同，半径匹配返回了查询描述符特定距离内的所有匹配。[注19]除了整数k代替最大距离maxDistance之外，参数及radiusMatch()的均值都与knnMatch()的相同。



不要忘记，在匹配的情况下，“最佳”由实现cv::DescriptorMatcher接口的单独派生类确定，所以“最佳”的确切含义在不同的匹配器可能也不同。另外，请记住，通常不会找到“最佳分配”，因此查询列表上的一个描述符可以匹配列表上的几个，反之亦然。

接下来三种方法——match()，knnMatch()和radiusMatch()——的替代形式支持两个描述符列表，通常用于跟踪。这些都与上述对应的输入相同，并加上trainDescriptors参数。这些方法忽略了内部字典中的任何描述符，而是将queryDescriptors列表中的描述符与提供的trainDescriptors进行比较。[注20]

在了解完这六种匹配方法之后，还需要了解一些处理匹配器对象的方法。read()和write()方法分别对应cv::FileNode和cv::FileStorage格式的对象，并允许你从磁盘读取和写入匹配器。尤其重要的是，当处理识别问题时，你可以通过从可能非常大的文件数据库加载信息来“训练”匹配器。这样可以节省你在每次运行代码时都需要加载实际的图像，并从每个图像重构关键点及其描述符。

最后，clone()和create()方法允许你分别创建一个描述符的副本或者通过名称创建一个新的描述符。第一种方法clone()采用一个布尔值emptyTrainData，如果为true，它将创建具有相同参数值的副本（被特定匹配器接受的任何参数），但不复制内部字典。

注18：顺序始终从最好的开始，并且按照第k个最佳向后排列（即$j \in [0, k-1]$）。

注19：然而，应当注意，这里的术语距离是由特定匹配器使用的度量定义的，并且既可以是也可以不是描述符的向量表示之间的标准欧式距离。

注20：请注意，即使将此第二个列表称为“训练”列表，也不会用add()匹配器“训练”列表；相反，它用来代替内部的“受过训练”的列表。

将emptyTrainData设置为false本质上是一个深层副本，它除了参数之外还复制字典。create()方法是一个静态方法，它将接受可以构造特定派生类的单个字符串。create()的descriptorMatcherType参数的当前可用值在表16-1中给出。个别情况的意义将在下一节中介绍。

表16-1：cv::DescriptorMatcher::create()方法descriptorMatcherType参数的可用选项

descriptorMatcherType 字符串	匹配器类型
"FlannBased"	FLANN（近似近邻的快速库）方法; 默认情况下将使用L2范数
"BruteForce"	使用L2范数进行元素直接比较
"BruteForce-SL2"	使用平方二分法进行元素直接比较
"BruteForce-L1"	使用L1范数进行元素直接比较
"BruteForce-Hamming"	使用Hamming距离进行元素直接比较[a]
"BruteForce-Hamming(2)"	使用多级Hamming距离的元素直接比较（两级）

a. 所有Hamming距离方法只能应用于用CV_8UC1类型编码的二进制描述符，即每个描述符字节的描述符的8个分量

核心关键点检测方法

近十年来，跟踪和图像识别取得了巨大的进步。在这个领域内，一个非常重要的主题是开发关键点，就像你们现在所知道的那样，图像的小碎片包含关于图像及其内容的最高密度的信息。关键点概念的最重要特征之一是，即使图像的分辨率变得非常高，它也允许将图像“消化”为有限数量的基本元素。在这个意义上，关键点提供了一种从潜在的非常高的维度像素表示中获得图像的方法，使其成为质量随着图像大小而增加但是其实际尺寸不变的更紧凑的表示。人们认为，人类视觉皮质将个体视网膜反应（基本上是像素）分组成更高级别的信息块，其中至少有一些类似于关键点中包含的信息类型。

早期的研究工作集中在如角点检测这样的概念上，研究者们试图通过越来越复杂的关键点，获得更有表现力的描述符（我们将在下一节中介绍描述符）。这些描述符可以获得各种计算机视觉工作所需的特性，比如旋转、尺寸不变性、微小仿射变换的不变性，而这些在先前的关键点检测器中并不存在。

然而，目前的技术状态是虽然有大量的关键点检测算法（和关键点描述符），但就是没有一个比其他的。因此，OpenCV库的方法一直为所有的检测器提供一个通用的界面，希望能够鼓励和促进在个人环境下实验和探索其相对的优点。有些快，有些比较慢。一些查找可以提取非常丰富的描述符的功能，有些则不能。有些展示一个或多个有用的不变性质，其中一些可能在你的应用程序中是非常必要的，有一些实际上可能对你可能还有实际的帮助。

在本节中，我们将依次了解每个关键点检测器，讨论其相对优点并深入研究每个检测器的实际原理，至少讨论深度足以让你感受到每一个检测器的可用性与其他的不同。正如我们已经了解的，对于每个描述符类型，都有一个检测器来定位关键点和描述符提取器。我们将在下面讨论每个检测算法。



一般来说，使用特征提取器在历史上与特定的关键点检测器相关联并不是绝对必要的。在大多数情况下，使用任何检测器找到关键点都有意义，然后可以继续使用任何特征提取器来表征这些关键点。然而，实际上，这两层通常是一起开发的，所以OpenCV库也使用这个模式

Harris-Shi-Tomasi特征检测器和cv::GoodFeaturesToTrackDetector

尽管其他人也提出了类似的定义，但最常用的角点定义是哈里斯[Harris88]提出的。这些角点被称为“哈尔角点”，可以被认为是原始的关键点，[注21]图16-11显示了一对图像上的哈尔角点，我们将继续用于本节中的其他关键点（为了方便进行视觉比较）。它们的定义依赖于一个小区域中的像素之间的自相关概念。简单来说，这意味着“如果图像被移动了少量（$\Delta x, \Delta y$）位置，它与原来的自己有多相似？”



图16-11. 同一车辆的两幅图像。在每个图像中，有1000个最强的Harris-Shi-Tomasi角点。注意，在右侧的图像中，背景中的角点比汽车上的角点更强，因此图像中大部分角点都来自这里

哈尔从以下自相关函数开始计算，对于图像强度像素$I(x, y)$：

$$c(x, y, \Delta x, \Delta y) = \sum_{(i,j)\in W(x,y)} w_{i,j}(I(i, j) - I(i + \Delta x, j + \Delta y))^2$$

注21：这些特征曾经与OpenCV库的旧版本中的“好跟踪的特征”（即cvGoodFeaturesToTrack()）相关联。这就是相关检测器现在被称为cv::GFTTDetector的原因，cv::HarrisCornerDetector这样可能更直观。

这只是点（x, y）周围的小窗口上的某个点（i, j）处的图像与移位（$\Delta x, \Delta y$）的某个其他点之间的平方差的加权和。加权因子$w_{i,j}$是高斯加权，使靠近窗口中心的平方差比离中心更远的平方差的权重更大。

哈尔推导出来的是少量的代数，并且由于Δx和Δy被假设为很小，所以近似值$I(i + \Delta x, j + \Delta y)$可以由$I(i, j) + I_x(i, j)\Delta x + I_y(i, j)\Delta y$）（这里$I_x$和$I_y$分别是$x$和$y$中的$I(x, y)$的一阶偏导数）。[注22]其结果以自相关的形式重新表达如下：

$$c(x, y, \Delta x, \Delta y) = [\Delta x \ \ \Delta y] M(x, y) \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

其中$M(x, y)$是由下式定义的对称自相关矩阵：

$$M(x, y) = \begin{bmatrix} \sum_{-K \le i,j \le K} w_{i,j} I_x^2(x + i, y + j) & \sum_{-K \le i,j \le K} w_{i,j} I_x(x + i, y + j) I_y(x + i, y + j) \\ \sum_{-K \le i,j \le K} w_{i,j} I_x(x + i, y + j) I_y(x + i, y + j) & \sum_{-K \le i,j \le K} w_{i,j} I_y^2(x + i, y + j) \end{bmatrix}$$

根据哈尔的定义，角点是图像中的自相关矩阵具有两个大的特征值的位置。实质上，这意味着在任何方向移动一小段距离都会造成图像改变。[注23]这种看待事物的优点在于，当我们只考虑自相关矩阵的特征值时，我们考虑对旋转不变的量，这很重要，因为我们正在跟踪的对象可能会旋转和转换。



在这种情况下，哈尔角点的这两个特征值比确定一个点是否是一个很好的跟踪特征（即，一个关键点）做得更多；它们还提供该点的识别签名（即，关键点描述符）。一般来说，虽然不是绝对普遍的，关键点的特征以这种方式与它们的描述符密切相关。在许多情况下，关键点本质上是和描述符相关联的（在这种情况下，$M(x, y)$的两个特征值）符合某些阈值标准。同时，值得注意的是，哈尔的原始阈值标准与Shi和Tomasi提出的阈值标准不一样，后者被证明对大多数跟踪应用来说有优势。

哈尔的原始定义涉及到$M(x, y)$的行列式，并减去其平方迹（具有一些加权系数）：

$$H = \det(M) - \kappa \mathrm{trace}^2(M) = \lambda_1 \lambda_2 - \kappa(\lambda_1 + \lambda_2)^2$$

注22：也即一阶泰勒近似。

注23：如果只有一个大的特征值，那么这一点就像一个边缘，因为边缘似乎不会改变图像，而垂直于边缘的运动会改变图像。如果没有大的特征值，则意味着你可以在任何方向上放置小窗口，什么都不会发生；换句话说，图像强度是恒定的。

然后，通过搜索该函数的局部最大值（并且通常也将该函数与预定阈值进行比较），找到“角点”（我们现在称为“关键点”）。称为“哈尔测度”的函数H将有效地比较M（在H定义中我们称之为λ_1和λ_2）的特征值，而不需要它们的显式计算。该比较隐含包含称为“灵敏度”的参数κ，后者可以有意义地设置为0到0.24之间的任何值，但通常设置在0.04左右。[注24]图16-12显示了一幅图像，其中一些单个关键点候选者周围的区域被放大显示。

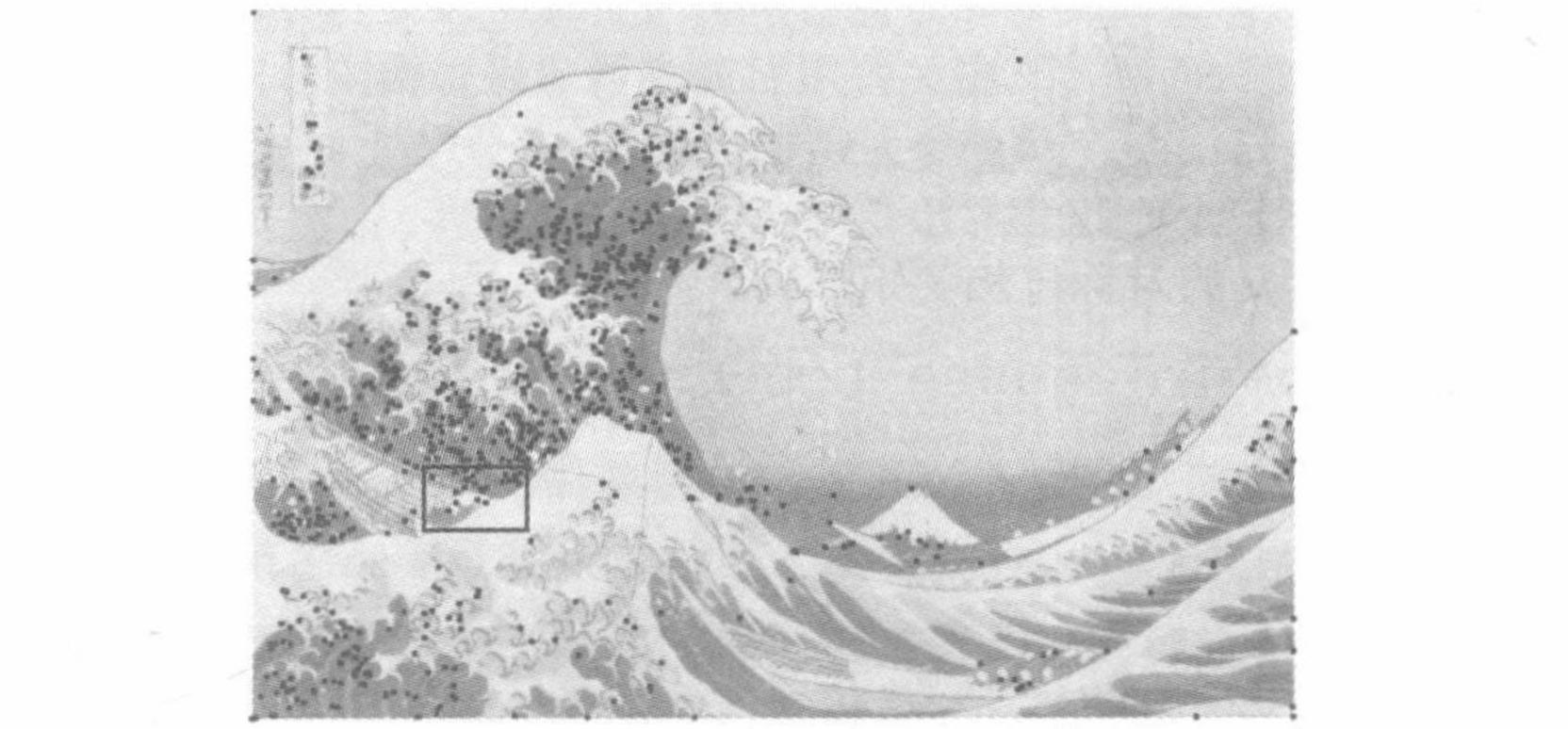


图16-12. 在经典图像（A）中，由Shi-Tomasi方法发现的关键点显示为黑点。下面是三个图像，它们是原图上小部分的放大图。左侧（B）显示（作为Xs）的点，不是关键点。这些点在两个维度上都具有小的特征值。在中心（C）中显示（作为Xs）的点也不是关键点；这些是边缘，并且具有与它们相关联的一个小特征值和一个大的特征值。右侧（D）是实际找到的关键点；对于这些点，两个特征值都很大。图中的椭圆显示了这些特征值的倒数

之后，Shi和Tomasi [Shi94]发现，只要两个特征值中较小的一个特征值大于最小阈值，就会产生良好的角点。Shi和Tomasi的方法不仅充分，而且在许多情况下比Harris的方法可以得到更为满意的结果。默认情况下，`cv::GFTTDetector`的OpenCV实现使用了Shi和Tomasi的测量，但我们稍后讨论的其他关键点检测器通常使用Harris的原始测量或其变体。

关键点检测器 Harris-Shi-Tomasi角点检测器也是`cv::Feature2D`（检测器部分）接口的最简单的实施。

注24：该数值越小，算法的灵敏度越高，因此0.04的值在更敏感的一边。

```
class cv::GFTTDetector : public cv::Feature2D {
public:
  static Ptr<GFTTDetector> create(
    int    maxCorners        = 1000,  // Keep this many corners
    double qualityLevel      = 0.01,  // fraction of largest eigenvalue
    double minDistance       = 1,     // Discard corners if this close
    int    blockSize         = 3,     // Neighborhood used
    bool   useHarrisDetector = false, // If false, use Shi Tomasi
    double k                 = 0.04   // Used for Harris metric
  );
...
};
```

cv::GFTTDetector的构造函数设置的参数包含了该算法所有基本运行参数。maxCorners代表你希望返回的角点的最大数量。[注25]qualityLevel表示将一个点认定为角点所需的最小低阶特征值（lower eigenvalue）。实际使用的最小特征值是qualityLevel和图像中观察到的最大低阶特征值（lower eigenvalue）的乘积。因此，qualityLevel的值不应大于1（一般将其设为0.10或0.01）。一旦选择出了角点的候选点，就可以进行进一步的筛选，最终的输出结果中同一区域不应出现两个角点。我们可以用来设置相邻角点间的最近像素距离（即两点之间的像素数量）。

blockSize是在计算导数的自相关矩阵时对应像素所在的区域。事实证明，对一个小窗内的点的导数求和绝大多数情况下都会获得比对单个点（即，blockSize为1）求导更好的结果。

useHarris值为true时，表示使用哈尔角点的定义而不使用Shi-Tomasi的定义。k是权重系数，用于在给定自相关矩阵Hessian的迹的情况下，同该矩阵的行列式相比较时的相关权重。

当然，当你要实际计算关键点时，可以使用detect()方法来实现，cV::GFTTDetector继承自cv::Feature2D基类。

附加函数 cv::GFTTDetector还支持使用set/get方法设置和检索各种属性。例如，你可以通过调用gfttdetector->setHarrisDetector(true)方法来打开Harris检测器，而不是默认的基于最小特征值的（Shi-Tomasi）GFTT算法。

一个简单的介绍

在内部，cv::goodFeaturesToTrack()和cv::GFTTDetector有一些特定的阶段：自

注25：由于通过detect()方法返回关键点的方法是填充cv::KeyPoint对象的STL向量，所以要求的关键点个数没有上限。然而，实际上，为了计算效率的目的来限制关键点的数量，或者在下游处理（特别是在实时应用中）约束计算时间，通常很有用。在任何情况下，根据自相关矩阵$M(x, y)$的较小特征值的大小，返回的角点都是“最佳”角点。

相关矩阵$M(x, y)$的计算、该矩阵的分析以及应用的某种阈值。关键步骤使用函数cv::cornerHarris()和cv::cornerMinEigenVal()完成：

```
void cv::cornerHarris(
  cv::InputArray  src,                                  // Input array CV_8UC1
  cv::OutputArray dst,                                  // Result array CV_32FC1
  int             blockSize,                            // Autocorrelation block sz
  int             ksize,                                // Sobel operator size
  double          k,                                    // Harris's trace weight
  int             borderType = cv::BORDER_DEFAULT // handle border pix
);
void cv::cornerMinEigenVal(
  cv::InputArray  src,                                  // Input array CV_8UC1
  cv::OutputArray dst,                                  // Result array CV_32FC1
  int             blockSize,                            // Autocorrelation block sz
  int             ksize,     = 3                        // Sobel operator size
  int             borderType = cv::BORDER_DEFAULT // handle border pix
);
```

这两个函数的参数完全类似于函数cv::goodFeaturesToTrack()，首先用Harris使用的特征值填充dst。

$$dst(x, y) = \det M^{(x,y)} - k \cdot (trM^{(x,y)})^2$$

第二次用Shi和Tomasi使用的特征值填充dst，即自相关矩阵$M(x, y)$的最小特征值。

如果你想实现自己的GFTT算法的变体，OpenCV提供最终的函数进行计算并给出图像上每个点的自相关矩阵的特征值和特征向量。这个函数叫cv::cornerEigenValsAndVecs()：

```
void cornerEigenValsAndVecs(
 cv::InputArray  src,                                   // Input array CV_8UC1
 cv::OutputArray dst,                                   // Result array CV_32FC1
 int             blockSize,                             // Autocorrelation block sz
 int             ksize,                                 // Sobel operator size
 int             borderType = cv::BORDER_DEFAULT // handle border pix
);
```

这个函数和cv::cornerMinEigenVal()之间唯一的显著区别是输出的性质。在这种情况下，结果数组dst将是类型CV_32FC6。六个通道将包含两个特征值，第一特征值的特征向量中的两个分量，以及第二特征值的特征向量中的两个分量（按顺序）。

简单的blob检测器和cv::SimpleBlobDetector

Harris、后来的Shi和Tomasi的工作成果所体现的角点检测的概念，代表了一个关键点概念的主要方法。从这个角度来看，关键点是存在于图像中大于正常数量的信息的点处的高

度局部化的结构。另一个替代观点是一个斑点的概念（图16-13）。斑点本质上没有那么明确的局部化，而是表示可能预期随时间有一定稳定性的感兴趣区域（图16-14）。



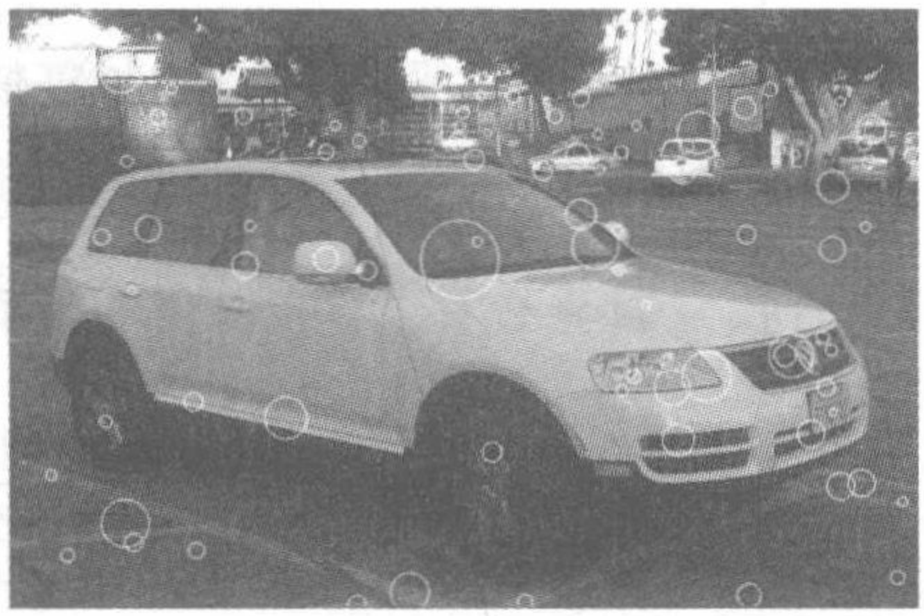
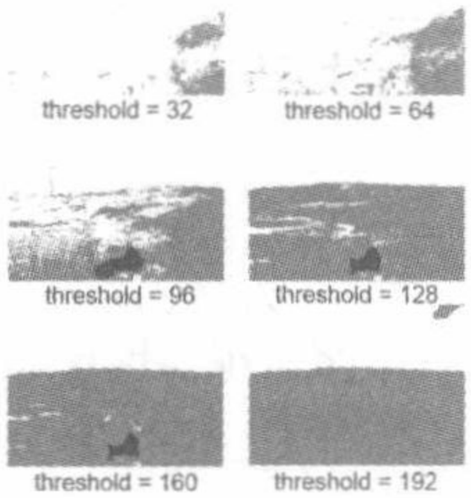


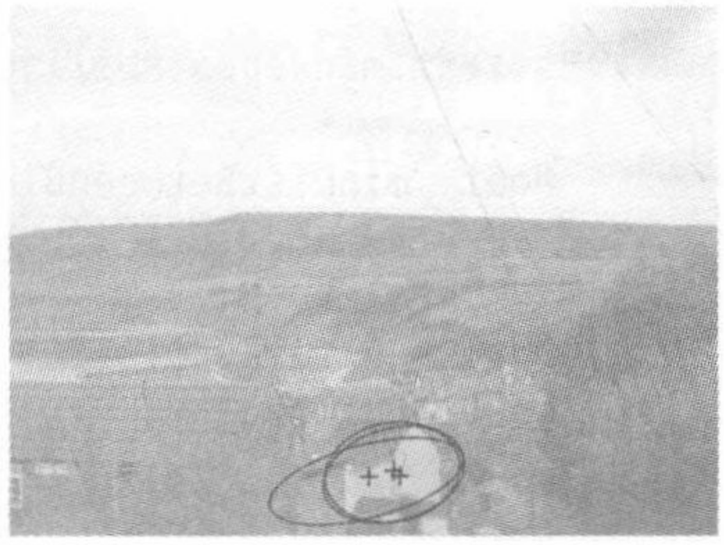
图16-13. 在含有相同车辆的两幅相似图像上运行简单的斑点检测器。两个图像中发现的斑点之间几乎没有一致性。Blob检测在简单的环境中预期要定位几个非常明确的对象时效果最好



(A)



(B)



(C)

图16-14. 从乡村场景（A）开始，生成六个阈值图像（B），并显示与原始图像底部中心的建筑物相对应的一组重叠的斑点候选项（C）。这些候选项将组合产生相关联的斑点的最终估计（未示出）。中心阈值图像中突出显示了影响这些斑点候选的轮廓（黑色）

有很多斑点检测算法。cv::SimpleBlobDetector类只是其中的一个实现。[注26]简单斑点检测器通过首先将输入图像转换为灰度图，然后从该灰度图像计算一系列阈值（二值）图像。二值图像的数量由算法的参数确定：最小阈值、最大阈值和阈值步长。一旦转换为二进制，连接的组件就会被提取出来——例如使用cv::findContours()——并且计算出来每个这样的轮廓的中心；它们是候选斑点中心。接下来，在空间中彼此相邻的候选斑点中心（由最小距离参数MinDistBetweenBlobs控制）和具有相邻阈值的图像（在应用阈值列表中不同一步）组合在一起。这些组合一旦确定，就会被分配一个半径和一个中心，它从形成组合的所有轮廓计算出来。生成的对象是关键点。

注26：正如我们将在本节中调查其他更复杂的特征检测器时所看到的，还有许多其他可能的方法来进行斑点检测。它们中的许多将作为更复杂算法的组成部分，我们将看它们如何工作，因为我们已经实现了它们。DoG，LoG和DoH都是斑点检测机制的例子。

一旦完成了斑点定位，就可以通过一些内置的过滤器进一步减少斑点的数量。过滤时可以按照颜色（其实也就是强度，因为我们现在处理的是一个单通道灰度图像），大小（即斑点像素所占面积）、圆度（实际斑点像素所占面积和斑点计算有效半径的圆的比值），“惯性比”（二阶距矩阵的特征值的比）或凸度（斑点的面积与其凸包的面积的比）来进行比较。

关键点查找　首先看看（有点简化的版本）blob检测器的声明。

```
class SimpleBlobDetector : public Feature2D {

public:
  struct Params {
      Params();
      float  minThreshold;              // First threshold to use
      float  maxThreshold;              // Highest threshold to use
      float  thresholdStep;             // Step between thresholds

      size_t minRepeatability;           // Blob must appear
                                         // in this many images
      float  minDistBetweenBlobs;       // Blob must be this far
                                         // from others

      bool   filterByColor;             // True to use color filter
      uchar  blobColor;                  // always 0 or 255

      bool   filterByArea;              // True to use area filter
      float  minArea, maxArea;          // min and max area to accept

      // True to filter on "circularity", and min/max
      // ratio to circle area
      bool   filterByCircularity;
      float  minCircularity, maxCircularity;

      // True to filter on "inertia", and min/max eigenvalue ratio
      bool   filterByInertia;
      float  minInertiaRatio, maxInertiaRatio;

      // True to filter on convexity, and min/max ratio to hull area
      bool   filterByConvexity;
      float  minConvexity, maxConvexity;

      void read( const FileNode& fn );
      void write( FileStorage& fs ) const;
  };

  static Ptr<SimpleBlobDetector> create(
    const SimpleBlobDetector::Params &parameters
      = SimpleBlobDetector::Params()
  );

  virtual void read( const FileNode& fn );
  virtual void write( FileStorage& fs ) const;
```

```
    ...
};
```

浏览这个声明可以看出，很显然，这里并没有太多的事情发生。有一个cv::SimpleBlobDetector::Params的定义，一个结构可以容纳实际运行一个简单的blob检测器所需的所有信息，一个构造函数（它采用一个Params参数）和read()和write()函数，这将允许我们存储我们的检测器的状态。当然，还有从cv::FeatureDetector接口继承的所有重要的detect()例程。

我们已经知道了detect()成员的工作原理，其意义在于它对所有特征检测器都是通用的。对我们来说真正重要的是如何在Params参数中设置构造函数的参数。第一组五个参数控制算法的基本功能。我们使用thresholdStep，minThreshold和maxThreshold来配置阈值化图像的集合。首先从minThreshold开始，并且每次加上thresholdStep直到达到但不包括maxThreshold。通常设置50~64之间的值为初值，每次增加一个小值（例如10），增加到220至235之间，从而避免强度分布信息较少的端点。minRepeatability确定了必须包含重叠候选斑点的（连续）阈值图像的数量，以便将候选斑点组合成一个斑点。这个数字通常是一个较小的整数，但很少小于2。“重叠”的实际含义由minDistBetweenBlob控制。如果两个候选斑点的中心小于此距离，就会被认为与与同一斑点的相关联。因为这是像素单位，所以应该进行图像缩放。cv::SimpleBlobDetector::Params的默认构造是将其设置为10，这一般适用于大约640×480的图像。

剩余的参数影响不同的过滤选项，并且排列成小的组合，每个组合包含打开或关闭特定过滤功能的布尔值以及一个或两个控制过滤（如果打开）的参数。第一个是filterByColor，它只有一个关联的参数。参数blobColor是保留候选斑点所需的强度值。因为候选斑点在二值阈值图像上生成，所以只有强度值为0和255才有意义。使用前者仅提取黑斑，后者仅提白斑，将特征全部关闭，可以获得两种斑点。

参数filterByArea（如果为true）将仅保留区域大于或等于minArea但严格小于maxArea的斑点。类似地，参数filterByCircularity（如果为true）将仅保留圆度大于或等于minCircularity但严格小于maxCircularity的斑点。对于filterByIntrit，minInertiaRatio和maxInertiaRatio，以及filterByConvexity，minConvexity和maxConvexity也是一样的。[注27]

注27：请注意，圆度、惯性（或惯性比）和凸度的确切定义在blob检测器的算法描述中提到过。

FAST特征检测器和cv::FastFeatureDetector

最初由Rosten和Drummond [Rosten06]提出的FAST（加速段测试的特征）特征检测算法是基于将点*P*与其包围圆内的点集的直接比较的思想（图16-15）。基本思想是，如果附近的几个点与*P*类似，那么*P*将成为一个很好的关键点。SUSAN算法是这个想法的早期实现，与*P*周围的所有点进行比较。FAST算法可以认为是SUSAN的继承者，它以两种方式改进了这一想法。

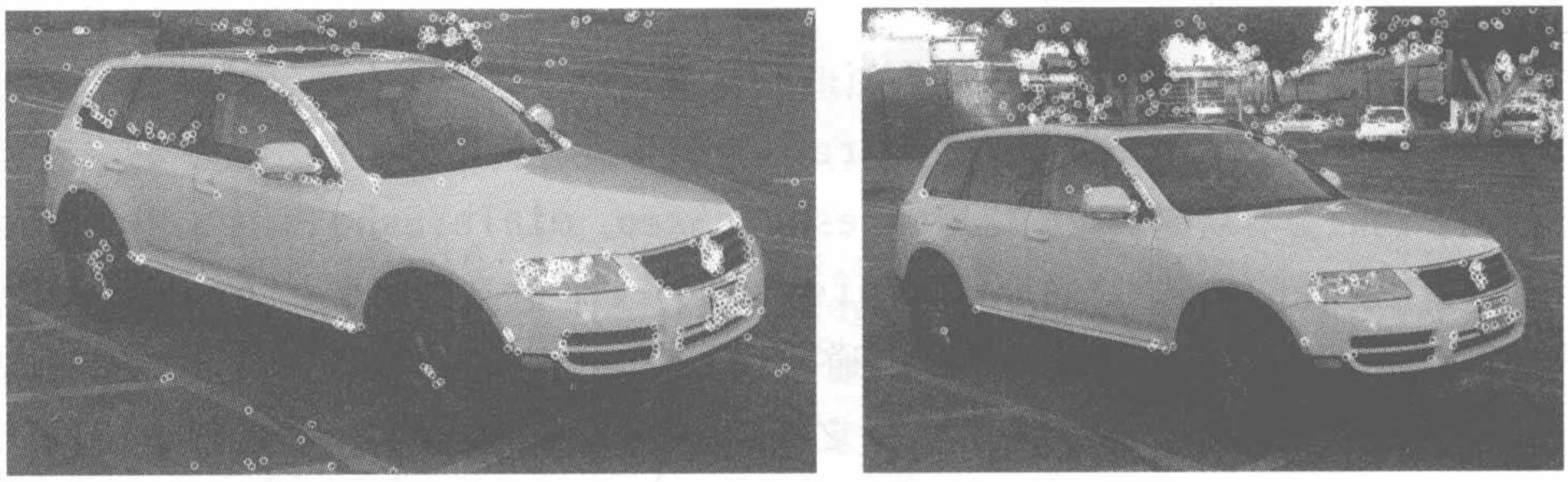


图16-15. 同一辆车的两幅图像。每个图像中都有1000个最强的FAST特征。注意，在右图中，与Harris-Shi-Tomasi的特征一样，背景中的角点比汽车上的角点更强，右图中大部分角点都来自背景

第一个区别是FAST只使用*P*周围的点。第二个是环上的各个点被分类为比P更暗，比P更亮，或者类似于*P*。这个分类阈值是t ，低于$I_p - t$就是较暗像素，高于$I_p + t$就是较亮像素，在此之间就是相似像素。一旦完成分类，FAST检测器就需要环上的一些或者比*P*更暗或者比*P*更亮的邻近点。如果环的像素总数为*N*，那么这条只包含更亮或更暗的弧必须至少包含$N/2 + 1$个像素（即，环上总数的一半以上）。

该算法已经非常快，但是我们还有另外一个想法。该测试允许一个仅测试四个等距点的更方便的优化。在这种情况下，如果至少有一对比*P*更亮或更暗的连续点，则点*P*就不能是FAST特征。实际上，这种优化大大减少了搜索整个图像所需的时间。

到目前为止所描述的算法的问题在于倾向于将多个相邻像素全部返回为角点。例如，在图16-6中，*P*点上方的像素等等也是FAST关键点。一般来说，这是不可取的。

为了避免这个问题，FAST算法定义每个角点的得分，并且可以删除与较高分数的关键点相邻的所有关键点。我们通过首先计算更亮的像素与中心像素之间的绝对差值的总和，然后对较暗的像素进行相同的处理，最后采用这两者中较大的一个来构造分数。

$$score = \max\left(\sum_{x\in\{brighter\}} |I_x - I_P| - t,\ \sum_{x\in\{darker\}} |I_x - I_P| - t\right)$$

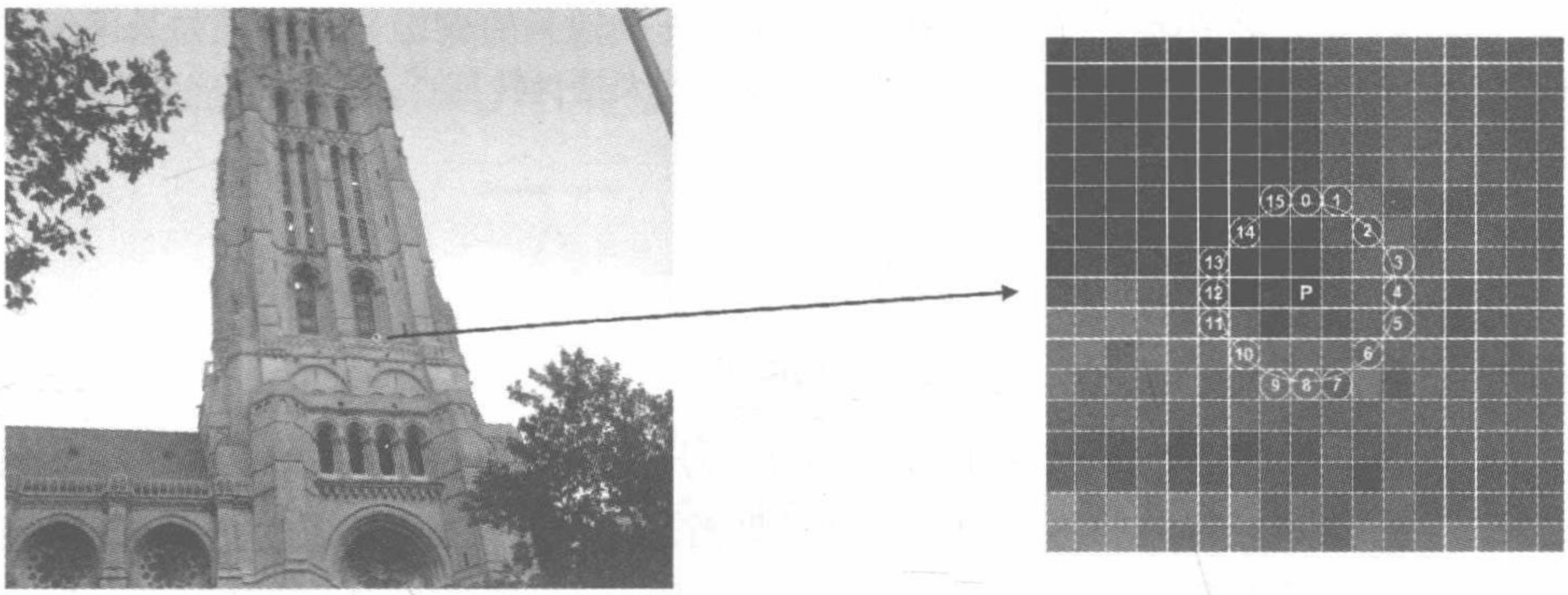


图16-16. 点*P*是FAST算法的关键点候选者。影响*P*分类的点的圈由*p*周围的圆确定。在这种情况下，该圆上有16个像素，这里编号为0-15

值得注意的是，主要是因为稍后在讨论ORB功能时，我们将再回到这一点，所以FAST特征如此定义并没有任何内在的方向。

关键点查找器 FAST特征检测器非常简单，看起来非常像哈尔角点检测器cv::GoodFeaturesToTrackDetector：

```
class cv::FastFeatureDetector : public cv::Feature2D {
public:
  enum {
    TYPE_5_8  = 0,                    //  8 points, requires 5 in a row
    TYPE_7_12 = 1,                    // 12 points, requires 7 in a row
    TYPE_9_16 = 2                     // 16 points, requires 9 in a row
  };

  static Ptr<FastFeatureDetector> create(
    int    threshold        = 10,      // center to periphery diff
    bool   nonmaxSupression = true,    // suppress non-max corners?
    int    type             = TYPE_9_16 // Size of circle (see enum)
  );
...
};
```

cv::FastFeatureDetector的构造函数有三个参数：阈值、布尔标志和运算符类型。Threshold以像素强度单位进行测量，因此是整型。布尔值nonMaxSupression打开或关闭得分较低的邻近点的抑制。最后一个参数设置运算符的类型，此类型确定采样点圆的周长。有三种可用类型，在cv::FastFeatureDetector类中定义为枚举。每种类型都指定圆的周长和该圆的中心被认为是关键点所需的邻近点的数量。例如，值cv::FastFeatureDetector::TYPE_9_16表示16点中的9点应该比中心点都更亮或者都要暗。



在大多数情况下，需要将阈值设置为一个较大的数值，如30。如果阈值太低，在实际强度变化非常小的区域将获得大量伪特征点。

SIFT特征检测器和cv::xfeatures2d::SIFT

最初于2004年由David Lowe提出的SIFT特征（尺度不变特征变换）[注28] [Lowe04]被广泛使用，是许多随后开发的特征的基础（参见图16-17）。与许多其他特征类型相比，SIFT特征计算开销很大，但是具有高度的表达能力，因此非常适合跟踪和识别任务。

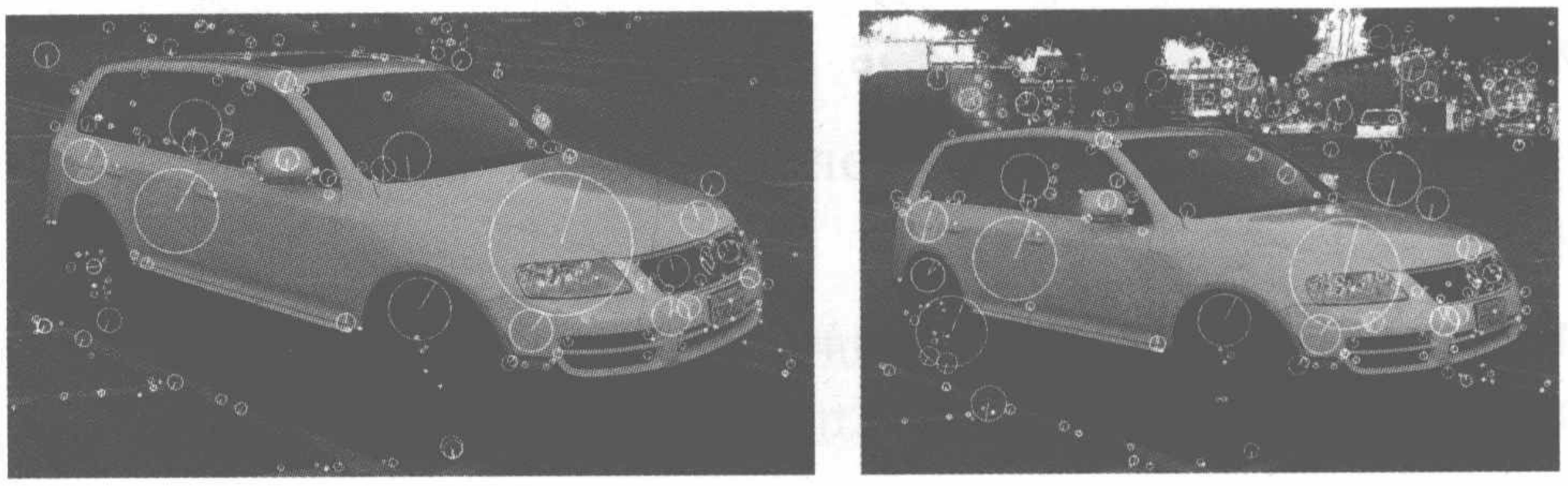


图16-17. 来自不同视角的相同车辆的两幅图像。在左图，找到237个SIFT特征，而在右图找到490。在这种情况下，你可以看到车上的特征相对稳定，可以看到很多对应关系。尽管在右图的背景中发现了更多的特征，但两幅图像中车辆上的特征密度大致相同

SIFT算法的初始阶段会计算输入图像和一组高斯核（这组高斯核的尺寸依次变大）之间的卷积，这赋予SIFT特征尺度不变的特性，同时也是SIFT算法如此命名的原因。然后将这些卷积和另一组卷积（图像与更大的高斯核卷积的结果）组合起来。该过程的结果是一组新的图像，这组图像和高斯（DoG）算子的差相类似。给定这组图像，可以将其看做堆栈，堆栈中每幅图像中的像素不仅与其所在图像中的相邻像素（有8个）进行比较，而且与堆栈的上下图像中对应的自身位置及其相邻位置的像素进行比较（上层9个像素，下层9个像素）。如果一个像素高斯卷积的差比这些相邻的26个像素都高，就会被认为是高斯算子的差尺度空间极值（图16-18）。

这里其实可以很直观地想象出来。比如我们现在有一幅除了中心区域为白色而其余皆为黑色的图像。高斯核的差（图16-19）将会在零交叉正好位于白色边缘时给出最强的响应。这种情形在计算时，高斯核的正值部分会乘以白色区域的正像素值，而高斯核的负

注28：和cv::features2d中使用的的开放和免费算法不同，cv::xfeatures2d使用的算法可能有专利问题，因此被放到一个特殊的*opencv_contrib*目录。

值部分则会全部乘以黑色背景对应的零像素值。相邻位置或在相同位置像素值大小不同都将给出更弱的响应。这样就可以找到白色区域的“特征”的位置和尺度了。

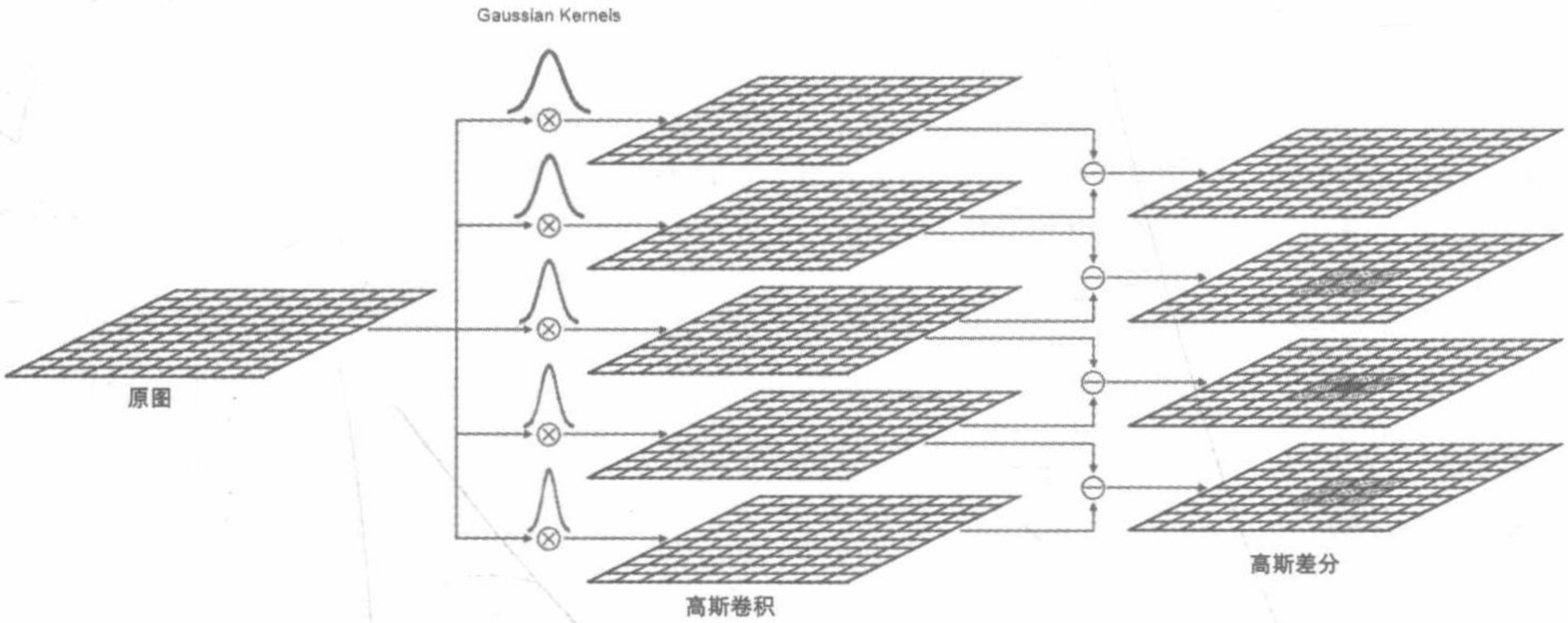


图16-18. 我们通过首先用各种大小的高斯内核卷积原始图像，然后在相邻尺寸的卷积之间计算差分图像来定位尺度空间极值。在差分图像中，在相同层和相邻层中将每个像素（显示为实心方块）与其所有邻居（示为Xs）进行比较。如果高斯信号的差比三层图像上所有相邻像素都强，则认为该像素是尺度空间极值

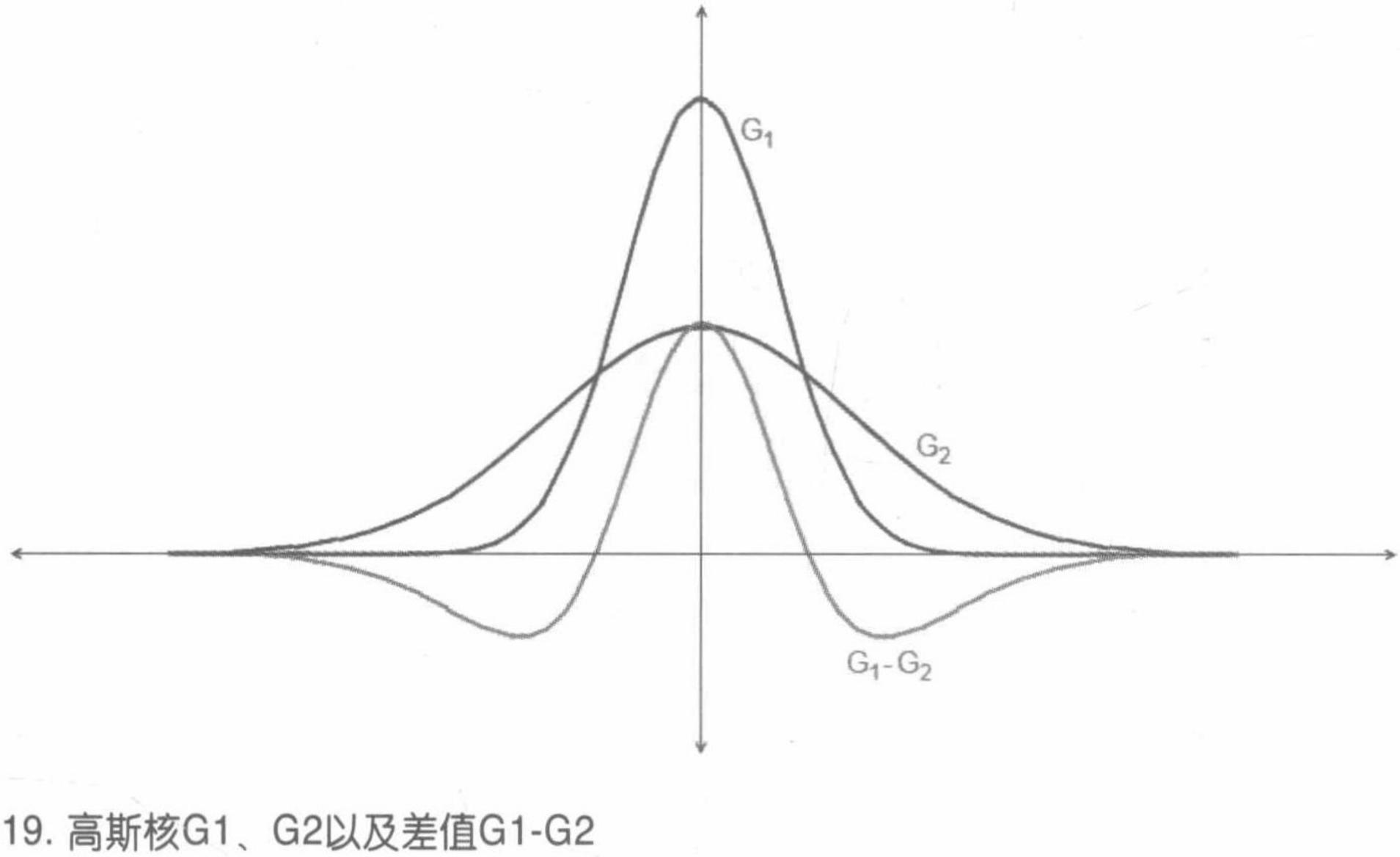


图16-19. 高斯核G1、G2以及差值G1-G2

一旦发现了一组特征，该算法就会对每个特征进行测试，确定其作为特征的质量并改进对其位置的估计。它通过将抛物面拟合到极值周围的3 × 3 × 3体素（三维为x，y和scale）来实现。从这个二次形式中，我们可以提取两个重要的信息。第一个是要添加到关键点位置的偏移量；该偏移是用于空间位置的亚像素校正，并且在尺度上以及从原始高斯卷积集可用的离散尺度之间进行内插。第二个是在这个极点处的局部曲率的估计，其形式为Hessian矩阵，其行列式可以用作阈值，通过该阈值来拒绝低辨别力的关键

点。类似地，该矩阵的纯空间部分的特征值之间的大比率表示该特征主要是边缘（而不是“角点”），并且也是候选特征被拒绝的方式。读者可以基于这些考虑与我们在本章前面看到的Harris角或Shi-Tomasi特征中使用的品质因数进行比较。

一旦找到所有这样的尺度空间极值，SIFT算法就会继续为每个这样的对象构造一个描述符，如图16-20所示。第一步是将关键点分配一个方向。该方向基于比较在关键点周围点的方向导数，并且选择对应于导数最大的方向。[注29]一旦找到这样的方向，就可以相对于这个主要方向分配描述符的所有后续属性。通过这种方式，SIFT特征不仅具有尺度不变性，而且还具有方向不变性——后者是说即使在给定新的旋转图像时，也能找到相同的关键点；它的方向是不同的，但是其余的特征描述符仍会进行匹配，因为它们是相对该方向进行测量。

最后，随着比例和方向的计算，可以开始计算局部图像描述符。局部图像描述符也可以由局部图像梯度形成，但是是在局部区域相对于描述符方向旋转到固定方向之后。接下来，它们被聚集到一起（通常在关键点周围的4×4模式中有16个区域或更多），并且对于每个区域，会根据关联区域中的所有点创建角度直方图。通常这个直方图将有8个项目。将每个区域的8个项目与16个区域相结合，得到具有128个分量的向量。这个128分量的向量就是SIFT关键点描述符。这种大量的分量对SIFT关键点的高度描述性至关重要。

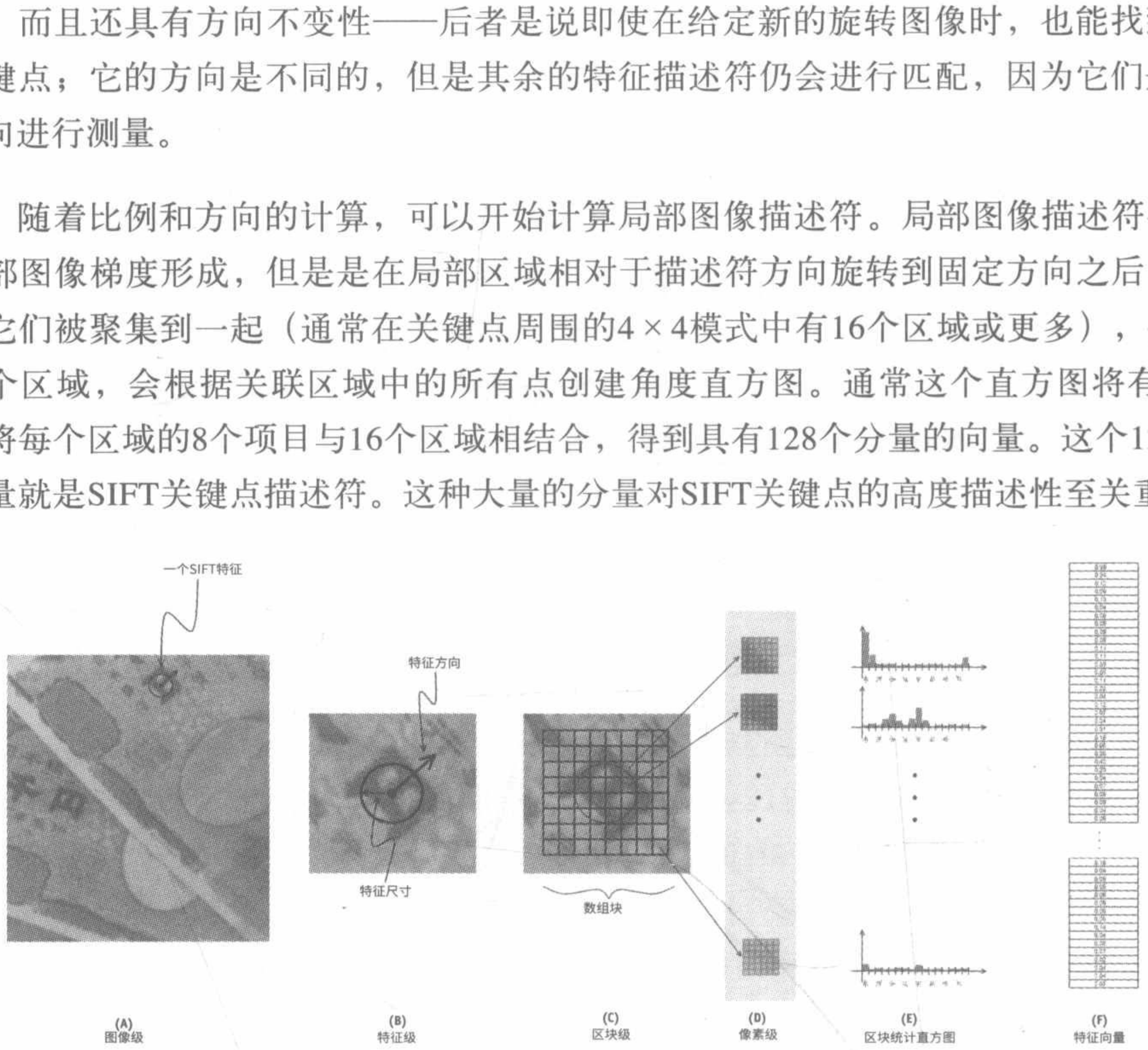


图16-20. 从图像(A)中提取SIFT特征。该特征具有大小和方向，如(B)所示。特征周围的区域被划分为块(C)，并且对于每个块，针对小区(D)中的每个像素计算方向导数。这些方向导数被聚合成每个块(E)的直方图，并且所有块的所有直方图中的每个bin中的幅度被级联成特征(f)的向量描述符

注29：算法的做法是首先使用已经确定的比例重新缩放关键点周围的像素。然后，在这种尺度归一化图像中，在x和y方向上使用Sobel差分算子，然后转换为极数坐标形式（幅值和方向）。对于特征区域中的每个点，这些差分算子先放入方向的直方图中，每个方向上的值根据其幅值大小进行加权。最后，找到直方图的最大值，用抛物线拟合该最大值及其相邻的值，并且将该抛物线的最大值用作特征整体方向的内插角度。

关键点查找器和特征提取器 OpenCV中的SIFT算法实现了cv::Feature2D接口的特征检测和描述符提取部分，并且像之前一样，建议使用detectAndCompute()方法，它将关键点检测和特征提取组合到一个步骤中。请注意，由于SIFT算法申请了专利，因此从OpenCV 3.0开始，我们将其放在opencv_contrib资源库中的xfeatures2d模块中。这是cv::xfeatures2d::SIFT对象的类定义的（稍微缩略的版本）：

```
class SIFT : public Feature2D {

public:
  static Ptr<SIFT> create (
    int    nfeatures          = 0,    // Number of features to use
    int    nOctaveLayers      = 3,    // Layers in each octave
    double contrastThreshold = 0.04, // to filter out weak features
    double edgeThreshold      = 10,   // to filter out "edge" features
    double sigma              = 1.6   // variance of level-0 Gaussian
  );

  int descriptorSize() const;         // descriptor size, always 128
  int descriptorType() const;         // descriptor type, always CV_32F
   ...
};
```

cv::xfeatures2d::SIFT构造函数所需的所有参数都用于构造尺度空间图像表示以及算法的关键点检测部分。在目前的实现中，实际的特征描述符总是用固定的参数集来计算。[注30]

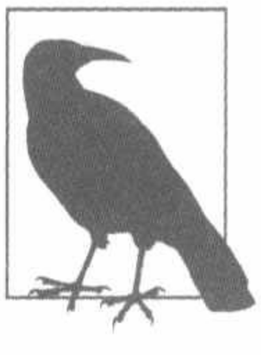
第一个参数nfeatures表示你想从图像中计算出的最大特征数。如果将其设置为默认值0，则算法将返回可以找到的所有特征。下一个参数nOctaveLayers可以确定每个*Octave*（图像金字塔中的图像）应该计算多少层（不同尺度的高斯卷积）。实际计算的层数是参数nOctaveLayers加3。因此，图16-18显示了金字塔中每个图像的五个图层的情况，这对应于nOctaveLayer的两个值。

接下来的两个参数是用于确定是否应该保留所找到的候选关键点的阈值。回忆一下一旦通过缩放空间搜索生成的候选关键点，那么关键点就将进行两次测试。第一个测试是DoG操作符的局部极值是否与周围区域完全不同。这个测试是contrastThreshold的值；此参数的典型值为0.04（默认值）。第二个测试与空间特征值的比例有关，并且用于滤除边缘的目的。此测试与edgeThreshold的值相反；此参数的典型值为10.0（默认值）。

最后一个参数sigma用于图像的预光滑处理，这也有助于设置尺度空间中第一层的尺

注30：一般来说，SIFT使用128个元素描述符最为合适。类似地，描述符计算使用的一些其他值（例如，放大倍数，如果你对SIFT很了解）同样有基本上始终保持不变的值。在将来的实现中，如果发现这样做有价值，其中一些可能会被添加进去。

度。典型的值是一个像素的倍数（默认为1.6像素），但对于包含一点噪声或其他干扰的图像来说，这个值稍大一些是很有用的。



你可以通过用自己的高斯滤波器进行卷积来预过滤图像，而不是使用sigma参数，但是如果这样做，算法将不会意识到在过滤中使用的尺度下没有信息。因此，它将浪费时间计算层数来搜索不存在的特征。因此，使用提供的预调节的σ参数更有效。实际上，这等于告诉算法“我不关心任何小于这个阈值的东西”。

创建cv::xfeatures2d::SIFT对象后，可以使用descriptorSize()和descriptorType()函数来查询要计算的特征的大小以及特征向量将包含的元素的类型。这两个函数总是返回128和CV_32F，但是当你使用多种特征对象并通过基类指针处理它们时，这通常很方便，只不过需要查询单个特征对象来查找返回的特征向量。

你将使用的主要函数是重载的detectAndCompute()方法。根据给定的参数，此运算符将只是计算关键点或计算关键点及其相关描述符。只计算关键点的情况只需要三个参数：img，mask和keypoints。第一个参数是你要从中提取关键点的图像（可以是彩色或灰度图像，但在前一种情况下，它将在算法开始之前在内部转化为灰度表示）。image参数应始终为CV_8U类型。使用mask参数，以便将生成的关键点仅仅限制在某些指定区域内。mask数组应该是CV_8U类型的单通道数组，但如果不需要此类过滤，则可以将其设置为cv::noArray()。cv::xfeatures2d::SIFT::detectAndCompute()的下一个参数是keypoints，它必须是对cv::KeyPoint对象的STL样式向量的引用。用于保存SIFT找到的关键点。

下一个参数descriptors，是一个输出数组，类似于我们已经看到的其他特征点描述符。如果descriptors是数组，则数组的每一行都将是一个单独的描述符，并且这些行的数量等于关键点的数量。

最后一个参数是布尔值useProvidedKeypoints。如果此参数设置为true，则不会搜索关键点，而是将关键点参数视为输入。在这种情况下，在这种情况下，将为关键点向量中指示的每个关键点生成描述符。

SURF特征检测器和cv::xfeatures2d::SURF

SURF特征（加速鲁棒特征）[注31]最初由Bay等人于2006年提出[Bay06，Bay08]。并且在许多方面是我们刚刚讨论的SIFT特征的演变（参见图16-21）。SURF的创造者对SIFT特

注31：与SIFT的情况一样，由于SURF也是一项获得专利的算法，因此OpenCVC将其放置在3.0版本的*opencv_contrib*库的xfeatures2d模块中。

征的不同组成部分的兴趣在于可以用更多的计算高效的技术来替代，这些技术可能（主要）会在识别任务中产生类似或更好的性能。所产生的特征不仅计算速度快得多，而且在许多情况下，它们微简的性质对SIFT特征观察到的方向或照明变化的鲁棒性也更强。

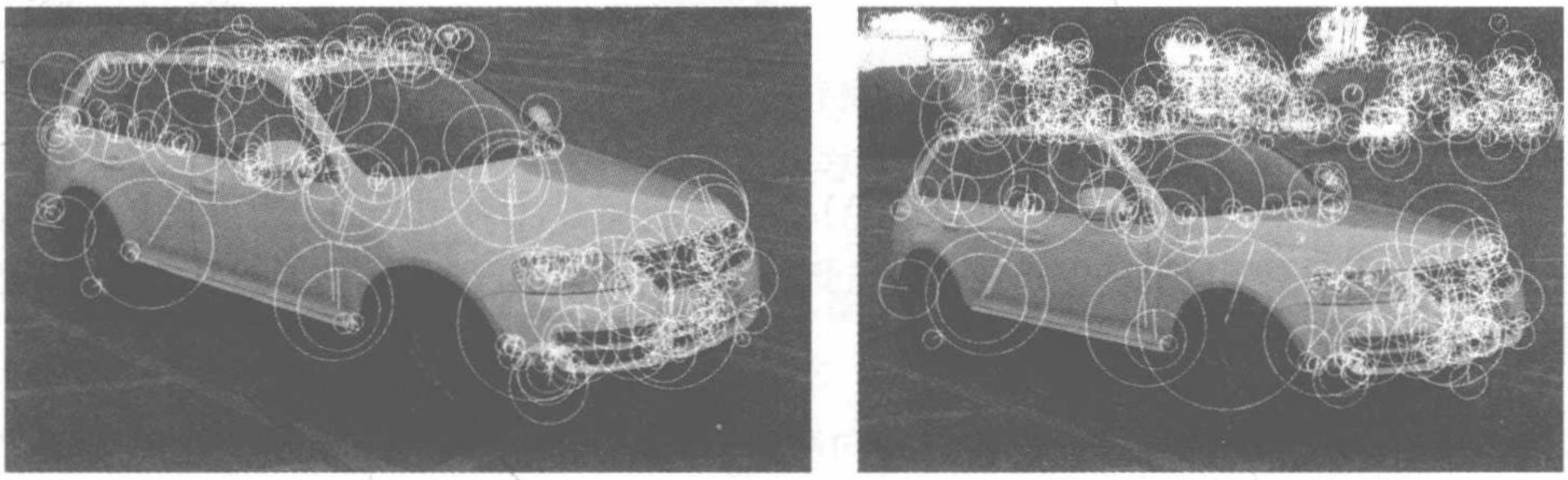


图16-21. 从两个不同的方向为相同的车辆计算SURF特征。在左边，找到了224个特征，而在右边发现了591个特征，许多新特征与右侧图像中的背景有关。SURF特征可以像SIFT特征一样定向。此图像生成的hessianThreshold设置值为1500

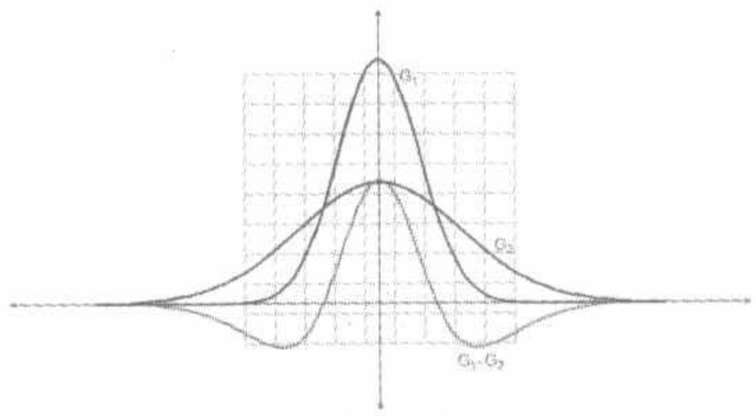
与SURF特征检测器操作的几个阶段相关的是积分图像的概念。回想一下，我们在第12章中遇到了积分图像，在那里我们看到它们使我们能够对整个图像进行单一变换，此后，转换后的图像使我们能够只用少量简单的操作来计算任何矩形区域的和。SURF特征严重依赖于可以通过使用积分图像技术大大加速的计算。

像许多其他检测器一样，SURF使用局部Hessian[注32]在给定的点上定义了一个关键点。之前我们看到，为了引入尺度的概念，SIFT检测器使用具有稍微不同宽度的高斯卷积的差来计算局部Hessian（图16-19）。然而，SURF检测器通过与近似两个高斯内核的差的盒式滤波器进行卷积来计算局部Hessian（图16-22）。[注33]这些盒式滤波器的主要优点是我们可以使用积分图像技术。

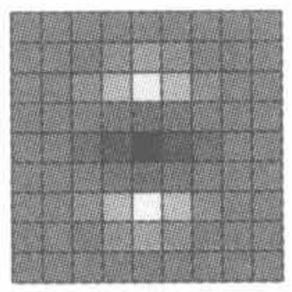
因为盒式滤波器的计算成本并不随着滤波器的尺寸而变化（因为积分图像），所以不必像SIFT那样生成图像的尺度金字塔。相反，可以使用各种更大的盒式滤波器来评估不同尺度的Hessian。从对这些盒滤波器的响应中，关键点特征在SURF中也定义为该Hessian的行列式超过某个阈值的局部极值。

注32：Hessian通常被理解为二阶导数的矩阵。然而，在这种情况下，实际上是所谓的“高斯二阶导数”的矩阵，它们由 $\frac{\partial^2}{\partial x_i \partial x_j} G(\vec{x}, \sigma)$ 定义，其中是大小为 $G(\vec{x}, \sigma)$ 的归一化高斯，图像在逼近导数之前被卷积。

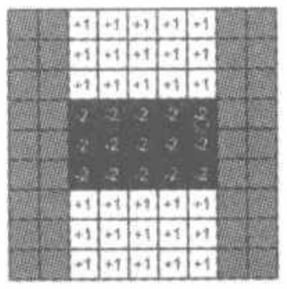
注33：请注意，在SIFT的情形下，实际是与两种不同的高斯内核进行卷积，然后减去结果。在SURF情况下，首先减去内核，并使用差分内核完成单个卷积。这两个操作是等效的，但是在SURF使用的盒滤波器近似的情况下，第二种操作更自然、更有效。



(A)高斯差分



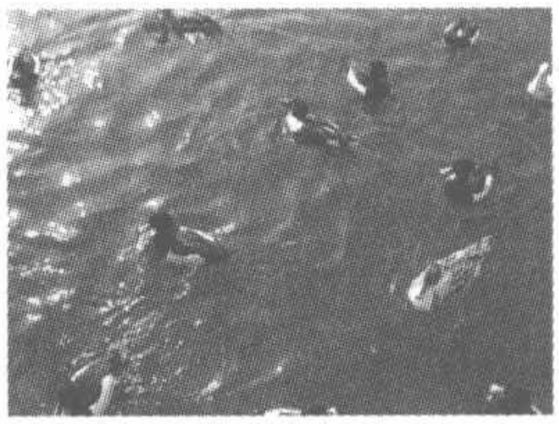
(B)DoG高斯差分



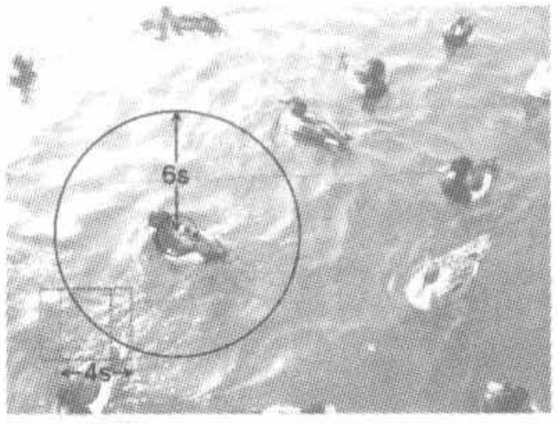
(C)盒滤波器

图16-22. 两个连续高斯核的差异如左图（A）所示。在中心（B）中显示出了离散的9×9滤波器核，其近似于垂直方向上的二阶导数。右侧（C）显示了DoG滤波器核心的一个盒滤波器近似值

像SIFT一样，SURF包括一个特征的方向概念，我们通过再次使用积分图像来估计特征周围区域的局部体积梯度。我们使用一对简单的Haar小波（图16-23）来近似局部梯度，并将这些小波应用于尺度空间的不同区域发现极值区域。如果特征的尺度被发现为s，那么我们使用大小为4s的小波来计算梯度，这些小波在以特征为中心的半径为6s的区域间隔一段距离（图16-23B）。然后我们通过考虑大小为的角度的滑动窗口来聚合这些梯度估计。通过对该方向窗口中的所有梯度进行求和（根据区域与特征中心距离确定加权因子），我们选择最大方向作为特征的方向（图16-23D）。[注34]一旦已经计算了方向，则可以以相对于该方向的方式生成特征向量，并且与SIFT一样，该特征具有方向不变性。



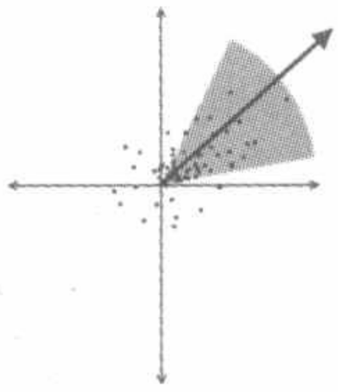
(A) 原始图像



(B) SURF候选区域和小波窗



(C) 小波（同尺度）



(D) 得到的梯度

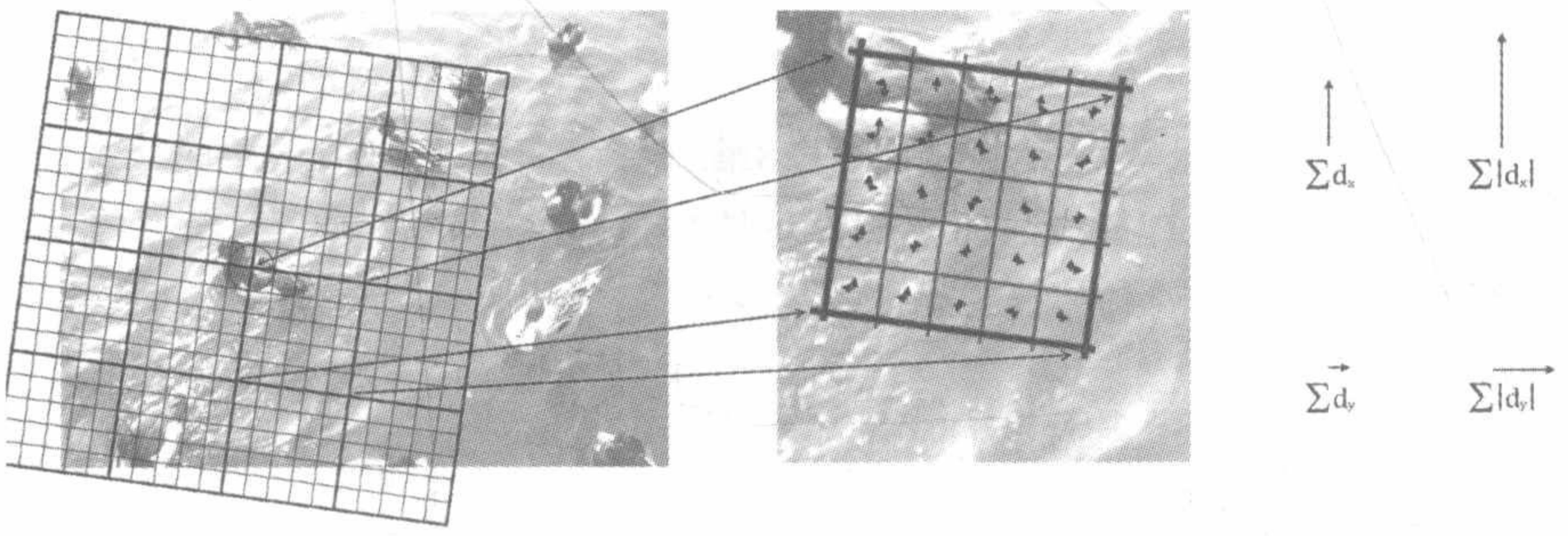
图16-23. 我们通过分析发现尺度空间极值的区域(B)来确定图像的SURF方向(A)。使用两个简单的小波(C)来近似局部梯度，并与该极值附近许多位置的图像进行卷积（在B中由实心圆表示的区域内规则采样的虚线框）。从以这种方式测量的所有梯度的分析中提取最终方向(D)

特征本身也以与SIFT特征设计相似的方式计算。对于尺度的特征，以特征为中心的20s×20s区域首先在4×4网格中被分为16个单元格。该网格根据刚刚计算的方向给出的角度相对于特征旋转。对于每一个这样的单元格，25对哈尔小波（与图16-23c所示的相同，只不过小得多）用于近似5×5阵列中的图像的x和y梯度（在4×4网格的每个单元格

注34：程序可能看起来很复杂，但请注意，计算此方向所需的实际评估数量实际上非常小。只有9个单元格，每个单元只需要6个其他单元，这81个点不到500次操作就计算出来了。

内）。[注35]对于这种类型的每个单元，计算25个x方向小波卷积的和以及25个y方向的小波卷积的和。对每个方向，计算总和及其绝对值之和。这个为4×4网格中的每个单元格提供4个数值，或总共64个数值。这64个值形成单个SURF的64维特征向量中的特征（图16-24）。

有一个变种SURF特征，称为“扩展”SURF特征，它们将卷积的总和分成八个和而不是四个。它通过将对应的y方向小波卷积为正的x方向小波卷积的那些值和相应的y方向小波卷积为负的x方向小波卷积的值相加，并且类似地，对于y方向小波卷积取决于相应的x方向卷积的符号。结果描述符较大，这意味着匹配它们会更慢，但在某些情况下，扩展特征的更强大的描述能力可以提高识别性能。



(A) 特性周围的4×4单元格　　(B) 5×5子单元格　　(C) 每个子单元格的四个值之和

图16-24. 我们通过估计400个子单元中的每一个的梯度来计算SURF特征。特征周围的区域首先划分为4×4的单元格(A)。然后将每个单元划分成25个子单元格，并为每个子单元估计方向导数(B)。然后将子单元的方向导数求和，以计算大网格(C)中的每个单元格的四个值

关键点查找器和特征提取器　与SIFT一样，OpenCV中SURF的最佳和最新的实现使用cv::Feature2D接口。构造过程由cv::xfeatures2d::SURF对象的单个构造函数完成，该对象为关键点查找以及特征描述符提取以及其他一些有用的函数提供了一个接口。以下是SURF对象的（略微缩写）定义：

```
class cv::xfeatures2d::SURF : public cv::Feature2D {
public:
  static Ptr<SURF> create (
    double hessianThreshold = 100,   // Keep features above this
    int    nOctaves         = 4,     // Num of pyramid octaves
    int    nOctaveLayers    = 3,     // Num of images in each octave
    bool   extended         = false, // false: 64-element,
                                     //  true: 128-element descriptors
```

注35：请注意，这里使用的x和y是相对于特征描述符的旋转坐标系，而不是图像的坐标系。

```
    bool   upright          = false, // true: don't compute orientation
                                     //  (w/out is much faster)
  );

  int descriptorSize() const;        // descriptor size, 64 or 128
  int descriptorType() const;        // descriptor type, always CV_32F

  ...
};
typedef SURF SurfFeatureDetector;
```

cv::xfeatures2d::SURF对象的构造方法create()有五个参数用于配置算法。第一个hessianThreshold设置Hessian行列式的阈值，该阈值是为了将特定局部极值视为关键点所需的。默认构造函数分配的值为100，但这是一个非常低的值，可以解释为“所有这些”。典型值的一个合理选择的是1500。[注36]

extended参数告诉特征提取器使用扩展（128维）特征集（在上一节中提到过）。参数upright表示不需要计算特征方向，它们都应该被视为“垂直”；这也称为“直立的SURF”或者只是“U-SURF”。



在诸如汽车或移动机器人的应用中，为了安全，通常将相机相对于想要检测的物体的方向固定。例如，想象汽车检测道路标志的情况。在这种情况下，upright参数的使用将提高速度，并且非常有可能提高匹配性能。

最后的参数nOctaves和nOctaveLayers与cv::xfeatures2d::SIFT()的相应参数非常相似。nOctaves参数决定了搜索关键点的“倍增”量。在SURF的情况下，可以通过与9×9像素滤波器的卷积来计算可以找到的最小尺寸特征。octaves默认值是4，通常足以满足大多数应用的要求。然而，当使用分辨率非常高的成像器时，可能需要增加此数字。然而，将它降低到3，只能在速度上产生很小的提高，因为与较低的octaves相比，较高octaves的尺度搜索非常经济。[注37]

注36：对于图16-21中的汽车图像，我们根据经验调整了Hessian阈值，以给出SIFT在图16-11中发现的数量的相似数量的特征。为了进行比较，OpenCV默认值100，将分别为图16-21中左右图分别生成2017和2475个特征。

注37：这里值得注意的是，与SIFT实际上减少了每个octave的图像大小不同，SURF增加了与其进行卷积的内核的大小。此外，随着内核本身的扩大，内核的“相邻”评估之间的步骤也在增加。结果，由于内核具有固定的计算花销，而不考虑大小（记住积分图像技巧），评估较高octave的计算开销会迅速下降。

对于每个octave，将评估几个不同的内核。然而，与SIFT不同，内核不会分散来均匀划分octave。实际上，如果使用两个以上的octave图层，在连续octave使用的内核的大小之间将存在重叠。这并不意味着更大的值是没有用的，只是更大的octave层的效果不是完全直观的。nOctaveLayers的默认值为3，但有些研究发现把它增加4也是有用的（尽管增加了计算成本）。

descriptorSize()和descriptorType()方法返回描述符向量中的元素个数（通常为64个，扩展SURF特征为128个）和描述符向量的类型。目前后者总是CV_32F。

还有一些重载的方法SURF::detect()，SURF::compute()和SURF::detectAndCompute()。像之前一样，当你需要关键点和描述符时，建议使用后一种方法。参数与SIFT::detectAndCompute()的参数完全相同。

cv::xfeatures2d::SURF提供的附加功能 cv::xfeatures2d::SURF还提供了一系列方法来即时设置和检索算法的参数。确保在处理一组图像时不改变参数。找到最佳参数并继续使用它们，否则，你可能会获得无与伦比的描述符。

Star/CenSurE特征检测器和cv::xfeatures2d::StarDetector

Star特征（图16-25）最初是用于视觉测距，只是从图像数据测量摄像机自身的运动[Agarwal08]。在这种情况下，哈尔角点或FAST等特征是高度局部化的。相比之下，像SIFT这样的特征，依靠图像金字塔，当在金字塔上移动时，可能会在原始图像空间中局部化。不幸的是，像哈尔角点或FAST这样的特征因为缺乏一个尺度的空间搜索，不像SIFT那样具有尺度不变性。Star特征，也被称为中心环绕极值（或CenSurE）功能，试图解决提供哈尔角点或FAST特征的局部化水平的问题，同时还提供尺度不变性。Star/CenSurE特征中没有附加的唯一描述符，在首次介绍的论文中，作者使用的是“直立SURF”或U-SURF特征描述符的变体。

在概念上，Star的方法是计算某些特征在所有尺度上的所有变体，并选择尺度和位置的极值。同时，目标是使特征非常快速地进行计算（回想一下视觉测距在机器人和许多其他实时环境中的应用）。CenSurE功能通过两个阶段的过程来解决这些竞争的目标。第一阶段是和SIFT（和其他）使用的高斯算子（DoG）操作的一个非常快速的近似，并且提取了这个操作的局部极值。第二阶段尝试使用Harris度量的比例调整版本来剔除看起来像边缘（而不是角点）的东西。

要了解快速DoG近似是如何完成的，可以回想一下SURF如何计算相似对象的包围框逼近（图16-22）。然而，在这种情况下，DoG正在逼近一个特征，它看起来像在图像平面中旋转对称的两个相似大小的高斯的差。因此，特征非常简单。

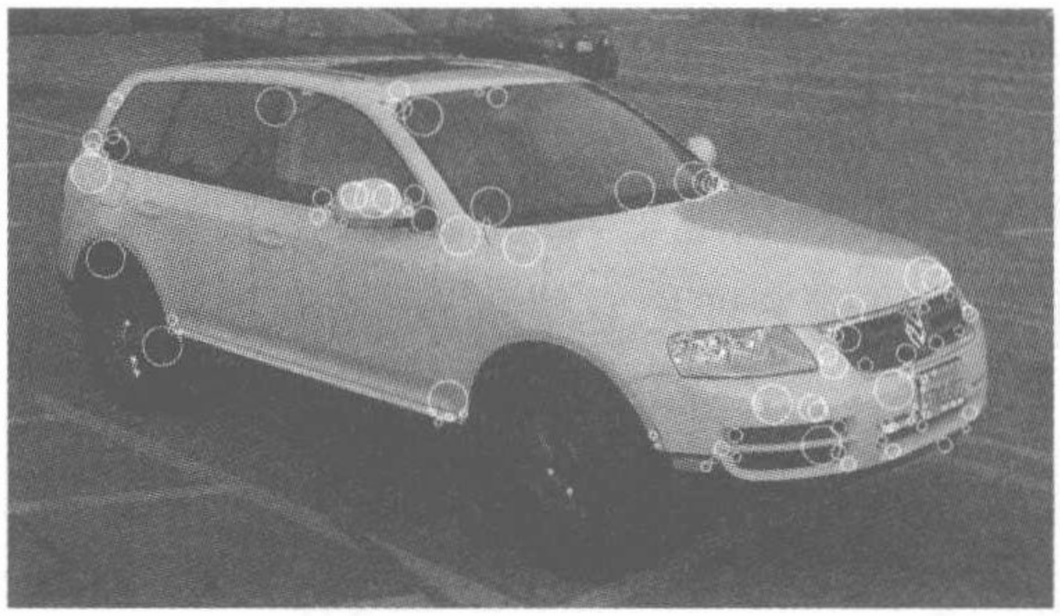
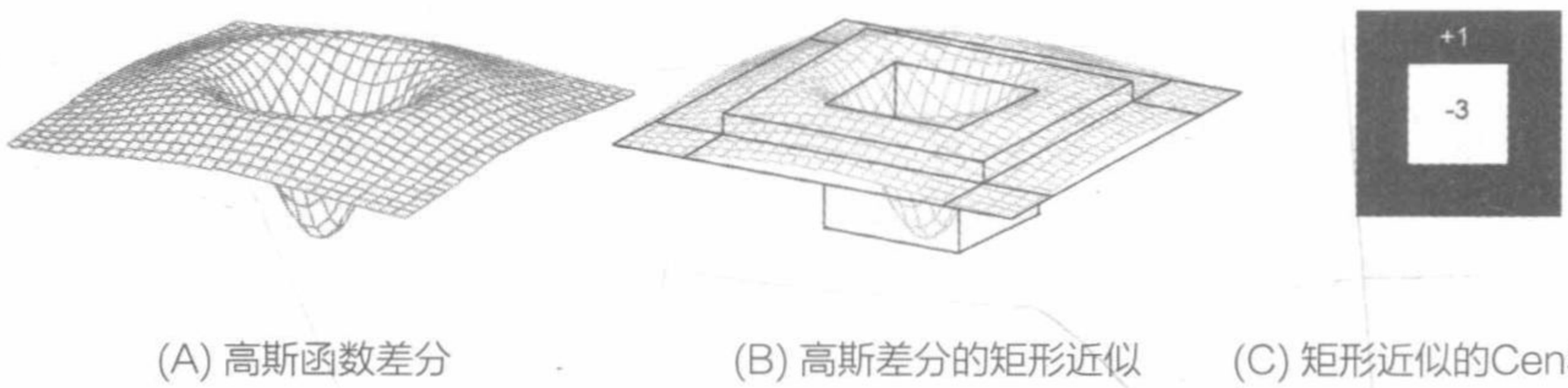




图16-25. 从两个稍微不同的方向计算相同车辆的Star特征。在默认参数下，左侧图像特征点数为137，右侧图像特征点数为336。在这两种情况下，汽车上的特征点数大致相同，第二张图像背景上的附加特征数较多。与其他方法相比，车上发现的特征很少，但你可以轻松看到两幅图像上的对应特征点，这表明特征非常稳定

所使用的近似是平方的（图16-26），因此可以构造成任何大小。它只有两个区域，可以通过积分图像计算。因此，特征的整体评估相当于依次进行外部平方的三次操作，三次内部的正方形，另外两次对它们进行放缩，最后的操作增加两个术语：总共九次操作。这个简单性就是为什么每个点都可以在许多尺度上进行测试。在实践中，这样可以构造的最小特征的边长为4，原则上每个大小都可以计算出来。当然，实际以指数而不是线性方式分布计算特征的大小是很自然的。该过程的这个阶段的结果是，对于图像中的每一点，都有一个特定大小的DoG内核（响应最大），和该响应的特定量纲。



(A) 高斯函数差分　　(B) 高斯差分的矩形近似　　(C) 矩形近似的CenSurE核实现

图16-26. CenSureE关键点查找器使用的DoG内核的包围框表示，对于特定尺寸S，特征的中心部分尺寸为S / 2

一旦DoG内核在任何地方近似，下一步是对该值进行阈值操作，然后过滤那些不是局部极值的点。我们通过与（x, y, scale） ——空间中的通常的$3 \times 3 \times 3$立方体的邻居进行比较，只保留那些在27个元素集合具有最高（或最低）值的那些实现这一步操作。

最后，由于这种特征仍然能够对边缘有较强的反应，所以Star算法可以计算出一个按比例调整的Harris度量。按比例适应的哈尔度量是根据与前面讨论的哈尔·史托西斯角（Harris-Shi-Tomasi）矩阵非常相似的矩阵进行计算的，只有两处重要的例外。第一个是对自相关矩阵的各个元素进行求和的窗口的大小与特征的比例成比例。第二个是自

相关矩阵是根据对判别特征的最大响应而不是图像强度构成的。然后实际执行是使用Harris进行测试，Harris会将该矩阵的行列式与乘以灵敏度常数的平方迹进行比较。

在OpenCV实现中，还有一个第二测试，其类似于描述符合比例尺度的Harris度量，除了这个测试还会根据与窗口中每个点的响应相关联的大小值构建自相关矩阵。这个方法被称为二进制尺度适应哈尔度量。它被称为“二进制化”，因为对于窗口中的每个点，基于相对于其邻居在该点处的最大响应的大小的变化率来决定分配值为1，0还是-1。回想一下，原来的哈尔度量使用图像强度的变化率，而尺度适应的哈尔度量使用了对DoG操作的响应变化率；二值化度量使用最大DoG算子的大小的变化率。这种二值化测试是量化特定点是尺度空间极值的程度的一种方式。

关键点查找器　如前所述，没有与Star算法相关联的特定描述符提取器。检测器cv::StarDetector直接从cv::Feature2D基类派生。以下是略微缩写的cv::StarDetector对象的定义：

```
// Constructor for the Star detector object:
//
class cv::xfeatures2d::StarDetector : public cv::Feature2D {

public:

  static Ptr<StarDetector> create(
    int maxSize                = 45,  // Largest feature considered
    int responseThreshold      = 30,  // Minimum wavelet response
    int lineThresholdProjected = 10,  // Threshold on Harris measure
    int lineThresholdBinarized = 8,   // Threshold on binarized Harris
    int suppressNonmaxSize     = 5    // Keep only best features
                                      //  in this size space
  );

  ...
);
```

Star检测器构造函数有五个参数。第一个是要搜索的特征的最大尺寸。maxSize参数只能设置为一个有限的值列表中的一个：4，6，8，11，12，16，22，23，42，45，46，64，90或128。对于任何你可能选择的值，此列表中的所有比你所选值小的值都将被检查。

responseThreshold参数指示应用于与CenSurE内核（图16-26C）卷积的阈值，以便找到候选关键点。对于最小值以上的所有尺度，内核将被归一化，使所有较大尺度的阈值等于最小内核上的给定值（即大小为4）。

接下来的两个参数lineThresholdProjected和lineThresholdBinarized是与前面描述的自适应尺度哈尔角点检测相关联的阈值。投影线阈值实际上是Harris测试对响应值的灵敏度常数的倒数；提高lineThresholdProjected的值将拒绝更多的特征行。

lineThresholdBinarized值做的非常类似，除了它是二进制自适应尺度哈尔角点检测的灵敏度常数。第二个参数强制要求CenSurE特征是一个尺度空间极值。两次比较之后，候选关键点必须相对于两个标准都成功才能被接受。

最后一个参数SupressNonmaxSize，如果Star功能不是该距离内最强的功能，则会设置Star特征被拒绝的区域。

BRIEF描述符提取器和cv::BriefDescriptorExtractor

BRIEF，即二进制鲁棒独立基本特征，是一种相对较新的算法，用于将一种新颖的特征分配给关键点（图16-27）。Calonder等人提出了BRIEF特征。并且由于这个原因也经常被称为“Calonder特征”[Calonder10]。BRIEF不找关键点；相反，它用于生成可通过任何其他可用的特征检测器算法定位的关键点的描述符。[注38]

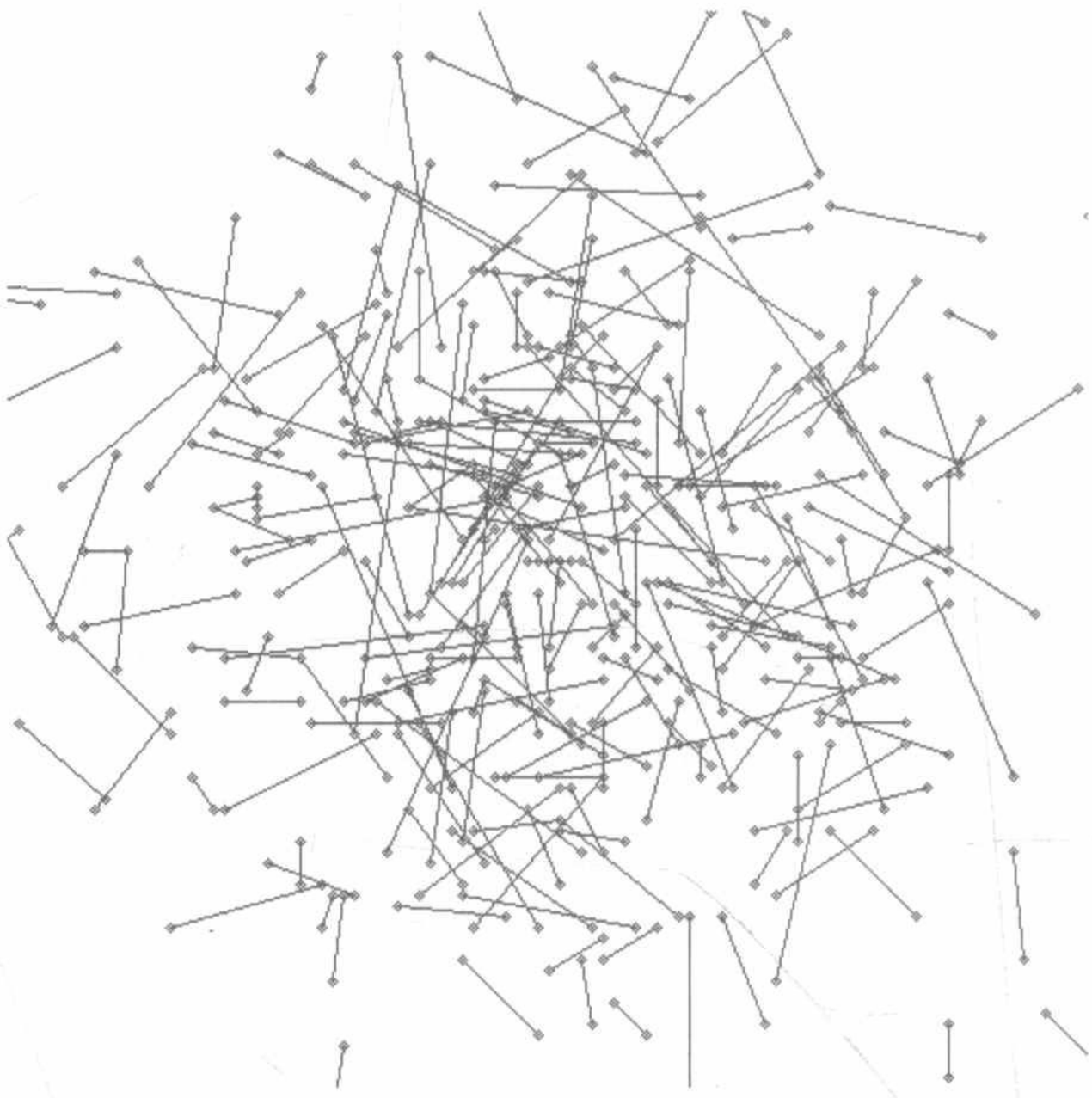


图16-27. 像素到像素测试的可视化，它们共同包含一个BRIEF描述符，每条线连接通过测试比较的一对像素

BRIEF描述符背后的基本概念是将特征描述为一系列测试，每个测试简单地将特征区域中的单个像素与一些其他单个像素进行比较，基于哪个部分更亮（图16-27）产生简单的二进制结果（即0或1）。BRIEF描述符仅仅是将这样的测试排列成二进制字符串的结果。为了避免描述符对噪声的过度敏感，BRIEF描述符首先通过与高斯内核卷积来平滑

注38：在介绍它们的论文中，BRIEF描述符用于通过U-SURF发现的特征。

图像。因为描述符是二进制字符串，它们不仅可以快速计算并高效存储，而且还可以非常有效地进行比较。[注39]

有许多方法可以从BRIEF描述符中生成将被匹配的实际对。最好的方法之一是简单地随机生成所有的对，首先从特征中心附近的高斯分布绘制一个点，然后通过从第一点（具有标准偏差的一半）的高斯分布进行绘制来计算第二点。将点绘制的区域（特征的总面积）称为像斑大小，而绘制点的分布的标准偏差称为内核大小。在图16-27的情况下，核心大小与像斑大小的比率约为1:5。现在OpenCV固定了这些大小，但是原则上它们是算法的可调参数。总体上生成的这种测试的数量通常为128,256或512，但遵循特征的原始创建者的风格，传统的是根据描述符中的字节数（即，分别为16，32或64字节）来指代这个大小。

特征提取器 如前所述，BRIEF算法专门用于提取特征描述符，因此相关方法直接从cv::Feature2D基类派生，只实现描述符提取部分。类定义的相关部分如下：

```
class cv::xfeatures2d::BriefDescriptorExtractor : public cv::Feature2D {

public:

  static Ptr<BriefDescriptorExtractor> create(
    int bytes            = 32,        // can be equal 16, 32 or 64 bytes
    bool use_orientation = false      // true if point pairs are "rotated"
                                      // according to keypoint orientation
  );


  virtual int descriptorSize() const;  // number of bytes for features
  virtual int descriptorType() const;  // Always returns CV_8UC1
};
```

这次，BRIEF描述符提取器的唯一两个用户可以配置的参数是包括特征的信息的字节数（等于测试的总数除以8）和使用方向标志，其类似于SURF算法的直立参数。同样的考虑也适用于这一点——当特征不太可能旋转时，例如，识别路标或缝合图像——use_orientation应该设置为false。否则，最好将其设置为true。

为了从图像和一组关键点位置实际计算描述符，cv::xfeatures2d::BriefDescriptorExtractor使用cv::Feature2D基类中定义的compute()接口。

BRISK算法

引入了BRIEF描述符不久，就出现了几种新的技术，使用类似的点对比概念作为产生

注39：许多现代处理器都有单周期指令，对256位字执行异或（例如Intel SSE4™指令集）操作。

可以快速比较的紧凑描述符的手段。Leutenegger等人介绍的BRISK[注40]描述符（参见图16-28）试图以两种不同的方式改进Brief（Leutenegger11）。首先，BRISK引入了自己的一个特征检测器（回想一下，Brief只是一种计算描述符的方法）。其次，BRISK的特征本身虽然与BRIEF原则相似，却尝试以提高整体功能的鲁棒性的方式进行二值比较。

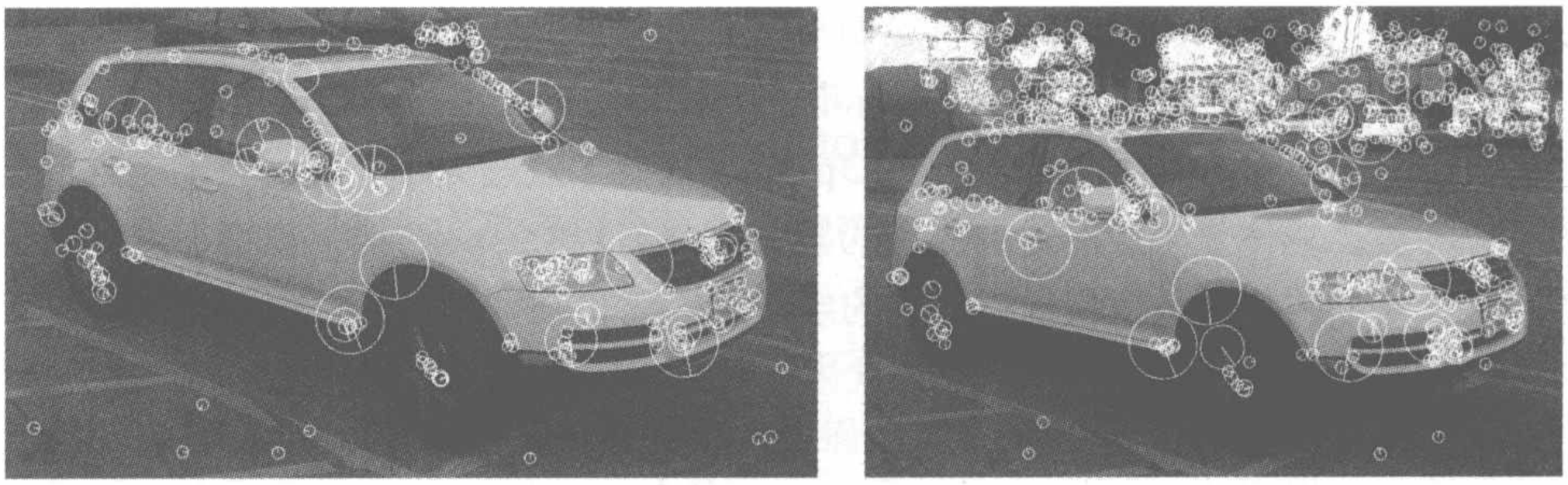


图16-28. 将BRISK特征检测器用于我们的两个参考图。左图包含232个特征点，右图包含734个特征点。右图中更复杂的背景造成了大部分新特征，但车辆的特征在数量和位置都相对稳定

BRISK的特征检测器部分基本上是基于与FAST相似的*AGAST*[注41]检测器，其改进是尝试识别特征的尺度大小以及方向[注42]。BRISK通过首先创建具有固定数量的尺度（两个关于大小的因子）的尺度空间金字塔，然后根据尺度计算固定数量的*intra-octave*来识别尺度。BRISK特征检测器的第一步是应用FAST（或实际上是AGAST）来查找所有这些尺度的特征。一旦完成，使用非最大值抑制；也就是说，其分数（在我们的FAST讨论中称为ρ_0）不是其所有邻居中最大的特征被去除；只留下“最大”特征。

一旦以这种方式找到功能列表，BRISK就会继续计算图像中相应位置的AGAST分数，并立即更大并立即更小（图16-29）。在这一点使用简单的二次曲线拟合AGAST分数（作为尺度的函数），并且将该二次方的最大点作为BRISK特征的真实尺度。以这种方式提取连续值，BRISK特征也不会被强制与在金字塔中计算的一个离散图像的相关联。在像素坐标中应用类似的内插方法来为特征分配亚像素位置。

注40：“BRISK”并不代表任何东西，它不是缩写。其实就是与“BRIEF”相似的单词而已。

注41：称为“AGAST”的特征检测器（基于加速段测试的自适应和通用角点检测）[Mair10]是对FAST和BRISK前导的改进。这里提到的算法是完整的，但在OpenCV库中没有单独实现。

注42：在原始实现中，每个尺度只有一个intra-octave，因此产生仅*N*个尺度，*N*-1个intra-octave共有2*N*-1个图像。

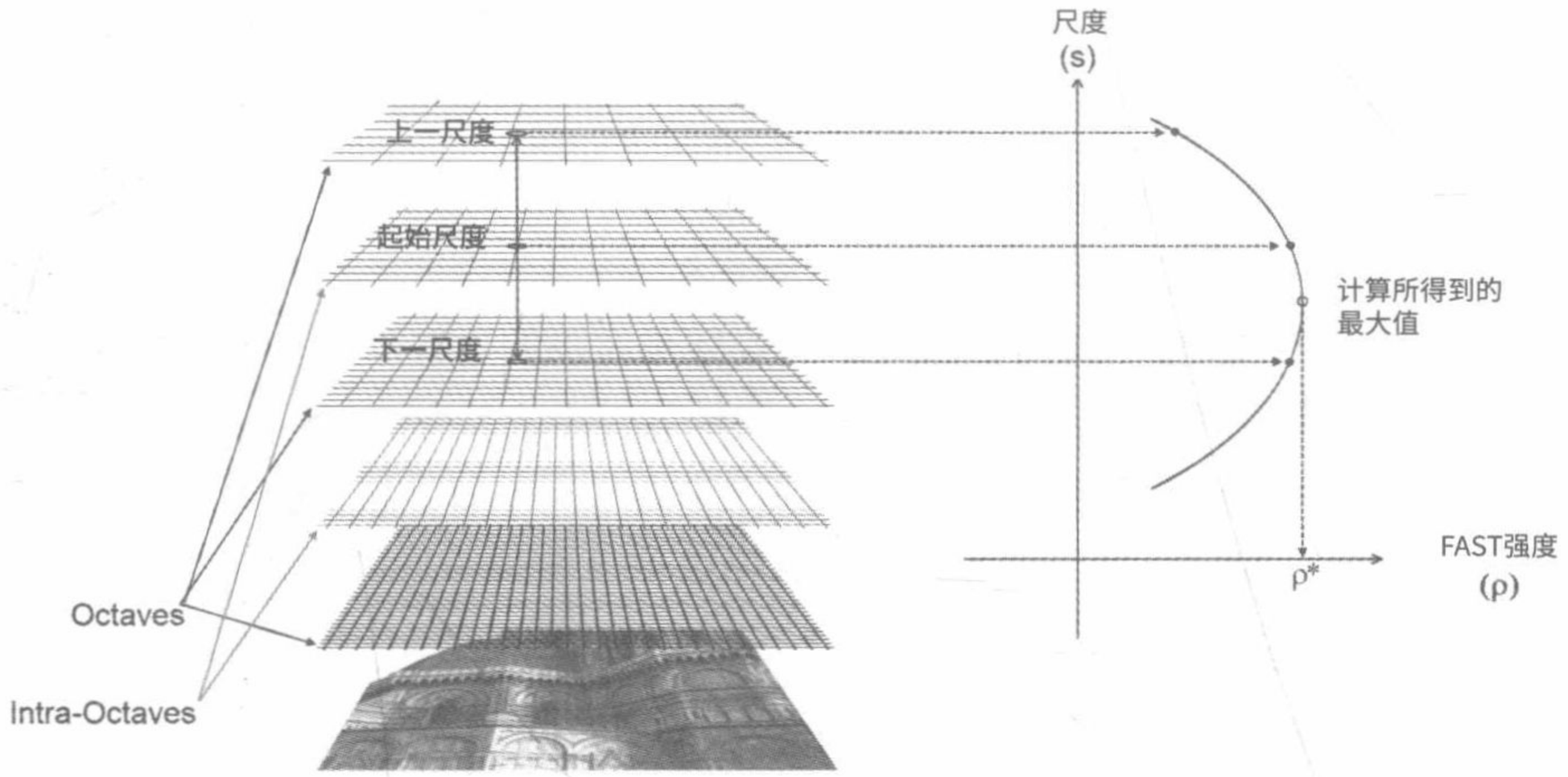


图16-29. BRISK使用其间具有“intra-octaves”的几个octaves构建尺度空间。当FAST特征在一个尺度上被发现时，它的FAST强度按照这个尺度计算，并在紧邻上方和下方（左）的尺度上计算。这些强度用于拟合二次曲线，并从该曲线推算最大分数的值（右）

除了尺度，BRISK特征也有方向。要了解其中的原理，我们需要首先了解BRISK使用的采样模式与Brief使用的随机采样模式的不同之处。BRISK描述符由围绕中心点的一系列环构成。每个环具有分配给它的一些的采样点K_i，并且每个采样点被分配直径等于特定圆C_i的圆周除以K_i的圆形区域（图16-30）。该区域对应于特定半径（$\sigma_i = C_i/2K_i$）的高斯卷积的图像，并在指定点进行采样。

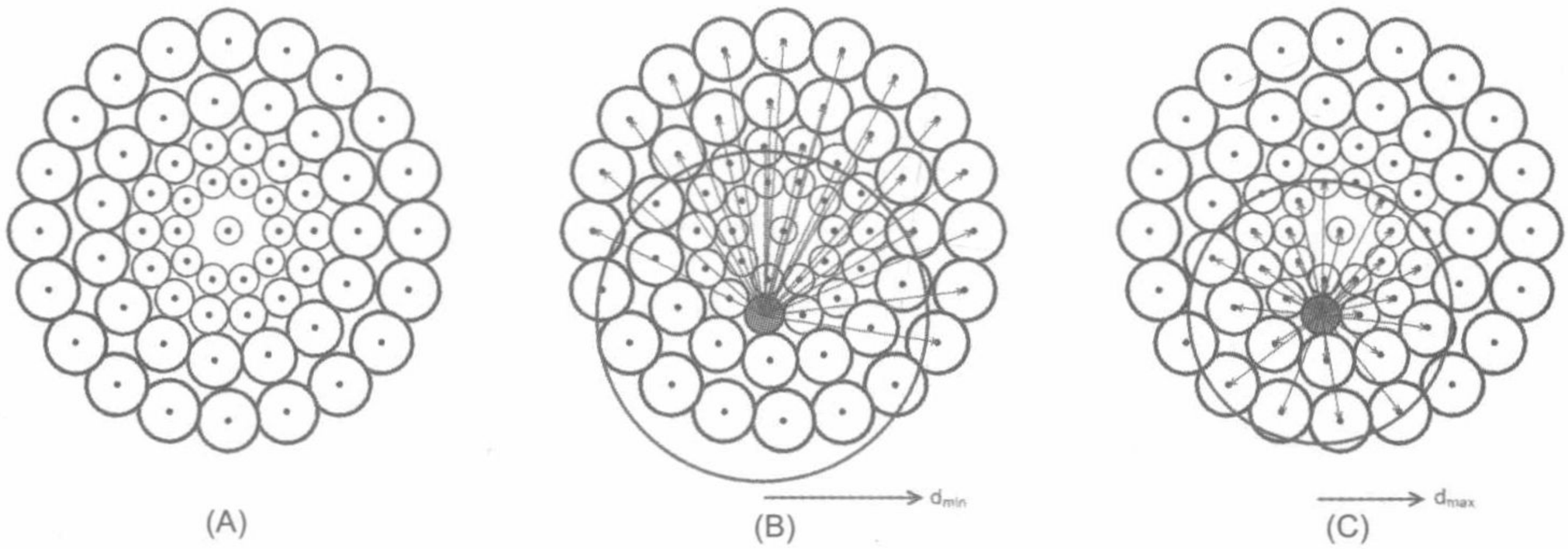


图16-30. BRISK描述符中的测试点用小的实心点在图中标识。对每个测试点有影响的区域显示为围绕每个点的圆。注意，随着测试点从描述符的中心移出，相关区域的大小在增加。左图仅显示点及其相关联的区域(A)。中间的图像显示与一个特定点(B)相关联的所有长程配对。右图显示与同一点相关联的所有短程配对(C)

在所有圆的配对之间计算构成逐位描述符（类似于BRIEF）的亮度比较。特别地，这些配对被分为两个子集，称为短距离和长距离配对。短距离配对是所有点之间的配对，它

们之间的距离小于某一特定距离$d_{\max}$，而远距离配对则是所有点之间的距离大于特定距离d_{min}之间的配对。短距离配对形成描述符，而长距离配对用于计算主导方向。

要了解这个主导方向是如何构建的，首先要注意，BRISK描述符，像BRIEF描述符一样，计算点对之间的强度差异。然而，BRISK描述符继续将这些差归一化为点之间的距离，从而实现了局部梯度的效果。通过对所有长距离对中的这些局部梯度求和，我们计算可用于定向描述符的方向。然后相对于该方向计算短距离特征，使得通过对这些短距离梯度进行阈值化而形成的描述符方向有效独立。通过调整每个圆的点数和d_{max}的值，我们可以将描述符设置成长度。按照惯例，选择这些值使得描述符长为512位，与（典型的）BRIEF描述符相同。

关键点查找器和特征提取器 cv::BRISK对象继承自cv::Feature2D接口，从而提供特征查找功能和描述符提取功能。这是cv::BRISK对象的（有缩略）定义：

```
class cv::BRISK : public cv::Feature2D {

public:
  static Ptr<BRISK> create(
    int                  thresh       = 30,   // Threshold passed to FAST
    int                  octaves      = 3,    // N doublings in pyramid
    float               patternScale = 1.0f  // Rescale default pattern
  );

  int descriptorSize() const;                  // descriptor size
  int descriptorType() const;                  // descriptor type


   static Ptr<BRISK> create(                   // Compute BRISK features
    const vector<float>& radiusList,          // Radii of sample circles
    const vector<int>&   numberList,          // Sample points per circle
    float                dMax        = 5.85f, // Max distance for short pairs
    float                dMin        = 8.2f,  // Min distance for long pairs
    const vector<int>&   indexChange = std::vector<int>() // Unused
  );


};
```

cv::BRISK::create()构造函数方法接受三个参数：AGAST阈值、octaves的数量以及模式的整体比例因子。如果你使用此构造函数，则抽样点的位置将从库中的固定查找表中获取。阈值参数thresh设置AGAST特征检测器将使用的阈值[注43]。octaves用于设置所有octaves的数量。请记住，如果将其设置为某个数字N，则计算的总计数将为$2N-1$（包括intra-octaves）。最终的patternScale参数将整体缩放因子应用于内置模式。

注43：在BRISK代码的内部，BRISK调用cv::FAST，并将该阈值精确传递给FAST算法。

重载的cv::BRISK::detectAndCompute()方法实现了从cv::Feature2D接口继承的常见的的特征检测和描述符提取功能。

cv::BRISK提供的附加函数 除了上述方法之外，你还可以使用从cv::Feature2D继承的任何函数，cv::BRISK有一个额外的扩展构造函数：cv::BRISK::create()。如果你不想依赖OpenCV库提供的采样点的内置模式，就使用此函数。如果你想建立自己的模式，必须以一个STL风格的radiusList参数向量的形式提供圆圈的半径列表。同样，你必须向numberList提供一个STL样式的整数（与radiusList长度相同）列表，该列表指示每个半径使用的采样点数。然后，你可以（可选）指定dMax和dMin，短距离配对的最大距离和长距离配对的最小距离（上一节的d_{Max}和d_{Min}）。最后一个参数indexChange目前没有任何效果，应该最好省略。

ORB特征检测器和cv::ORB

对于许多应用，特征检测器的速度不仅有用，而且很重要。对于在视频数据（例如增强现实或机器人应用程序）上实时运行的任务尤其如此。因此，SURF功能的开发是为了提供与SIFT类似的功能，但速度更快。类似地，创建了ORB功能[Rublee11]，其目标是为SIFT或SURF提供更高速的替代品（参见图16-31）。ORB功能使用非常接近于FAST（我们在本章前面看到的）的关键点检测器，但是使用了基本上不同的描述符，主要基于BRIEF。ORB通过方向计算增强了BRIEF描述符，该方向计算本质上使得ORB特征具有SIFT和SURF具有的的相同类型的旋转不变性[注44]。

ORB算法的第一阶段是使用FAST来定位候选的一组特征。FAST特征定位计算花销很低，但有几个缺点。一个是它们倾向于对边缘和角点做出反应。为了克服这个问题，ORB算法计算了所在FAST点的哈尔角点测量。你可能会想到，这个措施是对由特征[注45]位置附近的像素形成的自相关矩阵的特征值的约束。使用这个过程，我们然后构建一个图像金字塔，以便可以进行尺度空间搜索。由于哈尔角点测量不仅提供了FAST功能的质量测试，还提供了更好的特征质量指标，还可以用于选择图像中的“最佳”特征。当需要某些特定数量的特征时（在实际应用中通常是这种情况），特征由哈尔角点测量来排序；那些最好的会保留直到找到所需的数字。

注44：ORB中使用的特征也被称为（由其创建者以及其他人）rBRIEF（或旋转感知*BRIEF*）特征，并且与我们刚刚遇到的BRIEF特征密切相关。事实上，这是ORB的起源，它来源于“方向FAST和旋转感知BRIEF”。

注45：你们会回想起我们之前的讨论，Harris原来提出了一个稍稍不同于后面由Shi和Tomasi提出的约束。cv::GoodFeaturesToTrackDetector()算法（默认情况下）使用后者，而ORB使用前者。

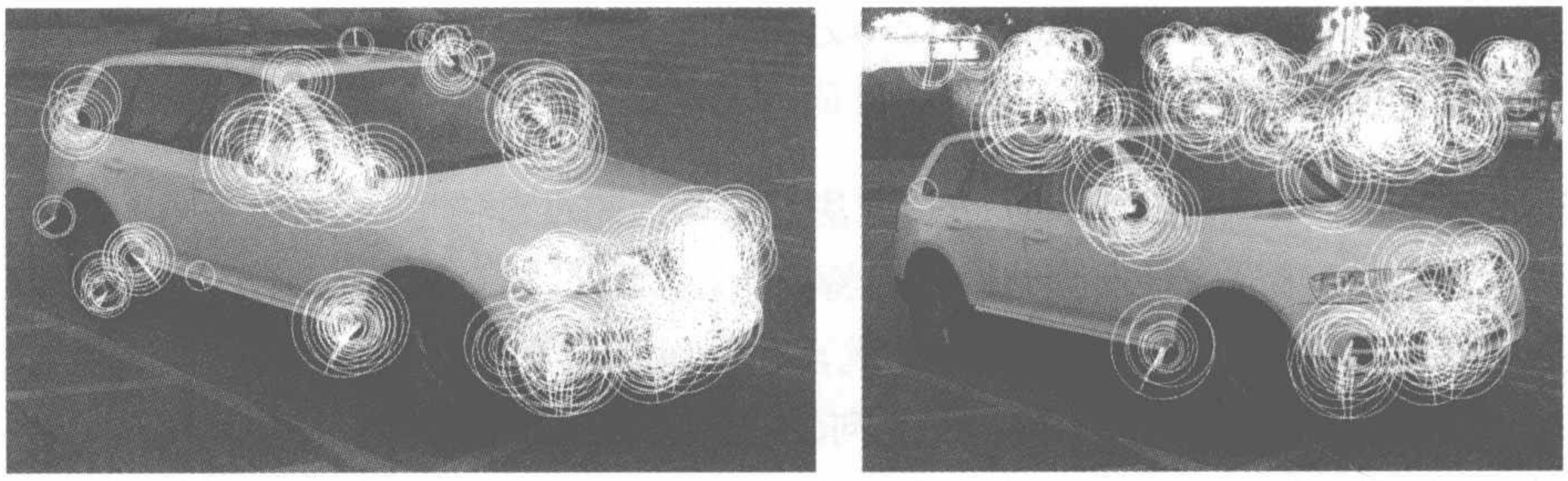
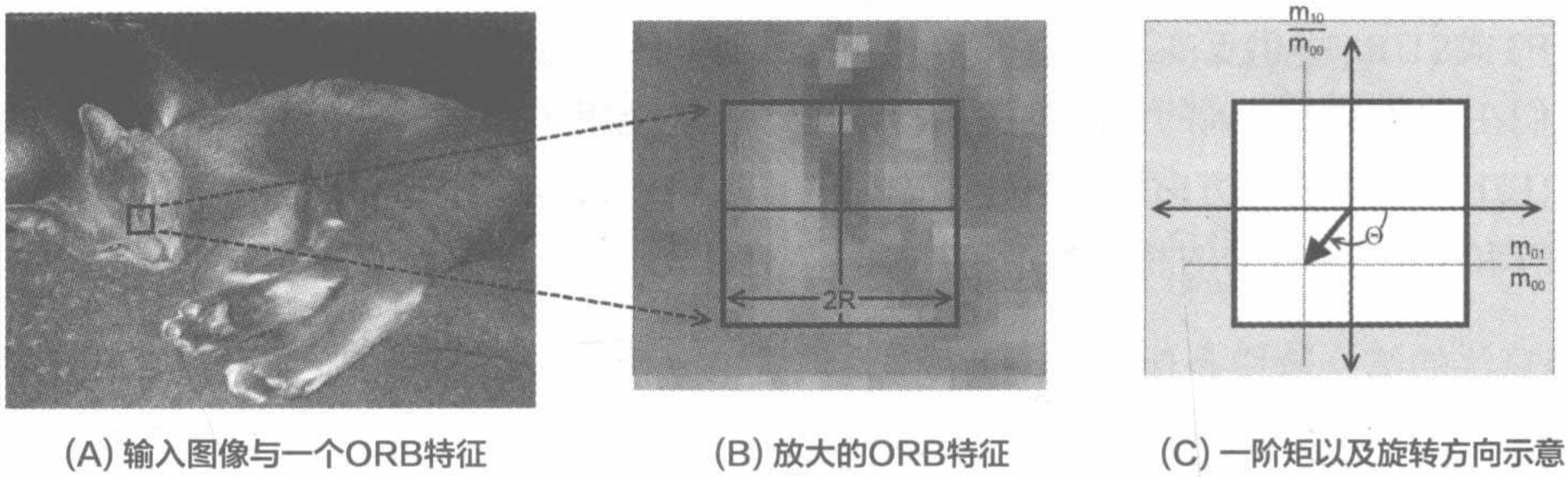


图16-31. 同一车辆的两张图像每个产生500个ORB特征。ORB的一个有趣的特征在这里是可见的，即如果图像中的一个角点很大，在同一个角点就会出现许多不同大小的ORB特征

ORB算法相对于FAST（或Harris角点）的一个重要贡献就是引入定位关键点的方向。方向是分两步进行的。首先，计算特征周围的框内强度分布的一阶矩（x和y）。该框的边长等于发现特征的大小的两倍（与尺度的给定半径的圆的近似值）。图16-32显示归一化的x和y梯度（它们被除以平均值），给出梯度方向相对于特征中心的方向。因此，此ORB功能也称为*oFAST*特征或*oriented-FAST*特征。



(A) 输入图像与一个ORB特征　(B) 放大的ORB特征　(C) 一阶矩以及旋转方向示意

图16-32. 我们通过分析由发现FAST特征的尺度给定大小的图像的一阶矩（平均强度）来计算ORB特征的方向；特征的方向由从特征的中心到由那些矩指示的点的方向向量给出

一旦定位了特征并且已经分配了方向，就可以计算相对于该方向的特征向量。所得到的特征可以以旋转不变的方式使用。[注46]ORB使用的特征描述符，如前所述，是基于BRIEF算法中使用的描述符，但引入方向信息是ORB特征相对于其前身BRIEF的重要区别。

ORB和BRIEF特征之间的第二个显著区别是，BRIEF的作者通过分析大量数据图像并寻找具有特定属性的测试对，实际上产生了旋转感知BRIEF描述符：高方差，平均结果接近0.5，与其他测试对的最小相关性。[注47]然而，为了进行这种分析，它们将每个描述符

注46：从我们之前对“BRIEF”的讨论来看，“BRIEF”描述符使用了随机的“测试”数组。但是，这个数组没有能力与特征“对齐”（例如，像SIFT那样）。

注47：精明的读者会认识到前两个属性实际上是二值变量分配的相同属性。

转换为相对于特征方向位于测试点的表示。该分析在其作者的ORB描述符的构建中进行了一次，并且在下文中被描述为“内置”。它们使用的数据集是包含多种图像的著名的图像数据集。[注48]

关键点查找器和特征提取器　与SIFT和SURF一样，ORB算法通过cv::Feature2D接口在OpenCV中实现。这是实现ORB算法的cv::ORB对象的（略微缩写）定义。

```
class ORB : public Feature2D {
public:
// the size of the signature in bytes
enum { kBytes = 32, HARRIS_SCORE = 0, FAST_SCORE = 1 };

static Ptr<ORB> create(
  int  nfeatures     = 500,    // Maximum features to compute
  float scaleFactor   = 1.2f,   // Pyramid ratio (greater than 1.0)
  int  nlevels       = 8,      // Number of pyramid levels to use
  int  edgeThreshold = 31,     // Size of no-search border
  int  firstLevel    = 0,      // Always '0'
  int  WTA_K         = 2,      // Pts in each comparison: 2, 3, or 4
  int  scoreType     = 0,      // Either HARRIS_SCORE or FAST_SCORE
  int  patchSize     = 31,     // Size of patch for each descriptor
  int  fastThreshold = 20      // Threshold for FAST detector
);

int descriptorSize() const;     // descriptor size (bytes), always 32
  int descriptorType() const;     // descriptor type, always CV_8U
};
```

ORB构造函数支持一个有点吓人的参数列表。大多数参数可以放心保留为默认值，你将获得令人满意的结果。第一个参数nfeatures，可能是你最有可能改变的，它只是确定你想要cv::ORB一次查找的关键点数。

因为ORB检测器使用图像金字塔，所以你必须告诉检测器金字塔中每层之间的比例因子以及金字塔有多少级别。你不想使用由cv::buildPyramid()创建的类型的粗糙二因子类型，因为两者之间的特征会丢失太多。缩放因子的默认值仅为1.2。控制比例因子和金字塔数量的变量分别是scaleFactor和nlevels。

由于关键点具有特定的像素大小，因此需要避免图像的边界。该距离由edgeThreshold设置。用于单个特征的像斑的大小也可以使用patchSize参数设置。你将注意到它们的默认值为31。如果更改了patchSize，则应确保edgeThreshold保持等于或大于patchSize。

注48：该数据集称为PASCAL-2006数据集，在互联网上公开发布。它是广泛应用于计算机视觉研究的知名基准数据集，在计算机视觉论文中得到广泛的引用。但是，任何特定类型的数据集的性能是否可能受到ORB特征选择在PASCAL-2006（相当通用）数据集上进行训练而不是在专门类型的数据集上训练的影响，这是一个悬而未决的问题。

firstLevel参数允许你设置比例金字塔，使得其比例为1的级别不一定是第一级。实际上，将firstLevel设置为非零值意味着金字塔中的某些图像实际上将大于输入图像。这对ORB特征是有意义的，因为它们的描述符本质上依赖于图像的平滑版本。然而，在大多数情况下，使firstLevel非常高将产生主要由噪声驱动的小尺度的特征。

参数WTA_K控制元组大小，从而进一步精确地控制二进制测试中描述符的构造方式。WTA_K=2的情况是我们之前描述的场景，其中每个描述符字节的每个位是测试点对之间单独进行比较。这些测试点来自预先列出的列表。在WTA_K=3的情况下，通过从该列表中的三个测试点的集合之间的三向比较，一次将两个描述符的位设置为2。类似地，对于WTA_K=4，通过四个测试点的集合之间的四路比较，一次将两个描述符的位设置为2。



然而，重要的是要了解，本节开头部分描述的测试点的预生产列表对元组大小为2，特征尺寸为31而言非常有意义。如果使用任何其他大小的特征，测试点将随机生成（而不是预先计算的列表）。如果你使用2以外的元组大小，则测试点也将随机排列成正确长度的元组。（所以如果你使用31作为特征大小，但元组大小不是2，你将使用预先计算的测试点列表，但是以随机的方式，这并不比仅仅使用随机测试点更好。）

cv::ORB构造函数的最后一个参数是得分类型。scoreType参数可以设置为两个值之一：cv::ORB::HARRIS_SCORE或cv::ORB::FAST_SCORE。前一种情况在本节开头进行了描述，其中发现了大量的特征，都使用哈尔度量重新计算了它们的分数，并且只保留了最佳的分数。不幸的是，这导致了两种计算成本的方式。一个是哈尔指标需要时间来计算，另一个是需要首先计算更多的特征度量。[注49]另一种方法是使用与FAST相关联的度量。特征不是很好，但是可以稍微提高运行速度。

cv::ORB提供的附加功能　cv::ORB还包括一组get*/set*方法，可以在构建类实例之后检索或修改各种算法参数。

FREAK描述符提取器和cv::xfeatures2d::FREAK

与BRIEF描述符一样，FREAK算法（图16-33）只计算一个描述符，并且没有自然关联的关键点检测器。FREAK描述符最初是作为Brief，BRISK和ORB的改进引入的，它是一个生物启发式的描述符，其功能非常类似于BRIEF，主要在于它计算二进制比较领域的方式[Alahi12]。第二个更微妙的区别是，FREAK不是对均匀平滑图像周围的像素进行

注49：像实现的那样，当你需要一些特定数量的特征时，使用FAST度量标准将保留许多特征的两倍，然后对于那些特征，将计算哈尔度量，并保留该度量最好的一半。

点比较，而是使用对应于不同大小的积分区域的点进行比较，距离描述符中心更远的点被分配更大的区域。这样FREAK可以捕获人类视觉系统的一个基本特征，从而得出它的名称Fast Retinal Keypoint。

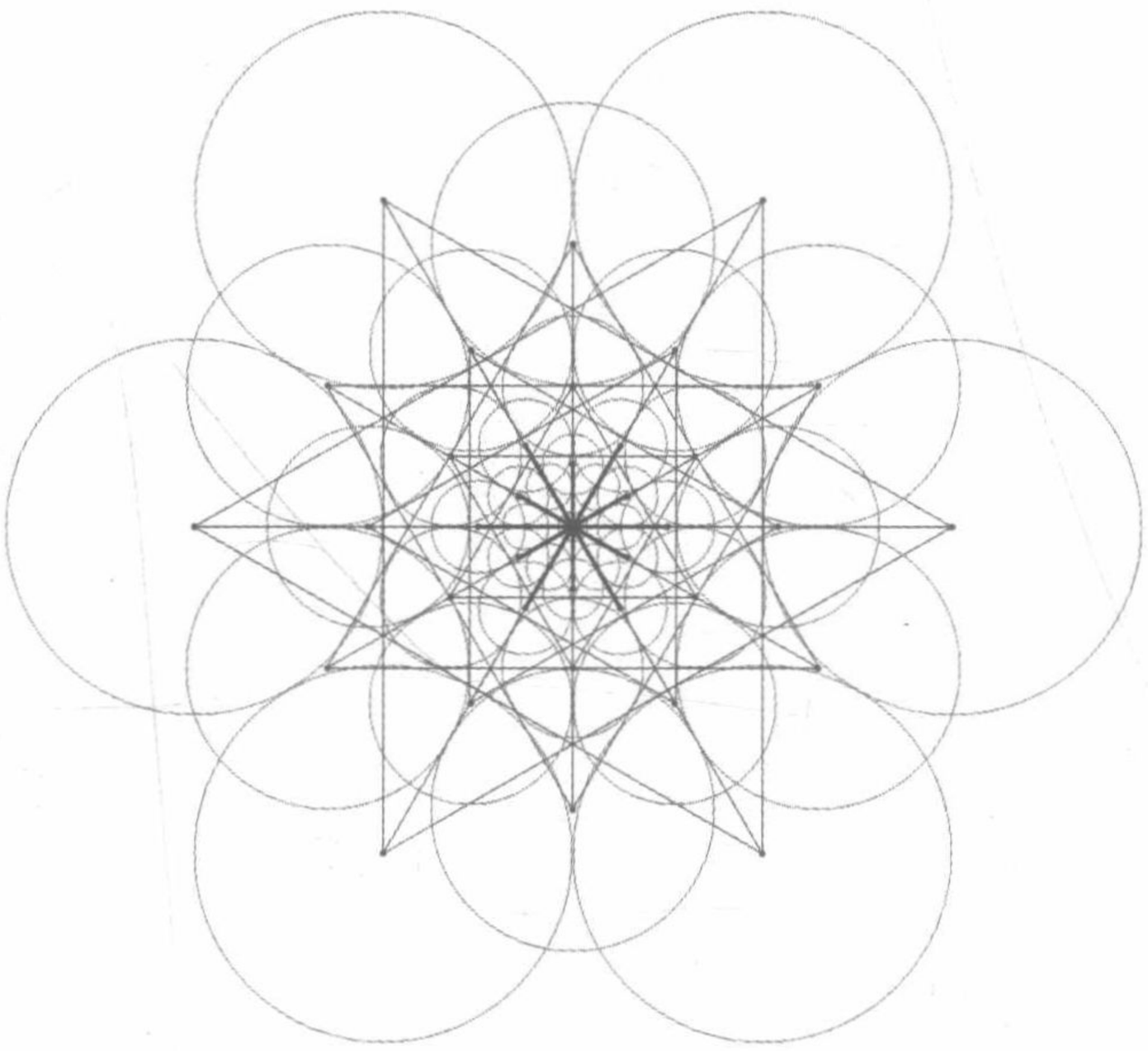


图16-33. 表示FREAK描述符的操作的图。直线表示顶点之间的可能比较。圆圈表示与每个顶点相关联的“接收场”。请注意，随着顶点距离描述符的中心越远，接收场的大小会越大。（图片使用经[Alahi12]许可）

要了解FREAK特征，先去重新浏览BRIEF特征是有用的，但从稍后介绍的角度来看，这是一个有点不同的观点。回想一下，BRIEF特征涉及在特征的紧邻附近的大量的单个像素强度对之间的按位比较。为了提高噪声的鲁棒性，BRIEF算法在计算这些像素强度之前先对图像应用高斯滤波器。

或者，我们可以将比较中使用的每个像素视为对应于输出像素的紧邻附近的输入图像中的像素的高斯加权和。在数学上，这两个是完全等价的，但直观上来看，这个新图片自然介绍了比较像素的接受领域的想法，并且考虑到视网膜的神经节细胞与该神经节细胞单个光感受器的集合之间的关系响应。

然而，BRIEF特征和人眼的接受领域之间的一个显著差异在于，尽管BRIEF中的接受区具有均匀的大小（回想一下高斯卷积），但人视网膜的软管随着越来越远离视网膜的中心越来越大。受此生物学观察的启发，FREAK描述符采用一系列接收场，其尺寸随着离特征中心的距离增大（图 16-33中的圆圈）而增加。在标准构造中有43个这样的视场。

与ORB描述符一样， FREAK描述符的创建者按照效果递减的顺序，继续使用学习技术来组织这些可接受领域之间可能的比较。以这种方式，比较具有最大歧视力（比较大的特征的训练集）的比较可以优先于具有相对较差的歧视力的那些。一旦这个顺序完成，只有当它们与较高的实用对比较显示出强烈的去相关时，才保留该对比较。在应用中，给出几十个不同大小的字段和数千个可能的比较，只有最有用的512个会被保留。

在这512个比较中，FREAK描述符将它们组织成四组128个比较。根据经验，观察每个组似乎连续在中心附近使用更多的小型接受场，但最开始的区别仍实在比较大的接受场之间。结果是，可以首先使用这些大的场值进行比较，然后，如果发现足够的相似性，则进行到更细的区域以改进匹配。每组128个比较仅对应于单个16字节值的异或运算（和位求和）。这个数字是重要的，因为许多现代处理器可以在单个周期中进行这样的比较。因为可以通过这个单一的比较来拒绝绝大多数可能的匹配，所以FREAK描述符是非常有效的。

特征提取器 FREAK特征在OpenCV中的类实现`cv::xfeatures2d::FREAK`，是从`cv::Features2D`接口继承并实现描述符提取部分。

```
class FREAK : public Features2D {

public:

  static Ptr<FREAK> create(
    bool orientationNormalized = true,  // enable orientation normalization
    bool scaleNormalized       = true,  // enable scale normalization
    float patternScale         = 22.0f, // scaling of the description pattern
    int nOctaves               = 4,     // octaves covered by detected keypoints
    const vector<int>& selectedPairs = vector<int>() // user selected pairs
  );

  virtual int descriptorSize() const;   // returns the descriptor length in bytes
  virtual int descriptorType() const;   // returns the descriptor type

  ...
};
```

FREAK构造函数有许多参数，通常情况下，大多数时候都可以放心地保留默认参数。前两个表示对已发布算法的轻微扩展。正如最初提出的那样，FREAK特征具有固定的方向和尺度。通过将`orientationNormalized`设置为`true`，可以要求OpenCV FREAK对象构建方向不变的描述符。该选项可以减少关键点的辨识度，但是作为交换，可以获得方向不变性。[注50]

要了解参数`scaleNormalized`的作用，首先必须回顾（大多数）关键点都一个大小，该

注50：请注意，方向SURF特征与“直立”（U-SURF）特征之间可以进行类似的权衡。

大小由首选位置的关键点检测器设置。如果scaleNormalized为真，则在计算特征向量之前，特征周围的图像像斑将首先根据该关键点的大小重新缩放。

patternScale参数用于均匀重新调整FREAK接受场模式。你不太可能想改变这个。与patternScale参数密切相关的是参数nOctaves。当创建FREAK描述符对象时，将生成一个查找表，其中包含所有必需信息，以便在一定范围内计算FREAK描述符。参数patternScale和nOctaves的组合允许你控制此查找表的范围（根据模式的大小）。生成的精确刻度由以下公式给出：

$$scale_i = patternScale * 2^{i*\left(\frac{nOctaves}{nbScales}\right)},\ for\ i \in \{0, 1, \ldots, nbScales\}$$

这里nbScales是目前的实现中的尺度数量，等于64。注意，在每种情况下，生成的尺度数量都是相同的，但是当nOctaves增加或减少时，尺度之间的间距会增加或减小。[注51]

构造函数的最后一个参数是selectedPairs此参数适用于真正的专家，并允许你覆盖描述符构造中用于比较的对列表。如果存在，selectedPairs必须是一个完全的512个整数的向量。这些整数索引到所有可能的字段配对的内部表格。[注52]这样，你可以将自己的对作为整数。大多数用户不可能使用此功能，它只适合专业的使用者使用——他们在阅读FREAK的原始论文后，会得出结论，有必要重复FREAK作者使用的学习过程，希望能够自己最大化描述符功效独特的数据集的使用者。

稠密特征网格和cv::DenseFeatureDetector类

还有一个其他的特征检测器，它真的是一种“几乎”的特征检测器，因为它不能真正检测到特征，它只是产生它们。cv::DenseFeatureDetector类[注53]的目的只是在图像中的网格中生成一个规则的特征数组（图16-34）。在这种情况下，仍旧没有完成这些特征（即，不计算描述符）。然而，一旦生成这些关键点，就可以计算你喜欢的任何描述

注51：术语“octave”源自cv::KeyPoint对象有一个元素octave，表示找到关键点的比例。cv::FREAK::FREAK()的nOctaves参数对应于相同的octave值。nOctaves（至少）应该设置为所使用的关键点检测器的最大octave，或者要计算描述的关键点集合中的最大octave。

注52：这些对索引从0到902，从(1,0)，(2,0)，(2,1)等开始，直到(42,41)。对于这种特定的排序，没有什么自然的解释，本模块中作者选择将外循环从索引i = 1到i < 43，内循环从j = 0到j < i的循环，并且构造全部的对[k]=(i,j)。然而，最重要的是，你不可能直接与这些索引进行交互。如果你使用这些索引，那么OpenCV可能会使用cv::FREAK::selectPairs()函数为你构建它们。

注53：DenseFeatureDetector没有在OpenCV3.0中实现。

符。事实证明，在许多应用中，不仅仅是足够而且特别需要在任何地方计算一个描述符，在这个意义上，由你选择的一些密度的统一网格表示“无处不在”。



图16-34. 在这里，汽车的两个视图产生了密集的特征，在这种情况下有三个级别，起始空间步长为50，缩放步长为2.0，特征步长就是特征尺寸[注54]

不仅计算图像空间中的均匀网格的特征是有用的，以均匀数量的尺度对该空间进行采样也是有用的。cv::DenseFeatureDetector已经实现了这些。

关键点查找器 稠密特征类的关键点查找器派生自cv::FeatureDetector接口，因为它没有描述符学习功能。因此，我们需要了解的重点是如何使用它的构造函数来配置它：

```
class cv::DenseFeatureDetector : public cv::FeatureDetector {

public:
  explicit DenseFeatureDetector(
    float initFeatureScale      = 1.f,  // Size of first layer
    int  featureScaleLevels     = 1,    // Number of layers
    float featureScaleMul       = 0.1f, // Scale factor for layers
    int  initXyStep             = 6,    // Spacing between features
    int  initImgBound           = 0,    // No-generate boundary
    bool varyXyStepWithScale    = true, // if true, scale 'initXyStep'
    bool varyImgBoundWithScale = false // if true, scale 'initImgBound'
  );

  cv::AlgorithmInfo* info() const;

  ...
};
```

稠密特征检测器的构造函数有很多参数，因为可能有很多不同的方法需要生成特征网格。第一个参数initFeatureScale设置第一层特征的大小。默认值1.0几乎肯定不是你想要的，但是请记住，特征的正确大小不仅取决于图像的性质，还依赖于要使用的描述符类型。

注54：这里的特征密度低的不切实际，我们选择这些特征仅是为了说明情况。

默认情况下，稠密特征检测器将生成单个关键点网格。你可以通过将featureScaleLevel设置为大于1的任何值来生成这样的关键点金字塔。在第一个之后生成的每个网格将生成特征的比例设置为initFeatureScale乘以featureScaleMul对每个级别高于第一个这样的规模生成的一个因子。[注55]图16-35显示了该检测器的使用情况。请注意，默认情况下，特征的位置不会缩放，仅仅是结论行设置了不同的scale参数。

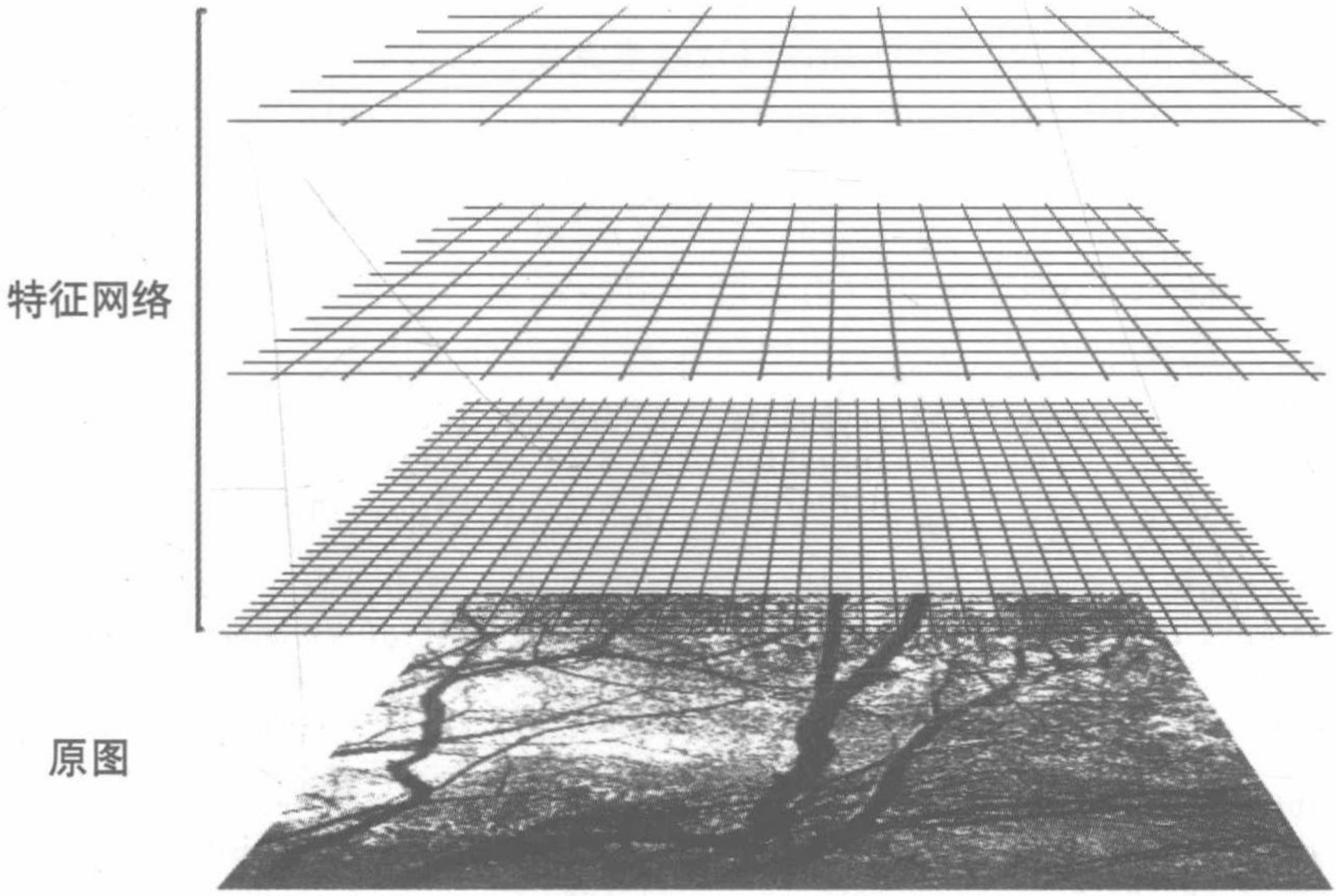


图16-35. cv::DenseFeatureDetector生成一个或多个不同比例关键点的网格

特征之间的间距由initXyStep设置。这将是所有级别上的所有特征之间的间距，除非varyXyStepWithScale设置为true，在这种情况下，特征之间的步长也将按照第一个尺度上每个级别的feature ScaleMul的分量进行缩放生成。

特征不会在图像的边界生成。你可以通过设置initImgBound来增加特征不会生成的区域的宽度。在大多数情况下，将initImgBound设置为等于initFeatureScale（即，生成的特征不会溢出图像的边缘）是有意义的。但是，为了在多个级别的情况下将此边界放大，你必须将vary ImgBoundWithScale设置为true；否则，它将保持恒定的大小。

关键点过滤

有一个相对常见的情况是，你发现自己有一些关键点列表，需要以某种方式修剪。有时候，则是因为列表太长了，你想丢弃一些质量较差的数据，直到找到一个可管理的数

注55：缩放倍数的默认值为0.1。它也可以是大于1的数字。将网格视为每一步都变得“更细”或“更粗”，这本质上就是个人风格的问题。

字。其他时候，你需要删除重复项或所有关键点以外的区域。OpenCV使用称为“关键点滤波器”的对象处理所有这些任务。关键点滤波器提供各种方法来剔除关键点列表。这些特征在各种关键点查找器的实现内部大量使用，但是你也许会发现它们在许多应用程序中都是非常有用的独立工具。

cv::KeyPointsFilter类

所有关键点过滤由单个cv::KeyPointsFilter类完成。cv::KeypointFilter类在我们已经遇到的许多关键点检测器内部使用，以便通过位置过滤（应用掩码）或将关键点的数量减少到列表中“最佳”的数量。这个类的定义如下：

```
class cv::KeyPointsFilter {
public:
  static void runByImageBorder(
    vector< cv::KeyPoint >& keypoints,  // in/out list of keypoints
    cv::Size                imageSize,  // Size of original image
    int                     borderSize  // Size of border in pixels
  );
  static void runByKeypointSize(
    vector< cv::KeyPoint >& keypoints,  // in/out list of keypoints
    float                   minSize,    // Smallest keypoint to keep
    float                   maxSize  = FLT_MAX // Largest one to keep
  );
  static void runByPixelsMask(
    vector< cv::KeyPoint >& keypoints,  // in/out list of keypoints
    const cv::Mat&          mask        // Keep where mask is nonzero
  );
  static void removeDuplicated(
    vector< cv::KeyPoint >& keypoints   // in/out list of keypoints
  );
  static void retainBest(
    vector< cv::KeyPoint >& keypoints,  // in/out list of keypoints
    int                     npoints     // Keep this many
  );
}
```

你可能首先注意到的是所有这些方法都是static，所以真正的cv::KeyPointsFilter比一个对象更像一个命名空间。五个滤波器方法中的每一个都引用了一个称为keypoints的cv::KeyPoint对象的STL样式向量。这是函数的输入和输出（即，你将给出关键点的函数，当你检查时，你会发现其中一些已经少了）。

cv::KeyPointsFilter::runByImageBorder()函数删除图像边缘的borderSize内的所有keypoints。你还必须通过imageSize参数告诉cv::KeyPointsFilter::runByImageBorder()一开始图像是多大。

cv::KeyPointsFilter::runByKeypointSize()函数将删除所有小于minSize或大于maxSize的keypoints。

cv::KeyPointsFilter::runByPixelsMask()函数删除与掩码中的零值像素相关联的所有keypoints。

cv::KeyPointsFilter::removeDuplicated()函数删除所有重复的keypoints。

cv::KeyPointsFilter::retainBest()函数移除keypoints，直到达到由npoint给出的目标数字。关键点按其质量的升序排列，如response所示。

匹配方法

一旦有关键点，你就想要使用它们来做一些有用的事情。正如我们早期讨论的，关键点理论两个最常见的应用是目标识别和跟踪。在这两种情况下，我们看到它是从cv::DescriptorMatcher基类派生的对象，为我们提供这个函数。在本节中，我们将看看可用于进行此类匹配的选项。

目前，你可以使用两种不同的匹配方法。第一个是暴力匹配，这是最基本和最明显的选择——只需将集合A中的所有内容与集合B中的所有内容进行比较。第二个称为*FLANN*，并且实际上是用于定位最近邻的方法集合的接口。

用cv::BFMatcher进行暴力匹配

cv::BFMatcher类的（重要部分）声明如下所示：

```
class cv::BFMatcher : public cv::DescriptorMatcher {

public:

  BFMatcher( int normType, bool crossCheck=false );

  virtual ~BFMatcher() {}

  virtual bool isMaskSupported() const { return true; }
  virtual Ptr<DescriptorMatcher> clone(
    bool emptyTrainData=false
  ) const;
  ...
};
```

暴力匹配器基本上实现了匹配问题最直接的解决方案。它需要查询集里面的每个描述符，并尝试将其与训练集中的每个描述符（内部字典或查询集一起提供的集合）进行匹配。创建暴力匹配器时，只需要决定使用什么距离度量计算距离进行比较。可用选项显示在表16-2中。

表16-2：强力匹配器的可用度量及其相关联的公式，总和超过特征向量的维数

度量	函数
NORM_L2	$dist\left(\vec{a}, \vec{b}\right) = \left[\sum_i \left(a_i - b_i\right)^2\right]^{1/2}$
NORM_L2SQR	$dist\left(\vec{a}, \vec{b}\right) = \sum_i \left(a_i - b_i\right)^2$
NORM_L1	$dist\left(\vec{a}, \vec{b}\right) = \sum_i abs\left(a_i - b_i\right)$
NORM_HAMMING	$dist\left(\vec{a}, \vec{b}\right) = \sum_i \left(a_i = = b_i\right)?1:0$
NORM_HAMMING2	$dist\left(\vec{a}, \vec{b}\right) = \sum_{i(even)} \left[\left(a_i = = b_i\right)\&\&\left(a_{i+1} = = b_{i+1}\right]?1:0$

值得注意的是，成员函数isMaskSupported()总是返回true。我们将在下一节中看到FLANN匹配器不是这样的。只有暴力匹配器（此时）支持我们讨论cv::DescriptorMatcher基类时描述的掩码结构。[注56]

暴力匹配对象的最后一个特征是所谓的交叉检查（cross-checking）。参数cross-checking打开对cv::BFMatcher构造函数的交叉检查。当启用交叉检查时，仅当train[j]是训练集中的query[i]最接近的邻域并且query[i]是查询集中的train[j]最接近的邻域时才会报告查询集合的对象i与训练集合的对象j之间的匹配。这对于消除假匹配非常有用，但是会花额外的时间来计算。

快速近似最近邻和cv::FlannBasedMatcher

术语*FLANN*是指用于近似近邻计算的快速库。OpenCV为FLANN提供了一个接口，它本身提供了各种算法，用于在高维空间中查找（或至少近似地找到）最近邻点。这正是我们需要的描述符匹配。FLANN的通用接口是通过cv::FlannBasedMatcher对象，它当然是从cv::DescriptorMatcher基类派生出来的：

```
class cv::FlannBasedMatcher : public cv::DescriptorMatcher {

public:

  FlannBasedMatcher(
    const cv::Ptr< cv::flann::IndexParams>&  indexParams
      = new cv::flann::KDTreeIndexParams(),
```

注56：注意，这些掩码是不掩盖图像的掩码，而是指出在匹配时哪些特性应该和不应该进行比较。

```
  const cv::Ptr< cv::flann::SearchParams>& searchParams
    = new cv::flann::SearchParams()
);

virtual void add( const vector<Mat>& descriptors );
virtual void clear();
virtual void train();
virtual bool isMaskSupported() const;

virtual void read( const FileNode& );     // Read from file node
virtual void write( FileStorage& ) const; // Write to file storage

  virtual cv::Ptr<DescriptorMatcher> clone(
    bool emptyTrainData = false
  ) const;
  ...
};
```

上述声明中的一切都是你现在应该需要的。一个重要的特征是cv::FlannBasedMatcher的构造函数，它使用一些特殊的结构来实际配置匹配。这些参数确定FLANN匹配器将使用什么方法以及如何参数化所选方法。例如，indexParams参数的默认值为new cv::flann::KDTreeIndexParams()。这告诉FLANN匹配器，它应该使用的索引方法是*kd-tree*以及使用多少个*kd-tree*（树的数量是cv::flann::KDTreeIndexParams()构造函数的一个参数，它的默认值为4）。searchParams参数更为通用，并且扮演着类似于cv::TermCriteria的角色，但是具有一些更通用的功能。我们将查看每个索引方法，然后返回到cv::flann::SearchParams对象。

线性索引与cv::flann::LinearIndexParams　你可以要求*FLANN*库像cv::BFMatcher一样，做基本相同的事情。这通常对于基准比较是有用的，但也可以用作比较以验证其他更逼近的方法是否能给出令人满意的结果。cv::LinearIndexParams的构造函数没有参数。作为示例，以下代码生成基本上等同于cv::BFMatcher的匹配器：

```
cv::FlannBasedMatcher matcher(
  new cv::flann::LinearIndexParams(), // Default index parameters
  new cv::flann::SearchParams()       // Default search parameters
);
```

使用cv::flann::KDTreeIndexParams进行KD树索引　cv::flann::KDTreeIndexParams索引参数告诉FLANN匹配器你要使用随机*kd-tree*进行匹配。FLANN的正常行为是假设你希望生成许多这样的随机化树作为索引方法，然后在尝试匹配[注57]时搜索它们。cv::flann::KDTreeIndexParams的构造函数接受一个参数，它指定要构造的这样的树有多

注57：这是一个玄妙的点，但事实是最近邻有一个主列表，并且当每个树下降时，候选最近邻不仅与该树上迄今为止已经发现的候选最近邻进行比较，还与目前为止在所有树上发现的候选最近邻进行比较。

少。默认值为4，但通常使此数字大到16左右。cv::flann::KDTreeIndexParams的声明如下：

```
struct cv::flann::KDTreeIndexParams : public cv::flann::IndexParams {
  KDTreeIndexParams( int trees = 4 ); // kd-tree needs number of trees
};
An example declaration of a matcher using cv::flann::KDTreeIndexParams is shown here:

cv::FlannBasedMatcher matcher(
  new cv::flann::KDTreeIndexParams( 16 ), // Index using 16 kd-trees
  new cv::flann::SearchParams()           // Default search parameters
);
```

cv::flann::KMeansIndexParams的分层k-means树索引 构建索引的另一个选择是使用分层*k*-means聚类。[注58]*k*-means聚类的优点在于它可以智能化使用数据库中点密度。分层*k*-means聚类是一个递归方案，通过该方案，数据点首先被分组成一些数量的聚类，然后每个聚类被分组成若干个子聚类，依此类推。显然，你有理由相信以这种结构保存数据时，会更有帮助。cv::flann::KMeansIndexParams的声明如下：

```
struct cv::flann::KMeansIndexParams : public cv::flann::IndexParams {

  KMeansIndexParams(
    int           branching    = 32,  // Branching factor for tree
    int           iterations   = 11,  // Max for k-means stage
    float         cb_index     = 0.2, // Probably don't mess with
    cv::flann::flann_centers_init_t centers_init
                               = cv::flann::CENTERS_RANDOM
  );

};
```

cv::flann::KMeansIndexParams结构的默认参数都是完全合理的值，因此在许多情况下，你可以放心使用这些参数。第一个参数branching是在分层*k*-means中使用的分支因子。它确定在树的每个级别将形成多少个聚类。下一个参数iterations，决定了k-means算法对于每个聚类的形成所允许的迭代次数。[注59]可以将其设置为-1，以强制聚类算法在树中的每个节点上运行直到完成。第三个参数控制集群中心的初始化。传统上，随机初始化是常见的（cv::flann::CENTERS_RANDOM），但近年来已经表明，在大多数情况下，审慎选择起始中心可以获得更好的结果。有两个附加选项cv::flann::CENTERS_GONZALES（Gonzales的算法[Tou77]）和cv::flann::CENTERS_KMEANSPP（所谓的“kmeans ++”算

注58：等到第20章介绍机器学习库时，我们将了解*k*均值算法。现在，知道它试图在*k*个不同的集群中组织一些大量的点已经足够。

注59：回想一下，*k*均值是一个NP-hard问题，所以用于计算*k*均值聚类的大多数算法都是近似的，并且使用迭代方案来找到给定一些特定候选集群中心的起始集合的局部最优解。

法[Arthur07]）可以选择，后者是越来越标准的选择。最终的参数cb_index（集群边界索引）是为FLANN库中真正的专家准备的；当树被搜索时使用它，并控制树如何被探索。最好保留其默认值，或设置为0，这表明排除一个域的搜索应该直接移到最接近的未开发的域。[注60]

将KD-tree和k-means与cv::flann::CompositeIndexParams相结合 该方法简单组合了先前描述的随机kd-tree和*k*-means方法，并尝试找到任一方法找到的最佳匹配。因为这些都是相似的方法，所以寻找另一种方式总是有潜在好处的。（你可以将其视为在kd-tree方法中具有多个随机树的逻辑的扩展。）cv::flann::CompositeIndexParams对象的构造函数将kd-tree和*k*-means方法的参数组合在一起：

```
struct cv::flann::CompositeIndexParams : public cv::flann::IndexParams {

  CompositeIndexParams(
    int              trees        = 4,   // Number of trees
    int              branching    = 32,  // Branching factor for tree
    int              iterations   = 11,  // Max for k-means stage
    float           cb_index     = 0.2,  // Usually leave as-is
    cv::flann::flann_centers_init_t centers_init
                                 = cv::flann::CENTERS_RANDOM

  );
};
```

使用cv::flann::LshIndexParams进行位置敏感哈希（LSH）索引 索引已知对象空间的另一种非常不同的方法是尝试使用散列函数将类似对象映射到相同的存储区中。在这样做的程度上，这些散列函数可以用于非常快速地生成候选对象的列表，然后可以对其进行评估和比较。这种技术称为局部敏感哈希（LSH）。作为FLANN库的一部分，OpenCV中实现的LSH的变化首先由Lv等人提出。[Lv07]：

```
struct cv::flann::LshIndexParams : public cv::flann::IndexParams {

  LshIndexParams(
    unsigned int table_number,      // Number of hash tables to use
    unsigned int key_size,          // key bits, usually '10' to '20'
    unsigned int multi_probe_level  // Best to just set this to '2'
  );

};
```

第一个参数table_number是要使用的实际哈希表的数量。这个数字通常是几十个，一般在10到30之间都比较合理。第二个参数key_size是散列键的大小（以位为单位）。该数

注60：如果此参数非零，则表示该算法应当考虑到域的总体大小，作为其下一步考虑的一部分。

字通常也大于10，但小于20。最后一个参数multi_probe_level控制如何搜索相邻的存储区；它是区分先前LSH实现中的多路径算法（之前引用）的一部分。multi_probe_level的推荐值为2.如果设置为0，则算法将退化为非多路传播LSH。



FLANN中的LSH索引仅适用于二进制特征（使用Hamming距离），它不宜应用于其他距离度量。

使用cv::flann::AutotunedIndexParams进行自动索引选择 也可以用OpenCV/FLANN来确定最佳的索引方案。不用说，这将需要一段时间。这种方法背后的基本思想是设置目标精度，即希望最近邻搜索返回正确的解的百分比。当然，设置的值越高，算法就越难找到可以提供的索引方案，实际生成一个完整索引所需要的时间就越长：

```
struct cv::flann::AutotunedIndexParams : public cv::flann::IndexParams {

  AutotunedIndexParams(
    float target_precision = 0.9,    // Percentage of searches required
                                     //  to return an exact result
    float build_weight     = 0.01,   // Priority for building fast
    float memory_weight    = 0.0,    // Priority for saving memory
    float sample_fraction  = 0.1     // Fraction of training data to use
  );

};
```

目标精度由参数targetPrecision设置。当你使用cv::flann::AutotunedIndexParams创建基于FLANN的匹配器时，还必须指出索引快速建立的重要程度，这由build_weight参数控制。如果不太在意建立索引需要多长时间，只要函数返回速度快，可以将其设置为非常小的值（如默认值为0.01）。如果需要经常建立索引，则需要将此数字设置得更高。类似地，memory_weight控制要最小化索引使用的内存量的优先级。此参数的默认值为0，这意味着不关心其优先级。

最后还有一个问题，就是要在这个搜索中实际使用多少训练数据。这由sample_fraction参数控制。显然，如果你把这个分数设得太大，就需要花费大量的时间才能找到满意的解。另一方面，如果把它设置太小，则你在完整数据集上观察到的性能可能会比生成索引时看到的差得多。对于大数据集，默认值0.1通常被认为是一个不错的选择。

FLANN搜索参数和cv::flann::SearchParams 除了之前用于indexParams参数的参数之外，cv::FlannBasedMatcher构造函数还需要一个类型为cv::flann::SearchParams的对象。这是一个简单的结构，可以控制一些匹配器的一般行为。它有以下简单的定义：

```
struct cv::flann::SearchParams : public cv::flann::IndexParams {

  SearchParams(
    int  checks = 32,   // Limit on NN candidates to check
    float eps    = 0,   // (Not used right now)
    bool sorted = true  // Sort multiple returns if 'true'
  );

};
```

参数checks在kd-tree和k-means算法中用法不同，但是在每种情况下，它本质上限制了在尝试找到真正最近邻时评估的候选最近邻的数量。参数eps现在没有使用。[注61]参数sort指示，在可以返回多个命中（例如，半径搜索）的搜索的情况下，返回的命中应该是距离查询点[注62]的距离的升序。

结果显示

现在，你可以计算所有这些特征类型并计算从一个图像到另一个图像的匹配。接下来要做的事情是在屏幕上实际显示关键点和匹配项。OpenCV为这些任务提供了一个函数。

使用cv::drawKeypoints来显示关键点

```
void cv::drawKeypoints(
const cv::Mat&                    image,      // Image to draw keypoints
const vector< cv::KeyPoint >& keypoints,  // List of keypoints to draw
cv::Mat&                          outImg,     // image and keypoints drawn
const Scalar&                     color    = cv::Scalar::all(-1),
int                               flags    = cv::DrawMatchesFlags::DEFAULT
);
```

给定一个图像和一组关键点，cv::drawKeypoints将把所有关键点标注到图像上并在outImg中输出结果。注释的颜色可以用color设置，当设置为特殊值cv::Scalar::all(-1)时，表示它们应当使用不同的颜色。flags参数可以设置为cv::DrawMatchesFlags::DEFAULT或cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS。在

注61：如果你是FLANN库的专家，就知道eps参数被称为KDTreeSingleIndex（目前未在OpenCV界面中公开）的kd-tree变体使用。在这种情况下，它确定何时可以终止特定分支的搜索，因为发现的点被认为已经足够近，以致于不太可能发现更近的点。

注62：排序参数对kNN搜索没有影响，因为在这种情况下，返回的命中值始终以升序排列。

前一种情况下，关键点用小圆标注。在后一种情况中，它们将用半径等于其size（如可获得）以及方位线由其angle（如可获得）给出的的圆圈来标注。[注63]

显示使用cv::drawMatches的关键点匹配

给定一对图像、相关联的关键点以及一个由其中一个匹配器生成的cv::DMatch 对象列表，cv::drawMatches()将会生成一幅图像，其中包含两个输入图像及所有关键点（以cv::drawKeypoints()的样式），并指明第一个图像中的哪些关键点与第二个图像中的关键点匹配（图16-36）。cv::drawMatches()有两种变化形式；除了两个参数外，其余部分都相同。

```
void cv::drawMatches(
  const cv::Mat&                  img1,         // "Left" image
  const vector< cv::KeyPoint >&   keypoints1,   // Keypoints (lt. img)
  const cv::Mat&                  img2,         // "Right" image
  const vector< cv::KeyPoint >&   keypoints2,   // Keypoints (rt. img)
  const vector< cv::DMatch >&     matches1to2,  // List of matches
  cv::Mat&                        outImg,       // Result images
  const cv::Scalar&               matchColor       = cv::Scalar::all(-1),
  const cv::Scalar&               singlePointColor = cv::Scalar::all(-1),
  const vector<char>&             matchesMask      = vector<char>(),
  int                             flags
                                        = cv::DrawMatchesFlags::DEFAULT
)
```

```
void cv::drawMatches(
  const cv::Mat&                  img1,         // "Left" image
  const vector< cv::KeyPoint >&   keypoints1,   // Keypoints (lt. img)
  const cv::Mat&                  img2,         // "Right" image
  const vector< cv::KeyPoint >&   keypoints2,   // Keypoints (rt. img)
  const vector< vector<cv::DMatch> >& matches1to2, // List of lists
                                                   // of matches
  cv::Mat&                        outImg,       // Result images
  const cv::Scalar&               matchColor    // and connecting line
                                        = cv::Scalar::all(-1),
  const cv::Scalar&               singlePointColor   // unmatched ones
                                        = cv::Scalar::all(-1),
  const vector< vector<char> >&   matchesMask   // only draw for nonzero
                                        = vector< vector<char> >(),
  int                             flags    = cv::DrawMatchesFlags::DEFAULT
);
```

注63：本章前面出现的许多图例都是用cv::drawKeypoints制作的。在这些图中，总是使用DrawMatchesFlags::DRAW_RICH_KEYPOINTS，但是在特征类型不存在尺度或角度信息的情况下，cv::drawKeypoints不会显示。图16-2、图16-4 和图16-9分别是只有位置数据可用的示例，只有位置，尺寸数据可用的示例以及位置，尺寸和方位数据均可用的示例。

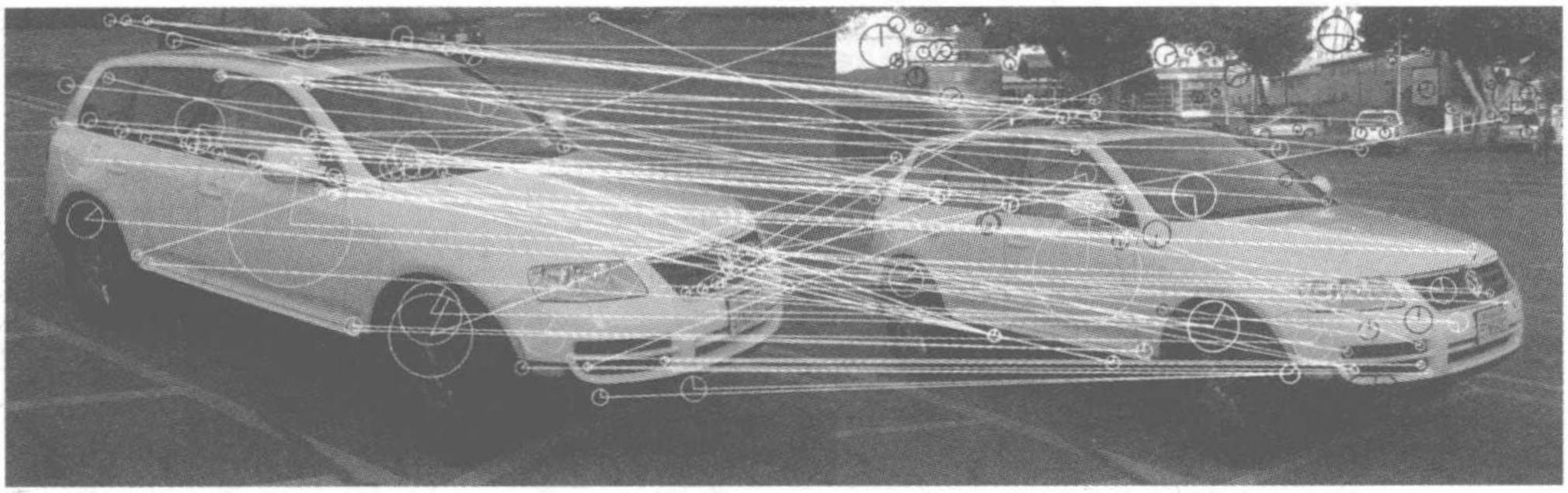


图16-36. 这里在同一汽车的两个视角图像中提取SIFT关键点及其描述符。使用基于FLANN的匹配器生成匹配，并使用cv::drawMatches()进行可视化。将找到的匹配标记为白色，并用一条线连接相对应的特征。每幅图中未找到匹配的关键点都以黑色标明

在两种情况下，两个图像都由img1和img2参数提供，而相应的关键点由keypoints1和keypoints2参数提供。matches1to2是在两种情况下不同的参数。它在两个版本的cv::drawMatches()中具有相同的含义，但在一种情况下，它是cv::DMatch对象的STL向量，而在另一种情况下，它是诸多向量的向量。第二种形式比较方便，当你想要一次性绘制许多不同匹配计算的响应时，它将会很有用。

结果放在图像outImg中。当绘制输出时，具有匹配的那些特征将被绘制在matchColor中（沿着一条连接它们的线），而未被匹配的那些特征将被绘制在singlePointColor中。向量matchesMask指示哪些匹配应该被绘制，只有matchMask[i]不为零的匹配项将被绘制。预期匹配的向量之向量的cv::drawMatches()的变化也需要matchMask参数的向量之向量。

Flags是cv::drawMatches()的最后一个参数。flags参数可以是能与OR运算符组合（相关）的四个值中的任何一个。

如果flags设置为cv::DrawMatchesFlags::DEFAULT，则输出图像将会创建在outImg中，并将关键点视为小圆（不需要额外的尺寸或方向信息）。[注64]

当flags包含cv::DrawMatchesFlags::DRAW_OVER_OUTIMG时输出图像将不会重新分配，但注释将被绘制到输出图像上。当有多组匹配并想要以不同的颜色进行绘制时，它很有用，在这种情况下，你可以对cv::drawMatches()进行多次调用，并在第一个调用之后对所有调用使用cv::DrawMatchesFlags::DRAW_OVER_OUTIMG。

默认情况下，不属于任何匹配的关键点将以singlePointColor所指示的颜色绘制。如果不想将它们绘制出来，可以设置cv::DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS标志。

注64：实际上，cv::DrawMatchesFlags::DEFAULT在数值上等于零，将它与其他数据组合在一起是没有意义的。

最后，标志cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS在cv::drawKeypoints中有相同的函数；它使关键点可以用尺度和方向信息进行绘制（如图16-36所示）。

小结

在本章一开始，我们回顾了亚像素角点和稀疏光流的基本概念，然后讨论了关键点的重要作用。从这里，我们看到OpenCV如何将关键点作为一般概念进行处理，并查看了通过OpenCV库实现的各种关键点检测的方案。我们还看到识别关键点过程与描述关键点特征过程的不同。这种表征是由描述符完成的。同关键点识别方法一样，有许多描述符类型，我们看到了OpenCV如何将这些描述符作为一般类来处理。

之后，我们讨论了了关键点及其描述符如何能够在目标识别或目标跟踪中以有效的方式进行匹配。我们通过查看OpenCV库中一个有用的函数来结束本章，这个函数使我们能够在可以找到关键点的图像环境中轻松地绘制关键点。注意：更多的特征检测器和描述符将在附录B中介绍xfeatures2d时提到。

练习

*.../samples/cpp*目录包含了OpenCV的示例代码例程，演示了本章中讨论的许多算法。请在以下练习中使用这些示例：

- *matchmethod_orb_akaze_brisk.cpp*（*samples/cpp*中的特征匹配部分）
- *videostab.cpp*（*samples/cpp*中用于稳定视频的特征跟踪部分）
- *video_homography.cpp*（*opencv_contrib/modules/xfeatures2d/samples*中的平面跟踪部分）
- *lkdemo.cpp*（*samples/cpp*中的光流部分）

1. 在cv::goodFeaturesToTrack()中使用的协方差Hessian矩阵计算了在该函数中由block_size设置的图像中的某个方形区域。

   a. 理论上，当块的大小增加时会发生什么？我们得到的“良好特征”会更多还是更少？为什么？

   b. 分析*lkdemo.cpp*代码，寻找cv::goodFeaturesToTrack()，并尝试改变block_size，观察不同。

2. 请参考图16-2研究实现亚像素角点寻找函数cv::findCornerSubPix()。

   a. 在图16-2中，如果角点扭曲以至于明暗交界处的直线形成交于一点的曲线，将会发生什么？还能寻找到寻找亚像素角点吗？请解释说明原因。

b. 如果扩展扭曲的棋盘角点的窗口大小（即扩展win和zero_zone参数之后），找到的亚像素的角点是更准确，还是开始出现偏差？解释你的答案。

3. 修改*matchmethod_orb_akaze_brisk.cpp*以在平面物体（例如，杂志或书籍封面）上训练，并用摄像机进行跟踪。研究并报告在修改后的代码中使用AKAZE，ORB和BRISK特征找到正确关键点的效果。

4. 在与练习3相同的平面图案上运行*video_homography.cpp*。使用程序中的键控制来学习此平面图案，然后跟踪它，注意如何使用单应性来产生稳定的输出。为了计算单应矩阵H，至少要找到多少个特征？

5. 使用你在练习3中学到的东西，测试*videostab.cpp*并描述它如何稳定视频。

6. 特征及其描述符也可用于识别。在空白的背景中拍摄三本不同书籍的俯视图，然后拍摄10张不同背景下的每本书封面的不同的偏心照片，最后拍摄10张在不同背景下没有书籍的照片。修改*matchmethod_orb_akaze_brisk.cpp*以分别存储每本书的描述符。然后再次修改*matchmethod_orb_akaze_brisk.cpp*以尽可能在偏心照片中检测（尽可能最好）正确的书，在没有书籍的照片中不显示任何书。报告结果。

7. 关于光流的练习：

   a. 描述一个使用块匹配比使用Lucas-Kanade光流跟踪效果更好的物体。

   b. 描述一个使用Lucas-Kanade光流比使用块匹配跟踪效果更好的物体。

8. 编译*lkdemo.cpp*。设置网络摄像头以查看纹理良好的对象（或使用对一个纹理有移动的对象事先捕获的序列）。在运行程序时，请注意，‘r’键自动初始化跟踪，‘c’键清除跟踪，单击鼠标将进入新点或关闭旧点。运行*lkdemo.cpp*并通过键入**r**来初始化点跟踪。观察效果。

   a. 现在打开代码并删除亚像素点放置函数cv::findCornerSubPix()。这样是否会对结果造成损失？如何造成的？

   b. 再次打开代码，替换cv::goodFeaturesToTrack()，仅在对象周围的ROI中放置一个点的网格。描述该点将产生什么结果以及原因。

   提示：产生结果的一部分是光圈问题导致的——给定一个固定的窗口大小和一条线，我们并不知道线是如何移动的。

10. 修改*lkdemo.cpp*程序以创建一个程序，使适当移动的相机保持简单的图像稳定。将稳定结果显示在比相机输出窗口大得多的窗口的中心，虽然图像流可能会摆动，但第一个点仍然可以保持稳定。

第17章

跟踪

跟踪中的概念

与处理单独的静态图像不同的是，当处理视频源时，我们经常会希望在视野中跟踪一个或一些特定的目标。在以前的章节中，我们了解了很多可以用来分离某个特定形状的方法，比如人或者汽车。我们还了解到这些目标如何被表示为关键点的集合，以及这些关键点如何在视频流中的不同图像或不同帧之间相互关联。

在实践中，计算机视觉跟踪的大多数问题出现在两种情形下：一种是跟踪已识别的目标；一种是跟踪未知目标，并且在许多情况下，还需要根据未知目标的动作对目标本身进行识别。

虽然通常而言，前几章所介绍的技术（例如矩或颜色直方图）已经可以用来识别帧中的目标，但在许多情况下，实验人员仍然需要通过分析运动本身来判断其中的目标是否存在，或者对目标的某些有趣的性质进行分析。

学习完前一章中关键点和描述符的内容，相信读者已经对稀疏光流有了一定的基础。在本章中，我们将介绍几种可用于稠密光流的技术。区别于稀疏光流，稠密光流的稠密性在于对给定区域中的每一个像素都进行计算。

除了跟踪之外，运动目标建模也是一个值得关注的问题。建模不仅有助于解决跟踪技术，甚至实现最优跟踪，还能从帧与帧的关系中对目标的实际位置进行精度有限的测量估计。在这方面，很多强大的数学方法已经被提出来，用这种方式对目标轨迹的测量进行估计，当然精度还是有限。这些方法适用于目标及其位置的二维或三维模型。在本章

的后半部分，我们将探讨如何使用OpenCV提供的工具来解决这个问题，并讨论其背后的一些原理。

本章将从详细讨论稠密光流开始，包括OpenCV库中几种已经实现的算法。每种算法对稠密光流的定义稍有不同，因此，所给的结果也会有略微的不同，每种算法在不同的情况下效果也有差异。在此之后我们将继续跟踪的部分。我们将看到的第一个跟踪算法的应用是区域跟踪，且基本上都以稠密的方式进行跟踪。这些方法包括meanshift和camshift跟踪算法以及运动模板法。

我们还将结合本章内容对卡尔曼滤波器（一种用于建立跟踪目标动作模型的方法）进行讨论。这将有助于我们把之前那些大量的不够精确的典型观测结果与我们现有的关于目标行为的知识进行融合，以便对真实世界中的目标行为进行最优估计。请注意，在附录B的*opencv_contrib*目录下，光流和跟踪函数还参考了更加新颖的光流和跟踪算法。

稠密光流

到目前为止，读者已经学习了如何从一幅图像和另一幅相邻图像中匹配单个特征。当用于光流问题时，这也必然会以一种稀疏的方式表示场景中目标的全局运动。在这种情况下，如果视频中有一辆汽车移动，我们会了解到汽车的某一部分到了哪里，也可能会对汽车的大范围的运动做出合理的推断，但是并不一定能够了解到场景中全部目标的运动情况。尤其是在包含许多车辆的复杂场景中，确定哪些车辆特征与哪辆车匹配并不总是那么容易。稀疏光流的概念选择是一个稠密的结构，其中运动矢量被分配给我们的图像中的每个像素。结果是能够获得一个速度矢量场，允许我们在分析这些数据时使用更多新的方法。

在实践中，计算稠密光流是不容易的。想象一下一张白纸的运动。在前一帧中的许多白色像素，在下一帧依然会保持白色。只有边缘部分想像素可能会改变，甚至只有那些垂直于运动方向的边缘像素才会改变。所以，稠密理论必须具有一些在更容易跟踪的点之间进行插值的方法，才能解决那些更易模糊的点的运动计算问题。这些困难在稠密光流的高计算代价中已经体现得很清楚。

早期的Horn-Schunck算法[Horn81]试图计算出这样的速度场，并解决这些插值问题。这种方法是对块匹配策略的改进，虽然简单但仍然存在问题。它只是简单地试图将每个像素周围的窗口从当前帧匹配到下一帧。

这些算法都是在早期版本的OpenCV库中实现的，效果也不是很好。然而，这个领域之后的许多工作已经产生了很多算法，尽管与稀疏关键点方法相比依然很慢，但单纯从使

用目的来说，已经好了。当前版本的OpenCV库支持两种这样较新的算法，分别是多项式扩展和双$TV\text{-}L^1$算法。

Horn-Schunck法和块匹配算法在库的遗留部分（即它们具有C接口）中仍然受支持，但是已经被官方弃用了。本书只研究多项式扩展算法和一些更为现代的算法，虽然它们最终也是得益于Horn和Schunck的工作，但与原始算法相比，改进明显且表现出了更好的性能。

Farneback多项式扩展算法

由G. Färneback [Farneback03]开发的多项式扩展算法试图基于一种类似于把图像视为连续表面的分析技术来计算光流。当然，真正的图像是离散的，所以Färneback方法还有一些额外的复杂操作，使我们能够将基本的方法应用于真实图像。作为一个函数，Färneback算法的基本思想是通过将多项式局部拟合到图像上的每个点来逼近图像。

算法的第一阶段，从它声明开始，是将图像变换成与每个点相关联的二次多项式的表示。该多项式是近似的基于加权的像素周围的窗口，以使拟合对更靠近窗口中心的点更敏感。因此，窗口的尺度决定了算法敏感特征的尺度。

在理想情况下（图像可以被视为平滑连续函数），图像上一部分的微小位移会导致该处点对应的多项式展开系数的解析计算的变化。从这个变化可以向后推进并计算该位移的大小（图17-1）。当然，这只对小的位移有意义。当然，处理更大的位移也得有一个很好的技巧。

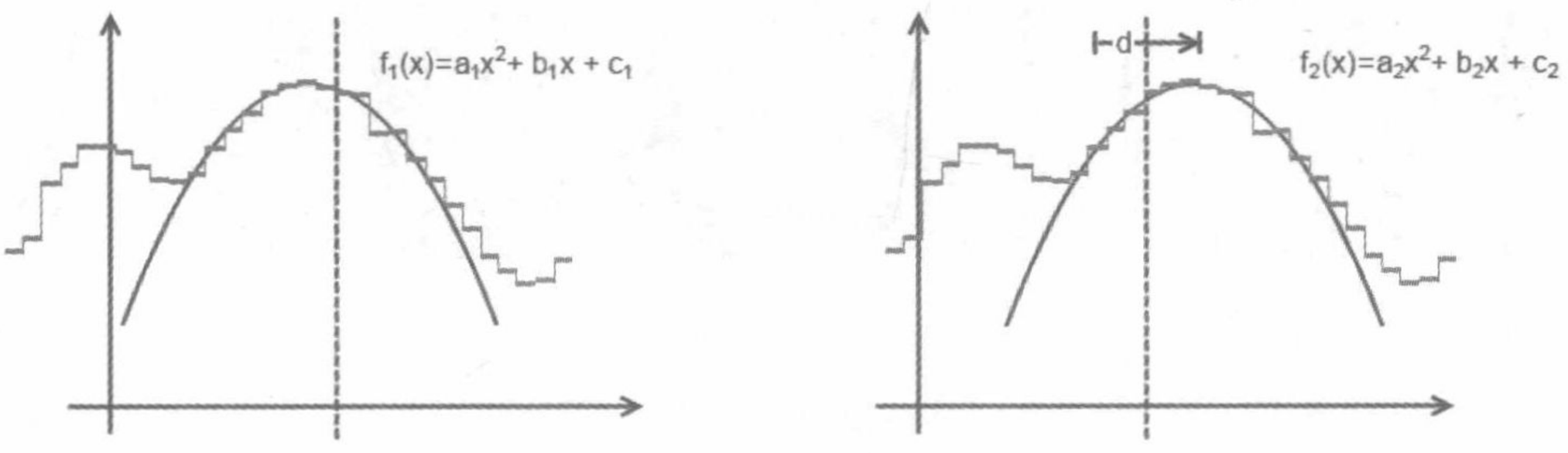


图17-1. 在一维情况下，图像由灰色直方图I（x）表示，前后两张图存在一个小位移d。在任何给定点（虚线），抛物线曲线可以拟合到附近的强度值。在平滑函数I（x）和小位移的近似中，得到的二阶多项式f1和f2通过以下分析公式联系起来：$a_2 = a_1$，$b_2 = b_1 - 2a_1*d$，$c_2 = a_1 * d^2 - b_1*d + c_1$。给定前后的拟合系数，并利用这些关系中的第二个关系，可解得：$d = -\frac{1}{2}a_1^{-1}(b_2 - b_1)$

诀窍是要首先注意到，如果知道一些位移的信息，你可以在其他点比较两个图像，除了检测到位移的相关点。在这种情况下，这种分析技术只计算一个对原始位移估计的校正。

事实上，这种机制可以用来简单地迭代算法，并且在每次迭代时对运动估计进行连续的改进。

这个想法也可以有助于发现更大幅度的运动。考虑首先从图像金字塔估计一对较低分辨率图像上的这些位移的情况。在这种情况下，运动看起来更小，必要条件“小位移”近似可能成立。然后，向下移动金字塔，每个先前计算的结果可以用作下一个的初始预测。最后，可以通过累积这些连续校正来获得原始图像的尺度的结果。

用cv::calcOpticalFlowFarneback计算稠密光流

OpenCV提供了一个完整的Farnebäck方法的实现，参见图17-2的示例输出。这个函数完全包含在cv::calcOpticalFlowFarneback()中，原型如下：

```
void cv::calcOpticalFlowFarneback(
  cv::InputArray       prevImg,    // An input image
  cv::InputArray       nextImg,    // Image immediately subsequent to 'prevImg'
  cv::InputOutputArray flow,       // Flow vectors will be recorded here
  double               pyrScale,   // Scale between pyramid levels (< '1.0')
  int                  levels,     // Number of pyramid levels
  int                  winsize,    // Size of window for pre-smoothing pass
  int                  iterations, // Iterations for each pyramid level
  int                  polyN,      // Area over which polynomial will be fit
  double               polySigma,  // Width of fit polygon, usually '1.2*polyN'
  int                  flags       // Option flags, combine with OR operator
);
```



图17-2. 比较来自左边和中间帧的运动，并用右边的矢量表示。原始图像为640x480，winsize = 13，numIters = 10，polyN = 5，polySigma = 1.1，并且使用盒形内核进行预平滑。运动矢量只显示在较低密度的子网格上，但实际结果为对每个像素都有效

前两个参数是用于计算位移的前一帧图像和下一帧图像。两者都应该是8位，单通道图像，并且尺寸相同。下一个参数，光流是结果图像；它的尺寸将与prevImg和nextImg相同，但是数据类型是双通道和32位浮点型（CV_32FC2）。pyrScale和levels影响图像金字塔的构造方式。pyrScale必须小于1，并指示金字塔相对于其前身的每个级别的大小，例如，如果pyrScale为0.5，那么金字塔将是“典型的”二分之一比例金字塔。levels决定了金字塔将有多少级。

winsize参数控制在拟合之前完成的预平滑操作。winsize（奇数）取值大于5时，图像噪声不会对拟合造成很大的麻烦，并且可以检测更大（更快）的运动。另一方面，它也将倾向于模糊所产生的运动场，并且使得检测小目标上的运动变得困难。这个平滑过程可以是高斯模糊或简单的滑动——平均窗口（由标志参数控制，很方便描述）。

iterations参数控制在金字塔的每个级别使用了多少次迭代。通常，增加迭代次数都会提高最终结果的准确性。虽然在某些情况下，三次甚至一次迭代就足够了，但算法的作者发现六次迭代对一些场景来说效果更好。

polyN参数决定在像素点附近拟合多项式时所考虑的区域的大小。不同于winsize，它只用于预平滑。polyN可以被认为与窗口大小（与Sobel微分有关联）类似。如果这个数字很大，高频扰动对多项式拟合就没有影响。与polyN密切相关的是polySigma，它是运动场的内在尺度的来源。拟合过程中的微分计算使用高斯内核（而不是与平滑相关的方法）方差为polySigma并且其总范围为polyN。polySigma的值应该超过polyN的20%以上。（实际过程中polyN=5且polySigma=1.1，或polyN=7且polySigma=1.5时，效果很好，建议在源代码中使用。）

cv::calcOpticalFlowFarneback()的最后一个参数是flags，它支持好几个能与逻辑运算符OR结合的选项。第一个选项是cv::OPTFLOW_USE_INITIAL_FLOW，它指示算法中序列流也应该被当作一个输入，并且应该被用作场景中运动的初始估计。在分析视频中的连续帧时，理论上通常认为一帧可能包含与下一帧相似的运动。第二个选项是cv::OPTFLOW_FARNEBACK_GAUSSIAN，它告诉算法在预处理（由winsize控制）中使用高斯内核。一般来说，使用高斯内核进行预处理结果更好，但是也会花更多计算时间。更高计算代价不仅来自于平滑求和中卷积核权重值的乘积运算，而且也因为使用高斯内核后，winsize趋向于需要一个更大的值。[注1]

Dual TV-L[1]模型

Dual TV-L[1]算法是Horn和Schunck（HS）算法的演变，我们在本章前面简要讨论过。OpenCV中的实现是基于Christopher Zach，Thomas Pock和Horst Bischof [Zach07]的原始论文以及Iavier Sánchez，Enric Meinhardt-Llopis和Gabriele Facciolo [Sánchez13]提出的改进策略。原始的HS算法是基于光流问题的一个公式，它解决了容易直接求解（尽管不是很快）的数值方法的问题，Dual TV-L[1]算法依赖于一个略微不同的公式，但解决问题

注1：实际应用的平滑内核的sigma等于0.3 • (*winsize* / 2)。实际上，这意味着高斯块质量的大部分区域只有核心区域的大约60%～70%。因此，内核需要再扩大30%左右，才能显示出盒子内核的优势。

更为有效。由于HS算法在Dual TV-L^1算法的演进中起着核心作用，所以我们将简要回顾一下，然后解释定义Dual TV-L^1算法的不同之处。

HS算法通过定义流向量场和能量损失进行操作。能量损失是先前帧强度、后续帧强度和该流矢量场的函数。这种能量由以下公式定义：

$$E(\vec{x}, \vec{u}) = \sum_{\vec{x}\varepsilon image} I_{t+1}(\vec{x} + \vec{u}) - I_t(\vec{x})^2 + \alpha^2(\| \nabla u_x \|^2 + \| \nabla u_y \|^2)$$

在这个表达式中，$I_t(\vec{x})$是图像在时间t和位置$\vec{x} = (x, y)$的强度。$u_x = u_x(\vec{x})$和$u_y = u_y(\vec{x})$是位置处的光流场的x方向和y方向的分量$\vec{x}$。矢量$\vec{u}$是(u_x, u_y)的一个简写。值α^2是影响第一约束（保真度）对第二约束（平滑度）2的相对影响的权重参数。[注2]HS算法尝试在所有可能的流场中最小化这种能量函数$\vec{u}$。实际实现这种最小化的方法（由原作者提出）是将能量函数转换为相关的欧拉－拉格朗日方程，并迭代求解。这种方法的主要问题是，因为欧拉－拉格朗日方程是完全局部的，所以每次迭代只能解决相对于像素的最近邻的问题，因此实际运行时，这些计算代价大的迭代执行次数可能会非常多，使用分层方法或许有所帮助。

下面让我们看一下Dual TV-L^1算法，它与HS算法在能量函数的公式和解决这个问题的用法都不同。“TV-L^1”这个名字意味着在平滑度约束使用L^1范数的情况下，保真度约束被替换为总体变化（TV），这两者都与HS算法形成对比，术语“总体变化”意味着将差分简单地求和，而不是平方和求和，同时将L^1范数应用于平滑度约束中的梯度，而不是由HS算法使用的L2范数，因此TV-L^1算法使用的能量函数如下：

$$E(\vec{x}, \vec{u}) = \sum_{\vec{x}\varepsilon image} \lambda \left| I_{t+1}(\vec{x} + \vec{u}) - I_t(\vec{x}) \right| + \left(\left| \nabla u_x \right| + \left| \nabla u_y \right| \right)$$

选择L^1范数的主要优点是局部梯度不会受到如此严重的惩罚，以致于算法在不连续性上表现更好[注3]。使用总体变化而不是平方差的总和的重要性在于，有助于解决这种可选的能量函数。Horn和Schunck依赖于一个迭代的欧拉－拉格朗日方法，而DualTV-L^1算法依赖于将能量最小化分为两个独立问题的巧妙技巧。其中一个问题有一个已知的解决方案（恰好是 Dual TV-L算法名称中的“Dual”），而另一个问题具有对于每个像素完全局部的非常理想的性质，因此可以逐点解决。让我们看看这是如何工作的。

注2：通常，保真度约束（能量函数中的第一平方差）表示每像素的一个约束。因为光流场是二维的，这样的问题总是没有得到充分的说明。从根本上说，这也是为什么总是需要一些连续性约束的原因。

注3：从$\alpha 2$到λ的变化是为了保持两种方法原作者所用变量的一致性。两者在其他方面完全相同。

首先，假设我们有一个非常接近最终流场$\vec{u}$[注4]的流场$\vec{u}^0$。基于这个假设，我们可以使用一阶泰勒展开来近似能量方程的差分，如下：

$$\rho(u) \equiv \nabla I_{t+1}(\vec{x} + \vec{u}^0) \cdot (\vec{u} - \vec{u}^0) + I_{t+1}(\vec{x} + \vec{x}^0) - I_t(x, y)$$

因此，在这个近似下，我们可以把能量写成如下形式：

$$E(\vec{x}, \vec{u}) = \sum_{\vec{x} \varepsilon image} \lambda \mid \rho(\vec{u}) \mid + \mid \nabla u_x \mid + \mid \nabla u_y \mid$$

接下来的内容是方法的核心，即引入一个新的场$\vec{v}$，使能量变成如下形式：

$$E(\vec{x}, \vec{u}, \vec{v}) = \sum_{\vec{x} \varepsilon image} \mid \nabla u_x \mid + \mid \nabla u_y \mid + \frac{1}{2\theta} \mid \vec{u} - \vec{v} \mid + \lambda \mid \rho(\vec{v}) \mid$$

这在使保真度和平滑度解耦方面非常有用。当然，我们现在还有另外一个场要解决，但是在极限情况下，θ非常小的时候，$\vec{u}$和$\vec{v}$实际上趋近于相等。然而，这个变化的最大好处是，现在我们可以先通过固定一个场，然后解决另外一个，然后固定另一个场，然后解决第一个问题（并以这种方式迭代）来解决这个场。当$\vec{v}$固定时，我们找到$\vec{u}$的值，后者对下式进行最小化：

$$\sum_{\vec{x} \varepsilon image} \mid \nabla u_x \mid + \mid \nabla u_y \mid + \frac{1}{2\theta} \mid \vec{u} - \vec{v} \mid$$

当$\vec{u}$固定时，我们找到$\vec{v}$的值，后者对下式进行最小化：

$$\sum_{\vec{x} \varepsilon image} \frac{1}{2\theta} \mid \vec{u} - \vec{v} \mid + \lambda \mid \rho(\vec{v}) \mid$$

其中第一个问题具有已知解决方案[注5]，而第二个是完全局部的问题，可以在每个像素上解决。然而，这第一个过程引入了两个新的参数：时间步长和停止准则。当$\vec{v}$固定时，这些参数控制着计算$\vec{u}$时如何和何时达到收敛。

使用cv::createOptFlow_DualTVL1计算稠密光流

Dual TV-L算法在OpenCV中的实现使用与本节中其他光流算法略有不同的接口风格。在名为`cv::createOptFlow_DualTVL1()`的函数库中有一个独立的工厂式函数，它构造一个

注4：实际上，该算法在图像的金字塔上实现，使得可以首先解决最粗糙的尺度，并且结果可以传播到更细微的尺度。这样，就可以一直得到$\vec{u}^0$。

注5：这就是所谓的总体变量去噪模型（total variation denoising model），可以通过Chambolle的二重性算法[Chambolle04]来求解。

类型为cv::OpticalFlow_DualTVL1的对象（派生自基类cv::DenseOpticalFlow），并返回它的指针：

```
cv::Ptr <cv::DenseOpticalFlow> createOptFlow_DualTVL1();
```

生成的对象具有成员变量。如果要更改这些变量的默认值，需要直接覆盖。这是cv::OpticalFlow_DualTVL1定义的相关部分：

```
// Function to get a dense optical flow object
//
cv::Ptr<cv::DenseOpticalFlow> createOptFlow_DualTVL1();
class OpticalFlowDual_TVL1 : public DenseOpticalFlow {

public:

OpticalFlowDual_TVL1();

  void calc( InputArray I0, InputArray I1, InputOutputArray flow );
  void collectGarbage();

  double tau;           // timestep for solver (default = 0.25)
  double lambda;        // weight of smoothness term (default = 0.15)
  double theta;         // tightness parameter (default = 0.3)
  int    nscales;       // scales in pyramid (default = 5)
  int    warps;         // warpings per scale (default = 5)
  double epsilon;       // stopping criterion (default = 0.01)
  int    iterations;    // max iterations (default = 300)
  bool   useInitialFlow; // use 'flow' as starting guess (default = false)

};
```

你可以设置变量来配置算法构造函数的参数。tau是数值求解器使用的时间步长。它可以设为小于0.125的任何值，并且保证收敛，但从经验上可以将它设到0.25以获得更快收敛（实际上，这是默认值）。lambda是最重要的参数；它设定了能量中平滑度的权重。λ的理想值将根据图像序列而变化，值越小，解越平滑。lambda的默认值为0.15。参数θ被原作者称为“紧密度参数”，这是耦合整个求解器的两个阶段的参数，原则上应该非常小，但算法对于更宽泛的值也很稳定，默认的紧密度参数为0.30。

图像金字塔中的尺度数由nscales设置。对于每个尺度，扭曲的次数是$\nabla I_{t+1}(\vec{x}+\vec{u}^0)$和$I_{t+1}(\vec{x}+\vec{u}^0)$在每个尺度上计算的次数。该参数允许在速度（扭曲更少）和精度（扭曲更多）之间进行权衡。默认情况下，有5个尺度，每个尺度有5个扭曲。

epsilon是数值求解器所用的停止准则。另外还有一个迭代标准，它设定允许的最大迭代次数。epsilon的默认值为0.01，而迭代的默认值为300。epsilon越小，所求解的值越精确，但时间代价也更大。

你要设置的最后一个参数可能是useInitialFlow。如果将此参数设置为true，那么在

调用cv::OpticalFlow_DualTVL1对象的calc()时，就将你选择的流参数作为计算的起始点。许多情况下，在序列视频上运行算法时，使用上一次的结果是有意义的。

使用cv::OpticalFlow_DualTVL1::calc()来计算实际的光流。calc()函数需要两个8位单通道图像作为输入，并从中计算出输出光流。如上所述，flow是一个输入输出参数，如果你设置cv::OpticalFlow_DualTVL1对象的useInitialFlow为true，那么当前光流中的任何数据都将被用作下一次计算的起始点。在任何情况下，光流的大小与输入图像的大小相同，并且类型为CV_32FC2。

Cv::OpticalFlow_DualTVL1的最后一个使用方法是collectGarbage()。该方法不需要参数，只是简单地释放cv::OpticalFlow_DualTVL1对象内的任何内部分配的内存。

简单光流算法

最近计算光流的另一种算法是简单流算法。最初由Michael Tao等人提出 [Tao12]，该算法的重要特征是要求图像中像素数为亚线性[注6]时间。这是通过基于金字塔的方案实现的。当从金字塔中的粗糙层向精细层移动时，确定在新层中的像素处是否需要光流计算。如果发现在较细的水平没有获得新的信息，则不进行光流计算。反之，光流被简单传播到新的层并插入到新的层中。

简单流程算法试图为每个点建立一个局部流向量，可以最优解释该点周围邻域的运动。这通过计算优化能量函数的（整数）流向量来实现。该能量函数基本上是邻域中每个像素的能量和，其中能量随着时间t附近的像素的强度和时间$t + 1$对应的像素（即被流向量移位）之间的差分二次增长，像素能量函数$e(x, y, u, v)$定义如下（其中u和v是流矢量的分量）：

$$e(x, y, u, v) = \| I_t(x, y) - I_{t+1}(x + u, y + v) \|^2$$

然后我们可以写出实际能量函数，用下面的式子最小化：

$$E(x, y, u, v) = \sum_{(i,j)\varepsilon N} w_d w_c e(x + i, y + j, u, v)$$

其中w_d和w_c有如下定义：

$$w_d = \exp\left(- \frac{\| (x, y) - (x + i, y + j) \|^2}{2\sigma_d} \right)$$

注6：从技术上讲，简单光流算法不是亚线性的，因为它具有必须在每个像素上操作的成分。然而，算法高计算代价的操作不需要应用于每个像素，因此在实践中计算代价是亚线性的。在他们的论文[Tao12]中，作者证明了4K（即“quad hd”，4096×2160）图像一样大的图像具有亚线性时间。

以及：

$$w_c = \exp\left(- \frac{\| \ I_t(x, \ y) - I_t\left(x + i, \ y + j\right)\| \ ^2}{2\sigma_c}\right)$$

这两个项的结果是产生一个双边滤波器[注7]。注意，如果对于小的σ_d值，在远离中心点(x, y)的像素上，w_d变得非常小。类似地，对于小的σ_c值，对于其强度不同于中心点(x, y)的像素，w_c变得非常小。

关于(u, v)的可能整数值的范围，能量$E(x, y, u, v)$首先被最小化。一旦完成，就拟合一条抛物线来寻找使得$E(x, y, u, v)$最小的整数(x, y)周围的3×3个单元格。以这种方式，可以内插最优的非整数值。然后将所得到的光流场（即以这种方式发现的所有(u, v)的集合）通过另一个双边滤波器[注8]。但是，请注意，此滤波器现在在(u, v)向量上运行，而不是能量密度$e(x, y, a, v)$。因此，这是一个独立的滤波器，它具有单独的参数σ_d^{fix}和σ_c^{fix}。不仅独立于σ_d和σ_c，甚至还没有相同的单位。前者是速度方差，后者是强度方差。

这解决了一部分问题，但为了计算帧中超过几个像素的运动，有必要在速度空间中搜索更大的窗口，以找到$E(x, y, u, v)$的最优值。像计算机视觉中常见的那样，简单光流使用图像金字塔解决了这个问题。这样，在金字塔的较高层次上可以发现粗略的运动，并在金字塔较细的水平上进行细化。从金字塔中最粗糙的图像开始，我们开始通过对先前层进行上采样操作并内插所有新像素来计算每个后续层。这种上采样是通过一种叫“联合双边上采样”[KopfO7]的方法完成的。这种技术利用了这样一个事实：尽管已经对速度场的解进行了上采样，但是我们已经可以以更高的分辨率访问图像。联合双边滤波器引入了两个表征滤波器的空间和色彩范围的参数：σ_d^{up}和σ_c^{up}。

简单流程算法的一个重要贡献是作者所说的光流不规则图。这个图背后的基本思想是为每个点计算相邻像素光流与该像素处的光流不同的量：

$$\mathrm{H}(x, \ y) = \max_{(\mathrm{i,j})\varepsilon \mathrm{N}} \| \ \ (u(x + i, \ y + j), \ v(x + i, \ y + j)) - (u(x, \ y), \ v(x, \ y))\|$$

在发现光流不规则性较小时（和某些截止参数τ有关），光流在块N的角上计算并在块内插入。在不规则度超过τ的情况下，在较细的等级层次上重复刚才描述的光流计算。

注7：有关双边滤波器的简要回顾，请查看第10章中对函数cv::bilatera1Fi1ter()的讨论。

注8：对于真正的专家，这个第二个滤波器有一点不易察觉，即不适用于发现被遮挡的区域。关于如何实现这一点的详细信息，请参阅原文[Tao12]。

使用cv::0ptflow::calcOpticalFlowSF()计算光流

现在看下Simple Flow算法在OpenCV中的实现函数[注9]，即cv::optflow::calcOpticalFlowSF()。

```
void cv::optflow::calcOpticalFlowSF(
  InputArray  from,                    // Initial image (input)
  InputArray  to,                      // Subsequent image (input)
  OutputArray flow,                    // Output flow field, CV_32FC2
  int         layers,                  // Number of layers in pyramid
  int         averaging_block_size,    // Size of neighborhoods (odd)
  int         max_flow                 // Velocities search region sz (odd)
);

void cv::calcOpticalFlowSF(
  InputArray  from,                    // Initial image (input)
  InputArray  to,                      // Subsequent image (input)
  OutputArray flow,                    // Output flow field, CV_32FC2
  int         layers,                  // Number of layers in pyramid
  int         averaging_block_size,    // Size of neighborhoods (odd)
  int         max_flow                 // Velocities search region sz (odd)
  double      sigma_dist,              // sigma_d
  double      sigma_color,             // sigma_c
  int         postprocess_window,      // Velocity filter window sz (odd)
  double      sigma_dist_fix,          // Sigma_dAfix
  double      sigma_color_fix,         // Sigma_cAfix
  double      occ_thr,                 // Threshold used in occlusion detection
  int         upscale_averaging_radius, // window for joint bilateral upsampling
  double      upscale_sigma_dist,      // Sigma_dAup
  double      upscale_sigma_color,     // Sigma_cAup
  double      speed_up_thr             // Tao
);
```

cv::calcOpticalFlowSF()函数的第一种调用形式不需要对Simple Flow算法有特别深入的了解。它需要之前和现在的图像（在函数中分别是from和to），并返回速度场（存在于数组flow中）。输入图像应为8位三通道图像（CV_8UC3）；结果数组flow是一个32位的双通道图像（cv_32rc2）。该函数唯一需要指定的参数是运行初始速度求解器（在每个级别）时希望算法考虑的层数、邻域大小和最大光流速度。这三个参数layers，averaging_block_size和max_flow可以分别合理地设为5，11和20。

cv::calcOpticalFlowSF()的第二种形式允许真正进入并调整算法的每个参数。除了简写形式使用的参数之外，长格式允许设置在能量最小化中使用的双边滤波器参数（sigma_dist和sigma_color，与上一讨论中的σ_d和σ_c相同），对于速度场交叉双边滤波器（postprcess_window）的窗口大小，速度场交叉双边滤波器（sigma_dist_fix和sigma_color_fix，和前面讨论的和相同）的参数，用于闭塞检测的阈值，上采样联合

注9：从OpenCV 3.0开始，将calcOpticalFlowSF()移到opencv_contrib的optflow模块。

双边滤波器（upscale_averaging_radius）的窗口大小，上采样联合双边滤波器的参数（upscale_sigma_dist和upscale_sigma_color，与之前讨论中的σ_d^{fix}和σ_c^{fix}相同）和用于确定不规则图何时表明必须在更精细的金字塔级别（speed_up_thr，与前面讨论中的τ相同）上重新计算流量的检测阈值。

这些参数的详细调整显然是专家的话题，如果想深入了解这些参数，应该阅读原始论文。然而，对实际需求来说，知道使用表17-1中列出的名义值是有用的，默认情况下，cv::calcOptica1FlowSF()的简短参数版本。

表17-1：cv::calcOpticalFlowSF()函数的默认参数

参数	名义值
sigma_dist	4.1
sigma_color	25.5
postprocess_window	18
sigma_dist_fix	55.0
sigma_color_fix	25.5
occ_thr	0.35
upscale_averaging_radius	18
upscale_sigma_dist	55.0
upscale_sigma_color	25.5
speed_up_thr	10

Mean-Shift算法和Camshift 追踪

在本节中，我们将介绍平均值*shgft*和Camshift（Camshift代表“持续自适应均值移位”）的两种技术，前者是许多应用中数据分析的一般技术（在第12章讨论计算机视觉的场景只有一个），在介绍平均移位的一般理论之后，我们将介绍OpenCV如何将其应用于图像跟踪中，后一种技术Camshift建立在均值——移动以允许跟踪在视频序列期间尺寸可能改变的对象。

Mean-Shift算法

Mean-Shift算法[注10]是一种在数据集密度分布中寻找局部极值的鲁棒方法。对连续分布，

注10：Mean-Shift是一个相当深奥的话题，这里的讨论主要是为了给读者一些直观性的认识。对于原始的正式推导，可以参见Fukunaga [Fukunaga90]和Comaniciu和Meer [Comani-ciu99]。

这个过程很简单；在这种情况下，基本上就是将爬山算法应用于数据的密度直方图。[注11] 然而，对于离散数据集，这个问题就不那么简单了。

“鲁棒”（robust）这个术语在这里用的是它的统计学意义。也就是说，Mean-Shift忽略了离群点，即远离数据峰值的数据点。它只处理数据的局部窗口中的点，然后通过移动该窗口来实现对离群点的过滤。

Mean-Shift算法运行如下。

1. 选择一个搜索窗口：
   - 初始位置；
   - 类型（均匀、多项式、指数或高斯）；
   - 形状（对称或倾斜、可能旋转、圆形或矩形）；
   - 尺寸（卷起或被切断的程度）。
2. 计算窗口（可能有权重）质心。
3. 将窗口以质心为中心居中。
4. 返回到步骤2，直到窗口停止移动（它肯定会停止）。[注12]

Mean-Shift算法更正式的解释是：它与核密度估计的规则有关，根据内核我们可以推测出一个主要集中于局部的函数（例如高斯分布）。通过足够适当的加权和大小的内核，就可以根据这些内核来表达数据的分布。Mean-Shift与核密度估计不同，因为它仅仅寻找估计数据分布的梯度（变化方向）。当这个变化为θ时，我们处于一个稳定（虽然也许是局部稳定）的分布峰值。附近或其他尺度可能还有其他山峰。

图17-3展示了Mean-Shift算法中涉及的方程式。我们可以通过考虑一个矩形核[注13]来简化这些方程，这样可以减少Mean-Shift向量方程对图像像素分布质心的计算：

$$x_c = \frac{M_{10}}{M_{00}},\ y_c = \frac{M_{01}}{M_{00}}$$

注11：用这个词基本上是因为Mean-Shift还有一方面与尺度有关。准确地说，Mean-Shift在连续分布中等效于首先与Mean-Shift核卷积，然后使用爬山算法。

注12：一般我们通过设置最大迭代次数或设置一个可以认为窗口位置收敛的最小移动阈值来终止迭代。但是，迭代最终都会被保证收敛。

注13：一个矩形内核是一个与中心的距离没有衰减的内核，直到单个锐度转换为θ值。这跟高斯内核的指数衰减和通常使用的Epanechnikov内核中心距离的平方衰减不同。

这里的零阶矩计算如下：

$$M_{00} = \Sigma\Sigma I(x, y)$$

一阶矩是：

$$M_{01} = \underset{x\ y}{\Sigma\Sigma} x \cdot I(x, y) \text{ 和 } M_{10} = \underset{x\ y}{\Sigma\Sigma} y \cdot I(x, y)$$

从对一个概率分布 $P(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} K(\mathbf{x} - \mathbf{x}_i)$ 的近似核 $K(\mathbf{X} - \mathbf{X}_i) = ck\left(\left\|\frac{\mathbf{X} - \mathbf{X}_i}{h}\right\|^2\right)$ 开始，我们重点关注梯度 $\nabla P(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla K(\mathbf{x} - \mathbf{x}_i)$ 。令 $g(\mathbf{x}) = -k'(\mathbf{x})$，即核的导数，我们可以得到

$$\nabla P(\mathbf{x}) = \frac{c}{n}\sum_{i=1}^{n} \nabla K_i = \frac{c}{n}\left[\sum_{i=1}^{n} g_i\left(\frac{\|x - x_i\|^2}{h}\right)\right]\left[\frac{\sum_{i=1}^{n} x_i g_i\left(\frac{\|x - x_i\|^2}{h}\right)}{\sum_{i=1}^{n} g_i\left(\frac{\|x - x_i\|^2}{h}\right)} - x\right]$$

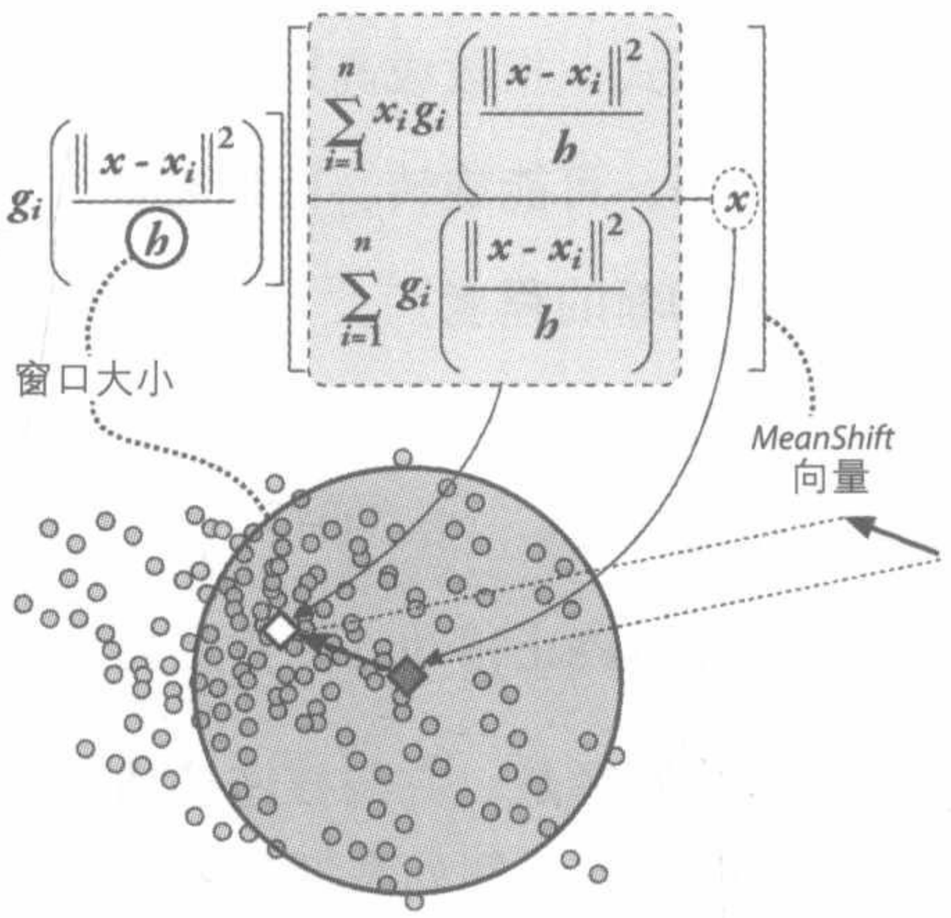


图17-3. Mean-Shift方程及其含义

在这种情况下，Mean-Shift向量使Mean-Shift窗口更新为以在该窗口内的计算中心为中心。当然，这个动作会改变窗口中的“下”，所以我们重复这个更新中心的过程。这种更新将总是收敛于Mean-Shift向量θ（即不可能再有更多的居中运动）。收敛位置位于窗口下方分布的局部最大值（峰值）处。不同的窗口大小会发现不同的峰值，因为“峰值”基本上是一个尺度敏感的结构。

在图17-4中，我们看到一个二维分布数据和一个初始（在这种情况下是矩形）窗口的例子。箭头表示分布中局部模式（峰值）收敛的过程。可以看到，像之前说的那样，这个峰值发现器在统计学上是鲁棒的，因为Mean-Shift窗口之外的点不会影响收敛——该算法不会被离群点“分心”。

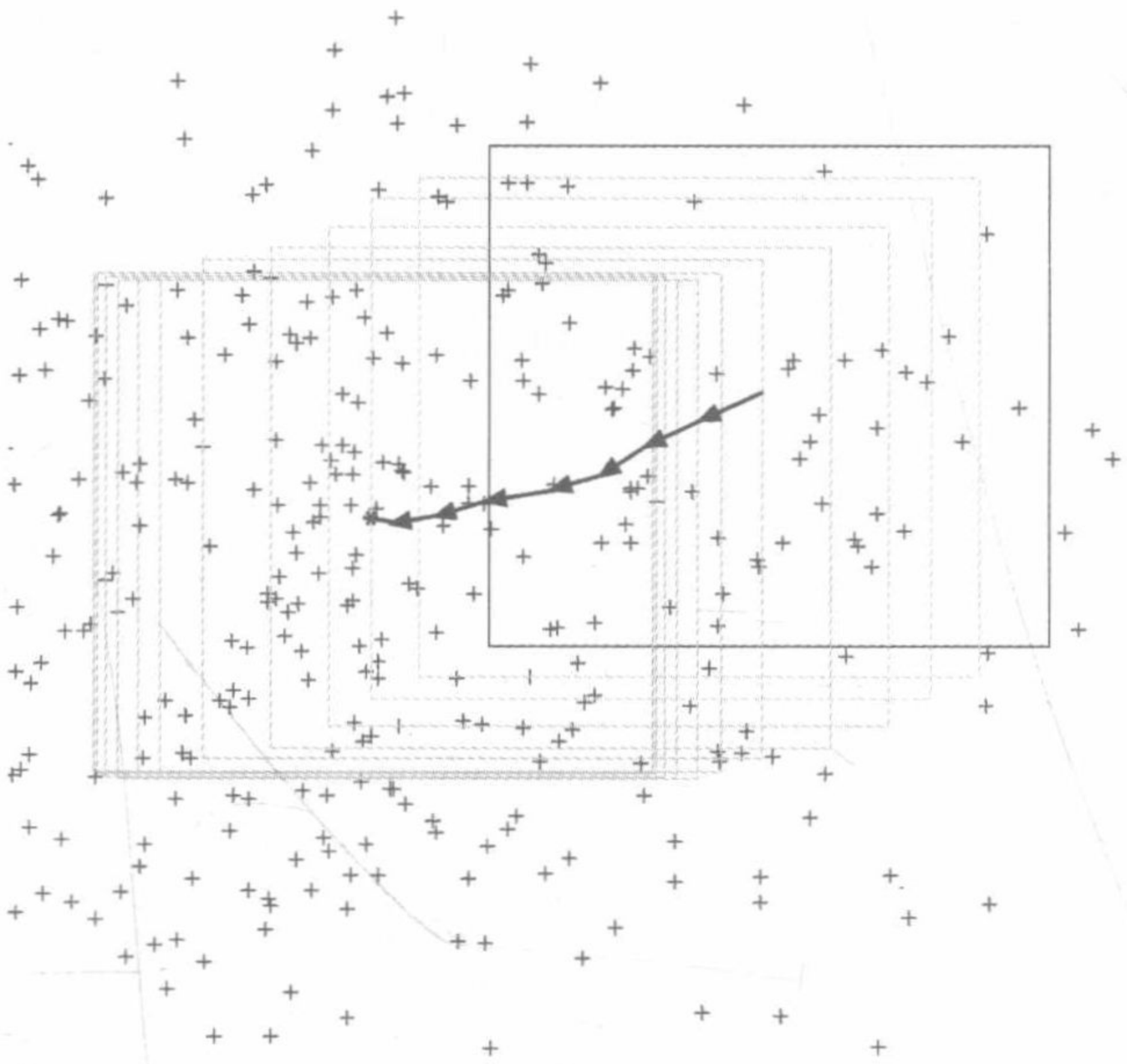


图17-4. Mean-Shift算法执行过程：将初始窗口放置在数据点的二维阵列上，并且在其数据分布的模式（或局部峰值）上依次更新中心直到收敛

1998年，这种模式搜索算法被发现可以用于跟踪视频中的移动物体[Bradski98a; Bradski98b]，并且算法已经大大扩展[Comaniciu03]。OpenCV执行Mean-Shift的函数是在图像分析中实现的。这意味着，Mean-Shift的OpenCV实现不是输入一些任意数据的点集（可能是任意维度数），而是以图像的形式输入正在进行分析的密度分布。可以将此图像看作是一个二维空间中的点密度的二维直方图。事实证明，对于视觉来说，这正是你大部分时间想要做的：跟踪一组感兴趣的特征的运动。

```
int cv::meanShift(                  // Return number of iterations to converge
    cv::InputArray    problmage,  // density of locations (CV_8U or CV_32F)
    cv::Rect&         window,     // initial location (and size) of kernel window
    cv::TermCriteria  criteria    // limits location update iterations
);
```

在cv::meanShift()中，problmage（表示可能位置的密度概率）可能是单通道，但可以是任一类型（byte型或float型）。window设置内核窗口的初始位置和大小。完成后，它将包含内核窗口的最终位置（其大小将保持不变）。criteria设置了Mean-Shift位置更新迭代次数的最大限制以及我们认为窗口位置收敛的最小移动阈值。[注14]返回值将是收敛前的迭代次数。

注14：依然，Mean-Shift将总是收敛的，但如果分布在那里相当“平坦”，则在分布的局部峰值附近的收敛可能非常慢。

函数cv::meanShift()是矩形窗口Mean-Shift算法的一种形式，但也可以用于跟踪。在这种情况下，首先选择特征分布来表示对象（例如，颜色+纹理），然后通过对象生成的特征分布开始Mean-Shift窗口，最后在下一个视频帧中计算选定的特征分布。从当前窗口位置开始，Mean-Shift算法将找到特征分布的新峰值或模式，（可能）首先位于生成颜色和纹理的对象的中心。通过这种方式，Mean-Shift窗口可以实现逐帧跟踪对象的移动。

Camshift

一种相关算法是Camshift[注15]跟踪器。它与Mean-Shift算法的主要不同之处在于搜索窗口可以调整自身大小。如果有良好的分段分布（例如，保持紧凑的面部特征），举个例子，当人们离相机更近或更远时，该算法会自动调整跟踪面部窗口尺寸的大小。实现Camshift算法的函数如下：

```
RotatedRect cv::CamShift(          // Return final window size and rotation
    cv::InputArray    probImage, // Density of locations (CV_8U or CV_32F)
    cv::Rect&         window,    // Initial location (and size) of kernel window
    cv::TermCriteria  criteria   // Limits location update iterations
);
```

cv:CamShift()函数使用的三个参数与cv::meanShift()算法意思相同。返回值将包含重新调整大小的框，还包括通过二阶矩计算的目标的方向。对于跟踪应用程序，我们将使用前一帧中生成的重新调整大小的框作为下一帧中的window。



许多人认为Mean-Shift算法和Camshift算法使用颜色特征进行跟踪，但这并不完全正确。这两种算法跟踪在probImage中表达的任何种类的特征的分布，因此，它们都是非常轻便、健壮和高效的跟踪器。

运动模板

运动模板法是由Bobick和Davis在麻省理工学院媒体实验室发明的[Bobick96；Davis97]，并与本书作者之一[Davis99; Bradski00]进行了进一步改进。这个更新的工作构成了在OpenCV中实现的基础。

注15：这里你可能觉得好奇，原始算法被命名为CAMSHIFT（所有大写字母），但不是首字母缩略词。因为Mean-Shift算法（以它为基础）是连字符，所以通常看到“cam-shift”，即使这是不正确的。OpenCV库中的名称cv::CamShift实际上是从不正确的连字符中导出的。因为算法名称不是首字母缩略词，所以将它看作Camshift（包括其作者）也是常见的，这也是我们在这里使用该表示的原因。

运动模板是跟踪一般运动的有效方式，尤其适用于手势识别。使用运动模板需要一个目标的轮廓（或轮廓的一部分）。目标轮廓可以通过多种方式获得。

- 获得目标轮廓最简单的方法是使用合适的静态相机，然后采用帧到帧的差分（如第15章所述）。这将产生目标的移动边缘，足以使运动模板工作。
- 也可以使用色度键控。例如，如果有一个已知的背景颜色，如亮绿色，你可以简单地将不是亮绿色的任何东西作为前景。
- 像第15章所介绍的，你可以学习一个背景模型，从中将新的前景物体/人物轮廓分隔出来。
- 可以使用动态的轮廓技术——例如，创建一个布满近红外光线的墙壁，并使近红外线敏感摄像机监控墙壁，任何在其中的目标都将显示为轮廓。
- 可以使用热像仪，任何有热辐射的目标（如脸部）都可以作为前景。
- 最后，你可以通过使用第15章中描述的分割技术（例如，金字塔分割或Mean-Shift分割）来生成轮廓。

现在，假设我们有一个好的分割目标的轮廓，如图17-5(A)的白色矩形所示。这里我们使用白色表示所有像素都被设置为包含最近系统时间戳的浮点值。当矩形移动时，新轮廓被捕获并用（新）当前时间戳覆盖，新轮廓是图17-5(B)和17-5(C)的白色矩形。较老的运动在图17-5(C)中显示为连续较暗的矩形。这些依次变暗的轮廓记录了先前运动的历史，因此被称为“运动历史图像”。

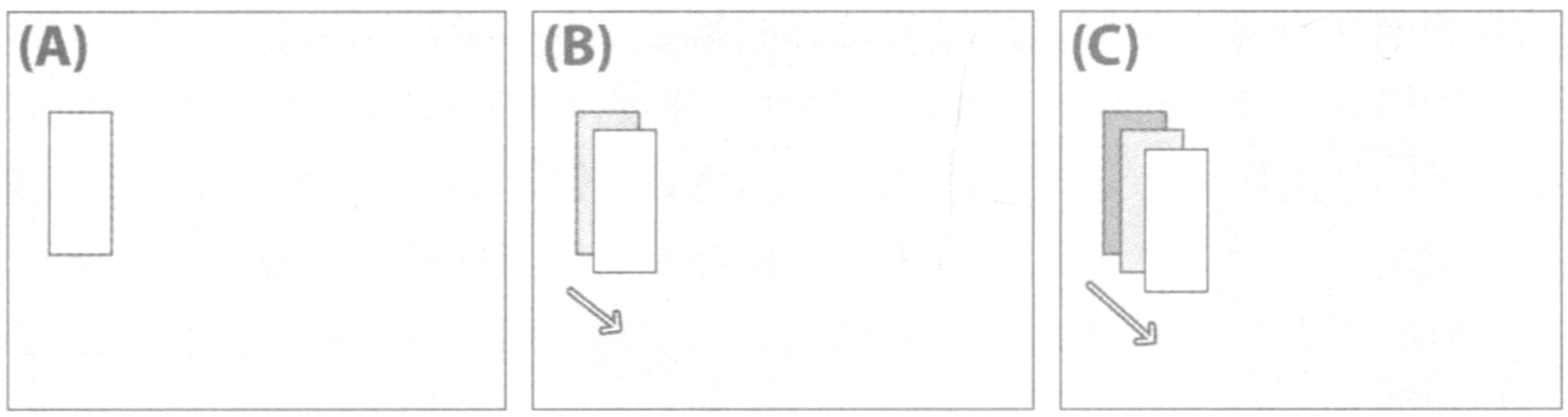


图17-5. 运动模板图：(A)当前时间戳（白色）的分割目标；(B)在下一个时步，目标移动并用（新）当前时间戳标记，留下旧的分割边界；(C)在下一步中，目标移动得更远，将较旧的分割留作连续较暗的矩形，编码运动的序列产生运动历史图像

时间戳超过当前系统时间戳的指定duration（持续时间）的轮廓被设为0，如图17-6所示。完成这个运动模板构建的OpenCV函数是cv::motempl::updateMotionHistory()。[注16]

注16：从OpenCV 3.0开始，运动模板函数（在本节中介绍）将被移到Opencv_contrib仓库的optflow模块下，使用motempl命名空间。

```
void cv::zmotemplz::updateMotionHistory(
   cv::InputArray          silhouette, // Nonzero pixels where motion occurs
   cv::InputeOutputArray   mhi,        // Motion history image
   double                  timestamp,  // Current time (usually milliseconds)
   double                  duration    // Max track duration ('timestamp' units)
);
```

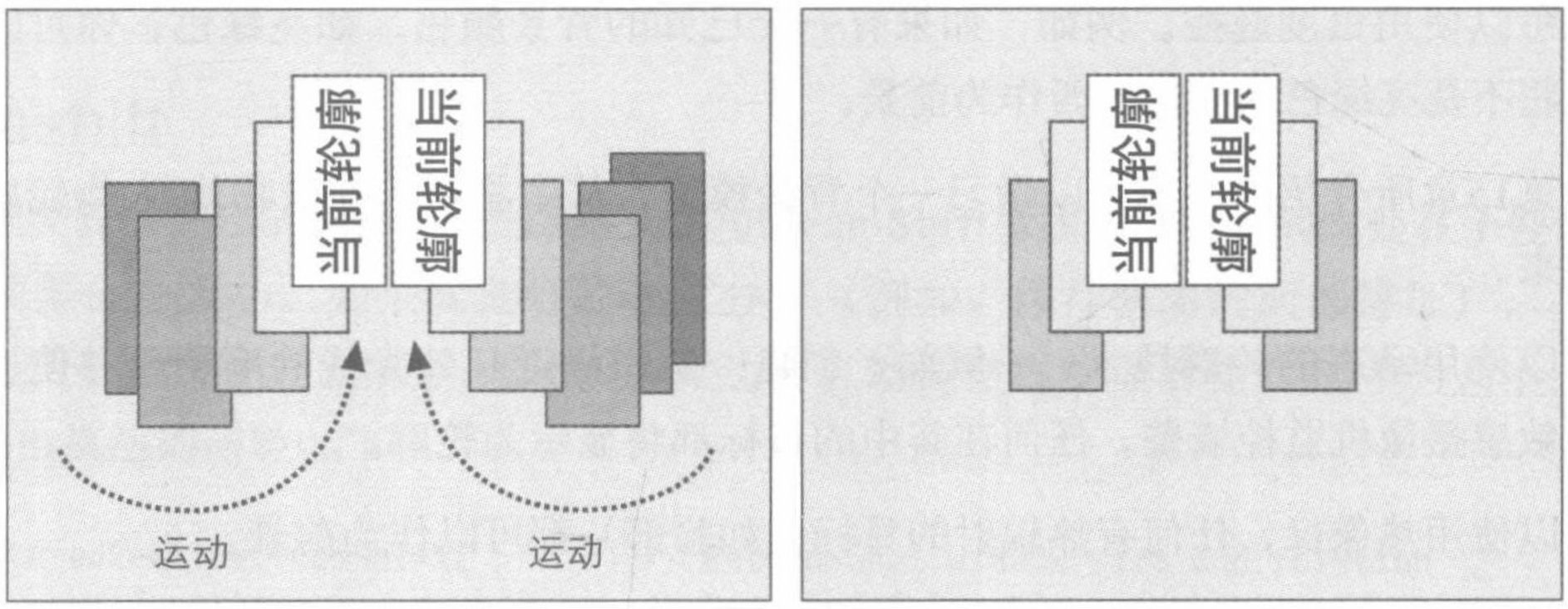


图17-6. 两个移动物体的运动模板轮廓（左图）；超出指定持续时间的轮廓设置为0（右图）

在cv::motempl::updateMotionHistory()中，所有的图像数组由单通道图像组成。轮廓图像是一个字节图像，其中非零像素表示前景目标的最新分割轮廓。参数mhi是表示运动模板（a.k.a.运动历史图像）的浮点图像。这里，timestamp是当前系统时间（通常为毫秒计数）和duration（如刚才所述，设置的允许运动历史像素保留在mhi中的时间长度）。换句话说，mhi中比timestamp减去duration更老（少于）的像素均被设为0。

一旦运动模板有了覆盖在时间上的对象轮廓的集合，我们就可以通过求mhi的梯度来导出整体运动的情况。当我们求这些梯度（例如，通过使用第12章中讨论的Scharr或Sobel梯度函数）时，一些梯度将是大而无效的。当mhi的较旧或非运动部分设为0时，梯度无效，这会在轮廓的外边缘周围产生人为的大梯度如图17-6所示。因为现在已经知道了我们通过cv::motempl::updateMotionHistory()）引入新轮廓的时间段，所以也就可以知道梯度（仅仅是dx和dy步骤的派生）应该是多大。因此，我们可以使用梯度幅度来消除太大的梯度，如图17-6所示。最后，我们可以收集全局运动的测量，如图17-6所示。实现函数是cv::motempl::calcMotionGradient()：

```
void cv::zmotemplz::calcMotionGradient(
   cv::InputArray    mhi,              // Motion history image
   cv::OutputArray   mask,             // Nonzero where valid gradients were found
   cv::OutputArray   orientation,      // Orientation of found gradients
   double            delta1,           // Minimal gradient allowed
   double            delta2,           // Maximal gradient allowed
   int               apertureSize = 3 // Size of gradient operator ('-1'=SCHARR)
);
```

在cv::motempl::calcMotionGradient()中，所有的图像数组都是单通道的。参数mhi必须是浮点运动历史图像。输入变量delta1和delta2分别为允许的最小和最大梯度幅度。预期的梯度幅度将只是在连续调用cv::motempl::updateMotionHistory()的每个轮廓之间的时间戳的平均毫秒数。将delta1设置为其预期梯度幅度一半以上（当然，还要比预期幅度小），delta2取在该平均值以上，通常运行效果都很好。变量apertureSize设置梯度运算符的宽度和高度的大小。该值可以设置为-1（3×3 cv::SCHARR梯度滤波器）、1（简单的两点中心差分导数）、3（默认3×3 Sobel滤波器）、5（5×5 Sobel滤波器）或7（7×7滤波器）。函数结果在mask和orientation中。前者将是一个单通道8位图像，其中非零项表示发现有效梯度的位置，后者将是一个浮点图像，给出了每个点的梯度方向角度。orientation以度为单位，范围在0到360之间。

函数cv::motempl::calcGlobalOrientation()找到整体运动方向作为有效梯度方向的向量和：

```
double cv::motempl::calcGlobalOrientation(
   cv::InputArray orientation, // Orientation image from calcMotionGradient()
   cv::InputArray mask,        // Nonzero where direction is to be calculated
   cv::InputArray mhi,         // Motion history img from updateMotionHistory()
   double timestamp,           // Current time (usually milliseconds)
   double duration             // Maximum duration of track ('timestamp' units)
);
```

当使用cv::motempl::calcGlobalOrientation()时，我们从cv::motempl::updateMotionHistory()中传递cv::motempl::calcMotionGradient()中计算的orientation和mask图像以及timestamp，duration和结果mhi。如图17-7所示，返回向量和全局方向。此方向将以度数为单位，范围在0到360之间。

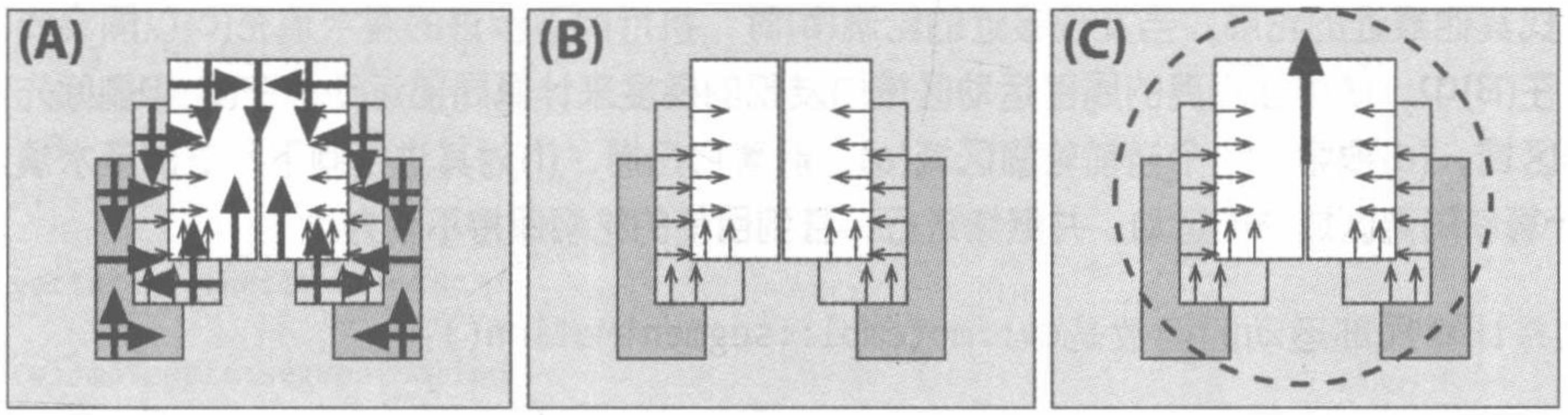


图17-7. MHI图像的运动梯度：(A)梯度大小和方向；(B)消除大梯度；(C)发现运动的整体方向

timestamp与duration一起告诉函数在mhi和运动orientation图像上要计算多长时间内的运动。可以从每个mhi轮廓的质心计算全局运动，但对预先计算的运动矢量进行求和要快得多。

我们还可以隔离运动模板mhi的区域并用该区域确定局部运动，如图17-8所示。在图中，

mhi是目前的轮廓区域的扫描。当找到标有最新时间戳的区域时，会在该区域的外围搜索足够近的运动（最近的轮廓）。当发现这种运动时，执行向下步进的漫水填充以隔离局部运动区域“溢出”目标对象的位置。一旦找到，我们可以计算溢出区域中的局部运动梯度方向，然后移除该区域，并重复该过程，直到找到所有区域（如图17-8所示）。

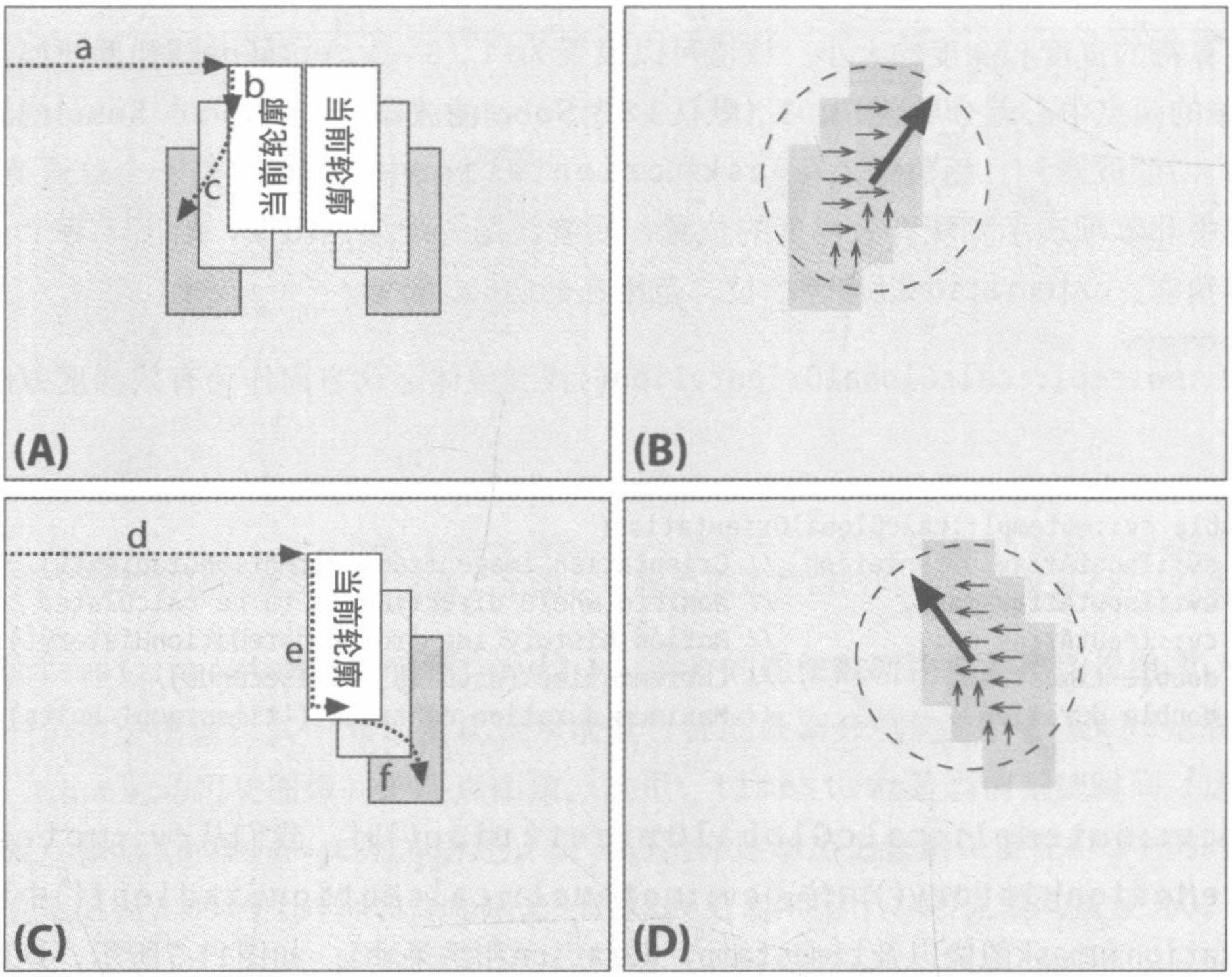


图17-8. 在MHI中划分局部运动区域。在(A)(a)中扫描MHI当前的轮廓，当发现它们时，围绕周边寻找其他最近的轮廓；当发现最近的轮廓(b)时，执行向下步进的漫水填充(c)以隔离局部运动。在(B)中，使用在隔离的局部运动区域内发现的梯度来计算局部运动。在(C)中删除先前找到的区域，(d)搜索下一个当前轮廓区域(e)，沿着它扫描，(f)对其进行向下步进的漫水填充。(D)计算新隔离区域内的运动，并继续进行，直到目前的轮廓保持不变为止

隔离并计算局部运动的函数是cv::motempl::segmentMotion()：

```
void cv::motempl::segmentMotion(
   cv::InputArray    mhi,            // Motion history image
   cv::OutputArray   segMask,        // Output image, found segments (CV_32FC1)
   vector<cv::Rect>& boundingRects, // ROI's for motion connected components
   double            timestamp,     // Current time (usually milliseconds)
   double            segThresh      // >= interval between motion history steps
);
```

在cv::motempl::segmentMotion()，mhi必须是单通道的浮点输入。参数segMask用作输出；返回一个单通道32位的浮点图像。各片段将被“标记”到该图像上，其中每个片段

被赋予不同的非零标识符（例如，1,2等等；零（0）表示“无运动”）。类似地，向量boundingRects将被感兴趣区域（ROI）填充，来连接运动部分。这允许你在每个这样连接的部分分别使用cv::motempl::calcGlobalOrientation来确定该特定部分的运动。

timestamp输入应该是mhi中（要从中分割局部运动的mhi）最新的轮廓的值。最后一个参数是segThresh，它是可被接受为附加动作的最大向下步长阈值（从当前时间到上一个运动）。提供这个参数是因为有些（你不想连接在一起的）当前和以前的运动轮廓可能会出现重叠。一般最好是把segThresh设置为轮廓时间戳平均差的1.5倍。

讨论到现在，你应该有能力理解在*opencv_contrib/modules/optflow/samples/directory*附带OpenCV的*motempl.cpp*示例。现在我们将从*motempl.cpp*中的update_mhi()函数中提取和解释一些关键的地方。update_mhi()函数通过阈值化帧差提取模板，然后将生成的轮廓传递给cv::motempl::updateMotionHistory()。

```
...
cv::absdiff( buf[idx1], buf[idx2], silh );

cv::threshold( silh, silh, diff_threshold, 1, cv::THRESH_BINARY );

cv::updateMotionHistory( silh, mhi, timestamp, MHI_DURATION );
...
```

然后得到生成mhi的梯度，并且通过cv::motempl::calcMotionGradient()产生了有效梯度的掩膜。然后生成的局部运动被分割到cv::Rect结构中：

```
...
cv::motemplz:calcMotionGradient(
  mhi,
  mask,
  orient,
  MAX_TIME_DELTA,
  MIN_TIME_DELTA,
  3
);

vector<cv::Rect> brects;

cv::motempl::segmentMotion(
  mhi,
  segmask,
  brects,
  timestamp,
  MAX_TIME_DELTA
);
...
```

然后，通过for循环遍历每个运动的外接矩形。迭代从-1开始，它已被指定为查找图像的全局运动的特殊情况。对于局部运动片段，首先滤除小区域，然后通过

cv::motempl:calcGlobal Orientation()计算出方向。这个例程不使用精确的掩码，而是把运动计算限制在包围局部运动的ROI中；然后计算已实际发现的局部ROI内的有效运动。任何太小的运动区域都被滤除。最后，例程画出运动。图17-9是一个挥舞手臂的人的输出示例，其中输出在原始图像上方绘制8个连续帧（排列成两行）。（相关的完整代码，请参阅*opencv_contrib*（如附录B所述），如果下载了*opencv_contrib*，则代码位于*.../opencv_contrib/modules/optflow/samples/motempl.cpp*中。）尽管形状识别不包括在样本代码中，这里还是按照相同的顺序，根据第14章中讨论的形状描述符（Hu矩）来对人的“Y”形姿势进行识别。

```
...
  for( i = -1; i < (int)brects.size(); i++ ) {

    cv::Rect roi; Scalar color; double magnitude;
    cv::Mat maski = mask;
    if( i < 0 ) {

      // case of the whole image
      //
      roi = Rect( 0, 0, img.cols, img.rows );
      color = Scalar::all( 255 );
      magnitude = 100;

    } else {

      // i-th motion component
      //
      roi = brects[i];
      if( roi.area() < 3000 ) continue; // reject very small components
      color = Scalar( 0, 0, 255 );
      magnitude = 30;
      maski = mask(roi);

    }

    double angle = cvzzmotemplz:calcGlobalOrientation(
      orient(roi),
      maski,
      mhi(roi),
      timestamp,
      MHI_DURATION
    );

    // ...[find regions of valid motion]...

    // ...[reset ROI regions]...

    // ...[skip small valid motion regions]...

    // ...[draw the motions]...
  }
...
```

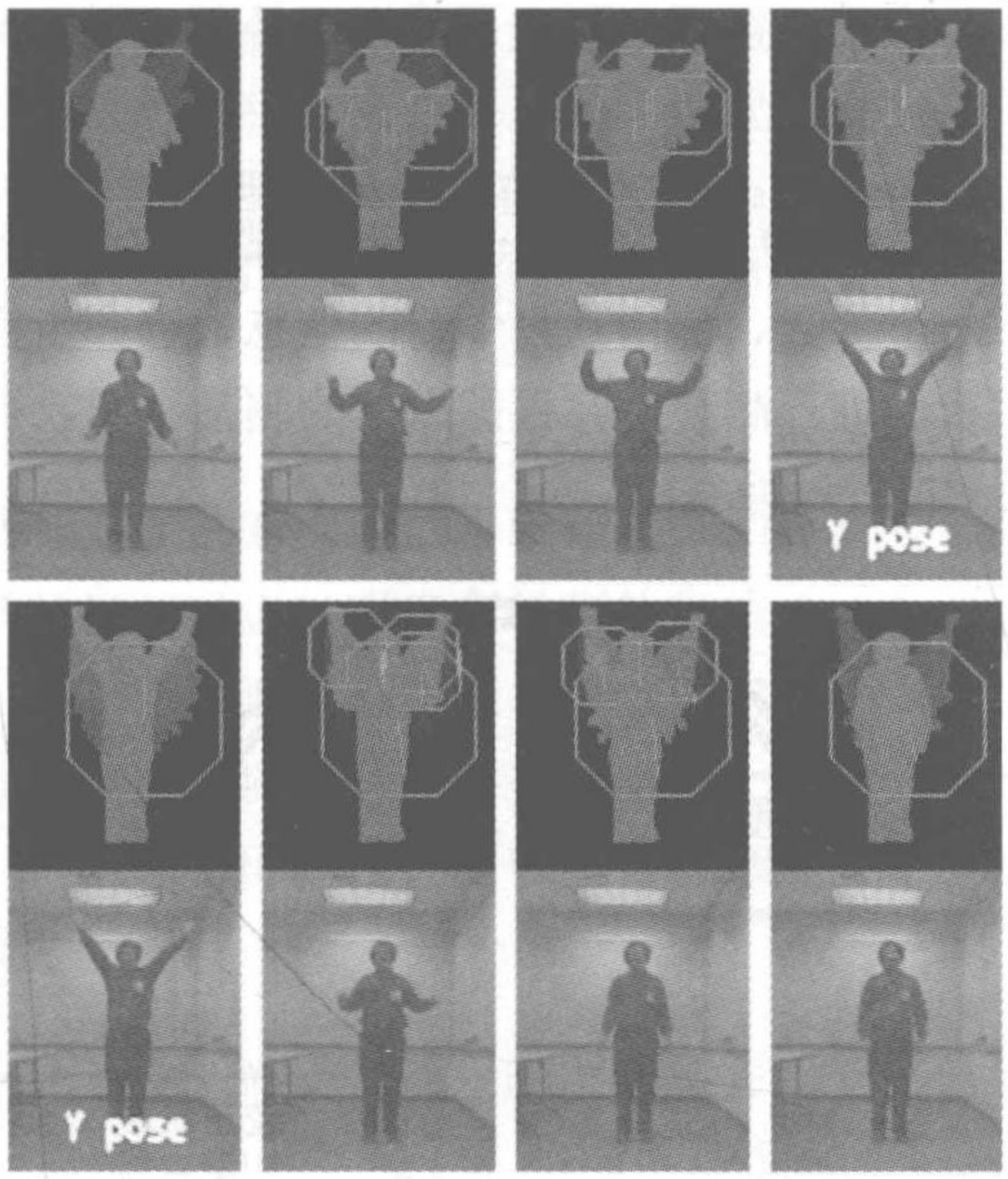


图17-9. 运动模板程序的结果：一个人的手臂上下挥舞，大八边形表示的所产生的全局运动和小八边形表示局部运动；也可以通过形状描述符（Hu 矩）来识别“Y”姿势

估计

假设我们正在跟踪一个在摄像机前走动的人。每一帧都要确定人的位置。像我们已经了解的那样，有很多种方式都可以做到。但其实所有这些重要的方法都属于我们对人的位置在每一帧上的估计。这个估计不太可能是非常准确的。原因有很多——例如，传感器误差，早期处理阶段的近似，由遮挡或阴影引起的问题，或由于人在行走时腿部和手臂摆动而产生的明显的形状变化。不管影响来源于哪里，我们都可以预计到这些测量值可能存在一个随机性的变化，这些值可能来自某些更理想的传感器或传感路径可能会收到的“实际值”。所有这些不准确的因素，在这里统称为“测量过程中的噪声”。

我们希望能够最大限度地利用所做的测量来估计这个人的动作。所以，多个测量的累积让我们检测出的人的观测轨迹中不受噪声影响的部分。一个小例子就是当我们知道这个人不动的时候。在这种情况下，我们的直觉告诉我们，我们可以对所有的测量结果求平均，以此来获得该人实际位置的最优估计。运动估计的问题涉及为什么我们的直觉给予一个固定的人这种解决方法，更重要的是，我们如何将方法推广运用于移动目标。

我们需要的关键附加成分是这个人的运动模型。例如，我们可以使用这样的陈述对人的运动进行建模：“一个人从一侧进入画面，以恒定的速度穿过画面。”给定这个模型，

我们不仅可查询这个人的位置，还能通过我们的观察确定模型的其他参数（比如在这种情况下，人的速度）。

这个任务分为两个阶段（图17-10）。在第一阶段，通常称之为“预测阶段”，我们为了估计人（或目标）的下一个位置，使用过去学习的信息，进一步完善我们的模型。在第二阶段，即校正阶段，我们进行测量，并根据以前的测量（即我们的模型）进行预测，最后将当前测量值和预测值进行融合。

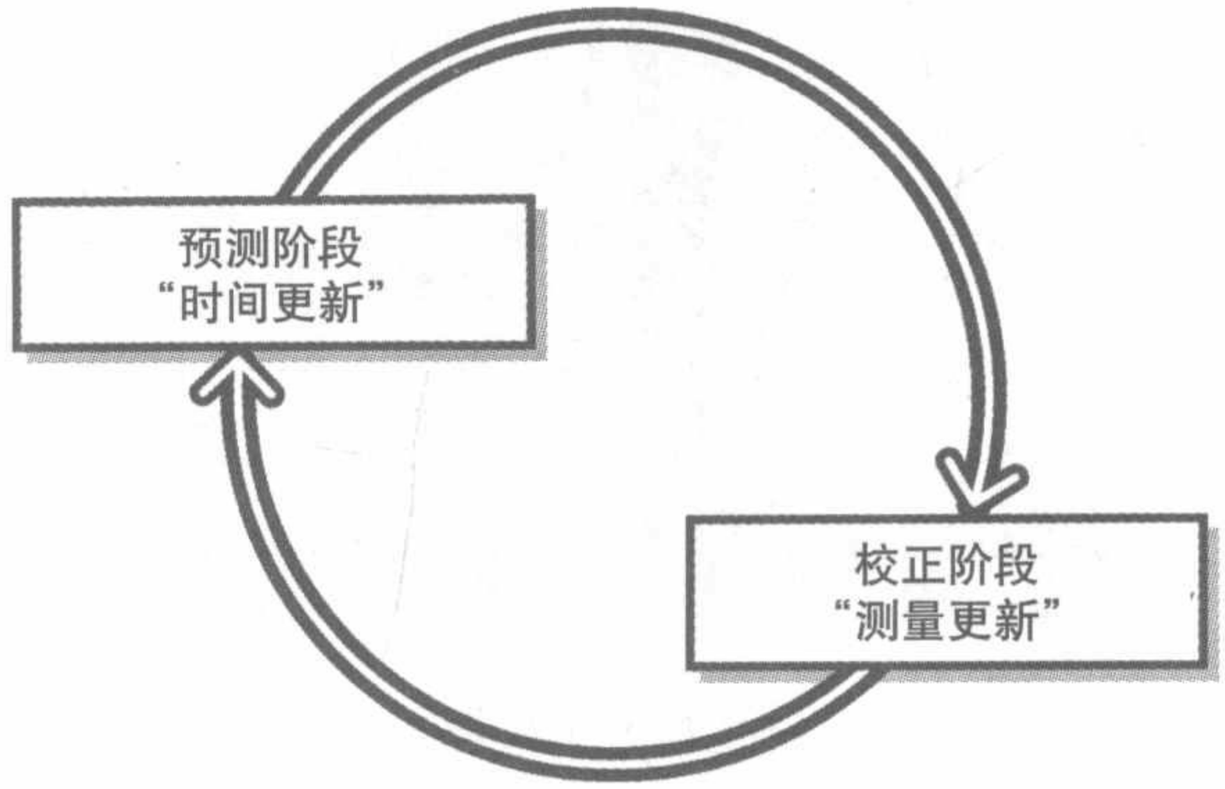


图17-10. 估计循环的两个阶段：基于先前数据进行预测，然后调整最新的测量

能够完成两阶段估计任务的机器通常都被冠以估计器（estimator）的名字，卡尔曼滤波器[Kalman60]是使用最广泛的技术。除了卡尔曼滤波器之外，另一个重要的方法是凝聚算法（condensation algorithm），这是更广泛的一种粒子滤波器在计算机视觉中实现。卡尔曼滤波器和凝聚算法的主要区别是如何描述状态概率密度。在下面的小节中，我们将详细探讨卡尔曼滤波器，然后接触一些相关技术。因为OpenCV库中没有实现它，所以我们不会介绍凝聚算法。然而，随着本书内容的推进，我们会接触到它和Kalman 滤波器在相关技术上的不同。

卡尔曼滤波器

在1960年提出之后，卡尔曼滤波器已经在各种信号处理领域中展示出其强大性。卡尔曼滤波器的基本思想是，在一个强而合理的假设条件下，[注17]有可能——给定系统历史测量情况下——为系统的当前状态建立模型，使模型的后验[注18]概率最大化。此外，事实

注17：“合理”的意思是“足够无限制，这种方法对现实世界中出现的各种实际问题有用”。“合理”听起来似乎语气不是很强烈。

注18：后验是“事后看来”的学术术语。因此，当我们说这样一个分布“最大化：后验概率”时，我们的意思是，这个分布本质上是对“实际发生的东西” 一个可能的解释，实际上是我们观察到（即回顾这一切）的数据中最有可能的分布。

证明，我们在不保留漫长历史测量的情况下也可以进行这个估计，反之，由于所做测量是连续的，我们迭代更新系统的状态模型并只保留最近的模型以供下一次迭代使用，这大大简化了这种方法的计算机实现。我们将在这里对Kalman滤波进行基本的介绍；对于更详细（但仍然很容易理解）的介绍，请参阅Welsh和Bishop[Welsh95]。

输入和输出

卡尔曼滤波器是一个估计器。这意味着它可以帮助我们整合关于系统状态的信息、关于系统动态的信息以及系统运行时通过观察所学到的新信息。在实践中，我们对每件事的表达都有重要的限制。第一个重要的限制是，卡尔曼滤波器只能在状态可以用向量或矩阵表示的系统上帮助我们，其中向量代表每个自由度上的当前值，矩阵代表每个自由度上的不确定性。这是一个矩阵，因为这种不确定性不仅包括个体自由度的差异，而且还包括它们之间可能的协方差。

状态向量可以包含我们认为与我们正在使用的系统相关的任何变量。在有个人走过图像的例子中，这个状态向量的分量可能是它们当前的位置和速度：$\vec{x}_i^* = (x^*, v_x^*)$。在某些情况下，对于位置和速度，可以有两个维度：$\vec{x}_i^* = (x^*, y^*, v_x^*, v_y^*)$。在复杂系统中，该向量可能有更多的分量。状态向量通常伴随着表示时间步长的下标；小的星号“*”表示我们正在谈论我们在特定时间的最优估计。[注19]状态向量伴随着协方差矩阵，通常写为$\sum_i^*$。如果$\vec{x}_i^*$具有n个元素，则该矩阵将是$n \times n$矩阵。

卡尔曼滤波器允许我们从状态$(\vec{x}_0^*, \Sigma_0^*)$开始，并估计后续状态$(\vec{x}_i^*, \Sigma_i^*)$。关于这个说法，有两个非常重要的意见。第一个是这些状态中的每一个都有相同的形式；也就是说，它表达为一种平均值，它代表我们认为系统最可能的状态以及表示我们对这个均值的不确定性的协方差。这意味着只有某些种类的状态可以用卡尔曼滤波器处理。例如，想象一辆正在行驶的汽车。使用卡尔曼滤波器对这样的系统进行建模是完全合理的。但是考虑到这辆汽车在道路上转向的情况。它可以向右或向左走，但不能向前。如果我们还不知道汽车转向哪个方向，就不适合用卡尔曼滤波器。这是因为由滤波器表示的状态是高斯分布，基本上是中间有峰值的分布。在转向的情况下，这辆车很可能已经向右走了，或者很可能已经向左走了，但不太可能直走并坠毁。不幸的是，任何赋予右侧和左侧相等权重的高斯分布必须给予中间更大的权重。这是因为高斯分布是单峰的。[注20]

注19：许多作者将使用下标而不是“*”。我们使用“*”，因为它让向量在视觉上看起来不会使“–”和“→”发生混淆。

注20：前面提到的粒子滤波器或凝聚算法主要是一种描述状态的替代方法，使得状态不需要是单峰的，甚至不是高斯的。

第二个重要的假设是，系统必须已经在可以用这样一个高斯分布描述的状态下开始了。这种状态（写作 $(\vec{x}_0^*, \Sigma_0^*)$）称为“先验分布”（注意零下标）。要知道，一个所谓“无信息先验分布”系统总有可能存在。该系统中平均值是任意的，并且协方差的值被认为很大——这基本上表示我们“什么都不知道”，但重要的是先验必须是高斯的，我们必须总得有一个。

卡尔曼滤波器所需的假设

在介绍使用卡尔曼滤波器的细节之前，让我们来看一下前面提到的“强而合理”的假设。在卡尔曼滤波器的理论构建中需要三个重要的假设：(1)建模的系统是线性的；(2)测量的噪声是“白”的；(3)这个噪声本质上是高斯的。第一个假设意味着（实际上）时间k处的系统的状态可以表示为向量，并且在时间$k + 1$（也是向量）的时刻的系统的状态可以表示为时间k处的状态乘以一些矩阵F。另一个假设，噪声是白色和高斯的，意味着系统中的任何噪声在时间上都不相关，并且其幅值可以仅用平均和协方差来精确建模（即，该噪声被其一阶矩和二阶矩完全描述）。虽然这些假设可能看起来有限制，但实际上它们对一般情况却出奇地适用。

“给出先前测量值后，最大化后验概率”是什么意思？ 这意味着在考虑到以前的模型及不确定性和新的测量与不确定性之后，我们构建的新模型是具有最大可能性的模型。出于我们的目的，考虑到三个假设，这意味着卡尔曼滤波器是在不同时间组合来自不同来源或相同来源的数据的最佳方式。从已知的信息开始，我们获得新的信息，然后根据旧的和新的信息加权组合，决定改变我们所知的关于旧信息和新信息的知识。

接下来让我们学习一下卡尔曼滤波器在一维运动场景中的应用。这会用到一些数学知识，当然你完全可以跳过下一节。但是线性系统和高斯如此友好，如果你连试都不试一下的话，卡尔曼博士可能会不高兴。

信息融合

那么卡尔曼滤波器的核心是什么？信息融合。假设你想知道某一点在一条线上的位置——一个简单的一维场景。[注21]由于噪声的存在，假设对于目标位置，你有两个（在高斯意义上）不可靠的值：位置x_1和x_2。因为这些测量中存在高斯不确定性，所以它们具有$\bar{x}_1$和$\bar{x}_2$以及标准偏差σ_1和σ_2的平均值。事实上，标准偏差实际上表示了我们测量好坏的不确定性。对于每个测量，作为位置函数的隐含概率分布是高斯分布：[注22]

注21：对于跟随类似轨迹的更详细的解释，请参考J. D. Schutter, J. De Geeter, T. Lefebvre和H. Bruyninckx的“Kalman Filters: A Tutorial”。

注22：符号$N(\bar{x}, \sigma^2)$是高斯分布或正态分布的常见简写，意思是“平均值和方差σ_2的正态分布”。我们将使用符号$N(x; \bar{x}, \sigma^2)$来显式描述恰好是正态分布的x的函数。

$$p_i(x) = N(x;\, \bar{x}_i,\, \sigma_i^2) \equiv \frac{1}{\sigma_i\sqrt{2\pi}}\exp\left(-\frac{(x - x_i)^2}{2\sigma_i^2}\right),\ (i = 1,\, 2)$$

给定两个这样的测量，每个都服从高斯分布，我们预期给定这样两个测量的x值的概率密度将与$p_{12}(x) = p_1(x) \cdot p_2(x)$成比例。事实证明，这个乘积是也服从高斯分布，我们可以计算这个新分布的平均值和标准偏差如下。

$$p_{12}(x) \propto \exp\left(-\frac{(x - x_1)^2}{2\sigma_1^2}\right)\exp\left(-\frac{(x - x_2)^2}{2\sigma_2^2}\right) = \exp\left(-\frac{(x - x_1)^2}{2\sigma_1^2} - \frac{(x - x_2)^2}{2\sigma_2^2}\right)$$

鉴于高斯分布的平均值是其最大值，我们可以简单通过计算$p(x)$对x的导数来找到平均值。函数最大时，其导数为0，所以有：

$$\left.\frac{d\,p_{12}}{dx}\right|_{x_{12}} = -\left[\frac{x_{12} - x_1}{\sigma_1^2} + \frac{x_{12} - x_2}{\sigma_2^2}\right] \cdot p_{12}(\bar{x}_{12}) = 0$$

由于概率分布函数$p(x)$从不为0，因此括号中的项肯定为0。求解x的等式可以得出这个非常重要的关系式：

$$\bar{x}_{12} = \left(\frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2}\right)\bar{x}_1 + \left(\frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}\right)\bar{x}_2$$

因此，新的平均值$\bar{x}_{12}$只是两个测量平均值的加权组合。然而，至关重要的是，权重是由（而且只由）两个测量的相对不确定性决定的。例如，可以观察到，如果第二次测量的不确定度σ_2越大，那么新的平均值将与平均值x_1基本上相同，前一次测量的确定性更多。

使用新的均值$\bar{x}_{12}$，并且在重新安排后（重新安排有点混乱），我们可以将此值替换进$p_{12}(x)$的表达式。[注23]识别不确定度σ_{12}^2为：

$$\sigma_{12}^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}.$$

在这一点上，你可能想知道这告诉我们什么。其实它告诉我们很多。当我们用新的平均值和不确定性进行新的测量时，可以将该测量与我们已经获得的新状态的平均值和不确定性相结合，该状态的特性在于更新的平均值和不确定性。我们现在也有这些东西的数学表达式，这将会派上用场。

注23：如果要验证所有这一切，要完成三步：(1)从和σ12项入手开始重排高斯分布p12(x)的等式；(2)把等式中和$\bar{x}_{12}$相关的替换成$\bar{x}_1$和$\bar{x}_2$，与σ_{12}有关的替换成σ_1和σ_2；(3)验证结果可以拆分成最开始的高斯的乘积。

这两个高斯测量结合时相当于单个高斯测量（具有可计算的平均值和不确定度）的性质是我们最重要的特征。这意味着当我们进行M次测量时，可以将前两个和第三个组合，然后前三个和第四个组合，等等。这是在计算机视觉中跟踪发生的事情：我们获得一个测量，接着是下一个。

把测量(x_i, σ_i)作为时间步长，我们可以计算估计的当前状态如下。首先，假设我们有一些初始的对象位置的信息，这是先验概率。这个初始信息将以平均值和不确定度x_0^*和σ_0^*为特征。回忆一下，我们说过用*来表示这是当前的估计，而不是当前的测量。

接下来，我们在时间步骤1得到我们的第一个测量(x_1, σ_1)。我们必须继续的是这个新的测量和以前的测量，但是我们可以将它们替换成我们的最优估计方程。我们的最优估计是把以前的状态估计（在这种情况下）即之前说的(x_0^*, σ_0^*)——与我们的新测量(x_1, σ_1)相结合的结果：

$$x_1^* = \left(\frac{\sigma_1^2}{\sigma_0^{*2} + \sigma_1^2}\right)x_0^* + \left(\frac{\sigma_0^{*2}}{\sigma_0^{*2} + \sigma_1^2}\right)x_1$$

重新排列这个方程式，得到下面这个有用的迭代形式：

$$x_1^* = x_0^* + \left(\frac{\sigma_0^{*2}}{\sigma_0^{*2} + \sigma_1^2}\right)(x_1 - x_0^*)$$

在担心这是否有用之前，我们还应该计算类似的方程式：

$$\sigma_1^{*2} = \frac{\sigma_0^{*2}\sigma_1^2}{\sigma_0^{*2} + \sigma_1^2}$$

和我们对x_1^*做的重新排列相似，一个生成了一个用于估计方差的迭代方程，给出了新的测量：

$$\sigma_1^{*2} = \left(1 - \frac{\sigma_0^{*2}}{\sigma_0^{*2} + \sigma_1^2}\right)\sigma_0^{*2}$$

从现在的形式来看，这些方程式使我们能够从“新”信息（我们最新的测量结果告诉我们的，不带*的变量）中清楚分离出“旧的”信息（我们在进行新的测量之前知道的，带*的变量）。在时间步长1看到的新信息$(x_1 - x_0^*)$被称为“创新”（innovation）。我们还可以看到，我们的最优迭代更新因子现在是：

$$K = \left(\frac{\sigma_0^{*2}}{\sigma_0^{*2} + \sigma_1^2}\right)$$

这个因子被称为“更新增益”（update gain）。使用这个K的定义，我们得到一个方便的递归形式。这是因为不管是到目前为止的推导，还是关于先验的分布，还是新的测量，

都没有什么特别之处。实际上，在我们的第一次测量之后，我们有一个新的状态估计(x_1^*, σ_1^*)，它相对于新的测量(x_2^*, σ_2^*)之间的函数关系与(x_0^*, σ_0^*)相对于(x_1^*, σ_1^*)的函数关系相同。实际上，(x_1^*, σ_1^*)是新的先验（即，在下一次测量到达之前的“先前认为”）。把这个实现放在方程式中就是：

$$x_k^* = x_{k-1}^* + K\left(x_k - x_{k-1}^*\right)$$

$$\sigma_k^{*2} = (1 - K)\sigma_{k-1}^{*2}$$

以及：

$$K = \left(\frac{\sigma_{k-1}^{*2}}{\sigma_{k-1}^{*2} + \sigma_k^2}\right)$$

在卡尔曼滤波器的文献中，如果讨论是关于一般的一系列测量，传统上将“当前”时步称为k，因此之前的时步是$k-1$（与$k+1$和k相反）。图17-11显示了此更新过程的剧烈变化和收敛于真实分布。

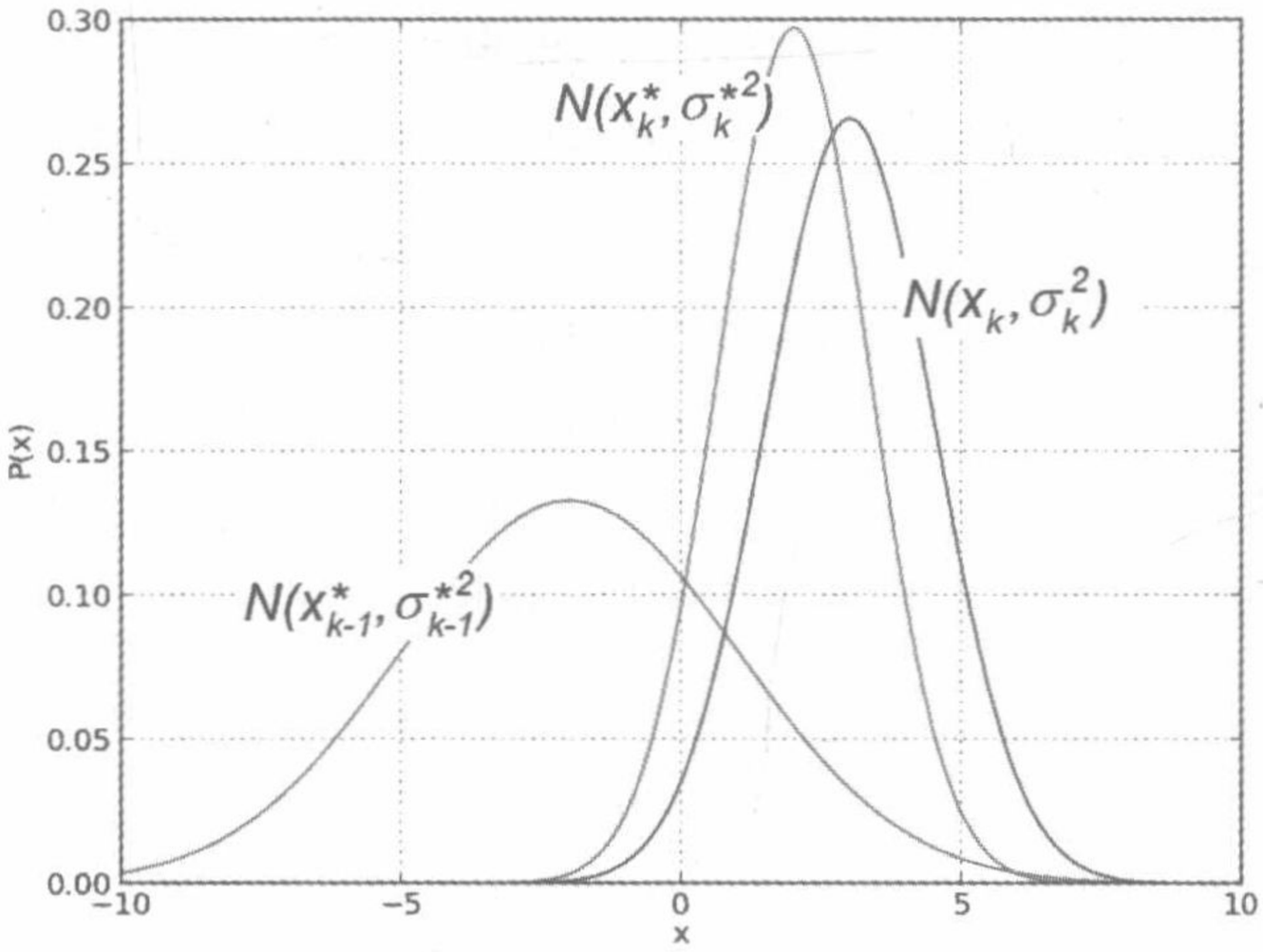


图17-11. 通过我们的测量观测$N(x_{k-1}^*, \sigma_{k-1}^{*2})$[左]结合我们的先验信念$N(x_k, \sigma_k^2)$[右]，结果是我们的新估计：$N(x_k^*, \sigma_k^{*2})$[中]

动态系统

在简单的一维示例中，我们考虑了一个目标对象位于某个点x的情况，以及该点的一系列连续测量。在这种情况下，我们没有专门考虑物体在实际测量中可能实际移动的情况。在这种新情况下，我们会有所谓的预测阶段。在预测阶段，我们使用自己所知道的来确定在尝试整合新测量之前期望的系统位置。

实际上，预测阶段是在进行新测量完成后，但在新测量结合到系统状态估计前进行的。比如当我们在时刻t测量一辆汽车的位置，然后在时刻$t + dt$再次测量。如果汽车有一定的速度v，那么我们不直接并入第二个测量。我们首先根据我们在时间t获得的信息快速推出模型，以便在新信息被纳入之前，我们不仅在时间t有一个模型，而且在时间$t + dt$的时候也有系统的模型。以这种方式，在时间$t + dt$获取的新信息不会与系统的旧模型融合，而是预测到时间$t + dt$的系统的旧模型融合。这是图17-10所示循环的含义。在卡尔曼滤波器的背景下，我们将考虑三种运动。

第一个是动态运动。这是我们预计的最后一次测量系统状态的直接结果。如果在t时刻测量系统在x位置以一定速度v运动，那么在时刻$t + dt$，我们预计系统位于$x + v \cdot dt$位置，速度可能仍然是v。

第二种运动形式称为“控制运动”。控制运动是我们希望的，因为某种已知的外部因素以某种原因作用在系统上。顾名思义，控制运动最常见的一个例子是，当我们估计一个受控系统的状态时，我们知道我们的控制会使系统产生什么样的运动。这种情况经常出现在机器人系统中，系统控制机器人，例如加速或向前。显然，在这种情况下，如果机器人处于位置x并且在时间t以速度v移动，则在时间$t + dt$，我们就预测机器人不是只移动到$x + v \cdot dt$（像它没被控制的时候那样），它走得更远，因为我们知道机器人要加速。

最后一类运动是随机运动。即使在简单的一维情形的示例中，如果观测的目标有因为任意原因自行移动的可能性，就需要在预测阶段包含这种随机运动。这种随机运动的结果是随着时间的推移增加状态估计的不确定性。随机运动包括任何未知或不受控的运动。然而，与卡尔曼滤波器框架中的其他任何东西一样，存在这样一个假设，即该随机运动是高斯（即，一种随机游走）或者至少可以有效建模为高斯。

我们在融合一个新的测量值前，要先进行“更新”以便将动态变化包含进仿真模型中。在更新阶段，首先根据其先前的状态在仿真模型中融入所有有关目标的运动信息，然后融入我们自己在系统上已经采取或从另一个外部代理程序产生的关于运动产生的任何附加信息，最后融合我们估计的可能改变系统状态的随机事件。一旦这些因素得到融合，我们就可以融合下一个新的测量值。

实际上，当系统的状态比我们的仿真模型更复杂时，动态运动就显得特别重要。通常，当物体移动时，状态矢量有多个分量，如位置和速度。在这种情况下，当然，状态根据我们认为拥有的速度而变化。下一节中，我们将讨论如何处理具有多个状态分量的系统。然而，为了处理这些新情况，我们还需要进一步学习一些数学知识。

Kalman方程

我们现在准备把所有东西放在一起，了解如何处理更一般的模式。这样的模型包括多维状态向量、动态运动、控制运动、随机运动、测量、测量不确定性以及将我们对系统所处位置的估计与我们随后测量的估计融合在一起所需的技术。

我们需要做的第一件事是将我们的讨论推广到包含许多状态变量的状态。最简单的例子或许是在视频跟踪的场景中，目标可以在两维或三维中移动。通常来说，状态可能包含其他元素，例如被跟踪目标的速度。我们将把状态在时刻k的描述推广为状态在时刻$k-1$的的以下函数：

$$\vec{x}_k = F\vec{x}_{k-1} + B\vec{u}_k + \vec{w}_k$$

这里$\vec{x}_k$现在是状态分量的n维矢量，F是$n \times n$矩阵，有时称为传递矩阵或转移矩阵，乘以$\vec{x}_{k-1}$。向量$\vec{u}_k$是新的。它让系统可以接受外部控制，它由一个称为“控制输入”的c维向量组成； B是将这些控制输入与状态变化相关联的$n \times c$矩阵。[注24]在实际应用中，这通常是第一个分离出的假设。变量$\vec{w}_k$是与直接影响系统实际状态的随机事件或力相关联的随机变量（通常称为“过程噪声”）。我们假设$\vec{w}_k$的分量对于一些$n \times n$协方差矩阵Q_k具有零均值高斯分布$N(\vec{0}, Q_k)$（原则上可随时间变化，但通常它并不会）。

一般来说，我们进行的测量$\vec{Z}_k$可能是也可能不是状态变量$\vec{X}_k$的直接测量。（例如，如果你想知道一辆汽车的移动速度，可以用雷达测量其速度，或者测量来自其尾管的声音；在前一种情况下，$\vec{Z}_k$将是$\vec{X}_k$加上一些附加的测量噪声，但在后一种情况下，这种关系并不这样直接）。我们可以由下式归纳这种情况。所测量的m维向量$\vec{Z}_k$：

$$\vec{z}_k = H\vec{x}_k + \vec{v}_k$$

这里H是$m \times n$矩阵，$\vec{v}_k$是测量误差，对于一些$m \times m$协方差矩阵R_k，[注25]也假定具有高斯分布$N(0, R_k)$。

在完全迷糊之前，让我们考虑一个在停车场测量一个行驶车辆的特殊现实情况。我们可以想象，汽车的状态可以通过两个位置变量x和y以及两个速度V_x和V_y来概括。这四个变量将是状态向量$\vec{X}_k$的元素。这表明F的正确形式是：

注24：聪明的读者或已经知道Kalman滤波器的人会注意到另一个重要的假设，我们在控制u_k和状态变化之间滑入——即（通过矩阵乘法）有一个线性关系。

注25：这些项中的k允许它们随时间变化，但现在不需要。在实际中，H和R不随时间变化是很常见的。

$$\vec{x}_k = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}, F = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

然而，当使用相机来测量汽车的状态时，我们可能只能测得位置变量：

$$\vec{z}_k = \begin{bmatrix} z_x \\ x_y \end{bmatrix}$$

这意味着H的结构类似于：

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

在这种情况下，我们可能不会相信汽车的速度是不变的，因此会分配一个Q_k的值来反映这一点。既然司机的行为不受我们控制，在某种意义上说，就可以从我们的角度把它视为是随机的。我们将根据我们对我们使用（例如）我们的图像分析技术在视频流上测量汽车位置的准确程度的估计来选择R_k。

现在剩的就是把这些表达式插入广义形式的更新方程。当然，基本思想是一样的。首先，我们计算状态的先验估计x_k^-。文献中使用上标减号代表“在新测量之前的时间”是相对普遍的（虽然不是通用的）， 我们也会在这里采用这个惯例。因此，先验估计由下式给出：

$$\vec{x}_k^- = F\vec{x}_{k-1} + B\vec{u}_{k-1} + \vec{w}_k$$

使用Σ_k^-来表示误差协方差，在时间k处的该协方差的先验估计从时间$k-1$处的值获得：

$$\Sigma_k^- = F\Sigma_{k-1}F^T + Q_{k-1}$$

这个方程式构成了估计预测部分的基础，并根据已知的内容告诉我们“我们想要的信息”。这里，我们定义（不再进行推导）Kalman增益或混合因子。它告诉我们如何根据已知的新信息加权：

$$K_k = \Sigma_k^- H_k^T \left(H_k \Sigma_k^- H_k^T + R_k\right)^{-1}$$

虽然这个方程式看起来有点儿吓人，但其实并不是很复杂。通过参考各种简单的情况，我们可以更容易地理解它。对于我们直接测量一个位置变量的一维示例来说，H_k只是一个只包含1的1×1矩阵。因此，如果我们的测量误差是σ_{k+1}^2，则Rk也是包含该值的1×1矩阵。类似地，Σ_k只是方差σ_k^2。所以大方程可以化为：

$$K_k = \frac{\sigma_k^2}{\sigma_k^2 + \sigma_{k+1}^2}$$

请注意，这正是我们认为的那样。我们之前在上一节中看到的增益允许我们在新的测量可用时优化计算σ_k^2和σ_k的更新值：

$$\vec{x}_k = \vec{x}_k^- + K_k(\vec{z}_k^- - H_k\vec{x}_k^-)$$

以及：

$$\Sigma_k = (I - K_k H_k)\Sigma_k^-$$

这些方程式乍一看同样很吓人，但是在我们简单的一维的场景讨论中，它并没有看起来那么复杂。除了将求解前的x的偏导数置为0来最小化位置状态x的不确定性外，这一次获得最优权重和增益的方法与一维情况相同。我们可以通过首先设置$F = I$（其中I是单位矩阵），$B = 0$和$Q = 0$来显示与更简单的一维情况的关系。然后，在更一般方程中进行以下替换，可以看出它和我们的一维滤波器求导的相似性：

$$\vec{x}_k \leftarrow \hat{x}_2$$

$$\vec{x}_k^- \leftarrow x_2$$

$$K_k \leftarrow K$$

$$\vec{z}_k \leftarrow x_2$$

$$H_k \leftarrow I$$

$$\Sigma_k \leftarrow \hat{\sigma}_2^2$$

$$\Sigma_k^- \leftarrow \hat{\sigma}_1^2$$

$$R_k \leftarrow \sigma_2^2$$

$$I \leftarrow 1$$

在OpenCV中使用cv::KalmanFilter 进行跟踪

有了所有这一切，你可能会觉得自己不需要OpenCV为自己做任何事情，或者你迫切需要OpenCV为你做所有这些事情。幸运的是，OpenCV两种需求都可以满足。卡尔曼滤波器在OpenCV中由一个名为`cv::KalmanFilter`的对象表示（并不奇怪）。`cv::KalmanFilter`对象的声明具有以下形式：

```
class cv::KalmanFilter {

public:
  cv::KalmanFilter();

  cv::KalmanFilter(
    int dynamParams,            // Dimensionality of state vector
    int measureParams,          // Dimensionality of measurement vector
    int controlParams = 0,      // Dimensionality of control vector
    int type          = CV_32F  // Type for matrices (CV_32F or F64)
  );

  //! re-initializes Kalman filter. The previous content is destroyed.
  void init(
    int dynamParams,            // Dimensionality of state vector
    int measureParams,          // Dimensionality of measurement vector
    int controlParams = 0,      // Dimensionality of control vector
    int type          = CV_32F  // Type for matrices (CV_32F or F64)
  );

  //! computes predicted state
  const cv::Mat& predict(
    const cv::Mat& control = cv::Mat() // External applied control vector (u_k)
  >;

  //! updates the predicted state from the measurement
  const cv::Mat & correct(
    const cv::Mat& measurement  // Measurement vector (z_k)
  );

  cv::Mat transitionMatrix;     // state transition matrix (F)
  cv::Mat controlMatrix;        // control matrix (B) (w/o control)
  cv::Mat measurementMatrix;    // measurement matrix (H)
  cv::Mat processNoiseCov;      // process noise covariance matrix (Q)
  cv::Mat measurementNoiseCov;  // measurement noise covariance matrix (R)

  cv::Mat statePre;           // predicted state (x'_k):
                              // x_k = F * x_(k-1) + B * u_k

  cv::Mat statePost;          // corrected state (x_k):
                              // x_k = x'_k + K_k * ( z_k - H * x'_k )

  cv::Mat errorCovPre;        // a-priori error estimate covariance matrix (P'_k):
                              // \Sigma'_k = F * \Sigma_(k-1) * FAt + Q

    cv::Mat gain;             // Kalman gain matrix (K_k):
                              // K_k = \Sigma'_k * H*t * inv(H*\Sigma'_k*H” t+R)

    cv::Mat errorCovPost;       // a posteriori error covariance matrix (\Sigma_k):
                                // \Sigma_k = ( I - K_k * H ) * \Sigma'_k

    ...
};
```

你可以使用默认构造函数创建滤波器对象，然后使用cv::KalmanFilter::init()方法进

行配置，也可以使用cv::KalmanFilter::init()调用共享相同参数列表的构造函数。在这两种情况下，需要四个参数。

第一个是dynamParams；这是状态向量$\vec{x}_k$的维数。动态参数什么样并不重要，重要的是它们的数量。请记住，它们的解释将由滤波器的各种其他组件（特别是状态转换矩阵F）来设置。下一个参数是measureParams；这个维数出现在测量中——z_k的维数。与x_k一样，滤波器的其他组件最终给出$\vec{z}_k$的含义，所以我们现在关心的是矢量的总维数（在这种情况下，$\vec{z}_k$的含义主要来自我们定义测量矩阵H的方式以及它与$\vec{x}_k$的关系。如果要对系统进行外部控制，还必须指定控制向量的$\vec{u}_k$维度。默认情况下，滤波器的所有内部组件将被创建为32位浮点类型。如果希望滤波器以更高的精度运行，可以将最终参数（类型）设置为cv::F64。

接下来的两个函数实现了Kalman滤波过程（图17-10）。一旦数据被输入到结构中（我们将在稍后讨论一下），我们可以通过调用cv::KalmanFilter::predict()来计算下一个时间步长的预测，然后通过调用cv::KalmanFilter::correct()[注26]整合新的测量。在这种情况下，statePost将设置为statePre的值。预测的方法（可选）需要控制向量$\vec{u}_k$，而校正的方法则需要测量矢量$\vec{z}_k$。运行这些例程之后，我们可以获得正在跟踪的系统的状态。cv::KalmanFilter::correct()的结果放在statePost中，而cv::KalmanFilter::predict()的结果放在statePre中。你可以从滤波器的成员变量中读取这些值，也可以只使用两种方法的返回值。

注意，与OpenCV中的许多对象不同，这个对象的所有成员变量都是公共变量。它们没有“get/set”函数，所以你可以直接访问它们。例如，状态转换矩阵F在这个场景中称为cv::KalmanFilter::transitionMatrix，这是你刚刚为自己的系统设置的矩阵。下面这个例子将更有意义。

Kalman示例代码

显然，现在是时候来看一个合适的例子了。让我们先找一个比较简单的例子，明确地实现出来。想象一下，我们有一个点在一个圆内移动，就像在赛道上的车。汽车以围绕轨道的恒定速度移动，但是存在一些变化（即，过程噪声）。我们使用诸如通过我们的视觉算法跟踪它的方法来测量汽车的位置。这也会产生一些（不相关和可能不同）的噪声（即，测量噪声）。

所以我们的模型很简单：汽车在任何时候都有一个位置和一个角速度。这些因素一起形成二维状态向量$\vec{x}_k$。然而，我们的测量只是汽车的位置，因此形成一维“向量”$\vec{z}_k$。

注26：在最近的OpenCV版本中，当没有测量手段可用时（例如，被跟踪的对象被遮挡），可以省略校正步骤。

我们要写一个程序（示例17-1），其输出将显示车辆围绕（红色）圆的运动以及我们进行的测量（黄色）和卡尔曼滤波器（白色）预测的位置。

我们从通常的调用开始，包括库头文件。我们还定义一个宏，当我们想要将车的位置从角度转换为笛卡尔坐标时，这是有用的，让我们可以在屏幕上绘制轨迹。

示例17-1：卡尔曼滤波器示例

```
#include "Opencv2/opencv.hpp"
#include <iostream>

using namespace std;
#define phi2xy( mat )                                                    \
  cv::Point( cv::cvROund( img.cols/2 + img.cols/3*cos(mat.at<float>(0)) ),    \
             cv: :cvROund( img.rows/2 - img.cols/3*sin(mat.at<float>(0)) ))

int main(int argc, char** argv) {

...
```

接下来，我们将创建一个随机数生成器，一个绘制的图像和卡尔曼滤波器结构体。请注意，我们需要告诉卡尔曼滤波器状态变量的大小（2）和测量变量的大小（1）。

```
    ...

        // Initialize, create Kalman filter object, window, random number
        // generator etc.
        //
        cv: :Mat img( 500, 500, CV_8UC3 );
        cv: :KalmanFilter kalman(2, 1, 0);

    ...
```

一旦我们建立了这些构建块，就可以为状态x_k($\vec{x}_k$)创建一个矩阵（实际是一个向量，但是在OpenCV中我们称之为“矩阵”），处理噪声w_k($\vec{w}_k$)，测量z_k($\vec{z}_k$)，以及最重要的转移矩阵transitionMatrix(F)。状态矩阵需要被初始化，所以我们用一些合适的随机数来进行填充。这些随机数分布在零附近。

转移矩阵至关重要，因为它把时间k处的系统的状态与时间$k + 1$的状态关联起来。在这种情况下，转移矩阵大小将是2×2（因为状态向量是二维的）。事实上，这是向状态向量的成分赋予意义的转换矩阵。我们用x_k表示汽车的角位置（φ）和汽车的角速度（ω）。在这种情况下，转移矩阵具有[[1, dt], [0, 1]]分量。因此，在我们乘以F后，状态(ϕ, ω)变为$(\varphi + \omega\ \delta\tau, \omega)$，即角速度不变，位置增加等于与时间步长相乘的角速度的量。在我们的例子中，为了方便，我们选择dt = 1.0，但实际上，我们需要使用连续视频帧间隔时间这样的数值。

```
...
  // state is (phi, delta_phi) - angle and angular velocity
  // Initialize with random guess.
  //
  cv::Mat x_k( 2, 1, CV_32F );
  randn( x_k, 0., 0.1 );

  // process noise
  //
  cv::Mat w_k( 2, 1, CV_32F );

  // measurements, only one parameter for angle
  //
  cv::Mat z_k = cv::Mat::zeros( 1, 1, CV_32F );

  // Transition matrix 'F' describes relationship between
  // model parameters at step k and at step k+1 (this is
  // the  "dynamics" in our model.
  //
  float F[] = { 1, 1, 0, 1 };
  kalman.transitionMatrix = Mat( 2, 2, CV_32F, F ).clone();

...
```

卡尔曼滤波器也具有其他的必须初始化的内部参数。具体而言，1×2测量矩阵H被初始化为[1,0]（这可能稍微不太容易发现）。过程噪声和测量噪声的协方差被设置为合理但有趣的值（可以自己试一试），并且后验误差协方差被初始化成单位矩阵（这样做是为了保证第一次迭代有意义，随后将被改写）。

与此类似，因为此时没有任何信息，所以我们将后验状态（先前的假设步骤）初始化为随机值。

```
...

  // Initialize other Kalman filter parameters.
  //
  cv::setIdentity( kalman.measurementMatrix,   cv::Scalar(1)    );
  cv::setIdentity( kalman.processNoiseCov,     cv::Scalar(1e-5) );
  cv::setIdentity( kalman.measurementNoiseCov, cv::Scalar(1e-1) );
  cv::setIdentity( kalman.errorCovPost,        cv::Scalar(1)    );

  // choose random initial state
  //
  randn(kalman.statePost, 0., 0.1);
  for(;;) {
...
```

最后,我们可以在实际动态系统上开始预测。首先，用卡尔曼滤波器预测这一步骤会产生什么（即在它获得任何新信息之前）；我们称之为y_k($\vec{x}_k^-$)。然后，我们继续为此迭代生成z_k（$\vec{z}_k$，测量）的新值。根据定义，该值是乘以随机测量噪声的测量矩阵H的“实数”值x_k($\vec{x}_k$)。这里必须提一下，除了类似这个的简单程序之外，不能直接从x_k生成

z_k；而是由状态或传感器产生一个生成函数。在这种模拟情况下，我们通过自己添加随机噪声从一个基本的“真实”数据模型生成测量值，这样一来，就可以看出卡尔曼滤波器的效果了。

```
...
    // predict point position
    //
    cv::Mat y_k = kalman.predict();
    // generate measurement (z_k)
    //
    cv::randn(z_k, 0., sqrt((double)kalman.measurementNoiseCov.at<float>(0,0)));
    z_k = kalman.measurementMatrix*x_k + z_k;

...
```

画出与我们先前融合的观察值相对应的三个点、卡尔曼滤波器预测的位置以及基本状态（在这种模拟情况下，我们碰巧已经知道这个状态）：

```
...
    // plot points (e.g., convert to planar co-ordinates and draw)
    //
    img = Scalar::all(0);
    cv::circle( img, phi2xy(z_k), 4, cv::Scalar(128,255,255) );    // observed
    cv::circle( img, phi2xy(y_k), 4, cv::Scalar(255,255,255), 2 ); // predicted
    cv::circle( img, phi2xy(x_k), 4, cv::Scalar(0,0,255) );        // actual

    cv::imshow( "Kalman", img );

...
```

现在，我们可以开始下一次迭代。首先要做的就是调用卡尔曼滤波器，并赋给其最新的测量值。接下来就是产生过程噪声。然后，我们使用转移矩阵F对时间步长x_k进行一次迭代，并添加生成的过程噪声。现在，我们又可以开始新一轮的计算了。

```
...
    // adjust Kalman filter state
    //
    kalman.correct( z_k );

    // Apply the transition matrix 'F' (e.g., step time forward)
    // and also apply the  “process" noise w_k
    //
    cv::randn(w_k, 0., sqrt((double)kalman.processNoiseCov.at<float>(0,0)));
    x_k = kalman.transitionMatrix*x_k + w_k;

    // exit if user hits 'Esc'
    if( (cv::waitKey( 100 ) & 255) == 27 ) break;
}
```

```
    return 0;
}
```

可以看出，卡尔曼滤波器部分并不复杂，有一半所需的代码只是产生一些填充信息。无论如何，为了确保这一切都有意义，我们都应该总结我们所做的一切事情。

从创建矩阵来表示系统的状态和我们将做的测量开始，我们定义了传递矩阵和测量矩阵，然后初始化了滤波器的噪声协方差和其他参数。

将状态向量初始化为随机值后，我们调用卡尔曼滤波器，并进行第一次预测。一旦获得了这个预测（虽然第一次没有太大意义），就将其在屏幕上绘制出来。我们还合成了一个新的观察结果，并在屏幕上绘制出来，与滤波器的预测进行对比。接下来，我们通过新测量值，将新信息传递给滤波器，并将其纳入其内部模型。最后，我们将模型融合成了一个新的“真实”状态，以便我们再次遍历循环。

当我们运行代码时，可以看到小红球围绕着圆形轨道运动。黄色的小球在红色球周围出现和消失，代表卡尔曼滤波器正在尝试“看透”的噪音。白球迅速收敛到在红色球周围的一个小空间 中运动，表明卡尔曼滤波器在我们的模型框架内给出了粒子（汽车）运动的合理估计。

我们在示例中没有讨论的一个问题是控制输入的使用。例如，如果这是一个无线电控制的汽车，并且我们知道人会控制其行为，就可以将这些信息包含在我们的模型中。在这种情况下，速度可以由控制器设定。然后我们需要提供一个矩阵B（`kalman.controlMatrix`），并为`cv::KalmanFilter::predict()`提供第二个参数，以适应控制向量$\vec{u}_k$。

扩展卡尔曼滤波器简述

你可能已经注意到，要求系统的动态特性在基本参数上是线性的，是非常苛刻的。事实证明，当动态特性是非线性时，卡尔曼滤波器仍然有用，并且OpenCV中的卡尔曼滤波器函数也是有用的。

回想一下，“线性”意味着（实际上）卡尔曼滤波器定义中的各个步骤可以用矩阵表示。什么情况下不是这样的呢？实际上有很多可能。例如，假设我们的控制测量是汽车的油门踏板被踩下的程度；汽车的速度与油门踏板被踩下的程度之间的关系不是线性关系。另一个常见的问题是施加在汽车上的力在笛卡尔坐标上表示更合适，而汽车的运动（如我们的例子）在极坐标中表示更合适。尤其是如果我们的汽车是沿着圆形路径移动的小船，但水流同时向某个特定方向匀速流动时，就可能出现这种情况。

在所有这些情况下，仅仅使用卡尔曼滤波器是不够的，在时刻$t+1$的状态不是时间t的状态的线性函数。解决这些非线性的一种方法是将相关过程（例如更新F或控制输入响应B）局部线性化。因此，我们需要根据状态x在每个时间步长计算F和B新的值。这些值只是在某个x的特定值附近的实际更新和控制函数，但实际上这通常就足够了。对Kalman滤波器的这种扩展被简单地称为扩展卡尔曼滤波器或*EKF* [Schmidt66]。

OpenCV没有提供任何具体的实现EKF的例程，不过实际上也不需要。我们需要做的是在每次更新之前重新计算，并重置kalman.transitionMatrix和kalman.controlMatrix的值。

一般来说，当介绍卡尔曼滤波器时，我们以前遇到的矩阵更新方程现在可以理解为一般形式的特殊情况：

$$\vec{x}_k = \vec{f}(\vec{x}_{k-1}, \vec{u}_k) + \vec{w}_k$$

$$\vec{z}_k = \vec{h}(\vec{x}_k) + \vec{v}_k$$

这里$f(x_{k-1}, u_k)$和$h(x_k)$是其参数的任意非线性函数。为了使用Kalman 滤波器提供的线性公式，我们根据由下式定义的矩阵F_k和H_k在每个时间步长重新计算矩阵F和H：

$$[F_k]_{i,j} = \left.\frac{\partial f_i}{\partial x_j}\right|_{x^*_{k-1}, u_k}$$

$$[H_k]_{i,j} = \left.\frac{\partial h_i}{\partial x_j}\right|_{x^*_{k-1}}$$

注意，形成新矩阵的偏导的评估是在前一时间步长（注意，右上角有x^*_{k-1}号）的估计位置处执行。我们可以将F_k和H_k的值代入一般的更新方程。[注27]

$$\vec{x}_k = F_k\vec{x}_{k-1} + \vec{w}_k$$

$$\vec{z}_k = H_k\vec{x}_k + \vec{v}_k$$

卡尔曼滤波器已被更优雅地扩展到非线性系统中，称为unscented粒子滤波器[Merwe00]。在[Thrun05]中给出了卡尔曼滤波器整个领域的非常好的概述，包括最近的进展，以及粒子滤波器（我们这里仅做了简要的介绍）。

注27：你会注意到矩阵B消失了。这是因为在一般的非线性情况下，状态$\vec{x}_k$和控制$\vec{u}_k$在更新中耦合并不罕见。因此，$F\vec{x}_k + B\vec{u}_k$的泛化表示为——而不是像EKF中的$\vec{f}(x_{k-1}) + \vec{b}(\vec{u}_k)$，因此$\vec{u}_k$的影响被吸收到$F_k$中。

小结

在本章中，我们继续对跟踪进行了讨论，将稠密光流，Mean-Shift，Camshift和运动模板添加到第16章所学的稀疏跟踪方法中。我们用一个关于卡尔曼滤波器等递归估计器的讨论结束本章。在对卡尔曼滤波器的数学理论进行概述之后，我们讨论了OpenCV实现这个构造。我们用一个关于更高级的滤波器的简要说明总结了使用Kalman 滤波器来处理不完全符合卡尔曼滤波器所需假设的更复杂的情况。附录B描述的*Opencv_contrib*目录中的optflow和tracking函数还引用了其他光流和跟踪算法。

练习

OpenCV库中包含示例代码例程，演示了本章讨论的许多算法。请在以下练习中使用这些例程：

- *lkdemo.cpp* (sparse optical flow in *opencv/samples/cpp*)
- *fback.cpp* (Farnebäck dense optical flow in *samples/cpp*)
- *tvl1_optical_flow.cpp* (dense optical flow in *samples/cpp*)
- *camshiftdemo.cpp* (mean-shift tracking of colored regions in *samples/cpp*)
- *motempl.cpp* (motion templates, in *opencv_contrib/modules/optflow/samples*)
- *kalman.cpp* (Kalman filter in *samples/cpp*)
- *768x576.avi* (movie data files in *samples/data*)

1. 假定当前状态取决于先前状态的位置和速度，使用运动模型，结合*lkdemo.cpp*（仅需要点击几次来输入点的位置）与卡尔曼滤波器更好地跟踪Lucas-Kanade点。显示每个点附近的不确定性。哪里没有跟踪成功？

   提示：使用Lucas-Kanade作为卡尔曼滤波器的观察模型，并调整噪声使其可以保持跟踪。保持运动的合理性。

2. 使用Farnebäck的稠密光流算法编译*fback.cpp*，并在*768x576.avi*上运行程序。修改程序，把算法的纯执行时间打印出来。在计算机上安装OpenCL驱动程序（如果用的是Mac，则可能已经安装好了），并且试一试是否使用OpenCL（请确保使用OpenCV 3.0或更高版本）执行速度会更快。尝试用TV-L1（参见*tvl1_optical_flow.cpp*）或Simple Flow调用替换Farnebäck，并再次测量算法运行时间。

3. 编译并运行*camshiftdemo.cpp*，使用网络摄像机或彩色运动对象的彩色视频。使用鼠标在移动物体周围画一个（紧的）框，这个程序会跟踪它。

a. 在*camshiftdemo.cpp*中，用cv::meanShift()替换cv::camShift()例程。描述在什么情况下一个跟踪器比另一个效果更好。

b. 编写一个函数，可以在初始cv::meanShift()框的时候产生一网格的点。一次运行两个跟踪器。

c. 如何将这两个跟踪器结合起来使用使跟踪更鲁棒？说明并/或进行实验。

4. 使用网络摄像头或使用以前存储的电影文件编译并运行运动模板代码*motempl.cpp*。

a. 修改*motempl.cpp*，使其可以做简单的手势识别。

b. 如果相机正在移动，请说明如何使用运动稳定代码来启用运动模板，以适应中等移动速度的相机。

5. 卡尔曼滤波器依赖线性运动和马尔科夫独立性（即，当前状态仅取决于紧邻的上一过去状态，而不依赖于过去所有的状态）。假设你想跟踪一个目标，它的运动与其先前位置及其之前的速度相关，但是你错误包含了一个仅依赖先前位置的动态项—— 换句话说，忘记了先前的速度项。

a. Kalman的假设仍然成立吗？如果成立，请解释原因；如果没有成立，解释是如何违反假设的。

b. 当你忘记某些动态项时，卡尔曼滤波器是怎样保持跟踪的？

提示：想想噪音模型。

6. 描述如何使用线性状态模型（非扩展）卡尔曼滤波器跟踪圆周（非线性）运动。

提示：如何预处理才能使其变回线性动态特性？

第18章

相机模型与标定

视觉开始于检测世界中的光线。从某种发射源（如灯管、太阳等）发出的射线形成光线，然后穿过空间照射到某个物体上。当光线照射到物体上时，大部分光线被吸收，我们认为没有被吸收的光线就是物体的颜色。反射的光线进入眼睛（或者相机），被我们的视网膜（或者我们的图像采集器）吸收。光线从物体发射开始，通过透镜到达眼睛或者相机， 然后到达视网膜或者图像采集器的几何过程是实用计算机视觉的尤为重要的一个方面。

一个简单而有用的解释这种现象的模型是针孔相机模型。[注1]针孔是一个虚拟的中心有一个微型小孔的墙，除了穿过中心小孔的光线，其他的光线都会被阻挡。在本章，我们将从一个针孔相机模型开始，来弄清楚投射光线的基本几何图形。遗憾的是，真实的针孔由于不能为快速曝光收集足够的光线，因此它不是一个得到图像的好方法。这也是为什么眼睛和相机都要使用透镜而不是仅仅用一个点来收集更多光线的原因。然而，这种利用透镜得到更多光线的缺点是，不仅使我们超出了所使用的简单针孔几何模型，而且引入了来自透镜的畸变。

在本章中，我们将学习如何使用相机标定来矫正（数学方式）因使用透镜而给简单针孔

注1： 透镜的知识至少可以追溯到罗马时代。针孔摄像机模型可以追溯将近1000年的海什木（al-Haytham，965—1040，阿拉伯学者、物理学家，数学字，全名穆哈默德·本·哈桑·本·海什木·巴士拉。也有译为“海桑”“哈金”，在光学研究方面有突出的成就），是引入视觉几何方面的经典方式。 在17世纪和18世纪，笛卡尔、开普勒、伽利略、牛顿、胡克、欧拉、费马和斯内尔（见O’Connor [O’Connor02]）等人推动了数学和物理学的进步。几何视觉的一些关键内容包括Trucco [Trucco98]，Jaehne（有时拼写为Jähne）[Jaehne95; Jaehne97]，Hartley和Zisserman [Hartley06]，Forsyth和Ponce [Forsyth03]，Shapiro和Stockman [Shapiro02]，Xu和Zhang[Xu96]。

模型带来的主要偏差。相机标定对于联系相机测量和真实三维世界测量也很重要。它的重要性在于场景不仅仅是三维的，也是用物理单位度量的空间。因此，确定相机的自然单位（像素）和物理世界的单位（例如米）的关系是三维场景重构的重要组成部分。

相机标定的过程既给出相机几何模型，也给出透镜的畸变模型。这两个信息模型定义了相机的内在参数。在本章中，我们使用这些模型来矫正透镜畸变。在第19章，我们会使用这些模型来阐述物理场景的完整几何学。

我们将从讨论相机模型和透镜畸变的原因开始。在这里，我们将研究单应性变换（homography transform），它是一种能够反映相机基本行为、各种失真和矫正特性的数学工具。我们会花一点时间来详细讨论特定相机的变换如何在数学上进行计算。一旦掌握了这些知识，我们可以转移到OpenCV函数来完成大部分工作。

本章阐述的所有知识是为了建立足够的理论以便读者能够真正理解OpenCV函数`cv::calibrateCamera()`，并且知道其外表下的内涵。这对正确使用该函数是非常重要的。这里说明一下，如果你已经是一个专家或者只是简单想了解如何使用OpenCV来做自己已经理解的东西，跳到后面介绍标定函数那一小节并使用它即可。请注意，附录B参考了`ccalib`函数数组中其他标定模式和技术。

相机模型

我们先看最简单的相机模型：针孔模型。在这个简单模型中，想象光线是从场景或一个很远的物体发射过来的，但只有一条光线从该场景中的任意特定点进入针孔。在实际针孔相机中，该点被“投影”到成像表面。其结果是在图像平面（也称为投影平面）上，图像被聚焦 ，与远处物体相关的图像大小用一个相机参数来描述：焦距。对于理想的针孔相机，从针孔到屏幕的距离就是焦距。[注2]如图18-1所示，f是相机焦距，Z是相机到物体的距离，X是物体的长度，x是图像平面上的物体图像。在该图中，我们可以通过相似三角形得到$-x/f = X/Z$，或：

$$-x = f \cdot \frac{X}{Z}$$

我们重新把针孔相机模型整理成另一种等价形式，使其数学形式更加简单。在图18-2中，我们交换针孔和图像平面，[注3]主要的差别是图像平面出现在了右边。针孔中的点

注2：您可能习惯于考虑透镜的焦距，其实焦距是特定透镜的属性，与射影几何过程无关。这是滥用术语的结果。明确地说，“投影距离”是几何属性，“焦距”是透镜的属性。上述方程式中的f实际上是投影距离。对于透镜，当且仅当配置的焦距与镜头的焦距匹配时，图像才会对焦，因此人们倾向于对这两个术语互换使用。

注3：这种典型的数学抽象无法在物理上构建，可以简单将图像平面看成所有通过投射中心的光线的“切片”。这个想法使绘制更加简单，同时数学处理更容易。

被理解为投影中心。这样，每一条光线从远处物体的某个点出发，到达投影中心。光轴与图像平面的交点被称为“主点”（principal point）。在这个与旧的投影平面（或图像平面）等价的新图像平面上（图18-2），远处物体的图像与图18-1中的图像大小完全一致。光束与图像平面相交生成图像，而平面与投影中心的距离是f。这样便形成了比之前更容易理解的相似三角形关系$x/f = X/Z$。负号被去掉了，因为目标图像不再是倒立的。

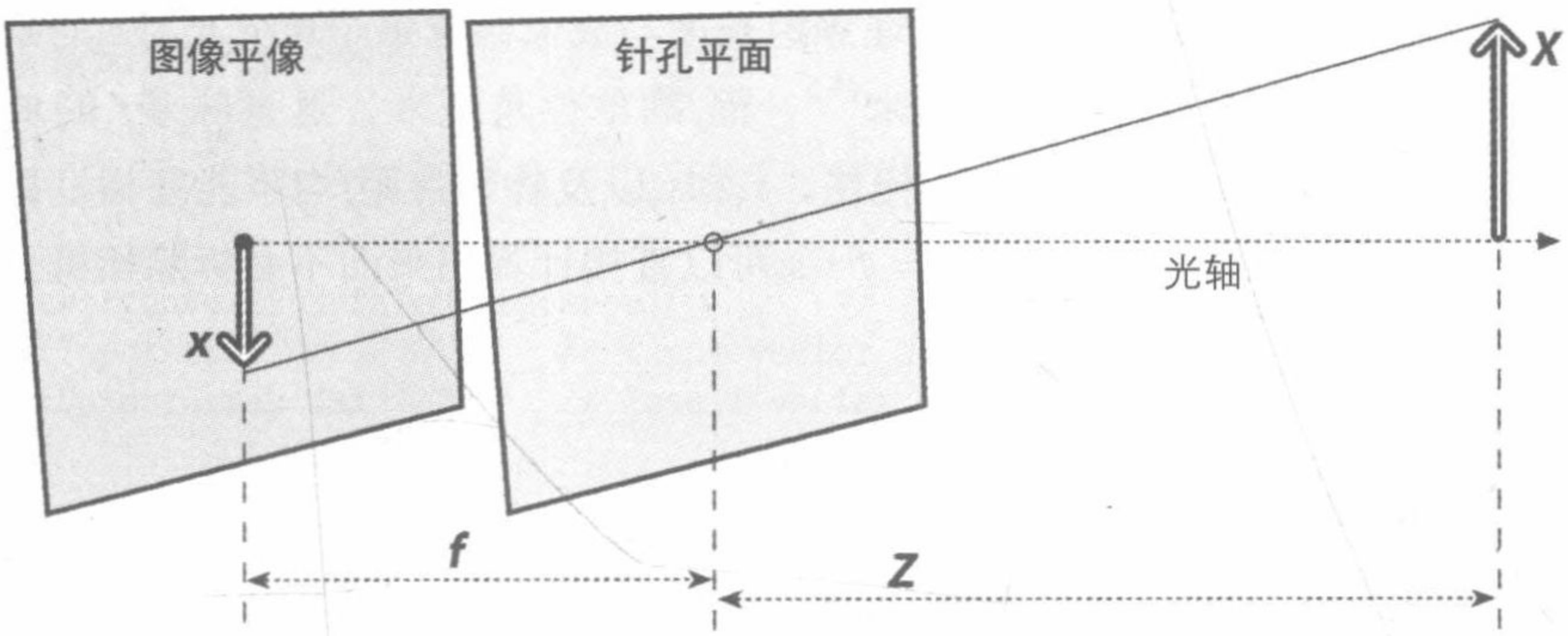


图18-1. 针孔相机模型：过空间某特定点的光线才能通过针孔（针孔光圈），这些光束被投影到图像平面形成图像

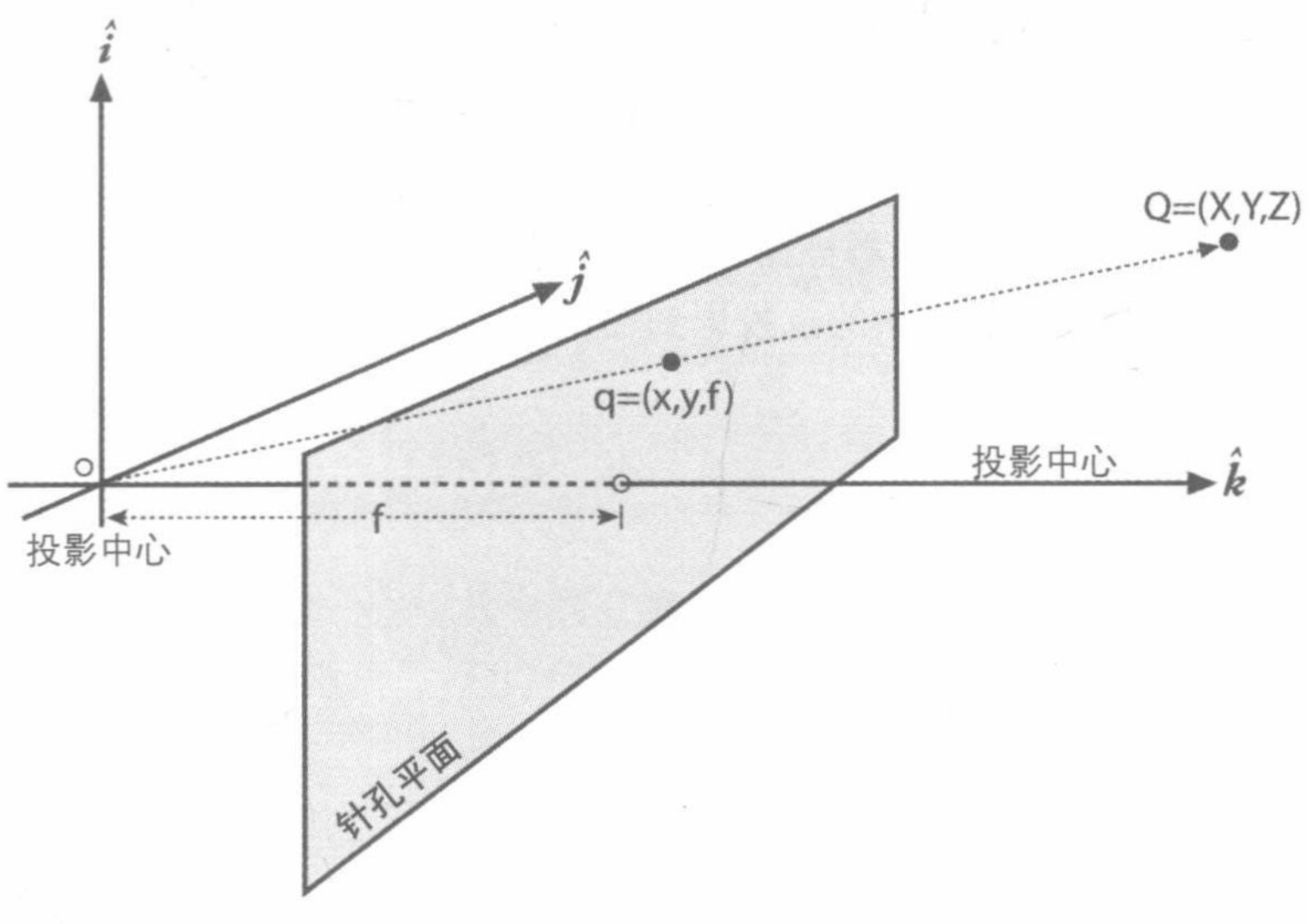


图18-2. $\vec{Q} = (X, Y, Z)$点由通过投影中心的光线投射到图像平面上，图像上相应的图像点为$\vec{q} = (x, y, f)$。图像平面实际上是把投影屏幕放到针孔的前方（数学上等价，但形式更简单）

你也许认为主点等同于成像装置的中心，但是这意味着某些人拿着镊子和胶水还把相机里面的成像装置以微米级别的精度安装。实际上，芯片的中心通常不在光轴上，我们因此引入两个新的参数c_x和c_y，对投影屏幕坐标中心可能的偏移（对光轴而言）进行建模。

这样一来物理世界中的点$\vec{Q}$，其坐标为(X, Y, Z)，根据下式投射到成像装置上某个像素位置(x_{screen}, y_{screen})：[注4]

$$x_{screen} = f_x \cdot \frac{X}{Z} + c_x \text{ , and } y_{screen} = f_y \cdot \frac{Y}{Z} + c_y$$

注意，我们引入了两个不同的焦距；原因是单个像素在低价成像装置上是矩形而不是正方形。例如，焦距f_x实际上是透镜的物理焦距长度与成像装置每个单元尺寸s_x的乘积（这样做的意义在于s_x的单位是像素/每毫米[注5]，而f的单位是毫米，这意味着f_x的单位是像素）。同样的道理也适用于f_y和s_y。请记住，s_x和s_y以及物理焦距f均不能在相机标定过程中直接测量，只有组合量$f_x = f \cdot s_x$和$f_y = f \cdot s_y$可以直接计算出来而不必拆除相机去直接测量其部件。

射影几何基础

将物理世界中坐标为(X_i, Y_i, Z_i)的一系列物理点$\vec{Q}_i$映射到投影平面上坐标为(x_i, y_i)的点的过程叫“射影变换”。采用这种变换，可以方便地使用我们所熟知的齐次坐标。齐次坐标把维数为n的投影空间上的点用$(n + 1)$维向量（例如x, y, z变为x, y, z, w）表示，其额外限制是任意两个点的值成比例时即为等价点。在这里，图像平面是一个二维投影空间，因此可以用一个三维向量$\vec{q} = (q_1, q_2, q_3)$来表示该平面上的点。由于所有点在投影空间中的比例值相同，因此我们可以通过除以q_3得到实际的像素坐标值。这样一来，便允许我们将定义相机的参数（如f_x, f_y, c_x和c_y）重新排列为一个3 × 3的矩阵，称为“相机的内参矩阵”[注6]（OpenCV对求解摄像机内参使用的方法源自Heikkila和Silven，详情参见[Heikkila97]）。将物理世界中的点投影到相机上，可以用下列简单的形式表示：

$$\vec{q} = M \cdot \vec{Q}$$

其中：

$$\vec{q} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \text{ and } \quad \vec{Q} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

注4：在这里，下标$screen$表示正在计算的坐标是屏幕（即成像装置）上的坐标。方程中的(x_{screen}, y_{screen})与图18-2中(x, y)之间的区别在于c_x和c_y。因此，我们使用screen下标来表示成像装置上的坐标。

注5：当然，毫米只是代替你喜欢的物理单位。它和米、微米或弗隆一样简单。重点是s_x将物理单位转换为像素单位。

将乘数提取出来，可以发现$w = Z$，并且是用齐次坐标表示的，我们可以通过除以w（或Z）来恢复我们之前的定义。没有减号是因为我们在针孔前面来看投影平面上的非倒立图像，而不是在针孔后面来看投影平面上的倒立图像。

当我们讨论齐次坐标的时候，需要介绍OpenCV库中的一些函数，函数cv::convertPointsToHomogeneous()和cv::convertPointsFromHomogeneous()允许我们在齐次坐标和非齐次坐标之间转换[注7]。

它们的原型如下：

```
void cv::convertPointsToHomogeneous(
  cv::InputArray src,          // Input vector of N-dimensional points
  cv::OutputArray dst          // Result vector of (N+1)-dimensional points
);

void cv::convertPointsFromHomogeneous(
  cv::InputArray src,          // Input vector of N-dimensional points
  cv::OutputArray dst          // Result vector of (N-1)-dimensional points
);
```

第一个函数需要一个N维点的向量（用一般形式表示），并且从该向量中构造出$(N + 1)$维点的向量。所有新构造的向量所添加的维度值设置为1。结果为：

$$\overrightarrow{dst}_i = (src_{i,0} \quad src_{i,1} \quad \dots \quad src_{i,N-1} \quad 1)$$

第二个函数从齐次坐标进行转换。给定N维点的输入向量，首先将每个点的所有分量都除以该点的最后一个分量，再将最后一个分量丢弃，从而构造出（N-1）维点的向量。结果为：

$$\overrightarrow{dst}_i = \left(\frac{src_{i,0}}{src_{i,N-1}} \quad \frac{src_{i,1}}{src_{i,N-1}} \quad \dots \quad \frac{src_{i,N-2}}{src_{i,N-1}} \right)$$

采用理想针孔，我们有一个对一些视觉中的三维几何有用的模型。但请记住，由于只有很少量的光线通过针孔，在实际中，无论使用何种图像采集器，都需要等待积累足够的光线，因此成像速度非常慢。[译注1]对于快速生成图像的相机而言，必须利用更大面积的光线，甚至让光线弯曲，从而让足够多的光线能够聚焦到投影点上。我们使用透镜来实

注7：还有第三个函数cv::convertPointsHomogeneous()，这是一个用来调用前两者之一的简便方法。它将查看*dst*中提供的点的维数，并自动确定是否要将其转换为或从齐次坐标转换。此函数现在被认为是过时的。它主要用于向后兼容。它不能用于新的代码，因为它大大降低了代码的可读性。

译注1：半导体光电元件的光电转化和光强度以及累积时间都有关系，相同的强度下累计时间变长也会使响应变强。

现这个目的。透镜可以聚焦足够多的光线到一个点上，使得图像生成更加迅速，但代价是引入了畸变。

Rodrigues变换

在三维空间操作时，通常使用3×3矩阵来表示空间中的旋转。这种表示通常最方便，因为将向量乘以该矩阵相当于以某种方式旋转向量。缺点是很难理解3×3矩阵表示什么样的旋转。我们将在本章中简要介绍一些OpenCV函数所使用的可选的旋转表示以及转换和形成这个可选表示的函数。

这种旋转表示通过一个向量表示沿着某一角度进行旋转，从而更易于[注8]可视化，向量的方向表示旋转轴的方向，向量的长度表示沿逆时针方向的旋转量。这个很容易做到，因为方向可以用任意幅度的向量表示。因此，我们可以选择我们的向量幅度等于旋转的角度。矩阵和向量的关系，用Rodrigues变换来表示。[注9]

令$\vec{r}$为三维向量$\vec{r} = [r_x \quad r_y \quad r_z]$，这个向量定义$\theta$，用$\vec{r}$的长度（或幅度）表示旋转的角度。我们将这种轴-幅度表示转化为下列的旋转矩阵：

$$R = \cos\theta \cdot I_3 + (1 - \cos\theta) \cdot \vec{r} \cdot \vec{r}^T + \sin\theta \cdot \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix}$$

我们也可以通过使用以下方式从旋转矩阵返回到轴－幅度表示：

$$\sin\theta \cdot \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

因此，我们有计算最简便的表示形式（矩阵表示）和更容易理解的另一个表示形式（Rodrigues表示）。OpenCV为我们提供了一种从一种表示转换为另一种表示的函数：

```
void cv::Rodrigues(
  cv::InputArray src,                          // Input rotation vector or matrix
  cv::OutputArray dst,                         // Output rotation matrix or vector
  cv::OutputArray jacobian = cv::noArray()     // Optional Jacobian (3x9 or 9x3)
);
```

注8：这种“更容易”的表示并非对于人类而言，三维空间的旋转包含三个自由度，从数值优化的角度而言，处理只有三个部分的Rodrigues表示比处理有几个部分的3×3旋转矩阵方便得多。

注9：Rodrigues（罗德里格斯）是19世纪的法国数学家。

假设我们有向量$\vec{r}$，需要求解对应的旋转矩阵表示R。我们设置src为3×1的向量$\vec{r}$，dst为3×3的旋转矩阵R。反过来，我们也可以设置src为3×3的旋转矩阵R，dst为3×1的向量$\vec{r}$。无论是哪种情况，cv::Rodrigues()都可以使用。最后一个参数是可选的，如果jacobian不是cv::noArray()，那么它应该是一个指向3×9或9×3矩阵的指针，该矩阵将填充输出矩阵元素相对于输入矩阵元素的偏导数。输出的jacobian主要用于cv::solvePnP()和cv::calibrateCamera()函数的内部优化，使用cv::Rodrigues()函数主要用于将cv::solvePnP()和cv::calibrateCamera()的输出从Rodrigues格式的1×3或3×1轴－角度向量转化为旋转矩阵。在此时，可以将jacobian设置为cv::noArray()。

透镜畸变

理论上可以定义一个没有引入畸变的透镜。但实际中，不存在完美的透镜。主要因为制造方面的原因，制造一个“球形”透镜比制造一个数学上理想的透镜更容易。同时在机械方面，也很难将透镜和成像装置对齐。这里我们主要讲述两个主要的透镜畸变以及如何对它们进行建模[注10]。径向畸变是由于透镜的形状造成的，而切向畸变则是由整个相机的组装过程造成的。

我们从径向畸变开始说明。实际相机中的透镜经常在成像装置边缘附近的像素位置产生显著的畸变。这个令人头痛的现象来自于“筒形”或“鱼眼”效应（一个很好的例子请参见图18-17顶端的屋子分割线）。图18-3给出了为什么产生径向畸变的直观解释。对于一些透镜来说，远离透镜中心的光线比靠近透镜中心的光线更加弯曲。对于一个廉价的透镜来说，这种现象更加严重。筒形畸变在廉价的网络相机中非常厉害，但在高端相机中不是很明显，因为这些透镜系统做了很多最小化径向畸变的工作。

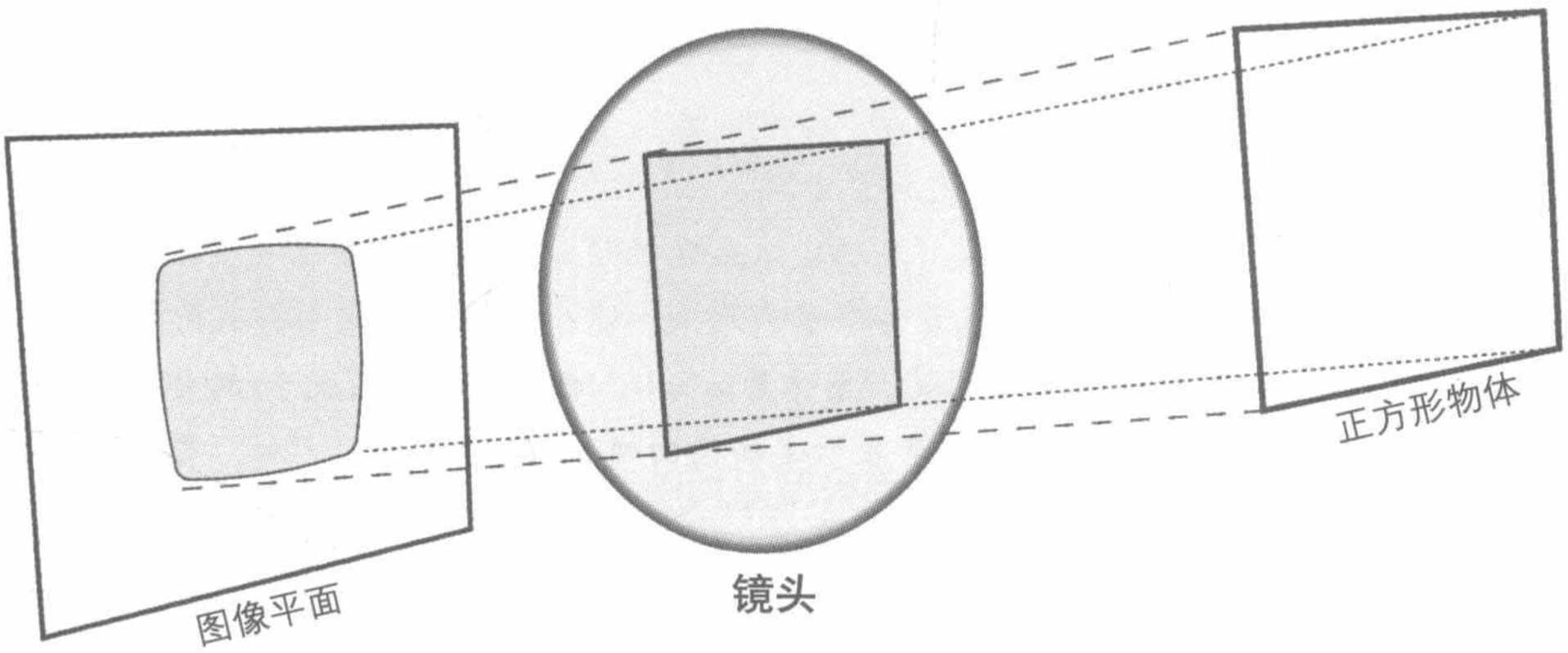


图18-3. 径向畸变：远离透镜中心的光线比靠近透镜中心的光线弯曲很多；因此，正方形的边在图像平面上是弯曲的（也称为筒形畸变）

注10：透镜畸变建模方法主要来自于Brown [Brown71]以及更早的Fryer和Brown[Fryer86]。

对于径向畸变，成像装置的（光学）中心处的畸变为0，随着向边缘移动，畸变越来越大。实际上，这种畸变很小，可以用$r = 0$[注11]附近的泰勒级数展开的前几项来描述。对于廉价的网络相机，我们通常使用前两项；其中通常将第一项称作k_1，第二项称作k_2。对于畸变很大的相机，比如鱼眼透镜，我们可以使用第三个径向畸变项k_3。通常，成像装置上某点的径向位置可以根据以下等式进行调整[注12]：

$$x_{corrected} = x \cdot \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$

和：

$$y_{corrected} = y \cdot \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$

这里(x, y)是（成像装置上）畸变点的原始位置，$(x_{corrected}, y_{corrected})$是矫正后的新位置。图18-4显示了矩形网格由于径向畸变产生的偏移。随着距光学中心的径向距离的增加，矩形网格上的外部点越来越向内偏移。

第二大常见的畸变是切向畸变。这种畸变是由于制造上的缺陷使透镜不与成像平面平行而产生的，如图18-5所示。

切向畸变可以用两个额外参数p_1和p_2来表示，得到[注13]：

$$x_{corrected} = x + \left[2p_1 xy + p_2\left(r^2 + 2x^2\right)\right]$$

和：

$$y_{corrected} = y + \left[p_1\left(r^2 + 2y^2\right) + 2p_2 xy\right]$$

注11：如果你不知道泰勒级数是什么，也不用担心。泰勒级数是用多项式形式在某些特定点附近来表示一个复杂函数的数学技术（多项式中包含的项越多，逼近得越准确）。在我们的例子中，在$r = 0$附近将畸变函数展开为多项式形式。这个多项式的通用形式为$f(r) = a_0 + a_1 r + a_2 r^2 + \ldots$，在这里，当$r$=0时，$f(r)$=0，意味着$a_0 = 0$。由于函数必须关于$r$对称，所以只有$r$的偶次幂的系数不为零。因此，用于表示这些径向畸变的参数是r^2，r^4和（有时）r的更高次项的系数。

注12：现在可以在OpenCV中使用更复杂的“有理”和“薄棱镜”模型。还支持全景（180度）摄像机、全立体摄像机和多摄像机标定。请参阅*opencv_contrib/modules/ccalib/samples*目录中的使用示例。

注13：这些方程的推导超出了本书的范围，但有兴趣的读者可以参考“铅锤”模型，详情参见D. C. Brown，“Decentering Distortion of Lenses”，Photometric Engineering 32，no. 3（1966）：444-462。

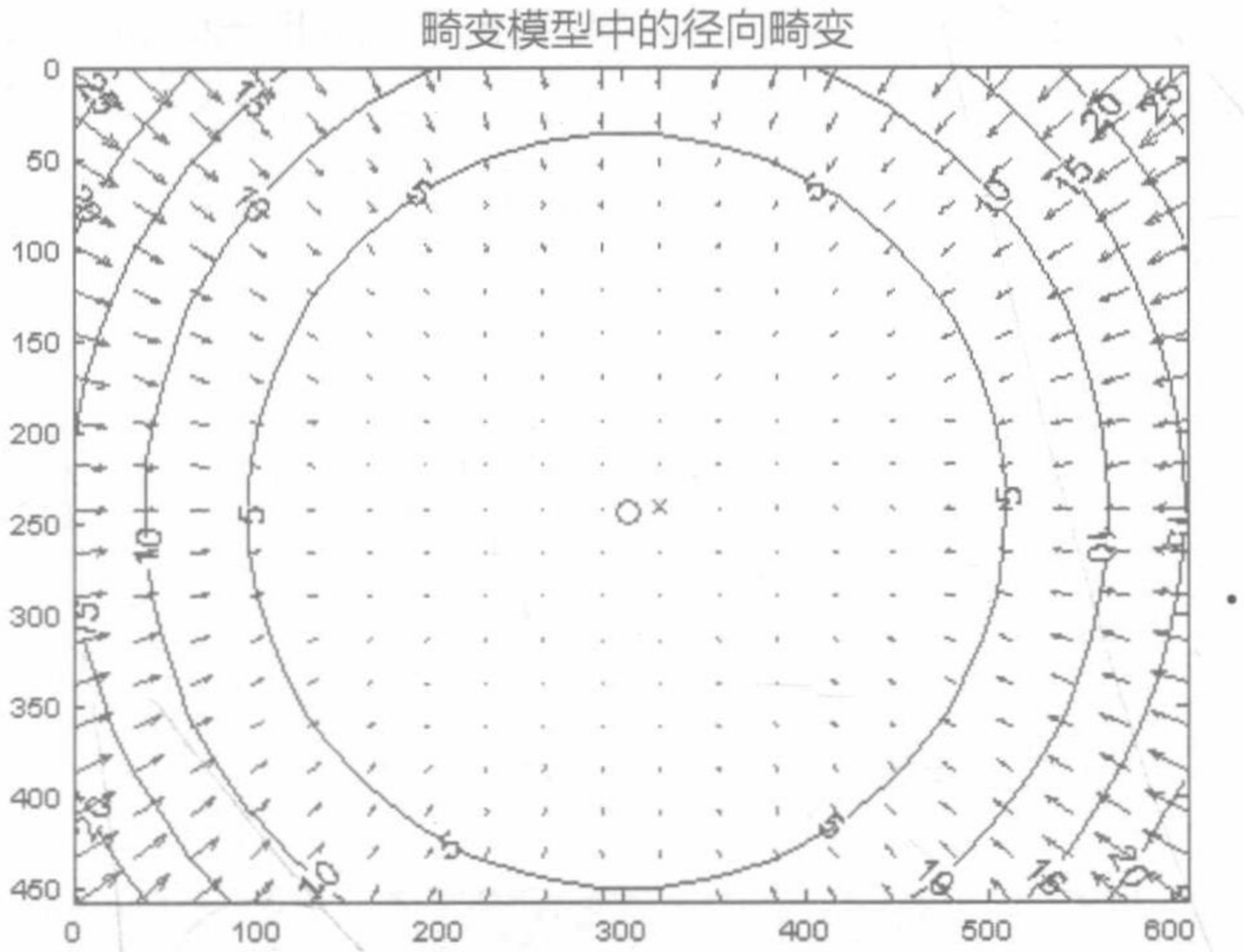
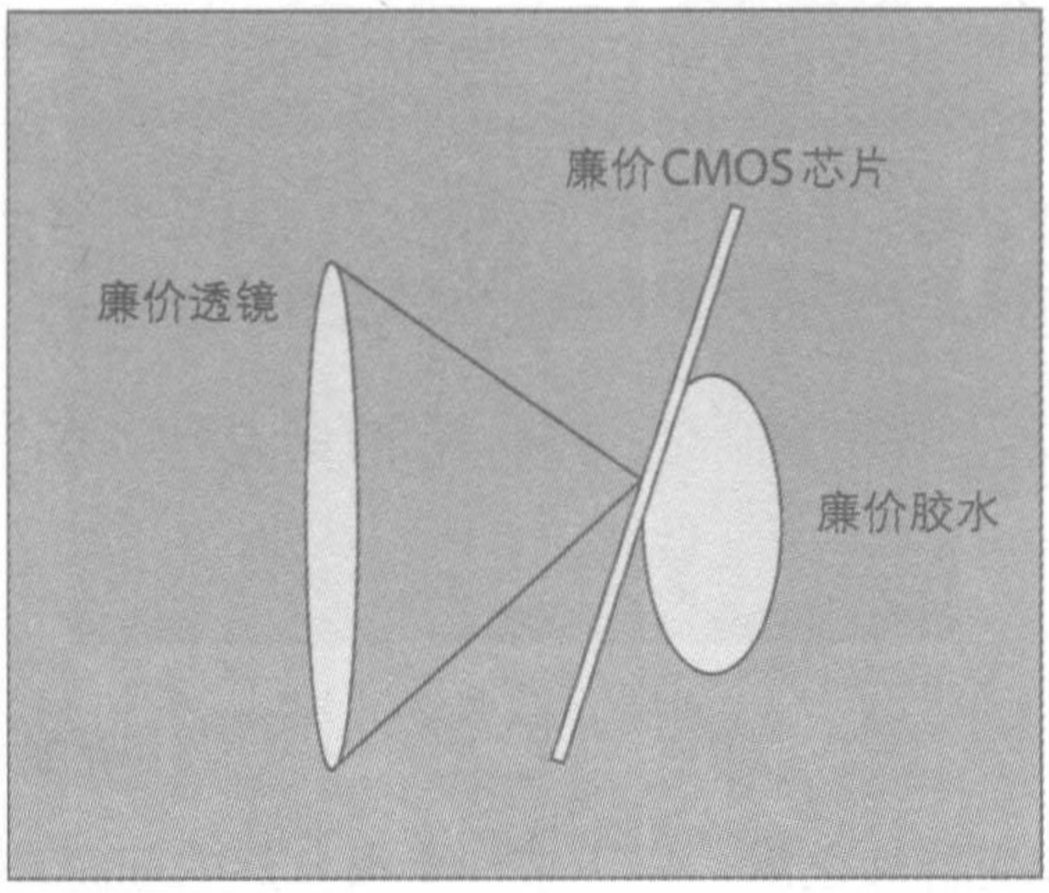


图18-4. 一个特定相机透镜的径向畸变图：箭头显示径向畸变图像上外部矩形网格点的偏移（Jean-Yves Bouguet授权使用）[注14]



廉价相机

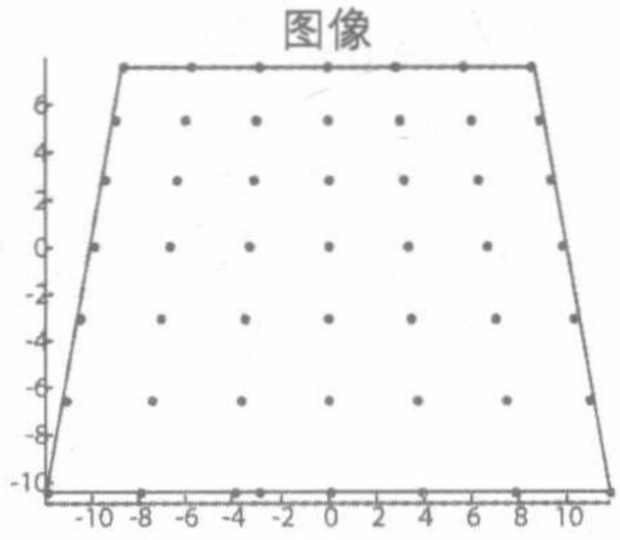


图18-5. 当透镜不完全与图像平面平行的时候则会产生切向畸变；在廉价的相机中，这种现象发生在成像装置被粘在相机背面的时候（Sebastian Thrun授权使用）

注14：一些传统的摄像机没有矩形传感器。由于一些不完美的制造技术（以及其他原因），这些传感器实际上是平行四边形。这种摄像机的摄像机内参矩阵具有形式[[f_x $skew$ c_x] [0 f_y c_y] [0,0,1]]。这是图18-4中“偏移”的由来。现在，几乎所有的摄像机都没有偏移，所以OpenCV标定函数假定偏移为零，不计算它）。

因此我们总共需要五个畸变参数。因为在大多数使用它们的OpenCV程序中，这五个参数都是必需的，因此它们被放到一个畸变向量中，这是一个5×1的矩阵包括k_1，k_2，p_1，p_2和k_3（按顺序）。图18-6显示了切向畸变对前面外部矩形网格点的影响。这些点的椭圆形偏移是位置和半径的函数。

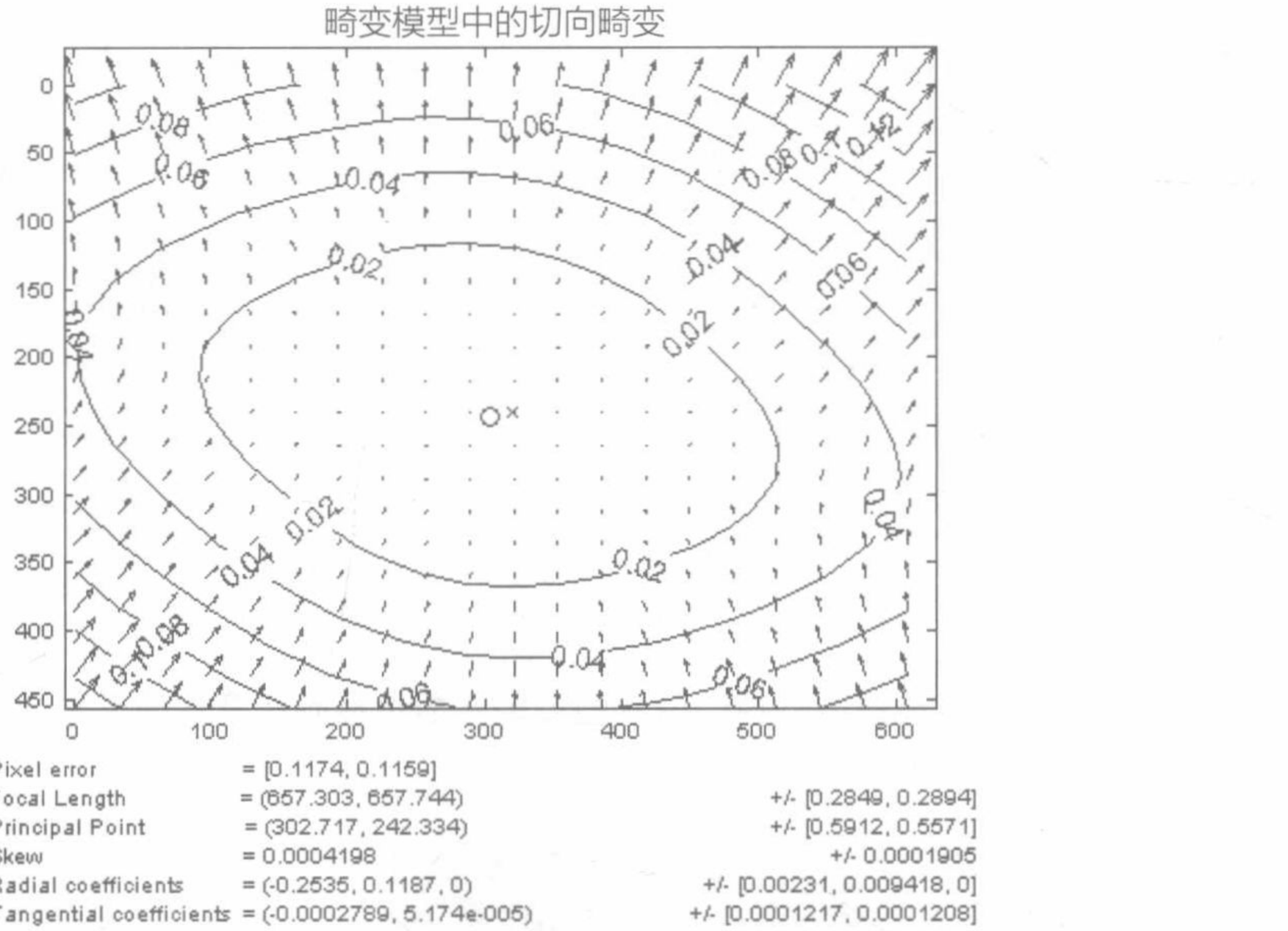


图18-6. 一个特定相机透镜的切向畸变图：箭头显示切向畸变图像上外部矩形网格点的偏移（Jean-Yves Bouguet授权使用）

在成像系统中还有很多其他类型的畸变，但是它们比径向和切向畸变的影响小。因此，我们和OpenCV都不会进一步处理它们。

标定

知道如何用数学来描述相机的内在和畸变特性之后，接下来一个很自然的问题是如何使用OpenCV来计算内参矩阵和畸变向量。[注15]

OpenCV提供了一些算法来帮我们计算这些内部参数。实际标定过程是通过cv::calibrateCamera()来完成的。在这个程序中，标定的方法是把相机对准一个具有很多独立可标识点的已知结构。通过从多个视角观察这个结构，我们可以计算拍摄每个图像时相机的（相对）位置和方向以及相机的内部参数（参见后面的图18-10）。为了提供

注15：有关摄像机标定有一个很好的网上教程，请参见Jean-Yves Bouguet的标定网站。

多个视角，我们需要旋转和平移物体，所以暂停一下，来了解旋转和平移的一些相关知识。

OpenCV一直在改进标定技术; 现在有许多不同类型的标定板模式，如后面的“标定板”一节所述。还有“与众不同的”相机的专用标定技术。对于鱼眼透镜，你需要在用户文档中的cv::fisheye类中使用鱼眼方法。

还有全景（180度）相机和多面体校准技术；请参阅*opencv_contrib/modules/ccalib/samples*，*opencv_contrib/modules/ccalib/tutorial/omnidir_tutorial.markdown*，*opencv_contrib/modules/ccalib/tutorial/multi_camera_tutorial.markdown*和/或分别搜索“omnidir”和“multiCameraCalibration”用户文档，参见图18-7。[注16]



图18-7. 全景相机[注17]

旋转矩阵和平移向量

对相机拍摄的一个特定物体的图像，我们可以用旋转和平移来描述物体相对于相机坐标系统的姿态，如图18-8所示。

注16：这些都是高度针对性的例子，具体细节超出了本书的讨论范围。然而，在了解这里所描述的普通摄像机的标定知识以后，你会发现它们很容易理解。

注17：这幅画起初是由浙江大学的赖百胜（Baisheng Lai）贡献给OpenCV库的。

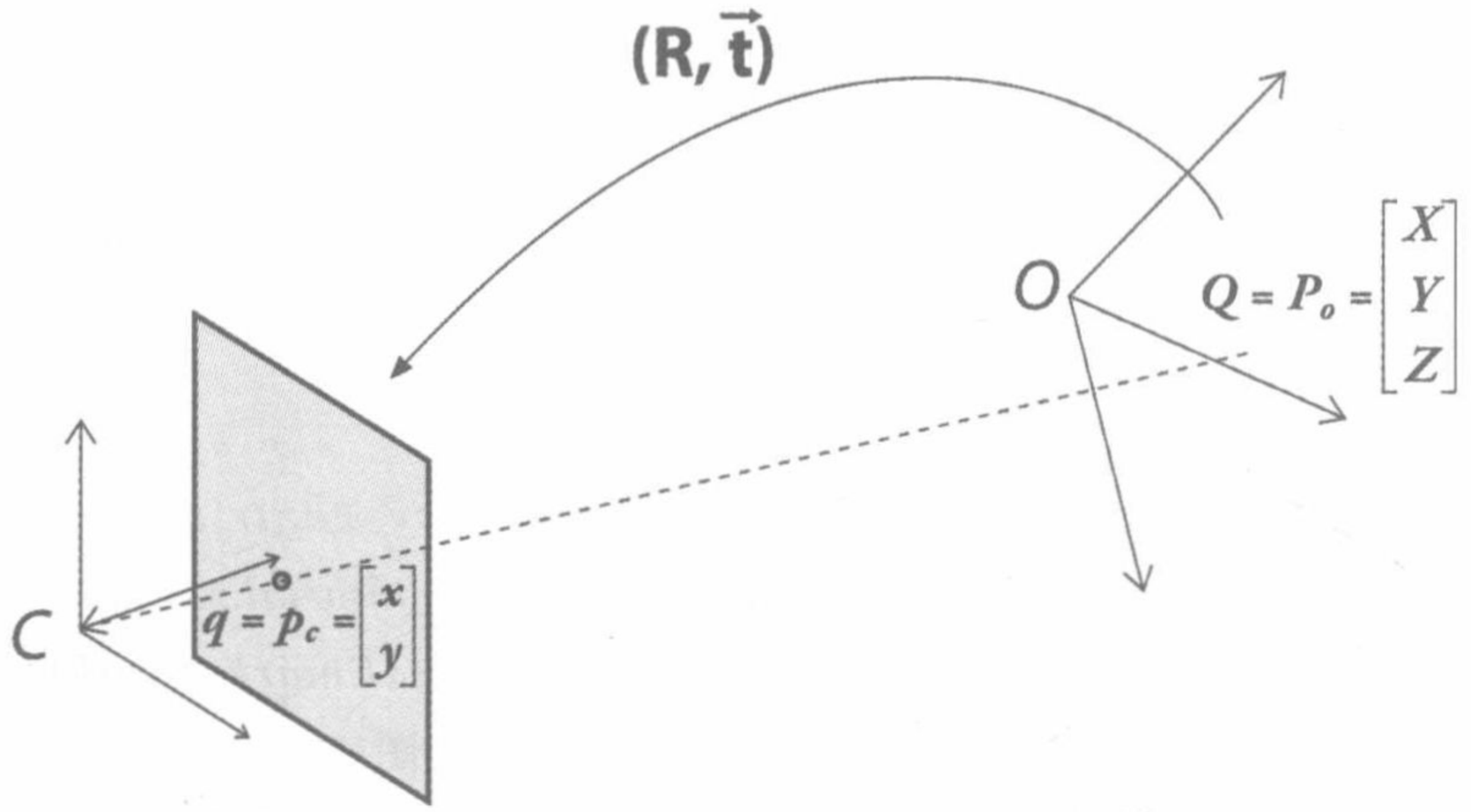


图18-8. 从物体到相机坐标系统的转换：物体上的点P对应图像平面上的点p，我们通过使用旋转矩阵R和平移向量t将点p和点P关联起来

通常，任何维数的旋转可以描述为坐标向量和适当尺寸的方阵的乘积。最终，旋转等同于在一个新的坐标系统中对点的位置引入一个新的描述。将坐标系统旋转θ的角度等同于将目标点绕此坐标原点反方向旋转同样的角度θ。用矩阵乘法表示二维旋转如图18-9所示。三维旋转可以分解成围绕每个轴的二维旋转，其中旋转轴的度量保持不变。如果依次[注18]绕x，y和z轴旋转角度ψ,ϕ和θ，总的旋转矩阵R是矩阵$R_x()$，$R_y(\phi)$和$R_z(\theta)$的乘积，其中：

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{bmatrix}$$

$$R_y(\psi) = \begin{bmatrix} \cos\varphi & 0 & -\sin\varphi 0 \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\varphi \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

因此$R = R_x(\psi) \cdot R_y(\varphi) \cdot R_z(\theta)$。旋转矩阵R具有其逆矩阵等于其转置的特性（我们只需旋转回来），因此我们有$R^T \cdot R = R \cdot R^T = I_3$,，其中，$I_3$是对角元素为1；其余元素为0的$3 \times 3$单位矩阵。

注18：需要清楚一点，这里所描述的旋转是先绕z轴，然后在新位置绕y轴，最后在新位置绕x轴。

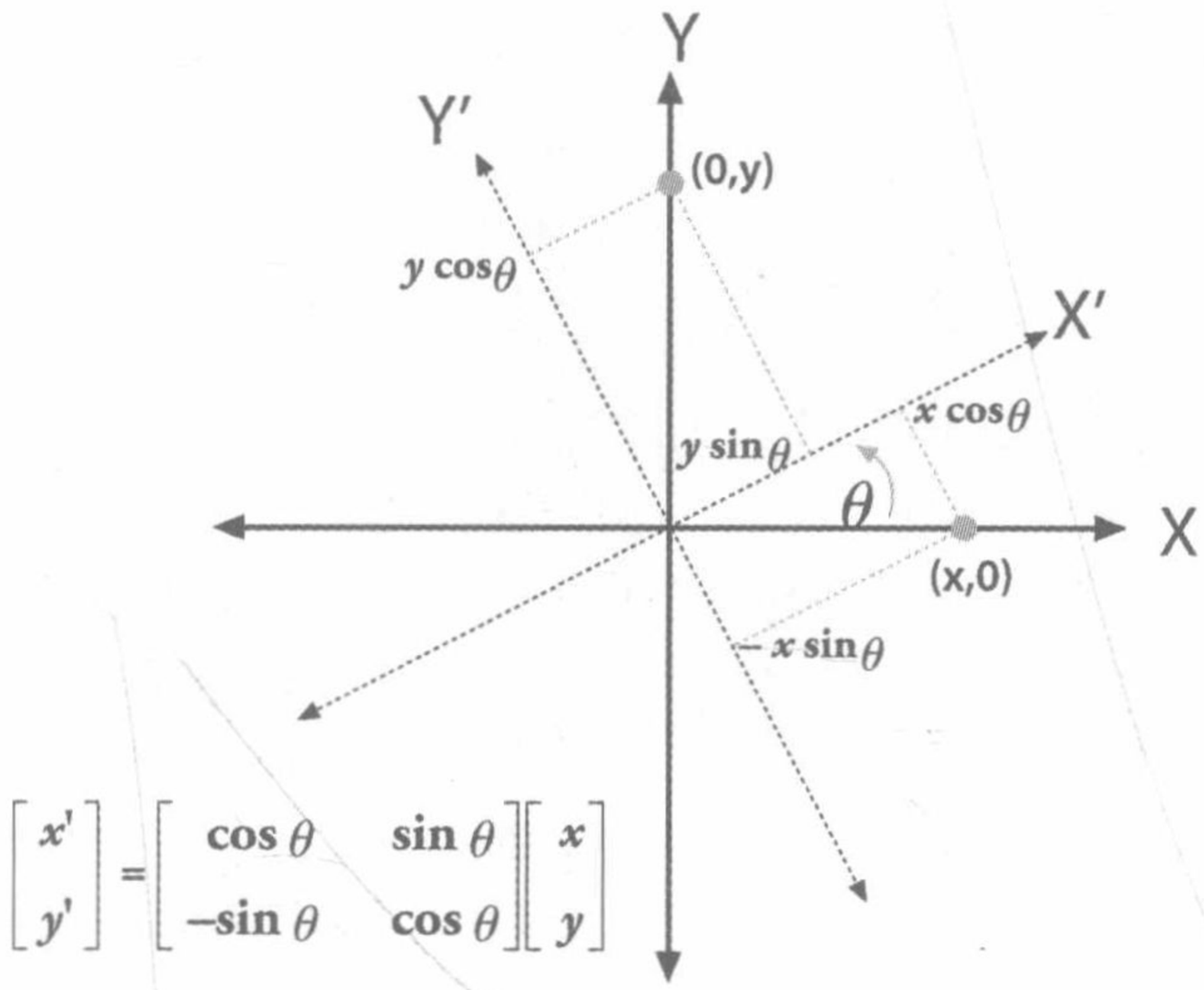


图18-9. 将点旋转角度θ（在这里，绕z轴旋转）等同于将坐标轴反向旋转θ，通过简单的三角函数，我们可以看到旋转如何改变点的坐标

平移向量用来表示如何将一个坐标系统移到另一个坐标系统，其原点也移动到另一个位置；换句话说，平移向量是第一个坐标系统的原点到第二个坐标系统的原点的偏移量。因此，从以物体中心为原点的坐标系到以相机中心为原点的坐标系，相应的平移向量为 $\vec{T} = origin_{object} - origin_{camera}$。我们因此得到（图18-8）一个点在物体（或世界）坐标系中坐标为 $\vec{P}_o$，对应相机坐标系中的坐标 $\vec{P}_c$：

$$\vec{P}_c = R \cdot (\vec{P}_o - \vec{T})$$

结合计算 $\vec{P}_c$ 的公式以及相机内参－校正，可以得到我们要求OpenCV解决的基本方程式。这些方程的解将包含我们所寻找的相机标定参数。

我们已经知道可以用三个角度来表示三维旋转，可以用三个参数(*x*, *y*, *z*)来表示三维平移，因此我们目前有六个参数。OpenCV相机内参矩阵有四个参数(f_x, f_y, c_x, and c_y)，所以每个视图都需要求解10个参数（注意，相机内在参数在不同视图保持不变）。使用一个平面物体，我们很快可以看到每个视图固定八个参数。因为在不同视图下旋转和平移的六个参数会变化，对于每个视图，我们对两个额外参数进行约束，随后使用它们来求解相机内参矩阵。因此，我们需要（至少）两个视图来求解所有的几何参数。

本章随后将进一步讲述有关参数及其约束的更多细节，但首先需要讨论标定物。OpenCV使用的标定物是下面描述的若干类型的平面图案。第一个OpenCV标定物是“棋盘”，如图18-10所示。下面的讨论将主要参考这种类型的图案，但我们也会看到其他图案也是可用的。

标定板

原则上，任何适当表征的物体都可以用作标定物。一个实际的选择是平面上的规则图案，如棋盘[注19]（图18-10）、圆网格（图18-15）、随机图案[注20]（图18-11）、ArUco（图18-12）或ChArUco图案[注21]（图18-13）。附录C具有OpenCV中所有可用的标定图案的示例。

文献中的一些标定方法是基于三维物体的（例如有标记点的盒子），但是平面棋盘图案更容易处理，因为相对于其他物体来说，制作精准二维标定物比三维标定物简单很多。因此，OpenCV选择使用平面物体的多个视图，而不是特别构造的三维物体的一个视图。目前我们将集中讨论棋盘模式。使用交替的黑色和白色方格的图案（图18-10）确保在测量中没有偏向一侧或另一侧。此外，所得到的网格角自然就可以参考第16章中讨论的亚像素定位函数。我们还将讨论另一种可用的标定板，称为“圆网格”（图18-15），它有一些所需的性质，在某些情况下可能会比棋盘的效果更好。有关其他模式，请参阅本章图中所引用的文档。作者使用ChArUco模式取得了非常好的效果。

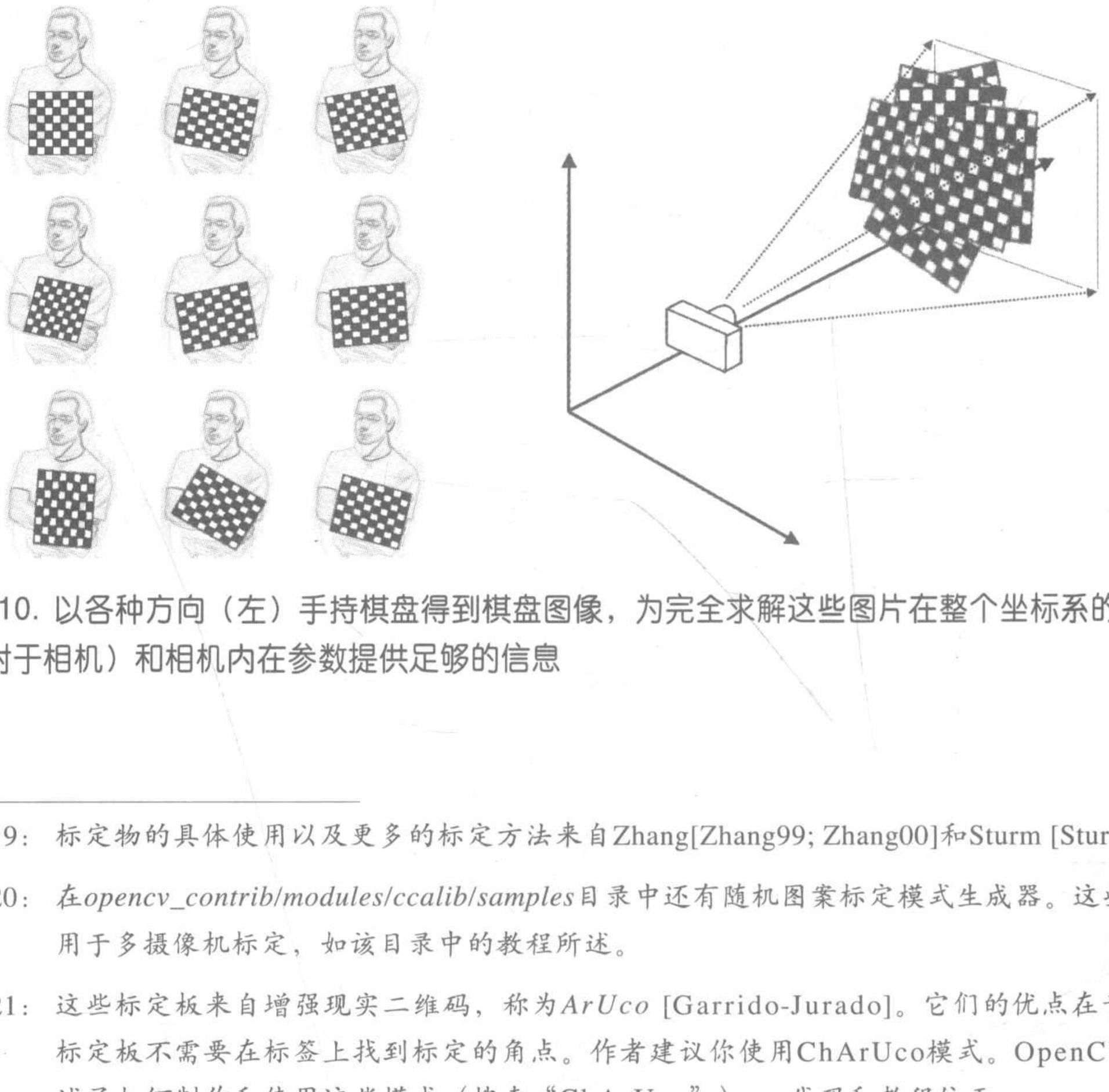


图18-10. 以各种方向（左）手持棋盘得到棋盘图像，为完全求解这些图片在整个坐标系的位置（相对于相机）和相机内在参数提供足够的信息

注19：标定物的具体使用以及更多的标定方法来自Zhang[Zhang99; Zhang00]和Sturm [Sturm99]。

注20：在*opencv_contrib/modules/ccalib/samples*目录中还有随机图案标定模式生成器。这些模式可用于多摄像机标定，如该目录中的教程所述。

注21：这些标定板来自增强现实二维码，称为*ArUco* [Garrido-Jurado]。它们的优点在于，整个标定板不需要在标签上找到标定的角点。作者建议你使用ChArUco模式。OpenCV文档描述了如何制作和使用这些模式（搜索“ChArUco”）。代码和教程位于*opencv_contrib/modules/aruco*目录中。

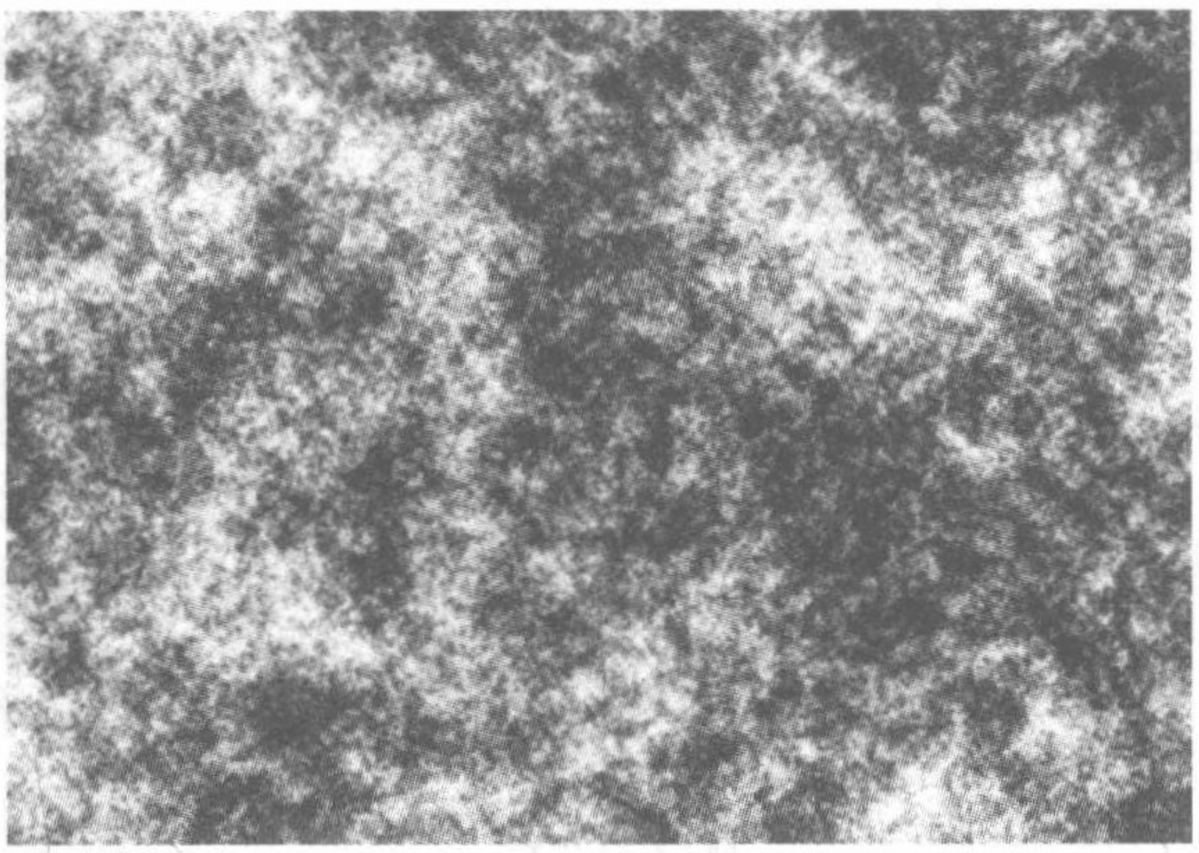


图18-11. 由高度纹理化的随机图案组成的标定图案，请参阅opencv_contrib/modules/ccalib/tutorial目录中的multamera标定教程[注22]

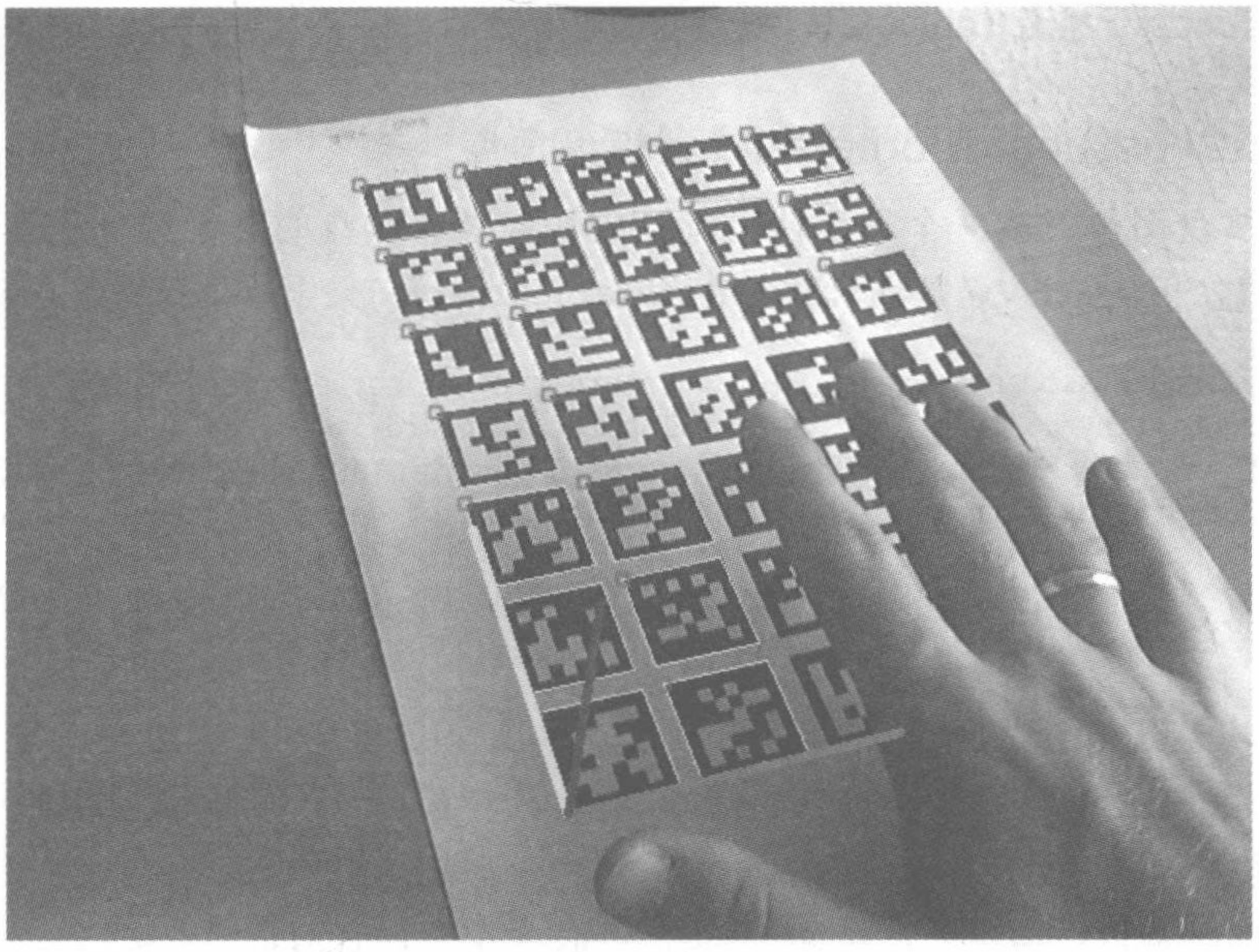


图18-12. 由ArUco（二维码）方格组成的标定图案。请注意，由于每个方格都由其ArUco图案识别，大部分可能被遮挡，但仍具有足够的空间标记点用于标定。请参阅OpenCV文档中的“ArUco标记检测（aruco模块）”[注23]

注22：这幅图起初由浙江大学的赖百胜（Baisheng Lai）贡献给OpenCV库。

注23：这幅图起初由Sergio Garrido贡献给OpenCV库。

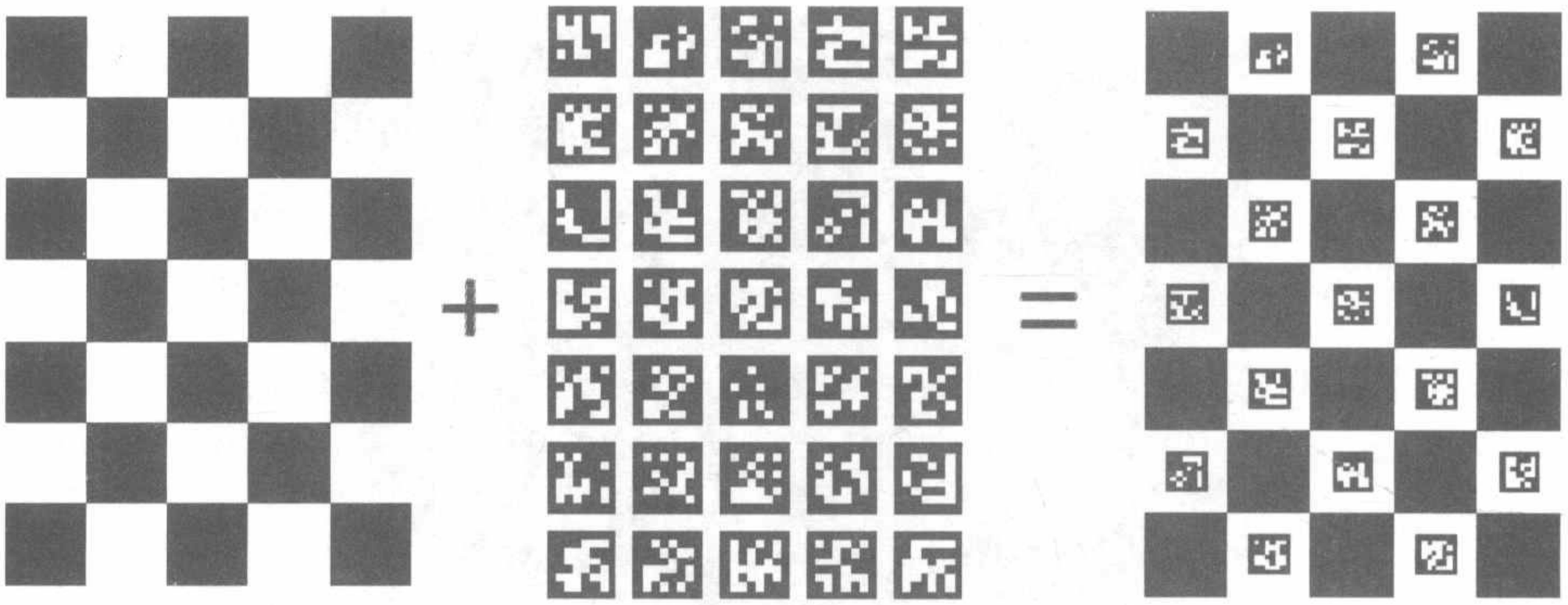


图18-13. 内部嵌有ArUco（ChArUco）的棋盘。每个部分由ArUco（二维码）图案进行标记的棋盘标定模式。允许大部分棋盘被遮挡，同时对角点达到较高的位置精度。请参阅OpenCV文档中的“ArUco标记检测（aruco模块）”[注24]

用cv::findChessboardCorners()找到棋盘角点

给定一个棋盘图像（或一个人手持棋盘，或任何具有棋盘的场景和合适的无干扰背景），可以使用OpenCV函数cv::findChessboardCorners()来定位棋盘的角点。

```
bool cv::findChessboardCorners(     // Return true if corners were found
  cv::InputArray  image,            // Input chessboard image, 8UC1 or 8UC3
  cv::Size        patternSize,      // corners per row, and per column
  cv::OutputArray corners,          // Output array of detected corners
  int             flags       = cv::CALIB_CB_ADAPTIVE_THRESH
                              | cv::CALIB_CB_NORMALIZE_IMAGE
);
```

这个函数将包含棋盘的单个图像作为参数，该图像必须是8比特图像。第二个参数patternSize表示棋盘的每行和每列有多少角点（例如cv::Size(cols,rows)）。这个值是内角点数；因此，对于一个标准的象棋棋盘，正确的值是cv::Size(7,7)。[注25]下一个参数corners是记录角点位置的输出矩阵。用像素坐标来表示角点位置的每个值。最后的参数flags用于实现一个或多个附加滤波步骤，以帮助找到棋盘上的角点。你可以使用布尔OR来组合以下任意或所有的参数：

cv::CALIB_CB_ADAPTIVE_THRESH

cv::findChessboardCorners()的默认方式是根据平均亮度对图像进行阈值化，如果设置此标志，则会使用自适应阈值法。

注24：这幅图起初由Sergio Garrido贡献给OpenCV库。

注25：在实际中，使用不对称的偶－奇维数的棋盘格通常更为方便，例如(5,6)。这种偶—奇不对称模式使棋盘只有一个对称轴，所以棋盘的方向可以唯一确定。

cv::CALIB_CB_NORMALIZE_IMAGE

如果设置此标志，则在应用阈值化操作之前使用cv::equalizeHist()来归一化图像。

cv::CALIB_CB_FILTER_QUADS

一旦图像阈值化，算法就会尝试从棋盘上的黑色方块的透视图中找出四边形。这是一个近似，因为假设四边形的每个边缘的线是直的，当图像中存在径向畸变时，就会不正确。如果设置了这个标志，就会对这些四边形应用各种附加约束，以防出现错误的四边形。

cv::CALIB_CV_FAST_CHECK

当出现此标志时，则对图像进行快速扫描，以确保图像中存在角点。如果不存在角点，则直接跳过此图像。如果你确定输入数据是“干净的”并且每幅图像中都有棋盘，则此标志就不需要存在。否则，如果在输入中存在没有棋盘的图像，那么使用该标志将节省大量的时间。

如果可以找到并排序图案中的所有角点，cv::findChessboardCorners()的返回值将被设置为true[注26]，否则为false。

棋盘上的亚像素角点和cv::cornerSubPix()

cv::findChessboardCorners()使用的内部算法仅提供角点的近似位置。因此，cv::cornerSubPix()由cv::findChessboardCorners()自动调用，以获得更准确的结果。这在实际中意味着这些位置是相对准确的。但是，如果你希望将它们定位到非常高的精度，则需要在输出上自己调用cv::cornerSubPix()（有效地再次调用它），但是需要更严格的终止条件。

使用cv::drawChessboardCorners()绘制棋盘角点

特别是在调试时，经常需要把找到的棋盘角点绘制到图像上（通常是我们第一次计算角点的图像）；这样，我们可以看到投影的角点是否与观察到的角点相匹配。为此，OpenCV提供了一个简便的例程来完成这项工作。函数cv::drawChessboardCorners()将cv::findChessboardCorners()找到的角点绘制到你提供的图像上。如果没有找到所有的角点，则可用的角点将被表示为小的红色圆圈。如果整个图案上的角点都找到，那么角点将被绘制成不同的颜色（每一行都将有自己的颜色），并将角点以一定顺序用线连接起来。

注26：在此处，排序意味着可以构建找到的点的模型，这与它们在平面上实际上是共线点的集合的命题是一致的。显然，并不是所有的图像都包含49个点，这是由平面上常规的7×7网格生成的。

```
void cv::drawChessboardCorners(
  cv::InputOutputArray image,          // Input/output chessboard image, 8UC3
  cv::Size             patternSize,    // Corners per row, and per column
  cv::InputArray       corners,        // corners from findChessboardCorners()
  bool                 patternWasFound // Returned from findChessboardCorners()
);
```

cv::drawChessboardCorners()的第一个参数是要绘制的图像。由于角点是用有颜色的圆圈表示的，因此必须为8位的彩色图像。在大多数情况下，它是你提供给cv::findChessboardCorners()的图像的一个副本（但是如果不是三通道图像，则需要转换为三通道图像）。接下来的两个参数patternSize和corners与cv::findChessboardCorners()中对应的参数一样。最后的参数patternWasFound表示是否整个棋盘图案上的角点都被成功找到，这可以设置为cv::findChessboardCorners()的返回值。图18-14显示了在棋盘图像上应用cv::drawChessboardCorners()的结果。

现在我们来看看一个平面物体比如标定板可以为我们做什么。当通过针孔或透镜观察时，平面上的点经过透视变换。该变换的参数放置在3 × 3的齐次矩阵中，在简要讨论方形网格的替代模式之后，我们将简述该部分。

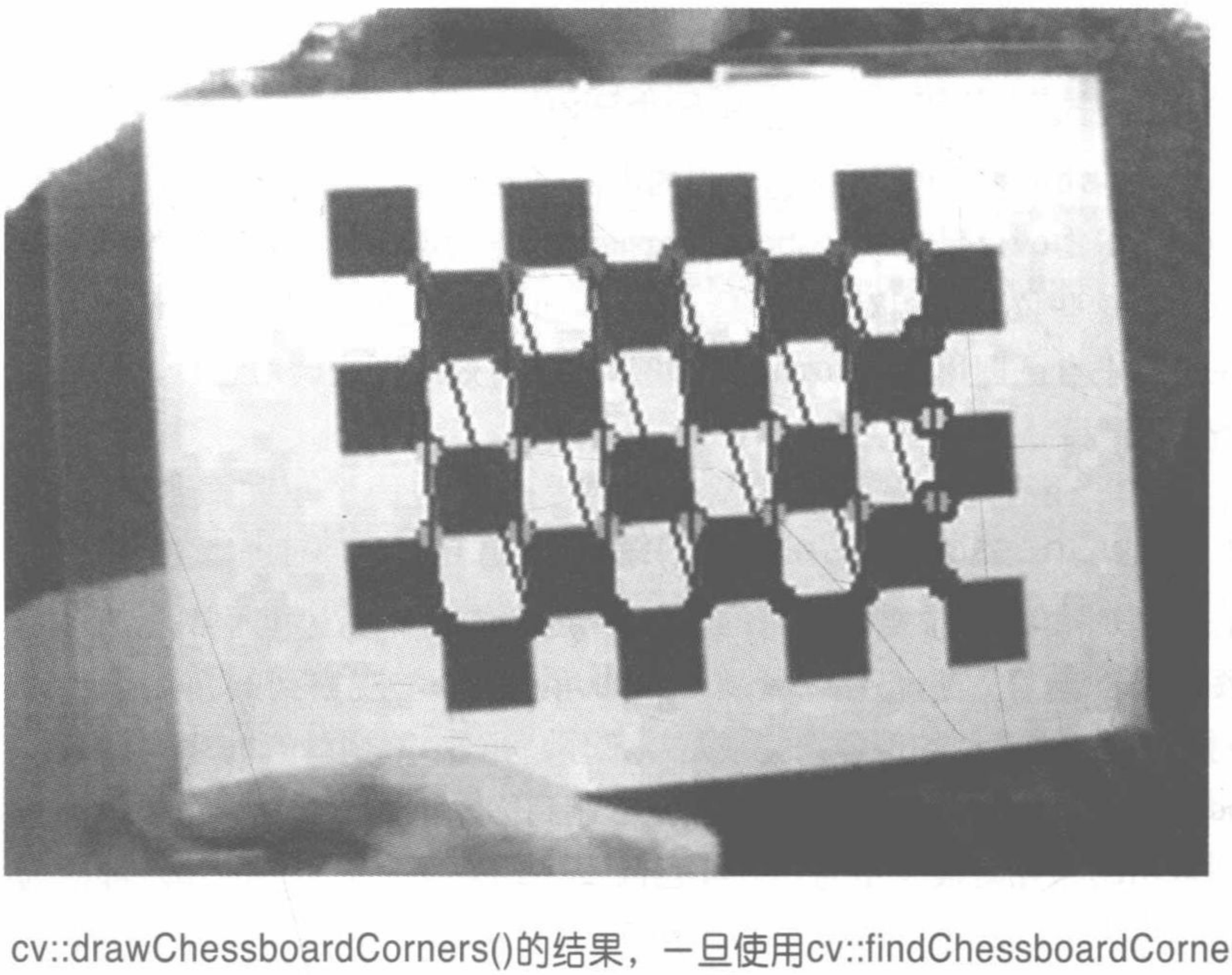


图18-14. cv::drawChessboardCorners()的结果，一旦使用cv::findChessboardCorners()找到角点，就可以对找到的角点进行映射（角点上的小圆圈）并以它们所属的位置排列（如圆圈间的直线）

圆网格和cv::findCirclesGrid()

可以用圆网格来替代棋盘。在概念上，圆网格类似于棋盘，在白色背景中包含一组黑色圆圈而不是一组交替的黑色和白色方块。

使用圆网格进行标定与使用cv::findChessboardCorners()的棋盘标定大致相同，不同之处在于调用不同的函数，并使用不同的标定图像。这个不同的函数是cv::findCirclesGrid()，它具有以下原型：

```
bool cv::findCirclesGrid(  // Return true if corners were found
  cv::InputArray  image,             // Input chessboard image, 8UC1 or 8UC3
  cv::Size        patternSize,       // corners per row, and per column
  cv::OutputArray centers,           // Output array of detected circle centers
  int             flags       = cv::CALIB_CB_SYMMETRIC_GRID,
  const cv::Ptr<cv::FeatureDetector>& blobDetector
                              = new SimpleBlobDetector()
);
```

与cv::findChessboardCorners()一样，它需要一个图像和一个cv::Size对象来定义圆形图案的数量（和布局）。它输出中心的位置，这相当于棋盘上的角点。

flags参数告诉函数如何排列圆圈。默认情况下，cv::findCirclesGrid()使用一个对称的圆形网格。“对称”网格是以与棋盘角点相同的方式将圆圈整齐地排列成行和列的网格。也可以选择非对称网格。我们通过将flags参数设置为cv::CALIB_CB_ASYMMETRIC_GRID来使用非对称网格。在“非对称”网格中，每行中的圆交错排列。图18-15所示的网格是非对称网格的示例。

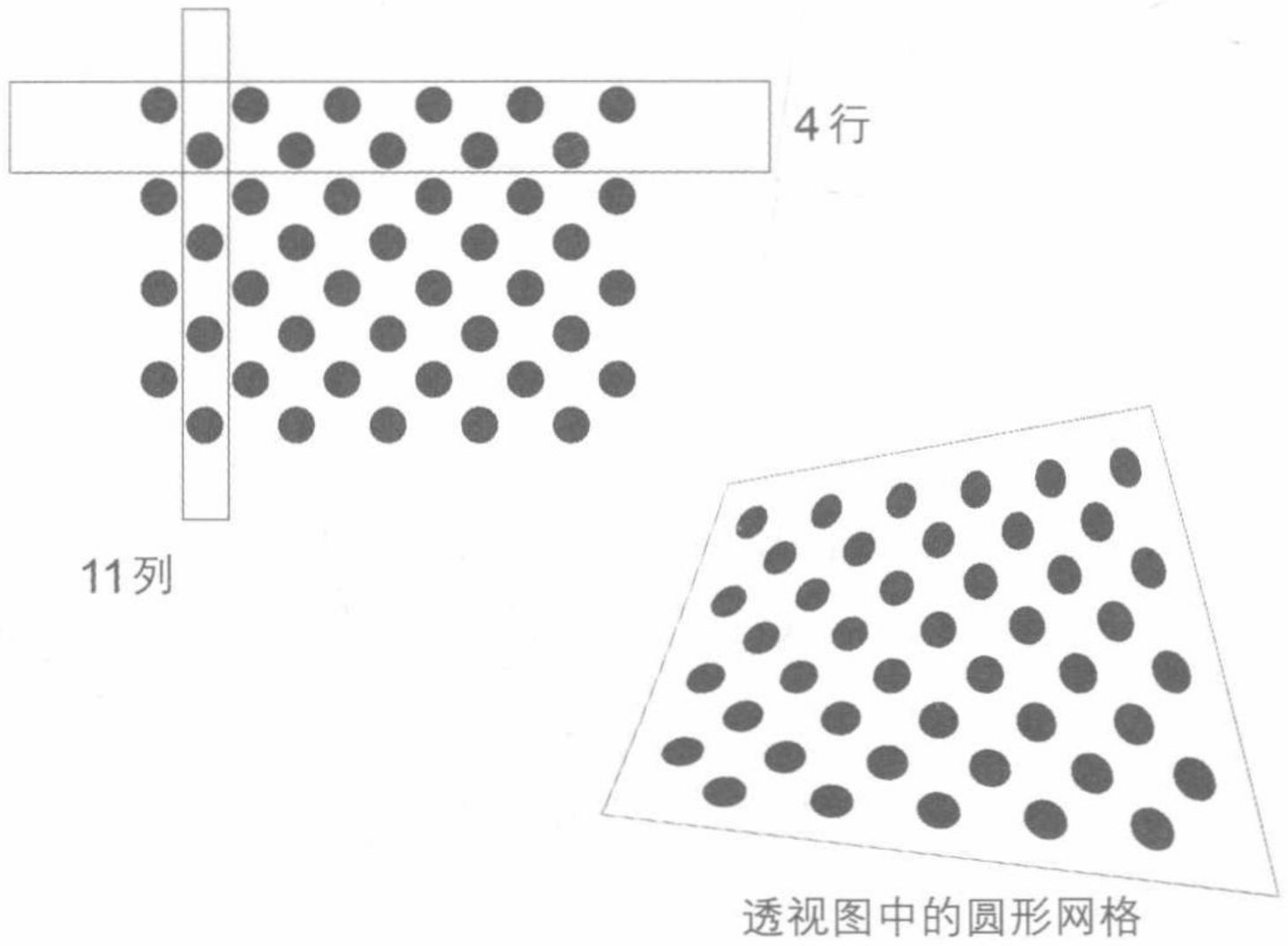


图18-15. 规则的一组圆圈（左上），圆的中心与棋盘的角点进行类似地标定。当从透视图（右下角）看时，圆的变形是规律的和可预测的

当你使用非对称网格时，记住如何计算行数和列数是很重要的。举个例子，在图18-15所示的情况下，由于是“交错”的行，所以显示的阵列只有4行和11列。flags的最后一个选项是cv::CALIB_CB_CLUSTERING。可以使用逻辑OR运算符与cv::CALIB_CB_SYMMETRIC_GRID或cv::CALIB_CB_ASYMMETRIC_GRID一起设置。如果选择此选项，则cv::findCirclesGrid()将使用稍微不同的算法来寻找圆圈。这个替代算法对透视畸变具有更高的鲁棒性，但是（造成的结果是）对背景的干扰也更加敏感。当你尝试标定具有非常宽的视野的相机时，这是一个不错的选择。



一般来说，无论是最终结果的质量还是多次运行的结果之间的稳定性，经常会发现不对称的圆网格优于棋盘。由于这些原因，不对称圆网格逐渐成为相机标定的标准工具包的一部分。在现代，诸如*ChArUco*（参见库的contrib实验代码部分）的模式也很受欢迎。

单应性

在计算机视觉中，我们将平面的单应性[注27]定义为从一个平面到另一个平面的投影映射，因此，二维平面上的点到相机的成像器上的映射就是平面单应性的一个例子。如果用齐次坐标来表示观察点$\vec{Q}$和点$\vec{Q}$映射到成像装置上的点$\vec{q}$，则可以使用矩阵乘法来表示此映射。如果我们如下定义：

$$\vec{Q} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \text{and } \vec{q} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

那么我们可以将单应性简单如下表示为：

$$\vec{q} = s \cdot H \cdot \vec{Q}$$

这里我们引入参数s，它是一个任意的比例因子（用于明确指出单应性被定义到该比例因子）。它通常是从H中分解出来的，我们在这里依然沿用这个方式。

使用一些几何和矩阵代数的知识，我们可以求解这个变换矩阵。最重要的是H有两个部分：用于定位观察的物体平面的物理变换和使用相机内参矩阵的射影变换，参见图18-16。

注27：术语“单应性”在不同学科上有不同的含义。比如，在数学上它有更通用的意义。在计算机视觉中，对单应性最感兴趣的部分只是该术语其他更通用意义的一个子集。

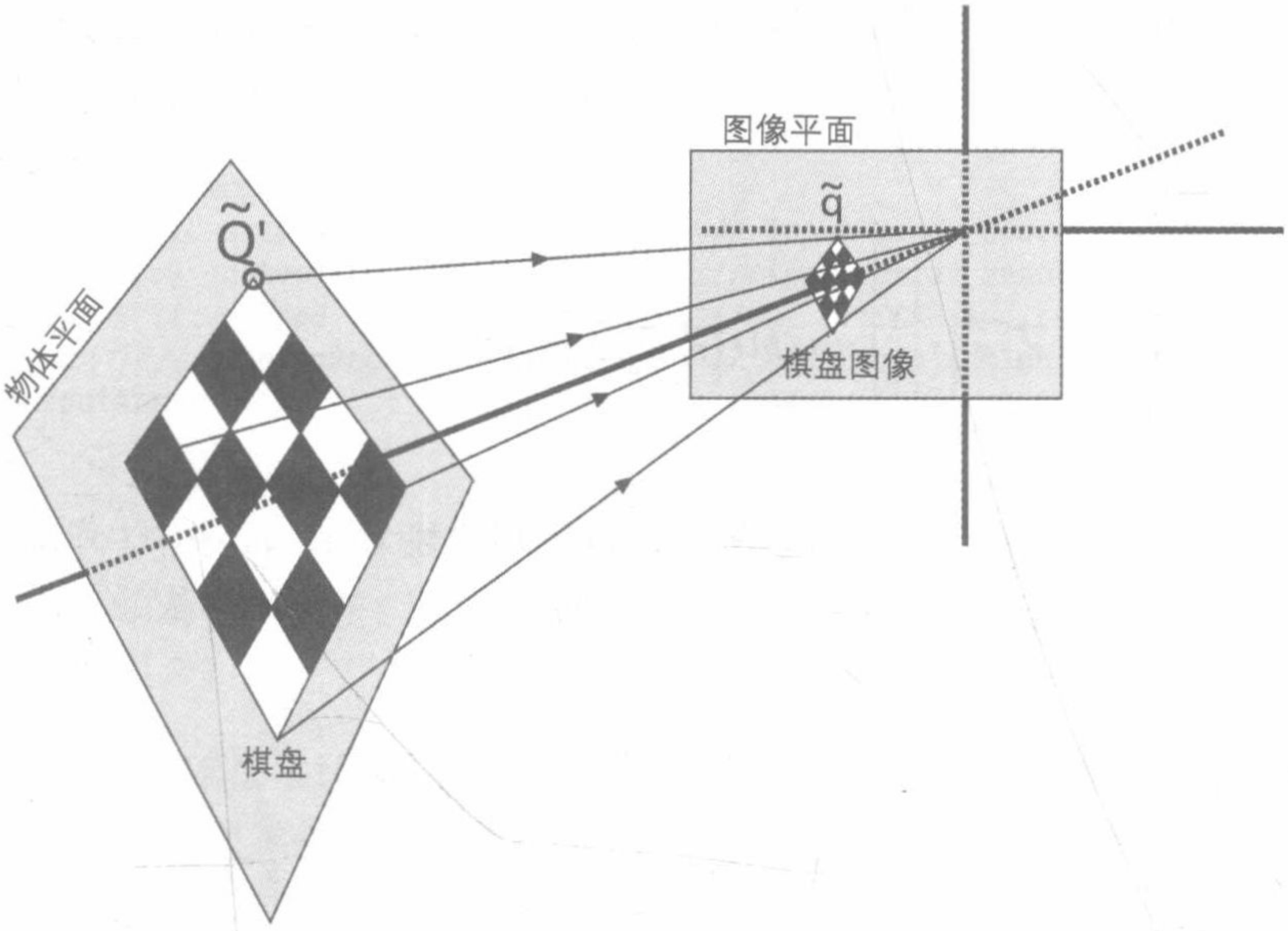


图18-16. 用单应性描述平面物体的视图：从物体平面到图像平面的映射，同时表示这两个平面的相对位置和相机的投影矩阵

物理变换部分是与观察的图像平面相关的旋转矩阵R和平移矩阵t的影响之和。因为使用齐次坐标，我们可以将它们组合到如下的矩阵中[注28]：

$$W = \begin{bmatrix} R & \vec{t} \end{bmatrix}$$

我们已经知道如何在投影坐标系中表示相机矩阵M，将该矩阵乘以$\vec{Q}$，得到：

$$\vec{q} = s \cdot M \cdot W \cdot \vec{Q}$$

其中：

$$M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_x & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

看上去我们已经完成了，但是实际上，我们的关注点不是定义在所有空间的坐标$\vec{Q}$，而只是定义在我们所观察的平面上的坐标$\vec{Q}'$。这就需要做一些简化。

注28：这里$W = \begin{bmatrix} R & \vec{t} \end{bmatrix}$是一个3 × 4矩阵前三列包含$R$的9个元素，最后一列由具有三个分量的向量所组成。

不失一般性，我们可以选择定义物体平面，使得$Z = 0$。我们这样做是因为如果将旋转矩阵分解成3个的3×1的列向量（即$R = [\vec{r}_1 \ \vec{r}_2 \ \vec{r}_3]$）），那么其中一个列向量就不需要了。具体如下：

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = s \cdot M \cdot \begin{bmatrix} \vec{r}_1 & \vec{r}_2 & \vec{r}_3 & \vec{t} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = s \cdot M \cdot \begin{bmatrix} \vec{r}_1 & \vec{r}_2 & \vec{t} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

将平面对象的点映射到成像装置的单应性矩阵H可以描述为$H = s \cdot M \cdot [\vec{r}_1 \ \vec{r}_2 \ \vec{t}]$，其中：

$$\vec{q} = s \cdot H \cdot \vec{Q}'$$

通过观察可以得出H是一个3×3矩阵。[注29]

OpenCV使用上述等式来计算单应性矩阵。它使用同一物体的多个图像来计算每个视图的平移和旋转矩阵以及内参矩阵（对于所有视图都是相同的）。正如我们所讨论的那样，旋转由三个角度来描述，平移由三个偏移量来定义；因此每个视图有六个未知数。这没有问题，因为一个已知的平面物体（如我们的棋盘）给出了八个方程——也就是映射一个正方形到四边形可以用四个(x, y)点来表示。每个新的图像帧给出了八个方程，代价是引入了六个新的外部未知数，所以给出足够多的图像，我们就可以计算任意数量的内在参数（比此处可以更多）。

单应性矩阵H通过以下简单方程将源图像平面上的点的位置与目标图像平面（通常是成像平面）上的点相关联：

$$\vec{p}_{dst} \equiv \begin{bmatrix} x_{dst} \\ y_{dst} \\ 1 \end{bmatrix} = H \cdot \vec{p}_{src}, \qquad \vec{p}_{src} \equiv \begin{bmatrix} x_{src} \\ y_{src} \\ 1 \end{bmatrix} = H^{-1} \cdot \vec{p}_{dst}$$

请注意，我们可以在不知道相机内在参数的情况下计算H。实际上，从多个视图中计算多个单应性矩阵是OpenCV用于求解相机内在参数的方法，我们将在下文中看到。

OpenCV为我们提供了一个简便的函数，即`cv::findHomography()`，它将一系列对应点作为输入，返回最能描述这些对应点的单应性矩阵。我们至少需要四点来求解H，但如果

注29：细心的读者会注意到，我们有一点忽略因子s了。在最后的表达式中出现的s与前面的方程式中出现的不一样，但不是它们的产物。在任何情况下，两个任意比例因子的乘积本身就是任意的比例因子，所以没有问题。

我们有更多点的话[注30]，可以提供更多的点（因为我们会使用大于3×3的棋盘）。使用更多的点是很有用的，因为这样可以减小噪音和其他不确定因素的干扰：

```
cv::Mat cv::findHomography(
  cv::InputArray  srcPoints,                  // Input array source points (2-d)
  cv::InputArray  dstPoints,                  // Input array result points (2-d)
  cv::int         method                 = 0, // 0, cv::RANSAC, cv::LMEDS, etc.
  double          ransacReprojThreshold = 3, // Max reprojection error
  cv::OutputArray mask                   = cv::noArray() // use only non-zero pts
);
```

输入矩阵srcPoints和dstPoints分别包含原始平面和目标平面中的点。这些都是二维点，因此它们必须是$N \times 2$或$N \times 1$的CV_32FC2的矩阵，或cv::Point2f的STL vector对象（或这些的任意组合）。

输入变量method用于决定计算单应性矩阵所使用的算法。如果使用默认值0，则会考虑所有点，计算结果会使重新投影的误差最小。在这种情况下，重新投影的误差是H乘以“原始”点与目标点之间的欧氏距离的平方之和。

方便的是，在使用这种误差计算方法的情况下，存在解决这些问题的快速算法。然而，不幸的是，以这种方式定义的误差会导致系统中一些离群数据点与大多数点具有完全不同的解，并对最终的解有很大影响。在实际情况下，例如相机标定，测量误差通常会产生异常值，由于这些异常值，所得到的结果通常与正确结果相去甚远。OpenCV提供了三种可靠的拟合方法，可用作备选方案，并且在存在噪声的情况下具有更好的表现。

第一个选择是把method设置为cv::RANSAC，使用RANSAC方法（也称为“一致性随机抽样”方法）。在RANSAC方法中，从所提供的点集中随机选择子集，并为该子集计算单应性矩阵。然后对所有剩余的数据点根据与该初始估计的一致性进行区分。“内部点”是那些一致的点，而“异常点”是那些不一致的点。RANSAC算法计算许多这样的随机样本，并保留具有最大内部点的样本。这种方法在实践中非常有效地避免异常数据的干扰，并找到正确结果。

第二种替代方案是LMeDS算法（也称为“最小平方中值”算法）。顾名思义，LMeDS背后的思想是最小化中位数误差，而不是使用默认的最小化均方误差。[注31]

注30：当然，要保证精确解，我们只需要4个对应点。如果提供更多的对应点，计算结果将是最小二乘误差意义上的最优解（可能也有一些点不计算，参见后面即将讨论的RANSAC）。

注31：要获得鲁棒方法的更多信息，请参考原始论文：Fischler和Bolles [Fischler81]的RANSAC; Rousseeuw [Rousseeuw84]的最小平方中值；Inui，Kaneko和Igarashi [Inui03]的使用LMedS进行直线拟合。

LMeDS的优点是它在运行时不需要任何进一步的信息或参数。缺点是只有当大部分数据点均为内部点时，它才能表现良好。相比之下，RANSAC可以正常工作，并在任何信噪比下，几乎都能给出令人满意的答案。然而，这样做的代价是，必须告诉RANSAC哪些点是“大致一致”的——原图像中重新投影点的最大距离仍然有必要包括在细分模型中。如果你使用cv::RANSAC方法，则输入ransacReprojThreshold控制此距离。如果使用任何其他方法，则可以忽略此参数。



ransacReprojThreshold的值以像素为单位。对于大多数实际的情况，将其设置为小的整数值（即小于10）就足够了，但是通常情况下，对于非常高分辨率的图像，则必须要增加这个数字的值。

最后，在[Bazargani15]中引入了可用于OpenCV 3的RHO算法，它是基于“加权的”RANSAC的修改算法PROSAC，并且在存在许多异常点的情况下运行更快。

最后一个参数mask只适用于鲁棒的方法，它是一个输出。如果提供了一个矩阵，cv::findHomography()将填充该矩阵，指示在计算H的最佳值时实际使用了哪些点。

返回值为3 × 3矩阵。因为单应性矩阵中只有8个自由参数，我们进行标准化使$H_{33} = 1$（通常是可行的，除了非常罕见的奇异情况$H_{33} = 0$）。单应性矩阵的缩放可以应用于第九个单应性矩阵的参数，但通常倾向于通过将整个单应性矩阵乘以比例因子来缩放，如本章前面所述。

相机标定

最后到了为相机内在参数和畸变参数进行相机标定。在本节中，我们将介绍如何使用cv::calibrateCamera()计算这些值，以及如何使用这些模型来矫正标定后的相机产生的图像中的畸变。首先，我们将说明为了求解内在参数和畸变参数，需要多少个棋盘视图。随后在转到简化这个工作的源代码之前，我们会就OpenCV如何求解这个系统进行概述。

棋盘角点个数和参数个数

首先，回顾我们的未知参数将是有启发性的，也就是说，我们通过标定要求解多少个参数。在OpenCV中，我们有四个相机内在参数（f_x, f_y, c_x, c_y）和五个（或更多个）畸变参数 ——由三个（或更多个）径向参数（k_1, k_2, k_3[, k_4, k_5, k_6]）和两个切向参数（p_1, p_2）组

成。[注32]内在参数控制实际物体与生成的图像之间的线性射影变换。因此，它们与外参矩阵合在一起，告诉我们该物体实际位于何处。

畸变参数与点集在最终图像中畸变的二维几何学有关。原则上，已知图案的三个角点，产生六条信息，可能都需要用于求解我们的五个畸变参数。因此，看起来只需要标定棋盘的一个视图就足够了。

然而，由于内在参数和外参数之间存在耦合，所以一个视图是不够的。为了理解这一点，首先要注意的是外参数包括三个旋转参数（ψ, ϕ, θ）和三个平移参数（T_x, T_y, T_z），每个棋盘视图共有六个外参数。由相机内参矩阵的四个参数和六个外参数共同构成十个需要求解的参数，在单个视图的情况下，每个额外的视图就会增加6个参数。

假设有N个角点和K个棋盘图像（不同位置）。我们需要看到多少视图和角点才能有足够的约束条件来求解所有这些参数？

- K个棋盘图像提供$2 \cdot N \cdot K$个约束（出现因子2是因为图像上的每个点都具有x和y两个坐标值）
- 忽略每次的畸变参数，我们有4个内在参数和$6 \cdot K$个外参数（因为我们需要在K个视图中找到棋盘位置的6个参数）。
- 求解的前提是$2 \cdot N \cdot K \geq 6 \cdot K + 4$（或等效为$(N-3) \cdot K \geq 2$）。

所以看起来如果$N = 5$，那么我们只需要$K = 1$的图像，但要注意！对我们来说，K（图像数量）必须大于1。要求$K > 1$的原因是我们使用棋盘进行标定，为K个视图拟合出一个单对应矩阵。如前所述，单应性矩阵可以从四个（x, y）对中产生最多八个参数。这是因为只需要四个点就可以表达平面透视图可以做的一切：它可以在四个方向上伸展正方形的边（参见第11章中的透视图像）。所以，无论我们在平面上发现多少角点，我们只得到四个有用的角点信息。对于每个棋盘视图，方程只能给我们四个角点信息，或者$(4-3) \cdot K > 1$，即$K > 1$。这意味着3×3棋盘（仅计算内部角点）至少需要两个视图才能求解我们的标定问题。考虑到噪声和数值稳定性，通常需要收集更大的棋盘的更多图像。在实践中，为了获得高质量的效果，需要至少10张7×8或更大的棋盘图像（只有在图像之间移动足够次数的棋盘才能获得“丰富的”视图）。

理论上至少需要2个图像，而实际上需要10个或更多视图，这种差异是因为内在参数对非常小的噪声具有非常高的灵敏度。

注32：通常将相机内参矩阵的参数和畸变参数的全部集合简单称为固有参数或内在参数。在一些情况下，矩阵参数也称为线性固有参数（因为它们共同定义线性变换），而畸变参数称为非线性固有参数。

实质是什么?

本小节是为想深入学习的人准备的，如果你只想调用标定函数，则完全可以跳过此节。

如果你还和我们在一起，问题仍然存在：实际中数学如何用于标定？虽然有很多方法来求解相机参数，但OpenCV选择了一种在平面物体上表现良好的方法。OpenCV用于求解焦距和偏移的算法是基于Zhang的方法[Zhang00]，但OpenCV使用另一个基于Brown[Brown71]的方法来求解畸变参数。

首先，我们假设在求解其他标定参数时相机没有畸变。对于棋盘的每个视图，我们得到单应性矩阵H，如前所述（即，从物理对象到成像装置的映射）。我们将把H作为列向量写出，$H = [\vec{h}_1, \vec{h}_2, \vec{h}_3]$，其中$h$是$3 \times 1$向量。随后根据前面对单应性矩阵的讨论，我们可以将$H$设置为相机内参矩阵$M$乘以前两个旋转矩阵$\vec{r}_1$和$\vec{r}_2$与平移向量$\vec{t}$的组合矩阵。再加上比例因子$s$，得到：

$$H = [\vec{h}_1, \vec{h}_2, \vec{h}_3] = s \cdot M \cdot [\vec{r}_1, \vec{r}_2, \vec{t}]$$

分解这些方程，得到：

$$\vec{h}_1 = s \cdot M \cdot \vec{r}_1 \text{ or } \quad \vec{r}_1 = \lambda \cdot M^{-1} \cdot \vec{h}_1$$

$$\vec{h}_2 = s \cdot M \cdot \vec{r}_2 \text{ or } \vec{r}_2 = \lambda \cdot M^{-1} \cdot \vec{h}_2$$

$$\vec{h}_3 = s \cdot M \cdot \vec{t} \text{ or } \vec{t} = \lambda \cdot M^{-1} \cdot \vec{h}_3$$

这里有：

$$\lambda = \frac{1}{s}$$

旋转向量通过构造是相互正交的，由于比例因子提到了前面，则$\vec{r}_1$和$\vec{r}_2$相互正交。正交矩阵表明两件事：旋转向量的点积为0，而且向量的模相等。从点积开始，我们有：

$$\vec{r}_1^{\,T} \cdot \vec{r}_2 = 0$$

对任意向量$\vec{a}$和$\vec{b}$，我们有$(\vec{a} \cdot \vec{b})^T = \vec{b}^T \cdot \vec{a}^T$，因此替换$\vec{r}_1$和$\vec{r}_2$，得到我们的第一个约束：

$$\vec{h}_1^{\,T} \cdot M^{-T} \cdot M^{-1} \cdot \vec{h}_2 = 0$$

其中M^{-T}是$(M^{-1})^T$的简写，已知旋转向量的模相等：

$$\| \vec{r}_1 \| = \| \vec{r}_2 \| \text{ 或 } \vec{r}_1^{\,T} \cdot \vec{r}_1 = \vec{r}_2^{\,T} \cdot \vec{r}_2$$

替换$\vec{r}_1$和$\vec{r}_2$，得到第二个约束：

$$\vec{h}_1^T \cdot M^{-T} \cdot M^{-1} \cdot \vec{h}_1 = \vec{h}_2^T \cdot M^{-T} \cdot M^{-1} \cdot \vec{h}_2$$

为了对公式进行简化，定义$B = M^{-\mathrm{T}} \cdot M^{-1}$。展开得：

$$B = M^{-T} \cdot M^{-1} \equiv \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix}$$

矩阵B有封闭形式的通解：

$$B = \begin{bmatrix} \frac{1}{f_x^2} & 0 & -\frac{c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & -\frac{c_y}{f_y^2} \\ -\frac{c_x}{f_x^2} & -\frac{c_y}{f_y^2} & \left(\frac{c_x}{f_x^2} + \frac{c_y}{f_y^2} + 1\right) \end{bmatrix}$$

使用B矩阵，两个约束都有其通用形式$\vec{h}_i^T \cdot B \cdot \vec{h}_j$，让我们将矩阵乘开，来看看每一项的组成。因为$B$是对称的，它可以写成六维向量的点积。重新排列$B$的元素，得到新向量$\vec{b}$，我们有：

$$\vec{h}_i^T \cdot B \cdot \vec{h}_j = \vec{v}_{i,j}^T \cdot \vec{b} =$$

$$\begin{bmatrix} h_{i,1}h_{j,1} & \left(h_{i,1}h_{j,2} + h_{i,2}h_{j,1}\right) & h_{i,2}h_{j,2} & \left(h_{i,3}h_{j,1} + h_{i,1}h_{j,3}\right) & \left(h_{i,3}h_{j,2} + h_{i,2}h_{j,3}\right) & h_{i,3}h_{j,3} \end{bmatrix}$$

$$\cdot \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix}$$

使用这个定义$\vec{v}_{i,j}^T$，两个约束可以写成：

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} \cdot \vec{b} = 0$$

如果同时采集到K个棋盘图像，那么我们可以将K个这些方程组合在一起：

$$V \cdot \vec{b} = 0,$$

其中V是$2 \cdot K \times 6$的矩阵。如前所述，如果$K \geq 2$，则方程对$\vec{b} = [B_{11} \quad B_{12} \quad B_{22} \quad B_{13} \quad B_{23} \quad B_{33}]^T$有解。相机内在参数可以从$B$矩阵的封闭形式的解中直接得到。

$$f_x = \sqrt{\lambda / B_{11}}$$

$$f_y = \sqrt{\frac{\lambda B_{11}}{B_{11}B_{22} - B_{12}^2}}$$

$$c_x = \frac{B_{13} f_x^2}{\lambda}$$

和

$$c_y = \frac{B_{12}B_{13} - B_{11}B_{23}}{B_{11}B_{22} - B_{12}^2}$$

其中：

$$\lambda = B_{33} - \frac{\left(B_{13}^2 + c_y(B_{12}B_{13} - B_{11} - B_{23})\right)}{B_{11}}$$

外参数（旋转和平移）可以从单应性条件的方程中计算得到：

$$\vec{r}_1 = \lambda \cdot M^{-1} \cdot \vec{h}_1$$

$$\vec{r}_2 = \lambda \cdot M^{-1} \cdot \vec{h}_2$$

$$\vec{r}_3 = \vec{r}_1 \times \vec{r}_2$$

和

$$\vec{t} = \lambda \cdot M^{-1} \cdot \vec{h}_3$$

这里的比例参数由正交条件 $\lambda - 1/\| M^{-1} \cdot \vec{h} \|$ 来确定。需要小心一些，因为当我们使用真实数据求解并将r个向量放到一起（$R = [\vec{r}_1 \ \vec{r}_2 \ \vec{r}_3]$），我们最终可能无法得到一个使$R^T R = RR^T = I_3$成立的精确的旋转矩阵[注33]。

为了求解这个问题，常用的技巧是采用R的奇异值分解（SVD）。如第5章所讨论的，SVD是将矩阵分解为两个正交矩阵U和V以及对角线元素为比例值的中间矩阵D的方法。这允许我们将R转换成$R = U \cdot D \cdot V^T$，因为R本身是正交的，所以矩阵D必须是单位矩阵I_3。使得$R = U \cdot I_3 \cdot V^T$，因此，我们可以通过对$R$进行奇异值分解，将其$D$矩阵设置为单位矩阵，用SVD的方法来产生新的符合要求的旋转矩阵R，从而将我们计算出的R强制转化为旋转矩阵。

尽管做了所有这些工作，我们还没有处理透镜畸变。我们使用前面得到的相机内在参数以及设置为0的畸变参数——作为我们的初始值，来开始求解更大的系统方程。

由于畸变，我们对图像“感知”的点实际上是在错误的地方。如果针孔相机是完好的，令（x_p, y_p）为点的位置，并令（x_d, y_d）为其畸变位置；然后：

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} f_x \frac{X_W}{Z_W} + c_x \\ f_y \frac{Y_W}{Z_W} + c_y \end{bmatrix}$$

通过下列替换，可以得到没有畸变的标定结果：[注34]

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2\left(r^2 + 2x_d^2\right) \\ p_1\left(r^2 + 2y_d^2\right) + 2p_2 x_d y_d \end{bmatrix}$$

收集并求解这些方程以找到畸变参数，之后重新估计内在参数和外参数。这些繁重的工作仅仅使用单个函数cv::calibrateCamera()就能解决！[注35]

注33：这主要是由于精度误差。

注34：相机标定函数现在应用非常广泛，包括鱼眼和全景相机，详情请参阅*opencv_contrib/modules/ccalib/src*和*opencv_contrib/modules/ccali/samples*目录。

注35：cv::calibrateCamera()函数为第19章中的立体标定函数的内部所使用。对于立体标定，我们需要在同一时间标定两个相机，并且希望将它们通过一个旋转矩阵和一个平移向量联系起来。

标定函数

一旦我们有几个图像的角点，就可以调用cv::calibrateCamera()。这个程序会做数字处理，并给我们提供我们想要的信息。具体来说，我们得到的结果是相机内参矩阵，畸变系数、旋转向量和平移向量。前两者构成相机的内在参数，后两者是告诉我们物体（如棋盘）的位置和方向的外参数。畸变系数（k_1，k_2，p_1，p_2和任意高阶的k_j）是我们之前遇到的径向和切向畸变方程的系数；它们帮助我们矫正这些畸变。相机内参矩阵可能是最有用的结果，因为它允许我们将三维坐标转换为图像的二维坐标。我们也可以使用相机矩阵进行反向操作，但是在这种情况下，我们只能计算给定图像点对应的三维世界中的一条线。之后会再讨论这个问题。

让我们查看一下相机标定的程序：

```
double cv::calibrateCamera(
  cv::InputArrayOfArrays  objectPoints,     // K vecs (N pts each, object frame)
  cv::InputArrayOfArrays  imagePoints,      // K vecs (N pts each, image frame)
  cv::Size                imageSize,        // Size of input images (pixels)
  cv::InputOutputArray    cameraMatrix,     // Resulting 3-by-3 camera matrix
  cv::InputOutputArray    distCoeffs,       // Vector of 4, 5, or 8 coefficients
  cv::OutputArrayOfArrays rvecs,            // Vector of K rotation vectors
  cv::OutputArrayOfArrays tvecs,            // Vector of K translation vectors
  int                     flags        = 0, // Flags control calibration options
  cv::TermCriteria        criteria     = cv::TermCriteria(
    cv::TermCriteria::COUNT | cv::TermCriteria::EPS,
    30, // ...after this many iterations
    DBL_EPSILON // ...at this total reprojection error
  )
);
```

调用cv::calibrateCamera()时，你需要清楚很多参数。好消息是，我们已经介绍了（几乎）所有这些参数，所以希望你可以理解它们。

第一个参数是objectPoints。它是向量的向量，每个向量包含特定图像的标定图案上的点的坐标。那些坐标在物体坐标系中，所以可以简单使它们在x维和y维中是整数，而在z维中为零。[注36]

接下来是imagePoints参数。它也是向量的向量，并且包含每个图像中找到的每个点的位置。如果使用的是棋盘，则每个向量将是相应图像中的角点输出矩阵。

注36：原则上，你可以为每个标定图像使用不同的物体。实际上，外向量通常只包含相同的点位置列表的K个副本，其中K是视图数。

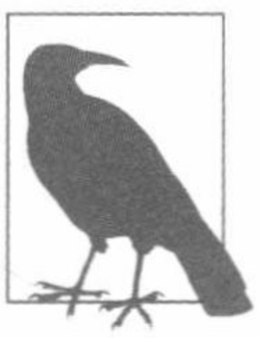


定义objectPoints输入时，将隐式更改cv::calibrateCamera的某些输出的比例。具体来说，你会影响tvecs输出。如果棋盘上的一个角点是（0, 0, 0），下一个是在（0, 1, 0），再下一个是在（0, 2, 0）等等，那么隐含地说明，你想要在“棋盘方格”中测量距离。如果想要输出的物理单位，你必须以该物理单位测量棋盘。例如，如果想要以米为单位定义距离，那么你需要测量棋盘，并用米来得到正确的方格尺寸。如果方格为25mm，则应将相同的角点设置为(0, 0, 0)，(0, 0.025, 0)，(0, 0.050, 0)等等。相反，相机内参矩阵总是以像素为单位表示的。

imageSize参数只是告诉cv::calibrateCamera()提取imagePoints中的点的图像有多大（以像素为单位）。

相机的内在参数返回到cameraMatrix和distCoeffs矩阵中。前者将包含线性内在参数，应为3×3矩阵。后者可以是4,5或8个元素。如果distCoeffs的长度为4，则返回的矩阵将包含系数（k_1，k_2，p_1和p_2）。如果长度为5或8，则元素分别为（k_1，k_2，p_1，p_2和k_3）或（k_1，k_2，p_1，p_2，k_3，k_4，k_5和k_6）。五元素形式通常只适用于鱼眼透镜。只有当你设置cv::CALIB_RATIONAL_MODEL，并且对非常高精度的特殊透镜进行校准时，才能使用八元素形式。请着重记住，你需要的图像数量将随着求解的参数数量而急剧增加。

rvecs和tvecs矩阵是向量的向量，像输入点的矩阵。它们分别包含每个棋盘的旋转矩阵（以Rodrigues形式表示，即以三分量向量表示）以及平移矩阵。



因为精度在标定中非常重要，即使你最初没有以双精度形式分配这些输入矩阵，CameraMatrix和distCoeffs矩阵（以及rvecs和tvecs矩阵）总是以双精度计算和返回。

通过优化找到参数是一个极富艺术性的过程。有时试图一次性求解所有参数可能会产生不准确或不均衡的结果，特别是在参数空间中选择的初始起始位置远离实际结果时。因此，通过分段逼近一个好的参数起始位置，通常可以更好地“解决”此问题。为此，我们经常固定一些参数来求解其他参数，然后把其他参数固定并求解原来的参数等等。最后，当我们认为我们所有的参数都接近于实际的结果时，我们把求得的参数作为起点，并求解所有的参数。OpenCV允许你通过设置flags来控制此过程。

flags参数允许对标定过程进行更精确的控制。以下值可以根据需要与布尔OR运算组合在一起。

cv::CALIB_USE_INTRINSIC_GUESS

通常，cv::calibrateCamera()计算内参矩阵时不需要其他信息。具体来说，参数c_x和c_y（图像中心）的初始值直接从imageSize参数中得到。如果设置此参数，则假定CameraMatrix中包含有效值，该值将作为初始猜测值被cv::calibrateCamera()进一步优化。



在许多实际应用中，我们知道相机的焦距，因为我们可以从镜头的侧面读取它们。[注37]在这种情况下，将这些信息放入相机矩阵中并使用cv::CALIB_USE_INTRINSIC_GUESS是一个好主意。在大多数情况下，使用cv::CALIB_FIX_ASPECT_RATIO也是可行的（也是一个好主意），稍后再讨论。

cv::CALIB_FIX_PRINCIPAL_POINT

该标志可以与cv::CALIB_USE_INTRINSIC_GUESS结合使用，也可以单独使用。如果单独使用，则主点固定在图像中心；如果共同使用，则主点固定在cameraMatrix中提供的初始值。

cv::CALIB_FIX_ASPECT_RATIO

如果设置这个标志，那么在调用标定程序时，优化过程将一起改变f_x和f_y，并且它们的比值保持在cameraMatrix中设置的值。如果没有设置cv::CALIB_USE_INTRINSIC_GUESS标志，那么cameraMatrix中的f_x和f_y的值可以是任意值，只是它们的比值是相关的。

cv::CALIB_FIX_FOCAL_LENGTH

该标志在优化时直接使用cameraMatrix中传递的f_x和f_y。

cv::CALIB_FIX_K1, cv::CALIB_FIX_K2, ... cv::CALIB_FIX_K6

修正径向畸变参数k_1，k_2到k_6。可以通过组合这些标志来设置径向参数。

cv::CALIB_RATIONAL_MODEL

该标志告诉OpenCV计算k_4，k_5和k_6三个畸变系数。这是因为向后兼容性问题，如果不添加此标志，则只计算前三个k_j参数（即使你为distCoeffs提供了一个八元素矩阵）。

cv::calibrateCamera()的最后一个参数是终止标准。通常，终止标准可以是迭代次

注37：这是一个轻微的简化。回想一下，内参矩阵中出现的焦距是以像素为单位进行测量的。所以如果你有一个1 / 1.8“传感器格式的2 048 × 1 536像素的成像装置，它给你3.45μm像素。如果你的相机具有25.0mm焦距的透镜，则f的初始猜测值不是25（mm），而是7 246.38（像素），或者如果你逼近到有效数字，则仅为7 250。

数，“epsilon”值或两者都有。在使用epsilon值的情况下，计算的是重投影误差。与cv::findHomography()一样，重映射误差是三维点到图像平面上的计算（投影）位置与原始图像中相应点的实际位置之间的距离平方之和。



使用非对称圆网格进行相机标定越来越普遍。在这种情况下，请务必记住objectPoints参数必须进行相应的设置。例如，图18-15中的对象点的一组可能的坐标是(0, 0, 0），（1, 1, 0)，(2, 1, 0)，(3, 1, 0)等等，下一行是(0, 2, 0)，(1, 3, 0)，(2, 2, 0)，(3, 3, 0)等等，以此遍历所有的行。

仅使用cv::solvePnP()计算外参数

在某些情况下，我们已经知道了相机的内在参数，因此只需要计算正在观察的对象的位置。这种情况与一般的相机标定明显不同，但它仍然是有用的工作。[注38]一般来说，此工作称为*N*点透视（Perspective *N*-Point）或PnP问题：

```
bool cv::solvePnP(
  cv::InputArray  objectPoints,              // Object points (object frame)
  cv::InputArray  imagePoints,               // Found pt locations (img frame)
  cv::InputArray  cameraMatrix,              // 3-by-3 camera matrix
  cv::InputArray  distCoeffs,                // Vector of 4, 5, or 8 coeffs
  cv::OutputArray rvec,                      // Result rotation vector
  cv::OutputArray tvec,                      // Result translation vector
  bool            useExtrinsicGuess = false, // true='use vals in rvec and tvec'
  int             flags             = cv::ITERATIVE
);
```

cv::solvePnP()的参数与cv::calibrateCamera()的对应参数类似，但是有两个不同的地方。首先，objectPoints和imagePoints参数是来自物体的单个视图的参数（即，它们的类型为cv::InputArray，而不是cv::InputArrayOfArrays）。其次，内参矩阵和畸变系数是直接提供的而不必计算（即，它们是输入而不是输出）。所输出的旋转向量以Rodrigues形式表示：由三个部分组成的旋转向量表示棋盘或点旋转的三维坐标轴，向量的幅度或长度代表逆时针旋转角度。这个旋转向量可以通过cv::Rodrigues()函数转换成我们之前讨论过的3 × 3旋转矩阵。平移向量是相机坐标中棋盘原点的偏移量。

useExtrinsicGuess参数可以设置为true，以表示rvec和tvec参数中的当前值应被视为求解的初始猜测值。默认值为false。

最后一个参数flags可以设置为三个值之一，即cv::ITERATIVE，cv::P3P或cv::EPNP，以表明应该使用哪种方法来求解整个系统。当使用cv::ITERATIVE时，会使用Levenberg-

注38：事实上，这个任务是相机标定的总体任务的一个组成部分，而这个函数是由cv::calibrateCamera()内部调用的。

Marquardt优化方法来最小化输入imagePoints和objectPoints的投影值之间的重投影误差。当使用cv::P3P时，则使用基于[Gao03]的方法，在这种情况下，应提供四个对象和四个图像点。cv::SolvePNP的返回值只有在方法成功时才为true。最后，当使用cv::EPNP时，则使用[Moreno-Noguer07]中描述的方法。需要注意的是，后两种方法都不需要迭代，因此，要比cv::ITERATIVE快得多。



虽然我们介绍了cv::solvePnP()用来计算多个帧中的物体（例如，棋盘）相对于静止的相机的姿态，但是相同的函数也可以有效地求解其逆问题。例如在使用移动机器人的情况下，我们对静止物体（可能是固定物体，也许仅仅是整个场景）和移动相机的情况更感兴趣。在这种情况下，你仍然可以使用cv::solvePnP()，唯一的区别是如何解释所得的rvec和tvec向量。

只用cv::solvePnPRansac()计算外参数

cv::solvePnP的一个缺点是对异常值不够鲁棒。在相机标定时，这不是一个问题，主要是因为棋盘本身为我们提供一种可靠的方法来找到我们关心的各个特征，并通过它们的相对几何位置来验证我们正在看的实物和我们认为的一致。然而，当我们用相机定位的不是棋盘上的点而是真实世界中的点（例如，使用稀疏关键点特征）时，可能发生错配并导致严重的问题。回想一下我们在前面讨论单应性时讲过RANSAC方法可以成为处理这种离群值的有效方法：

```
bool cv::solvePnPRansac(
  cv::InputArray  objectPoints,              // Object points (object frame)
  cv::InputArray  imagePoints,               // Found pt locations (img frame)
  cv::InputArray  cameraMatrix,              // 3-by-3 camera matrix
  cv::InputArray  distCoeffs,                // Vector of 4, 5, or 8 coeffs
  cv::OutputArray rvec,                      // Result rotation vector
  cv::OutputArray tvec,                      // Result translation vector
  bool            useExtrinsicGuess = false, // read vals in rvec and tvec ?
  int             iterationsCount   = 100,   // RANSAC iterations
  float           reprojectionError = 8.0,   // Max error for inclusion
  int             minInliersCount   = 100,   // terminate if this many found
  cv::OutputArray inliers           = cv::noArray(), // Contains inlier indices
  int flags = cv::ITERATIVE                  // same as solvePnP()
)
```

cv::solvePnPRansac()与cv::solvePnP()的相同参数解释相同。RANSAC算法还由一些新参数控制。具体来说，iterationsCount参数设置RANSAC迭代的次数，而reprojectionError参数表示将某点设置为内部点的最大重投影误差[注39]参数

注39：对于PnP问题，需要有效寻找由四个点完全确定的透视变换。因此，对于RANSAC迭代，所选点的初始数为4。重投影误差的默认值对应于这些点中的每个点与其相应的重投影之间的平均距离开2次根。

minInliersCount的命名有一定的误导性，如果在运行RANSAC时，内部点数量超过minInliersCount，则该过程将被终止，并且该组将被认为是内部点组。这样做可以显著提高性能，但如果设置得太低也会出现很多问题。最后，inliers参数是一个输出，如果提供的话，将用内部点的索引值（从objectPoints到imagePoints）来填充该参数。

矫正

正如我们已经提到的，标定相机通常需要做两件事情：第一是纠正畸变的影响，第二是根据获得的图像重构三维场景。我们先来讨论纠正畸变的影响，然后在下一章中讨论更复杂的第二个任务，重构三维场景。

OpenCV为我们提供了可以直接使用的矫正算法，即通过输入原始图像和由函数cv::calibrateCamera()得到的畸变系数，生成矫正后的图像（图18-17）。我们既可以只用函数cv::undistort()使用该算法一次性完成所需的任务，也可以用一对函数cv::initUndistortRectifyMap()和cv::remap()来更有效地处理此事，这通常适用于视频或者同一相机中获取多个图像的应用中。[注40]

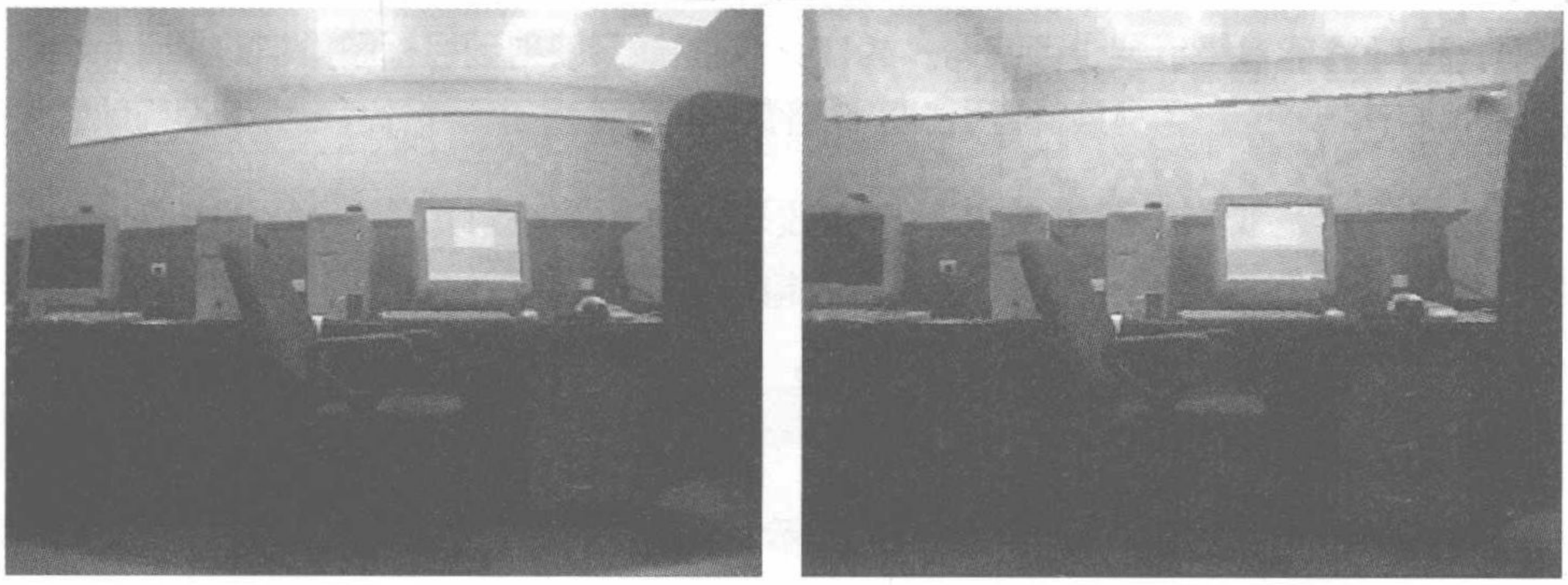


图18-17. 矫正前（左）和矫正后（右）的相机图像

矫正映射

当进行图像矫正时，我们必须指定输入图像中的每个像素在输出图像中移动到的位置，称为“矫正映射”（或有时为“畸变映射”）。这样的映射有以下几种表示。

首先介绍的是双通道浮点数表示，这种表示最为直接。在该表示中，如图18-18所示，$N \times M$图像的重映射由双通道浮点数的$N \times M$矩阵表示。对于图像中的任意给定元素（下

注40：我们需要花一定时间来明确区分矫正和校正。矫正是在数学上去掉透镜畸变，而校正是在数学上将两个（或更多）图像整齐排列。后者在下一章中很重要。

标）（i, j），该元素的值为一对数字（i^*, j^*），表示输入图像的像素（i, j）应被重新定位的位置。当然，因为（i^*, j^*）是浮点数，表示在目标图像中出现了内插。[注41]

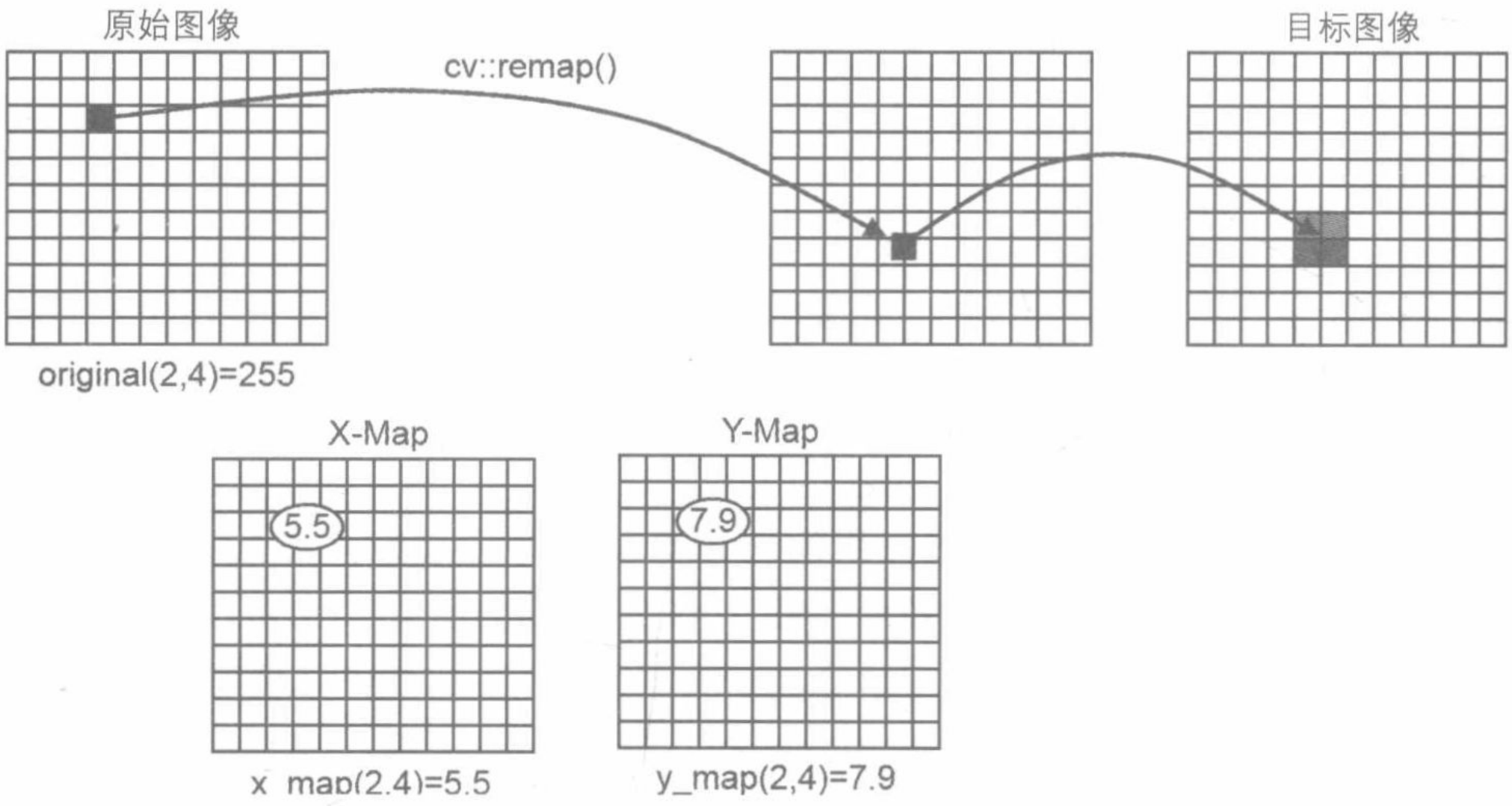


图18-18. 在浮点数-浮点数表示中，两个不同的浮点矩阵，即X-map和Y-map，将原始图像中的（i，j）像素的最终位置进行编码。（i，j）处的像素映射到目标图像中的（x_map（i，j），y_map（i，j））。因为该位置不一定是整数，所以使用插值来计算最终的图像像素强度

下一个表示方式是双矩阵浮点数表示。在该表示中，重映射由一对$N \times M$矩阵描述，每个$N \times M$矩阵都是单通道浮点矩阵。第一个矩阵包含在位置（i, j）处的值i^*，即原始矩阵中像素（i, j）重映射位置的x坐标。相似地，第二矩阵包含相同位置的j^*，即像素（i, j）重映射位置的y坐标。

最后一种表示方式为定点表示。在该表示中，映射由双通道有符号整数矩阵（即CV_16SC2类型）表示。该矩阵与双通道浮点数表示方式相同，但使用此格式的操作要快得多。在需要更高精度的情况下，重映射时内插所需的必要信息编码在第二个单通道无符号整数矩阵（即CV_16UC1）中。该矩阵中的元素（下标）是指用于插值的内部查找表（因此在此矩阵中没有“用户可维护的部分”）。

使用cv::convertMaps()在不同表示方式之间转换矫正映射

因为在矫正映射中有多个表示方式，所以人们很自然地希望在这些表示方式之间能够进行相互转换。我们用cv::convertMaps()函数便能够做到这一点。这个功能允许你提供

注41：通常情况下，是以相反的方向进行插值。意思是指给定映射，为了计算最终图像中的像素，我们通过确定原始图像中哪些像素映射到其附近，然后适当地在这些像素值之间进行插值。

我们刚才讨论的四种格式的映射，并将其转换为任何其他格式。cv::convertMaps()的原型如下：

```
void cv::convertMaps(
  cv::InputArray  map1,           // First in map: CV_16SC2/CV_32FC1/CV_32FC2
  cv::InputArray  map2,           // Second in map: CV_16UC1/CV_32FC1 or none
  cv::OutputArray dstmap1,        // First out map
  cv::OutputArray dstmap2,        // Second out map
  int             dstmap1type,    // dstmap1 type: CV_16SC2/CV_32FC1/CV_32FC2
  bool            nninterpolation = false // For conversion to fixed point types
);
```

输入map1和map2是你现有的映射，输出dstmap1和dstmap2是转换后的映射。参数dstmap1type告诉cv::convertMaps()要创建什么样的映射。从该指定的类型中推断出你想要的映射。当转换为定点类型时，最后一个参数nninterpolation表示是否还要计算插值表。可能的转换以及每次转换的dstmap1type和nninterpolation的正确设置如表18-1所示。

表18-1：在cv::convertMaps()中的类型转换

map1	map2	dstmap1type	插值	dstmap1	dstmap2
CV_32FC1	CV_32FC1	CV_16SC2	true	CV_16SC2	CV_16UC1
CV_32FC1	CV_32FC1	CV_16SC2	false	CV_16SC2	cv::noArray()
CV_32FC2	cv::noArray()	CV_16SC2	true	CV_16SC2	CV_16UC1
CV_32FC2	cv::noArray()	CV_16SC2	false	CV_16SC2	cv::noArray()
CV_16SC2	CV_16UC1	CV_32FC1	true	CV_32FC1	CV_32FC1
CV_16SC2	cv::noArray()	CV_32FC1	false	CV_32FC1	CV_32FC1
CV_16SC2	CV_16UC1	CV_32FC2	true	CV_32FC2	cv::noArray()
CV_16SC2	cv::noArray()	CV_32FC2	false	CV_32FC2	cv::noArray()

注意，因为从浮点到定点的转换自然是会失去精度的（即使使用nninterpolation），如果你再将其转换回来，不应该期望转换回一开始数值。

使用cv::initUndistortRectifyMap()计算矫正映射

既然我们已经理解了矫正映射，下一步是了解如何从相机参数中计算它。实际上，矫正映射比这一个特定的情况更为普遍，但就我们目前的目的来看，感兴趣的是知道如何使用相机标定的结果来创建完好的图像。到目前为止，我们一直在讨论单目成像的情况。事实上，矫正更重要的应用之一是准备一对图像，用于对深度图像进行立体计算。我们将在下一章回到立体的主题，但是值得记住这个应用，部分原因在于这些矫正函数中的一些参数主要用于立体视觉。

基本过程是首先计算矫正映射，然后将其应用于图像。分开操作的原因是，在大多数实际应用中，你只需要计算一次相机的矫正映射，然后随着相机输入新的图像，你将一次又一次地使用这些映射。函数cv::initUndistortRectifyMap()从相机标定信息中计算矫正映射：

```
void cv::initUndistortRectifyMap(
  cv::InputArray  cameraMatrix,          // 3-by-3 camera matrix
  cv::InputArray  distCoeffs,            // Vector of 4, 5, or 8 coefficients
  cv::InputArray  R,                     // Rectification transformation
  cv::InputArray  newCameraMatrix,       // New camera matrix (3-by-3)
  cv::Size        size,                  // Undistorted image size
  int             m1type,                // 'map1' type: 16SC2, 32FC1, or 32FC2
  cv::OutputArray map1,                  // First output map
  cv::OutputArray map2,                  // Second output map
);
```

函数cv::initUndistortRectifyMap()计算矫正映射（或一些映射）。前两个参数是相机内参矩阵和畸变系数，都与从cv::calibrateCamera()得到的形式相同。

下一个参数*R*可以使用或者设置为cv::noArray()。如果使用，它必须是一个3×3的旋转矩阵，这将在矫正前预先使用。该矩阵的功能是补偿相机相对于相机所处的全局坐标系的旋转。

与旋转矩阵类似，newCameraMatrix可用于影响图像的矫正过程。如果使用，它将在矫正之前将图像“更改”为在不同相机的不同内在参数下的图像。实际上，以这种方式改变的是相机中心，而不是焦距。在处理单目成像时通常不会使用它，但在立体图像分析时则会很重要。对于单目图像，通常只需要将此参数设置为cv::noArray()。



除非你正在处理立体图像，否则不太可能使用旋转或newCameraMatrix参数。这时，传递给cv::initUndistortRectifyMap()的矩阵将由cv::stereoRectify()计算出来。调用cv::stereoRectify()后，使用R1和P1矩阵对来自第一台相机的图像进行校正，并使用R2和P2矩阵对来自第二台相机的图像进行校正。我们将在下一章讨论立体成像时重新讨论这个话题。[注42]

参数size是用来进行输出映射的尺寸，对应于用来矫正的图像尺寸。

最后三个参数m1type，map1和map2分别指定最终的映射类型，提供该映射的存储空间。m1type的可能值为CV_32FC1或CV_16SC2，对应于map1的表示类型。当为CV_32FC1时，则

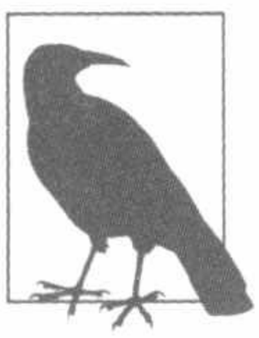
注42：非常细心的读者可能已经注意到相机矩阵是3×3矩阵，称为P1和P2的cv::stereoRectify()的返回值实际上是3×4的投影矩阵。不用担心，事实上，P1和P2的前三列包含与相机矩阵相同的信息，这是cv::initUndistortRectifyMap()在内部实际使用的。

map2也是CV_32FC1类型，这对应于我们在上一节中遇到的映射的双矩阵浮点表示。当为CV_16SC2时，map1将为定点表示（请注意，该单通道矩阵包含插值表系数）。

使用cv::remap()矫正图像

一旦计算了矫正映射，就可以使用cv::remap()将它们应用于传入的图像。在先前讨论一般图像变换时，我们遇到过cv::remap()，这里是该函数的一个重要应用示例。

如前所述，cv::remap()函数有两个对应于矫正映射的映射参数，例如由cv::initUndistortRectifyMap()计算得到的映射参数。cv::remap()接受我们讨论的任何矫正映射格式：双通道浮点型、双矩阵浮点型或定点格式（带或不带插值表索引矩阵）。



如果用两个不同的双通道浮点型表示或者不带插值表矩阵的定点类型表示来使用cv::remap()，则应该将cv::noArray()传递给map2参数。

使用cv::undistort()进行矫正

在某些情况下，只需要校正一个图像，或者对每一个图像重新计算矫正映射。在这种情况下，可以使用更加简洁的cv::undistort()，它可以有效地计算映射并将其应用于单个图像。

```
void cv::undistort(
  cv::InputArray  src,                          // Input distorted image
  cv::OutputArray dst,                          // Result corrected image
  cv::InputArray  cameraMatrix,                 // 3-by-3 camera matrix
  cv::InputArray  distCoeffs,                   // Vector of 4, 5, or 8 coeffs
  cv::InputArray  newCameraMatrix = noArray() // Optional new camera matrix
);
```

cv::undistort()的参数与cv::initUndistortRectifyMap()的对应参数相同。

使用cv::undistortPoints()进行稀疏矫正

偶尔出现的另一种情况是，不是矫正整个图像，而是从图像中收集一组点，而你只关心这些点的位置。在这种情况下，可以使用cv::undistortPoints()来计算特定的一系列点的“正确”位置：

```
void cv::undistortPoints(
  cv::InputArray  src,                          // Input array, N pts. (2-d)
```

```
    cv::OutputArray dst,                                // Result array, N pts. (2-d)
    cv::InputArray  cameraMatrix,                       // 3-by-3 camera matrix
    cv::InputArray  distCoeffs,                         // Vector of 4, 5, or 8 coeffs
    cv::InputArray  R                 = cv::noArray(), // 3-by-3 rectification mtx.
    cv::InputArray  P                 = cv::noArray()  // 3-by-3 or 3-by-4 new camera
                                                       // or new projection matrix
);
```

与cv::undistort()一样，cv::undistortPoints()的参数类似于cv::initUndistortRectifyMap()的对应参数。主要区别在于参数src和dst是二维点的向量，而不是二维矩阵。和往常一样，这些向量可以是常用的形式：$N \times 1$的cv::Vec2i对象的矩阵，$N \times 1$的float对象的矩阵，cv::Vec2f对象的STL风格的向量等。

cv::undistortPoints()的参数P对应于cv::undistortPoints()的newCameraMatrix。如前所述，这两个额外的参数主要与第19章讨论的函数在立体校正方面的使用有关。矫正相机矩阵P可以是3×3的3×4的矩阵，由cv::stereoRectify()返回的相机矩阵P1或P2（对应于左或右相机，详见第19章）的前三列或前四列组成。这些参数默认为cv::noArray()，即函数将其解释为单位矩阵。

与标定结合

最后我们总结一下。示例18-1表示实现以下任务的程序：查找用户指定尺寸的棋盘，尽可能多获取用户要求的完整图像（即，可以找到所有棋盘角点的图像）以及计算相机内在参数和畸变参数。最后，程序进入显示模式，从而让我们可以看到矫正后的相机图像。



在使用这个程序时，你需要在成功捕获图像后，大幅度地改变棋盘视图。否则，用于求解标定参数的点的矩阵可能形成一个病态（秩缺陷）矩阵，并且你最终会得到一个不好的结果或根本没有结果。

示例18-1：读取棋盘的宽度和高度，读取和收集所需的视图数量，并标定相机

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

void help( char *argv[] ) {
  ...
}

int main(int argc, char* argv[]) {

  int  n_boards = 0; // Will be set by input list
  float image_sf = 0.5f;
```

```
float delay    = 1.f;
int  board_w  = 0;
int  board_h  = 0;

if(argc < 4 || argc > 6) {
  cout << "\nERROR: Wrong number of input parameters";
  help( argv );
  return -1;
}
board_w  = atoi( argv[1] );
board_h  = atoi( argv[2] );
n_boards = atoi( argv[3] );
if( argc > 4 ) delay    = atof( argv[4] );
if( argc > 5 ) image_sf = atof( argv[5] );
int      board_n  = board_w * board_h;
cv::Size board_sz = cv::Size( board_w, board_h );

cv::VideoCapture capture(0);
if( !capture.isOpened() ) {
  cout << "\nCouldn't open the camera\n";
  help( argv );
  return -1;
}

// ALLOCATE STORAGE
vector< vector<cv::Point2f> > image_points;
vector< vector<cv::Point3f> > object_points;

// Capture corner views: loop until we've got n_boards successful
// captures (all corners on the board are found).
double last_captured_timestamp = 0;
cv::Size image_size;

while( image_points.size() < (size_t)n_boards ) {

  cv::Mat image0, image;
  capture >> image0;
  image_size = image0.size();
  cv::resize(image0, image, cv::Size(), image_sf, image_sf, cv::INTER_LINEAR );

  // Find the board
  vector<cv::Point2f> corners;
  bool found = cv::findChessboardCorners( image, board_sz, corners );

  // Draw it
  drawChessboardCorners( image, board_sz, corners, found );

// If we got a good board, add it to our data
double timestamp = (double)clock()/CLOCKS_PER_SEC;

if( found && timestamp - last_captured_timestamp > 1 ) {

  last_captured_timestamp = timestamp;
  image ^= cv::Scalar::all(255);

  cv::Mat mcorners(corners); // do not copy the data
  mcorners *= (1./image_sf); // scale the corner coordinates
```

```
      image_points.push_back(corners);
      object_points.push_back(vector<Point3f>());
      vector<cv::Point3f>& opts = object_points.back();
      opts.resize(board_n);
      for( int j=0; j<board_n; j++ ) {
        opts[j] = cv::Point3f((float)(j/board_w), (float)(j%board_w), 0.f);
      }
      cout << "Collected our " << (int)image_points.size() <<
        " of " << n_boards << " needed chessboard images\n" << endl;
    }
    cv::imshow( "Calibration", image ); //show in color if we did collect the image
      if((cv::waitKey(30) & 255) == 27)
        return -1;
    }
    // END COLLECTION WHILE LOOP.

    cv::destroyWindow( "Calibration" );
    cout << "\n\n*** CALIBRATING THE CAMERA...\n" << endl;

    // CALIBRATE THE CAMERA!
    cv::Mat intrinsic_matrix, distortion_coeffs;
    double err = cv::calibrateCamera(
      object_points,
      image_points,
      image_size,
      intrinsic_matrix,
      distortion_coeffs,
      cv::noArray(),
      cv::noArray(),
      cv::CALIB_ZERO_TANGENT_DIST | cv::CALIB_FIX_PRINCIPAL_POINT
    );

    // SAVE THE INTRINSICS AND DISTORTIONS
    cout << " *** DONE!\n\nReprojection error is " << err <<
      "\nStoring Intrinsics.xml and Distortions.xml files\n\n";
    cv::FileStorage fs( "intrinsics.xml", FileStorage::WRITE );

    fs << "image_width" << image_size.width << "image_height" << image_size.height
      <<"camera_matrix" << intrinsic_matrix << "distortion_coefficients"
      << distortion_coeffs;
    fs.release();

    // EXAMPLE OF LOADING THESE MATRICES BACK IN:
    fs.open( "intrinsics.xml",    cv::FileStorage::READ );
    cout << "\nimage width: " << (int)fs["image_width"];
    cout << "\nimage height: " << (int)fs["image_height"];

    cv::Mat intrinsic_matrix_loaded, distortion_coeffs_loaded;
    fs["camera_matrix"] >> intrinsic_matrix_loaded;
    fs["distortion_coefficients"] >> distortion_coeffs_loaded;
    cout << "\nintrinsic matrix:" << intrinsic_matrix_loaded;
    cout << "\ndistortion coefficients: " << distortion_coeffs_loaded << endl;

    // Build the undistort map which we will use for all
    // subsequent frames.
```

```
cv::Mat map1, map2;
cv::initUndistortRectifyMap(
  intrinsic_matrix_loaded,
  distortion_coeffs_loaded,
  cv::Mat(),
  intrinsic_matrix_loaded,
  image_size,
  CV_16SC2,
  map1,
  map2
);

// Just run the camera to the screen, now showing the raw and
// the undistorted image.
for(;;) {
  cv::Mat image, image0;
  capture >> image0;
  if( image0.empty() ) break;
  cv::remap(
    image0,
    image,
    map1,
    map2,
    cv::INTER_LINEAR,
    cv::BORDER_CONSTANT,
    cv::Scalar()
  );
  cv::imshow("Undistorted", image);
    if((cv::waitKey(30) & 255) == 27) break;
  }

  return 0;
}
```

小结

在本章开始，我们简要回顾了小孔相机模型并对射影几何基础进行了概述。在介绍Rodrigues变换作为旋转的替代表示之后，我们引入了透镜畸变的概念，并学习了在OpenCV中如何对它们进行建模，该模型在相机内参矩阵中进行了整合。

理解这个模型以后，我们开始学习如何使用棋盘或圆网格标定相机（还有其他标定模型和技术，参考附录B的ccalib函数组）。我们看到了OpenCV如何使用多个标定图像中找到的角点或圆圈来计算这些内在和外在参数。标定函数引入了图像单应性的话题。我们学习了标定中内在参数和外在参数之间的差异，并将这些参数与普通PnP姿态估计问题相关联。我们最后了解了如何通过计算得到的相机的内在参数来矫正图像，从而消除真实世界中镜头产生的常见畸变。

最后，我们展现了一个完整例子，即获取图像，从这些图像中提取标定数据，从该数据中计算相机的内在参数，并使用相机信息来矫正输入视频。

练习

1. 如果只有一个尺子，你会如何确定相机的焦距？ 假设相机的畸变可忽略，主点是图像的中心。
2. 根据图18-2，利用相机中心位置偏移的相似三角形推导方程得出方程$x = f_x \cdot (X/Z) + c_x$和$y - f_y \cdot (Y/Z) + c_y$。
3. 真实中心位置(c_x, c_y)的估计误差是否会影响其他参数的估计，比如焦距的估计？

   提示：参见方程$q = MQ$。
4. 分别根据以下条件绘制一个正方形的图像：

   a. 在径向畸变下；

   b. 在切向畸变下；

   c. 在两种畸变下。
5. 参考图18-19，对于射影变换，解释以下问题。

   a. 无穷远的线来自何处？

   b. 为何目标平面的平行线收敛到图像平面的一个点？

   c. 假定物体平面和图像平面相互垂直。在物体平面上，从一点开始，从图像平面开始，远离10个单位到，在图像平面上的相应移动距离是多少？
6. 图18-3显示向外凸出的“筒形畸变”效应，它在图18-12中更为明显，一些透镜是否能产生向内弯曲的效果？如何做到这一点？
7. 使用一个廉价的网络相机或者手机拍摄有径向畸变和切向畸变的同心正方形或棋盘图像。

   a. 使用`cv::calibrateCamera()`和至少15张棋盘图片来标定相机，然后利用相机标定的旋转和平移向量，使用`cv::projectPoints()`把正交于棋盘的箭头（表面法线）投影到每个棋盘图像中。

   b. 显示矫正前后的图片。
8. 如果用因子2对图像矩阵进行下采样（在x方向和y方向上，每隔一个像素，就跳过当前像素），标定的焦距和主点会发生什么？
9. 通过收集棋盘的大量图像并对其进行“良好”标定，对数值稳定性和噪声进行实验。当减少棋盘图像的数量，看看标定参数如何改变。将结果以图的形式表现出来：相机参数作为棋盘图像数量的函数。

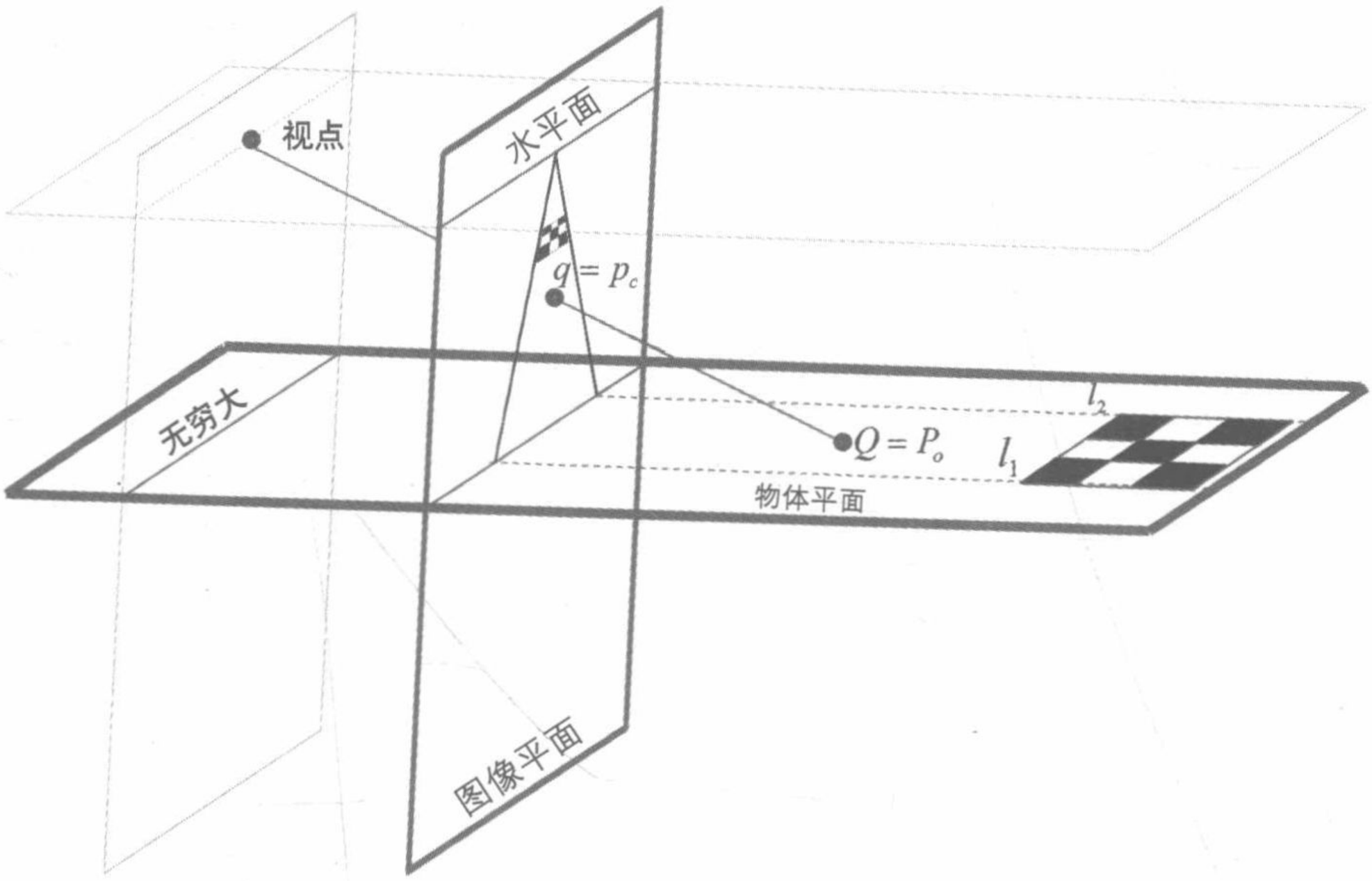


图18-19. 显示物体平面与图像平面的交点，以及表示投影中心的视点的单应图

10. 高端相机通常有可以物理矫正图像畸变的透镜系统。如果强行对该相机使用多项式矫正模型会发生什么现象？

    提示：此情况被称为过拟合。

11. 三维操纵杆。标定相机，使用视频，摇晃棋盘，并使用cv::solvePnP()作为三维操纵杆。记住，cv::solvePnP()的输出为3×1或1×3的旋转轴向量以及三维平移向量，其中旋转向量的幅度指的是逆时针的旋转角。

    a. 当你移动棋盘时，将棋盘的轴线和旋转角度及其位置（即平移）实时输出。处理棋盘不在视线中的情况。

    b. 使用cv::Rodrigues()将cv::solvePnP()的输出转换为3×3旋转矩阵和平移向量。利用它来动态模拟一个飞机被实时渲染到图像上的简单三维简笔画（火柴人造型），如同在视频相机的视场中移动棋盘一样。

第19章

投影与三维视觉

在这一章中，我们将开始三维视觉的学习。首先，研究三维到二维的投影和逆向变换（这个操作可以尽可能地反转），并且了解多摄像机立体深度感知的相关内容。为了学好本章，我们将利用第18章的一些概念。摄像机内联参数矩阵M、畸变系数、旋转矩阵R以及平移向量$\vec{T}$，特别是还会用到单应性矩阵H。[注1]

我们将从讨论投影开始，通过相机标定，逐渐深入三维世界，并回顾仿射与投影之间的变换（首次出现于第11章）；之后会通过一个实例了解如何获得地面鸟瞰图。[注2]同时还将更详细地讨论在第18章第一次出现的函数`cv::solvePnP()`。在这部分中，我们将看到该算法怎样用于搜寻图像中一个已知三维物体的三维姿态（位置和旋转角）。

在了解这些概念之后，我们将进一步学习多幅图像的三维几何。目前来说，还没有可靠的方法能够在单幅图像的情况下进行标定和提取三维信息。利用多幅图像进行三维场景重建最典型的实例是立体视觉。在立体视觉中，通过把从不同摄像机同时获取的两个（或更多）图像中的特征，与其他图像中的相关特征进行匹配，并分析特征之间的偏差，产生深度信息。另一个实例是通过运动获取目标结构信息。在这种情况中，我们可能只有一个摄像机，但是可以从不同时间和不同地点获取多幅图像。对于前者，侧重于将视差效应（三角剖分）作为一种计算距离的方法。对于后者，通过计算基础矩阵（关联两个不同场景）作为场景理解的数据源。

注1：同时，就像在第18章提到的，也像在附录B中描述的，*opencv_contrib*包含全景相机和多个相机（在`ccalib`中）的进一步的标定算法，不同类型的标定模式（在`aruro`和`ccalib`中），以及最后的色彩平衡和去噪算法（`xphoto`）。因为这是实验代码，这里不会详细介绍，但是许多这些工具对改善立体标定和视觉十分有用。

注2：这是机器人以及许多其他视觉应用中的一个常见问题。

由于所有这些问题都在不同程度上依赖于将三维世界投影到二维图像或者摄像机的能力，所以我们将从第18章已学内容开始，逐步建立知识体系。

投影

完成摄像机标定后（详见第18章），可以将物理世界中的点准确地投影图像中的点。这意味着，一旦给定一个摄像机附近的三维物理坐标系的位置，就可以计算出在外部三维点在成像设备中（即在像素坐标系）出现的位置。这个转换在OpenCV例程cv::projectPoints()中进行了实现：

```
void cv::projectPoints(
  cv::InputArray  objectPoints, // 3xN/Nx3 Nc=1, 1xN/Nx1 Nc=3,
                                // or vector<Point3f>
  cv::InputArray  rvec,         // Rotation *vector*
                                // (see cv::Rodrigues())
  cv::InputArray  tvec,         // Translation vector
  cv::InputArray  cameraMatrix, // 3x3 Camera intrinsics matrix
  cv::InputArray  distCoeffs,   // 4, 5, or 8 elements vector,
                                // or cv::noArray()
  cv::OutputArray  imagePoints, // 2xN/Nx2 Nc=1, 1xN/Nx1 Nc=2,
                                // or vector<Point2f>
  cv::OutputArray  jacobian    = cv::noArray(), // Optional,
                                                // 2N x (10+nDistCoeff)
  double           aspectRatio = 0              // If nonzero, fix
                                                // fx/fy at this value
);
```

乍一看，参数数量可能有一些吓人，但实际使用中这是一个简单的函数。函数cv::projectPoints()被设计用来适应（非常普通的）环境，即欲投影的点位于一些刚体上。在这种情况下，这些点的位置可以在以物体自身为坐标原点的坐标系上表示出来，而不是在摄像机坐标系中的位置；之后通过加入旋转和平移来定义物体坐标系和摄像机坐标系之间的关系。实际上，cv::projectPoints()函数在cv::calibrateCamera()中被内部调用，这也是cv::calibrateCamera()函数组织内部运行的方式。所有的可选参数都可以被函数cv::calibrateCamera()调用，但是有经验的用户可以根据自己的需求使用这些参数。

第一个参数objectPoints，是需要投影的点序列，它可以是任何常见的格式：$N\times 3$，$3\times N$，$N\times 1$或$1\times N$的cv::Vec3f对象数组，或者仅是一个单纯的旧STL类型向量，其中包含点的位置。将这些代入物体自身坐标系，之后可以得到旋转矢量[注3]和平移矢量，将

注3：“旋转矢量”是弧度表达，已经在第18章开篇做过介绍。

物体坐标系和摄像机坐标系联系起来。如果情况特殊，直接在摄像机坐标系中计算更为简单，那么可以只设置该坐标系中的objectpoints，并把旋转矢量和平移矢量内的元素设置为0。[注4]

摄像机矩阵和畸变系数都只是摄像机的内部信息，并且畸变系数来自于第18章已经讨论过的函数cv::calibrateCamera()。imagePoints的参数是一个由二维点组成的数组（可以为任何常规形式），用于存储计算结果。

之后是可选参数jacobian，如果选择改参数，它将被赋予每一个点所在位置旋转矢量和平移矢量的分量，摄像机矩阵的元素，以及畸变系数相对应的偏导数的值。因此，jacobian将是一个（$2 \cdot N_p$）（$10+N_d$）的数组——其中，N_p是点的数量，N_d是畸变系数的数量。图19-1列出了jacobian各项的具体内容。如果不需要计算jacobian（实际上大多数情况下都不需要计算），可以将其设置为函数cv::noArray()，就不会计算。

最后一个参数aspectRatio，同样是可选的；只有当函数cv::calibrateCamera()或者函数cv::stereoCalibrate()宽高比固定时，这个参数用于求导。如果这个参数不是0.0，就调整jacobian中的导数。

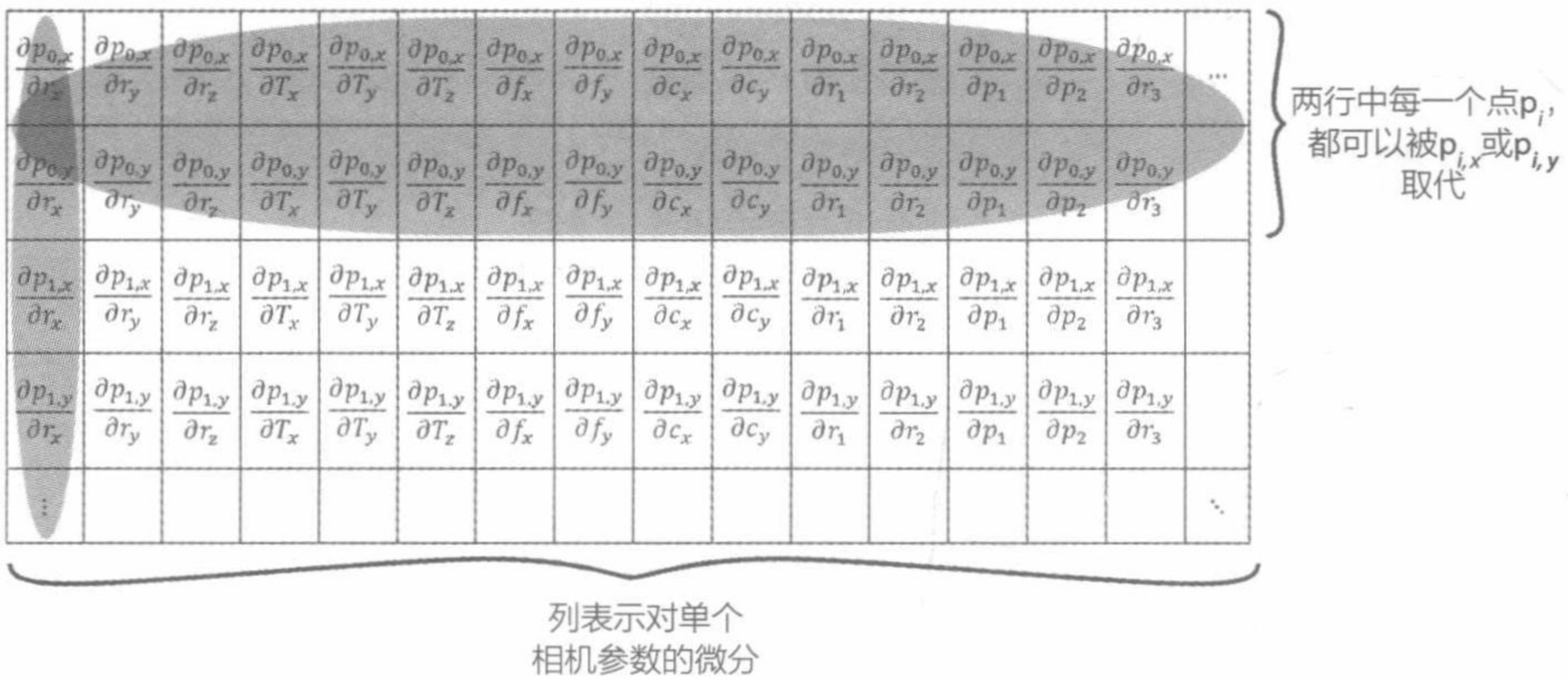


图19-1. jacobian数组的条目

仿射变换与透视变换

在OpenCV例程中经常出现的两类变换——编写应用时往往同时出现——仿射变换和透视变化。这些概念首次出现于第11章。回顾之前所学内容，如OpenCV中所实现的，这些例程对点列表或整个图像进行操作，并且将图像中的一个位置点映射到不同位置，并

注4：　旋转矢量表示旋转角，所以如果将所有0设置为0，表示它为零幅值，“没有旋转”。

沿路径进行子像素插值。也可以这样想，仿射变换可以通过一个矩形产生任何平行四边形；透视变换更加普遍，可以通过一个矩形产生任何梯形。

透视变换与投影是紧密结合的。透视投影将三维物理世界的点沿着一组投影线映射成二维图像平面上的点，这些投影线相交于一点，即投影中心。透视投影是一种特殊的单应[注5]，它把两种不同的图像联系起来，是同一三维物体在两个不同投影平面上的替代投影。值得注意的是，对于复杂一点儿的构造（例如平面与三维相交的物体），同样可以说单应把两个不同的投影中心联系起来。

尽管这些和透视投影相关的函数已经在第11章详细讨论过，但为了方便，还是将这些函数总结如下。

表19-1：仿射与透视变换函数

函数	用法
cv::transform()	仿射变换一系列点
cv::warpAffine()	仿射变换整幅图
cv::getAffineTransform()	从点计算仿射矩阵
cv::getRotationMatrix2D()	计算仿射矩阵，获得旋转
cv::perspectiveTransform()	透视变换一系列点
cv::warpPerspective()	透视变换整幅图
cv::getPerspectiveTransform()	填充透视变换矩阵的参数

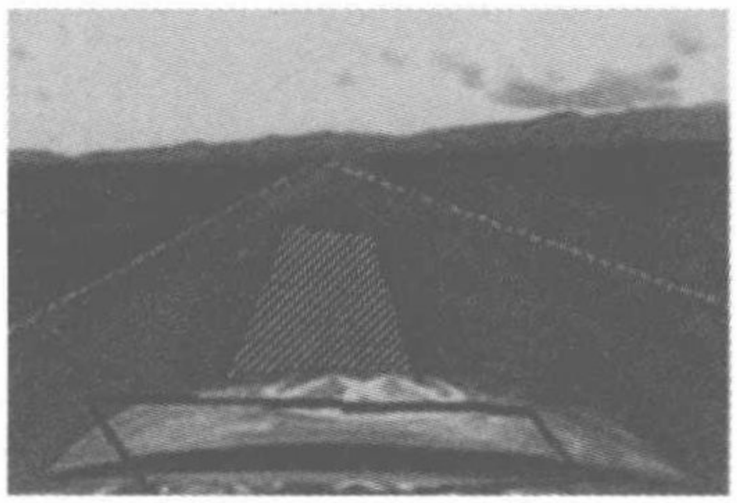
鸟瞰图变换实例

用于目标规划的一项机器人常规导航任务，是将机器人的场景视图转换为自上而下的“鸟瞰”视图。在图19-2中，机器人的场景视图被转换为鸟瞰视图，以便在该平面之上进行规划和导航，也可能再融合一些由声呐、激光测距仪或相似的在运动平面内操作的传感器获得的世界表示信息。通过已学知识，我们可以更详细地学习一些代码，以便通过摄像机标定来对这一视图进行计算。

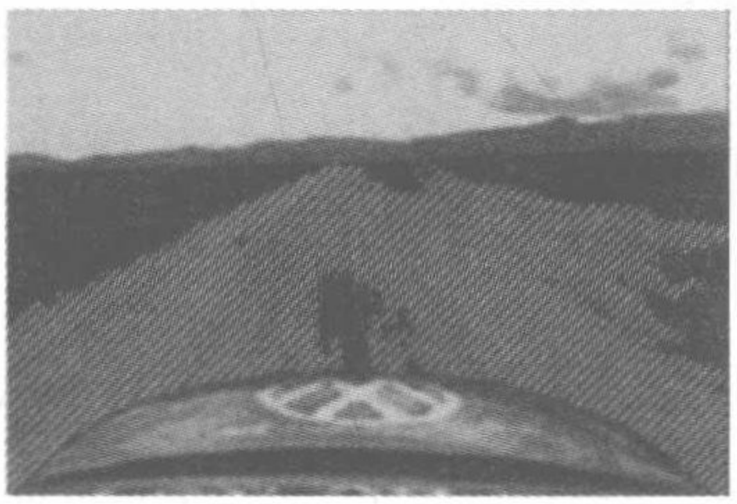
为了得到鸟瞰图，[注6]我们将用到标定中获得的摄像机矩阵和畸变参数。因为方法是多种多样的，我们将从磁盘数据文件中读取这些数据。在本例中，我们把棋盘放在地面上，用它来获取微型机器车的地平面图像；之后重新将图像映射成鸟瞰图。算法流程如下。

注5：在第18章中，这种特殊的单应性被称为“平面单应性”。

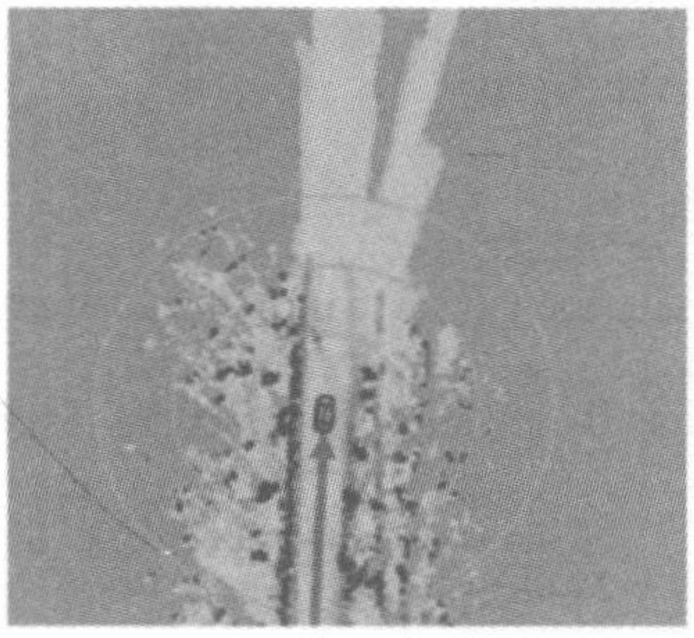
注6：鸟瞰图技术同样适用于将任何平面（例如墙面或天花板）的透视视图转到前向平行视图。



(A) 路面上用于训练的区域



(B) 泛化



(C) 路面鸟瞰图

图19-2. 鸟瞰图：安装在机器人汽车上的摄像机，对道路场景进行扫描，激光测距仪识别汽车前方的“道路”区域，并用框线(A)标记；视觉算法将平坦的疑似道路的区域(B)分割出来；分割出来的道路区域被变换为鸟瞰图，同时融入鸟瞰激光地图(C)

1. 从摄像机中读取内联模型和畸变模型。
2. 在地平面上找到已知物体（在本例中为棋盘）。以亚像素精度获得至少四个点。
3. 将找到各点带入函数cv::getPerspectiveTransform()（详见第11章），计算地平面图的单应性矩阵H。
4. 利用函数cv::warpPerspective()（详见第11章）和标志cv::WARP_INVERSE_MAP|cv::INTER_LINEAR获得地平面的正面平行（鸟瞰）图。

示例19-1：展示鸟瞰图的全部代码

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;

void help( char *argv[] ){
  ...
}

// args: [board_w] [board_h] [intrinsics.xml] [checker_image]
//
int main( int argc, char* argv[] ) {

  if( argc != 5 ) {
    cout << "\nERROR: too few parameters\n";
    help( argv );
```

```
  return -1;
}

// Input Parameters:
//
int            board_w = atoi(argv[1]);
int            board_h = atoi(argv[2]);
int            board_n = board_w * board_h;
cv::Size       board_sz( board_w, board_h );

cv::FileStorage fs( argv[3], cv::FileStorage::READ );
cv::Mat         intrinsic, distortion;

fs["camera_matrix"] >> intrinsic;
fs["distortion_coefficients"] >> distortion;
if( !fs.isOpened() || intrinsic.empty() || distortion.empty() )
{
  cout << "Error: Couldn't load intrinsic parameters from "
       << argv[3] << endl;
  return -1;
}
fs.release();

cv::Mat gray_image, image, image0 = cv::imread(argv[4], 1);
if( image0.empty() )
{
  cout << "Error: Couldn't load image " << argv[4] << endl;
  return -1;
}

// UNDISTORT OUR IMAGE
//
cv::undistort( image0, image, intrinsic, distortion, intrinsic );
cv::cvtColor( image, gray_image, cv::BGR2GRAY );

// GET THE CHECKERBOARD ON THE PLANE
//
vector<cv::Point2f> corners;
bool found = cv::findChessboardCorners( // True if found
  image,                                // Input image
  board_sz,                             // Pattern size
  corners,                              // Results
  cv::CALIB_CB_ADAPTIVE_THRESH | cv::CALIB_CB_FILTER_QUADS
);
if( !found ) {
  cout << "Couldn't acquire checkerboard on " << argv[4]
    <<", only found " << corners.size() << " of " << board_n
    << " corners\n";
  return -1;
}

// Get Subpixel accuracy on those corners
//
cv::cornerSubPix(
  gray_image,           // Input image
  corners,              // Initial guesses, also output
  cv::Size(11,11),      // Search window size
```

```
    cv::Size(-1,-1),     // Zero zone (in this case, don't use)
    cv::TermCriteria(
      cv::TermCriteria::EPS | cv::TermCriteria::COUNT,
      30, 0.1
    )
);

// GET THE IMAGE AND OBJECT POINTS:
// Object points are at (r,c):
// (0,0), (board_w-1,0), (0,board_h-1), (board_w-1,board_h-1)
// That means corners are at: corners[r*board_w + c]
//
cv::Point2f objPts[4], imgPts[4];
objPts[0].x = 0;         objPts[0].y = 0;
objPts[1].x = board_w-1; objPts[1].y = 0;
objPts[2].x = 0;         objPts[2].y = board_h-1;
objPts[3].x = board_w-1; objPts[3].y = board_h-1;
imgPts[0]   = corners[0];
imgPts[1]   = corners[board_w-1];
imgPts[2]   = corners[(board_h-1)*board_w];
imgPts[3]   = corners[(board_h-1)*board_w + board_w-1];

// DRAW THE POINTS in order: B,G,R,YELLOW
//
cv::circle( image, imgPts[0], 9, cv::Scalar( 255,   0,   0), 3);
cv::circle( image, imgPts[1], 9, cv::Scalar(   0, 255,   0), 3);
cv::circle( image, imgPts[2], 9, cv::Scalar(   0,   0, 255), 3);
cv::circle( image, imgPts[3], 9, cv::Scalar(   0, 255, 255), 3);

// DRAW THE FOUND CHECKERBOARD
//
cv::drawChessboardCorners( image, board_sz, corners, found );
cv::imshow( "Checkers", image );

// FIND THE HOMOGRAPHY
//
cv::Mat H = cv::getPerspectiveTransform( objPts, imgPts );

// LET THE USER ADJUST THE Z HEIGHT OF THE VIEW
//
double Z = 25;
cv::Mat birds_image;
for(;;) {                                   // escape key stops
  H.at<double>(2, 2) = Z;
  // USE HOMOGRAPHY TO REMAP THE VIEW
  //
  cv::warpPerspective(
    image,                                  // Source image
    birds_image,                            // Output image
    H,                                      // Transformation matrix
    image.size(),                           // Size for output image
    cv::WARP_INVERSE_MAP | cv::INTER_LINEAR,
    cv::BORDER_CONSTANT,
    cv::Scalar::all(0)                      ¡// Fill border with black
  );
  cv::imshow("Birds_Eye", birds_image);
  int key = cv::waitKey() & 255;
```

```
  if(key == 'u') Z += 0.5;
  if(key == 'd') Z -= 0.5;
  if(key == 27) break;
}

// SHOW ROTATION AND TRANSLATION VECTORS
//
vector<cv::Point2f> image_points;
vector<cv::Point3f> object_points;
for( int i=0; i<4; ++i ){
  image_points.push_back( imgPts[i] );
  object_points.push_back(
    cv::Point3f( objPts[i].x, objPts[i].y, 0)
  );
}

cv::Mat rvec, tvec, rmat;
cv::solvePnP(
  object_points,       // 3-d points in object coordinate
  image_points,        // 2-d points in image coordinates
  intrinsic,           // Our camera matrix
  cv::Mat(),           // Since we corrected distortion in the
                       // beginning,now we have zero distortion
                       // coefficients
  rvec,                // Output rotation *vector*.
  tvec                 // Output translation vector.
);
cv::Rodrigues( rvec, rmat );

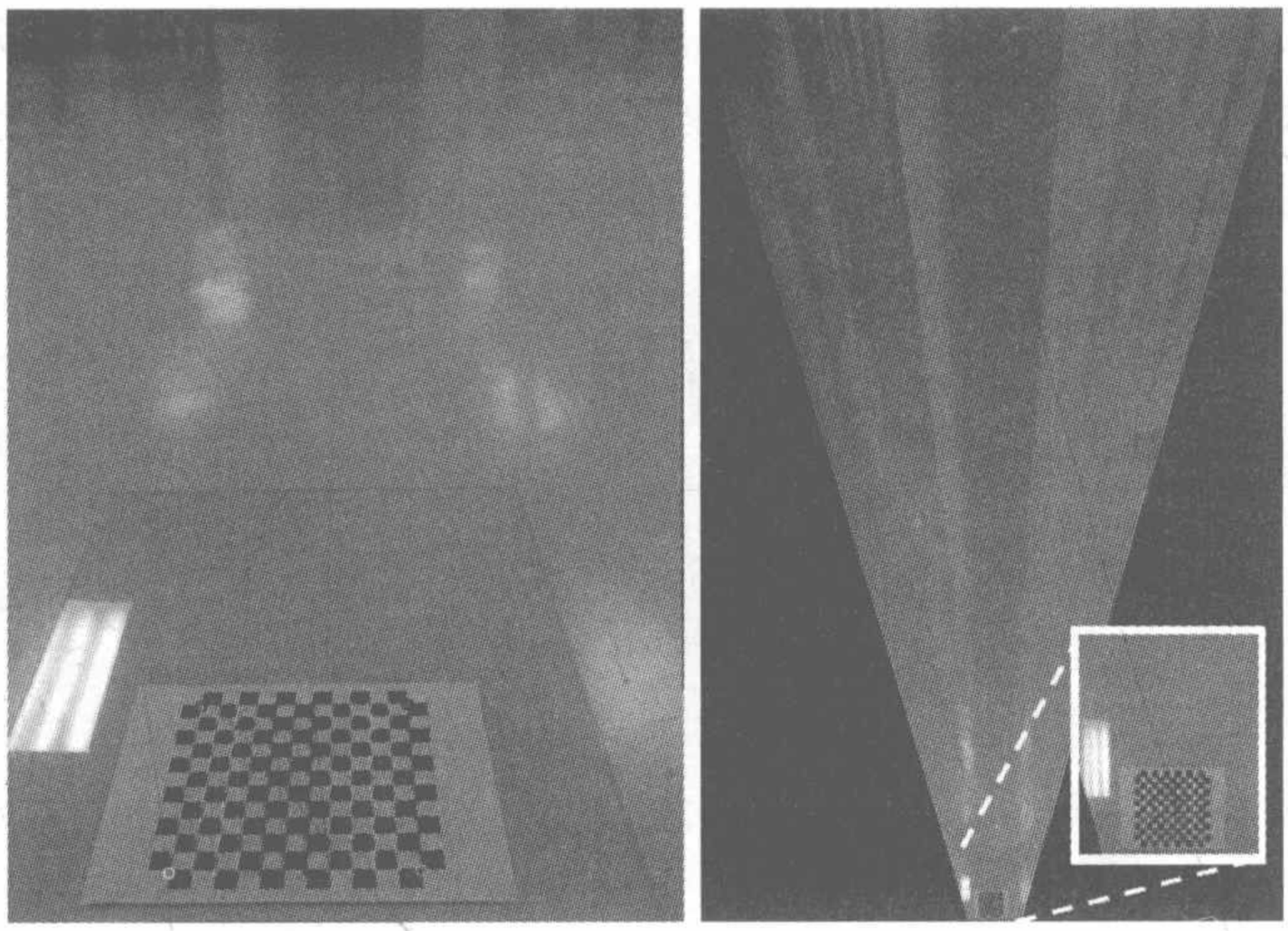
  // PRINT AND EXIT
  cout << "rotation matrix: "            << rmat    << endl;
  cout << "translation vector: "         << tvec    << endl;
  cout << "homography matrix: "          << H       << endl;
  cout << "inverted homography matrix: " << H.inv() << endl;

  return 1;
}
```

一旦获得希望得到的单应性矩阵和高度参数集合，就可以删除棋盘并移除微型小车，制作一个道路的鸟瞰图视频。这将留给读者作为一个练习。图19-3展示了鸟瞰图的输入（左）和输出（右）。

三维姿态估计

三维物体的姿态估计问题可以用单摄像机和多摄像机解决。在多摄像机的情况下，我们将用从各个摄像机单独观测到的内容之间的关系来确定物体的具体位置（例如，利用三角测量）。这种技术的优点是，它可以在物体未知甚至整个场景未知的情况下工作。但缺点是该方法需要多个摄像机。然而，利用单摄像机计算已知物体的姿态是可能的。我们将首先考虑这种情况。除了它本身是一个具有实用性的技术，理解单摄像机姿态估计问题也可以加深对多摄像机问题的了解，这在接下来的学习中将进行讨论。



一个平坦表面的视图　　鸟瞰图

图19-3. 鸟瞰图代码的输入图像（左图）及输出图像（右图）

单摄像机姿态估计

为了理解如何解决单摄像机的姿态估计问题，可以参考图19-4。如果已经在对象中确定了一些关键点（详见第16章），并且这些关键点在对象的坐标系中的位置已知（图19-4A），那么这个对象为“已知”。如果这些相同的对象呈现新的姿态，同样可以查找相同的关键点（图19-4B）。如果想求对象姿态与摄像机之间的关系，一个基本的要求是，所找到的每一个关键点必须位于从摄像机成像仪上的像素位置发出且通过摄像机光圈的特定射线上。

当然，我们不可能知道摄像机与每一个特定点之间的距离，但是在给予足够的约束时，一个刚体将唯一满足所有这些约束（图19-4C）。[注7]

利用函数cv::solvePnP()计算已知物体姿态

在之前的学习中，我们已经接触过能够解决物体姿态计算问题的函数cv::solvePnP()或者紧密相关的cv::solvePnPRansac()。[注8]在第18章中，我们介绍了这个函数，主要用于

注7：实际上，对于具有某种内在对称性的对象来说，比这更复杂一些。如果是这种情况，将可能有多种能满足约束的解决方案。基于对象的对称性，这些解决方法可以是离散的，也可以是连续的。

注8：在函数库的C语言版本中，与原来D的POSIT函数[DeMenthon92]相比，cv::solvePnP()是解决刚体姿态问题更为普遍的方法。POSIT仍然存在于OpenCV中，你可以在OpenCV在线文档中通过搜索“POSIT tutorial”找到教程。

解决棋盘或其他校准设备的姿态问题。然而实际上，这个函数可以被用来解决一般的*N*点透视问题（Perspective N-Points，PNP）。为了理解问题的一般性，请参考图19-4。

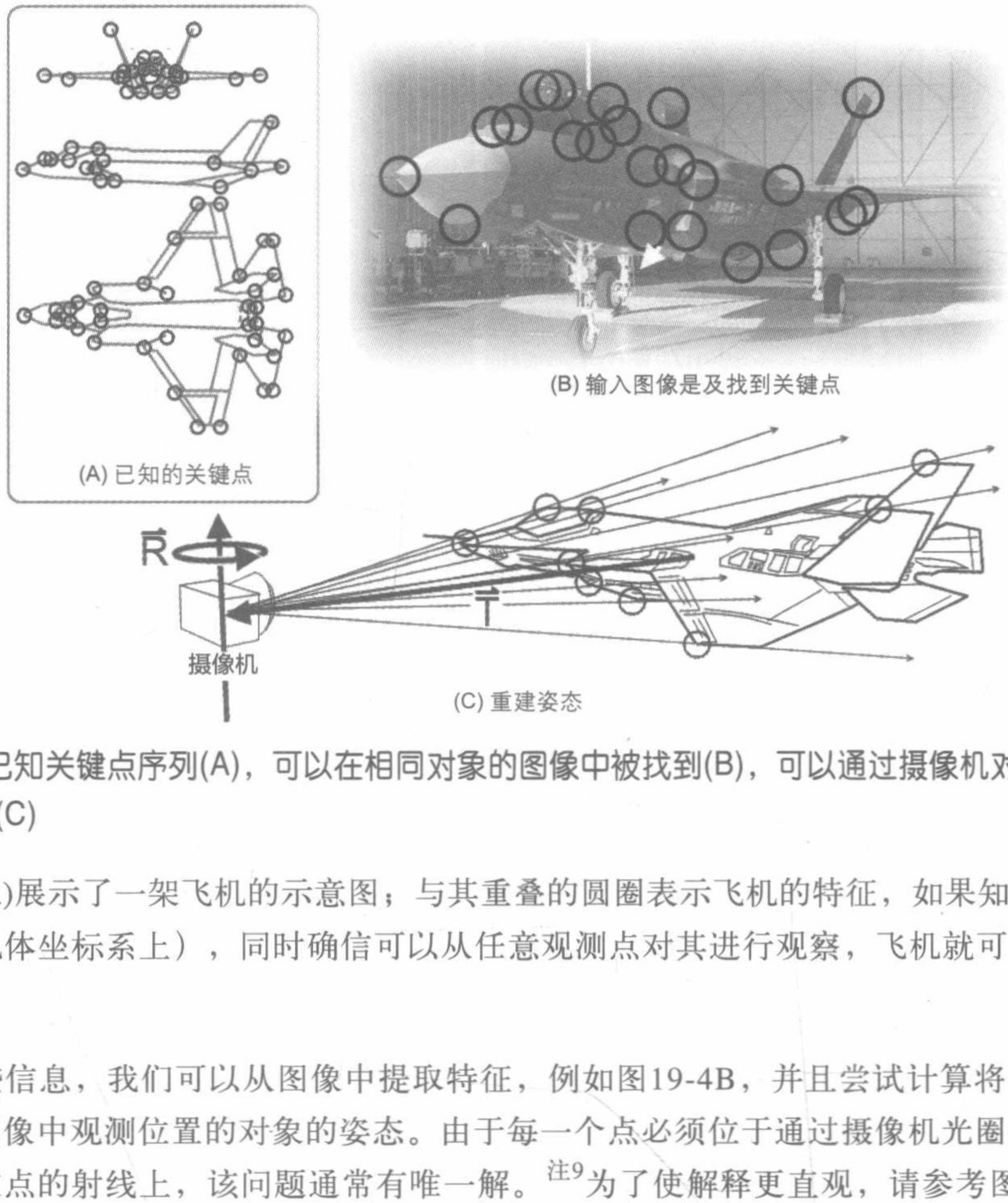


图19-4. 已知关键点序列(A)，可以在相同对象的图像中被找到(B)，可以通过摄像机对目标姿态进行重构(C)

图19-4(A)展示了一架飞机的示意图；与其重叠的圆圈表示飞机的特征，如果知道确切的位置（机体坐标系上），同时确信可以从任意观测点对其进行观察，飞机就可以得以成功辨识。

利用这些信息，我们可以从图像中提取特征，例如图19-4B，并且尝试计算将已知各点映射到图像中观测位置的对象的姿态。由于每一个点必须位于通过摄像机光圈并落在成像仪特定点的射线上，该问题通常有唯一解。[注9]为了使解释更直观，请参考图19-4C，所发现的特征被各自限制在一条特定的射线上。虽然关键点在射线上的具体位置无法知道，但可以确定只有一种方式放置对象，使得所有的射线约束被同时满足。示例19-1展示了函数`cv::solvePnP()`的使用方法。



为了辨识对象并计算其姿态，并不需要将对象全部了解，或者识别出大部分特征。同时也不需要将一个特征与一个特定的物理位置相联系。实际上，关键点检测器仅从一个小角度识别特征，因此在一个位置用多描述符捕获从不同位置感知特征的方式，是行之有效的。

注9：　如果对象在一定程度上是对称的，就有多种解决方法对应于对象的可能对称排列。

请注意，PNP问题并不总是只有唯一解。在两种重要的情况下，PNP不能提供可靠的结果。第一种情况是没有足够的关键点。理论上，这个问题可以用三次匹配解决。然而实际上，由于使用了很少的点，并且由于使用任何方法的准确性（例如，关键点匹配）产生的位置中都有自然噪声，其姿态将被遮挡很大一部分。一个很好的经验法则是，最好进行十几次甚至更多次的匹配。第二种情况是对象较远。在这种情况下，用来限制特征位置的射线将变得近似平行。射线之间的差保证物体的独特尺度，或者等价于物体距离大小的唯一解。[注10]

这种估计物体姿态（和距离）的单目方法与人眼观察远距离物体十分类似。这就是为什么在没有其他背景条件时，无法测量一个实际大小未知（特征位置位于其本身的坐标系中）的物体的距离。这也是“强迫透视”错觉的基础，例如为什么有些建筑为了使其看起来距离地面更高会在较高楼层上安装逐渐减小的窗户。在下一节中，我们将讨论立体成像技术，利用两个或更多摄像机消除这些最终不确定性，其结果也会让我们能同时推断出一个不同的[注11]对象的结构和姿态。

立体成像

现在我们开始学习立体成像。[注12]我们都对双眼的立体成像能力十分熟悉。但是在计算机系统中，我们可以模仿这种能力到什么程度呢？计算机通过寻找两个成像仪上的对应点来完成这项任务。通过这种对应和摄像机之间已知的基线距离，可以计算得到点的三维位置。尽管对应点的搜寻计算成本较高，我们可以利用系统的几何知识，尽可能限制搜寻范围。实际上，使用双摄像机的立体成像具体包括以下四个步骤。

1. 使用数学方法消除径向和切向透镜畸变，这称为“去畸变”，已在第18章做过详细说明。这一步的输出是去畸变图像。
2. 调整摄像机之间的角度和距离，这一过程称为“标定”。这一步的输出是整帧行对齐的图像（后者意味着两个图像平面是共面的，并且两个成像仪上的对应图像行实际上相对于彼此共线）。

注10：作为一个实用的提示，在许多情况下可以通过考虑可以看到什么关键点来简单粗略的估计远距离对象的方向。这不使用cv::solvePnP()，但是这在实际系统中是一个有用的技术，在通过cv::solvePnP()得到一个完整解决方法不可行时，经常可以从精度较差的反馈中得到益处。

注11：在本书中，同时在机器学习和计算机视觉中，“新”意味着这种情况之前在系统中从来没有遇到过，同时系统没有先验知识。

注12：这里我们仅给出高层次的理解。对于更多细节，我们推荐阅读以下文献：Trucco and Verri [Trucco98], Hartley and Zisserman [Hartley06], Forsyth and Ponce [Forsyth03]以及Shapiro and Stockman [Shapiro02]。这些书的立体标定章节讲给你本章中处理原始文献引用的背景。

3. 在左右摄像机视图[注13]中找到相同特征，这一过程称为“匹配”。这一步的输出是视差图，视差是指在左右摄像机观察得到的相同特征在x坐标上的差值：$x_l - x_r$。
4. 如果摄像机的几何排列已知，那么可以通过三角测量将视差图转化为距离。这一步称为重投影，输出为一个等深度图。

我们从最后一步开始，然后分析前三个步骤。

三角测量

首先考虑一个“理想”立体装置，如图19-5所示。在这种情况下，假设我们有一个完美的无畸变、对准、已测量好的系统：两个摄像机的图像平面彼此完全共面，具有完全平行的光轴（光轴是从投影中心O发出，通过主点c的一条射线，也称为“主光线”），它们是已知的距离，并且具有相等的焦距：$f_l = f_r$。同时假设主点c_x^{left}和c_x^{right}已经被校准为在其相应的左右图像中有相同的像素坐标。[注14]

此外，假设成像仪是完全行排列的，使得一个摄像机的每一个像素行与另一个摄像机中的对应行精确的对齐[注15]（我们称之为“摄像机前向平行排列”）。同时假设我们可以在物理世界中找到一点$\vec{P}$，在左右图像上的成像点为$\vec{p}_l$和$\vec{p}_r$，对应的横坐标为$\vec{x}_l$和$\vec{x}_r$。

在这种简化的情况下，我们可以发现深度与视差成反比，其中视差被简化定义为。这种情况如图19-5所示，通过相似三角形容易求出深度Z。参考示意图，我们得到下式：[注16]

$$\frac{T - (x_l - x_r)}{Z - f} = \frac{T}{Z} \Rightarrow Z = \frac{f \cdot T}{x_l - x_r}$$

注13：每次我们提到左右摄像机，同样可以使用垂直的、方向为上下的摄像机，这里视差位于y轴而非x轴。

注14：同样，不要将这些主点和图像中心搞混。主点定义为主光线和像平面的相交位置。这个交点取决于镜头的光轴，如我们在第18章所看到的，像平面实际上永远难以严格与镜头对准，所以成像仪的中心不可能严格与主点对准，除此外也许存在于“理想”立体实验台。

注15：这有许多假设，但是现在我们只看基础知识。记住，标定的过程（我们将很快返回这里）是当这些假设不是真实的时，我们如何在数学上完成事情。同样，在下一句我们将暂时“假定”对应问题。

注16：这个公式基于相交于无穷远处的主光线，如你将在后面“立体标定”一节看到的，我们相对于主点c_x^{left}和c_x^{right}导出立体标定。在我们的推导中，如果主光线相交于无穷远处，那么主点拥有相同的坐标，因此深度公式保持原状。但是如果主光线在一个有限距离内相交，那么主点将不相等并且因此深度等式将变为$Z = \frac{fT_x}{x - (c_x^{left} - c_x^{right})}$。

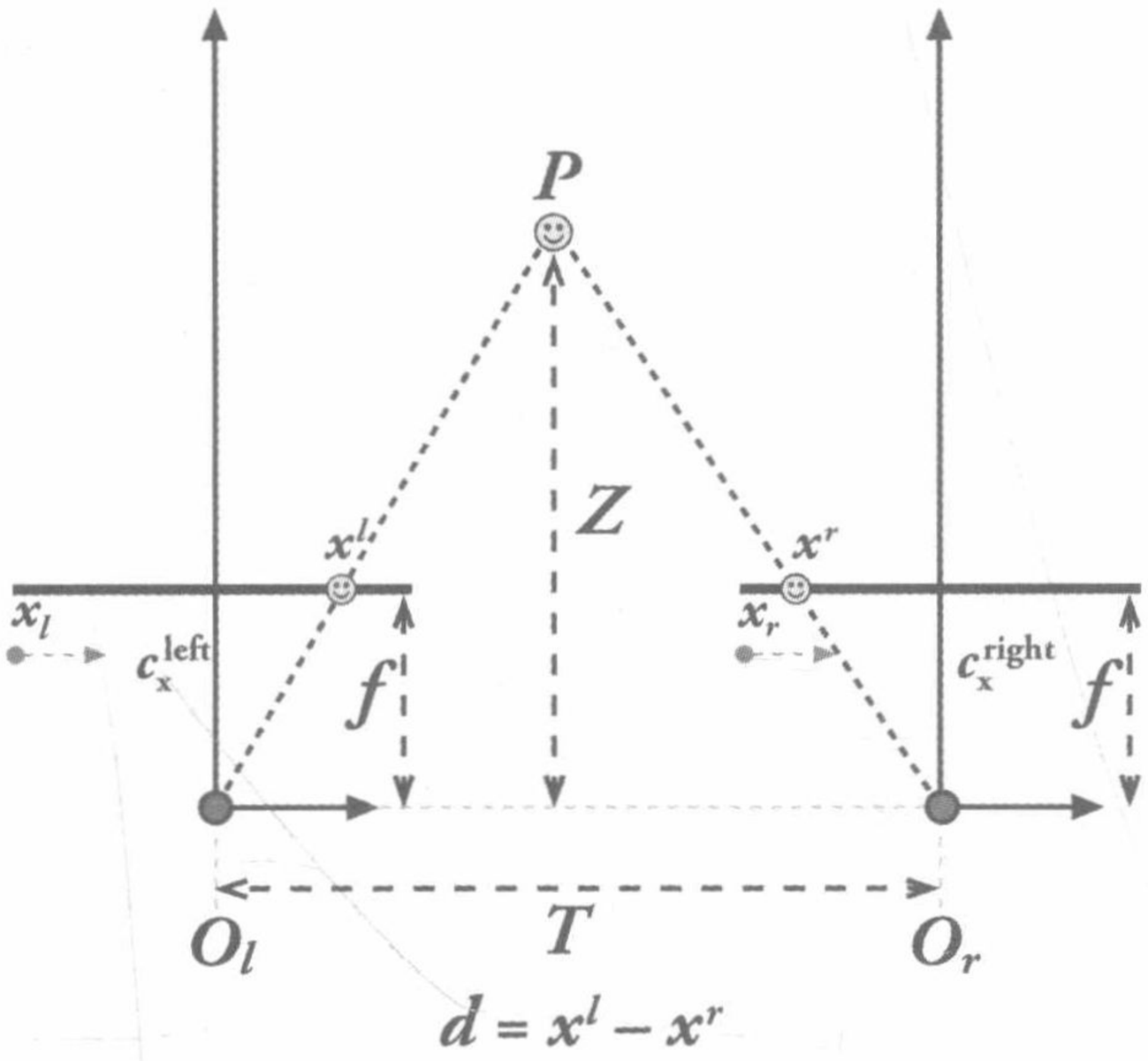


图19-5. 通过完美无畸变的标定试验台和已知的对应关系，可以通过相似三角形找到深度Z；成像仪的主光线从投影中心O_l、O_r开始，延伸通过左右图像平面的主点c_l和c_r

由于深度与视差成反比，显然，二者之间存在非线性关系。当视差趋近于零时，微小的视差变化会造成很大的深度变化。当视差较大时，微小的视差不会造成深度有太大改变。可以得出结论，立体视觉系统只有在物体与摄像机距离较近时具才有较高的深度分辨率，如图19-6所示。

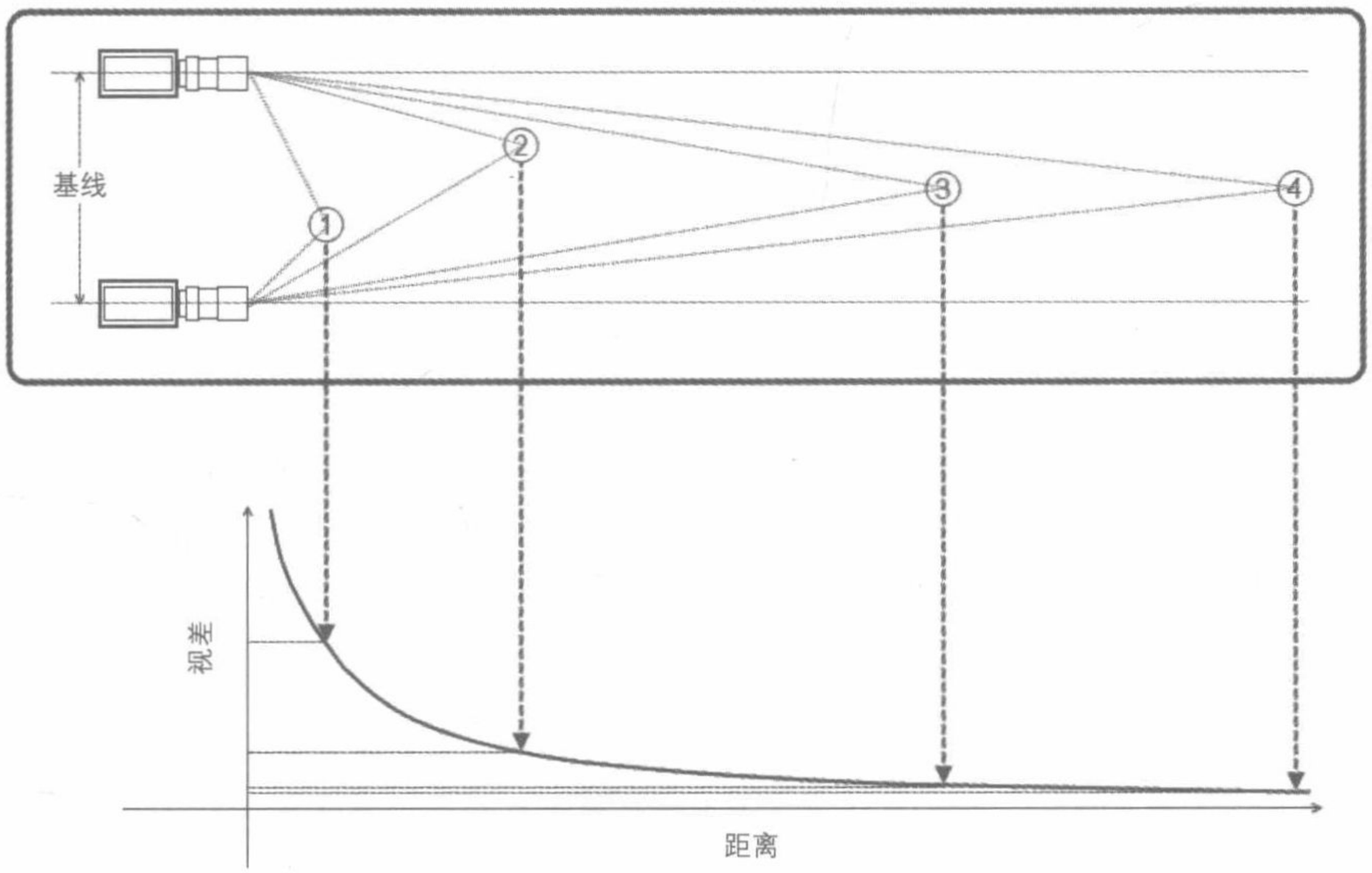


图19-6. 深度与视差成反比，所以分辨率较高的深度测量被限制于摄像机附近的物体

图19-7为用于立体视觉中OpenCV的二维和三维坐标系。注意，这是一个右手坐标系：如果将右手食指指向x轴方向，并将右手中指弯曲到y轴方向，那么大拇指将指向主光线的方向。左右成像仪的像素原点在图像的左上角，像素分别由(x_l, y_l)和(x_r, y_r)表示。投影中心位于O_l和O_r，主光线与图像平面相交于主点（并非中心）(c_x, c_y)。经过数学标定，摄像机是行对齐的（共面且水平对准），相距为T，具有相同的焦距f。这种安排使距离相对容易解决。

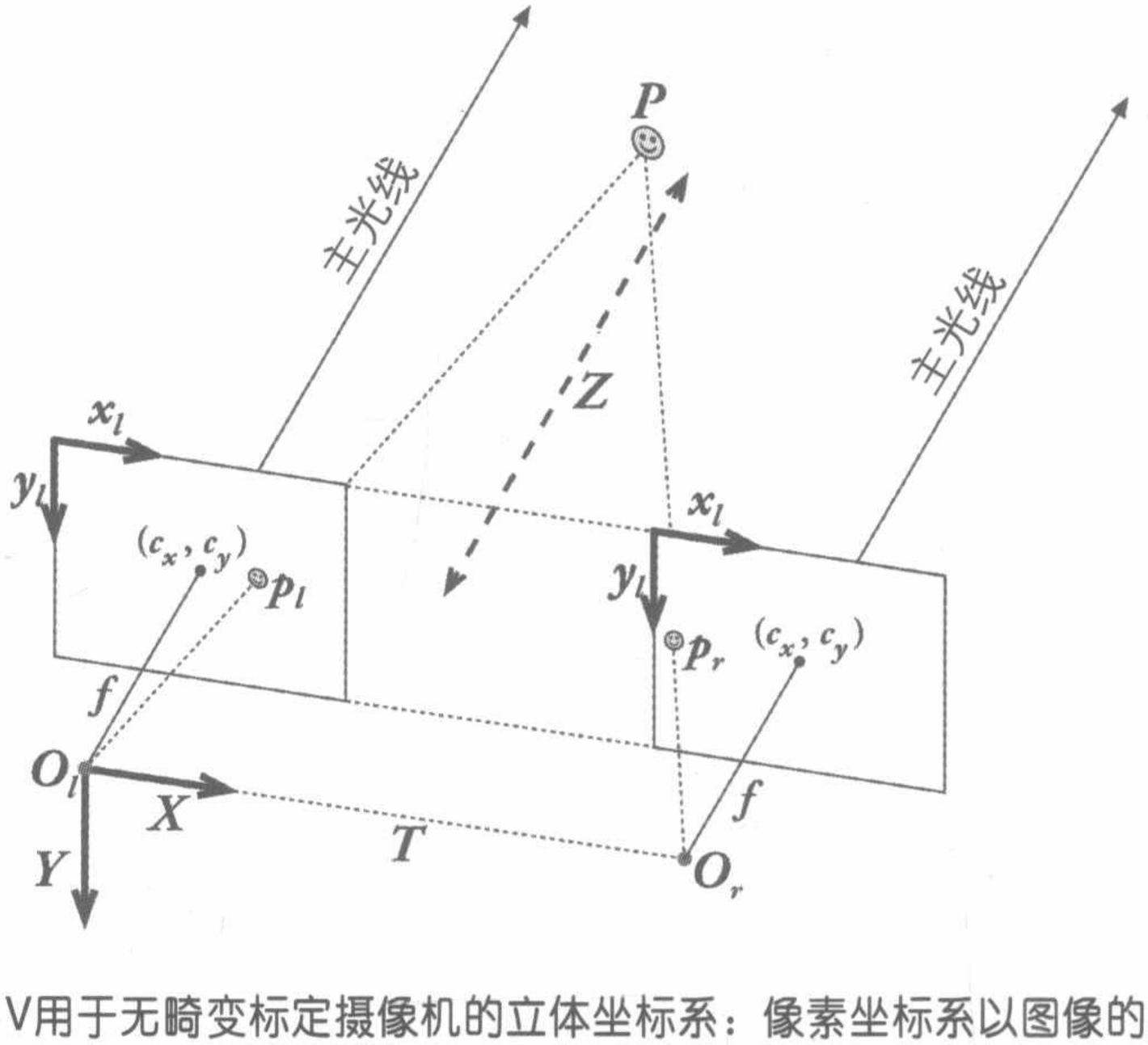


图19-7. OpenCV用于无畸变标定摄像机的立体坐标系：像素坐标系以图像的左上角为原点，两个平面行对齐的，摄像机坐标系以左摄像机的投影中心为原点

学习了这种简化模型后，我们可以继续深入探讨更严谨的任务，了解如何将现实世界的摄像机设备映射到一个类似这种理想布置的几何体。在现实世界中，摄像机不会像图19-5中一样严格的前向平行对准。但是，我们可以通过数学方法找到图像投影和畸变图，将左右图像标定为前向平行排列。在设计立体实验台时，通常最好将摄像机近似前向平行排列，并且尽可能使摄像机水平对准（尽管在某些情况下有意创造汇聚几何图形是有意义的）。这种实际的前向平行对准将使数学变换更容易处理。如果摄像机没有尽可能地对准，那么数学转换的结果将产生严重的图像畸变，从而减少或者消除所产生图像的立体重叠区域。[注17]为了获得较好的结果，同样需要同步摄像机。如果摄像机没有

注17：这个建议的例外情况是我们在一个较近范围内需要更高分辨率的应用；在这种情况下，我们将摄像机稍微倾斜，使它们的主光线在有限距离内相交。经过数学上的对准，这种内向摄像机的效果是引入从视差中减去的x偏移。这可能造成负视差，但是我们可以因此在附近感兴趣的深度处获得更好的深度分辨率。

在同一时刻捕获图像，[注18]那么一旦任一物体在场景中移动（包括摄像机本身），就会出现问题，并且将被限制为使用固定摄像机观察静态场景。

图19-8描述了两台摄像机之间的真实情况以及我们想要实现的数学标定。要实现这种数学标定，我们需要了解更多两个摄像机观察场景的几何体。一旦这个几何体确定，并且可用一些术语和符号来描述，我们就可以回归到标定问题。

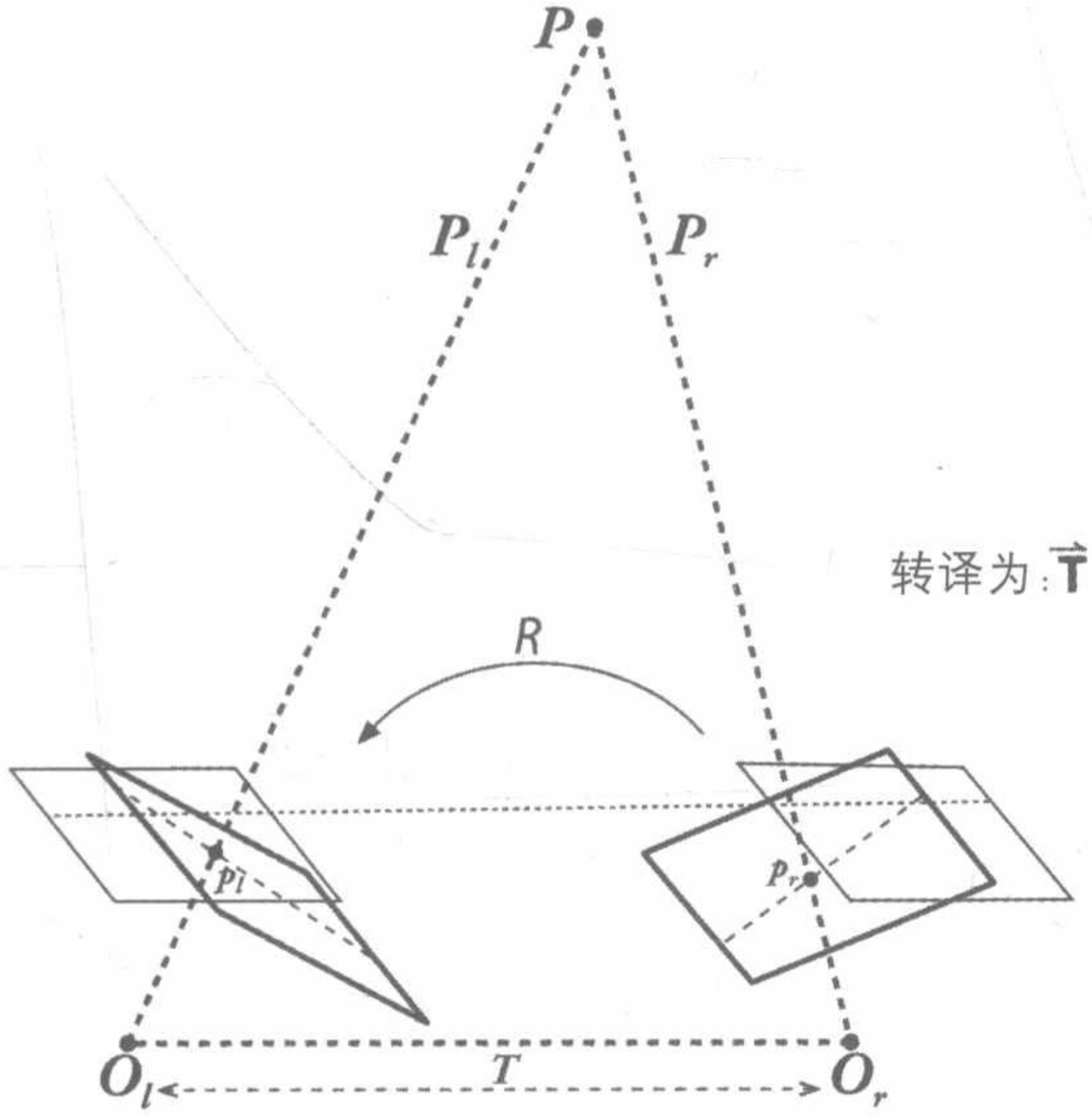


图19-8. 我们的目标是在数学上（而非物理的）将两个摄像机对准到一个观察平面，使得摄像机之间的像素行是彼此精确对准的

对极几何

立体成像系统的几何基础被称作“对极几何”。实质上，这种几何结合了两个针孔模型（每一个针孔对应一个摄像机[注19]）和一些有趣的新点，被称为“极点”（图19-9）。在解释极点有何用处之前，我们首先明确它们的定义，并且引入一些新的相关术语。完成这些之后，我们将对整个对极几何整体有一个简明的了解，并且还会发现我们可以大大缩小两个立体摄像机上对应点的可能位置。这个额外的发现将对实际的立体实现产生重要的影响。

注18：当然，“相同”在背景下确实是个问题。在这里，重要的是场景中没有移动对象，或者摄像机本身没有任何足够快的移动，导致两个成像仪在充分不同的瞬间捕获场景，物体看起来在两次捕获之间移动。

注19：由于我们实际上是在处理真实镜头而非针孔摄像机，所以两个图像没有畸变十分重要，详情可以参考第18章。

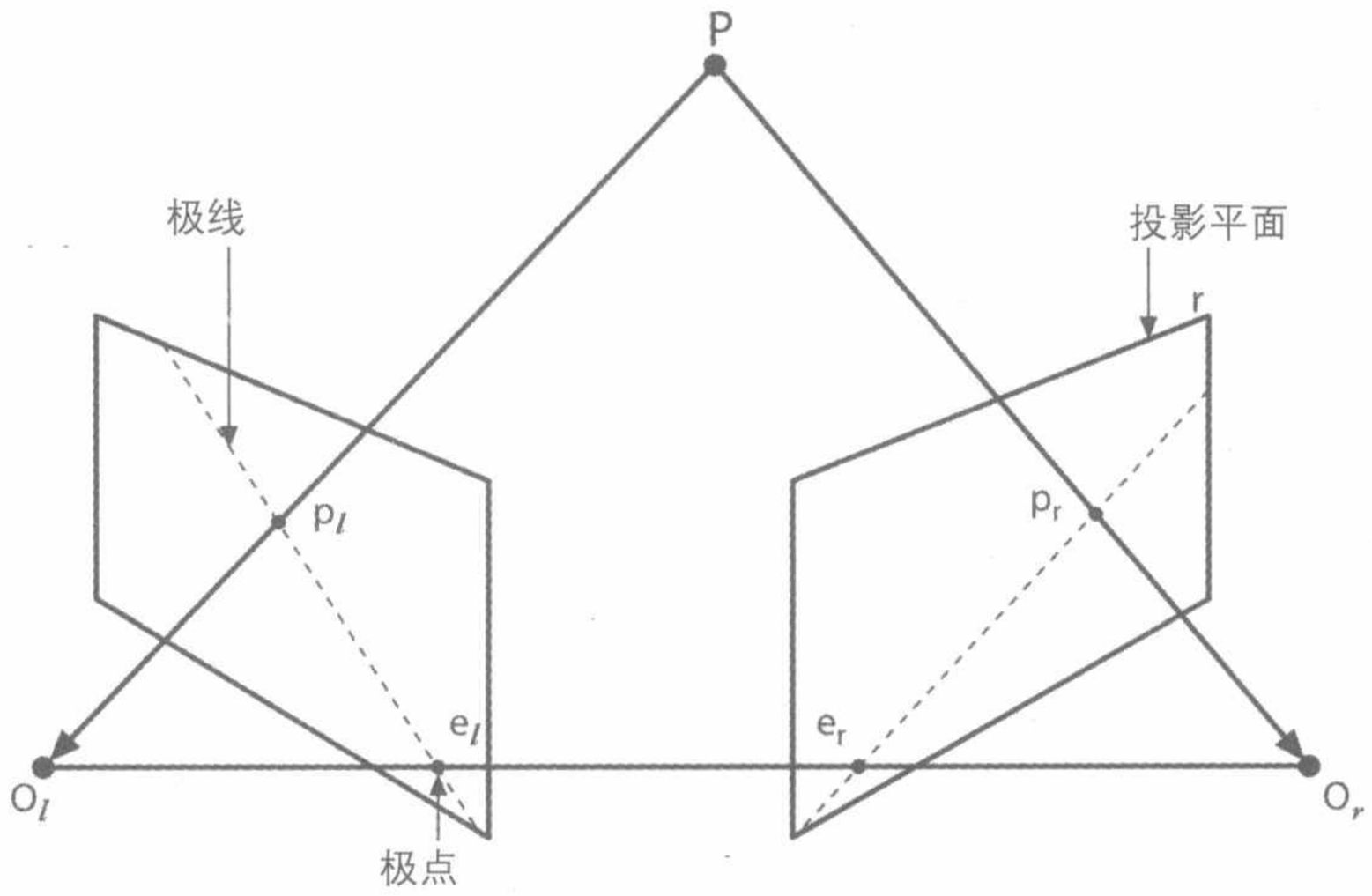


图19-9. 极平面由观测点P和两个投影中心O_l、O_r确定，极点位于投影中心的连线和两个投影平面的交点处

对于每一个摄像机，都有单独的投影中心$\vec{O}_l$和$\vec{O}_r$以及一对对应投影平面Π_l和Π_r。物理世界中的点$\vec{R}$在每个投影平面中都有一个投影，将其记为$\vec{p}_l$和$\vec{p}_l$。新感兴趣的点即为极点。在图像平面Π_l（或Π_r）上的极点$\vec{e}_l$（或$\vec{e}_r$）被定义为该平面上另一个摄像机$\vec{O}_l$（或$\vec{O}_r$）投影中心的图像。由实际观察点P和两个极点$\vec{e}_l$、$\vec{e}_r$（或等价通过两个投影中心$\vec{O}_l$和$\vec{O}_r$）构成的空间中的平面被称为“极面”，线$p_l e_l$和$p_l e_r$（从投影上的点指向对应的极点）被称为“极线”。[注20]

要了解极点的实际用途，首先需要回顾一下，当我们观察到一个物理世界中的点投影到右侧（或者左侧）像平面时，该点实际上可能位于从$\vec{O}_r$出发，指向$\vec{p}_r$（或者从O_l出发指向$\vec{p}_l$）的射线上的任意各处，因为在单摄像机的情况下，我们无法知道观察点与摄像机之间的距离。更具体地说，以观察点p被右侧相机“看到”。由于摄像机仅能观察到$\vec{p}_r$（点$\vec{p}$在平面Π_r上的投影），实际点$\vec{p}$可以位于$\vec{p}_r$和$\vec{O}_r$之间线段上的任意位置。这条线段显然包含点$\vec{p}$，但是其同样包含其他各点。感兴趣的读者可能会问，这条直线投影到左侧像平面Π_l上时会是什么样；实际上，这就是由$\vec{p}_l$和$\vec{e}_l$确定的极线。翻译过来就是，在一个成像仪上看到的所有可能位置的点都是穿过另一个成像仪的极点和对应点的直线图像。

下面总结一下有关立体摄像机对极几何的内容（以及为什么我们要研究这些）。

注20：你可以看出为什么在之前极点没有被提出：当平面趋于完全平行时，极点趋于无穷远。

- 摄像机视图中的每一个三维点都被包含在与每个图像相交的极面中。二者相交产生的直线是极线。
- 给定一幅图像中的特征，在另一图像中的匹配视图一定位于对应极线上。这被称为“极线约束”。
- 极线约束意味着，一旦我们知道立体实验设备之间的对极几何，在两幅图像间匹配特征的二维搜索可以转变为沿着极线的一维搜索。这不仅会极大地节省计算成本，同时会帮助我们排除许多导致虚假匹配的点。
- 顺序保持不变。如果点A和点B在两幅图像上都是可见的，并且按顺序水平出现在其中一个成像仪上，那么两点将在另一个成像仪上按顺序水平出现。[注21]

本征矩阵和基本矩阵

你可能认为接下来会介绍一些OpenCV函数，来计算这些极线，但是在实际介绍函数之前，我们还需要了解一些内容，分别是本征矩阵E和基本矩阵F。[注22]本征矩阵E包含关于物理空间中两个摄像机的平移和旋转的信息（图19-10）；除了包含E中相同的信息外，矩阵F还包含两个摄像机的内参数。[注23]由于F包含内参数信息，它可以在像素坐标系上将两台摄像机关联。

补充说明矩阵E和矩阵F之间的差异：本征矩阵E是一个纯几何的，与成像仪无关。它将左侧摄像机所观察到的点P的物理坐标位置与右侧摄像机所看到的相同的点位置相关联（将$\vec{p}_l$与$\vec{p}_r$关联）。基本矩阵F将图像坐标系（像素）中一个摄像机的像平面上的点与另一个摄像机的像平面上的点关联（在这里，记为$\vec{q}_l$和$\vec{q}_r$）。

注21：由于遮挡和重叠视野的区域，确实有可能两个摄像机不能观察到相同的点。然而，将保持一定顺序。如果点$\vec{A}$，$\vec{B}$和$\vec{C}$在左成像仪上从左到右排列，同时如果由于遮挡在右成像仪上不能看到$\vec{B}$，那么右成像仪依然可以观察到$\vec{A}$和$\vec{C}$从左到右排列。

注22：接下来的小节有一些数学化。如果你不喜欢数学，那就快速浏览一下；除非你有自信能完全理解这些内容。对于简单的应用，你可以不需要在接下来几页中的所有细节知识，仅使用OpenCV提供的机理。

注23：敏锐的读者将发现E和之前小节中的单应矩阵H有几乎完全一样的描述方式。尽管它们通过相似的信息构建，它们并不是同一矩阵并且不能搞混。H定义的本质部分是我们将平面视为是由摄像机观察到的，因此可以将该平面上一点和摄像机上一点联系起来。矩阵E没有这样的假设，所以只能将一个平面上的点与摄像机上的线相关联。

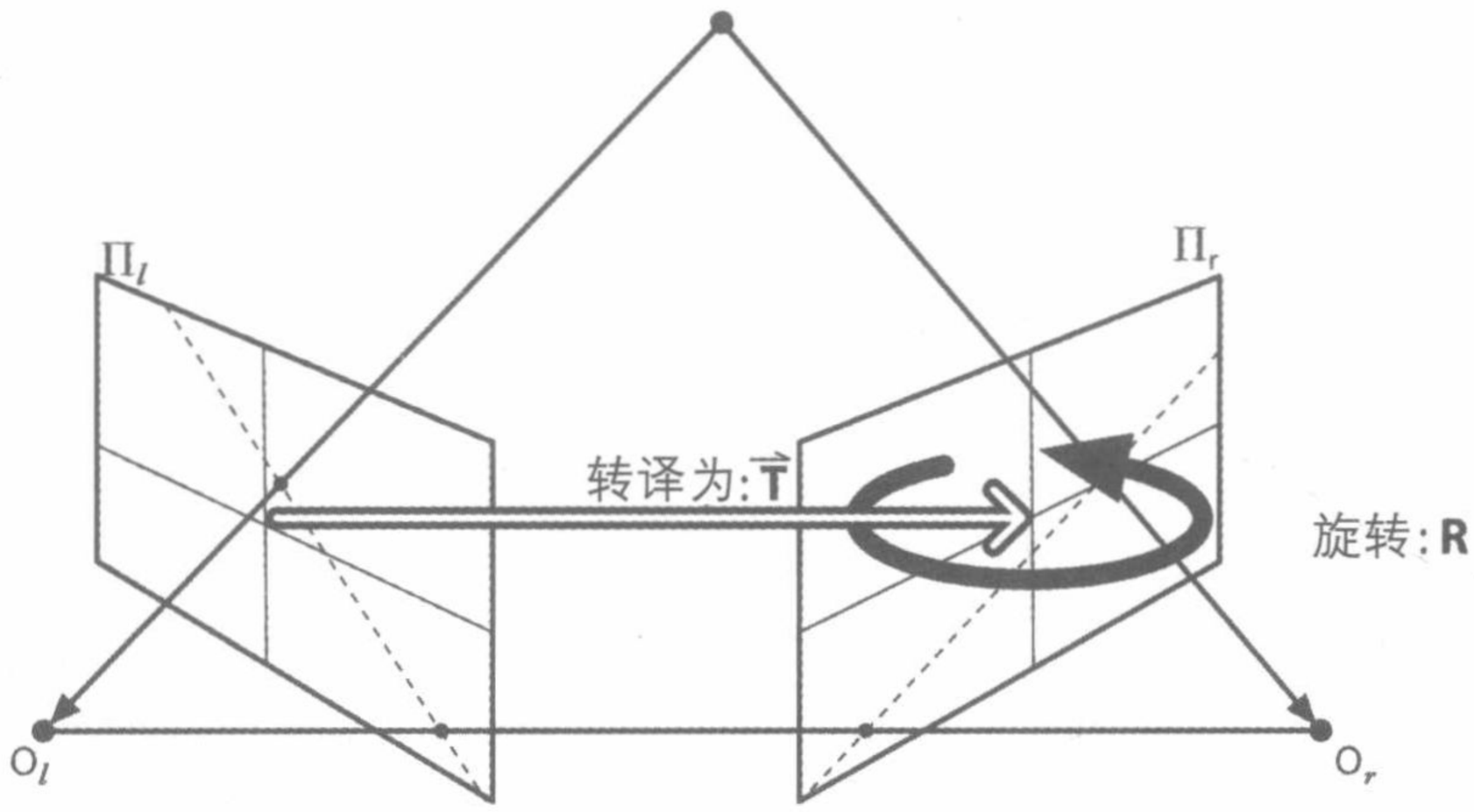


图19-10. 立体成像的本征几何由本征矩阵*E*捕获，该矩阵包含关于平移*T*和旋转*R*的所有信息，描述了球坐标系中第二个摄像机相对于第一个摄像机的位置

本征矩阵的数学原理

现在我们将深入学习一些数学知识，以便更好地理解用来解决立体几何难题的OpenCV函数。

已知一点$\vec{P}$，我们要推导出在两个成像仪上的观测位置$\vec{p}_l$和$\vec{p}_r$之间的关系。这种关系将反过来作为本征矩阵的定义。首先考虑$\vec{p}_l$和$\vec{p}_r$在两个摄像机坐标中观察到的点的物理坐标之间的关系。利用对极几何可以将二者关联起来，就跟我们已经看到的那样。[注24]

选择左侧或者右侧相机的一组坐标系来进行计算。两组当然都没有问题，但我们这里还是选择以左侧相机$\vec{O}_l$为中心的坐标系。在这些坐标系中，观察点的位置为$\vec{p}_l$，另一个摄像机的原点位于$\vec{O}$。在摄像机坐标系中，点$\vec{P}$在右侧摄像机中观察为$\vec{p}_r$，其中$\vec{p}_r = R \cdot (\vec{P}_l - \vec{T})$。这里面关键的一步是引入我们已知的极面来把所有这些关联起来。当然，我们可以用任何方式表示一个平面，但是为了实现我们的目的，让法向量为$\vec{n}$的平面上的所有点x，满足以下约束条件还是很有必要的：

$$(\vec{x} - \vec{a}) \cdot \vec{n} = 0$$

现在回想一下，极面包含向量$\vec{p}_l$和$\vec{T}$；因此如果已知一个垂直于二者的向量（例如，

注24：请不要将投影像平面上的两个点$\vec{p}_l$和$\vec{p}_r$，和两个摄像机坐标系上点$\vec{P}$的位置$\vec{P}_l$和$\vec{P}_r$弄混。

$\vec{P}_l \times \vec{T}$）[注25]，则可以用以表示平面方程中的$\vec{n}$。这样，通过点$\vec{T}$的所有可能点$\vec{P}_l$和包含这两个向量的方程表示为：[注26]

$$(\vec{P}_l - \vec{T})^T (\vec{P}_l \times \vec{T}) = 0$$

记住，我们的目标是通过首先关联$\vec{P}_l$和$\vec{P}_r$来关联$\vec{q}_l$和$\vec{q}_r$。通过等式，将$\vec{P}_r$的代入图像，可以将等式$\vec{P}_r = R \cdot (\vec{P}_l - \vec{T})$改写为：$(\vec{P}_l - \vec{T}) = R^{-1} \cdot \vec{P}_r$。使用这种代换方式，并且根据$R^{-1}=R^T$，将得到；

$$(R^T \cdot \vec{P}_r)^T (\vec{P}_l \times \vec{T}) = 0$$

总是可以将一个叉乘重新写作一个（有些庞大）的矩阵乘法形式。故将矩阵S定义为：

$$\vec{T} \times \vec{P}_l = S \cdot \vec{P}_l \Rightarrow S = \begin{bmatrix} 0 & -T_x & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}$$

这样可以推导出的第一个结论。将上式代入叉乘中，得到：

$$\vec{P}_r^T \cdot R \cdot S \cdot \vec{P}_l = 0$$

乘积$R \cdot S$是我们对本征矩阵E的定义，可以写成下面的简洁形式：

$$\vec{P}_r^T \cdot E \cdot \vec{P}_l = 0$$

当然，我们实际需要的是在成像仪上观察点之间的关系，但这仅仅是其中的一步。利用投影公式$\vec{p}_l = (f_l / z_l) \cdot \vec{P}_l$和$\vec{p}_r = (f_r / z_r) \cdot \vec{P}_r$进行简化代换，之后用$Z_l Z_r / f_l f_r$除整个部分，得到最终结果：

$$\vec{p}_r^T \cdot E \cdot \vec{p}_l$$

这看起来像是如果已知其他项，就可以完全确定p项，但是E是一个秩亏矩阵[注27]（一个

注25：向量的交叉相乘产生一个与前两个向量正交的第三个向量。它的方向由“右手定则”确定：如果你指向$\hat{a}$并弯曲中指指向$\hat{b}$，那么叉积$\hat{a} \times \hat{b}$垂直于$\hat{a}$和$\hat{b}$并沿着拇指的方向。

注26：在这里，我们利用向量的转置代替点乘的矩阵乘法。

注27：对于像E一样的$n \times n$方阵，秩不足基本上意味着存在少于n个非零特征值。因此，由秩缺矩阵确定的线性方程组不具有唯一解。如果秩（非零特征值的数量）是$n-1$，将存在由一系列点组成的直线，所有点都满足方程组。由秩为$n-2$的矩阵确定的方程组将组成一个平面，以此类推。

秩为2的3 × 3本征矩阵），以致这个矩阵实际上最终是一条线的方程。在本征矩阵中有五个参数——三个用于旋转，两个用于平移的方向（缩放没有设置）——以及其他两个约束。本征矩阵中的另外两个约束是：(1)行列式为0，因为该矩阵是秩亏矩阵；(2)两个非零奇异值相等，因为矩阵S是对称的，R是旋转矩阵。这总共产生了七个约束。再次注意，E中不包含摄像机的内参数；因此它在物理或者摄像机坐标系中将各点彼此相关联，而非像素坐标系。

基本矩阵数学

矩阵E包含两个摄像机相关的所有几何信息，但不包括摄像机本身的任何信息。在实践中，我们通常对像素坐标系感兴趣。为了找到一个图像中的像素和另一个图像中相对应的极线之间的关系，必须引入两个摄像机的内参数信息。为此，对于$\vec{p}$（像素坐标），用$\vec{q}$和相关的摄像机内在矩阵代替。已知$\vec{q} = M \cdot \vec{p}$（其中$M$是摄像机的内参数矩阵），或者等价地：$\vec{p} = M^{-1} \cdot \vec{q}$。因此$E$的等式可以描述为：

$$\vec{q}_r^{\,T} \cdot (M_r^{-1})^T \cdot E \cdot M_l^{-1} \cdot \vec{q}_l = 0$$

尽管这看起来有一些混乱，但是可以通过定义基本矩阵F来整理：

$$F = (M_r^{-1})^T \cdot E \cdot M_l^{-1}$$

因此：

$$\vec{q}_r^{\,T} \cdot F \cdot \vec{q}_l = 0$$

简而言之，基本矩阵F和本征矩阵E没有什么差别，除了基础矩阵F在图像像素坐标中操作，而本征矩阵E在物理坐标系中操作。类似本征矩阵E，[注28]基本矩阵F的秩为2。基本矩阵有七个参数，两个表示极，三个表示两个像平面的单应矩阵（通常的四个参数中没有缩放因素）。

OpenCV的处理方法

我们可以通过提供一些已知对应关系，以类似前面部分中计算图像单应性的方式来计算F。在这种情况下，我们甚至不需要分别校准摄像机，因为我们可以直接对F进行处理，其中隐含地包含两个摄像机的基本矩阵。处理整个过程的这个例程叫做`cv::findFundamentalMat()`：

注28：注意这个联系着基本矩阵和本征矩阵的方程。如果我们标定平面并通过除以焦距对点进行归一化，那么内参数矩阵M成为单位阵，同时$F = E$。

```
cv::Mat cv::findFundamentalMat(                 // computed fundamental mtx
  cv::InputArray  points1,                      // Points image 1 (floats)
  cv::InputArray  points2,                      // Points image 2 (floats)
  int             method = cv::FM_RANSAC,       // Method for computing mtx
  double          param1 = 3.0,                 // RANSAC max distance
  double          param2 = 0.99,                // RANSAC/LmedS confidence
  cv::OutputArray mask   = cv::noArray()        // (Optional) array
                                                // indicates used elements
);
```

前两个参数是二维或者三维点的数组，以任何常规方式排列。[注29]

第三个参数决定了通过对应点计算基本矩阵的方法，并且可以采用四个值之中的一个。对于每个值，对于point1和point2中所需（或允许的）点的数量有特定的约束，如表19-2所示。

表19-2：cv::findFundamentalMat()中方法参数的约束

方法的值	点数	算法
cv::FM_7POINT	$N=7$	7点算法
cv::FM_8POINT	$N\geq 8$	8点算法
cv::FM_RANSAC	$N\geq 8$	RANSAC算法
cv::FM_LMEDS	$N\geq 8$	LMedS算法

7点算法只使用7个点，并且遵循矩阵F的秩必须为2，来完全约束矩阵。这个约束的优点在于，矩阵F的秩永远为2，使其不可能有一个不为零、非常小的特征值。其缺点在于，这个约束不是唯一的，因此函数有可能返回三个不同的矩阵（这就是为什么返回的数组可能是9×3的矩阵，所有三个返回都要顾及）。8点算法通过线性方程组对矩阵F进行求解。如果提供超过8个点，则要求所有点的线性最小二乘解。7点算法和8点算法的问题是，它们对异常值十分敏感（即使在八点算法中有超过8个点）。这个问题由*RANSAC*算法和*LMedS*算法解决，正如在第18章看到的，它们被划分为鲁棒算法，因为它们具有辨识和消除异常值的能力。[注30]对于这两种方法，都要求最少输入8个点。

下面两个参数仅被RANSAC算法和LMedS算法所使用。第一个是param1，被RANSAC算法使用，是从点到极线（像素）的最大距离，超过此距离被认为是异常值。第二个参数

注29：你可能想知道$N\times 3$或者三通道矩阵是什么。算法可以很好地处理在标定对象中测量得到的实际三维点（x, y, z）。三维点最终将被缩放为(x/z, y/z)，或者可以在单应性坐标系中输入二维点（x, y, 1），用相同的方式处理。如果你输入（x, y, 0），那么算法将忽略0。很少见到使用实际三维点的，因为在标定物体上通常只能检测到二维点。

注30：想要获得更多信息，可以参考原始文献：Fischler and Bolles [Fischler81]以了解RANSAC；Rousseeuw [Rousseeuw84]以获得最小中值平方；Inui, Kaneko, and Igarashi [Inui03]以获得使用LMedS的直线拟合。

param2被RANSAC算法和LMedS算法使用，是期望置信度（0~1之间），实际上告诉算法的迭代次数。

返回的结果是一个cv::Mat数组，通常是一个与点精度相同的3×3的矩阵。（回忆在7点算法的特例中，同样可能返回9×3的数组）。经过计算的基本矩阵之后通常被传递到函数cv::computeCorrespondEpilines()，以找到对应指定点的极线，或者传递到cv::stereoRectifyUncalibrated()，以计算得到标定变换。示例19-2给出了寻找基本矩阵的示例程序。

示例19-2：利用RANSAC算法计算基本矩阵

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

void help( char* argv[] ) {
...
}

// args: [board_w] [board_h] [number_of_boards] [delay]? [scale]?
//
int main( int argc, char* argv[] ) {

  int  n_boards = 0;       // Will be set by input list
float image_sf = 0.5f;
float delay    = 1.f;
int  board_w  = 0;
int  board_h  = 0;

if( argc < 4 || argc > 6 ) {
  cout << "\nERROR: Wrong number of input parameters";
  help( argv );
  return -1;
}
board_w  = atoi( argv[1] );
board_h  = atoi( argv[2] );
n_boards = atoi( argv[3] );
if( argc > 4 ) delay    = atof( argv[4] );
if( argc > 5 ) image_sf = atof( argv[5] );

int board_n       = board_w * board_h;
cv::Size board_sz = cv::Size( board_w, board_h );
cv::VideoCapture capture(0);
if( !capture.isOpened() ) {
  cout << "\nCouldn't open the camera\n"; help();
  return -1;
}

// Allocate Storage
//
vector< vector< cv::Point2f> > image_points;
vector< vector< cv::Point3f> > object_points;
```

```
// Capture corner views; loop until we've got n_boards number of
// successful captures (meaning: all corners on each
// board are found).
//
double last_captured_timestamp = 0;
cv::Size image_size;
while( image_points.size() < (size_t) n_boards ) {

  cv::Mat image0, image;
  capture >> image0;
  image_size = image0.size();
  resize(
    image0, image, cv::Size(),
    image_sf, image_sf, cv::INTER_LINEAR
);

// Find the board
//
vector< cv::Point2f > corners;
bool found = cv::findChessboardCorners(
 image, board_sz, corners
);

// Draw it
//
cv::drawChessboardCorners( image, board_sz, corners, found );

// If we got a good board, add it to our data
//
double timestamp = (double) clock() / CLOCKS_PER_SEC;
if( found && timestamp - last_captured_timestamp > 1 ) {

  last_captured_timestamp = timestamp;
  image ^= cv::Scalar::all(255);
  cv::Mat mcorners( corners );       // do not copy the data
  mcorners *= ( 1./ image_sf );      // scale corner coordinates
  image_points.push_back( corners );
  object_points.push_back( vector< cv::Point3f >() );
  vector< cv::Point3f >& opts = object_points.back();
  opts.resize( board_n );
  for( int j=0; j<board_n; j++ ) {
    opts[j] = cv::Point3f(
      (float)(j/board_w), (float)(j%board_w), 0.f
    );
  }
    cout << "Collected our " <<(int) image_points.size()
      <<" of " << n_boards <<" needed chessboard images\n" << endl;
  }

  // in color if we did collect the image
  //
  cv::imshow( "Calibration", image );
  if( (cv::waitKey(30) & 255) == 27 )
    return -1;

}           // end collection while() loop.
```

```
cv::destroyWindow("Calibration");
cout <<"\n\n*** CALIBRATING THE CAMERA...\n" << endl;

// Calibrate the camera!
//
cv::Mat intrinsic_matrix, distortion_coeffs;
double err = cv::calibrateCamera(
  object_points,       // Vector of vectors of points
                       // from the calibration pattern
  image_points,        // Vector of vectors of projected
                       // locations (on images)
  image_size,          // Size of images used
  intrinsic_matrix,    // Output camera matrix
  distortion_coeffs,   // Output distortion coefficients
  cv::noArray(),       // We'll pass on the rotation vectors...
  cv::noArray(),       // ...and the translation vectors
  cv::CALIB_ZERO_TANGENT_DIST | cv::CALIB_FIX_PRINCIPAL_POINT
);

// Save the intrinsics and distortions
cout << " *** DONE!\n\nReprojection error is " << err
  <<"\nStoring Intrinsics.xml and Distortions.xml files\n\n";
cv::FileStorage fs( "intrinsics.xml", cv::FileStorage::WRITE );

fs << "image_width"     << image_size.width << "image_height"
   << image_size.height << "camera_matrix" << intrinsic_matrix
   << "distortion_coefficients" << distortion_coeffs;
fs.release();

// Example of loading these matrices back in:
//
fs.open( "intrinsics.xml", cv::FileStorage::READ );
cout << "\nimage width: "  << (int)fs["image_width"];
cout << "\nimage height: " << (int)fs["image_height"];

cv::Mat intrinsic_matrix_loaded, distortion_coeffs_loaded;
fs["camera_matrix"]           >> intrinsic_matrix_loaded;
fs["distortion_coefficients"]  >> distortion_coeffs_loaded;
cout << "\nintrinsic matrix:"        << intrinsic_matrix_loaded;
cout << "\ndistortion coefficients: " << distortion_coeffs_loaded
     << endl;

// Compute Fundamental Matrix Between the first
// and the second frames:
//
cv::undistortPoints(
  image_points[0],     // Observed point coordinates (from frame 0)
  image_points[0],     // undistorted coordinates (in this case,
                       // the same array as above)
  intrinsic_matrix,    // Intrinsics, from cv::calibrateCamera()
  distortion_coeffs,   // Distortion coefficients, also
                       // from cv::calibrateCamera()
  cv::Mat(),           // Rectification transformation (but
                       // here, we don't need this)
  intrinsic_matrix     // New camera matrix
);
```

```
cv::undistortPoints(
  image_points[1],     // Observed point coordinates (from frame 1)
  image_points[1],     // undistorted coordinates (in this case,
                       // the same array as above)
  intrinsic_matrix,    // Intrinsics, from cv::calibrateCamera()
  distortion_coeffs,   // Distortion coefficients, also
                       // from cv::calibrateCamera()
  cv::Mat(),           // Rectification transformation (but
                       // here, we don't need this)
  intrinsic_matrix     // New camera matrix
);

// Since all the found chessboard corners are inliers, i.e., they
// must satisfy epipolar constraints, here we are using the
// fastest, and the most accurate (in this case) 8-point algorithm.
//
cv::Mat F = cv::findFundamentalMat(  // Return computed matrix
  image_points[0],                   // Points from frame 0
  image_points[1],                   // Points from frame 1
  cv::FM_8POINT                      // Use the 8-point algorithm
);
cout << "Fundamental matrix: " << F << endl;

// Build the undistort map which we will use for all
// subsequent frames.
//
cv::Mat map1, map2;
cv::initUndistortRectifyMap(
  intrinsic_matrix_loaded,  // Our camera matrix
  distortion_coeffs_loaded, // Our distortion coefficients
  cv::Mat(),                // (Optional) Rectification, don't
                            // need.
  intrinsic_matrix_loaded,  // "New" matrix, here it's the same
                            // as the first argument.
  image_size,               // Size of undistorted image we want
  CV_16SC2,                 // Specifies the format of map to use
  map1,                     // Integerized coordinates
  map2                      // Fixed-point offsets for
                            // elements of map1
);
// Just run the camera to the screen, now showing the raw and
// the undistorted image.
//
for(;;) {
  cv::Mat image, image0;
  capture >> image0;
  if( image0.empty() ) break;
  cv::remap(
    image0,                 // Input image
    image,                  // Output image
    map1,                   // Integer part of map
    map2,                   // Fixed point part of map
    cv::INTER_LINEAR,
    cv::BORDER_CONSTANT,
    cv::Scalar()            // Set border values to black
  );
```

```
    cv::imshow( "Undistorted", image );
    if( ( cv::waitKey(30) & 255 ) == 27 ) break;
  }
  return 1;
}
```

一点警告——与返回cv::noArray()的可能性有关——如果提供的点形成了退化形态，会导致算法失败。当提高的点少于所需信息的数量时，将产生退化形态，例如当一个点出现不止一次时或者多个点共线或与其他点共面时。三不五时地检查cv::findFundamentalMat()的返回值是十分重要的。

计算极线

现在我们已经获得了基本矩阵，希望可以计算极线。OpenCV函数cv::computeCorrespondEpilines()根据一幅图像中的点列，计算其在另一幅图像中对应的极线。注意，一幅图像中给定的任意点，在另一幅图像中都有不同对应的极线。每条计算的极线以三点（a, b, c）的形式编码，这样一来，极线定义如下：

$$a \cdot x + b \cdot y + c = 0$$

为了计算这些极线，函数需要利用通过函数cv::findFundamentalMat()得到的基本矩阵：

```
void cv::computeCorrespondEpilines(
  cv::InputArray  points,     // Input points, Nx1 or 1xN (Nc=2)
                              //  or vector<Point2f>
  int             whichImage, // Index of image which contains
                              //  points ('1' or '2')
  cv::InputArray  F,          // Fundamental matrix
  cv::OutputArray lines       // Output vector of lines, encoded as
                              //  tuples (a,b,c)
);
```

这里的第一个参数points，是二维或三维点的输入数组——它可以为任何常规形式，但是这些点应该为浮点型。参数whichImage必须是1或2，并标明哪幅图像的点被定义，与函数中的数组points1和points2相关联。F是由函数cv::findFundamentalMat()返回的3×3的矩阵。最后，lines是一个浮点型数组，表示将写入的结果极线。每一条极线都以一个三元素向量$\vec{L}=(a,b,c)$的形式编码，包含极线等式$a \cdot x+b \cdot y + c=0$的参数。由于极线等式与参数$a,b,c$的所有归一化无关，它们被默认归一化，使得$a^2 + b^2=1$。

立体校正

我们已经建立了很多关于摄像机和三维点的理论机制，并可以使用。这一节介绍立体校正，下一节介绍立体校正。立体校正是计算空间中两幅图像之间几何关系的过程。与之相对反，立体校正是“纠正”各个图像的过程，使之它们看起来像是由两个像平面行对

准的两个摄像机拍摄（回顾图19-5和图19-8）。通过校正，两个摄像机的光轴（或主射线）是平行的（即所谓的相交于无穷远处）。当然我们可以将两个摄像机图像标定成为任何形态，但是在这里（在OpenCV中）我们关注一些更普遍和简明的实例，来使主光线相交于无穷远处。

立体校正取决于查找两个摄像机之间的旋转矩阵R和$\vec{T}$平移矢量T，如图19-10所示。R和都通过函数cv::stereoCalibrate()计算，与第18章中看到的函数cv::calibrateCamera()类似，只不过我们现在有两个摄像机，并且新函数可以计算（或者利用之前的计算）摄像机的畸变矩阵、本征矩阵和基本矩阵。立体校正和单摄像机标定之间另一个主要的区别是，在函数cv::calibrateCamera()中，我们以摄像机和棋盘视图之间的一系列旋转和平移结束。而在函数cv::stereoCamera()中，我们搜寻单个旋转矩阵和平移向量来联系左右摄像机。

我们已经说明了如何计算本征矩阵和基本矩阵。接下来的问题是，如何计算左右摄像机的R和和$\vec{T}$。我们从对任意给定三维点$\vec{P}$的物体坐标系的观察开始，可以分别对两个摄像机使用单摄像机标定，将$\vec{P}$输入每个摄像机的摄像机坐标系：$\vec{P}_l = R_l \cdot \vec{P} + \vec{T}_l$，$\vec{P}_r = R_r \cdot \vec{P} + \vec{T}_r$（分别对应左右摄像机）。从图19-10也能清晰看出$\vec{P}$的两个视图（从两个摄像机中）与$\vec{P}_l = R^T \cdot (\vec{P}_r - \vec{T})$相关，[注31]其中$R$和$\vec{T}$分别是摄像机之间的旋转矩阵和平移向量。通过这三个等式并且分别求解旋转矩阵和平移向量，得到如下简单的关系：[注32]

$$R = R_r \cdot R_l^T$$

和

$$\vec{T} = \vec{T}_r - R \cdot \vec{T}_l$$

给定棋盘角点或类似标定物体的多个联合视图，cv::stereoCalibrate()使用cv::calibrateCamera()来计算每一个摄像机分别的旋转和平移参数（参考第18章中“实质是什么”那一小节的讨论来回忆这一步如何实现）。之后，该例程将这些左右旋转和平移的解代入上述等式中，求解两个摄像机之间的旋转和平移参数。由于图像噪声和舍入误差，每一对棋盘都会使R和$\vec{T}$的值产生微小的差别。cv::stereoCalibrate()将R和$\vec{T}$参数的中间值作为真解的初始近似值并采用鲁棒的Levenberg-Marquardt迭代算

注31：注意这些术语的含义：$\vec{P}_l$和$\vec{P}_r$分别表示三维点$\vec{P}$在左右摄像机的坐标系上的位置；R_l和$\vec{T}_l$(右侧分别为R_r和$\vec{T}_r$)表示从摄像机到左侧（右侧类似）摄像机上三维点的旋转和平移矢量；R和$\vec{T}$是将右侧摄像机坐标系代换到左侧的旋转和平移。

注32：可以反转两个方程的下标或者通过反转下标并将R的转置带入转换方程，以此来逆转这些方程中的左右摄像机。

法，已找到棋盘角点在两个摄像机视图上的（局部地）最小投影误差，并返回R和$\vec{T}$的最终结果。为了使立体校正的结果更清晰：旋转矩阵被用作到与左摄像机共面的右摄像机上；这样就保证了两个像平面平行但是不是行对准的（在后面“立体校正”一节中，我们将看到行对准是如何实现的）。

函数cv::stereoCalibrate()有很多参数，但是它们都相当简单，许多与第18章中的函数cv::calibarteCamera()相同：

```
double cv::stereoCalibrate(                 // Return reprojection error
  cv::InputArrayOfArrays objectPoints,  // Vector of vectors of
                                        // calib. pattern points
  cv::InputArrayOfArrays imagePoints1,  // Vector of vectors of
                                        // image points (cam 1)
  cv::InputArrayOfArrays imagePoints2,  // Vector of vectors of
                                        // image points (cam 2)
  cv::InputOutputArray   cameraMatrix1, // Intrinsics for cam 1
                                        // (input/output)
  cv::InputOutputArray   distCoeffs1,   // Distortion coeffs for
                                        // cam 1 (input/output)
  cv::InputOutputArray   cameraMatrix2, // Intrinsics for cam 2 (
                                        // input/output)
  cv::InputOutputArray   distCoeffs2,   // Distortion coeffs for
                                        // cam 2 (input/output)
  cv::Size               imageSize,     // Size of images (assume both
                                        // cams are same)
  cv::OutputArray        R,             // Computed relative
                                        // rotation *matrix*
  cv::OutputArray        T,             // Computed relative
                                        // translation vector
  cv::OutputArray        E,             // Computed essential matrix
  cv::OutputArray        F,             // Computed fundamental matrix
  cv::TermCriteria       criteria     = cv::TermCriteria(
                                          cv::TermCriteria::COUNT
                                            | cv::TermCriteria::EPS,
                                          30,
                                          1e-6
                                        ),
  int                    flags = cv::CALIB_FIX_INTRINSIC
);
```

第一个参数objectPoints是一组点数组。数组顶层中的每一个条目与其中一个校准图像相关联。每个这样的条目本身就是一个包含标定图像（在标定图像本身的坐标系上）上点的位置的数组。这些点需要为三维点，尽管在一些情况下每一个点位置的z坐标将为零。这与函数cv::calibrateCamera()的情况完全相同，实际上并不必须使用平面标定对象，但通常这样做最方便。

imagePoints1和imagePoints1同样也是数组，同样，数组顶层包含与输入图像相对应的条目每一个条目包含观测到的标定点的位置；imagePoints1包含第一个（通常是左侧）

摄像机观察到的点，imagePoints2包含第二个（通常是右侧）摄像机观察到的点。[注33] 与objectPoints中的点不同，imagePoints1和imagePoints2中的点是二维点，在图像中它们是像素位置。



如果使用棋盘或圆网格对两个摄像机进行标定，那么imagePoints1和imagePoints2就只是相应调用函数cv::findChessboardCorners()的返回值或另一个角（或圆）的网格查找函数的返回值，分别用于左右摄像机视角。

参数cameraMartix1和cameraMartix2都是3×3的摄像机矩阵，distCoeffs1和distCoeffs2分别是摄像机1和摄像机2的四项（或五项或七项）畸变系数向量。记住，在这些矩阵中，前两个径向参数首先出现；紧接着是两个切向参数，最后是后续的径向参数（参考第18章对畸变系数的讨论）。[注34]

cv::stereoCalibrate()使用摄像机内联函数的方式由flags参数控制。如果flags包含cv::XALIB_FIX_INTRINSIC，则使用这些矩阵中的值，因为它们是已经通过标定过程得到（即，它们不是由算法计算得到）。如果flags设置为cv::XALIB_FIX_INTRINSIC_GUESS，那么这些矩阵被用作进一步优化每个摄像机的内联和畸变参数的初始值，并将其从cv::stereoCalibrate()中的返回值设为改进值。在这两个flags都未使用的情况下，摄像机的内联函数由cvStereoCalibrate()从头开始计算。因此，如果愿意，使用cvStereoCalibrate()可以一次性计算内参数、外参数和立体参数。[注35]

同时可以添加任何提供给cv::calibrateCamera()的flags值（如第18章所述）。除此之外，还有一个新的flag可用：cv::CALIB_SAME_FOCAL_LENGTH。这个标志对于cv::CALIB_FIX_FOCAL_LENGTH来说是一个不太严格的选择。虽然后者强制焦距与摄像机矩阵1、2中的焦距相等，前者仅要求摄像机矩阵焦距之间相等。在这种情况下，cv::stereoCalibrate()将解决两个摄像机的单个未知焦距。

注33：为了简化，我们通常认为“1”代表左侧摄像机，“2”代表右侧摄像机。实际上，只要在讨论中始终采用相反的表达方式来对待所产生的旋转和平移方案，二者可以换用。最重要的事是在物理上对准摄像机，使其扫描线大致匹配，从而得到较好的标定结果。

注34：第三个和更高的径向畸变最后考虑，因为它们在OpenCV的开发中引入较晚。

注35：注意。一次性尝试解决过多参数有时会导致结果产生分歧，成为没有意义的值。求解方程组有时是一门艺术，并且必须验证结果。你可以在标定和标定代码实例中看到其中一些注意事项，通过使用示例19-2中的极性约束来检验我们的标定结果。

参数imagesize是像素中图像大小。仅用于细化或计算内参数，或者当flags不等于cv::CALIB_FIX_INTRINSIC时。

R和T项输出参数，在函数返回时被填入待求的（联系左右摄像机的）旋转矩阵和平移向量。参数E和F是可选的。如果二者没有被设置为cv::noArray()，那么函数cv::stereoCalibrate()将使用3×3的本征矩阵和基本矩阵进行计算并填入这些数组。最后还有termCrit，我们在之前见过很多次。它将内部优化设置为在一定数量的迭代后终止或者当计算参数小于termCrit结构中指示的阈值时停止。这个函数的典型参数是cv::TermCriteria(cv::TermCriteria::COUNT | cv::TermCriteria::EPS, 30, 1e-6)。在cv::TERMCRIT_EPS的情况下，相关联的终止标准的值是由当前参数估计给出的重新投影的误差值。就像cv::calibrateCamera()一样，算法将对所有可用视图上的所有点的总重新投影误差最小化，但是现在对于两个摄像机，cv::stereoCalibrate()的返回值就是重新投影误差的最终值。



如果已经对两个摄像机进行标定，并确定了结果，则可以使用cv::CALIB_FIX_INTRINSIC“硬设置”之前单个摄像机的标定结果。如果认为两个摄像机额初始标定可以但是不够好，则可以使用算法，通过设置cv::CALIB_USE_INTRINSIC_GUESS中的flags细化内参数和畸变参数。如果摄像机没有单独标定，则可以用与第18章中函数cv::calibrateCamera()中flags参数相同的设置。

一旦有了旋转和平移值（R，$\vec{T}$），或者基本矩阵F，我们就可以使用这些结果来标定两幅立体图像，使得极线沿着图像行对准，且扫描线在两个图像上是相同的。尽管R和T并不决定唯一的立体校正，我们将在下一小节中探讨这些项如何与其他约束一起对图像进行标定。

立体校正

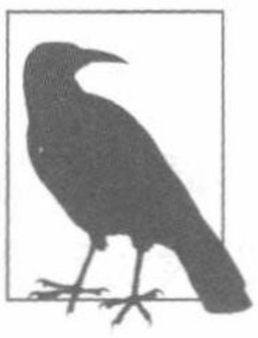
当两个像平面精确对准时，最容易计算立体视差（图19-5）。但不幸的是，正如之前讨论的，在实际立体系统中理想的对准配置并不存在，因为两个摄像机几乎不存在完全共面的行对齐的像平面。图19-8说明了立体校正的目的：将两个摄像机的像平面重新映射，使二者位于完全相同的平面上，图像行完全对准到前向平行配置。如何选择在数学上对齐摄像机的特定平面取决于所使用的算法。接下来，我们将讨论OpenCV提出的两种情况。

最后，经过标定，我们希望两个摄像机之间的图像行是对准的，以使立体对应（在两个不同摄像机视图中找到同一个点）可靠和计算容易。特别的，可靠性和计算效率都通过只搜索一行以与另一图像中的点进行匹配来增强。在包含每个图像的公共像平面进行水

平对准的结果是，极点都位于无穷远；即在一幅图像上的投影中心成像与另一个像平面平行。但是，由于可以选择的前向平行平面的数量是有限的，我们仍然需要引入一些约束，这些约束取决于视图重叠最大化或畸变最小化。

对准两个像平面过程的结果有八项，每四个分别对应左右摄像机。对于每个摄像机将得到一个畸变矢量distCoeffs、一个旋转矩阵R_{rect}（应用于摄像机）、标定和未标定的摄像机矩阵（分别为M_{rect}和M）。通过这些项，利用函数cv::initUndistortRectifyMap()，我们可以创建一个映射，表明我们从原始图像中插入像素的位置，以创建一个新的标定图像。[注36]

有很多算法可以计算标定项，OpenCV实现了两个：(1)Hartley算法[Hartley98]，只使用基本矩阵产生非标定立体视觉；(2)Bouguet算法，[注37]使用两个标定摄像机中的旋转和平移参数。Hartley算法可以通过单个摄像机记录的运动推导出立体结构，虽然与Bouguet标定算法相比，（立体校正时）将产生更多畸变图像。



在可以使用标定模式的情况下——例如机械臂或安全摄像机装备上——使用Bouguet算法更自然。

非标定立体校正：Hartley算法

Hartley算法旨在找到将极点映射到无穷远处的单应矩阵，同时使两幅立体图像之间的计算误差最小化，这可以通过匹配两幅图像之间的对应点实现。通过这种方法，我们避免了计算两个摄像机的摄像机内参数，因为这些内参数信息隐含在点的匹配中。因此，我们只需要计算基本矩阵，它可以通过如前述的函数cv::findFundamentalMat()所获得的的两个场景视图中的7个及以上的一系列匹配点来获得。或者，基本矩阵可以通过函数cv::stereoCalibrate()计算得到。

Hartley算法的优点是我们可以通过观察场景中的点简便地进行在线立体校正。其缺点是场景的图像尺寸未知。例如，如果用棋盘来生成点对，那么我们将无从知道棋盘从一端到另一端是100米还是100厘米。同时我们也将无法清楚地知道摄像机内参数矩阵，因为

注36：只有当极点位于图像矩形外时，OpenCV中图像的立体标定才可能存在。因此，这个标定算法不适用于基线非常宽或者多个摄像机非常趋近彼此的立体配置。

注37：Bouguet算法是对Tsai[Tsai87]和Zhang[Zhang99; Zhang00]提出算法的完善和简化。在知名的matlab标定工具箱实现了他的标定算法之前，Jean-Yves Bouguet一直没有公开过他的算法。

摄像机可能有不同的焦距、倾斜像素、不同的投影中心和/或不同的主点。因此我们只能通过投影变换决定三维物体重构。这意味着同一个物体的不同尺寸或者投影可以看起来是相同的（比如说，特征点有相同的二维坐标，即使三维物体不同）。这些问题如图19-11所示。

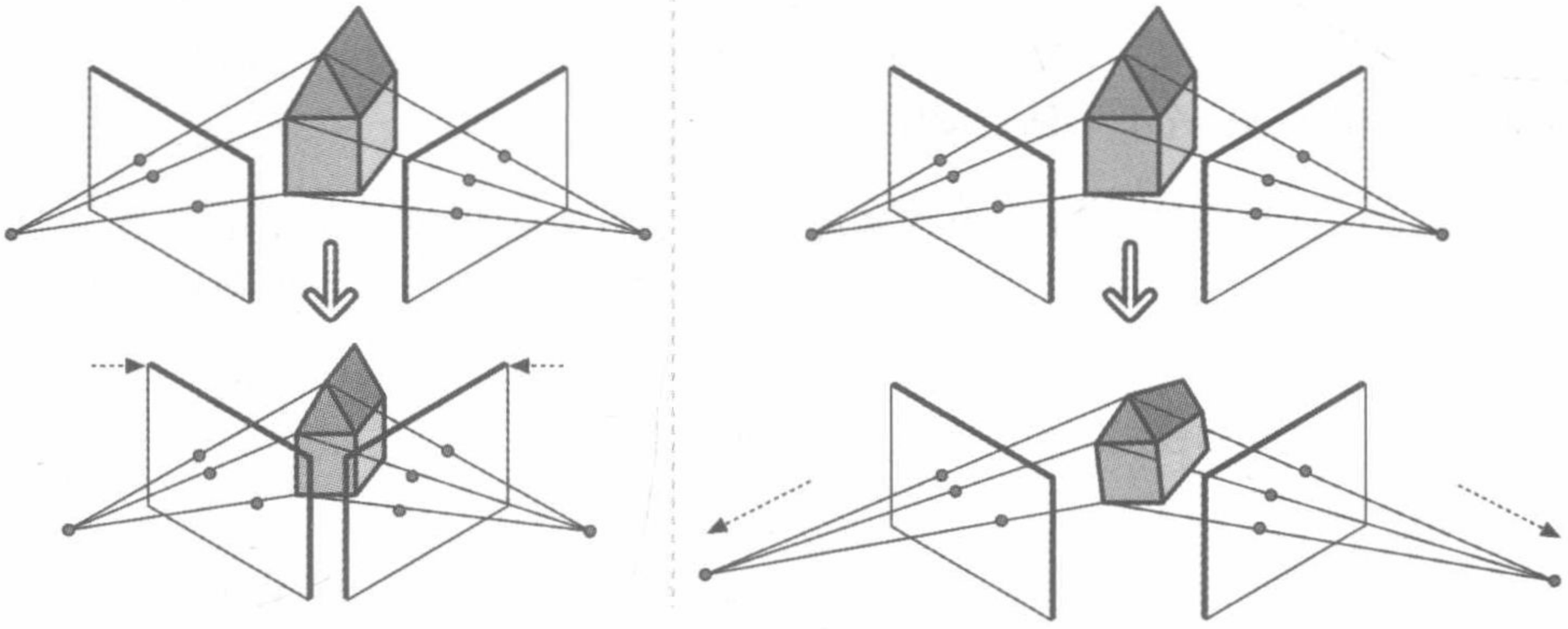


图19-11. 立体重构的非唯一性：如果物体大小未知，那么不同大小物体可以看起来相同，这取决于它们与摄像机之间的距离（左图）；如果摄像机内参数未知，那么不同投影可以看起来相同——例如，在焦距和主点不同的情况下（右图）

假设已知基本矩阵F（要求7个或以上各点进行计算），Hartley算法的过程如下（更多细节，参考Hartley的原作[Hartley98]）。

1. 使用基本矩阵，通过关系$F \cdot \vec{e}_l = 0$和$\vec{e}_r^T \cdot F = 0$分别计算左右两个极点。
2. 先求第一个单应矩阵H_r，它将右极点映射到无穷远$(1, 0, 0)^T$处的二维齐次点。由于一个单应矩阵有7个约束（尺寸未知），使用其中三个用来进行无穷远处的映射，剩下的四个自由度进行选择矩阵H_r。这四个自由度大都容易导致混乱，因为H_r的大多数选择将导致高度扭曲的图像。为了找到一个较好的H_r，我们在图像中选择一点产生扭曲的可能性最小，仅允许刚性旋转和平移（并不是剪切）。对这一点的一个合理的选择是图像原点，同时我们进一步假设极点$\vec{e}_r = (k, 0, 1)^T$位于X轴上（这可以由一个旋转矩阵实现，之后将进行解释）。给定这些坐标，矩阵如下：

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/k & 0 & 1 \end{pmatrix}$$

将使这样一个极点映射到无穷远处。

3. 对于在右侧图像中选择的一个感兴趣点（我们选择原点），通过计算平移矩阵$\vec{T}$将把该点映射到左侧图像原点（在本例中为0），同时旋转矩阵将极点映射搭配$\vec{e}_r = (k, 0, 1)^T$。所求的单应矩阵将由$H_r = G \cdot R \cdot T$得到（在这里，T是一个由$\vec{T}$得到的4×4的平移矩阵）。

4. 接下来我们搜寻一个匹配的单应矩阵H_l，它将左极点发送到无穷远，并使两幅图像行对齐。通过步骤2中的三个约束我们可以容易地将左极点发送到无穷远处。为了使行对准，我们依靠事实行对准使两幅图像的所有匹配点间距离和最小。这也就是说，我们搜寻H_l使左右匹配点$\sum_i d(H_l \cdot \vec{p}_{li}, H_r \cdot \vec{p}_{ri})$总的视差最小化。这两个单应矩阵定义了立体校正。

尽管这个算法的细节有一些棘手，但函数cv::stereoRectifyUncalibrated()可以处理所有复杂的工作。这个函数的名称有些不恰当，因为它没有标定非标定立体图像；反之，它计算了可以用于标定的单应矩阵。算法调用如下：

```
bool cv::stereoRectifyUncalibrated(
  cv::InputArray  points1,         // Feature points from image 1
  cv::InputArray  points2,         // Corresponding points in image 2
  cv::InputArray  F,               // Fundamental matrix
  cv::Size        imgSize,         // Size of images used
  cv::OutputArray H1,              // Rectification (homography) matrix
                                   // for image 1
  cv::OutputArray H2,              // Rectification (homography) matrix
                                   // for image 2
  double          threshold = 5.0 // (Optional) outlier
                                   // threshold ('0'=ignore)
);
```

在函数cv::stereoRectifyUncalibrated()中，算法将两组二维特征点作为输入，它们是左右图像上的对应点数组points1和points2。刚计算得到的基本矩阵以数组F(F)传入。我们对imagesize十分熟悉，这在标定过程中用于描述图像的宽度和高度。我们的结果校正单应矩阵，由函数变量H1(H_l)和H2(H_r)返回。最终，如果点到其对应极线之间的距离超过设定阈值，对应点由算法消除。[注38]

如果摄像机有大致相同的参数并设置在大致水平对齐的前向平行配置中，那么我们通过Harrley算法得到的最终校正输出将看起来非常近似于接下来讨论的校正示例。如果大小或者场景中图像的三维几何已知，我们就可以获得与标定示例中相同的结果。

注38：当图像已经被单摄像机标定标定过，Hartley算法运用效果最好。当存在高度畸变时，算法将无法运行。非常讽刺的是，我们的“无校准”例程仅适用于未畸变的图像输入，其参数通常来自之前的校准。对于另一种未校准的三维方法，请参阅Pollefeys [Pollefeys99a]。

已标定的立体校正：Bouguet算法

给定两幅立体图像的旋转和平移（R，$\vec{T}$），立体校正的Bouguet算法旨在使两幅图像中每幅图像的重投影变化最小化（由此使重投影畸变的结果最小化）同时最大化视图的公共视野。

为了使图像重投影畸变最小化，将右侧摄像机的图像平面旋转岛左侧摄像机的旋转平面的旋转矩阵R在两个摄像机之间被分为两半；我们分别将左右摄像机的合成旋转矩阵为r_r和r_l。每个摄像机旋转一半，则它们的主光线最终平行于原始主光线所指的矢量和。我们应该注意到，这样的旋转将摄像机设置为共面对准而非行对准。为了计算将左摄像机极点变换到无穷远处并使极线水平对准的矩阵R_{rect}，我们构造一个由极点$\vec{e}_1$方向开始的旋转矩阵。将主点（c_x, c_y）作为左侧图像的原点，极点（经过归一化的）方向就是两台摄像机投影中心之间的平移矢量方向：

$$\vec{e}_1 = \frac{\vec{T}}{\| \vec{T} \|}$$

下一个矢量$\vec{e}_2$必须和$\vec{e}_1$正交，否则不受约束。对于$\vec{e}_2$，选择一个正交于主光线的方向（趋于沿像平面）是一个较好的选择。通过使用$\vec{e}_1$与主光线方向的交叉乘积来实现，然后进行归一化，以便获得另一个单位矢量：

$$\vec{e}_2 = \frac{1}{\sqrt{T_x^2 + T_y^2}}(-T_y, T_x, 0)^T$$

始终存在第三个矢量$\vec{e}_3$，通过交叉相乘与$\vec{e}_1$和$\vec{e}_2$正交：

$$\vec{e}_3 = \vec{e}_1 \times \vec{e}_2.$$

将左侧摄像机中极点转换到无穷远处的矩阵即为：

$$R_{rect} = \begin{bmatrix} \vec{e}_1^T \\ \vec{e}_2^T \\ \vec{e}_3^T \end{bmatrix}$$

这个矩阵将左侧摄像机围绕投影中心旋转，使得极线水平且极点位于无穷远处。两个摄像机可以通过如下设置实现行对准：

$$R_l = R_{rect} \cdot r_l$$

和

$$R_r = R_{rect} \cdot r_r$$

我们还将计算标定的左右摄像机矩阵$M_{rect,l}$和$M_{rect,r}$，但将它们返回与投影P_l'和P_r'结合：

$$P_l = M_{rect,l} \cdot P_l' = \begin{bmatrix} f_{rect,l} & \alpha_l & c_{x,l} \\ 0 & f_{y,l} & c_{y,l} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

和

$$P_r = M_{rect,r} \cdot P_r' = \begin{bmatrix} f_{rect,r} & \alpha_r & c_{x,r} \\ 0 & f_{y,r} & c_{y,r} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

（像我们在第18章看到的一样，这里的α_l和α_r允许在现代摄像机，像素偏移因子几乎总是为0。）投影矩阵通过以下步骤将单应坐标系中的一个三维点转换到一个二维单应坐标系：

$$P \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

屏幕坐标可以根据$(x, y) = (x/w, y/w)$计算。给定屏幕坐标和摄像机内参数矩阵二维中的点同样可以重新投影到三维中。重新投影矩阵为：

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -\frac{1}{T_x} & \frac{c_x - c_x'}{T_x} \end{bmatrix}$$

这里的参数来自左侧视图，除了c_x'，它是右侧视图中x坐标轴主点。如果主光线相交于无穷远处，那么$c_x = c_x'$，且右下角的项为0。给定一个二维均匀点和其相关偏差d，可以将点投影到三维中通过：

$$Q \cdot \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

三维坐标即为$(X/W, Y/W, Z/W)$。

应用刚才描述的Bouguet标定方法可产生理想的立体配置，如图19-5所示。之后为旋转的图像选择新的图像中心和新的图像边界，以便最大化重叠的视图区域。这主要是将两个图像区域的均匀摄像机中心和共同的最大高度和宽度设置为新的立体观察平面。

在OpenCV库的文本中，Bouguet算法由函数cv::stereoRectify()实现。给定两个摄像机的内参数和畸变系数，以及关联两个摄像机位置的平移和旋转矩阵，cv::stereoRectify()函数将计算我们所需的校正、投影和视差图，从摄像机对的立体图像中提取深度信息。

```
void cv::stereoRectify(
  cv::InputArray  cameraMatrix1, // Intrinsics (cam 1)
  cv::InputArray  distCoeffs1,   // Distortion coefficients (cam 1)
  cv::InputArray  cameraMatrix2, // Intrinsics (cam 2)
  cv::InputArray  distCoeffs2,   // Distortion coefficients (cam 2)
  cv::Size        imageSize,     // Size of imgs used for calibration
  cv::InputArray  R,             // Rotation *matrix* between
                                 // camera coordinates
  cv::InputArray  T,             // Translation vector between
                                 // camera coordinates
  cv::OutputArray R1,            // 3x3 Rectification xform (cam 1)
  cv::OutputArray R2,            // 3x3 Rectification xform (cam 2)
  cv::OutputArray P1,            // 3x4 (New) projection mtx (cam 1)
  cv::OutputArray P2,            // 3x4 (New) projection mtx (cam 2)
  cv::OutputArray Q,             // 4x4 Disparity to depth
                                 // mapping matrix
  int             flags        = cv::CALIB_ZERO_DISPARITY,
  double          alpha        = -1,         // [0,1] output crop
  cv::Size        newImageSize = cv::Size(), // Output mig size (use
                                 // '0,0' = "as input")
  cv::Rect*       validPixROI1 = 0,          // (Optional) guaranteed
                                 // valid pix (img1)
  cv::Rect*       validPixROI2 = 0           // (Optional) guaranteed
                                 // valid pix (img2)
);
```

对于函数cv::stereoRectify()，[注39]我们首先输入由cv::stereoCalibrate()函数返回的原始摄像机矩阵和畸变向量。其后是imageSize，用于执行标定的棋盘图像的大小。同

注39：cv::stereoRectify()的名称有一些不正确，因为该函数计算了我们可以用于校正的项，但实际上并不能校正立体图像。

时传入由cv::stereoCalibrate()返回的左右摄像机之间的旋转矩阵R和平移向量T。

返回参数是R1（P_l）和R2P_r），是从前述公式推导得到的左右摄像机平面间的行对准的3×3标定旋转矩阵。同样的，我们得到3×4的左右投影方程P1（P_l）和P2（P_r）。一个可选的返回参数是Q，为之前描述的4×4的重新投影矩阵。

标志参数默认设置视差为无穷大，正常情况如图19-5所示。未设置标志意味着我们想要摄像机相互重合（比如轻微的“cross-eyed”）以使在有限距离的视差为0（这可能在特定距离附近的更高深度分辨率的情况下需要）。

如果flags参数未设置为cv::CALIB_ZERO_DISPARITY，那么在实现校正系统时必须更小心。回忆一下，我们是相对于左右摄像机的左右点（c_x, c_y）校正我们的系统。所以对图19-5的测量必须和这些位置相关。基本上，我们必须修正距离，以使$\tilde{x}_l = x_r - c_x^{right}$同时$\tilde{x}_l = x_r - c_x^{left}$。当视差被设置为无穷远，我们可以得到$c_x^{right} = c_x^{left}$（即当cv::CALIB_ZERO_DISPARITY被传递到cv::stereoRectify()），同时可以将普通像素坐标（或视差）传递给深度公式。但是如果cv::stereoRectify()没有调用cv::CALIB_ZERO_DISPARITY，那么通常$c_x^{right} \neq c_x^{left}$。因此，即使公式$Z = f \cdot T_x / (x_l - x_r)$不变，要切记，$x_l$和$x_r$不是针对图像中心，而是分别对应主点$c_x^{left}$和$c_x^{right}$，这与$x_l$和$x_r$不同。所以，如果计算视差$d = x_l - x_r$，应该在计算$Z = f \cdot T_x/(d - c_x^{left} - c_x^{right})$之前进行调整。

校正映射

一旦有了立体校正项，我们可以使用对cv::initUndistortRectifyMap()函数的独立调用来预计算左右摄像机视图的左右校正查找图。与任何图像到图像映射功能一样，前向映射（其中我们仅计算像素从源图像到目标图像的位置）将不会由于浮点目标位置而命中所有目标图像对应的像素位置——目标图像将看起来像瑞士奶酪。所以我们反之采用后向工作方式：对于目标图像中每一个整体像素位置，我们查找源图像上的浮点位置，之后利用周围源像素的整型值插值出新的值。这种查找一般使用双线性插值，在第11章的cv::remap()已经遇到。

标定过程如图19-12所示。如该图中公式流程所示，实际标定过程从(c)到(a)向后延伸，被称为逆向映射。对于标定图像(c)上每一个整型像素，我们在未畸变的图像(b)中找到其坐标，并使用它们来查找原始图像(a)中实际（浮点）坐标。浮点坐标上的像素值通过在原始图像上的邻近整型像素位置插值得到，这个值被赋给目的图像(c)上的标定后整型像素位置。在标定图像都被赋值之后，我们经常对它进行剪切，以增大左右图像间的叠加面积。

实现图19-12所示的数学描述函数为cv::initUndistortRectifyMap()。这个函数被调用两次，一次为立体对的左侧图像左，另一次为立体对的右侧图像。

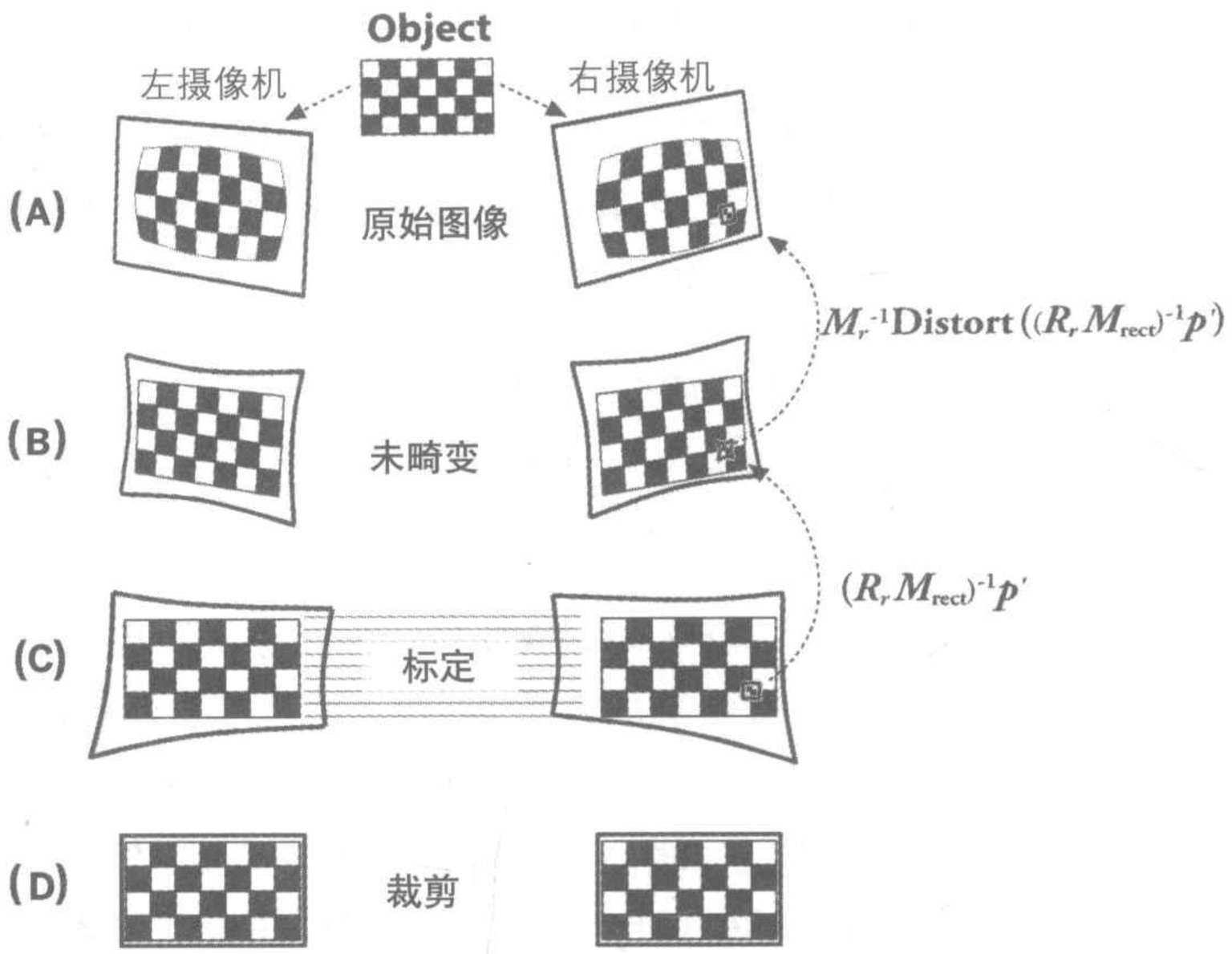


图19-12. 立体校正：对于左右摄像机，原始图像(A)未畸变(B)并且已校正(C)，最终裁剪(D)，以集中在两个摄像机之间的重叠区域上；校正计算实际上从(C)到(A)逆向进行

```
void cv::initUndistortRectifyMap(
  cv::InputArray  cameraMatrix,     // Camera intrinsics matrix
  cv::InputArray  distCoeffs,       // Camera distortion coefficients
  cv::InputArray  R,                // (Optional) 3x3 rectification
                                    // transformation
  cv::InputArray  newCameraMatrix,  // New camera matrix usually
                                    // from cv::stereoRectify()
  cv::Size        size,             // Undistorted image size
  int             m1type,           // Method for encoding result maps
  cv::OutputArray map1,             // First output undistortion map
  cv::OutputArray map2              // Second output undistortion map
);
```

cv::initUndistortRectifyMap()函数以3*3的摄像机矩阵cameraMatrix，5×1的摄像机畸变参数distCoeffs，3×3的旋转矩阵R，和经过标定的3×3的摄像机矩阵newCameraMatrix作为输入。

如果使用cv::stereoRectify()标定立体摄像机，则我们可以直接从cv::stereoRectify()中读取并输入cv::initUndistortRectifyMap()，先使用左参数，以校正左侧摄像机，之后利用右参数校正右摄像机。对于R，使用cv::stereoRectify()函数中的R_l和R_r；对于cameraMatrix，使用cameraMatrix1或者cameraMatrix2。对于newCameraMatrix，我们可以使用cv::stereoRectify()中3×4的P_l或P_r的前三列，但是为了方便，该函数允许我们直接传递P_l或P_r，并从它们读取newCameraMatrix。

另一方面，如果我们使用cv::stereoRectifyUncalibrated()来标定立体摄像机，那么我们必须对单应矩阵进行适当的预处理。尽管我们可以——在原则上和在实际中——不使用摄像机内参数进行立体校正，OpenCV没有可以直接进行这种操作的函数。如果在之前的标定中没有得到newCameraMatrix，则正确的步骤是将newCameraMatrix设置为与CameraMatrix相等。之后，对于cv::initUndistortRectifyMap()中的R，需要分别对左右标定计算$R_{rect,l} = M^{-1}_{rect,l} \cdot H_l \cdot M_l$（或者仅是$R_{rect,l} = M_l^{-1} \cdot H_l \cdot M_l$，如果$M^{-1}_{rect,}$不可用）。最后，我们还需要每个摄像机的畸变系数，以填入5×1的参数distCoeffs。

参数size指明了图像无畸变的大小（因此可产生映射数组的大小）。参数m1type决定映射格式的产生。如在第18章中所述，它可以设置为CV_32C1或CV_16SC2，并且分别生成浮点型或整型的映射。

函数cv::initUndistortRectifyMap()返回查找映射表map1和map2作为输出。这些映射表说明该从什么地方为每个目标图像插值源像素；映射表之后可以直接插入cv::remap()中，这个函数我们在第11章首次见过。正如之前提到的，函数cv::initUndistortRectifyMap()分别被左右摄像机调用，使得我们可以获得它们不同的重映射表map1和map2。然后可以调用函数cv::remap()，先使用左侧之后右侧的映射，每次获得新的左右立体图像进行校正。图19-13为立体无畸变和立体对图像标定的结果。注意特征点是如何水平对准在无畸变标定图像上的。

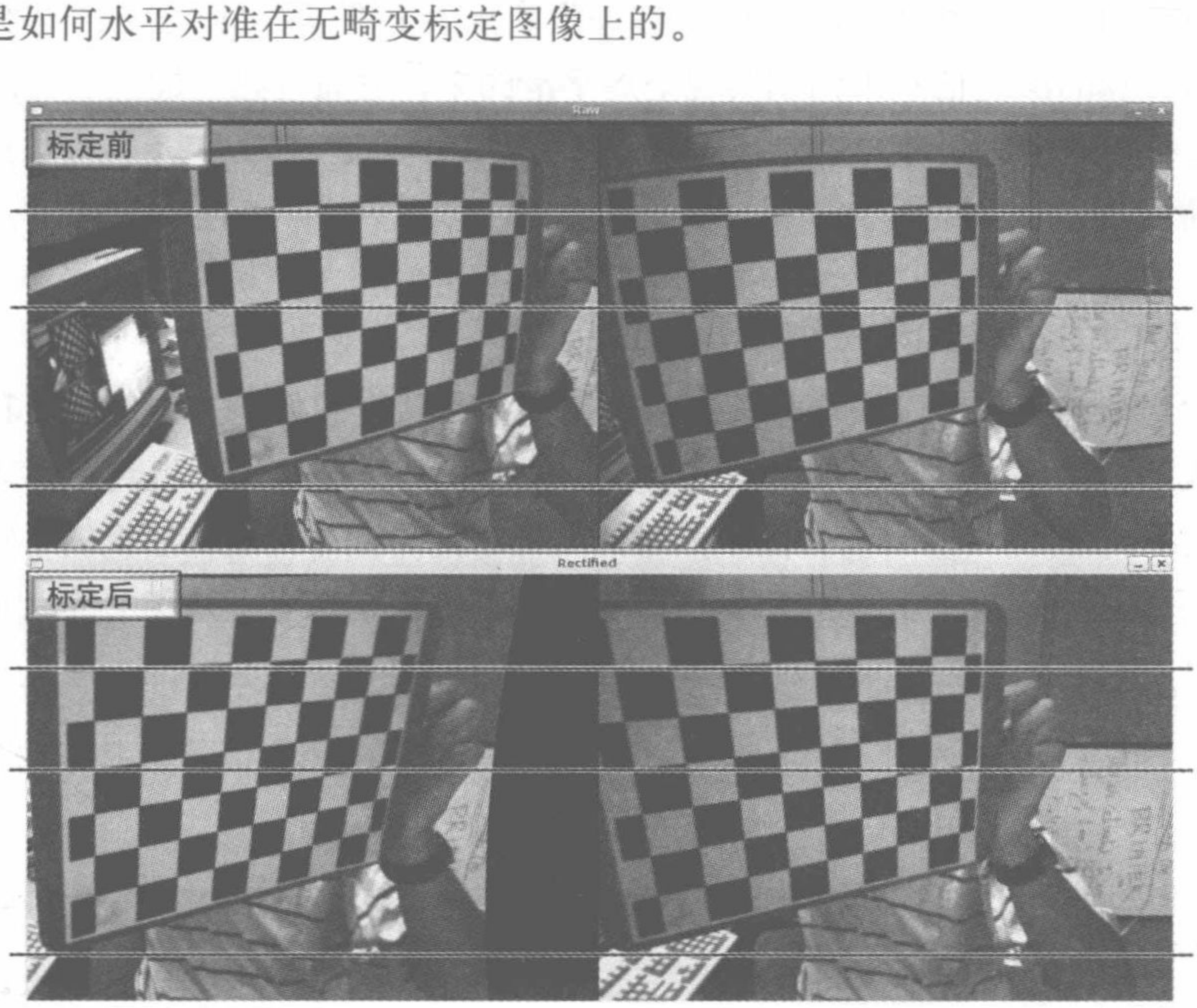


图19-13. 立体校正：原始左右图像对（上图）和经过立体校正的左右图像对（下图）。注意，桶形畸变（在棋盘图案的顶部）已经被纠正并且扫描线已在标定图像中对齐

立体匹配

立体匹配——在两个不同摄像机视图中匹配一个三维点——只能在两个摄像机视图重叠的可视区域上进行计算。再重复一下，这是为什么将摄像机尽可能前向平行放置（除非你成为立体视觉方面的专家）将获得更好结果的原因。一旦知道摄像机的物理坐标或者场景中物体的大小，我们可以通过两个不同摄像机视图上对应点间的三角测量视差值$d = x^l - x^r$（或$d = x^l - x^r - (c_x^{left} - c_x^{right})$，如果主光线相交于无穷远处）获得深度测量。如果没有这些物理信息，我们就只能计算深度的比例关系。如果没有摄像机内参数，当时用Hartley算法时，只能计算投影变换意义上的点的位置（图19-11）。

OpenCV实现了两种立体匹配算法，它们共享（一个几乎）通用的对象接口。首先，是块匹配算法（block matching，BM），这是一个快速有效的算法，与Kurt Konolige [Konolige97]提出的一个算法类似。它使用小的“绝对值之差的和”（SAD）窗口来寻找左右立体校正图像之间的匹配点。[注40]这个算法只搜寻两个图像之间的强匹配（强纹理）。因此，在一个强纹理场景中，例如出现在室外的森林中，每一个像素都有可计算的深度。在一个弱纹理场景中，例如一个室内走廊，只能算出很少点的深度。第二个算法是半全局块匹配算法（semi-global block matching，SGBM）。SGBM是在[Hirschmuller08]中引入的SGM的变形形式，主要在两个方面与BM算法不同。第一个是使用Birchfield-Tomasi度量[Birchfield99]在子像素水平上进行匹配。第二个不同是SGBM试图基于所计算的深度信息来强制实现全局平滑约束，它通过感兴趣区域的许多一维平滑度约束进行考虑。这两种方法是互补的，看上去BM稍快一些，但不能提供SGBM的可靠性和精确性。

立体匹配类cv::StereoBM和cv::StereoSGBM

不同的立体匹配算法都可以实现相同的基础功能：将两个图像，一个左图像和一个右图像，转换成单个深度图像。深度图像将与每个像素相关联，从摄像机到该像素点所代表对象之间的距离。[注41]OpenCV为刚刚描述的两种不同的立体匹配算法提供了实现：块匹配（BM）和半全局块匹配（SGBM）。尽管这两个算法在历史上是相关的（正如它们名称所示，它们的主要功能相同），它们都有各自的接口。[注42]

注40：这个算法在Videre的FPGA立体硬件系统可以得到。

注41：通常在OpenCV中，深度图像中的像素对应于左侧摄像机图像的像素，只不过这种惯例不是很常用。

注42：因为有这样的共同点，你可能会期望一些虚拟基类应用于定义实现这两种方法的对象的公共接口，就像在库中其他地方一样。但是实际上，用于初始化各个算法的信息存在少许不同，所以实际的实现方法没有使用公共接口（至少不是暂时的），而是使用两个单独的但非常相似的类。

块匹配

OpenCV中实现的块匹配立体算法被认作对立体计算的（也许是）规范技术的稍加改进的版本。基本机制是标定并且对准图像，使得只在各行进行比较，并且之后在两幅图像上进行算法查找来进行像素组的匹配。当然，存在一些额外的细节使算法可以运行更好或运行更快。最终结果是一个快速、相对可靠的算法，同时在广泛应用中被大量使用。

块匹配立体匹配算法有三个步骤，工作于无畸变、标定立体图像对。

1. 预过滤以使图像亮度归一化，并增强纹理。
2. 使用SAD窗口，沿着水平极线进行匹配查找。
3. 后置滤波以消除不良对应匹配。

在预过滤一步，将输入图像归一化以减少亮度差异并增强图像纹理。我们通过在图像上运行大小为5×5、7×7（默认）、…、21×21（最大）的窗口来实现。窗口下的中心像素I_c由$min(max(I_c-\bar{I}, -I_{cap}), I_{cap})$代替，其中$\bar{I}$是窗口中的平均值，$I_{cap}$是一个正数范围，其默认值为30。替代方案是将输入的图像基本上转换为它们的x-Sobel导数。这也具有去除与亮度相关工件的作用。

之后通过滑动SAD窗口计算匹配。对于左图像中的每一个特征，我们在右图像中查找对应行以实现最优匹配。校正之后，每一行都是一条极线，因此右图像中的匹配位置一定沿着左图像中的相同行（同一y坐标）；如果特征具有足够强度的纹理可以识别并且在右摄像机视图中未被遮挡（图19-17），匹配位置可以找到。如果左侧特征像素坐标位于(x_0, y_0)，那么对于水平前向平行的摄像机排列，匹配（如果有）一定会在相同行或x_0左侧找到（图19-14）。对于前向平行摄像机，x_0为零视差，并且左侧视差更大。对于彼此之间成角度的摄像机，匹配可能发生在负视差（位于x_0右侧）。算法需要被告知产生最小视差的位置。

然后通过预先选择的视差数进行视差搜索，以像素计数（默认为64像素）。视差具有一个离散的亚像素精度，等于低于单个像素级别的4位分辨率。当输出图像是一个32位浮点图像时，将返回非整型视差。当输出图像是一个16位整型时，视差将以4位定点形式返回（即乘以16并舍入为整数）。

设置最小视差和待查找的视差个数建立一个双目视界，这个三维体被立体算法的搜寻范围所覆盖。图19-14展现了起始于三个不同视差限制的五个像素的视差搜索极限：20，17和16。每个视差极限定义了一个距摄像机固定深度的平面（图19-15）。如图19-14所示，每一个视差极限——与视差个数相结合——设置了可以检测深度的不同的双目视界。在范围之外，不能获得深度，并且在深度图上表现为一个“洞”，其中深度是未知

的。可以通过减少摄像机之间的基线距离$\| \vec{T} \|$，减小焦距，增加立体视差查找范围，或者增加像素宽度来扩大双目视界。

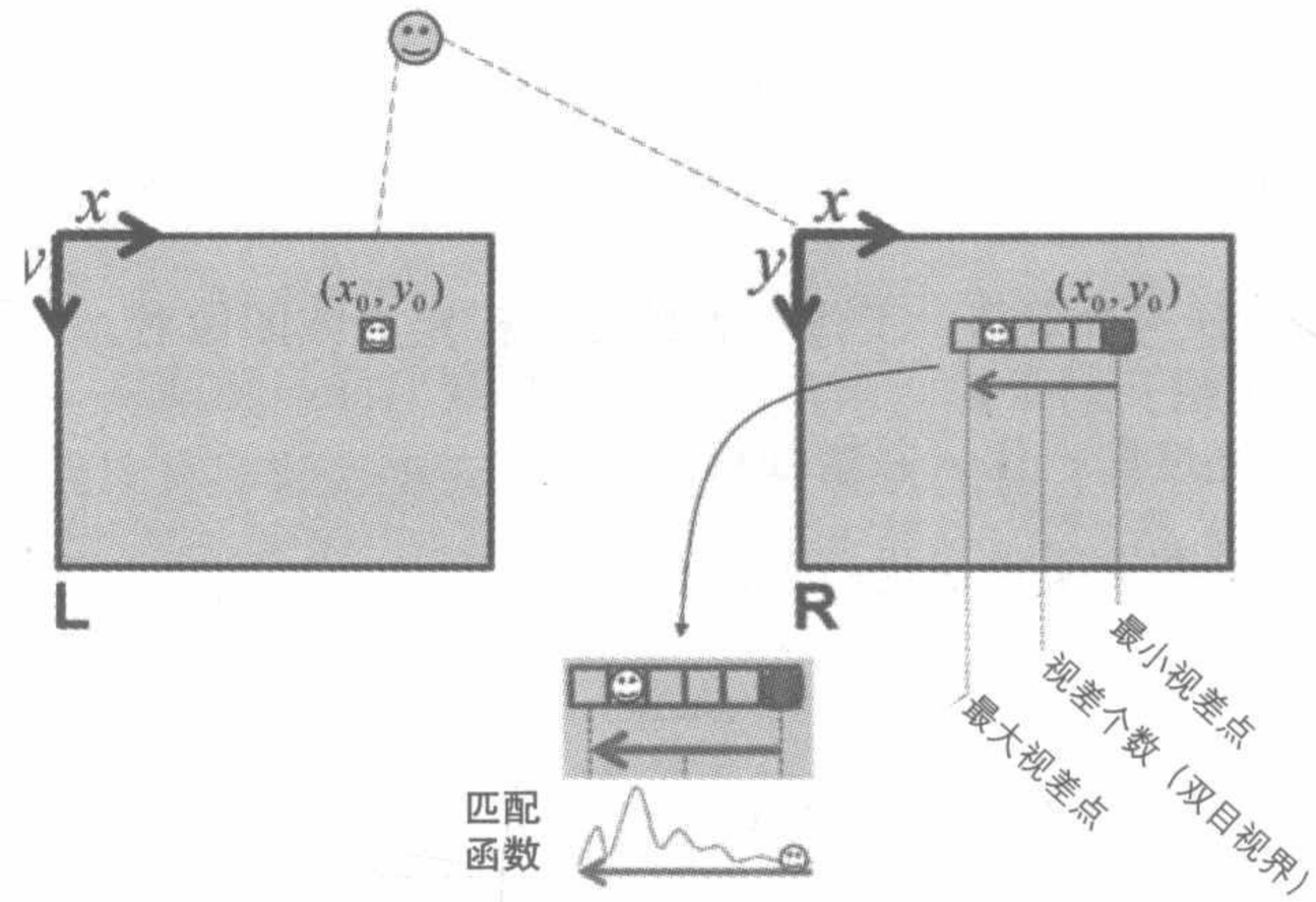


图19-14. 任何一个左侧图像特征的右侧图像匹配必须产生于相同行，并且开始于（或者位于其左侧）相同的坐标点，其中匹配查找起始于最小视差点（在这里是0），向左移动视差个数的位置；基于窗口特征匹配的特性匹配函数如图下半部分所示

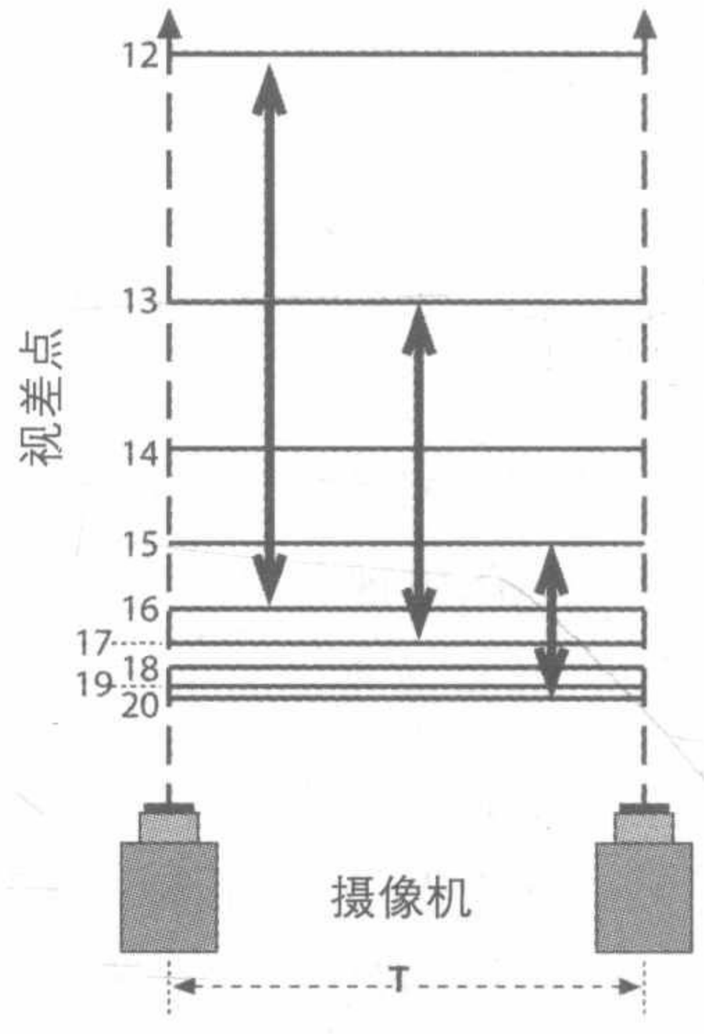


图19-15. 每条直线表示整型像素从20至12变换时的恒等视差平面；5个像素内的视差搜索范围将覆盖不同的双眼视界范围，如垂直箭头所示，并且不同的最大视差产生不同的双眼视界

双目视界内部的对应有一个内置约束，称为“顺序约束”，它简单表示特征的顺序不能从左侧视图变化到右侧。可能存在缺失的特征——这归因于遮挡和噪声，在左侧查找到的一些特征不能在右侧找到——但是这些搜索到的特征的顺序保持相同。相同的，许多右侧的特征与左侧的不同（这称为“插入”），尽管它们可能会扩散这些特征，但是插

入不影响特征的顺序。图19-16中所示的过程反映了顺序限制，其中特征被匹配在一条水平扫描线上。

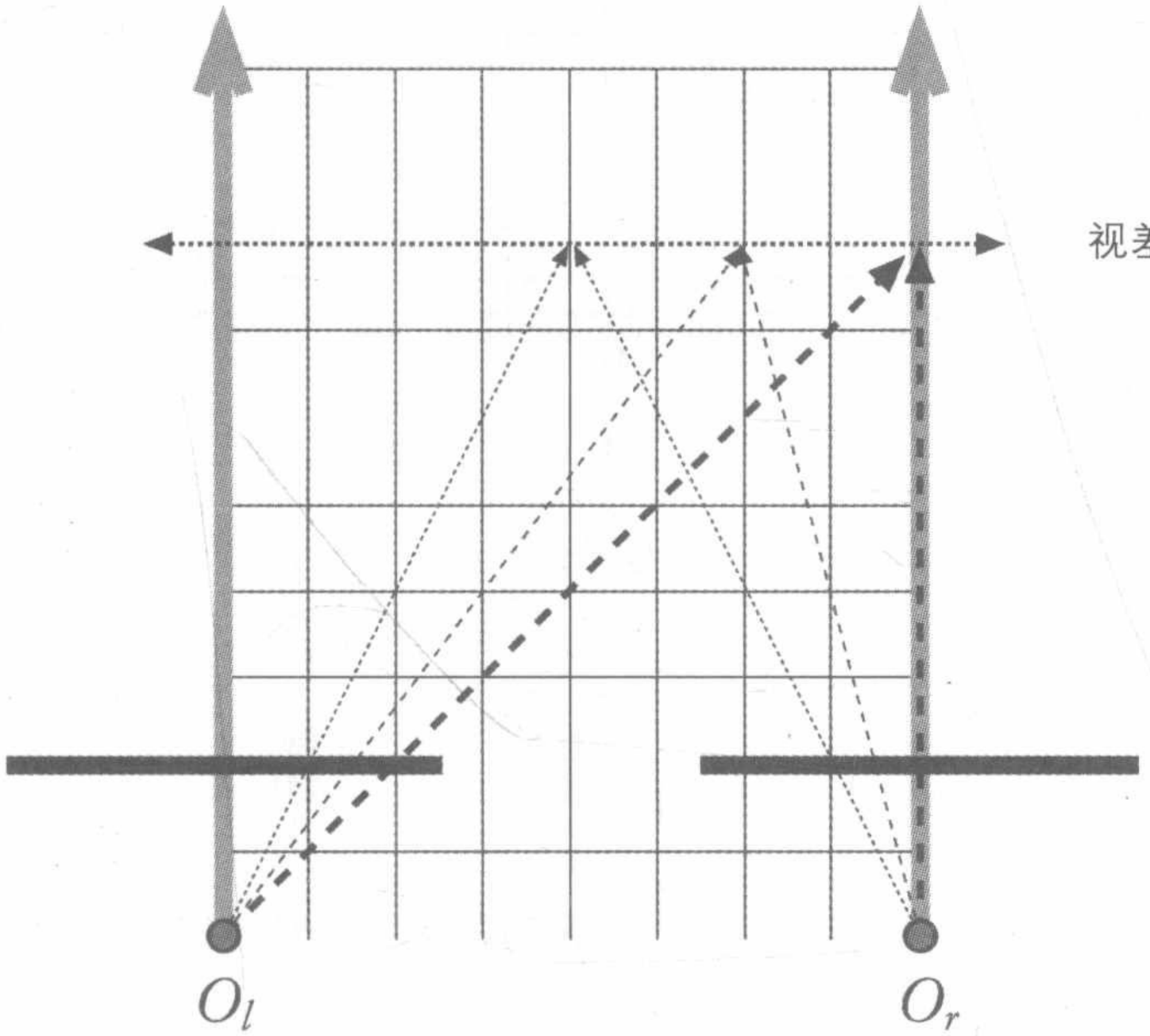


图19-16. 固定的视差形成与摄像机之间固定距离的平面

已知最小允许视差增量Δ_d，可以使用以下公式确定最小可实现的深度范围分辨率Δ_z：

$$\Delta_z = \frac{z^2}{f \cdot T_x}\Delta_d$$

记住这个公式是很有用的，因为你可以知道从立体试验台中能期待获得哪种类型的深度精度。

匹配后，我们转向后过滤处理。图19-13的下半部分展现了典型的匹配功能响应，因为特征值从最小视差到最大视差被“扫描”。注意，匹配通常具有强烈的中心峰被副瓣所包围的特征。一旦获得两个视图之间的待选特征匹配，我们可以使用后过滤处理防止错误匹配。OpenCV通过uniqueness ratio参数的概念来使用匹配功能模式。该比例基本上实现了当前像素的匹配值大于通过某种余量观察得到的最小匹配值的要求。

为了确保有足够的纹理来克服匹配过程中的随机噪声，OpenCV同时还会尝试使用纹理阈值。这只是对SAD窗口相应的一个约束，以确保响应值低于某个最小值的匹配不予考虑。

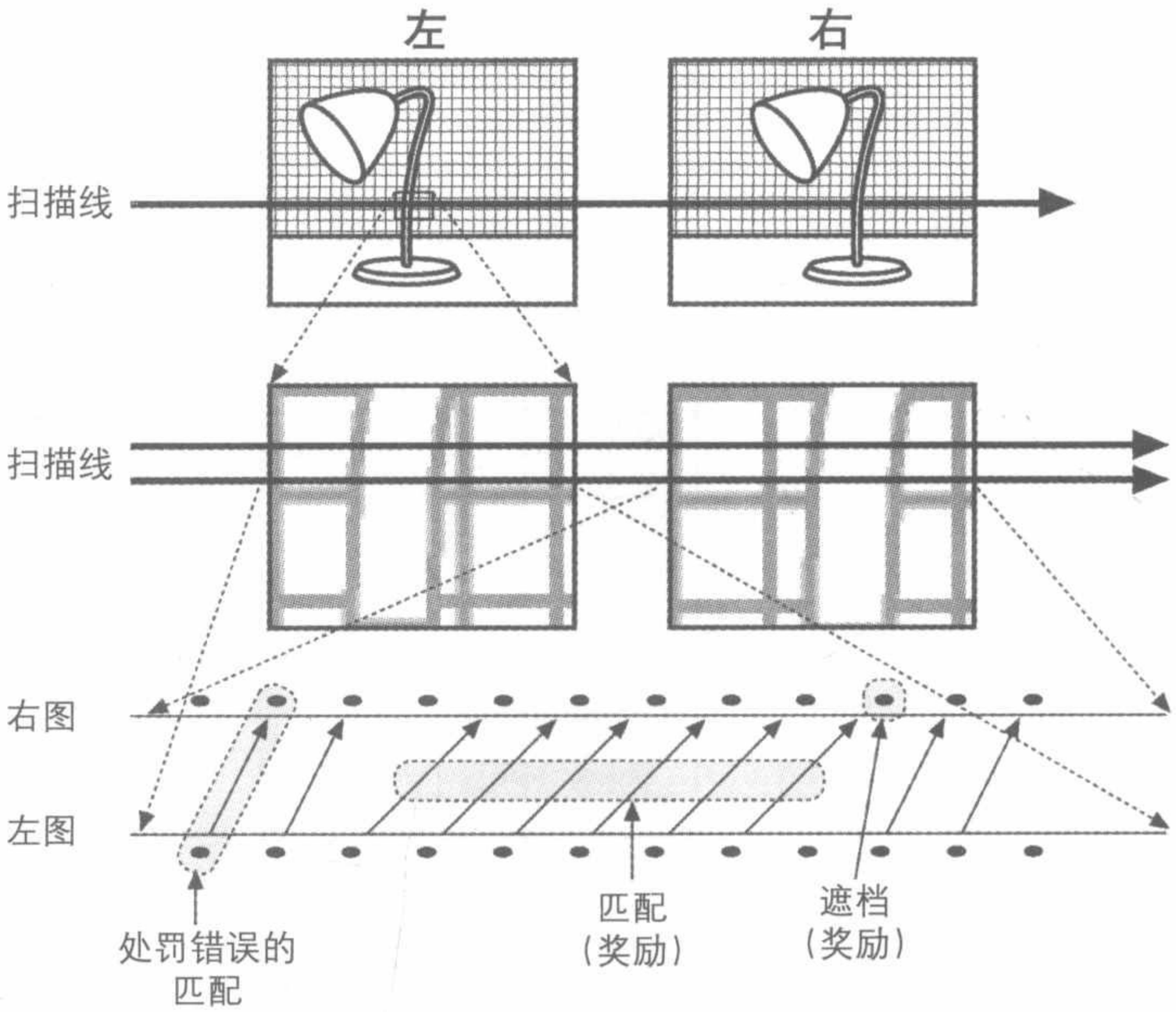


图19-17. 立体匹配从左右图像中相应行之间分配点的匹配开始：灯的左右图像（上图）；单个扫描线的扩大（中图）；对应分配的可视化（下图）

最后，基于块的匹配可能在物体边缘产生问题，因为匹配窗口捕获一侧的前景和另一侧的背景。这种导致同时产生大小视差的局部区域，我们称之为“散斑”。为了防止这种边界匹配，我们在散斑窗口上设置散斑检测器。这种方式的作用是将每个像素用作构建由可变范围漫水定义的连通组件的基础。仅当其在当前像素的某个范围内时，可变范围注满填充包括一个邻近像素。一旦连通部分被计算，如果它比散斑窗口小，就会被视为散斑。该范围的大小称为“散斑范围”。这个范围通常是一个较小的量，通常为1或2，只不过4并不罕见。

使用cv::StereoBM计算立体深度

在OpenCV中，块匹配算法被实现为保存所有必需参数的对象，并提供用于计算视差图像的重载compute()的方法。算法用cv::StereoBM类表示，从cv::StereoMatcher类中得到。这些类如何定义如下：

```
class StereoMatcher : public Algorithm {

public:
    // the returned depth map has CV_16UC1 type and the elements are
    // actually fixed-point numbers with DISP_SHIFT=4 fractional bits
    enum {
      DISP_SHIFT = 4,
      DISP_SCALE = (1 << DISP_SHIFT)
    };
```

```
// the main method, takes two grayscale 8-bit rectified images and
// outputs 16-bit fixed-point disparity map
virtual void compute(
  InputArray left,
  InputArray right,
  OutputArray disparity
) = 0;

// the minimum disparity, usually 0 (points are at the infinity
virtual int getMinDisparity() const                          = 0;
virtual void setMinDisparity(int minDisparity)               = 0;

// the range of disparities, between minimum (inclusive) and
// maximum (exclusive) disparity
virtual int getNumDisparities() const                        = 0;
virtual void setNumDisparities(int numDisparities)           = 0;

// size of blocks used by algorithm
virtual int getBlockSize() const                             = 0;
virtual void setBlockSize(int blockSize)                     = 0;

// maximum size of a speckle to be considered as speckle
// and marked as such
virtual int getSpeckleWindowSize() const                     = 0;
virtual void setSpeckleWindowSize(int speckleWindowSize) = 0;

// the allowed difference between neighbor pixels;
// used by floodfill-based speckle filtering algorithm
virtual int getSpeckleRange() const                          = 0;
virtual void setSpeckleRange(int speckleRange)               = 0;
...
};

class cv::StereoBM : public cv::StereoMatcher {

    enum {
      PREFILTER_NORMALIZED_RESPONSE = 0,
      PREFILTER_XSOBEL = 1
    };

    // choose between PREFILTER_NORMALIZED_RESPONSE and
    // PREFILTER_XSOBEL
    //
    virtual int getPreFilterType() const                     = 0;
    virtual void setPreFilterType(int preFilterType)         = 0;

    // size of block used in PREFILTER_NORMALIZED_RESPONSE mode
    //
    virtual int getPreFilterSize() const                     = 0;
    virtual void setPreFilterSize(int preFilterSize)         = 0;

    // saturation threshold applied after pre-filtering
    //
    virtual int getPreFilterCap() const                      = 0;
    virtual void setPreFilterCap(int preFilterCap)           = 0;

    // textureness threshold, the blocks with textureness
    // characteristics (sum of abs derivatives) less than the
```

```
    // threshold are marked as areas without defined disparity
    //
    virtual int getTextureThreshold() const                    = 0;
    virtual void setTextureThreshold(int textureThreshold)     = 0;

    // if there is no clear winner in the cost function across the
    // disparity range, the pixel is having no defined disparity.
    // Uniqueness threshold defines what it means to be a
    // "clear winner," the margin in %% between the best and
    // second-best.
    //
    virtual int getUniquenessRatio() const                     = 0;
    virtual void setUniquenessRatio(int uniquenessRatio)       = 0;

    // the constructor function, the first parameter is made default
    // for techinical reasons, but you should actually specify it
    // here or later using setNumDisparities()
    //
    static Ptr<StereoBM> create(
      int numDisparities = 0,
      int blockSize = 21
    );
};
```

cv::StereoMatcher和cv::StereoBM中最重要的元素是compute()和create()。

静态create()有两个参数，numDisparities和blockSize。

参数numDisparities是可能被返回的不同视差的最大数（事实上，就是指探索深度）。实际上，它设置了法将尝试搜索匹配的范围。

第二个参数是blockSize。这将设置每个像素周围的区域的大小，也会计算“绝对差分符号”度量。这个值越大，找到的错误匹配越少。然而需要注意，算法的计算成本不仅与窗口面积（即窗口大小的平方）相关，还存在可能由隐含假设引起的视差实际上与窗口面积相同的问题。靠近不连续处（对象的边缘），这个假设将不成立，你将有可能找不到匹配。结果将是空白区域，其中在接近对象边缘处不存在视差。随着窗口大小的增加，这些空白区域的厚度也将增加。同时，blockSize越大，获得的深度图将越模糊，意味着视差图中对象的剪影将越平滑，以一种近似的方式捕获实际剪影。

在调用create()并获取一个智能指针到cv::StereoBM对象后，你可以通过调用许多set*进一步配置。对于这种结构中的许多成分，可能需要侧重于调整预过滤参数和后过滤参数。

预过滤尝试从光照或其他在SAD度量下将导致两个图像之间错误匹配的来源中去除变量。preFilterType的可能值是cv::StereoBM::PREFILTER_NORMALIZED_RESPONSE和cv::StereoBM::PREFILTER_XSOBEL。前者只归一化窗口中的强度，后者实际上将图像转换成（剪辑版本）的第一个Sobel导数（在x方向上）。preFilterType的值设置了使用的

滤波器大小（仅用于NORMALIZED_RESPONSE中），并且preFilterCap是用于固定预过滤输出参数I_{cap}的大小（如之前章节中所述）。

后滤波器尝试去除输出图像中的异常值和噪声。textureThreshold设置了在两个区域之间计算视差之前必须存在的纹理最小值。

将uniquenessRatio应用于SAD窗口，并被解释为最佳匹配与第二好匹配之间的差异程度，用于将差异考虑为明确的。阈值定义为$SAD(d) \geq SAD(d^*) \cdot (1.0 + \frac{uniquenessRatio}{100.0})$。其中，$d$和$d^*$分别为当前视差和下一个最佳视差。典型的uniqueness ratio的值在5到15之间。

参数speckleWindowSize和speckleRange共同作用。它们使能一个后过滤，将尝试去除所有与其周围值本质上不同的小的、孤立的散斑。speckleWindowSize设置了这种散斑的大小，同时speckleRange设置将包含在同一个散斑中的视差之间的最大差异。该参数直接与视差值进行比较。这意味着如果使用固定精确度表示形式的的视差，则该值实际上会被乘以16。设置参数时，需要把这一点考虑进去。

一旦配置好cv::StereoBM对象，你可以使用compute()计算视差图像。重载期望获得三个参数：左右图像（left和right）和输出图像 （disparity）。产生的视差为固定点表达形式，有4位的局部精确度，所以在使用这些视差时需要除以16。

半全局块匹配

OpenCV中可用于替换块匹配的是半全局块匹配算法（SGBM），是SGM算法的一种变形形式[Hirschmuller08]。比块匹配算法发展起步晚了近十年，SGM算法应用了许多新的思路，但是计算成本远远大于BM算法。[注43]SGM引入的最重要的新思路是使用互信息（OpenCV的实现使用Birchfield-Tomasi度量，同时用于原始文件作为更简单的选择）作为局部对应的一种优越度量以及沿水平线（极线）以外方向执行一致性约束。在高层次上，这些添加的效果是为了左右图像之间的光照和其他变化提供更大的鲁棒性，并通过在整个图像上实施更强的几何约束以消除误差。

类似BM，SGBM操作的时候无畸变、已校正的立体图像对。SGBM有如下基本步骤。

1. 在StereoBM中使用PREFILTER_XSOBEL模式预处理每个图像。使用Birchfield-Tomasi指标预计算匹配left_image(x, y)和right_image(x-d, y)的每个像素的成本映射$C(x, y, d)$。以0初始化累加3D成本映射$S(x, y, d)$。

注43：在当前的处理器速度下，通常可以以相对较高的分辨率的，即使在只有CPU的实现中，视频的实时处理可以接受的帧速率运行BM。相比之下，SGBM需要多于大约一个数量级的计算时间，因此通常被认为不适合实时视频应用。但是，有SGBM的FPGA实现，可以在甚至非常大的图像中以很高的速度运行。类似的，BM也已经在FPGA中实现，也在GPU中实现——后者已经在OpenCV中得到了支持。

2. 对于每一个3、5、8方向(r)（图19-18），使用迭代过程计算$S^{(r)}(x, y, d)$。对于每一个r's，添加$S^{(r)}(x, y, d)$进入$S(x, y, d)$。为了最优化存储器流程并最小化存储器占用，前三个或前五个方向(W, E, N[, NW, NE])在前向传递中一起处理，并且在八个方向模式的情况下有第二个传递处理剩下的三个方向(S, SW, SE)。在三个或五个方向的算法情况下，对于全部像素，$C(x, y, d)$和$S(x, y, d)$没有明确存储；我们只需要一次存储最后三行或四行的缓冲区。

3. 一旦$S(x, y, d)$是完整的，我们发现$d^*(x, y)$是$S(x, y, d)$的`argmin`。我们使用相同唯一性检验和相同亚像素插值作为StereoBM的算法。

4. 进行左-右检验以保证从左到右和从右到左的对应是连续的。将没有完美匹配的像素标记为“无效视差”。

5. 使用`cv::filterSpecles`过滤散斑，如同SteroBM算法。

算法的关键要素（以及实际上其他任何立体对应算法）指对任何可能的视差如何为每个像素分配成本，这里记作$S(x, y, d)$。本质上，这与我们进行块匹配有一些类似，但是这里有一些新的变化。第一个变化是我们使用了一些更好的亚像素Birchfield-Thomasi指标来比较像素，而非简单的绝对误差。第二个曲折是我们引入了一个非常重要的视差连续性假设（相邻像素可能有相同或相似的视差）且同时使用更小的块（通常是3×3或者5×5）而不是使用更大、完全独立的StereoBM窗口（11×11或者更大）。那些大窗口是BM实际存在的问题，因为要比较两个窗口，必须首先假设在获得正确的视差时它们能够匹配，但这意味着要有一个独立视差来解释两个窗口之间的关系。在接近不连续处（物体的边缘）这条假设不成立，所以在这些不连续处你会得到很多BM中的问题。

在SGBM中，大窗口的替代方案是使用较小的窗口（以补偿噪声）结合路径（图19-18）。这些路径在原则上从图像的边缘延伸到单个像素并且（同样是在原则上）包括从边缘到像素的每一条可能路径。SGBM关注的是沿着路径代价最低的这类路径（类比于BM中成本最低的窗口）。沿着任意一条路径的代价是每个像素的成本（即围绕它的小块）加上当一个像素与其周围在视差上轻微变化产生的微小补偿，或者当一个像素与周围在视差上剧烈变化产生的大的补偿。

要想了解在没有沿着大量路径的不可能求和数情况下，如何计算一些像素的特定视差成本，我们只需要意识到如果我们知道相邻像素的成本（多个视差），唯一可能的情况就是这个新像素的最小成本路径来自附近像素之一。因此我们可以从边缘开始，跨越图像

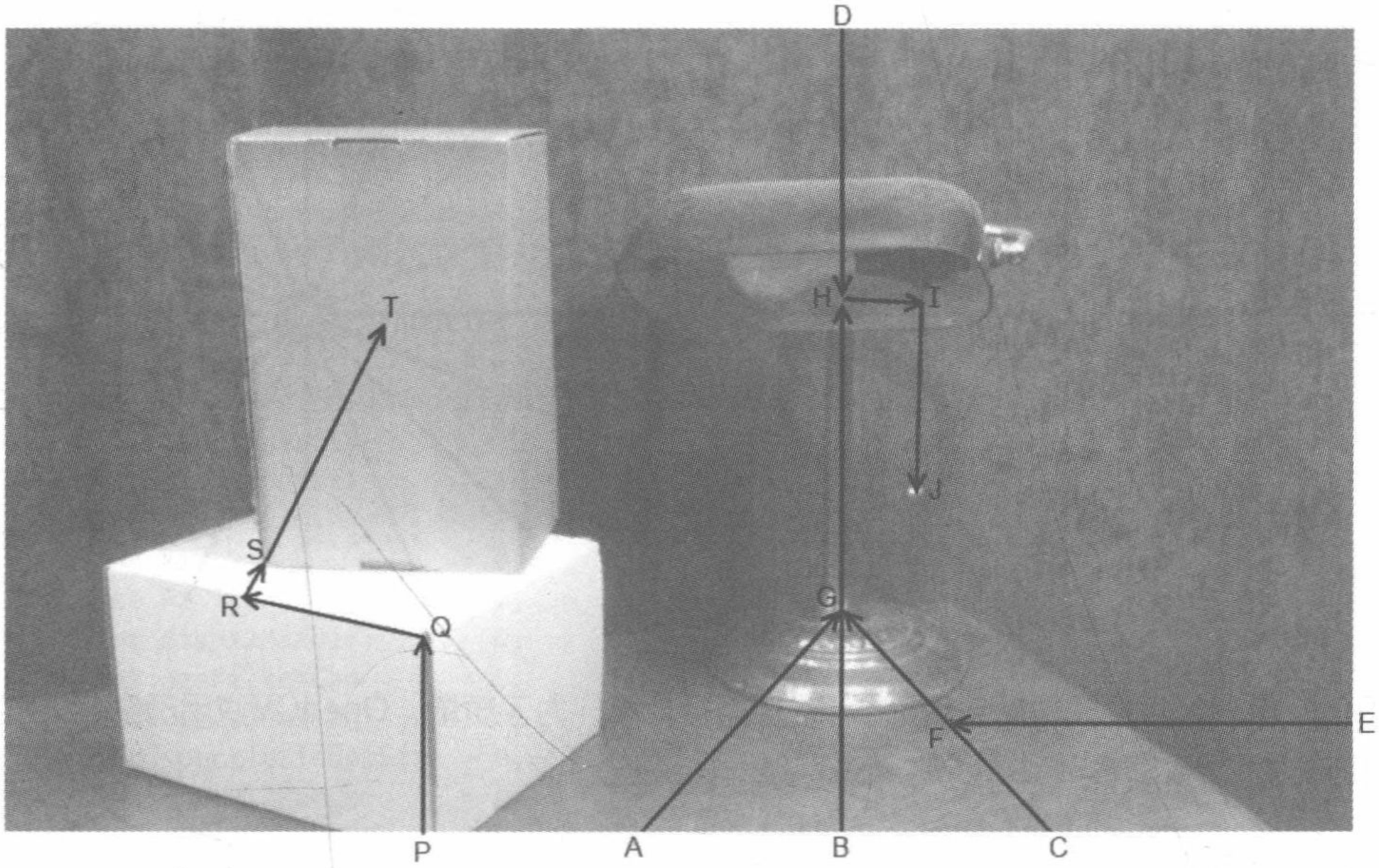


图19-18. 在一个简单的场景中，一些可能的路径可以被可视化为几个点。路径AGHIJ指向台灯链的终点。沿着这条链的视差变化非常缓慢，所以它可能由于其他路径比如EFGHIJ或者DHIJ。为了比较J处的成本，我们需要知道关于I的是BGHI是对于I成本最低的。实际情况下，这些路径中的连接在像素尺度上

进行计算。首先计算所有我们能计算得到的像素之所有可能的视差成本，找到最佳视差，之后移动到它的相邻像素。[注44]

理想情况下，我们会对每个像素的每一种可能路径进行考虑。在原则上，这不局限于仅仅是该像素的8个最邻近像素，而是可以包含一条沿着任意角度直线进入该像素的路线。在实际中，对于每个像素，我们必须限制进行考虑并用于计算的邻近像素数量（如图19-19）。在理论上，最好设置路径数为8或16（如原始SGM文献中所示）以获取高质量的结果，后者在许多实际场景中给出了更好的结果，但是在计算时间方面成本很高。在实际中，可以使用5个，甚至少至3个方向，在大多数情况下结果相似。单纯地，算法的计算成本将随着路径数量线性增长，但是存在容易被忽视的、显著的非线性影响。当从5个方向移动到8个方向是，由于需要为所有像素存储$C(x, y, d)$和$S(x, y, d)$（为了在这些大型数组上进行两次传递），算法可能比预计运行慢很多。

注44：显然，在这里我们掩饰了一个重要的事实，即一个边缘像素可能有来自图像对面的最低成本路径。实际上，如我们描述的，没有一个完全正确的起点。事实上，结果证明可以顺序处理每个方向的整个图像。因此，实际的算法不会逐个像素进行。但是，这个实现方法的具体细节已经远远超出了本书的讨论范围。

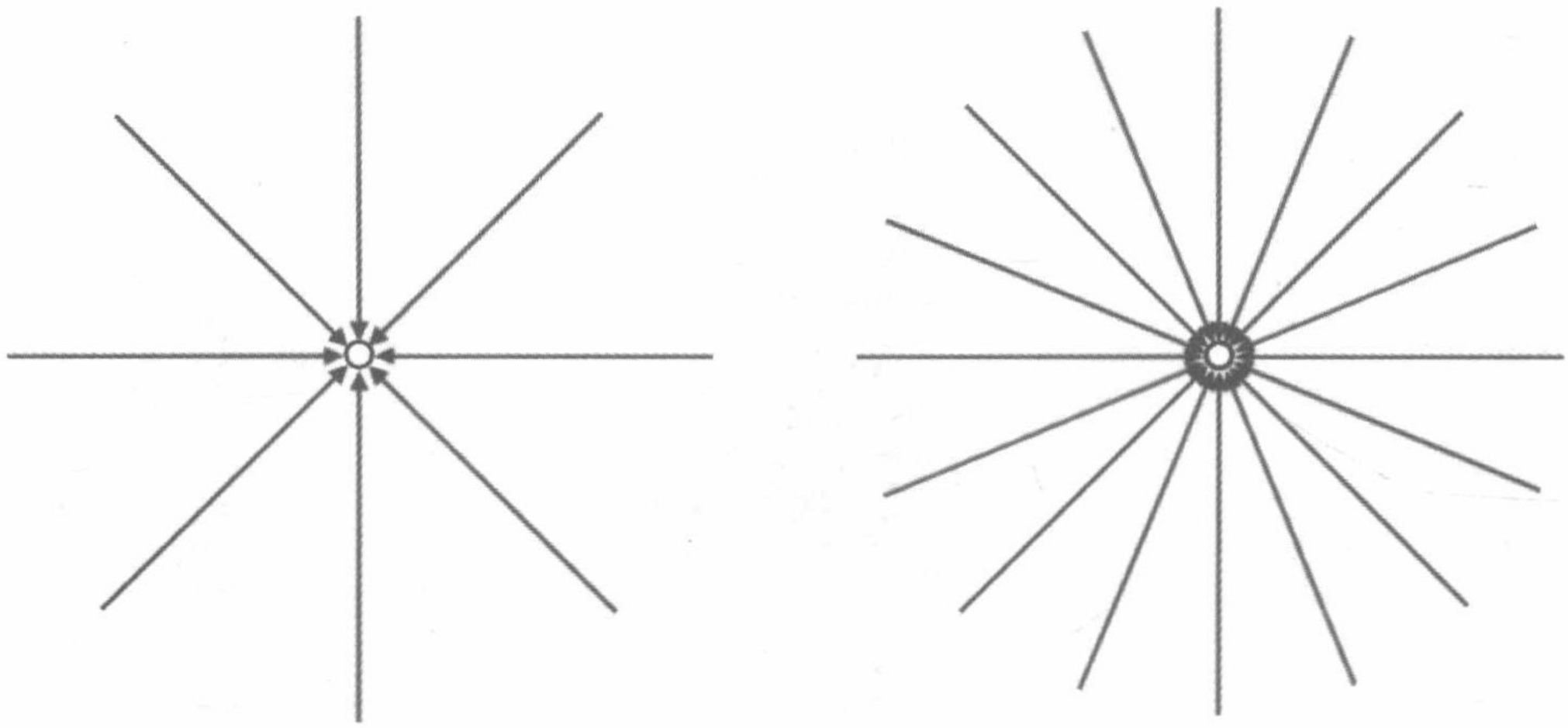


图19-19. 原始半全局块匹配使用8路径或16路径计算。为了质量，OpenCV使用8路径，为了速度，则使用5路径或者3路径

使用cv::StereoSGBM计算立体深度

如同cv::StereoBM一样，半全局块匹配算法由OpenCV呈现为一个对象，它保存所有必要的参数，并为实际计算的视差提供了一个重载的compute()。

```
class StereoSGBM : public StereoMatcher {

public:

  enum {
      MODE_SGBM      = 0,// 5-dir mode
      MODE_HH        = 1,// 8-dir mode (slow, eats lots of memory!)
      MODE_SGBM_3WAY = 2 // 3-dir mode, the fastest one
  };

  // same as in StereoBM
  //
  virtual int getPreFilterCap() const                    = 0;
  virtual void setPreFilterCap(int preFilterCap)         = 0;

  // same as in StereoBM
  //
  virtual int getUniquenessRatio() const                 = 0;
  virtual void setUniquenessRatio(int uniquenessRatio) = 0;

  // penalty for the difference in disparity between
  // neighbor pixels = 1.
  //
  virtual int getP1() const                              = 0;
  virtual void setP1(int P1)                             = 0;

  // penalty for the difference in disparity between
  // neighbor pixels > 1.
  //
```

```
    virtual int getP2() const                                   = 0;
    virtual void setP2(int P2)                                  = 0;

    // choose between MODE_SGBM, MODE_HH, MODE_SGBM_3WAY
    //
    virtual int getMode() const                                 = 0;
    virtual void setMode(int mode)                              = 0;

    // the constructor function
    //
    static Ptr<StereoSGBM> create(
      int minDisparity,
      int numDisparities,
      int blockSize,
      int P1                = 0,
      int P2                = 0,
      int disp12MaxDiff     = 0,
      int preFilterCap      = 0,
      int uniquenessRatio   = 0,
      int speckleWindowSize = 0,
      int speckleRange      = 0,
      int mode              = StereoSGBM::MODE_SGBM
    );

};
```

在创建一个cv::StereoSGBM对象时，有三个必需的参数，之后是一整套可选的参数。第一个参数minDisparity是待考虑的最小视差（通常为0）。numDisparities是需要考虑的视差总数，所以最大可能的视差等于minDisparity加上numDisparities。

参数blockSize与在StereoBM中有相同的含义，但这里建议将其设置为一个更小的值，通常3或者5就够了，但实际上可以将其设置为1以模拟原始文献的操作（只不过OpenCV互信息实现更复杂的准则）。这个值必须始终为奇数的。

参数P1、P2分别对应于SGM算法的参数，将它们设置为零，使得实现基于图像分辨率和blockSize来计算它们的一些最佳值。对于接下来的参数，preFilterCap、uniquenessRatio、speckleWindowSize和speckleRange都为cv::StereoBM中相同的含义。

disp12MaxDiff的值用于通过从左到右计算和通过从右到左计算得到的视差之间的最终比较。如果二者不同于disp12MaxDiff，则像素被声明为未知。

mode的值为默认为StereoSGBM::MODE_SGBM（算法的五向版本）。如果将其设置为StereoSGBM::MODE_SGBM_3WAY，算法将更快；如果设置为StereoSGBM::MODE_SGBM_HH，算法将如前面章节中描述的操作，并且使用双通道方法来解析传递的所有八个方向。双

通道传递方法的问题是，如同我们前面讨论的，它需要大量的内存来存储传递之间的即时结果。[注45]

作为替代，将mode设为StereoSGBM::MODE_SGBM或者StereoSGBM::MODE_SGBM_3WAY仅适用于八个方向中的三个或五个来有效的计算每个像素的成本。这减少了最终结果的精确度，但是不会导致双通道所需要的巨大内存缓冲区的成本。

立体校正、标定和对应的示例代码

让我们把所有代码放在一个示例程序中，该示例程序将从名为*list.txt*的文件中读取多个棋盘模式。这个文件包含交替的左右立体图像对的列表，用于标定摄像机并标定图像。再次注意，我们假设已经将摄像机排列好了，所以它们的图像扫描线大致上是物理对准的，从而每个摄像机本质上有同一个视图区域。这将避免极点在图像内的问题，[注46]还可以在重投影视差最小化的同时使立体重叠区域面积最大化。

在代码中（示例19-3），我们首先读取左右图像对，查找亚像素精度的棋盘角点，然后在可以找到所有棋盘图像上设置目标和图像点。这个过程可以选择性显示。给定合格棋盘图像上的查找点序列，代码调用cv::stereoCalibrate()来标定摄像机。标定给出两个摄像机的摄像机矩阵_M和畸变矢量_D；同时产生旋转矩阵_R、平移向量_T、本征矩阵_E和基本矩阵_F。

接下来是一个小插曲，通过检查一个图像上的点与另一个图像极线的距离来评价标定的精确度。为了实现这一步，我们使用cv::undistorPoints对原始点做去畸变处理（参考第18章），使用cv::computerCorrespondEpilines()计算极线，并计算这些点和线的点积（在理想情况下，这些点积均为0）。积累的绝对距离形成误差。

之后，代码可以选择使用未标定的（Hartley）方法cv::stereoRectifyUncalibrated()或者标定的（Bouguet）方法cv::stereoRectify来计算标定映射。如果使用未标定标定，代码进一步允许从头开始计算所需的基本矩阵，或者仅适用于立体校正得到的基本矩阵。我们接下来使用cv::remap()计算标定图像。在我们的示例中，绘制直线穿过图像对，这有助于观察标定图像是如何排列的。示例的一个结果可以参见图19-13，其中可以看到在原图像上的桶形畸变从顶端到底部得以极大限度纠正，并且图像与水平扫描线对准。

注45：实际内存需要的大致为图像乘以可能视差数量的面积对于一个640×480的图像和1024个可能视差，每个16位的视差有600 MB。对于一个HD图像，将增加到超过4 GB。

注46：当极点位于图像帧内，OpenCV不进行（还没有）处理标定立体图像的情况。请看，例如Pollefeys, Koch, and Gool [Pollefeys99b]对这种情况的讨论。

最后，如果已经对图像标定，我们接下来便可以利用cv::StereSGBM计算视差映射。[注47]我们的代码示例允许使用水平对准（左－右）或者垂直对准（顶部－底部）摄像机；但是要注意，对于垂直对准的情况，函数cv::StereSGBM只能计算未标定标定的视差，除非你自己添加图像转置的代码。对于水平摄像机排列，cv::StereoSGBM()可以找到标定的或者未标定标定立体图像对的视差。参见下一节示例“视差结果”中的图19-20。

示例 19-3：立体校正、校正和匹配

```
#pragma warning( disable: 4996 )

#include <opencv2/opencv.hpp>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

using namespace std;

void help( char* argv[] ) {
  ...
}

static void StereoCalib(
  const char* imageList,
  int         nx,
  int         ny,
  bool        useUncalibrated
) {

  bool              displayCorners   = false;
  bool              showUndistorted  = true;
  bool              isVerticalStereo = false; // horiz or vert cams
  const int         maxScale         = 1;
  const float       squareSize       = 1.f;   // actual square size
  FILE*             f                = fopen( imageList, "rt" );
  int               i, j, lr;
  int               N                = nx*ny;
  vector<string>                imageNames[2];
  vector< cv::Point3f >         boardModel;
  vector< vector<cv::Point3f> > objectPoints;
  vector< vector<cv::Point2f> > points[2];
  vector< cv::Point2f >         corners[2];
  bool                          found[2]         = {false, false};
  cv::Size                      imageSize;

  // READ IN THE LIST OF CIRCLE GRIDS:
  //
```

注47：这个代码还包括（注释掉）使用cv::StereoBM的情况。在这种情况下使用cv::StereoBM::BASIC_PRESET，然后直接调整cv::StereoBM对象中状态成员变量的一些附加参数。

```
if( !f ) {
  cout << "Cannot open file " << imageList << endl;
  return;
}

for( i = 0; i < ny; i++ )
  for( j = 0; j < nx; j++ )
    boardModel.push_back(
      cv::Point3f((float)(i*squareSize), (float)(j*squareSize), 0.f)
    );

i = 0;
for(;;) {

  char buf[1024];
  lr = i % 2;
  if( lr == 0 ) found[0] = found[1] = false;


  if( !fgets( buf, sizeof(buf)-3, f )) break;
  size_t len = strlen(buf);
  while( len > 0 && isspace(buf[len-1]))  buf[--len] = '\0';
  if( buf[0] == '#') continue;

  cv::Mat img = cv::imread( buf, 0 );
  if( img.empty() ) break;
  imageSize = img.size();
  imageNames[lr].push_back(buf);

  i++;

  // If we did not find board on the left image,
  // it does not make sense to find it on the right.
  //
  if( lr == 1 && !found[0] )
    continue;

  // Find circle grids and centers therein:
  for( int s = 1; s <= maxScale; s++ ) {

    cv::Mat timg = img;
    if( s > 1 )
      resize( img, timg, cv::Size(), s, s, cv::INTER_CUBIC );
    found[lr] = cv::findCirclesGrid(
      timg,
      cv::Size(nx, ny),
      corners[lr],
      cv::CALIB_CB_ASYMMETRIC_GRID | cv::CALIB_CB_CLUSTERING
    );
    if( found[lr] || s == maxScale ) {
      cv::Mat mcorners( corners[lr] );
      mcorners *= (1./s);
    }
    if( found[lr] ) break;
```

```
}
if( displayCorners ) {

  cout << buf << endl;
  cv::Mat cimg;
  cv::cvtColor( img, cimg, cv::GRAY2BGR );

  // draw chessboard corners works for circle grids too
  cv::drawChessboardCorners(
    cimg, cv::Size(nx, ny), corners[lr], found[lr]
  );
  cv::imshow( "Corners", cimg );
    if( (cv::waitKey(0)&255) == 27 ) // Allow ESC to quit
      exit(-1);

  }
  else
    cout << '.';

  if( lr == 1 && found[0] && found[1] ) {

    objectPoints.push_back(boardModel);
    points[0].push_back(corners[0]);
    points[1].push_back(corners[1]);

  }
}
fclose(f);

// CALIBRATE THE STEREO CAMERAS
cv::Mat M1 = cv::Mat::eye( 3, 3, CV_64F );
cv::Mat M2 = cv::Mat::eye( 3, 3, CV_64F );
cv::Mat D1, D2, R, T, E, F;
cout <<"\nRunning stereo calibration ...\n";
cv::stereoCalibrate(
  objectPoints,
  points[0],
  points[1],
  M1, D1, M2, D2,
  imageSize, R, T, E, F,
  cv::TermCriteria(
    cv::TermCriteria::COUNT | cv::TermCriteria::EPS, 100, 1e-5
  ),
  cv::CALIB_FIX_ASPECT_RATIO
    | cv::CALIB_ZERO_TANGENT_DIST
    | cv::CALIB_SAME_FOCAL_LENGTH
);
cout <<"Done\n\n";

// CALIBRATION QUALITY CHECK
// because the output fundamental matrix implicitly
// includes all the output information,
// we can check the quality of calibration using the
// epipolar geometry constraint: m2^t*F*m1=0
vector< cv::Point3f > lines[2];
```

```
double avgErr = 0;
int nframes = (int)objectPoints.size();

for( i = 0; i < nframes; i++ ) {

  vector< cv::Point2f >& pt0 = points[0][i];
  vector< cv::Point2f >& pt1 = points[1][i];

  cv::undistortPoints( pt0, pt0, M1, D1, cv::Mat(), M1 );
  cv::undistortPoints( pt1, pt1, M2, D2, cv::Mat(), M2 );
  cv::computeCorrespondEpilines( pt0, 1, F, lines[0] );
  cv::computeCorrespondEpilines( pt1, 2, F, lines[1] );

  for( j = 0; j < N; j++ ) {
    double err = fabs(
      pt0[j].x*lines[1][j].x + pt0[j].y*lines[1][j].y + lines[1][j].z
    ) + fabs(
      pt1[j].x*lines[0][j].x + pt1[j].y*lines[0][j].y + lines[0][j].z
    );
    avgErr += err;
  }

}

cout << "avg err = " << avgErr/(nframes*N) << endl;

// COMPUTE AND DISPLAY RECTIFICATION
//
if( showUndistorted ) {

  cv::Mat R1, R2, P1, P2, map11, map12, map21, map22;

  // IF BY CALIBRATED (BOUGUET'S METHOD)
  //
  if( !useUncalibrated ) {
    stereoRectify(
      M1, D1, M2, D2,
      imageSize,
      R, T, R1, R2, P1, P2,
      cv::noArray(), 0
    );
    isVerticalStereo =
      fabs(P2.at<double>(1, 3)) > fabs(P2.at<double>(0, 3));
    // Precompute maps for cvRemap()
    initUndistortRectifyMap(
      M1, D1, R1, P1, imageSize, CV_16SC2, map11, map12
    );
    initUndistortRectifyMap(
      M2, D2, R2, P2, imageSize, CV_16SC2, map21, map22
    );
  }
  // OR ELSE HARTLEY'S METHOD
  //
  else {
    // use intrinsic parameters of each camera, but
```

```
// compute the rectification transformation directly
// from the fundamental matrix
vector< cv::Point2f > allpoints[2];
for( i = 0; i < nframes; i++ ) {
  copy(
    points[0][i].begin(),
    points[0][i].end(),
    back_inserter(allpoints[0])
  );
  copy(
    points[1][i].begin(),
    points[1][i].end(),
    back_inserter(allpoints[1])
  );
}
cv::Mat F = findFundamentalMat(
  allpoints[0], allpoints[1], cv::FM_8POINT
);
cv::Mat H1, H2;
cv::stereoRectifyUncalibrated(
  allpoints[0], allpoints[1],
  F,
  imageSize,
  H1, H2,
  3
);

R1 = M1.inv()*H1*M1;
R2 = M2.inv()*H2*M2;
// Precompute map for cvRemap()
//
cv::initUndistortRectifyMap(
  M1, D1, R1, P1,
  imageSize,
  CV_16SC2,
  map11, map12
);
cv::initUndistortRectifyMap(
  M2, D2, R2, P2,
  imageSize,
  CV_16SC2,
  map21, map22
);
}

// RECTIFY THE IMAGES AND FIND DISPARITY MAPS
//
cv::Mat pair;
if( !isVerticalStereo )
  pair.create( imageSize.height, imageSize.width*2, CV_8UC3 );
else
  pair.create( imageSize.height*2, imageSize.width, CV_8UC3 );

// Setup for finding stereo corrrespondences
//
```

```
cv::Ptr<cv::StereoSGBM> stereo = cv::StereoSGBM::create(
  -64, 128, 11, 100, 1000,
  32, 0, 15, 1000, 16,
  StereoSGBM::MODE_HH
);

for( i = 0; i < nframes; i++ ) {

  cv::Mat img1 = cv::imread( imageNames[0][i].c_str(), 0 );
  cv::Mat img2 = cv::imread( imageNames[1][i].c_str(), 0 );
  cv::Mat img1r, img2r, disp, vdisp;

if( img1.empty() || img2.empty() )
  continue;

cv::remap( img1, img1r, map11, map12, cv::INTER_LINEAR );
cv::remap( img2, img2r, map21, map22, cv::INTER_LINEAR );
if( !isVerticalStereo || !useUncalibrated ) {
  // When the stereo camera is oriented vertically,
  // Hartley method does not transpose the
  // image, so the epipolar lines in the rectified
  // images are vertical. Stereo correspondence
  // function does not support such a case.
  stereo->compute( img1r, img2r, disp);
  cv::normalize( disp, vdisp, 0, 256, cv::NORM_MINMAX, CV_8U );
  cv::imshow( "disparity", vdisp );
}
if( !isVerticalStereo )
{
  cv::Mat part = pair.colRange(0, imageSize.width);
  cvtColor( img1r, part, cv::GRAY2BGR);
  part = pair.colRange( imageSize.width, imageSize.width*2 );
  cvtColor( img2r, part, cv::GRAY2BGR);

  for( j = 0; j < imageSize.height; j += 16 )
    cv::line(
      pair,
      cv::Point(0,j),
      cv::Point(imageSize.width*2,j),
      cv::Scalar(0,255,0)
    );
}
else {
  cv::Mat part = pair.rowRange(0, imageSize.height);
cv::cvtColor( img1r, part, cv::GRAY2BGR );
part = pair.rowRange( imageSize.height, imageSize.height*2 );
cv::cvtColor( img2r, part, cv::GRAY2BGR );

for( j = 0; j < imageSize.width; j += 16 )
  line(
    pair,
    cv::Point(j,0),
    cv::Point(j,imageSize.height*2),
    cv::Scalar(0,255,0)
  );
```

```
      }
      cv::imshow( "rectified", pair );
      if( (cv::waitKey()&255) == 27 )
        break;
    }
  }
}

int main(int argc, char** argv) {

  help( argv );
  int board_w = 9, board_h = 6;
  const char* board_list = "ch12_list.txt";
  if( argc == 4 ) {
    board_list = argv[1];
    board_w    = atoi( argv[2] );
    board_h    = atoi( argv[3] );
  }
  StereoCalib(board_list, board_w, board_h, true);
  return 0;

}
```

来自三维重投影的深度映射

许多算法仅直接使用视差映射——例如，检测目标是否在工作台上（或从工作台上伸出）。但是对于三维形状匹配、三维模型学习、机器人抓取等等，我们需要实际三维重构或者深度映射。幸运的是，所有我们已经建立的立体机制都可以容易的实现这些。回顾在标定立体校正小节中介绍过的4×4的重投影矩阵Q同时回忆一下，给定视差d和一个二维点(x, y)，我们可以得到三维深度使用以下方程：

$$Q \cdot \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

这里三维坐标由下式给出：$(X/W, Y/W, Z/W)$。值得注意的是，矩阵Q表示了摄像机视线是否收效（双目对视），也表示了摄像机的基线与两幅图像的主要的点。所以，我们不需要明确说明收敛或者前向平行摄像机，而是简单通过矩阵乘法提取深度。OpenCV有两个可以进行这种操作的函数。第一个函数你已经十分熟悉，操作的是一序列点及其关联视差。函数名为cv::perspectiveTransform()：

```
void cv::perspectiveTransform(
  cv::InputArray src,     // Input 2 or 3 channel array (a list
                          // of 2d or 3d vectors)
  cv::OutputArray dst,    // Output array, same size as 'src'
  cv::InputArray Q        // 3x3 or 4x4 floaring point
```

```
                                    // transformation matrix
);
```

第二个函数（之前我们没有遇到）是cv::reprojectImageTo3D()，操作的是整个图像：

```
void cv::reprojectImageTo3D(
  cv::InputArray  disparity, // Input disparity image, any of:
                             // U8, S16, S32, or F32
  cv::OutputArray _3dImage,  // image: 3d location of each pixel
  cv::InputArray  Q,         // 4x4 Perspective Transformation
                             // (from stereoRectify())
  cv::bool        handleMissingValues = false, // map "unknowns" to
                             // large distance
  cv::int         ddepth = -1 // depth for '_3dimage',
                             // can be any of:
                             // CV_16S, CV_32S,
                             // or CV_32F (default)
);
```

该程序采用单通道视差图像，并利用4×4重投影矩阵Q，将每个像素的（x, y）坐标和视差（例如矢量$(x, y, d)^T$）转换到相应的三维点$(X/W, Y/W, Z/W)$。输出是与输入相同大小的三通道图像。默认情况下，该图像将是32位的浮点型，但是它可以用参数ddepth控制，该参数可以被设置为CV_32F、CV_32S或者CV16S中的任何一种。最后一个参数handleMissingValues，控制cv::reprojectImageTo3D在视差图上与像素做什么，其中不能计算视差。在handleMissingValues是false的情况下，这些点不会出现在输出图像上。如果是true，那么点将产生，但会被设置为一个非常大的深度值（目前为10000）。

当然，这两个函数可以让你通过cv::stereoRectify计算任何透视变换（例如规范的）或者它的叠加和任意的三维旋转、平移，等等。杯子和椅子图像的cv::reprojectImageTo3D结果如图19-20所示。

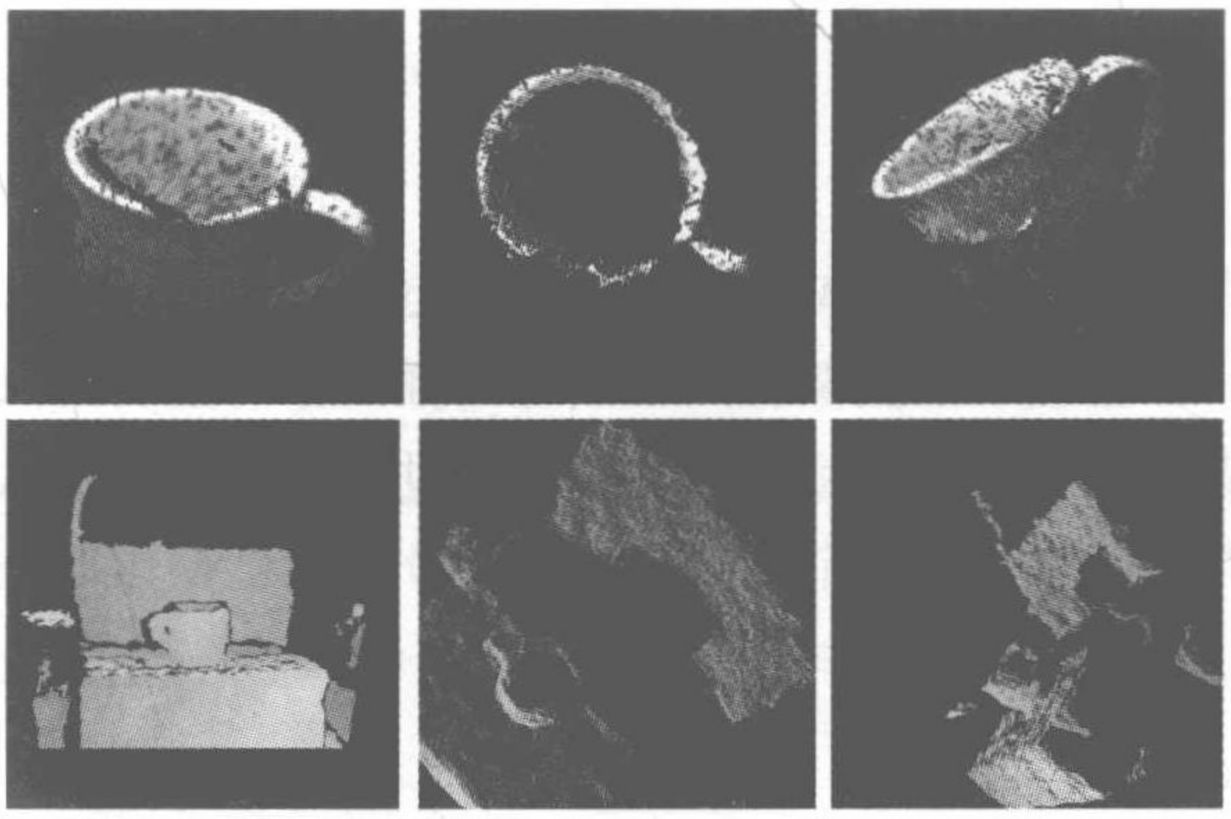


图19-20. 使用StereoBM和reprojectImageTo3D()计算的深度映射输出实例（对于杯子和椅子）（图像柳树车库授权使用）

来自运动的结构

来自运动的结构（SfM）是移动机器人和更广泛的视频图像分析，例如来自手持摄像机等领域的一个重要话题。来自运动的结构，这一话题十分宽泛，在领域内已经做了大量研究工作。但是，我们可以通过一个简单的观察完成许多工作：在静态场景中，由移动摄像机拍摄的一幅图像和任何另一台摄像机拍摄的图像没有什么不同。所以，所有的直觉和数学以及算法机制，都可以迅速应用在这种情形中。当然，描述词“静态”是十分严格的，但在许多实际情形下，场景要么是静止的，要么是近似静止的，即场景中只有很少几个移动点，可以通过鲁棒拟合方法，作为异常点来进行处理。

考虑一台摄像机移动通过建筑的情况。如果环境中可辨识的特征相对较多，那么我们可以计算连续帧之间的对应关系。例如，我们可以通过光流技术找到对应的点，如`cv::calcOpticalFlowPyrLK()`。如果我们可以跟踪帧到帧之间足够多的点，我们可以重建摄像机的轨迹。[注48]利用这一轨迹，我们之后可以进行操作来构造建筑的整体三维结构和该建筑中所有上述特征的位置。*opencv_contrib/modules/sfm*中全新的（在写作时）模块包含一个即用型SfM管道实现及其使用教程，[注49]详情请参阅附录B。此代码利用libmv和Ceres库，并提供相关的下载说明。

二维与三维直线拟合

本章中最后一个感兴趣的话题是常规的直线拟合。这可以由于许多原因和许多情况而出现。我们选择在这里讨论，因为出现直线拟合特别频繁的一种情况是分析三维点（尽管这里描述的函数也能拟合二维直线）。直线拟合算法通常使用静态鲁棒技术[Inui03, Meer91, Rousseeuw87]。OpenCV直线拟合算法`cv::fitline()`可以在任何直线拟合的情况下使用。

```
void cv::fitLine(
  cv::InputArray  points,    // 2d or 3d, can be Nx2, 2xN,
                             // vector<Point2d> etc...
  cv::OutputArray line,      // Output lines, array of:
                             // Vec4f (2d) or Vec6f (3d)
  int             distType, // Distance type used (see Table 19-3)
  double          param,     // Parameter 'C' used by some distance
                             // types (see Table 19-3)
  double          reps,      // Sufficient accuracy radius
  double          aeps       // Sufficient accuracy angle
);
```

注48：我们需要的信息包含在本征矩阵E中，可以通过基本矩阵F和摄像机内参数矩阵M计算得到。我们需要在视频流中对每一对顺序帧提取这些信息。

注49：这是2015年谷歌代码之夏（GSoC）项目的结果。

数组points可以是任何常规形式，并可能既包括二维，有包括三维的点。参数distType表示要对所有点列最小化的距离准则，如表19-3所示。

表19-3：用于计算distType值的准则

disType的值	测量	
cv::DIST_L1	$\rho(r) = r$	
cv::DIST_L2	$\rho(r) = \frac{r^2}{2}$	
cv::DIST_L12	$\rho(r) = \left[\sqrt{1 + \frac{r^2}{2}} - 1\right]$	
cv::DIST_FAIR	$\rho(r) = c^2\left[\frac{r}{c} - \log\left(1 + \frac{r}{c}\right)\right]$	$c = 1.3998$
cv::DIST_WELSCH	$\rho(r) = \frac{c^2}{2}\left[1 - \exp - \left(\frac{r}{c}\right)^2\right]$	$c = 1.3998$
cv::DIST_HUBER	$\rho(r) = \begin{cases} \frac{r^2}{2} & r < c \\ c\left(r - \frac{c}{2}\right) & r \geq c \end{cases}$	$c = 1.345$

参数param用于设置表19-3中的参数c。如果选择使用表中的值，它可以被设置为0。在描述line后，我们将回到reps和aeps。

参数line是结果存储的位置。如果points包含二维点，则line将是一个指向4个浮点数数组的指针数组（比如cv::Vec4f）。如果points包含三维点，则line将是一个指向6个浮点数数组的指针（比如cv::Vec6f）。在前一种情况中，返回值将是(v_x, v_y, x_0, y_0)，其中$(v_x, v_y,)$是和拟合直线平行的归一化向量，(x_0, y_0)是直线上的一点。同样的，在后一种（三维）情况中，返回值将是$(v_x, v_y, v_z, x_0, y_0, z_0)$，其中$(v_x, v_y, v_z)$是一个平行于拟合直线的归一化向量，$(x_0, y_0, z_0)$是直线上的一点。已知直线的表达形式，精度评价参数reps和aeps如下：reps是x0，y0[，z0]估计的请求精度，aeps是vx，vy[，vz]的请求角度精度。OpenCV文档对这两个精度的推荐值都是0.01。

我们将以一个程序结束直线拟合，如示例19-4所示。在这段代码中，我们首先在一条直线上对一些二维噪声点进行合成，之后引入一些对直线没有影响的随机点（例如，异常点），最后将点拟合成直线并显示这条直线。cv::fitline()例程能很好地忽略异常点；这在测量值被高噪声和传感器故障等破坏的实际应用中非常重要。

示例19-4：二维直线拟合

```
#include "opencv2/opencv.hpp"
#include <iostream>
#include <math.h>

using namespace std;

void help( argv) {
  ...
}
```

```
int main( int argc, char** argv ) {

  cv::Mat img(500, 500, CV_8UC3);
  cv::RNG rng(-1);
  help( argv );
  for(;;) {

    char key;
    int i, count = rng.uniform(0,100) + 3, outliers = count/5;
    float a       = (float) rng.uniform(0., 200.);
    float b       = (float) rng.uniform(0., 40.);
    float angle   = (float) rng.uniform(0., cv::PI);
    float cos_a   = cos(angle), sin_a = sin(angle);
    cv::Point pt1, pt2;
    vector< cv::Point > points( count );
    cv::Vec4f line;
    float d, t;

    b = MIN( a*0.3f, b );

    // generate some points that are close to the line
    for( i = 0; i < count - outliers; i++ ) {
      float x = (float)rng.uniform(-1.,1.)*a;
      float y = (float)rng.uniform(-1.,1.)*b;
      points[i].x = cvRound(x*cos_a - y*sin_a + img.cols/2);
      points[i].y = cvRound(x*sin_a + y*cos_a + img.rows/2);
    }

    // generate outlier points
    for( ; i < count; i++ ) {
      points[i].x = rng.uniform(0, img.cols);
      points[i].y = rng.uniform(0, img.rows);
    }

    // find the optimal line
    cv::fitLine( points, line, cv::DIST_L1, 1, 0.001, 0.001);

    // draw the points
    img = cv::Scalar::all(0);
    for( i = 0
      cv::circle(
        img,
        points[i],
        2,
        i < count - outliers
          ? cv::Scalar(0, 0, 255)
          : cv::Scalar(0,255,255),
        cv::FILLED,
        cv::AA,
        0
      );

    // ... and the long enough line to cross the whole image
    d = sqrt( (double)line[0]*line[0] + (double)line[1]*line[1] );
    line[0] /= d;
    line[1] /= d;
    t = (float)(img.cols + img.rows);
```

```
    pt1.x = cvRound(line[2] - line[0]*t);
    pt1.y = cvRound(line[3] - line[1]*t);
    pt2.x = cvRound(line[2] + line[0]*t);
    pt2.y = cvRound(line[3] + line[1]*t);
    cv::line( img, pt1, pt2, cv::Scalar(0,255,0), 3, cv::AA, 0 );

    cv::imshow( "Fit Line", img );

    key = (char) cv::waitKey(0);
    if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC'
      break;
  }
  return 0;
```

小结

在这章的开始，我们对摄像机系统的几何进行了回顾，并且了解到将三维世界中的点投影到成像器二维世界的基本映射是投影变换。我们了解到在一些情况中——特别是那些一列点位于一个平面中——这种变换可逆的。

即使由三维世界到图像平面的投影通常不可逆，我们了解到如果我们可以在许多图像中看到相同的一列点，就可以重建三维场景或者已知物体的姿态。附录B中描述的单独加载的目录opencv_contrib还有针对不同类型的校准模式（aruco和ccalib），以及色彩平衡和去噪程序（xphoto）的全景相机和多个摄像机（ccalib）的进一步标定算法。

最后，我们看到这样的几何信息是如何用于构建立体深度测量的。为了可靠地实现这一步，我们要计算立体图像摄像机之间的确定关系——该过程称为“立体校正”。一旦立体摄像机被标定，我们就可以使用OpenCV提供的两种可选算法中的一个计算深度。块匹配算法更快，但提供的场景覆盖不够完整，而半全局块匹配算法以更大的计算时间为代价提供更好的结果。

最终，我们介绍了其他一些有用的功能用于处理三维的点和线，如投影变换、重新投影和线拟合。

练习

1. 仿射和投影（透视投影）变换（如图11-3所示）：

$$\begin{pmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \\ c_1 & c_2 & 1 \end{pmatrix}$$

参数a形成旋转、缩放和斜倾矩阵，并构成仿射变换。参数b形成一个(x,y)平移矢量，c形成透视投影矢量，二者结合形成一个完整的透视投影矩阵。

a. 想象一个面向棋盘的摄像机。什么摄像机移动可以和一个仿射变换与一个透视投影变换一样建模？

b. 一个平面上要多少个点定义一个仿射变换？多少个点定义一个透视投影变换？

c. 在仿射投影中：直线是否保持为直线？它们是否保持相同长度？平行线是否保持平行？如果两条直线在原图像中相交，经过仿射投影它们是否始终相交？

d. 在透视投影中：直线是否保持为直线？它们是否保持相同长度？平行线是否保持平行？如果两条直线在原图像中相交，经过仿射投影它们是否始终相交？

2. 使用`cv::calibrateCamera()`和至少15个棋盘对摄像机进行标定。之后使用`cv::projectPoints()`通过摄像机标定的旋转和平移向量将正交箭头（表面法线）投影到每一个棋盘图像上。

3. 三维操纵杆：使用至少四个测量的非共面可跟踪特征点的简单已知对象作为输入量输入`cv::solvePnP()`算法。使用对象作为三维操纵杆在图像中移动一个小的附着图像。

4. 在鸟瞰图实例的相关阐述中，通过飞机上方的摄像机对水平面上俯视，我们看到地平面的单应性有条水平线，超出该水平线，单应性无效。怎么使无穷平面上拥有水平线？为什么它不会永远都有？

5. 在观测地平面上的摄像机中应用鸟瞰图。实时运行并探索当目标围绕鸟瞰图图像和常规摄像机移动时会发生什么？

6. 设置两个摄像机或一台在拍摄两张图像之间移动的摄像机。

   a. 计算、存储并检验基本矩阵。

   b. 重复计算基本矩阵多次。计算的稳定性怎样？

7. 如果有一个已标定摄像机，并且正在跟踪两个摄像机之间的移动点，你是否能想法利用基本矩阵找到跟踪误差？

8. 计算和绘制两个摄像机的极线来建立立体视觉。

9. 建立两个视频摄像机；进行立体校正和标定并检验深度精确度。

   a. 在场景中引入镜子时，会发生什么？

   b. 改变场景中的纹理并报告结果。

   c. 尝试不同视差方法并汇报结果。

10. 安装立体摄像机并在手臂上放一些纹理。使用distType方法对手臂进行直线拟合。比较不同方法的精确性和可靠性。

11. 假设在晴朗的日子里发出一台附着在氦气气球上、向下俯视的摄像机。气球向上飞，直到一定高度。单用摄像机和其帧速度，如何计算气球弹出时摄像机的高度？

第20章

机器学习基础

在本章中，我们将开始讨论将视觉转化为感知的机器（将视觉输入转化为有意义的视觉语义）。

在前几章中，本书介绍了将二维或三维传感器获取的视觉信息转换为特征、聚类或几何信息的方法。在接下来的三章中，本书将利用得到的特征、分割结果和几何结构来获取识别到的场景或目标，即将原始视觉信息转化为感知——机器看到的**结果**和目标相对于相机的**位置**。

本章将覆盖机器学习的基础知识，重点阐述相关概念。为了帮助读者更好地从整体上理解机器学习，本章将介绍一些OpenCV库中简单的机器学习功能，在下一章中，我们将详细介绍如何在OpenCV库中实现机器学习模型。[注1]

什么是机器学习

机器学习[注2]的目的是将数据转化为信息。在收集数据后，我们希望机器能够回答这些有关数据的问题：其他数据中，有哪些与此数据最相似？图像中有没有汽车？哪个广告最

注1：注意，机器学习与其他模块一样，扩展代码都包含在附录B的*opencv_contrib*中。有关详细信息，请参阅深度神经网络库模块*cnn_3dobj*和*dnn*。

注2：机器学习是一个很大的话题。OpenCV主要处理统计机器学习，而不是像贝叶斯网络、马尔可夫随机场和图形模型这样的主题。机器学习中的一些好文章包括Hastie，Tibshirani和Friedman [Hastie01]的文章，DudaHart [Duda73]; Duda、Hart和Stork [Duda00]以及Bishop [Bishop07]。关于如何并行机器学习的讨论，参见Ranger等人的[Ranger07]和Chu等人的[Chu07]。

能得到用户的响应？考虑到价格因素，这个问题可能会变成：“如果公司向用户发布一系列利润较高的产品的广告，用户最有可能购买哪一种产品？”机器学习通过从数据中提取规则或模式将数据转换为信息。

训练集和测试集

机器学习的处理对象是原始数值，例如温度值、股票价格和颜色强度等。这些数据经常被预处理为特征。[注3]假设有一个包含10 000张人脸图像的数据库，对每张人脸图片进行边缘检测，然后从收集特征，例如，边缘方向、边缘强度和边缘相对人脸中心的偏移等。这样可以从每张脸中提取出了500个特征值（可以表示为一个500维特征向量）。然后，我们就可以利用机器学习技术根据特征数据构建某种模型。如果只是想把人脸图像分成几类（包括粗分、细分等），那么聚类算法是不错的选择。但如果我们想通过边缘特征预测一个人的年龄，则需要一个“分类算法”。为了达到目的，机器学习算法分析我们收集的特征数据，调整权重、阈值和其他模型参数来提高性能。这个调整参数过程以达到目的的过程被称为“学习”。

深入了解机器学习算法的性能非常重要。一般情况下，将可用数据集分为一个较大的训练集和较小的测试集（上一段的例子中，有9000张人脸图片作为训练集，剩下的1000张人脸图片作为测试集）。然后在训练集上训练的分类器，学习年龄预测模型。训练完成后就可以用测试集中的图像测试年龄预测分类器。

测试集不能用于训练，分类器在开始预测之前并不知道测试集实际的年龄标签。在经过训练后，我们在测试数据集中的1000个人脸图片上运行分类器，得到其预测的年龄（基于特征向量），并计算预测的年龄与真实年龄的匹配度。如果分类器效果不佳，可以尝试添加新的特征，或者考虑使用不同类型的分类器。本章将介绍多种分类器和训练算法。

如果分类器效果好，我们就获得了一个潜在的有价值的模型，可以据此在现实世界中部署数据。或许这个系统可以根据用户年龄来设置视频游戏的行为。当有人准备玩时，他的脸将被提取表示成500个特征（边缘方向、边缘强度、偏离脸部中心距离等）。把这些特征传递给分类器后，系统将根据分类器返回的年龄设置相应的游戏行为。在系统部署之后，对于以前看到从未见过的面孔，分类器会根据学习结果做出判断。

最后，在训练分类系统时，经常使用验证数据集。有时，测试整个系统是一个非常大的工程。在提交分类器进行最终测试之前，经常需要修改参数。为此，本书把包含10,000

注3：在写本书的时候，OpenCV还没有包含深度学习，因为虽然发展前景比较好，但这些技术仍然太新，还不能确定到底要包含哪些内容。然而，卷积神经网络[Fukushima80；LeCun98a；Ciresan11]绝对是深度学习未来的重点内容之一。

张人脸数据的原始数据集分为三个部分：8000张作为训练集，1000张作为验证集；1000张作为测试集。当训练集上训练完成之后，我们可以在验证集上测试分类器的效果。只有对验证集合的性能感到满意时，才能用在分类器在测试集上进行最终判断。



相较于在一个含有9000个样本的数据集中采用8000个样本作为训练集，1000个样本作为测试集，显然有更好的做法。事实上，常见的做法是把数据集分块，多次按块重复取样。在本例中，把数据集分为9块效果最好。每次都用不相同的1000个数据进行验证，其余8000个数据进行训练，最终取9次验证结果的平均值。这个过程称为*k*折交叉验证，这里的*k*取值为9。

有监督学习和无监督学习

有的时候，数据是没有标签的，例如，我们可能只想根据边缘检测信息来判定人脸更可能属于哪一类。而有时，数据有标签，例如每个图像中人的年龄。这意味着针对这类数据的机器学习是有监督的（即，可以利用与数据特征向量附带的“信号”或“标签”进行训练），但是对于无监督学习，机器学习算法无法获取“标签”并自己找出数据的结构（图20-1）。

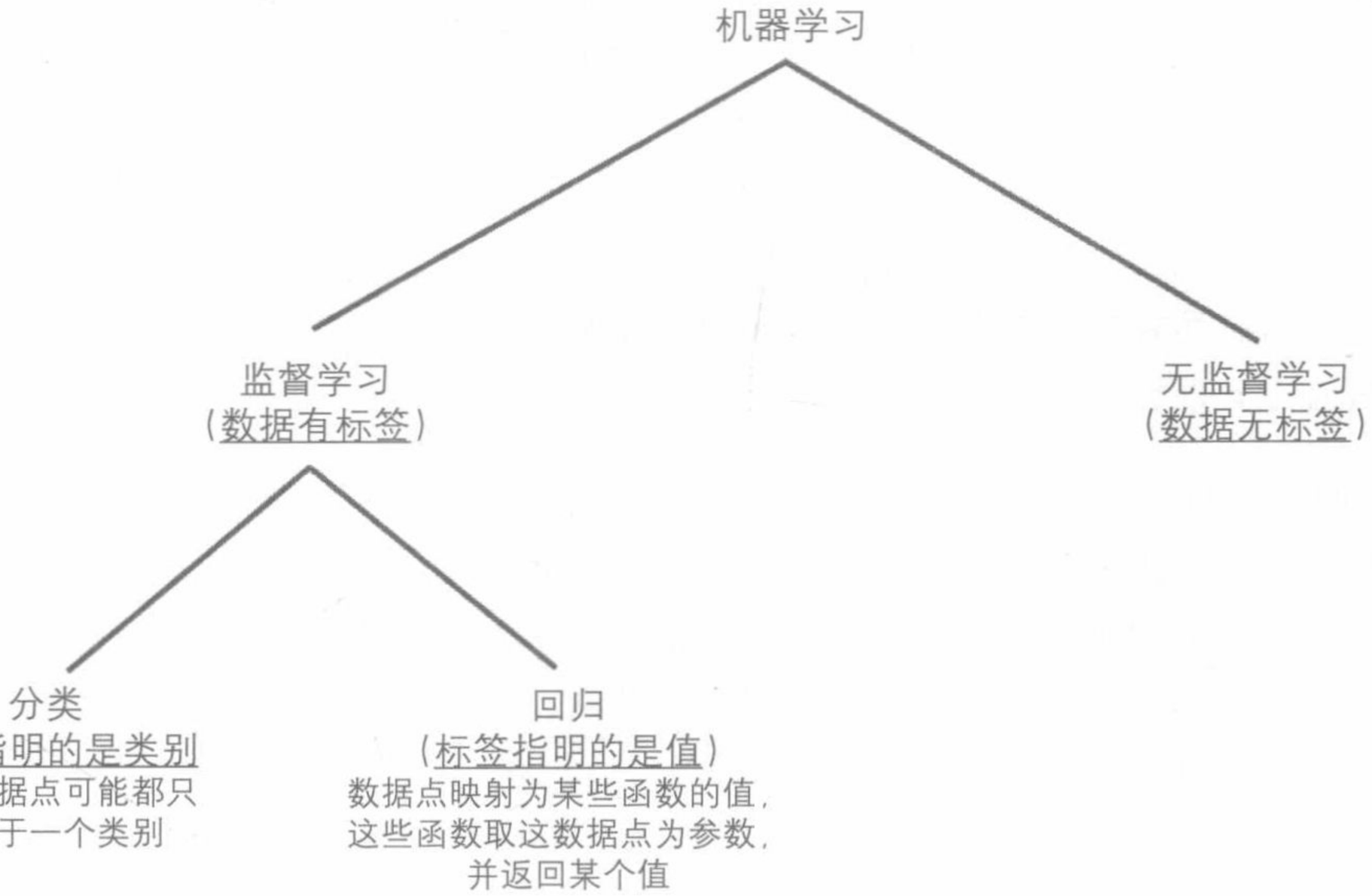


图20-1. 机器学习分为监督学习和无监督学习。监督学习又细分为两个重要类别：分类和回归

监督学习可以用于分类，例如把名字和脸对应起来，或者赋予数据一个有序的数字标签，例如年龄。当数据具有名称（类别）作为标签时，则表明这是一个分类问题。当数据是数字时，则表明这是一个回归问题。回归问题通过类别或数字输入数据来拟合一个数值输出。

但监督学习也存在“灰色地带”，它包括数据向量与标签的一一匹配，还包括强化学习（有时称为“延迟学习”）。在强化学习中，数据标签（也称为“奖励”或“惩罚”）能在单独的数据向量观察之后存在长期性的影响。当一只老鼠在迷宫中寻找食物的时候，老鼠可能会经过一系列的转弯，才能最终找到食物，这是对它的奖励。这种奖励必须能反作用于老鼠本身，影响老鼠在找到食物之前的所有行为和过程。强化学习也是一样：系统收到延迟信号（奖励或惩罚），并据此尝试推断下一步的策略。在这种情况下，正在学习的策略实际上是决策的方法，例如在迷宫中的每一步怎么走。有监督学习也适用于标签不完整的情况，即有一些标签丢失（这也称为“半监督学习”），或者可能含有噪声（一些标签是错误的）。大部分机器学习算法只能处理以上一种或两种情况。例如，有的算法可以处理分类问题，但不能解决回归问题，有的算法能够进行半监督学习，但不能进行强化学习；还有的算法可以处理数字数据，却不能处理分类数据，等等。

相比之下，有的时候我们没有数据标签，但我们对数据是否自然地分组感兴趣。这种无监督学习的算法通常称为“聚类算法”。其目的是把未标记的数据向量根据距离（事先确定的或者学习过程中得到的）远近分组。我们可能只想看看人脸的分布情况：是胖是瘦？是长还是方？如果我们正在研究癌症数据，能否根据各自化学信号的不同把癌症数据分为不同的聚类？ 无监督算法的获得的聚类数据也经常作为上级监督分类器的特征向量使用。我们可能首先将人脸聚类为不同的脸型（宽，窄，长，短），然后将其用作输入，再辅以其他数据（如平均声频），以此预测一个人的性别。

这两种常见的机器学习任务（分类和聚类）与计算机视觉中最常见的两个任务（识别和分割）有共通之处。当希望计算机识别物体是什么的时候就使用识别；当我们希望计算机给出目标出现的位置时，就使用分割。因为计算机视觉广泛应用机器学习，所以OpenCV的ML库中包含许多强大的机器学习算法，位于*.../opencv/modules/ml*和*.../opencv/modules/flann*目录中。



OpenCV机器学习代码是通用的，也就是说，虽然它在视觉任务中取得了较好的效果，但代码本身的应用范围并不限于此。例如，可以用某些函数来研究基因组序列。当然，在这里，我们仍然主要关注目标识别，其特征向量是从图像中提取的。

生成式模型和判别式模型

现在已经有了许多算法来解决分类和聚类问题。OpenCV包含当前应用最广泛的一些统计机器学习方法。基于概率的机器学习方法，如贝叶斯网络和图模型等在OpenCV中支持得不够充分，部分原因在于这些算法较新且仍处于积极的开发阶段。OpenCV倾向于支持判别式算法，而不是生成式算法。判别式算法是通过给定数据来判定类别P(L | D)的概率，而生成式算法是通过给定标签来生成数据的分布P(D | L)。虽然这两者的区别不是非常明显的，但是判别式模型在根据给定数据做出预测这一方面更有优势，而生成式模型则具有更好的数据表达能力，在有条件的生成新数据时也更有优势。（比如，想象一个大象，你是在根据条件“大象”来生成数据）。

由于生成式模型是对数据的产生进行建模，生成式模型通常更容易被理解。判别式模型则经常根据一些似乎是随意选取的阈值作出决定。例如，假设有一小片区域在一个场景中被定义为一条路，仅仅因为它的颜色中“红色”分量小于125。但这是否意味着红色分量为126的区域一定不是路呢？ 这些问题难以解释清楚。生成式模型通常根据给定的数据类别得到数据的条件分布，因此可以很容易看出这个分布跟实际的分布是否相似。

OpenCV机器学习算法

OpenCV中的机器学习算法如表20-1所示。ML库中包含许多算法，但仍然有一些算法包含在其他一些库中。比如，Mahalanobis算法和K-均值算法在core库中，目标识别和人脸识别在objdetect库中。

表20-1：Opencv支持的机器学习算法，算法描述后附有原始参考文献

算法名称	描述
Mahalanobis	通过除以协方差来对数据空间进行变换，由此得到一个标志着空间拉伸程度的距离度量。如果协方差矩阵是单位矩阵，那么这个度量将等价于欧氏距离[Mahalanobis36]。
K-均值算法	这是一种非监督的聚类方法，用*k*个聚类中心表示数据的分布，*k*的大小由用户定义。该方法与期望最大化算法（expectation maximization）的区别在于*k*-均值的中心不符合高斯分布。而由于每个聚类中心都竞争着去俘获与之距离较近的点，所以聚类的结果更像是肥皂泡，聚类区域经常被用作稀疏直方图的bin，用来描述数据。该方法最初由Steinhaus [Steinhaus56]提出，由Lloyd [Lloyd57]进一步推广
正态/朴素贝叶斯	这是一个生成式的分类器。它假设特征符合高斯分布且在统计学上服从独立同分布，但是这个过于苛刻的假设在很多情况下是不能满足的。因此，它又被称为“朴素”贝叶斯分类器。尽管如此，在许多情况下，这个方法的效果却出奇得好。该方法最早出自[Maron61；Minsky61]

表20-1：Opencv支持的机器学习算法；算法描述后附有原始参考文献（续）

算法名称	描述
决策树	这是一个判别分类器。该树在当前节点通过寻找数据特征和一个阈值来实现数据到不同类别的最优划分。处理流程是不断将数据进行划分，然后下降到树的左侧子节点或者右侧子节点。虽然该算法的性能不是最好的，但往往是测试时的首选，因为它运行速度快，性能好[Breiman84]
期望最大化算法（EM）	这是一种用于聚类的无监督生成算法。它会用*N*维高斯分布去拟合数据，其中*N*的大小是由用户自己定义的。这样仅仅用很少的几个参数（方差和均值）就可以表示一个较为复杂的数据分布。该方法经常用于分割，它与*k*-均值算法的比较见文献[Dempster77]
Boosting	它是由一组判别分类器组成。最终的决策是由组成它的各个分类器的加权组合来决定。在训练中，逐个训练子分类器。且每个子分类器都是弱分类器（性能上只是优于随机选择）。这些弱分类器由单变量决策树构成，被称为“桩”（stumps）。在训练中，决策桩（decision stump）不仅从数据中学习分类决策而且根据预测精度学习“投票”的权重。当训练下一个分类器的时候，数据样本的权重会被重新分配。使之能够给予分错的数据更多的注意力。训练过程不停地执行，直到总错误（加权组合所有决策树组成的分类器产生的错误）低于某个已经设置好的阈值。为了达到好的效果，这个方法通常需要很大量训练的训练数据[Freund97]
随机森林	它是由许多决策树组成的“森林”，每个决策树向下递归以获取最大深度。在学习过程中，每棵树的每个节点只从特征数据的一个随机子集中选择。这保证了每棵树是统计意义上不相关的分类器。在识别过程中，将每棵树的结果进行投票来确定最终结果，每棵树的权重相同。这个分类器方法经常很有效，而且对每棵树的输出进行平均，可以处理回归问题[Ho95; Criminisi13；Breiman01]
*K*近邻	*K*近邻可能是最简单的分类器。训练数据跟类别标签存放在一起，离测试数据最近的（欧氏距离最近）*K*个样本进行投票，确定测试数据的分类结果。该方法通常比较有效，但是速度比较慢且对内存需求比较大[Fix51]，详见FLANN 入口
快速近似最近邻法（FLANN）[a]	Opencv包含完整的快速近似最近邻算法函数库，该库由Marius Muja开发，可算实现*K*近邻的快速近似和匹配[Muja09]
支持向量机（SVM）	它可以进行分类，也可以进行回归。该算法需要定义一个高维空间中任两点的距离函数。（将数据投影到高维空间会使数据更容易地线性可分。）该算法可以学习一个分类超平面，用来在高维空间里实现最优分类器。当数据有限的时候，该算法可以获得非常好的性能，而boosting和随机森林只能在拥有大量训练数据[b]时才有好的效果
人脸识别/级联分类器	这个物体检测算法巧妙地使用了boosting算法。OpenCV提供了正面人脸检测的检测器，它的检测效果非常好。你可以使用OpenCV提供的训练算法，也可以使用这个分类器提取你自己想要的特征或使用其他的特征，使之能够检测其他物体。该算法非常擅长检测特定视角的刚性物体。这个分类器也称“Viola-Jones分类器”，这是以发明者的名字命名的[Viola04]

表20-1：Opencv支持的机器学习算法；算法描述后附有原始参考文献（续）

算法名称	描述
Waldboost	它是Viola 算法的衍生算法（参见上一条），Waldboost是一个非常快的目标检测器，它在一系列任务中的表现都要优于传统的级联分类器[Sochman05]。它存在于在.../*opencv_contrib*/模块中
隐变量支持向量机（Latent SVM）	隐变量支持向量机是一个局部模型。通过识别目标的各个独立部分并学习一个模型去发现这些部分之间联系来识别复杂目标[Felzenszwalb10]
词袋模型	词袋模型是在文档分类和视觉图像分类中都有大量应用的技术。该算法功能强大，因为它不仅可以用来识别单个目标，还可以用来识别场景和环境

a. *FLANN*中的*L*代表*library*（FLANN =快速近似最近邻库）

b. 什么样的数据才称得上是“大数据”？ 没有答案，因为它取决于底层生成过程的变化。但这是一个非常粗暴的经验法则：每个类别/对象在每个有意义的维度（特征）上有10个数据点。所以，两个类别，三个维度需要至少2 * 10 * 10 * 10 = 2000个数据点才称得上这个问题是“大数据”问题

机器学习在视觉中的应用

基本上，所有表 20-1中的算法都使用特征构成的数据向量作为输入，特征数可达数以千计。假设你的任务是识别特定的物体（例如，人）。你遇到的第一个问题是怎么采集数据并给数据定标签（场景里有人还是没有人）。你会发现人以不同的方式出现：人的图像可能只包含一点点像素，或者你只能看到一只耳朵出现在整个屏幕上。更坏的情况下，人可能处于遮挡状态下：车中的人、女人的脸以及树后面的一条腿。你需要给出有人出现在场景的确切定义。

接下来需要解决数据收集问题。是使用自己的摄像头采集，还是使用图片共享网站尝试在“人物”标签下寻找？或者两种方法都使用？采集数据时需要采集运动信息吗？除此之外，还需要采集其他信息吗？如场景中的门是开还是关，时间，季节，温度需要采集吗？用于海滩上检测人的算法可能不适用于滑雪场。所以需要捕获数据的各种变化：不同视角，不同光线，不同天气，阴影，等等。

获取了数据之后，该怎么给数据定标签？首先需要决定标签的含义。需要知道场景中的人在什么位置吗？运动信息（跑步、步行、爬行、追踪）重要吗？如果有大量的图像，又应该怎样给所有的图像定标签？有很多诀窍，比如在约束情况下做背景分割，并收集运动前景。还可以使用一些数据服务来帮助你进行图像分类。例如，可以通过Amazon

Mechanical Turk（MTurk）[译注1]，雇人帮你为数据打标签。如果任务很简单，服务的价格甚至可以低至一美分。此外，还可以使用GPU或计算机集群通过计算机图形学技术渲染目标、人物、脸、手等。利用已经生成的数据和已知相机模型参数，我们可以得到非常逼真的图像。

数据定好标签之后，需要决定要从目标中提取那些特征。再强调一次，你必须知道自己的目的。如果人们总是正面出现，就没有必要使用旋转不变的特性，也没有必要预先旋转物体。总之，必须寻找表达物体固有的属性特征，比如梯度直方图、色彩或目前流行的SIFT特征。[注4]如果有背景信息，可以首先把背景去除，提取物体，然后进行图像处理（归一化图像、尺度改变、旋转和直方图均衡），计算很多特征。物体的特征向量将于物体的标签对应。

一旦数据被转换特征向量，便需要把数据分成训练集、验证集和测试集。在这里，我推荐交叉使用训练集、验证集和测试集。也就是说：所有的数据分成K个子集，然后每次随机选取其中一部分来训练，剩余部分用来测试。[注5]测试结果求平均，得到最终的性能结果。使用交叉验证，我们能够更清楚地看到处理新数据时分类器的性能。

数据已经准备好，下面是选择分类器。一般分类器的选择需要考虑计算速度，数据形式和内存大小。一些应用中，在线用户优先选择建模，所以分类器需要能够快速完成训练。在这种情况下，最邻近算法，朴素贝叶斯和决策树都是不错的选择。如果需要考虑内存因素，决策树和神经网络是理想的选择。如果不需要训练很快，而需要判断很快，但是需要精度很高，可选择boosting和随机森林。如果选取的特征比较好，仅仅需要一个简单易懂的分类器，就选择决策树和最邻近算法。要获得最好的性能，还是离不开boosting和随机森林算法。

译注1：李飞飞认为，有了MTurk，构建ImageNet大规模数据集是可行的。该众包平台提供了一种标准化的向人类发布任务并收集数据的方式，完全由价格驱动，可以想象这样的服务“Human As a Service”，可以用于标注数据、验证甚至生产数据，它通过API和资本连接着虚拟世界和真实世界，代表人力资源的未来，按需取用的未来。观察源于知乎“围博磊”“陈然”“亦云”等，更多信息请参见*https://www.zhihu.com/question/30185115*。

注4：可以登录*http://www.cs.ubc.ca/~lowe/keypoints/*查看Lowe的“SIFT keypoint feature demo”。

注5：一般的训练（可能的验证）和测试周期是5～10次。



所谓“最好的”分类器（即可以适应任何情况），是不存在的（详见*http://en.wikipedia.org/wiki/ No_free_lunch_theorem*）。首先对所有可能的数据分布进行平均处理，所有分类器的表现都差不多。因此，我们不能说表20-1中的哪个算法是“最好的”。然而，在给定的数据分布或数据集中，最好的分类器是存在的。因此，面对真实的数据时，不妨多尝试一些分类器。在选择分类器时，首先考虑我们的目的：只是为了获得正确的分数，还是要对数据进行解释？ 是否寻求在计算时消耗更小的内存或更快的速度？要求或决定的置信限度？ 不同的分类器在面对这些需求时有不同的表现。

变量的重要性

表20-1中的两个算法可用来评估变量的重要性。[注6]拿到一个特征向量，我们怎么才知道哪个特征对分类器的准确性有较大的贡献？二叉决策树可以解决这个问题：通过在每个节点选择最能分裂出数据的变量。最上面的变量是最重要的变量，第二层的变量第二重要，以此类推。[注7]随机森林采用Leo Breiman发明[译注2]的一项技术来测量变量的重要性。这个技术可以用于任何分类器，但是在OpenCV中，目前只有决策树和随机森林实现了这个技术。

变量重要性的一个用途是减少分类器需要考虑的特征的个数。一开始有很多特征，当用户训练分类器时，发现其中一个特征与另外一些特征有关联，则可以去掉其中一些不重要的特征。排除不重要的特征可以提高速度，使训练和测试更快。若没有足够的数据，去除不重要的变量还可以提高分类器的准确率，同时使处理速度更快，效果更好。

Breiman的变量重要性算法流程如下。

1. 用训练集训练一个分类器。
2. 使用验证集或测试集来确定分类器的准确率。
3. 对于每一个数据样本和一个选择的特征，随机选择其他特征数据中的该特征的值来替代（替补抽样法）。这样可以保证特征的分布与原始数据集的一样，但是特征的结构或者意义被抹去了（因为它的值是从剩余数据中随机选择的）。[注8]

注6： 这通常称为“可变重要性”，意味着变量的重要性，而不是变化或波动的重要性。

注7： 这种技术本身对噪声非常敏感。二进制树真正做的是为每个节点构建代理分叉（导致几乎相同结果的其他分裂特征）并计算所有节点上所有分叉的重要性。

译注2：尼奥•布雷曼（1928—2005）加州伯克利的统计学家，美国国家科学院院士，机器学习先驱者，分类回归树作者之一，Bagging方法的发明人，对模式识别领域有巨大的贡献。

注8： 随机树的实际实现是改变特征值，而不是产生全新的值。

4. 用改变后的训练集训练分类器，然后用改变后的测试集或者验证集来评价分类器。如果完全打乱特征会使正确率降低很多，则说明这个特征很重要。如果完全打乱特征并没有使正确率下降多少，则说明这个特征并没有多重要，可以考虑删除。
5. 把原始数据重新换一个特征按照3和4操作，直到所有的特征全部分析完毕，得到的结果就是每个特征的重要性。

这个过程内置到随机森林和决策树中，所以可以用以上两种算法来确定使用哪些变量作为特征，然后使用删减过的特征向量来训练分类器。

诊断机器学习中的问题

将机器学习用好，不仅仅是一门技术，更是一门艺术。算法经常“部分”能用，但又不能完全与要求一致。这就是艺术的所在：需要指出哪些地方出了问题并解决问题。在这里不详细讲述，只列举一些常见问题。例如一些重要的规律：大量数据比少量数据好；好的特征比好的算法更重要。如果选取的特征好，即独立性最大，在最不同环境之下的变化最小，那么大部分算法都可以获得比较好的效果。除此之外，还有几个经常遇到的问题。[注9]

欠拟合

模型假设太严格，所以模型不能拟合到实际数据上。

过拟合

算法不仅学习了数据，还把噪声当作信号学习了，这样一来，导致算法的泛化能力很差。

错误

学习一种模式时，机器学习的代码可能包含严重的错误。这种错误在预期完全错误时会降低计算性能。

统计机器学习的基本结构如图20-2所示。我们的工作是根据输入和输出对判别函数*f*做一个建模。该函数可能是一个回归函数（例如，从人脸预测一个人的年龄[注10]）或分类预测问题（例如，给定面部特征，识别出特定的人）。在现实生活中，噪声和一些其他因素可能导致实际输出与理论值不同。例如，在人脸识别中，我们可以对眼睛、嘴巴和鼻子之间距离进行建模，据此模型来识别人脸。但是，附近的光照变化可能会在测量中引

注9：斯坦福大学的吴恩达教授在网络公开课中提供了“应用机器学习的建议”（*http://www.stanford.edu/class/cs229/materials/ML-advice.pdf*）的细节。

注10：精明的读者可能会想到，由于年龄可能被报告为整数，所以这应该是分类问题，而不是回归问题。然而，输出的连续性并不代表问题是回归问题，而是输出的有序性。因此，即使对于整数年龄，这仍然是一个回归问题。

入噪声，此外工艺较差的相机镜头可能会导致图像畸变，这些失真都是在模型中没有考虑的。这将导致预测的准确性降低。

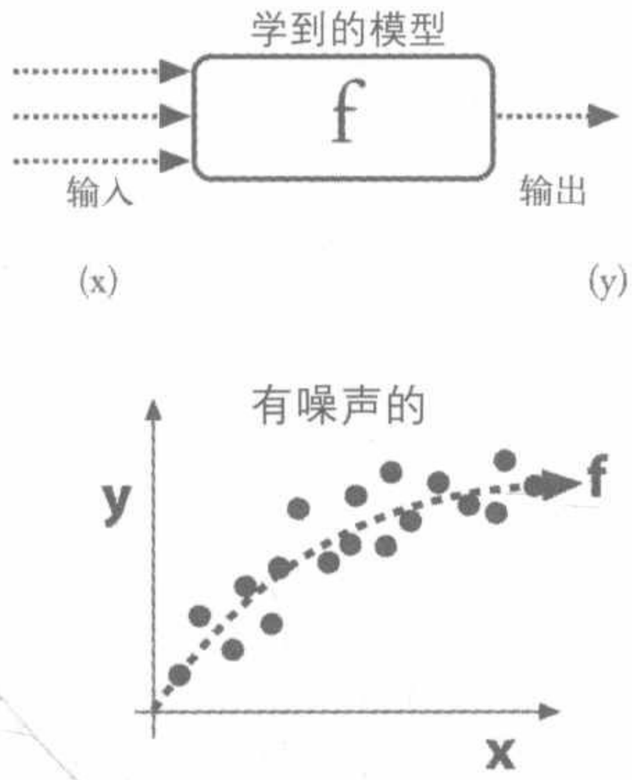


图20-2. 统计学习的结构。训练一个分类器来拟合一个数据集，真正的模型*f*会被噪声和一些位置的因素淹没

图20-3显示了数据的模型偏差（欠拟合）和方差（过度拟合）。下面两个图显示了误差与训练集大小的关系。如图20-3的左侧所示，我们尝试训练一个分类器来预测图20-2中的数据。如果我们使用一个很严格的模型（粗直的虚线），那么我们永远不能很好地拟合真正的抛物线*f*（细虚线）。如此，即使有大量数据，训练数据和测试数据的拟合程度也会很差。在这种情况下，因为欠拟合，会导致训练和测试数据预测不佳。如图20-3的右侧所示，尽管模型能够精确地拟合训练数据，但也使函数包含噪声，导致它在测试数据上的效果很差。低训练误差与高测试误差说明存在着过拟合问题。

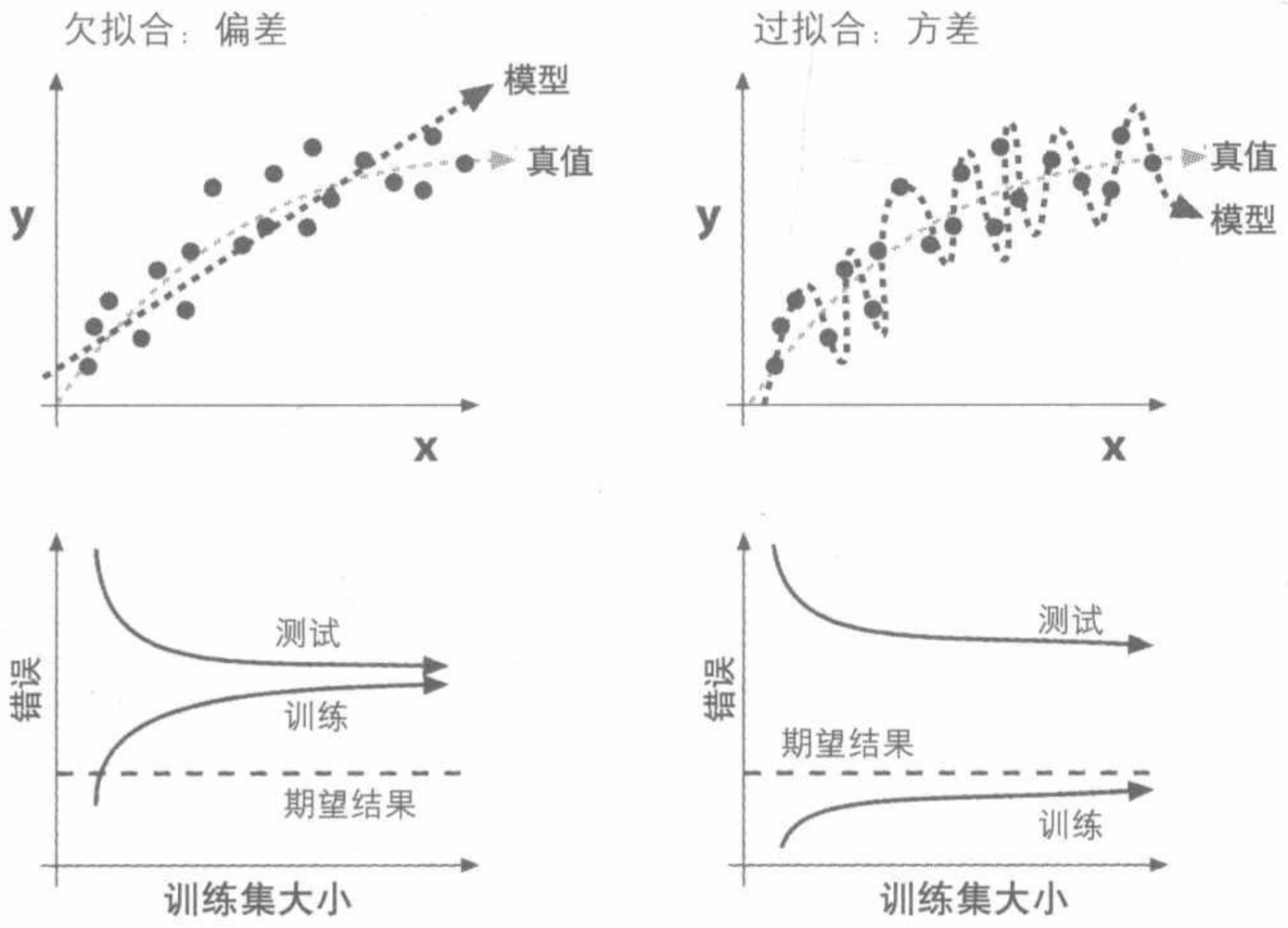


图20-3. 机器学习中比较差的拟合模型以及在训练和测试时的性能，真正的模型函数是上图中的细虚线：欠拟合模型（左上图）在训练和测试数据上都有很大的误差，而过拟合模型（右上图）在训练数据上的误差比较小，但是在测试数据上的误差很大

有些时候，需要注意正在解决的是否是自己想要解决的问题。如果训练和测试结果都很好，但算法在实际应用中效果不好，则表明数据集可能是从非实际条件中获得的，也许是因为这些条件可以让人更容易地收集和模拟数据。如果算法不能重现训练集或者测试集，表明问题出在算法，可能是因为从数据提取的特征无效，也可能是因为有用的信息不在收集的数据中。

表20-2列出了一些可能的解决方法。当然，这并不是所有的解决方案。为了使机器学习方法发挥作用，必须仔细思考并设计应该采集哪些数据，计算哪些特征。机器学习方法出现问题时，还必须做一些系统开发级的诊断。

表20-2：机器学习中遇到的问题以及可能的解决方案，采用更好的特征对任何问题都有帮助

问题	可能的解决方案
欠拟合	• 使用更多的特征有利于拟合 • 选用一个学习能力更强一些的拟合算法
过拟合	• 增加训练数据数量可以使得拟合曲线更光滑 • 减少特征的数量可降低过拟合程度 • 使用一个学习能力差一点的算法
训练和测试效果好，实际应用效果差	• 采用更加真实的数据
模型无法学习数据	• 重新选择特征，使特征更能表达数据的不变特性 • 采集更新，更相关的数据 • 选一个学习能力更强一些的拟合算法

交叉验证、bootstrapping、ROC曲线和混淆矩阵

最后介绍一些用来评估机器学习结果的工具。在监督学习中，如何了解算法的性能是一个最基本的问题。分类器在分类或拟合数据时准确么？你可能会说：“简单，我只需要在测试集或者验证集上运行一下，就可以获得结果。”但是在实际情况中，我们必须考虑噪声，采样误差和采样错误等因素。测试集或验证集可能并不能精确地反映数据的真实分布。为了更准确地评估分类器性能，我们可以采用交叉验证（cross-validation）的技术或者与之比较相近的*bootstrapping*。[注11]

回想一下最基本的形式，交叉验证首先把数据分为k个不同的子集。然后用$k-1$个子集进行训练，用没有用来训练的子集进行测试。这样做k次（每个子集都有一次机会做测试集），然后把结果平均。

bootstrapping跟交叉验证法类似，但是验证集是从训练集中随机选取的。选择的点仅用

注11：有关这些技术的更多信息，请参阅“交叉验证和引导”（*http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-12.html*）。

于测试，不在训练中使用。这样做*N*次，每次随机选择一些验证集，最后把得到的结果平均。这意味着一些数据样本会出现在不同的验证集中，bootstrapping的效果一般胜于交叉验证。[注12]

使用其中的任意一种技术都可以使实际效果的评估更准确，更高精确度又可以用来指导学习系统的参数调整。再训练，再评估，再调整，如此反复。

在评估和调整分类器的方法中，还有两个比较有用的方法，分别是画ROC曲线和填充混淆矩阵，如图20-4所示。ROC曲线评估了分类器参数变换对分类器性能的影响（ROC曲线上的每个点都可以使用交叉验证 ）。假设这个参数是一个阈值。更具体一点，假设我们需要在图中识别黄色，探测器使用一个阈值来定义黄色。黄色阈值设置得非常高，意味这我们的分类器错误识别率为0，同时正确识别率也为0。（如图20-4左下角的曲线）另一方面，如果黄色阈值设置得非常低，则意味着所有的信号，不管是黄色的花，还是红色或者橙色的花都会被识别成黄色。分类器的错误识别率为100%，同时正确的识别率为100%（图20-4的右上角）。最好的ROC曲线是从原点沿*y*轴上升到100%，在沿*x*轴到达右上角的折线。如果做不到这样，那么，曲线越靠近左上角越好。我们可以计算ROC曲线下方的面积和整个ROC图的面积之比，越接近于1，就表明分类器越好。[注13]

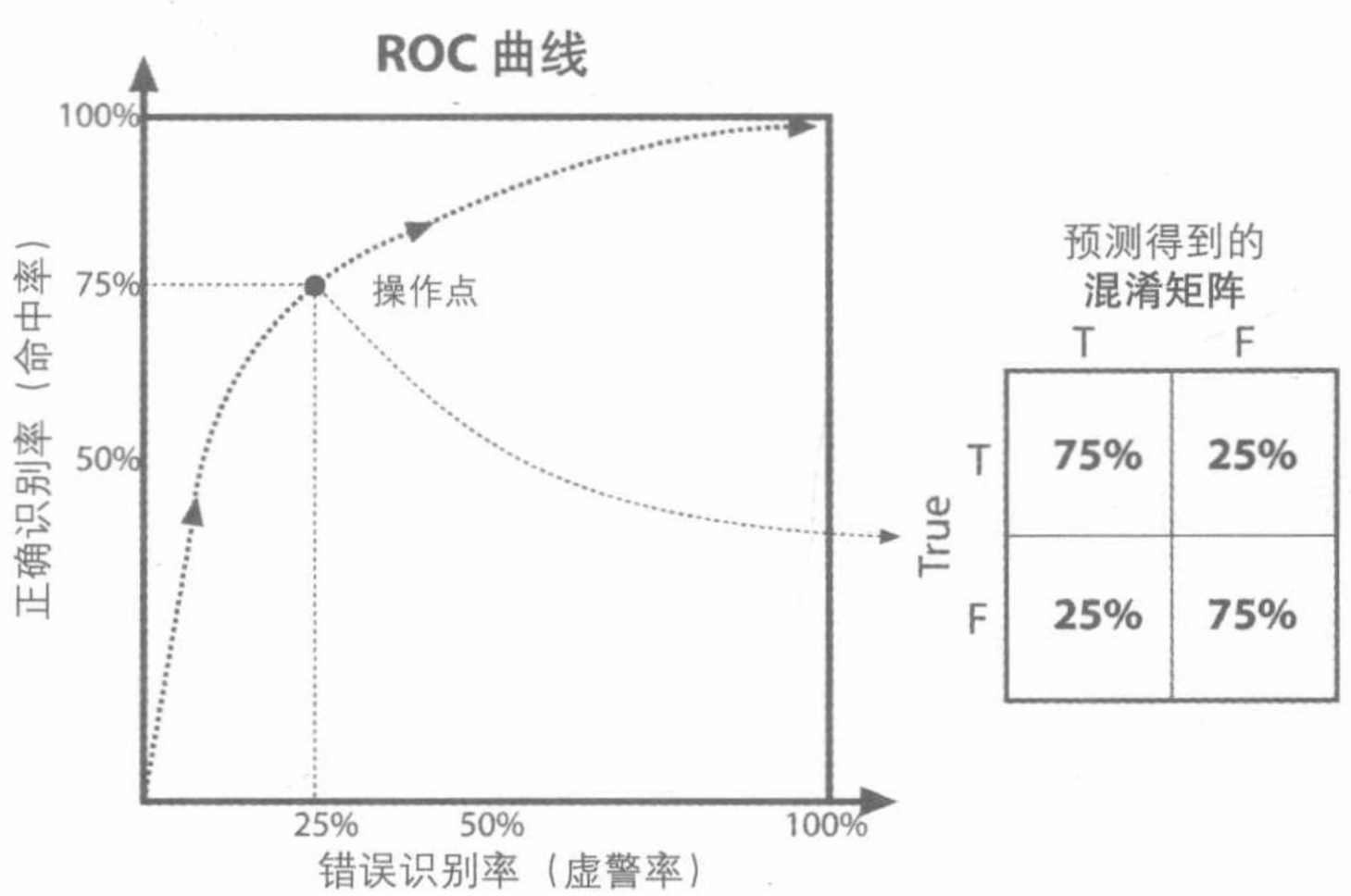


图20-4. ROC曲线和与之相关的混淆矩阵，ROC曲线显示分类器参数在所有的取值范围上变化时，正确识别率和错误识别率的关系；混淆矩阵显示错误识别率和错误拒绝率的关系

注12：引导的主要目标之一是找到最佳的训练参数，因为虽然平均训练误差容易，但将几个模型“平均”成一个模型可能是不常见的和/或低效的。

注13：值得注意的是，字面上分类器的“品质因素”是无穷无尽的，还有许多理由倾向于另一种。然而，这种特定的内容比较常见，但并不是因为它一定能对分类器质量有更好的表现，而是因为它对于几乎任何分类问题而言都容易理解，并且具有良好的定义。

图20-4还展示了一个混淆矩阵，这是一个包含错误识别率、正确识别率、错误拒绝率及正确拒绝率的图。这是评估分类器性能的快速方法：理想情况下，我们将看到左上-右下的对角线上的两个值是100%，而其他的两个值是0%。如果我们的分类器可以处理多类问题（多层感知神经网络和随机森林），那么混淆矩阵将推广到多类，你需要跟踪每一个带标签的数据样本所被分配的类别。

分错类的代价。下面我们讨论分错类的代价。假设我们要设计分类器来检测毒蘑菇，我们希望错误拒绝率（可食用的蘑菇被认为有毒）很大，错误识别率（有毒的蘑菇却被误认为可食用）很小。ROC曲线可以处理这种情况：我们设置ROC参数使操作点下降。还有另一种方法是生成ROC曲线的时候对错误识别赋更大的权值。可以把每个错误识别的权值设置为错误拒绝权值的10倍。[注14]一些OpenCV算法（如决策树和SVM），能够通过指定类别的先验概率或者指定个别的训练样本的权重来自动平衡“命中率和虚警率”。

特征方差不一致。另一个训练分类器的问题是特征向量中特征的方差不一致。例如，如果一个特征是由小写的ASCII码字母表示的，那么它最多有26个值。与此不同的是，显微镜载物片上的生物细胞数是以亿计的。类似*K*-近邻算法的一类算法会认为第一个特征相比于第二个特征可以忽略。解决这个问题的方法是预处理每个特征变量，使其方差一致。如果特征不相关，这个步骤很重要，如果特征相关，你可以用它们的协方差或者平均方差来归一化。决策树[注15]等一类算法不受方差是否一致的影响，所以不需要预先处理。如果算法以距离度量为准则，就需要预先将方差归一化。我们可以利用马氏距离来归一化特征方差。

接下来介绍一些OpenCV中支持的机器学习算法。

ML库中遗留的机器学习算法

许多ML库使用通用的C++接口，这些接口是面向对象的，对象中每个算法都是从一个基类中派生的。通过ML库，用户可以对其中的算法进行标准化的使用和访问。然而，OpenCV库中仍有一些最基本的方法不符合这个标准接口，因为这些方法是在通用接口尚未设计时就已经实现了的。[注16]由于一些最基本的算法并没有采用面向对象的接口，所以本章先介绍这些算法。在下一章中，将介绍面向对象的接口和通过它实现的许多算

注14：如果有一些具体的，两个错误类型的相对成本的先验概念，会很有用。例如，遇到在超市结账，将一个产品错误分类为另一个产品的成本是很容易事先量化的。

注15：决策树不受特征变量方差差异的影响，因为每个变量仅针对有效的分离阈值进行搜索。换句话说，只要可以找到明确的分离值，变量的范围有多大无关紧要。

注16：给这些旧功能写入新接口，能确保这些功能符合MLL的其他部分使用的基于对象的风格。但在当时，对一些人来说，这只是OpenCV库要做的一个开源项目。

法。下面将介绍的方法是*K*-means聚类，马氏距离及其在*K*-means聚类中的作用，在最后还要介绍一种可以提高速度及*K*-means聚类的速度和精度的分割技术。

K均值

K-means（K均值）尝试找到数据的自然类别。用户设置类别个数，*K*-means迅速找到“好的”类别中心。“好的”意味着聚类中心位于数据的自然类别中心。*K*-means是最常用的聚类方式与EM（期望最大化）算法[注17]很相似，也与均值漂移算法（在CV库中实现为cv::MeanShift()）相似。*K*-means是一个迭代算法，在OpenCV中采用的是Lloyd算法，也叫Voronoi迭代。算法流程如下。

1. 输入数据集合和类别数K（由用户指定）。
2. 随机分配类别中心点的位置。
3. 将每个点放入离它最近的类别中心点所在的集合。
4. 移动类别中心点到它所在集合的中心。
5. 转到第3步，直到收敛。

图20-5展示了*K*-means是怎么工作的。在这个例子中，只用两次迭代就达到了收敛。在现实数据中，算法经常很快收敛，但有的时候需要迭代很多次。

问题和解决方案

K-means是一个极其高效的聚类算法，但是它存在以下三个问题。

- 它不保证能找到定位聚类中心的最佳方案，但它能保证收敛到某个解决方案（例如迭代不再无限继续下去）。
- *K*-means无法指出应该使用多少类别。在同一数据集中，例如对图20-5中的例子，选择两个类别和选择四个类别，得到的结果不一样，甚至是不合理的。
- *K*-means假设空间的协方差矩阵不会影响结果或者已经归一化（参考稍后对马氏距离的讨论）。

这三个问题都有“解决办法”。这些解决方法中最好的两种都是基于“解释数据的方差”。在*K*-means中，每个聚类中都有其数据点，我们计算这些点的方差。[注18]

注17：具体来说，*K*-means类似于高斯混合模型的EM算法。该EM算法也在OpenCV（作为ML库中的cv::EM()中实现）。本章稍后将提到此算法。

注18：在这种情况下，点的方差是该点与聚类中心的距离。点（多个）的方差通常是这些点上的正交和。这个总和也叫*compactness*（紧凑度）。

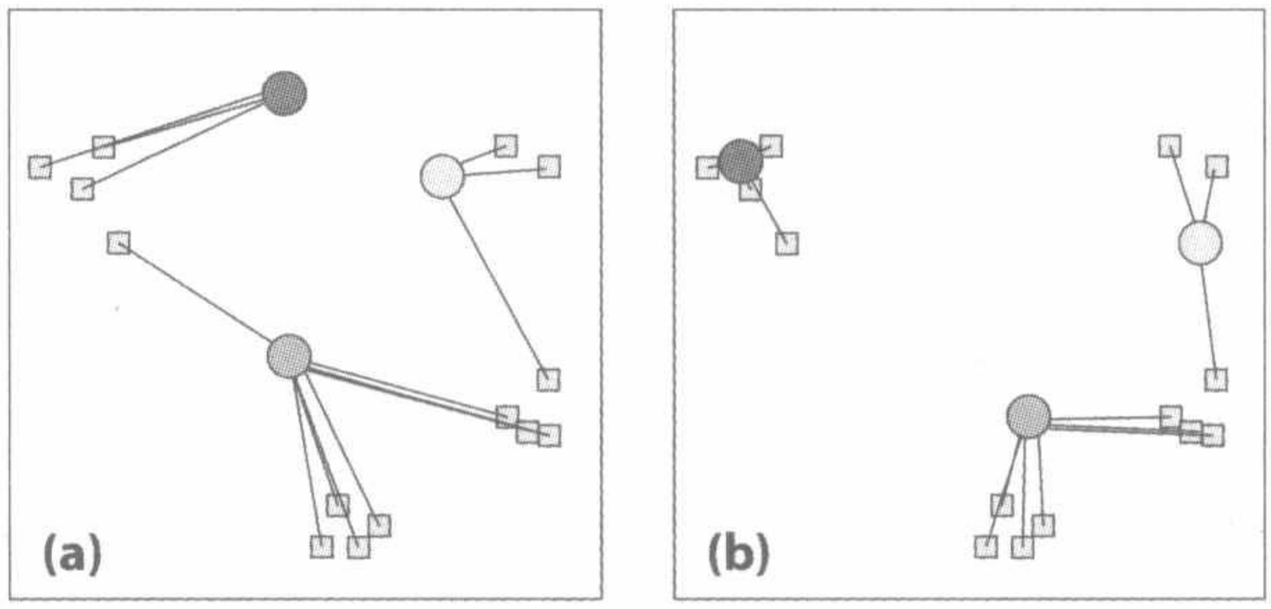


图20-5. *k*-means算法的迭代中有两步：(a)随机放聚类中心然后将样本聚到离它最近的中心；(b)数据中心移动到它所在类别的中心；(c)数据点根据最邻近规则重新聚到类别中心；(d)聚类中心再次移动到它所在的类别中心

最好的聚类能在不引起太大的复杂度的情况下使方差达到最小。所以，列出的问题可以改进如下。

1. 多进行几次*K*-means，每次初始的聚类中心点都不一样（很容易做到，因为OpenCV随机选择中心点），最后选择方差最小的那个结果。
2. 首先将类别数设为1，然后提高类别数（到达某个上限），每次聚类的时候使用前面提到的方法1。一般情况下，总方差会很快下降，直到到达一个拐点；这意味着再加一个新的聚类中心不会显著减少总方差。在拐点处停止，保存此时的类别数。
3. 将数据乘以逆协方差矩阵。例如：如果输入向量是按行排列，每行一个数据点，则通过计算新的数据向量D^*，$D^* = D\Sigma^{-1/2}$来归一化数据空间。

*K*均值代码

K-means的调用示例如下：

```
double cv::kmeans(                                  // returns (best) compactness
cv::InputArray       data,                          // Your data, in a float type
int                  K,                             // Number of clusters
cv::InputOutputArray bestLabels,                    // Result cluster indices (int's)
cv::TermCriteria     criteria,                      // iterations and/or min dist
int                  attempts,                      // starts to search for best fit
```

```
    int                 flags,                      // initialization options
    cv::OutputArray     centers   = cv::noArray() // (optional) found centers
    );
```

该数据结构是一个多维数据点构成的矩阵，每行一个数据样本，数据的每个元素都是浮点类型（即CV_32FC1）。此外，数据还可以是简单的向量，其中每一列是一个多维点（即，CV_32FC2或CV_32FC3或CV_32FC(M)）。[注19]参数*K*是要查找的聚类数，返回向量best标签包含每个数据点的最终聚类索引。在标准情况下，可以指定要运行算法的最大迭代次数，或指定一个最小距离，当两次迭代之间聚类中心移动的距离小于给定的最小距离时，停止迭代。当然，也可以同时使用这两个标准。

参数attempts可以设定cv::kmeans()自动运行若干次，每次从一组新的种子点开始，并记录最佳结果。运行的结果好坏由紧凑度（即与该点相关联的聚类中每个点和中心之间的距离平方和）来衡量。

flags参数可以是以下任何值（第一个为默认值）：cv::KMEANS_RANDOM_CENTERS, cv:: KMEANS_USE_INITIAL_LABELS或cv::KMEANS_PP_CENTERS。当选用cv::KMEANS_RANDOM_CENTERS时，就在数据集中随机选取聚类中心的起始位置。当选用cv::KMEANS_USE_INITIAL_LABELS时，就调用存储在参数bestLabels中的值将它用于计算初始聚类中心。而cv::KMEANS_PP_CENTERS选项则会使cv::kmeans()使用Sergei Vassilvitskii即被David Arthur [Arthur07]称为*K*-means++的方法来分配聚类中心。这种方法更谨慎地选择了聚类中心的起点，通常能以少于默认方法的迭代得出更好的结果。在现代应用中，*K*-means++正在逐渐成为应用标准方法。



在理论上，*K*-means算法可能会得到很差的结果，但事实上它却被大量使用，因为它在多数情况下都效果不错。在最坏的情况下，聚类问题是一个NP问题，即没有一个真正的最佳答案，但对于应用程序来说，有一个“好”的答案已经足够了。此外，*K*-means聚类算法还存在其他问题。其中之一就是，在某些情况下，它可能产生所称的数学家称之为“任意坏”的结果。这意味着，无论你觉得当前的情况有多糟糕，总有人可以得到更糟糕的情况。在过去几年中也产生了一些改进性能的技术，在最佳情况下，可以通过一些可证明的界限让用户对预期结果有更多信心，例如*K*-means++。

最后，当计算完成后，聚类的中心将放置到矩阵中心。如果不需要，可以忽略中心（在这种情况下，通过传递cv::noArray()来实现）。

注19：回想一下，这种情况实际上完全等同于$N \times M$矩阵，其中N行是数据点，M列是每个点所在位置的各个组件，底层数据类型是cv::32FC1。其实仔细一想，由于用于数组的内存布局，这些表示之间其实是没有区别的。

该函数总是返回计算的紧凑度。由*K*-means算法的完整示例（示例20-1）可以得到一些启发。该示例的好处在于数据生成部分也可以用于测试其他机器学习程序。

示例20-1：使用*K*-means算法

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/core/core.hpp"
#include <iostream>

using namespace cv;
using namespace std;

static void help( char* argv[] ) {
  cout << "\nThis program demonstrates kmeans clustering.\n"
   "  It generates an image with random points, then assigns a random number\n"
   "  of cluster centers and uses kmeans to move those cluster centers to their\n"
   "  representative location\n"
   "Usage:\n"
   <<argv[0] <<"\n" << endl;
}

int main( int /*argc*/, char** /*argv*/ ) {

  const int MAX_CLUSTERS = 5;
  cv::Scalar colorTab[] = {
    cv::Scalar(   0,   0, 255 ),
    cv::Scalar(   0, 255,   0 ),
    cv::Scalar( 255, 100, 100 ),
    cv::Scalar( 255,   0, 255 ),
    cv::Scalar(   0, 255, 255 )
};

cv::Mat img( 500, 500, CV_8UC3 );
cv::RNG rng( 12345 );

for(;;) {

  int k, clusterCount = rng.uniform(2, MAX_CLUSTERS+1);
  int i, sampleCount = rng.uniform(1, 1001);
  cv::Mat points(sampleCount, 1, CV_32FC2), labels;

  clusterCount = MIN(clusterCount, sampleCount);
  cv::Mat centers(clusterCount, 1, points.type());

  /* generate random sample from multigaussian distribution */
  for( k = 0; k < clusterCount; k++ ) {
    cv::Point center;
    center.x = rng.uniform(0, img.cols);
    center.y = rng.uniform(0, img.rows);
    cv::Mat pointChunk = points.rowRange(
      k*sampleCount/clusterCount,
      k == clusterCount - 1 ? sampleCount : (k+1)*sampleCount/clusterCount
    );
    rng.fill(
      pointChunk,
```

```
      RNG::NORMAL,
      cv::Scalar(center.x, center.y),
      cv::Scalar(img.cols*0.05, img.rows*0.05)
    );
}

randShuffle(points, 1, &rng);

kmeans(
  points,
  clusterCount,
  labels,
  cv::TermCriteria(
    cv::TermCriteria::EPS | cv::TermCriteria::COUNT,
    10,
    1.0
  ),
  3,
  KMEANS_PP_CENTERS,
  centers
);

img = Scalar::all(0);

for( i = 0; i < sampleCount; i++ ) {
  int clusterIdx = labels.at<int>(i);
  cv::Point ipt = points.at<cv::Point2f>(i);
  cv::circle( img, ipt, 2, colorTab[clusterIdx], cv::FILLED, cv::LINE_AA );
    }

    cv::imshow("clusters", img);

    char key = (char)waitKey();
    if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC'
      break;
  }

  return 0;
}
```

在这段代码中，包含highgui来使用窗口输出，又因为它包含cv::kmeans()，所以还包含*core.hpp*。[注20]在main函数中，我们设置了类别显示的颜色；程序的基本操作是，首先选择一些要生成的聚类，为这些聚类生成中心，然后在生成的中心周围生成点云。之后，我们再来看看cv::kmeans()是否可以重新发现我们放入样本数据中的这个结构。在main()中，我们首先做一些小的内部处理，比如设置稍后用于显示聚类的着色。然后设置一个主循环，用户可以控制程序一遍又一遍地运行，生成不同的测试数据集。

该循环首先确定底层数据中将存在多少个聚类以及将生成多少数据点。然后对于每个聚

注20：这是传统问题的一个例子。cv::kmeans()早于ML库的正式创建，所以它的原型在*core.hpp*而不是*ml.hpp*（你可能会这样想）。

类，生成中心，并且从围绕该中心按照高斯分布生成数据点。之后随机打乱数据点，使其不再属于原来的聚类。

之后把*K*-means聚类算法应用在数据上。在这个例子中，我们并不搜索可能的存在聚类数，我们只是告诉cv::kmeans()有多少个集群。计算得到的标签，然后将它放在labels中。

最后用for{}循环画出结果，在图中显示。然后释放数组，等待（ cv::waitKey(0)）用户的输入进入下一次计算或者Esc键退出。

马氏距离

我们在第5章用马氏距离衡量点与对分布形状敏感的分布中心之间的距离。在*K*-means算法的场景中，马氏距离常有以下两种用法。马氏距离可以视为在拉伸的空间上测量欧几里得距离。这使得我们可以使用马氏距离来对数据进行重新定标，这可以显著提高*K*-means算法的性能。此外，通过计算马氏距离，还可以将新的数据点分配给由K-means算法定义的聚类。

使用马氏距离来约束输入数据

在示例20-1中，我们将简要介绍一种可能存在的情况——数据在空间中以非常不对称的方式分布。而*K*-means的核心观点是数据是以不均匀的方式聚类的，该算法试图挖掘一些与这些聚类有关的信息。然而，“不对称”和“不均匀”之间存在着重要的区别。例如，如果所有的数据在某个维度上分散很远，而在其他维度上距离相对较小，则*K*-means算法不会有很好的效果，如图20-6所示。

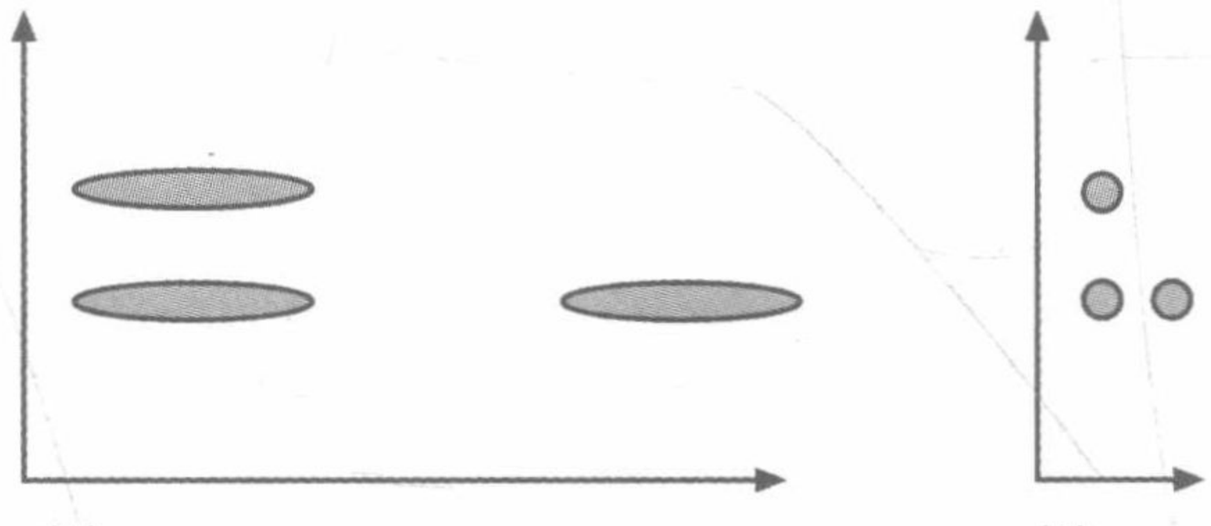


图20-6. 马氏距离可以使我们理解数据的协方差为数据空间的拉伸：(A)，原始数据的垂直距离比水平距离小；(B)通过对空间进行方差归一化，数据之间的水平距离小于垂直距离

这种情况往往只是因为数据向量在不同维度有不同的底层单位。例如，如果用身高、年龄和学龄代表一个人，那么由于身高和年龄基本单位的不同，数据在不同维度上会产生不同的分布。同样，年龄和学龄虽然有相同的单位，但在自然人口中也存在着差异。

然而，有一种简单的技术也许可以解决问题。该技术是将数据集作为一个整体来看待，并计算整个数据集的协方差矩阵。我们可以使用它重新调整整个数据集。这种技术将重新分配图20-6(A)中的数据，使之更接近于图20-6(B)。

回想一下，本书第5章曾介绍了马氏距离的函数，传统意义上，马氏距离是以该点在特定方向上的方差来度量数据点与分布之间距离。（马氏距离是将Z-score推广到多维空间）我们使用分布的协方差的倒数计算马氏距离：

$$\Sigma_{i,j} = E[(\vec{X}_i - \vec{\mu})(\vec{X}_i - \vec{\mu})^T] = \frac{1}{N}\sum_i[(\vec{X}_i - \vec{\mu})(\vec{X}_i - \vec{\mu})^T]$$

其中，$E[\cdot]$表示数学期望，马氏距离实际可以表示为：

$$D_{mahalanobis}(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T \Sigma^{-1}(\vec{x} - \vec{y})}$$

对此，可以有以下两种方式解决：在*K*-means算法中使用马氏距离而不是欧式距离，或者首先将数据进行尺度变换，然后在缩放后的空间中使用欧几里得距离。第一种方法更为直观，但第二种方法更容易计算，因为不必修改已经封装好的、较为完备的K-means算法。而且数据转换是线性的，可实现的。

显然，可以通过以下操作对数据进行重新调整：

$$D^* = D\Sigma^{-1/2}$$

这里D^*是即将使用的一组新的数据向量，D是原始数据。$\Sigma^{-1/2}$的因子是逆协方差的平方根。[注21]

在这种情况下，实际上并不直接使用`cv::Mahalanobis()`。相反，我们使用`cv::calcCovarMatrix()`计算集合协方差，使用`cv::invert()`（使用`cv::DECOMP_EIG`[注22]）。因此，`DECOMP_EIG`通常是此任务的最佳选择。）进行逆变换，最后计算平方根。[注23]

注21：为什么在这个公式中右边的逆协方差是正确的？因为ML库中的约定是将数据集D表示为每个点的N行和M列，因此，数据是行而不是列。

注22：在这种情况下，也可以使用`DECOMP_SVD`，但它比`DECOMP_EIG`稍微慢一些，不太准确。`DECOMP_EIG`，即使它比`DECOMP_LU`慢，但如果空间的维度远小于数据点的数量，那么仍然应该使用。在这种情况下，总体计算时间将由`cv::calcCovarMatrix()`控制。因此，花更多的时间更精确地计算逆协方差矩阵，这样做可能更明智。更准确地说，如果该集合集中在较小维度的子空间中。

注23：库中没有通用的函数来计算这个平方根，但是由于矩阵∑^{-1}的特性非常合适，平方根可以通过这样先对角化来计算，然后取特征值的平方根，后使用之前已经对角化的同一个特征向量旋转回原始坐标系。所以这种技术被称为“对角化方法”。

利用马氏距离进行分类

利用*K*-means聚类或任何其他方法给定一组聚类标签，利用这些标签可以预测一些新的数据点最可能属于哪个聚类。假设聚类本身服从高斯分布，将马氏距离的概念引入该分类问题也是非常意义的。

首先根据其平均协方差来表示每个聚类。如此一来，就可以计算任意一个新的数据点到每个聚类中心的马氏距离。

也许你会认为具有最小马氏距离的点将是最后的赢家，但事实并非如此。当且仅当所有聚类都拥有相同数量的数据点时（每个集群内数据点的先验概率大小相等），此结论才为真。[注24]

贝叶斯规则简明扼要地说明了这一区别，对于两个命题*A*和*B*，给定*B*情况下，*A*发生的概率（一般情况下）不等于给定*A*的情况下*B*发生的概率。如下式所示：

$$P(A|B) \neq P(B|A)$$

相反，在给定*A*情况下*B*发生的概率，乘以*A*的自然概率等于给定*B*的情况下*A*发生的概率乘以*B*的自然概率。如下式所示：

$$P(A|B)P(B) = P(B|A)P(A)$$

那么这与马氏距离有什么关系呢？马氏距离表示一个特定的样本来自一个特定的聚类的概率。用不同的方式去看问题，我们想知道当给定值为$\vec{x}$的点，该点出现在聚类*C*中的概率，而马氏距离告诉我们是恰好相反的，即给定聚类C中，求$\vec{x}$在其中的概率。这里给出一个方程（贝叶斯规则）可用于相互转换：

$$P(C|\vec{x}) \propto \left(\frac{N_c}{N_D}\right)\left(|\Sigma|^{-1/2} e^{-\frac{1}{2}r_M^2}\right)$$

这意味着为了比较两个不同聚类之间的两个马氏距离，我们应该考虑聚类的大小。考虑到每个聚类的概率服从高斯分布，聚类之间的似然性可以如下表示为：

$$P(C|\vec{x}) = \frac{P(C)}{P(\vec{x})} P(\vec{x}|C)$$

在该方程式中，N_c除以N_D得到的是聚类*C*中的数据点相对于整个数据集中数据点的比

注24：问题就好像是，发现一只看似恐龙的极其奇怪的蜥蜴的可能性远远大于找到相对正常的恐龙的可能性。这是因为真的恐龙比蜥蜴更为罕见。

率。该比率是聚类*C*的先验概率。第二项包含聚类*C*的协方差的行列式的平方根和包含马氏距离平方的指数因子。

小结

本章讨论什么是机器学习，并让读者初步了解了OpenCV库中哪些部分是用来解决大数据空间问题的，哪些不是。本章还阐述了训练数据和测试数据之间的区别。生成式模型是在没有监督或标记的训练数据的情况下从现有数据中找到结构的模型，而判别式模型是从示例中学习并从其中概括出来的模型。此外，本章介绍了OpenCV库中提供的两个非常基本的工具：*K*-means聚类和马氏距离。它们可以用来构建简单的模型并解决有趣的问题。OpenCV现在也支持深层神经网络，包含在*openCV_contrib*中。代码在附录B（*cnn_3dobj*和*dnn*）中。

练习

1. 如果每个数据点的特征在范围上有很大变化（比如说第一个特征在1到100之间，第二个特征在0.0001到0.0002之间），则说明在对该数据集使用以下算法时是否会出现一些问题，说明为什么。

   a. 支持向量机

   b. 决策树

   c. 反向传播算法

2. 数据归一化可以消除尺度差异。常见的有两种做法。第一种用原始数据除以其标准差，第二种用原始数据除以最大值与最小值的差。对于每种归一化方法，请分别设计一组数据，令一种方法在此数据上能取得较好的效果，同时另一种方法的效果却很差。

3. 考虑在使用K-means算法之前，先使用马氏距离重新调整输入数据集的情况。证明根据公式$D^* = D\Sigma^{-1/2}$预分解数据完全等效于修改算法（在内部使用马氏距离）。

4. 尝试从股票的历史价格中学习未来的股票价格。假设你有20年的股票数据。讨论将数据转化为训练和测试数据集的各种方法及其影响，并说明以下方法的优缺点。

   a. 将编号为偶数的数据作为训练集和奇数的作为测试集。

   b. 将数据集随机分为训练和测试集。

   c. 将数据分为两部分，前半部分用于训练，后半部分用于测试。

5. 将数据集划分成许多模块，代表训练集和一个测试集，参见图20-3。请说明是否存在这样一种情况：算法在测试集上的错误率低于在训练集上的错误率。

6. 图20-3描述了一个回归问题。标记图A上的第一点，第二点B，第三点A，第四点B等。在两个类（A和B）之间绘制一条分隔线，表示：

   a. 偏差

   b. 方差

7. 参见图20-4。

   a. 绘制量通用的ROC曲线。

   b. 绘制最不通用的ROC曲线。

8. 绘制一个在测试数据上随机执行的分类器的曲线。

9. 考虑变量的重要性。

   a. 如果两个特征完全一样，是否可以根据之前描述的变量重要性分辨哪个特征更为重要？

   b. 如果不能，有没有什么其他途径可以分辨出这两个特征哪个比较重要，或两者都重要？

第21章

StatModel：OpenCV中的基准学习模型

在上一章中，我们简要介绍了机器学习，并展示了很久之前已经在库中实现的基本算法。在本章中，将介绍几种应用广泛的现代技术。然而，在此之前，本章先介绍cv::ml::StatModel，它是在本章中所有更高级算法的接口实现的基础。在读者了解cv::ml::StatModel之后，本章的其余部分将按照被引入计算机视觉社区的时间介绍OpenCV库中提供的各种学习算法。[注1]

ML库中的常见例程

现在ML库中的例程都是在公共基类cv::ml::StatModel的派生类中实现的。这个基类定义了可用于所有算法的通用接口。其中一些方法在基类cv::Algorithm中声明，cv::ml::StatModel本身也是从这个基类派生的。下面就是cv::ml::StatModel基类在机器学习（ML）库中的定义：

```
// Somewhere above...
// namespace cv {
//   namespace ml {

class StatModel : public cv::Algorithm {

public:

    /** Predict options */
    enum Flags {
```

注1：请注意，OpenCV目前正在进行扩展以支持深层神经网络（DNN），详情参见附录B代码库*cnn_3dobj*和*dnn*。在写本书（2015年）时，DNN正在日渐成为计算机视觉的重要工具。然而，OpenCV仍然在一个不断发展变化的过程中，所以这里不再赘述。

```
        UPDATE_MODEL        = 1,
        RAW_OUTPUT          = 1,
        COMPRESSED_INPUT    = 2,
        PREPROCESSED_INPUT = 4
    };

    virtual int  getVarCount() const  = 0; // number training samples
    virtual bool empty() const;            // true if no data loaded

    virtual bool isTrained()    const = 0; // true if the model is trained
    virtual bool isClassifier() const = 0;  // true if the model is a classifier

    virtual bool train(
      const cv::Ptr<cv::ml::TrainData>& trainData,     // data to be loaded
      int                               flags    = 0    // (depends on model)
    );

    // Trains the statistical model
    //
    virtual bool train(
      InputArray samples,  // training samples
      int        layout,   // layout See ml::SampleTypes
      InputArray responses // responses associated with the training samples
    );

    // Predicts response(s) for the provided sample(s)
    //
    virtual float predict(
      InputArray  samples,                    // input samples, float matrix
      OutputArray results = cv::noArray(),  // optional output results matrix
      int         flags   = 0                 // (model-dependent)
    ) const = 0;

    // Computes error on the training or test dataset
    //
    virtual float calcError(
      const Ptr<TrainData>& data, // training samples
      bool test,                  // true:   compute over test set
                                  //  false: compute over training set
      cv::OutputArray resp        // the optional output responses
    ) const;

    // In addition, each class must implement static `create()` method with no
    // parameters or with all default parameter values.
        //
        // example:
        // static Ptr<SVM> SVM::create();
    };
```

显然，cv::ml::StatModel继承自cv::Algorithm。虽然本章不会讲述该类中的所有算法，但对于可能出现在cv::ml::StatModel场景中的重要的一些算法，还是有必要介绍一下其用法：

```
// Somewhere above
// namespace cv:: {
//   namespace ml:: {

class Algorithm {
...
public:

  virtual void save(
   const String& filename
  ) const;

  // Calling example: Ptr<SVM> svm = Algorithm::load<SVM>("my_svm_model.xml");
  //
  template<typename _Tp> static Ptr<_Tp> static load(
   const String& filename,
   const String& objname = String()
  );

  virtual void clear();
...
}
```

cv::StatModel方法提供了一种方法可以从磁盘读取和写入训练模型以及清除模型中的数据。以上三种都是通用的重要操作。[注2]但是训练模型的算法和将其应用于预测的算法至少在接口方面是不同的。不同算法的训练和预测能力均不相同，参数自然不同。

训练方法和cv::ml::TrainData的结构

训练和预测方法都包含在cv::ml::StatModel中。在本节中，我们将介绍这些函数方法的构造和用法。

回想一下，在cv::ml::StatModel原型中有两个名为train()的方法。第一种train()方法通过cv::ml::TrainData结构体指针获取数据和各种算法依赖的训练参数标志。第二种方法是利用提供的样本和数据标记直接构建与训练数据结构相同的快捷方式。正如所见，cv::ml::TrainData接口使训练模型更具表现力，通过它，用户可以更有效地准备数据。

cv::ml::TrainData的构造

cv::ml::TrainData类允许用户将数据打包，并对如何在训练过程中解包并应用数据做了一些说明。下面介绍create()方法，它可以用于生成新的cv::ml::TrainData对象。

注2：在以前的版本中，有两对读/写函数：save()/load()和write()/read()，后一对是与现在传统CvFileStorage文件接口交互的低级函数结构体。这一对函数连同它们所对应的结构都应被废弃，现在代码中只应该使用save()/load()这对新的接口。

```
// Construct training data from the specified matrix of data points and responses.
// It's possible to use a subset of features (a.k.a. variables) and/or subset of
// samples; it's possible to assign weights to individual samples.
//
static cv::Ptr<cv::ml::TrainData> cv::ml::TrainData::create(
  cv::InputArray samples,                          // Array of samples (CV_32F)
  int            layout,                           // row/col (see ml::SampleTypes)
  cv::InputArray responses,                        // Float array of responses
  cv::Inputarray varIdx        = cv::noArray(), // Specifies training variables
  cv::InputArray sampleIdx     = cv::noArray(), // Specifies training samples
  cv::InputArray sampleWeights = cv::noArray(), // Optional sample wts (CV_32F)
  cv::InputArray varType       = cv::noArray()  // Optional, types for each
                                                //  input and output
                                                //  variable (CV_8U)
);
```

该方法根据训练样本的预分配阵列和相关响应构建训练数据。样本矩阵必须是CV_32FC1类型（32位、浮点、单通道）。虽然cv::Mat类能够表示多通道图像，但是机器学习算法只需要一个通道，也就是二维数组。通常，在阵列中数据点按行排列，每个数据点代表一个特征向量。所以，每列存放的显然就是每个数据点的某个特定特征，数据点堆叠起来构成二维单通道训练矩阵。即，典型的数据矩阵：（行，列）=（数据点，特征）。

一些算法可以直接处理转置矩阵。由参数确定数据的存储方式：

layout = cv::ml::ROW_SAMPLE

意味着特征向量按行存储（这是最常见的布局）

layout = cv::ml::COL_SAMPLE

意味着特征向量按列存储

你可能会问："如果训练数据不是浮点数，而是字母表中的字母或是代表音符和植物名称的整数呢？"答案很简单，只要把它们转化为唯一的32位浮点数再填充进cv::Mat。如果有字母作为特征或标签，则可以在填充数据数组时将字符的ASCII码转换为浮点数。这种方法同样适用于整数。只要转换是一对一的就不会出现问题。但是某些算法对于特征之间差异范围很敏感，所以最好如上一节所述，将特征方差归一化。基于树的算法（决策树、随机树和boosting）能够处理按照类别或者一定顺序输入的数据，但其他OpenCV中的机器算法仅支持有序输入。现在出现了一些流行的技术，通过以"1-radix"或"1-hot"符号表示数据的方式，可以同时处理使有序输入的数据和按类别输入的数据。例如，如果输入变量可能有七个不同的颜色值，就可以用七个二进制变量表示它，其中只有一个变量可以设置为1。[注3]

注3：请注意，这不同于一个值为二进制编码的输入。值$0100000b = 32$与值为[0, 1, 0, 0, 0, 0, 0]的七维输入向量非常不同。

响应参数包含分类标签，例如在分类蘑菇时的有毒或无毒，或者是用温度计获取的回归值（数字），例如体温。数据点的响应值或“标签”通常保存在一个一维向量中。然而仍有例外，比如神经网络。每个数据点都有与之对应的响应向量。对于分类问题，响应值类型必须是整数（CV_32SC1）；对于回归问题，响应为32位浮点型（CV_32FC1）。在神经网络中，如前所述，通常会对此进行一些变通，使用回归框架进行分类。在这种情况下，前面提到的1-hot编码用于表示各种类别，并且用浮点数表示多个输出。网络本质上是被训练来计算输入回归到每个类别的概率。

回想一下，一些算法只能处理分类问题，另一些算法只能处理回归问题，但仍有一些算法能同时处理这两种问题。在最后一种情况下，输出变量的类型可以作为单独的参数传递，也可以通过var Type向量传递。该向量可以是单列还是单行，并且必须是CV_8UC1或cv::S8C1类型。varType中的条目数等于输入变量的数量（N_f）加上响应数（通常为1）。[注4]第一个项将表示对应的输入特征的类型，而其他的则表示输出的类型。varType中的每个条目都应设置为以下的值之一：

cv::ml::VAR_CATEGORICAL
: 意味着输出值是离散类标签

cv::ml::VAR_ORDERED (= cv::ml::VAR_NUMERICAL)
: 意味着输出值是有序的，也就是说，不同的值可以作为数字进行比较，因此这是一个回归问题



回归算法只能处理有序输入变量。而有时如果分类变量有序，也可以对分类标签进行排序，但是可以这样做有时候会导致回归的困难，因为当强加的顺序没有实际的意义时，假装“有序”的值可能会在很大范围内变动。

ML库中的许多模型可以在所选择的特征集或训练集中选定样本的子集上进行训练。为了方便用户使用，cv::ml::TrainData::create()方法包含了向量varIdx和sampleIdx作为参数。varIdx向量用于标识感兴趣的特定变量（特征），而sampleIdx则标识特定的感兴趣的数据点。而这些也可以简单省略或设置为cv::noArray()（默认值），表示使用“所有特征”或“所有点”。两个向量都提供从零开始的索引，并可以作为活动变量/样本的掩码，其中非零值表示活动。在前一种情况下，向量必须是CV_32SC1类型，长度任意。而在后一种情况下，向量必须是CV_8UC1类型，且必须具有与特征或样本相同的长度

注4：要清楚，这意味着输入要素的数量，而不是输入数据点的数量。

（如适用）。当读入大量数据时，功能强大的sampleIdx参数允许用户无需先将数据集其分解成两个不同的向量，就可以实现用其中的一些作为训练集，另一些作为测试集。

从存储的数据中构建cv::ml::TrainData

通常，如果磁盘中的数据是CSV格式（逗号分隔值），或者可以转化为该格式，就可以直接使用cv::ml::TrainData::loadFromCSV从CSV文件中创建一个新的cv::ml::TrainData对象。

```
// Load training data from CSV file; part of each row may be treated as the
// scalar or vector responses; the rest are input values.
//
static cv::Ptr<cv::ml::TrainData> cv::ml::TrainData::loadFromCSV(
   const String& filename,                    // Input file name
   int           headerLineCount,             // Ignore this many lines
   int           responseStartIdx = -1,       // Idx of first out var (-1=last)
   int           responseEndIdx   = -1,       // Idx of last out var plus one
   String&       varTypeSpec      = String(), // Optional, specifies var types
   char          delimiter        = ',',      // Char used to separate values
   char          missch           = '?'       // Used for missing data in CSV
);
```

CSV阅读器跳过第一个headerLineCount行，然后逐行[注5]读取数据，每个特征间用逗号分隔。如果在CSV文件中有其他分隔符，也可以用它来替换默认的逗号（例如，以空格或分号）。通常，响应保存在最左边或最右边的列中，但是用户也可以根据需要指定任何一列或几列保存它。用户还可以在区间[responseStartIdx，responseEndIdx）内绘制响应，如果需要，变量类型可以通过文本字符串单独指定，例如：

```
"ord[0-9,11]cat[10]"
```

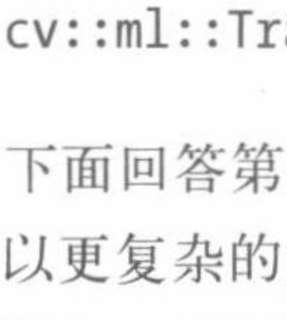
意味着数据有12列，前10列存储有序值，之后有一列分类值和另一列有序值。如果没有提供变量类型规范，则阅读器会尝试通过一些简单的规则来“做正确的事情”。当数据清楚地包含着非数值量（例如，“狗”“猫”），则这是一个分类问题。否则认为数据是有序的。如果只有一个输出变量，那么它会遵循几乎相同的规则，但如果有多个，那么认为它们是有序的。

如果没有提供变量类型规范，reader会尝试通过遵循一些简单的规则来“做正确的事情”。它会考虑输入排好序的（数值）变量，除非它们清楚地包含非数值变量（例如，“狗”“猫”），在这种情况下它们是已经分好类的。如果只有一个输出变量，那么它将遵循几乎相同的规则，[注6]但如果有多个，那么它们总是被认为是有序的。

注5：也就是说，ROW_SAMPLE布局是假设的。

注6：“几乎”意味着有一个有趣的例外，如果输出变量总是一个整数，那么它仍然被认为是分类的。

对于数据丢失问题，可以使用missch参数指定一个特殊字符但是，一些算法不能够处理这样的缺失值。在这种情况下，应该在训练前通过补全或其他方式手工处理数据，抑或是直接拒绝接受损坏的记录。[注7]



实际中常有丢失数据的问题出现。例如，在工作人员喝咖啡休息期间离开的同时，机器正在产生数据，这样一些测量特征就会丢失。而有时，实验数据会被遗忘，例如，在医学实验期间忘记记录药物试验期间患者一天之中的体温变化情况。

秘诀和cv::ml::TrainDataImpl

在源代码中查看cv::TrainData的类定义，你会发现它几乎全是纯虚函数。事实上，它只是一个接口，用户可以从中派生并创建自己的训练数据容器。这就引出了两个问题："为什么你要这么做？""如果cv::ml::TrainData是一个虚拟类型，那么cv::ml::TrainData::create()内部执行了什么？"

下面回答第一个问题。现实生活中，训练数据可能非常复杂、庞大，因而有时很有必要以更复杂的策略来管理数据和存储数据。因此用户可能想要实现自己的训练数据容器，例如，使用数据库来管理大部分可用数据。通过cv::TrainData接口，用户可以实现自己的数据容器，并在该数据上"透明地"运行相应算法。

对于第二个问题，cv::ml::TrainData::create()方法实际上创建了一个不同于它的类的对象。有一个名为cv::ml::TrainDataImpl的类基本上就是数据容器的默认实现。这个对象按照可以按照用户期望的方式来管理数据——以数组的形式存储各种所需数据。

实际上，除非用户直接查看库的源代码，否则这个类几乎是不可见的。当然，如果需要构建自己的cv::ml::TrainData -derived容器类，那么查看cv::ml::TrainDataImpl in .../*opencv/modules/ml/src/data.cpp*会对你很有帮助。

分割训练数据

现实生活中，在训练机器学习系统时，并不会使用所有的数据。因为在完成训练后，仍需要保留一些数据来测试算法。否则，无法估计训练过的系统在新数据集上的效果。当构建一个新的TrainData实例时，默认它可以用于训练数据，因而不会有数据被用于测试。cv::ml::TrainData::setTrainTestSplit()可以将数据分成训练集和测试集

注7：各种算法，如决策树和朴素贝叶斯，以不同的方式处理缺失值。决策树使用替代划分（Breiman [Breiman84]所谓的"代理划分"），朴素的贝叶斯算法则是推算得出这些值。不幸的是，目前在OpenCV中执行ML决策树/随机树还不能处理这种缺失的测量。

两部分，只需使用训练集来训练模型。假如用户已经标记了想要用于训练的数据，则cv::ml::StatModel::train()会自动调用训练数据。

```
// Splits the training data into the training and test parts
//
void cv::ml::TrainData::setTrainTestSplit(
  int    count,
  bool   shuffle = true
);

void cv::ml::TrainData::setTrainTestSplitRatio(
  double ratio,
  bool   shuffle = true
);

void cv::ml::TrainData::shuffleTrainTest();
```

cv::ml::TrainData中的三个方法setTrainTestSplit()，setTrainTestSplitRatio()和shuffleTrainTest()对解决数据集和训练集分配问题很有帮助。第一个方法有一个计数参数，指定数据集中应该被标记为训练数据的向量个数（剩余部分是测试数据）。第二个方法在功能上与第一个类似，但指定的是数据被标记为训练数据的百分比（例如，0.90 = 90%）。第三个方法是随机分配训练和测试向量（同时保持固定的数量）。前两种方法都支持shuffle参数。如果它为真，则测试和训练标签将随机分配；否则，训练样本从头计数，剩下的为测试样本。

需要注意的是，在内部，有三个独立类似的索引默认实现IMPL：样本索引，训练数据索引和测试数据索引。它们都是容器中整个样本数组的索引列表，记录着在特定场景中使用哪些样本。样本索引是列出将要使用的所有样本的数组。训练数据索引和测试数据索引，列出了哪些样本用于训练，哪些用于测试。正如前文中提到的，这三个指标之间存在联系，但不太直观。

如果定义了训练集的索引，那么测试集的索引通常也被定义了。这是使用刚才描述的这些创建内部索引的函数的很自然的结果。

只有当两个索引都没有被定义的时候，才会使用相同的索引。在这种情况，所有样本索引所指示的数据都被假定是可以用于训练的，同时没有数据会被标记成测试数据。

访问cv::ml::TrainData

一旦构建好训练数据，无论是否进行预处理，都可以使用下面描述的方法来检索数据，cv::ml::TrainData::getTrainSamples()仅检索训练数据矩阵。

```
// Retrieve only the active training data into a cv::Mat array.
//
cv::Mat cv::ml::TrainData::getTrainSamples(
```

```
    int  layout           = ROW_SAMPLE,
    bool compressSamples = true,
    bool compressVars     = true
  ) const;
```

当参数compressSamples或compressVars均为true时，在构建数据结构时，该方法会分别保留由sampleIdx和varIdx设置的行或列。如果所需的结构与原始结构不同，该方法会对数据进行转置。如果定义了样本索引或训练数据索引，则只返回指定的样本。但是如果两者均是被定义的，那么就会返回起决定性作用的训练数据索引。类似地，cv::ml::TrainData::getTrainResponses()方法仅返回发挥作用的响应向量。

```
// Return the train responses (for the samples selected using sampleIdx).
//
cv::Mat cv::ml::TrainData::getTrainResponses() const;
```

对于cv::ml::TrainData::getTrainSamples()，如果同时定义样本索引和训练数据索引，则只返回指定的样本。但如果两者都是定义的，那么就会返回起决定性作用的训练数据索引。

与cv::ml::TrainData::getTestSamples()类似，cv::ml::TrainData::getTestResponses()只返回利用测试样本构造的类似数组。如果没有定义测试数据索引，则返回一个空数组。

最后，介绍一些能够告诉你数据容器中有多少种样本的函数， 如下所示：

```
int getNTrainSamples() const;  // Number of samples indicated by train idx
                               //  or total samples if samples idx is not defined

int getNTestSamples()  const;  // Number of samples indicated by test idx
                               //  or zero test idx is not defined

int getNSamples()      const;  // Number of samples indicated by samples idx
                               //  or total samples if samples idx is not defined

int getNVars()         const;  // Number of features indicated by variable idx
                               //  or total samples if variable idx is not defined

int getNAllVars()      const;  // Number of features total
```

预测

回想一下predict()方法的函数原型，如下所示：

```
float cv::ml::StatModel::predict(
  cv::InputArray  samples,                    // input samples, float matrix
  cv::OutputArray results = cv::noArray(),  // optional output matrix of results
  int             flags   = 0               // (model-dependent)
) const;
```

该方法用于预测面对新的输入数据向量时系统的响应。对于分类问题，predict()返回一个类标签。对于回归问题，此方法返回一个数值。预测时，输入样本必须与训练数据有相同的参量。[注8]一般来说，样本是一个输入的浮点数组，每个样本占一行，输出时，每个结果占一行。当仅有单个样本时，预测结果将从predict()函数返回。但是在某些情况下，对于不同的派生分类器，这些一般行为会略有不同。特定的算法会有相应的附加标志，并允许一些特定的行为 ，例如在基于树的方法中丢失特征值等。函数后缀const告诉我们，预测不会影响模型的内部状态。该方法在线程级别是安全的，可以并行运行，这对于在Web服务器上实现多个客户端图像检索以及加快机器人的场景扫描而言，非常有意义。

除了能够进行预测，还可以用cv::ml::StatModel::calcError()计算模型在训练或测试数据上的误差。当模型用于分类时，能够获取分类不正确样本的百分比；当处理回归问题时，则返回均方误差。如下所示：

```
float cv::ml::StatModel::calcError(
  const cv::Ptr<cv::ml::TrainData>& data, // training samples
  bool                              test, // false: compute over training set
                                          //  true: compute over test set
  cv::OutputArray                   resp  // the optional output responses
) const;
```

这种情况下，通常会使用与训练数据完全相同的cv::ml::TrainData数据容器。然后使用测试参数来确定训练算法在训练数据（test设为false）或测试数据（test设为true）上的的效果。最后，可以使用resp数组来收集所测试的各个向量的响应。虽然它是可选的，但参数不能为空，如果你对输出响应不感兴趣，则必须在此处传递cv::noArray()。

接下来，本章将为读者介绍ML库中的朴素贝叶斯分类器，之后还将介绍讨论决策树算法（决策树、boosting、随机森林和哈尔级联分类器）。对于其他算法，本章将提供简短的描述和使用示例。

使用cv::StatModel的机器学习算法

现在，读者应该对OpenCV中的ML库有了一个很好的认识，下面继续研究如何使用独立的学习方法。本节简要介绍了八种机器学习算法，其中后四种最近已经添加到OpenCV中。这些算法都是众所周知的，在书籍、发表的论文和互联网上可以看到大量参考文献。随着时间的推移，将来必将出现更多的新算法。

注8：与train()一起使用的var_idx参数被“记住”，并在使用predict()方法时，从输入示例中提取必需的组件。因此，样本中的列数应与train_data中的列数相同，即使你会使用var_idx来忽略某些列。

朴素贝叶斯分类器

第20章研究了机器学习库系统化之前的一些传统例程；现在我们将学习一个通过cv::ml::StatModel接口实现的简单分类器。本章将从OpenCV中最简单的监督分类器cv::ml::NormalBayesClassifier开始，或者称之为"普通贝叶斯分类器”或“朴素贝叶斯分类器”。之所以说它是“朴素的”，是因为在它的数学实现中，必须假设所有特征是彼此的独立变量（现实中这种情况很少）。例如，检测到一只眼睛通常意味着另一只眼睛也在附近；这些都不是不相关的。然而，在实践中通常可以忽略这种相关性，并且仍然可以获得良好的结果。Zhang讨论了这种分类器[Zhang04]有时能够取得令人惊讶的效果的可能原因。朴素贝叶斯不能用于回归，但它是一个有效的分类器，可以处理多分类问题，而不局限于二分类。这种分类器是庞大且不断增长贝叶斯网络中的最简单的例子，或者称为“概率图模型”。[注9]

举个例子，假设有一个图像集合，其中一些是人脸图像，剩下的是其他东西（也许是汽车和花）的图像。图21-1描绘了一个模型，该模型认为是用户观察的对象——人脸导致了某些可测量特征的存在。一般来说，贝叶斯网络是因果模型。具体来说，如果图像中的面部特征是一个物体的存在引起的。那么这个物体很可能就是人脸，反之亦然。简单地说，图中的物体是”人脸“或其他类型的物体，蕴含着“存在一个左眼”，“存在一个右眼”等五个断言的真假。一般来说，这样的图形通隐含着额外的信息，例如每个节点的可能取值和每个气泡中取值的实际概率，作为节点指向气泡的值函数。在本章的例子中，节点*O*可以取值“face” “car”或“flower”，而其他五个节点的取值可以呈现，也可以缺省。每个特征的概率以及给定对象的性质也可以从数据中学习。

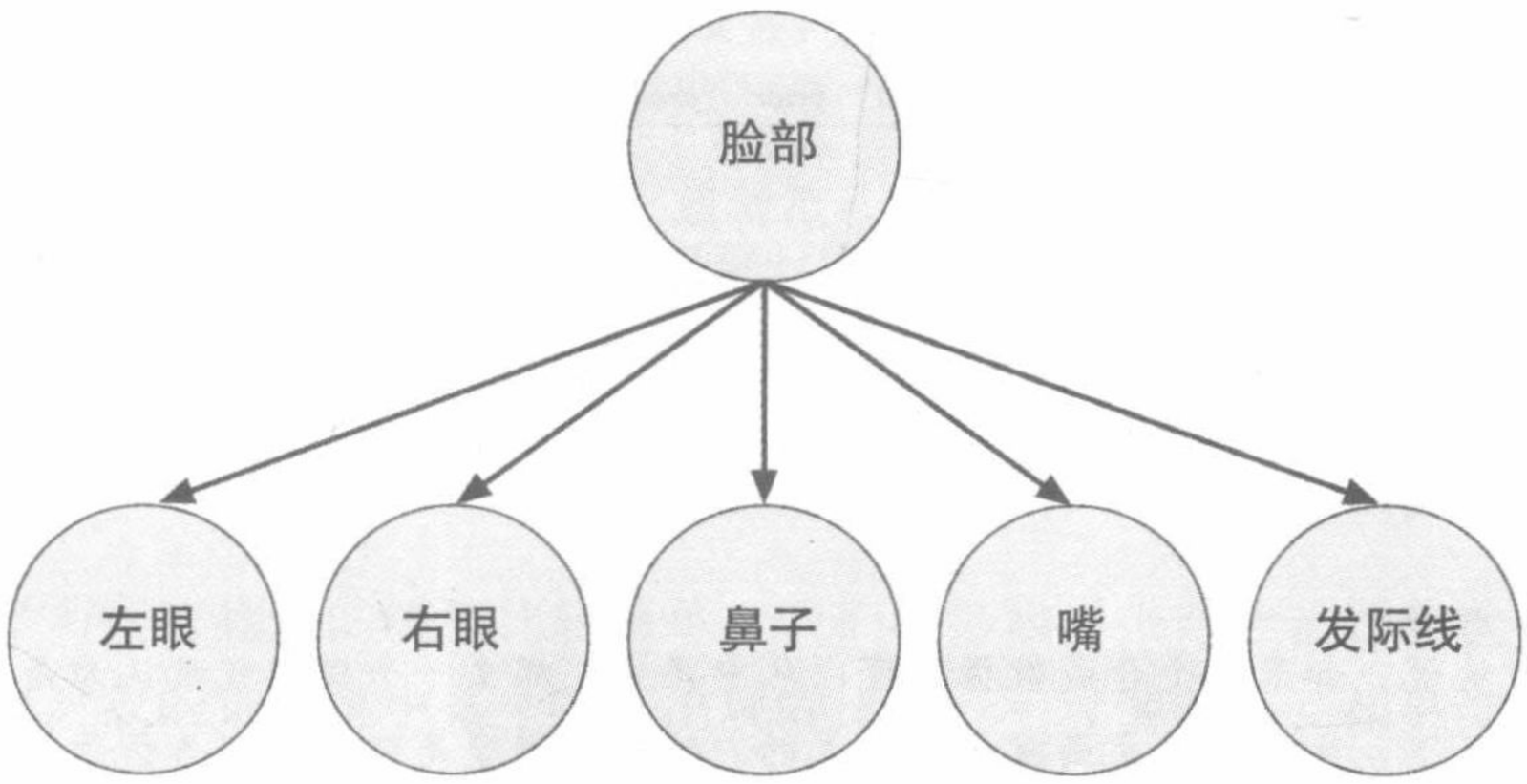


图21-1. 一个朴素的贝叶斯网络，因为一个对象（人脸）的出现，产生底层特征

注9：关于该主题的介绍，可以查看[Neapolitan04]。有关概率图形模型主题的详细讨论，请参见[Koller09]。



这正是图“不相关”的性质所在，具体而言，存在鼻子的概率仅取决于检测对象是否是“人脸”，与是否存在嘴和发际线等问题之间并无关联。因此要学习的案例组合非常少，因为我们关心的一切基本上都可以分解为特征与对象的统计学相关的问题。而这种分解是正是不相关的。

在实际应用中，朴素贝叶斯分类器中的对象通常是隐藏变量，而特征就是对输入图像通过图像处理操作观察到的证据，即对象的值是任何类型（例如“人脸”）。这样的模型被称为“生成式模型”，因为对象据此产生面部特征。由于是生成的，所以在训练之后我们假设面部是存在的，然后在面部存在的前提下，随机采样生成了（或者没生成）哪些特征。[注10]因为它是生成的，在训练之后，我们可以从假设对象节点的值为“脸”的开始，然后随机抽样，假定我们假设一个面孔存在，概率地生成哪些特征。[注11] 这种自上而下的由模型生成数据的方法可以处理判别式模型不能处理的问题。比如，可以生成人脸用于计算机图形显示，机器人可以据此想象怎样生成场景，并让物体与之交互。与图20-1形成对照的是，判断式模型会将图中的所有箭头翻转。

贝叶斯网络很深奥，一开始学习的时候很难理解。但是朴素贝叶斯算法衍生于贝叶斯法则。[注12]在这个例子中，给定特征下发现人脸的条件概率p有（在图21-1中，由LE，RE，N，M和H表示为左至右）：

$$p(\mathrm{O} = "face" | \mathrm{LE}, \mathrm{RE}, \mathrm{N}, \mathrm{M}, \mathrm{H}) = \frac{p(L|E, RE, N, M, H|\mathrm{O} = "face")\, p(\mathrm{O} = "face")}{p(LE, RE, N, M, H)}$$

换而言之，这个方程可以写作：

$$posterior \quad probability = \frac{likelihood * prior \quad probability}{evidence}$$

这个方程的意义在于，实际情况下，我们计算证据，然后找出产生证据的原因。因为用于计算的证据相同，所以我们可以不考虑这个条件，如果有很多模型，则需要找到有最大分子的那个。分子是模型和证据的联合概率：$p(O="face", LE, RE, N, M, H)$。

注10：更一般地说，如果整个合成数据集可以从中产生，那么一个模型就被认为是生成式的。在这种情况下，生成的对立面就是判定式的。歧视性模型是任何可以告诉你提供给它的任何数据点的模型，但不能用于合成数据。

注11：用朴素贝叶斯生成一张脸会很愚蠢，因为它假定了特征的独立性。但是，更普遍的贝叶斯网络可以根据需要在特征依赖中轻松构建。

注12：回想一下，当我们在第20章中K-means分类的上下文中讨论了马氏距离的效用时，我们就已经遇到过贝叶斯法则。

到目前为止，我们还没有真正见识到朴实的贝叶斯分类器的“朴素”的部分。而这些方程对于任何贝叶斯分类器都是正确的。为了利用不同特征在统计学上彼此独立的假设（回想一下，这是图21-1中的主要内容），我们现在使用概率的链式规则来导出联合概率：

$$p(O = "face", LE, RE, N, M, H) =$$
$$p(O = "face") * p(LE|O = "face") * p(RE|O = "face", LE) * p(N|O = "face", LE, RE)$$
$$* p(M|O = "face", LE, RE, N) * p(H|O = "face", LE, RE, N, M)$$

最后，当对特征应用独立性假设时，条件特征就被视为无效。例如，观察到鼻子的概率表明物体是“人脸”的概率，则根据我们的假设，在“人脸”存在这个事实基础上，观察到左眼和右眼（即，$p(N|O = "face", LE, RE)$）的概率等于观察到鼻子的概率$p(N|O = "face")$。将此逻辑代入上述方程右侧，结果如下：

$$p(O = "face", LE, RE, N, M, H) = p(O = "face") \prod_{feature}^{\{LE,RE,N,M,H\}} p(feature|O = "face").$$

把“人脸”抽象为物体，把“获得的特征”抽象为所有特征，可以简化方程如下：

$$p(\text{object, all features}) = p(\text{object}) \prod_{i=1}^{all\ features} p(feature_i | object)$$

我们为所需的对象学习模型，并将其用作整体分类器。在运行模型时，我们计算特征并找到使该方程值最大化的特定对象。然后测试该“获胜”对象的概率是否超过给定阈值。如果是，则称其为我们的目标；如果没有，则认为没有检测到目标。



如果只有一个感兴趣的对象，通常也是这样，那么你会问：“我正在计算的概率与什么有关？”在这种情况下，一直会有一个暗含的第二个对象：背景。背景是除去我们感兴趣的要学习和识别物体之外的所有对象。

在实践中，学习所有物体的模型比较简单。我们拍摄很多张照片；计算这些照片的特征，计算每个对象的训练集中某个特征出现的比例。一般情况下，如果没有很多的数据，简单的模型（朴素贝叶斯）会比很多复杂模型获得很好的性能，因为复杂模型使用了太多的假设，以至产生欠拟合。

朴素贝叶斯分类器和cv::ml::NormalBayesClassifier

下面介绍普通贝叶斯分类器的类定义。类名cv::ml::NormalBayesClassifier实际上是另

外一层接口定义，cv::ml::NormalBayesClassifierImpl则是实现普通贝叶斯分类器的实际类的名称。为了方便起见，此定义列出一些重要的继承方法作为注释。

```
// Somewhere above...
// namespace cv {
//   namespace ml {
//
class NormaBayesClassifierImpl : public NormaBayesClassifier {
                                          // cv::ml::NormaBayesClassifier is derived
                                          // from cv::ml::StatModel
public:
  ...

  float predictProb(
    InputArray   inputs,
    OutputArray  outputs,
    OutputArray  outputProbs,
    int          flags      = 0
  );

  ...

  // From class NormaBayesClassifier
  //
  // Ptr<NormaBayesClassifier> NormaBayesClassifier::create(); // constructor

};
```

朴素贝叶斯分类器的训练方法继承自cv::ml::StatModel，如下所示：

```
bool cv::ml::NormalBayesClassifier::train(
  const Ptr<cv::ml::TrainData>& trainData,   // your data
  int flags = 0                              // 0=new data or UPDATE_MODEL=add
);
```

flags参数可以为0或cv::ml::StatModel::UPDATE_MODEL标志，该标志意味着使用附加的训练数据更新模型，而不是从头开始重新训练。

cv::NormalBayesClassifier实现了cv::ml::StatModel中描述的的predict()接口，它能够计算并返回其输入向量最可能的类别。如果样本矩阵中提供了多个输入数据向量（行），则在结果向量的相应行中返回预测。如果样本中只有一个输入，则所得到的预测也将通过predict()方法以浮点值的形式返回，并将结果数组设置为cv::noArray()。

```
float cv::ml::NormalBayesClassifier::predict(
  cv::InputArray  samples,                    // input samples, float matrix
  cv::OutputArray results = cv::noArray(),    // optional output results matrix
  int             flags   = 0                 // (model-dependent)
) const;
```

此外，普通贝叶斯分类器还提供了方法predictProb()。此方法使用的参数与cv::ml::N

ormalBayesClassifier::predict()参数相同，还可以使用数组resultProbs，这是一个number_of_samples×number_of_classes大小的浮点矩阵，其中存储了计算得到的概率（相应样本属于特定类）。[注13]这种预测方法的调用格式如下：

```
float cv::ml::NormalBayesClassifier::predictProb( // prob if single sample
  InputArray  samples,                             // one sample per row
  OutputArray results,                             // predictions, one per row
  OutputArray resultProbs,                         // row=sample, column=class
  int         flags  = 0                           // 0 or StatModel::RAW_OUTPUT
) const;
```

虽然朴素贝叶斯分类器对小数据集是非常有用，但是在面临庞大的数据规模时，通常表现不佳。为了解决这一问题，我们继续讨论基于树的分类器，当有足够多的数据时，它可以比朴素贝叶斯分类器更为简单。

二叉决策树

我们将具体讨论二叉决策树，它们最常用，并且实现了机器学习库中大量的功能，因此将被作为指导性的例子来讲解。二叉决策树由Leo Breiman及其同事提出。[注14，译注3]他们称之为“分类和回归树”（CART）。OpenCV实现的就是“分类回归树”。算法的要点是给树的每个节点定义一个衡量标准。比如，当拟合一个函数的时候，我们使用真实值和预测值之差的平方和，这就是衡量标准。算法的目地是使差的平方和最小。这就是衡量标准。算法的目的是使差的平方和最小。对于分类问题，我们定义一个度量，使得当一个节点的大多数值都属于同一类时，这个度量最小。三个最常用度量是熵（entropy）、基尼系数（Giniindex）和错分类（misclassification），这些度量都会在本节中介绍。一旦我们定义了度量，二叉树就负责搜寻整个特征向量，搜寻哪个特征和哪个阈值可以正确分类数据或正确拟合数据（在本书中称为数据“纯净”）。根据惯例，我们说特征值大于这个阈值的数据为“真”，被分配到左分支；其他的点放到右分支。[注15]从二叉树的每个节点递归使用这个过程直到数据都“纯净”了，或者节点里的数据样本数达到最小值。图21-2展示了一个例子。

注13：请注意，如果你不需要概率，可以用cv::noArray()传递给resultProbs。实际上，cv::ml::NormalBayesClassifier::predict()只是一个这样的predictProb()的封装。

注14：Leo Breiman等人。Classification and Regression Trees (Belmont, CA: Wadsworth, 1984)。

注15：显然放到左边还是右边是完全任意的。然而，不坚持这样也不会让真正有决策树经验的人混淆。

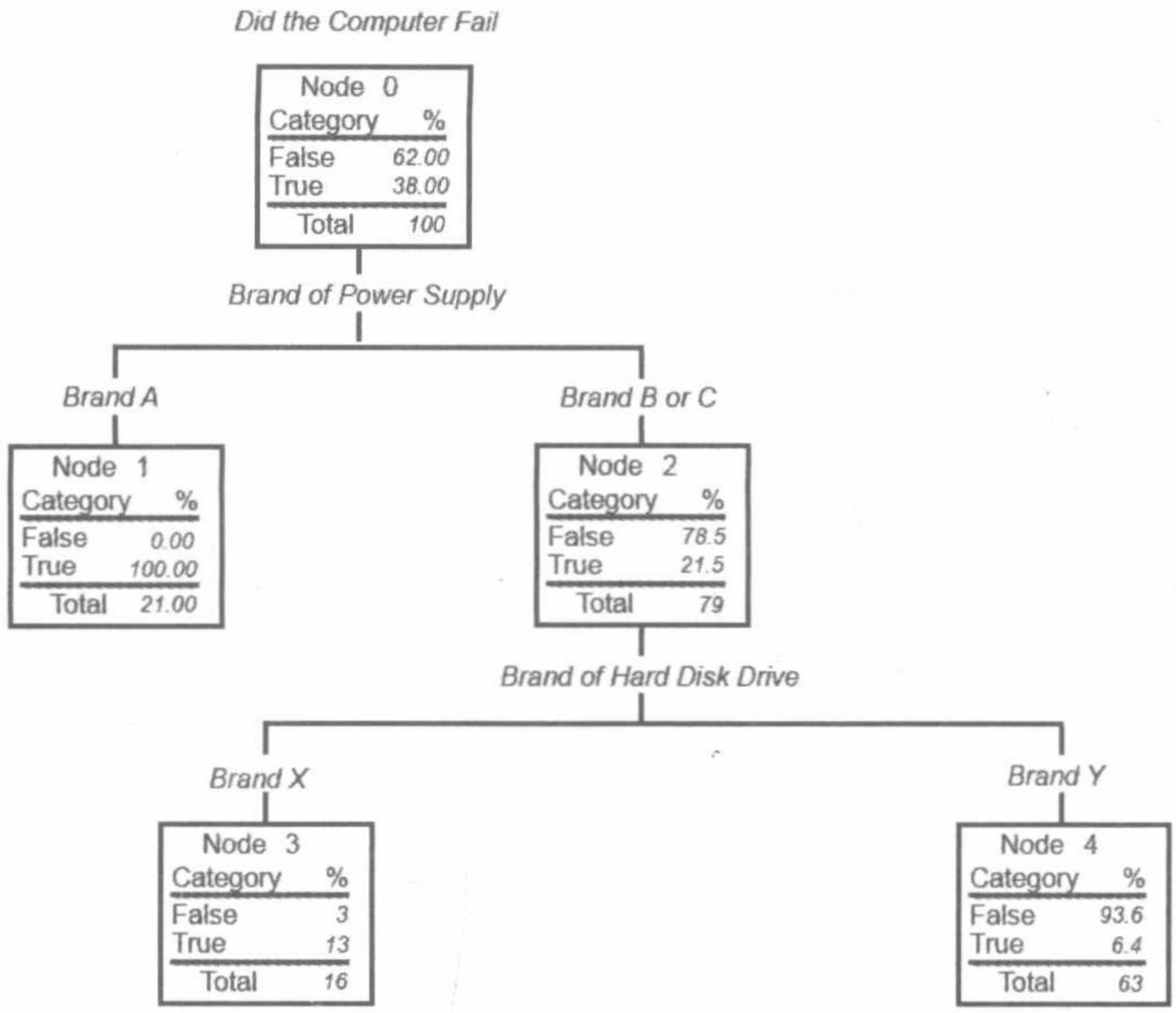


图21-2. 在这个例子中，分析了100台笔记本电脑，并且根据导致故障率的主要因素来构建分类树；树的叶节点代表这100台计算机

随后会给出几个关于节点不纯度$i(N)$定义的等式：不同的定义适用于不同的问题情形，包括我们首先需要两种情况来分析回归和分类。

回归不纯度

回归或函数拟合中，节点不纯度就只是节点值y和数据值x之差的平方。我们需要最小化这个等式：

$$i(N) = \sum_j (y_j - x_j)^2$$

分类不纯度

分类中，决策树经常使用熵不纯度、基尼不纯度和错分类不纯度这三个度量中的某一个来衡量。这些方法中，我们使用$P(\omega_i)$来表示节点N中类别ω_i所占的比例。不同方法中，不纯度有一些细微的差别。基尼不纯度经常用，但所有的算法旨在试图使节点中的不纯度最小。

图21-3显示了我们想要的最小化不纯度。最好是对每个不纯度都进行一次尝试，以此来确定一个最有效的验证集。

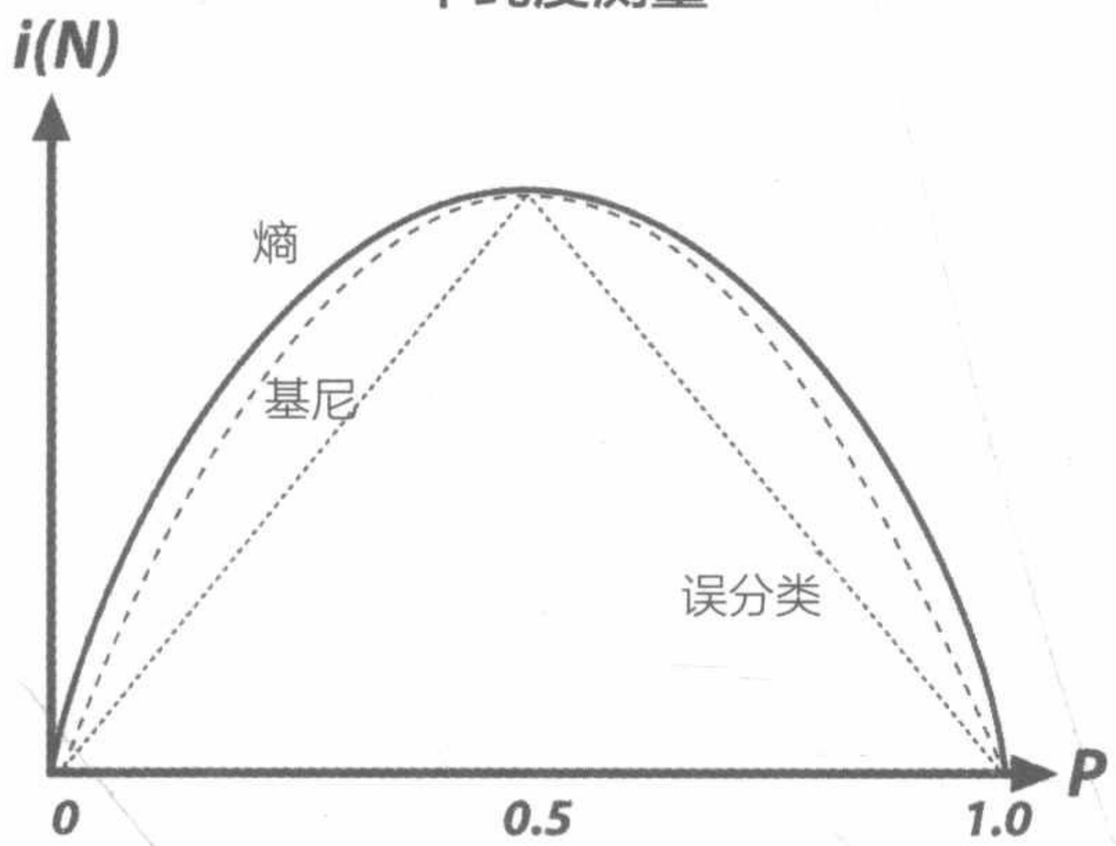


图21-3. 决策树不纯度

熵不纯度：

$$i(N) = \sum_{j} P(\omega_j)\log P(\omega_j)$$

吉尼不纯度：

$$i(N) = \sum_{j} 1 - \max P(\omega_j)$$

错分类不纯度：

$$i(N) = \sum_{j} P(\omega_j)P(\omega_j)$$

决策树可能是最常用的分类方法。因为它实现简单，结果容易解释，适应不同的数据类型（包括类别、数值数据、未归一化的和混合数据），能够通过替代划分（surrogate split）处理数据丢失，通过划分的顺序能够赋予数据特征不同的重要性。决策树构成了其他算法的基础，比如boosting和随机树，我们稍后会简单介绍这些算法。

Opencv中的实现

下面是对cv::ml::DTrees的简单描述，train()方法从基类派生而来，在定义中，我们需要了解的是如何设置和加载模型的参数，以下是cv::ml::DTrees的声明：

```
// Somewhere above...
// namespace cv {
//   namespace ml {
//
class DTreesImpl : public Dtrees {    // cv::ml::DTrees is derived
```

```
                                                    // from cv::ml::StatModel
public:

  // (Inherited from cv::ml::DTrees)
  //
  //enum Flags {
  //  PREDICT_AUTO     = 0,
  //  PREDICT_SUM      = (1<<8),
  //  PREDICT_MAX_VOTE = (2<<8),
  //  PREDICT_MASK     = (3<<8)
  //};

  int     getCVFolds() const;                 // get num cross validation folds
  int     getMaxCategories() const;           // get max number of categories
  int     getMaxDepth() const;                // get max tree depth
  int     getMinSampleCount() const;          // get min sample count
  Mat     getPriors() const;                  // get priors for categories
  float   getRegressionAccuracy() const;      // get required regression acc.
  bool    getTruncatePrunedTree() const;      // get to truncate pruned trees
  bool    getUse1SERule() const;              // get for use 1SE rule in pruning
  bool    getUseSurrogates() const;           // get to use surrogates

  void    setCVFolds( int val );              // set num cross validation folds
  void    setMaxCategories( int val );        // set max number of categories
  void    setMaxDepth( int val );             // set max tree depth
  void    setMinSampleCount( int val );       // set min sample count
  void    setPriors( const cv::Mat &val );    // set priors for categories
  void    setRegressionAccuracy( float val ); // set required regression acc.
  void    setTruncatePrunedTree( bool val );  // set to truncate pruned trees
  void    setUse1SERule( bool val );          // set for use 1SE rule in pruning
  void    setUseSurrogates( bool val );       // set to use surrogates

  ...

  // Experts can use these, but non-experts can ignore them.
  //
  const std::vector<Node>&  getNodes()   const;
  const std::vector<int>&   getRoots()   const;
  const std::vector<Split>& getSplits()  const;
  const std::vector<int>&   getSubsets() const;

  ...

  // From class DTrees
  //
  //Ptr<DTrees> DTrees::create(); // The algorithm "constructor",
  //                              //  returns Ptr<DTreesImpl>

};
```

首先，你可能已经注意到这个名字是复数DTrees。这是因为在计算机视觉中，独立决策树并不常用，常用的是决策树的集合，它可以产生一些联合决策。在OpenCV中实现了两个这样的算法：RTree（随机森林）和Boost（boosting），在本章后面将逐一介绍，

两者共享很多内部机制。DTrees是一种基础类，在目前的情况下，这可以被看作是由一棵树构成的集合。

此外，我们再次看到了数据容器cv::ml::TrainData，这些方法原型大多是纯虚函数。实际上有一个衍生自cv::ml::DTrees 的隐藏类cv::ml::DTreesImp，它包含所有的默认实现。用户可以在前面的类定义中忽略这些虚拟方法。如果在某些时候你想去看默认的实现，则需要在... */modules/ml/src/tree.cpp*中查看相关成员函数和信息。

使用cv::ml::DTrees::create()构建一个cv::ml::DTrees对象后，需要配置各种运行时参数。你可以通过以下两种方法之一来实现。第一种方法是构造一个包含所需参数的结构体来配置树。这个结构体是cv::ml::TreeParams。接下来会给出其定义的主要部分。

此外，还可以使用其构造函数创建结构，该构造函数包含每个组件的参数，或者也可以调用默认构造函数创建它——在这种情况下，所有内容都将设置为默认值 ，然后可以使用单独的访问器来设置要自定义的值。

表21-1包含cv::ml::TreeParams()的参数及其默认值和含义的简要说明。

表21-1：cv::ml::TreeParams()构造函数的参数及其含义

参数	构造函数的默认参数	含义
maxDepth	INT_MAX	树的最大深度，但是不能太深
minSampleCount	10	如果节点的样本数小于该值，则不分割节点
regressionAccuracy	0.01f	如果训练样本中实际值和估计值之间的差异小于regressionAccuracy，则停止分割
useSurrogates	false	允许使用替代分叉点来处理丢失的数据[尚未实施]
maxCategories	10	限制决策树预分类的数量
CVFolds	10	如果（CVFolds> 1）则使用K-fold交叉修剪决策树，其中K等于CVFolds
use1SERule	true	True表示更大力度的修剪，这会导致树的规模更小，但准确性更差，可以用于解决过拟合问题
truncatePrunedTree	true	如果为true，从树中删除被剪掉的分支
priors	cv::Mat()	为错误答案设置可选择的权重

下面对其中两个参数要做进一步说明。maxCategories限制分类数量，在此之前决策树会预先进行分类，分支数必须不超过$2^{max_categories} - 2$个。[注16]当类别数大于maxCategories时，那些多余的类别将会被合并到maxCategories中相近的值。通过这种方式，决策树一次测试的层次数不会超过maxCategories，这导致每个分类输入的可能决策子集不会超过$2^{max_categories}$。当该参数设置为较小的值时，可以降低计算量，但会以牺牲准确性为代价。

最后一个参数priors设置错误分类的相对权重。也就是说，如果我们构建一个两级分类器，如果第一个输出类的权重为1，第二个输出类的权重为10，则在预测第二个类产生的一个错误相当于在预测第一个类中产生*10*个错误 。我们用食用蘑菇和毒蘑菇来举例。在这种情况下，将“有毒蘑菇”误认为可食用食物的惩罚是将食用菌误认为毒蘑菇的10倍以上是有非常意义的。参数priors是一个float数组，与类的元素数相同。分配的值与类本身的顺序相同。

如下所示，train()方法直接衍生自cv::ml::StatModel：

```
// Work directly with decision trees:
//
bool cv::ml::DTrees::train(
const cv::Ptr<cv::ml::TrainData>& trainData,   // your data
int                               flags    = 0  // use UPDATE_MODEL to add data
);
```

在train()方法中，训练数据存储在一个浮点矩阵中。在使用决策树时，如果要将数据按列排列而不是按行排列，你可以在构建trainData时将结构设置为cv::COL_SAMPLE，这是该算法最高效的布局方式。示例21-1详细说明了决策树的创建和训练。

决策树进行预测的方法与基类相同，即cv::ml::StatModel：

注16：关于分类与有序划分的更多细节：有序变量上的划分具有“如果$x < a$则向左移动，否则向右移动”的形式，分类变量上的分割具有“如果$x \varepsilon \{v1, v2, v3, \ldots, vk\}$左移，否则右移”的形式，其中是变量的一些可能的值。因此，如果分类变量具有N个可能值，则为了在该变量上找到最佳分割，需要尝试$2N-2$个子集（空集和完整子集被排除）。因此，使用近似算法，并根据当前分析的节点中的样本的统计，所有N个值被分组为$K \leq$(max_categories)簇（通过K均值算法）。此后，该算法尝试簇的不同组合并选择最佳分割，这通常会给出相当好的结果。注意，对于两个最常见的任务，二分类和回归，可以在不需要进行任何聚类的情况下有效地找到最佳分类分割（即，最佳的子集）。因此，聚类仅适用于具有$N >$ (max_categories)可能值的分类变量的$n > 2$类分类问题。因此，对于将maxcategories设置为大于20的情况，你应该慎重考虑，这意味着每次拆分将有超过一百万次操作！

```
float cv::ml::DTrees::predict(
  cv::InputArray  samples,
  cv::OutputArray results = cv::noArray(),
  int             flags   = 0
) const;
```

这里，样本数据保存在一个浮点矩阵中，每个样本有一行。在单个输入的情况下，返回值足够，结果可以设置为cv::noArray()。当需要评估多个向量时，输出结果将包含对每个输入向量的预测。最后，flags用于指定各种可能的选项。例如，cv::ml::StatModel::PREPROCESSED_INPUT表示每个分类变量j的值被归一化到范围$0..N_j-1$，其中N_j是第j个变量的类别。假设某个变量可能只取两个值A和B，则在归一化后A转换为0，B转换为1。因为算法知道了数据可能波动的界限，归一化数据到（0, 1）区间仅仅是加速了决策树的计算和预测，对准确性没有影响。当标志为cv::ml::StatModel::RAW_OUTPUT时，此方法返回归一化后的预测值，否则，返回的预测值会转换到原始标签空间。

大多数用户只训练并使用决策树，但是高级用户或者研究人员可能想测试树节点，改变树节点或者改变分叉标准。如本节开头所述，有关如何执行此操作的信息，请参见ML文档，网址为*http://docs.opencv.org*。这里分析的重点是类结构cv::ml::DTrees、节点结构cv::ml::DTrees::Node以及分叉结构cv::ml::DTrees::Split。

决策树的用法

接下来本节将通过具体的例子来介绍决策树的具体用法。构造一个程序去学习识别有毒蘑菇。首先介绍一个名为*agaricuslepiota.data*的公共数据集，其中包含大约8 000种不同类型的蘑菇的信息。它列出了许多可以区分蘑菇外观特征，例如帽子的颜色，“伞盖”的尺寸和间距，蘑菇是否有毒等。[注17]数据文件采用CSV格式，由标签'p'或'e'（分别表示有毒或可食用），后跟22个分类属性，分别由单个字母表示。当数据文件中特定的数据点缺失时（即该特定类型的蘑菇的一个或多个属性是未知的）则用'?'填充该项。

仔细研究这个程序，显然它使用二叉树决定树来学习各种可见属性，以此来区分食用蘑菇和有毒蘑菇（示例21-1）。[注18]

注17：该数据集广泛应用于机器学习教育和算法测试。它在网络上广泛可用，特别是UCI机器学习代码库（*https://archive.ics.uci.edu/ml/datasets/mushroom*）。原始的蘑菇记录来自G. H. Lincoff（Pres.），*The Audubon Society Field Guide to North American Mushrooms*（New York: Alfred A. Nnopf，1981）。

注18：前面讲过，cv::ml::DTrees目前无法处理丢失的数据。这种支持曾经存在于旧的实现中，并且可能会及时再次出现，但是目前你可能希望删除包含'?'标记的条目。

示例21-1：创建和训练二叉树

```
#include <opencv2/opencv.hpp>
#include <stdio.h>
#include <iostream>

using namespace std;
using namespace cv;

int main( int argc, char* argv[] ) {

    // If the caller gave a filename, great. Otherwise, use a default.
    //
    const char* csv_file_name = argc >= 2
    ? argv[1]
    : "agaricus-lepiota.data";

    cout <<"OpenCV Version: " <<CV_VERSION <<endl;

    // Read in the CSV file that we were given.
    //
    cv::Ptr<cv::ml::TrainData> data_set = cv::ml::TrainData::loadFromCSV(
      csv_file_name,   // Input file name
      0,               // Header lines (ignore this many)
      0,               // Responses are (start) at thie column
      1,               // Inputs start at this column
      "cat[0-22]"      // All 23 columns are categorical
    );                 // Use defaults for delimeter (',') and missch ('?')

    // Verify that we read in what we think.
    //
    int n_samples = data_set->getNSamples();
    if( n_samples == 0 ) {
        cerr <<"Could not read file: " <<csv_file_name <<endl;
        exit( -1 );
    } else {
        cout <<"Read " <<n_samples <<" samples from " <<csv_file_name <<endl;
    }

    // Split the data, so that 90% is train data
    //
    data_set->setTrainTestSplitRatio( 0.90, false );
    int n_train_samples = data_set->getNTrainSamples();
    int n_test_samples  = data_set->getNTestSamples();

    cout <<"Found " <<n_train_samples <<" Train Samples, and "
    <<n_test_samples <<" Test Samples" <<endl;

    // Create a DTrees classifier.
    //
    cv::Ptr<cv::ml::RTrees> dtree = cv::ml::RTrees::create();

    // set parameters
    //
    // These are the parameters from the old mushrooms.cpp code
```

```
// Set up priors to penalize "poisonous" 10x as much as "edible"
//
float _priors[] = { 1.0, 10.0 };
cv::Mat priors( 1, 2, CV_32F, _priors );

dtree->setMaxDepth( 8 );
dtree->setMinSampleCount( 10 );
dtree->setRegressionAccuracy( 0.01f );
dtree->setUseSurrogates( false /* true */ );
dtree->setMaxCategories( 15 );
dtree->setCVFolds( 0 /*10*/ );  // nonzero causes core dump
dtree->setUse1SERule( true );
dtree->setTruncatePrunedTree( true );
//dtree->setPriors( priors );
dtree->setPriors( cv::Mat() ); // ignore priors for now...

// Now train the model
// NB: we are only using the "train" part of the data set
//
dtree->train( data_set );

// Having successfully trained the data, we should be able
// to calculate the error on both the training data, as well
// as the test data that we held out.
//
cv::Mat results;
float train_performance = dtree->calcError(
                                        data_set,
                                        false,      // use train data
                                        results //cv::noArray()
                                        );
std::vector<cv::String> names;
data_set->getNames(names);
Mat flags = data_set->getVarSymbolFlags();

// Compute some statistics on our own:
//
{
    cv::Mat expected_responses = data_set->getResponses();
    int good=0, bad=0, total=0;

    for( int i=0; i<data_set->getNTrainSamples(); ++i ) {
        float received = results.at<float>(i,0);
        float expected = expected_responses.at<float>(i,0);
        cv::String r_str = names[(int)received];
        cv::String e_str = names[(int)expected];

        cout <<"Expected: " <<e_str <<", got: " <<r_str <<endl;

        if( received==expected ) good++; else bad++; total++;
    }
    cout <<"Correct answers:   " <<(float(good)/total) <<"%" <<endl;
    cout <<"Incorrect answers: " <<(float(bad)/total)  <<"%" <<endl;
}
```

```
    float test_performance  = dtree->calcError(
                                                data_set,
                                                true,       // use test data
                                                results //cv::noArray()
                                                );

    cout <<"Performance on training data: " <<train_performance <<"%" <<endl;
    cout <<"Performance on test data:     " <<test_performance  <<"%" <<endl;

    return 0;
}
```

我们首先通过命令行解析单个参数，读取CSV文件；如果没有这样的文件，我们读取默认的蘑菇数据文件。首先打印OpenCV版本号，[注19]然后继续解析CSV文件。该文件没有文件头，结果存储在第一列，因此指定读取位置非常重要。所有的输入是绝对的，所以需要明确加以声明。只要阅读CSV文件，就必须说明读取的样本数量，这样既可以保证阅读顺利，又可以保证当前阅读的是用户认为自己正在阅读的文件。

接下来，我们使用setTrainTestRatio()方法进行训练测试数据拆分。首先要指定训练数据占的百分比；当前情况下是90%（0.90）。此外，默认的拆分行为是随机选取数据。如果不想随机打乱，则需要在第二个参数中传递false。拆分完成后，用户可以打印出有分别有多少训练样本和测试样本。

数据全部准备好之后，就可以继续创建将要用于训练的cv::ml::DTrees对象。该对象调用各种set*()方法进行配置。值得注意的是，将一组值传递给setPriors()就可以设定将食用蘑菇错误标记为毒蘑菇与把毒蘑菇误识别为食用蘑菇之间的相对权重。前一个参数为1。0和后一个参数为10。0是因为*e*在字母表中的*p*之前（因此为了获取数值，要把它先转换为ASCII，然后转换为浮点数）。

在这个例子中，我们只是训练DTrees，然后在一些测试数据上预测结果。实际应用中，决策树也可以通过save()保存到磁盘，并通过load()加载（见下文）。以这种方式，用户可以提前训练分类器，然后将训练好的分类器加载到代码中，而无需发布数据或让用户每次都重新训练。以下代码显示如何保存和加载树文件，称为*tree.xml*。

```
// To save your trained classifier to disk:
//
dtree->save("tree.xml","MyTree");

// To load a trained classifier from disk:
//
dtree->load("tree.xml","MyTree");
```

注19：这可能是一个好习惯，特别是在与机器学习库一起使用时，OpenCV的这一部分处于相对活跃的发展阶段。

```
// You can also clear an existing trained classifier.
//
dtree->clear();
```



使用*.xml*存储一个XML数据文件，如果我们使用*.yml*或*.yaml*扩展名，则存储一个YAML数据文件。可选的“MyTree”标记是可选的，它表示*tree.xml*文件中的树的标签。与机器学习模块中的其他统计模型一样，使用`save()`时，不能将多个对象存储在单独一个*.xml*或*.yml*文件中。当想要存储多个对象时，则需要使用`cv::FileStorage()`和`operator <<()`。但是`load()`不同：即使存在文件中存在其他数据，该函数也可以通过其名称加载相应的对象。

决策树结果

通过修改以前的代码和各种实验参数，我们可以从*agaricus-Lepiota.data*文件中学习蘑菇是有毒还是无毒。如果我们并不利用先验知识训练决策树且不做修剪，它可以很好的学习数据，我们可以得到如图21-4所示的树。尽管完全的决策树可以很好地学习训练，但是回想一下第20章讨论诊断机器学习问题时有关方差/过拟合的相关知识。如图21-4所示，树模型在同时也学习了误差和噪声，因此不太可能在实际数据有良好的表现。这就是为什么OpenCV决策树和CART树需要包含一个附加的步骤来通过修剪树来达到复杂度和性能的平衡。[注20]

图21-4显示了一个修剪过的树，它在训练集上的表现并不是非常好，但是在实际数据中会有更好的性能。因为它在过拟合和欠拟合中达到了平衡。但是这种分类器有一个缺点：尽管它在数据的总误差上获得了比较好的性能，但仍然有1.23%的概率将有毒蘑菇判定为“可以食用的”。

注20：只有在复杂性与性能平衡之前，才可以使用其他决策树实现来增长树，因此我们将修剪阶段与学习阶段相结合。然而，在ML库的开发过程中，我们发现先完成生长再修剪的树（如OpenCV中所实现的）往往比在生长阶段就与修剪相结合的树表现更好。

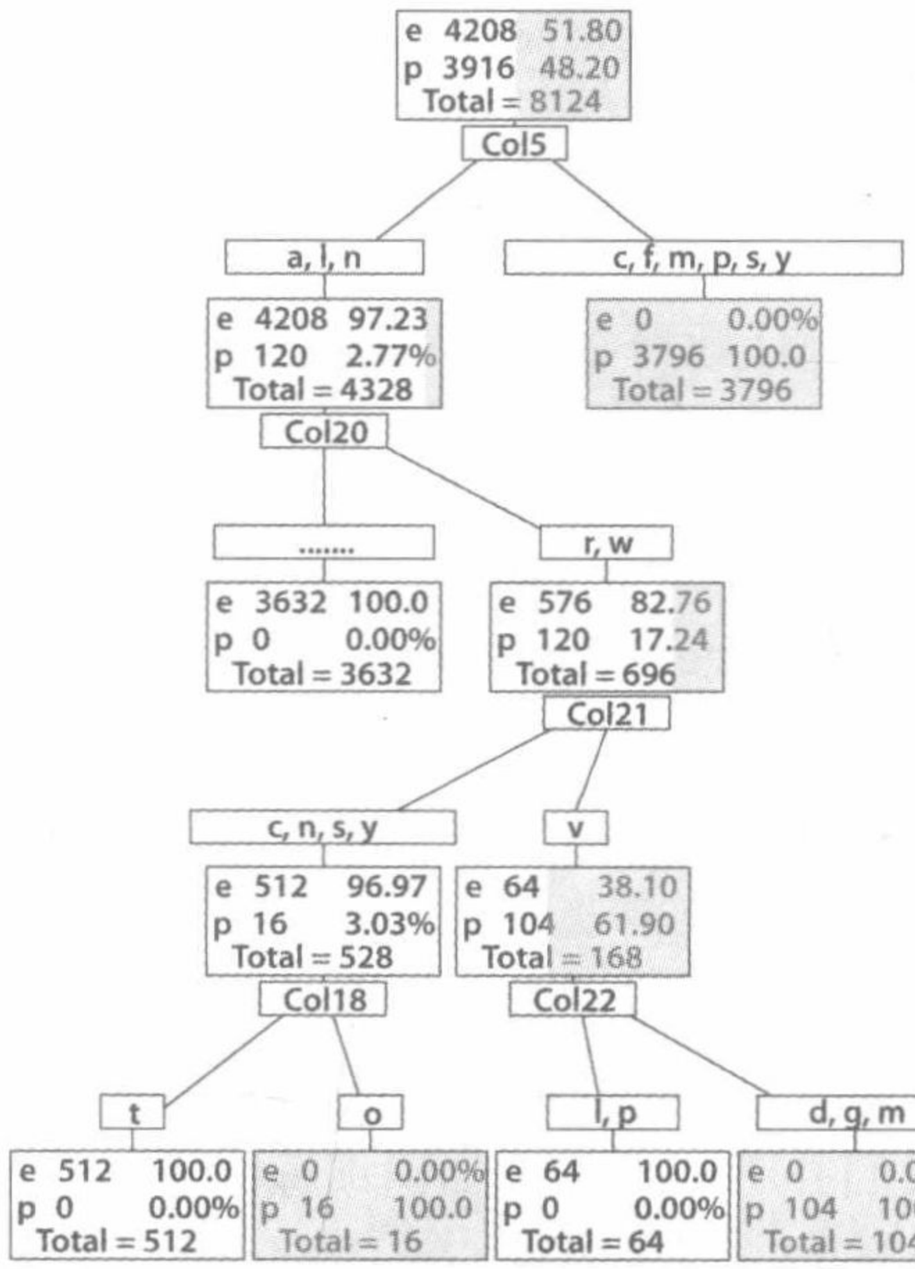


图21-4. 区分有毒（p）和无毒（e）蘑菇的完整决策树、该决策树非常复杂，使训练集上的错误率为0%，但是有可能不能处理测试数据或者真实数据

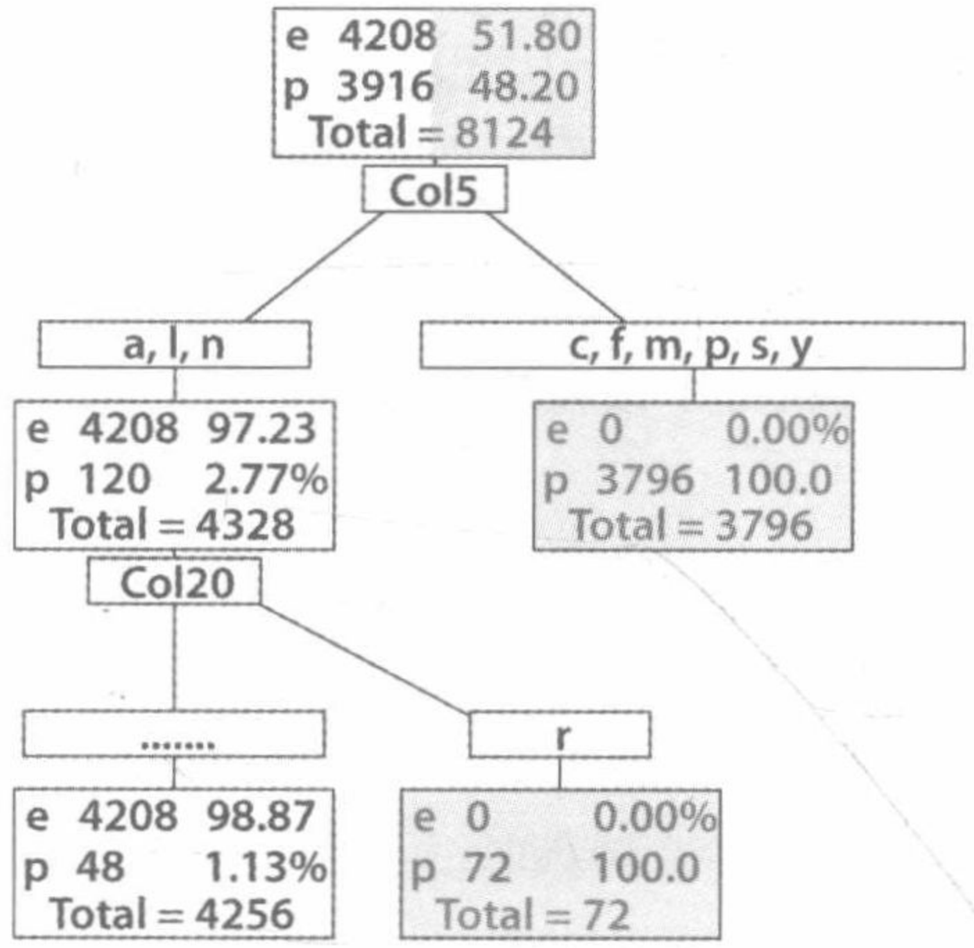


图21-5. 区分有毒（p）和无毒（e）蘑菇的修剪过的决策树、虽然修剪导致该决策树在训练集上有一个较低的错误率，但是它一定能够较好地处理真实数据

这个分类器的优点很明显，它可能会将许多食用菌标记为有毒，但不会让我们吃有毒的蘑菇！正如前面所讲的，我们可以通过有意设置偏好或增加分类和数据的代价来创建这样的分类器。如示例21-1所示，我们增加了错误分类毒蘑菇的成本。通过调整priors向量，我们在分类器中增加代价，结果改变了一个“坏”数据点与“好”数据点间关系的权重。如果不能通过修改分类器代码来改变prior，则可以通过重采样“坏”数据来等

效施加额外的成本。复制“坏”数据点隐式赋予“坏”数据更高的权重，这种技术可以用于任何分类器中。

图21-6展示了一个如果将有毒蘑菇错分为可食用蘑菇，则有10倍惩罚的决策树。这个树以牺牲判断无毒蘑菇的准确率来使判断毒蘑菇完全正确。“安全比犯错好”。无偏和有偏的决策树混淆矩阵如图21-7所示。

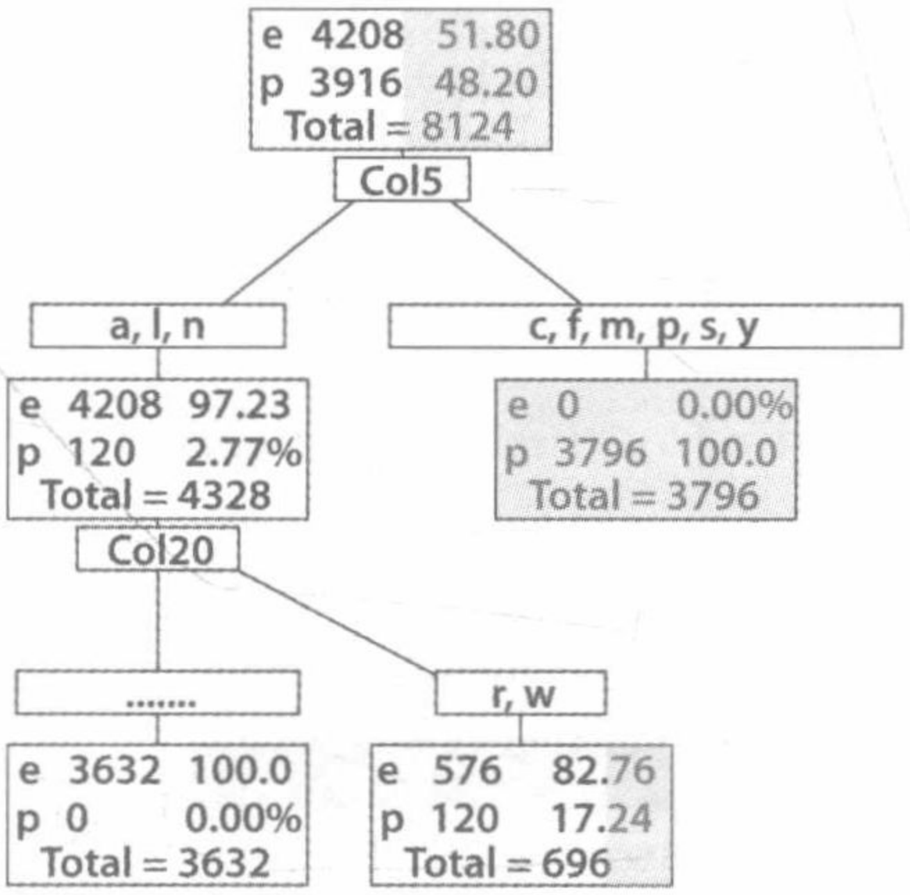


图21-6. 一个食用蘑菇分类器，如果将有毒蘑菇错分为可食用蘑菇，则有10倍的惩罚。请注意最右下边的矩形，虽然大部分是可食用蘑菇，但是达不到有毒蘑菇的10倍，所以仍被认为是不可食用的

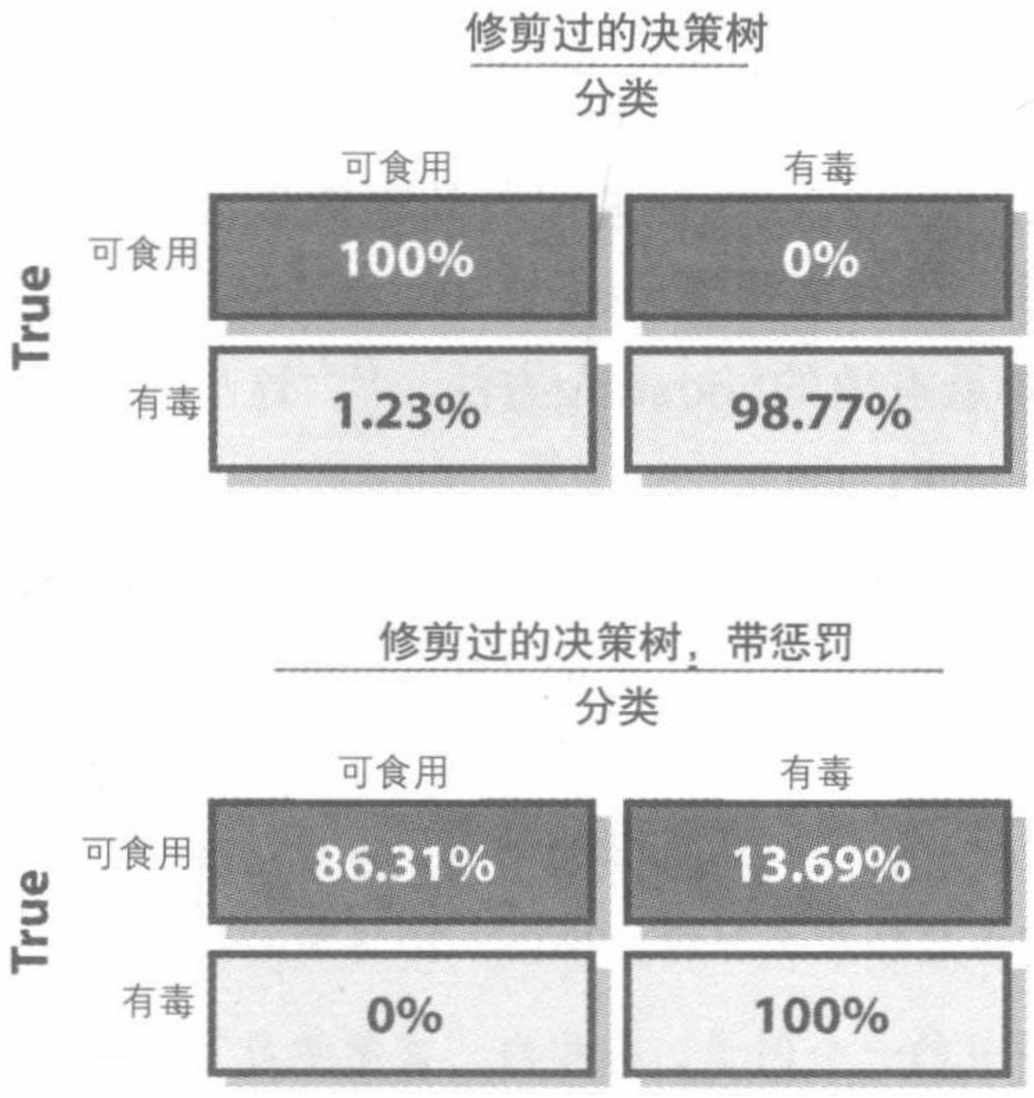


图21-7. 用于区分可食用蘑菇的修剪决策树的混淆矩阵。无惩罚的分类获得了更好的总识别率（上图），但是会将一些有毒蘑菇误分为可食用蘑菇；有惩罚的分类虽然总的识别率不是那么高（下图），但是不会将有毒蘑菇误分为可食用菌

Boosting方法

决策树很有用，但并不是最好的分类器。在本节和下节，我们讲述两种技术：boosting和随机森林。它们在内部使用决策树，所以继承了树的很多有用的性能（能够处理混合数据类型、没有归一化的数据、特征缺失）。这些技术能够获得相当好的性能，因此它们通常是库中最好用的监督学习算法。[注21]

在监督学习领域，有一个叫统计提升（statistical boosting）的元学习算法（meta-learning，1988年首次由Michael Kerns[译注1]阐述），Kerns想知道可不可能从很多弱分类器中学习到一个强分类器。“弱分类器”的输出与真正的分类方法只有很弱的相关性，而“强分类器”的输出与真正的分类密切相关。因此，弱和强只是统计意义上的定义。

第一个boosting算法叫AdaBossting，由Freund和Schapire[Freund97]提出。OpenCV包含4种类型的boosting，如表21-2所示。

表21-2：可用的boosting方法,每一个都是AdaBoost的变种

boosting 方法	OpenCV枚举值
Discrete AdaBoost	`cv::ml::Boost::DISCRETE`
Real AdaBoost	`cv::ml::Boost::REAL`
LogitBoost	`cv::ml::Boost::LOGIT`
Gentle AdaBoost	`cv::ml::Boost::GENTLE`

在大多数情况下，稍做改动的`AdaBoost()`的“real””和“gentle”变体最有效。*Real AdaBoost*是一种利用置信度预测并且能很好处理分类数据的技术。*Gentle AdaBoost*把异常数据点的的权重设置的较小，因此处理回归问题效果很好，而 *LogitBoost*在回归问题上也有不错的效果。因为只需要设置一个标志，所以完全可以在数据集上尝试所有的boosting类型，然后选择最有效的boosting方法。[注22]这里我们要介绍原始的AdaBoost。

注21：“没有免费午餐”定理告诉我们，没有一个先验的“最佳”分类器。但对于许多以专注于视觉为数据集，boosting和随机树表现相当好。

译注1：宾大网络数据科学中心主管，Hunch网站投资人兼咨询顾问，1991—2001年间，在AT & T贝尔实验室研究机器学习和人工智能。2016—2018年，兼作MANA Partners的首席科学家。

注22：这个程序是机器学习的一个例子，被称为“玄学学习”（voodoo learning）或“巫术编程”（voodoo programming）。虽然没有原则，但通常是实现最佳性能的有效方法。有时候，经过仔细的思考，可以弄清楚为什么最好的方法是最好的，这可以引发我们进一步了解数据。有时不是。

对于分类问题，OpenCV所实现的boosting是一个多分类器，[注23]与决策树或随机树分类器不同，它可以一次性处理多个类。

请注意：OpenCV执行的boosting算法是一个两类分类器（不像决策树和随机森林）。还有LogitBoost和GentleBoost，不仅可以用来解决二分类问题，还可以用于回归问题。

AdaBoost

boosting算法训练了N_w个弱分类器，h_w, $w\varepsilon\{1, \cdots, N_w\}$。这些弱分类器很简单。大多数情况下，它们只是包含一次划分（称为决策桩)）或仅有几次划分（可能到3次）的决策树。最后做决定的时候将复制权重α_w给每个分类器。AdaBoost训练时输入的特征向量是$\vec{x}_i$，向量的类别标签是（这儿，$i\varepsilon\{1, \cdots, N_t\}$，$N$是样本总数），且$y_i\varepsilon\{-1, +1\}$。首先，我们初始化数据样本的权值$D_w(i)$来告诉分类器。将一个数据点分类错误的代价是多少。boosting算法的主要特征是，在训练过程之中，这个代价将会更新，使得后来的弱分类器更加关注与前面的分类器没有分对的数据点。算法如下。

1. $D_w(i) = 1 / N_t, i = 1, \cdots, N_t$
2. 对于$w\varepsilon\{1, \cdots, N_w\}$则有：
   a. 寻找使得权重为$D_w(i)$的总错误最小的分类器h_w：
      求$h = \text{argmin}_{h_j}\varepsilon_j$，这里$\varepsilon_j = \sum_i^{Nt} D_t(i)$（其中$y_i \neq h_j(\vec{x}_j)$），如果最小错误满足$\varepsilon < 0.5$;，则继续，否则退出。
   b. 设置的权重$\alpha_i = \frac{1}{2}\log\left[\frac{1-\varepsilon_i}{\varepsilon_i}\right]$，这儿$\varepsilon_i$是步骤2a中的最小错误。
   c. 更新数据点的权重：

$$D_{w+1}(i) = \frac{1}{Z_w} D_w(i) e^{-\alpha_w y_i h_w(\vec{x}_i)}$$

这里，Z_w将所有数据点$i\varepsilon\{1, \cdots, N_t\}$,的权重归一化，所以对于所有的$w$，有$\sum_{i=1}^{N_t} D_w(i) = 1$。

注意：在2a步骤，如果找不到小于20%错误率的分类器，则停止，表明可能需要更好的特征。

当前面讲的训练算法结束之后，最后的强分类器接受输入向量$\vec{x}$，使用所有弱分类器h_w的加权和来进行分类。

$$H(x) = \text{sign}\left(\sum_{w=1}^{T} \alpha_w h_w(\vec{x})\right)$$

注23：有一个称为“展开”的技巧可用于适应N类分类问题的任何二分类器（包括boosting），但这会使训练和预测的开销显著增加，详请参见.../opencv/samples/c/letter_recog.cpp。

这里，符号函数把所有正数变为1，把所有的负数变为-1（0仍然为0）。显而易见，刚训练过的决策树中叶子节点的值以尺度α_i缩放，然后$H(x)$作为符号x在弱分类器上的响应的总和，必然会减小。

Boosting代码

这一类代码类似于决策树的代码，cv::ml::Boost类派生自cv::ml::DTrees，并带有一些额外的控制参数。正如我们在库中其他地方看到的那样，当调用cv::ml::Boost::create()时，返回值为一个cv::Ptr<cv::ml::Boost>类型的对象，但是这个对象实际上是一个指向（通常是不可见的）cv::ml::BoostImpl类的对象指针。

```
// Somewhere above..
// namespace cv {
// namespace ml {
//
class BoostImpl : public Boost {  // cv::ml::Boost is derived from cv::ml::DTrees

public:

  // types of boosting
  // (Inherited from cv::ml::Boost)
  //
  //enum Types {
  //  DISCRETE = 0,
  //  REAL     = 1,
  //  LOGIT    = 2,
  //  GENTLE   = 3
  //};

  // get/set one of boosting types:
  //
  int    getBoostType() const;        // get type: DISCRETE, REAL, LOGIT, GENTLE
  int    getWeakCount() const;        // get the number of weak classifiers
  double getWeightTrimRate() const;   // get the trimming rate, see text

  void setBoostType(int val);         // get type: DISCRETE, REAL, LOGIT, GENTLE
  void setWeakCount(int val);         // get the number of weak classifiers
  void setWeightTrimRate(double val); // get the trimming rate, see text

  ...

  // from class Boost
  //
  //static Ptr<Boost> create(); // The algorithm "constructor",
  //                            //  returns Ptr<BoostImpl>

};
```

成员setWeakCount()设置用于最终构成强分类器的弱分类器的数量，其默认值为100。

弱分类器的数量不同于每个分类器的最大深度。后者由setMaxDepth()控制，表示单个弱

分类器可以具有的最大层数。如前所述，最常见的值为1，在这种情况下，这些小的树结构只是用来占位的，即只包含一个选择。

下一个参数是权重消减率，它可以使计算更快，更有效率，随着训练的进行，许多数据点变得不重要了。也就是说，第i个数据点的权重$D_t(i)$变得非常小。setWeightTrimRate()方法设置介于0和1（含）之间的阈值，这个阈值用于在给定的增强迭代中丢弃一些训练样本。例如，假设权重微调率设置为0.95（默认值）。这意味着权重为大小为前95%的“最重”样本（即具有最大权重的样本）会进入下一次训练，而剩余的权重“最轻”的5%的样本被暂时排除在下一次迭代之外。注意是从下一次迭代，而不是永久丢弃样本。当训练下一个弱分类器时，对所有样本重新计算权重，因此一些先前不重要的样本可能返回到下一个训练集。通常，由于修剪，只有约20%的样本等参加了每一轮的训练，因此训练加速了5倍左右。要关闭此功能，请调用setWeightTrimRate(1.0)。

请注意，cv::ml::BoostImpl的参数是通过cv::ml::Boost从cv::ml::DTrees继承的，因此我们可以通过继承的接口函数设置与决策树相关的参数。总体而言，boosting模型的训练、预测方式与cv::ml::DTrees或来自ml模块的任何其他StatModel派生类完全相同。

OpenCV包中的代码.../*opencv/samples/cpp/letter_recog.cpp*是一个boosting的使用示例。训练代码片段如示例21-2所示。本示例使用分类器尝试从公共数据集识别$a-z$字符。该数据集有20 000个条目，每个条目有16个特征和一个“结果”。其中特征是浮点数，结果是一个字符。因为boosting只能用于两类区别，所以该程序需要使用“unrolling “技术，之前我们曾遇到过这个技术，在这里将详细讨论它。

在循环展开时，数据集从一组训练数据扩展到26组，每组都扩展为一次，现在将响应变为了一个特征。同时，这些扩展向量的新响应现在只有1或0：true或false。以这种方式训练分类器分类器，实际上是回答这个问题：通过学习关系$\{\vec{x}_i, y_i\}$和$\{\vec{x}_i, not \quad y_i\} = false$能否判断$\vec{x}_i$等于$z_i$。[注24]详见示例21-2。

示例21-2：boosted分类器的训练片段

```
...
cv::Mat var_type( 1, var_count + 2, CV_8U );    // var_count is # features (16 here)
var_type.setTo( cv::Scalar::all(VAR_ORDERED) );
var_type.at<uchar>(var_count) = var_type.at<uchar>(var_count+1) = VAR_CATEGORICAL;
...
```

首先要做的是创建数组var_type，说明如何处理每个特征和结果（示例21-2）。然后创建训练数据结构。它存储的内容比你想象的更广泛。不仅存储着var_count（在这种情

注24：同样，可以说，分类器正在接受训练，将$\vec{x}_i \rightarrow y_i$（即“向量$\vec{x}_i$暗示了结果$y_i$”）的命题分为两类，哪些是真命题，哪些假命题。

况下，恰好是16）个原始数据，还存储着响应。此外，在循环展开之前，还有一个额外的列用于存储字母字符响应。

```
cv::Ptr<cv::ml::TrainData> tdata = cv::ml::TrainData::create(
  new_data,                    // extended, 26x as many vectors, each contains y_i
  ROW_SAMPLE,                  // feature vectors are stored as rows
  new_responses,               // extended, 26x as many vectors, true or false
  cv::noArray(),               // active variable index, here just "all"
  cv::noArray(),               // active sample index, here just "all"
  cv::noArray(),               // sample weights, here just "all the same"
  var_type                     // extended, has 16+2 entries now
  );
```

接下来要做的是构建分类器。除了大多数很常见的用法，值得注意的是priors。此时得到错误答案的代价被提高到原来的25倍，所以有时得到一个错误答案，不是因为一些字母是错误的，而是因为循环展开。这就是说，与其说“字母”真的不是“字母”的，不如说是25倍的权重导致“字母”不是字母。因为有25倍的向量执行“负”规则，所以“正”规则需要相应的得到更多的权重。[注25]

```
vector<double> priors(2);
priors[0] = 1;  // For false (0) answers
priors[1] = 25  // For true (1) answers

model = cv::ml::Boost::create();
model->setBoostType( cv::ml::Boost::GENTLE );
model->setWeakCount( 100 );
model->setWeightTrimRate( 0.95 );
model->setMaxDepth( 5 );
model->setUseSurrogates( false );

cout << "Training the classifier (may take a few minutes)...\n";
model->setPriors( cv::Mat(priors) );

model->train( tdata );
```

boosting的预测函数也与决策树类似，使用model->predict()。如前所述，在boosting的场景中，该方法计算弱分类器响应的加权和，获得和的符号，然后将其转换为输出类标签。在某些情况下，获取实际总和值可能是有用的，例如评估决策的置信度。为此，要将cv::ml::StatModel::RAW_OUTPUT标志传递给预测函数。事实上，这是非常必要的。在处理展开数据时，获得两个（或多个）“真实”响应并不罕见。在这种情况下，通常选择置信度最高的答案。

```
Mat temp_sample( 1, var_count + 1, CV_32F ); // An extended sample "proposition"
float* tptr = temp_sample.ptr<float>();      // Pointer to start of proposition
```

注25：回想一下，我们以前看到的是，乘以正数或负数相当于以前的权重。在这种情况下，由于不同的原因（展开），我们乘以负向量的数量，所以为了补偿负面例子作用的增加而增大了正面样本。

```
double correct_train _answers = 0, correct_test _answers = 0;

for( i = 0; i < nsamples_all; i++ ) {

  int          best_class = 0;              // Strongest proposition found so far
  double       max_sum    = -DBL_MAX;       // Strength of current best prop
  const float* ptr        = data.ptr<float>(i);  // Points at current sample

  // Copy features from current sample into temp extended sample
  //
  for( k = 0; k < var_count; k++ ) tptr[k] = ptr[k];

  // Add class to sample proposition, then make a prediction for this proposition
  // If this proposition is more true than any previous one, then record this
  // one as the new "best".
  //
  for( j = 0; j < class_count; j++ ) {
    tptr[var_count] = (float)j;
    float s = model->predict(
      temp_sample, noArray(), StatModel::RAW_OUTPUT
    );
    if( max_sum < s ) { max_sum = s; best_class = j + 'A'; }
  }

  // If the strongest (truest) proposition matched the correct response, then
  // score 1, else 0.
  //
  double r = std::abs( best_class - responses.at<int>(i) ) < FLT_EPSILON ? 1 : 0;

    // If we are still in the train samples, record one more correct train result.
    // Otherwise, record one more correct test result.
    // Hope nobody shuffled the samples!
    //
    if( i < ntrain_samples )
      correct_train _answers += r;
    else
      correct_test  _answers += r;
  }
```

当然，这不是处理多分类问题的最快或最简便的方法。随机森林可能是一个不错的选择，下面我们要进一步介绍。

随机森林

OpenCV包含一个随机森林类，它是根据Leo Breiman的随机森林理论实现的。[注26]随机森林可以通过收集很多树的子节点对各个类别投票，然后选择获得最多投票的类别作为判断结果。通过计算“森林”的所有子节点上的值的平均值来解决回归问题。随机森林包

注26：Breiman的大多数随机森林工作成果可以从一个网站上很方便地收集：*http://www.stat.berkeley.edu/users/breiman/RandomForests/cc_home.htm*。

含随机选择的一些决策树，在ML库建立的时候，它是当时最好的分类器之一。即使是在共享内存的系统上，随机森林在并行执行方面也很大的潜力可挖。这个特征使它在未来的发展中占据优势。随机森林建立时的基本子系统也是决策树。在建立决策树时会一直继续下去直到数据纯净。因此（图20-3的上半部分—过拟合），尽管每个树都很好学习了训练数据，但是各个树之间仍有很大的不同。为了消除这些不同，我们把这些树放在一起来平均（因此叫随机森林）。

当然，如果所有的树都很相似，随机森林就没有很大的作用。为了克服这点，随机森林通过在树的建立过程中随机选择特征子集来使各个树不相同。例如，一个目标识别树可以有很多可能的特征：颜色、质地、倾斜度、倾斜方向、变量和值的比等。树的每个节点可以从这些特征中随机的选择子集来决定怎样最好地切分数据。每个后来的节点获得新的、随机选择的特征子集来切分。随机子集的规摸一般是特征数*t*的开方。因此，如果我们有100个可能的特征，每个对接点将随机选择10个特征，并利用这10个特征对数据进行最好地切分。为了提高鲁棒性，随机森林使用袋外（out of bag）方法来检验切分。给定任何节点，训练是发生在一个随机选择然后替换的数据子集上进行的。[注27]没有选到的数据叫“out of bag”（OOB）数据，将用于估计分叉的性能。OOB数据一般为所有数据的1/3。

如同其他基于树的方法，随机森林继承了树的很多属性：缺失值的代理切分。可以处理标签和数值特征．不需要归一化数据，容易获得对预测很重要的变量 。随机森林还使用OOB错误来估计它在处理未曾见过的数据时的性能。如果训练数据与测试数据的分布相似，00B性能预测会更精确．

最后，随机森林可以用来确定两个数据样本的亲近度（是相似度，而不是距离）。方法如下：(1)将数据样本放入判别树；(2)计算它们到达同样的叶子（子节点）的次数；(3)用相同的叶子数除以树的总数，亲近度为1意味着完全亲近，0意味着完全不亲近。亲近度可以用来检测异常（样本与其他的很不相似）或者用来聚类（把相似的样本聚在一起）。

随机森林的代码

相信读者现在对于ML模块的工作原理已经很熟悉了，随机树森林也不例外。我们从类定义开始介绍。类cv::ml::RTreesImpl继承自cv::ml::RTrees，而cv::ml::RTrees本身继承自cv::ml::DTrees类。

```
// Somewhere above..
// namespace cv {
//   namespace ml {
//
class RTreesImpl: public RTrees {  // cv::ml::RTrees is derived from cv::ml::DTrees
```

注27：这意味着一些数据点可能是随机重复的。

```
public:

  // whether variable importance should be computed during the training
  // makes the training noticeably slower
  //
  bool getCalculateVarImportance() const;          // Get if importance should be
                                                   //  computed during training
  int  getActiveVarCount() const;                  // Get N Var for each split
  TermCriteria getTermCriteria() const;            // Get max N trees or accuracy
  void setCalculateVarImportance( bool val );      // Get if importance should be
                                                   //  computed during training
  void setActiveVarCount( int val );               // Set N Var for each split
  void setTermCriteria( const TermCriteria& val ); // Set max N trees or accuracy

  ...

  Mat getVarImportance() const;

  ...

  // from class RTrees
  //
  //static Ptr<RTrees> create(); // The algorithm "constructor",
  //                             //  returns Ptr<RTreesImpl>
};
```

方法setCalculateVarImportance()用来设置是否在训练期间计算每个特征变量的重要性（会有一些额外时间开销）。

图21-8显示了我们之前讨论的蘑菇数据集计算得到的变量重要性。

Variable Name	Random	Boosting	DecisionTree
Col5	100.0	100.0	100.0
Col20	35.2	58.89	57.37
Col21	16.47	6.11	34.51
Col19	13.35	4.57	26.11
Col9›	13.01	43.15	45.96
Col13	10.02	24.47	26.85
Col8	9.52	37.51	42.28
Col12	9.09	27.66	28.90
Col22	8.29	0.28	20.00
Col7	6.08	0.10	21.33
Col15	4.06	1.84	21.41
Col11	3.52	0.44	16.29
Col4	3.12		14.67
Col14	2.98	0.25	20.81
Col18	2.68		0.70
Col3	2.56	0.11	9.15
Col2	2.22	0.39	12.14
Col10	1.79		2.67
Col1	0.41	0.24	7.26
Col17	0.18	0.32	0.54
Col0			
Col6			
Col16			

图21-8. 通过随机森林，boosting和决策树计算出蘑菇数据集的变量重要性：随机森林使用较少的重要变量获得了最好的预测性能（在随机选择的20%数据上测试得到100%的准确率）。注意，在OpenCV 3.x中，只能将变量重要性应用于随机森林，而不是决策树或boosting

setActiveVarCount()方法是用来设置给定节点上要测试特征集规模，这些特征集是随机选择的。如果用户未明确设置，则通常将其设置为总数量的平方根。

使用setTermCriteria()，可以设置森林包含的树的最大数量（cv::TermCriteria::maxIter）。和OOB错误的终止条件一样，该方法可以停止继续生成树（cv::TermCriteria::epsilon）。用户可以应用两个停止标准中的一个或两个都用（通常都是cv::TERMCRIT_ITER | cv::TERMCRIT_EPS）。否则，随机森林的训练就与决策树、boosting的训练形式相同了。

之前在boosting.../*opencv/samples/cpp/letter_recog.cpp*下，我们观察到有同样的多类学习示例，它也支持训练随机森林。以下摘录了一些突出显著的点。请注意，与boosting不同，我们可以利用随机森林直接对多类数据进行训练。

```
using namespace cv;
...
Ptr<ml::RTrees> forest = ml::RTrees::create();
forest->setMaxDepth(10);
forest->setMinSampleCount(10);
forest->setMaxCategories(15);
forest->setCalculateVarImportance(true);
forest->setActiveVarCount(4);
forest->setTermCriteria(
  TermCriteria(
    TermCriteria::MAX_ITER+TermCriteria::EPSILON,
    100,
    0.01
  )
);
forest->train(tdata, 0);

...
```

随机森林预测与其他所有模型不同。下面是一个来自*letter_recog.cpp*文件的预测调用示例：

```
...

for( int i = 0; i < nsamples_all; i++ ) {

  cv::Mat sample = mydata.row( i );

  float r = forest->predict( &sample );
r = fabs( (float)r - responses.at<float>[i]) <= FLT_EPSILON ? 1 : 0;

// Accumulate some statistics using 'r'
  if( i < ntrain_samples )
    correct_train _answers += r;
  else
    correct_test  _answers += r;

}
```

在这段代码中，返回变量r被转换为一项正确的预测。用户可以使用通用方法cv::ml::StatModel::calcError()计算可以得到相同的统计信息。

最后介绍随机森林的分析和实用函数。假设在训练之前调用了setCalculateVarImportance(true)，则可以使用cv::ml::RTrees成员函数cv::Mat RTree::getVarImportance()const;来获取每个变量的相对重要性。在这种情况下，返回值是一个向量，包含每个特征的相对重要性。

随机森林的使用

我们已经说过，随机森林算法经常获得最好（或者最好的之一）的性能，但是最好的策略仍然是在训练集上尝试很多分类器。我们在蘑菇数据集合中使用了随机森林、boosting 和决策树。从 8124 个数据样本中，我们随机抽取了 1624 个测试样本，其他的作为训练样本．在训练了这三个基干树的分类器之后，我们获得了表21-3的测试结果。蘑菇数据集相当简单，所以尽管随机森林的性能最好，我们也不能言之凿凿地说随机森林分类器在这个实例中是最好的。

表21-3：三个基于树的分类器对 OpenCV 蘑菇数据集（使用 1624 个随机选择的样本作为测试样本，没有对错误分类毒蘑菇进行惩罚）的实验结果

分类器	实验结果
随机森林	100%
AdaBoost	99%
决策树	98%

如图21-8所示，最有意思的是变量重要性（也可通过分类器测得），从图中可以看出，随机森林和 boosting各自所用的重要变量都明显少于决策树。重要性超过15%的特征中，随机森林只使用了3个变量，boosting使用了6个，决策树却使用了13个。因此，我们可以收缩特征集大小，在不损失性能的前提下减少计算量和内存使用。当然，在决策树中，只需要建立一裸树，而在随机森林和AdaBoost算法中需要建立多棵树。因此，哪个算法拥有最少的开销与数据有关。

期望最大化算法

期望最大化（Expectation Maximization，EM）是一种流行的无监督聚类技术。EM的基本概念类似于*K*-Means算法，是依据混合高斯分布来建模的。[注28]然而，学习这些高斯分量的过程是两个交替的迭代过程，分别称为“期望最大化的阶段”（算法也因此得名）。

注28：虽然期望最大化算法实际上比高斯混合更常见，但在这里，OpenCV只支持我们描述的具有高斯混合的EM的特殊情况。

在制定EM算法的过程中，训练集中的每个数据点与隐变量相关联，隐变量代表该变量的高斯混合分量，该变量采用其观测值（这种变量通常称为权值）。理想情况下，需要先计算高斯分量的参数以及权值，才能使得它们的观测变量获得最大似然解。然而实际上，通常不可能同时最大化所有变量。EM算法与*K*-means的不同之处在于分配给数据点的任务不一定是要最接近的集群中心。

EM算法把问题的两个方面分别处理，先在E步骤求期望，然后在M步骤实现高斯分量的参数最大化，在这两个步骤之间交替执行，直到收敛。

期望最大化和cv::EM()

在OpenCV中，EM算法在cv::ml::EM类中实现，声明如下：

```
// Somewhere above..
// namespace cv {
//   namespace ml {
//
class EMImpl: public EM {       // cv::ml::EM is derived from cv::ml::StatModel

public:

  // Types of covariation matrices
  // (Inherited from cv::ml::EM)
  //
  //enum Types {
  //  COV_MAT_SPHERICAL = 0,
  //  COV_MAT_DIAGONAL  = 1,
  //  COV_MAT_GENERIC   = 2,
  //  COV_MAT_DEFAULT   = COV_MAT_DIAGONAL
  //};

  // get/set the number of clusters/mixtures (5 by default)
  //
  int  getClustersNumber() const;                  // Get number of clusters
  int  getCovarianceMatrixType() const;            // Get cov. mtx. type
  TermCriteria getTermCriteria() const;            // Get max iter/accuracy

  void setClustersNumber( int val ) ;              // Set number of clusters
  void setCovarianceMatrixType( int val );         // Set cov. mtx. type
  void setTermCriteria( const TermCriteria& val ); // Set max iter/accuracy

  ...

  // the prediction method, see below
  //
  Vec2d predict2(
    InputArray  sample,
    OutputArray probs
  ) const;
  bool trainEM(...); // see below
  bool trainE(...);  // see below
  bool trainM(...);  // see below
```

```
    ...

    // from class EM
    //
    //static Ptr<EM> create(); // The algorithm "constructor",
    //                         //  returns Ptr<EMImpl>

};
```

在构建和配置cv::EM对象时，用户必须通过setClusterNumber()方法告诉算法有多少个聚类存在。如果是K-means算法，则可以分别使用不同数量的聚类进行测试，但是基本的cv::EM对象一次只能处理一个特定数量的集群。

成员函数setCovarianceMatrixType()指定了应用于EM算法的协方差矩阵，该矩阵与高斯混合模型的各个组件相关联。此参数的取值有以下三种：

- cv::ml::EM::COV_MAT_SPHERICAL
- cv::ml::EM::COV_MAT_DIAGONAL
- cv::ml::EM::COV_MAT_GENERIC

在第一种情况下，假设每个混合分量是旋转对称的。这意味着它只有一个可以在M步骤中最大化的自由参数。协方差就是该参数乘以一个单位矩阵。在第二种情况下，cv::EM::COV_MAT_DIAGONAL（这是默认值），矩阵是对角阵，因此参数的数量等于矩阵的维数（即数据的维数）。最后，介绍cv::EM::COV_MAT_GENERIC的情况，其中每个协方差矩阵的特征用任意对称矩阵中的$(N_d^2 - N_d) / 2$个变量表示。一般来说，训练的模型的复杂性很大程度上取决于可用的数据量。实际上，如果数据很少，cv::EM::COV_MAT_SPHERICAL可能是个不错的选择。相反，除非有大量的数据，否则cv::EM::COV_MAT_GENERIC可能不是一个好主意。



数据量有多少是不足，有多少才叫大量呢？如果你正在有N_k个聚类的N_d维数据上使用EM算法，那么很容易计算出要解决的自由变量的数量。在对称分布（COV_MAT_SPHERICAL）的情况下，仅需计算每个聚类的一个协方差以及聚类的中心位置，总共$N_k(N_d + 1)$个变量。在协方差矩阵为对角矩阵时（COV_MAT_DIAGONAL），每个聚类的协方差有N_d个自由度，或总共$N_k (N_d + N_d) = 2N_kN_d$个总变量。在完全一般的协方差矩阵（COV_MAT_GENERIC）的情况下，每个聚类的协方差有$(N_d^2 - N_d) / 2$个自由度，$N_k (N_d + (N_d^2 - N_d) / 2) = N_kN_d^2 + \frac{1}{2}N_kN_d$个需要求解的总变量。因此，如果保守估计，在一般情况下求解的花费是在最受限制情况下的$O(N_d)$倍，然而即使是最有效的学习算法，计算量随着数据量的增长，也是达不到线性标准的。

EM算法最终需要setTermCriteria()函数来设置终止标准。所需的termCrit参数通常是cv::TermCriteria类型，该参数指定允许的最大迭代次数或设置当似然性中的最大小时停止迭代。

在创建一个cv::EM对象的实例并设置参数之后，就可以使用EM算法中的trainEM()，trainE()和trainM()等三种训练方法之一进行训练：

```
bool cv::ml::EM::train(
  cv::InputArray  samples,
  cv::OutputArray logLikelihoods = cv::noArray(),
  cv::OutputArray labels         = cv::noArray(),
  cv::OutputArray probs          = cv::noArray()
);

virtual bool cv::ml::EM::trainE(
  cv::InputArray  samples,
  cv::InputArray  means0,
  cv::InputArray  covs0          = cv::noArray(),
  cv::InputArray  weights0       = cv::noArray(),
  cv::OutputArray logLikelihoods = cv::noArray(),
  cv::OutputArray labels         = cv::noArray(),
cv::OutputArray probs            = cv::noArray()
);

bool cv::ml::EM::trainM(
  cv::InputArray  samples,
  cv::InputArray  probs0,
  cv::OutputArray logLikelihoods = cv::noArray(),
  cv::OutputArray labels         = cv::noArray(),
cv::OutputArray probs            = cv::noArray()
);
```

cv::EM中的trainEM()方法输入样本数组，并返回响应（标签），似然性（logLikelihoods）和分配概率（probs）。这些响应保存在标签数组中，列中每行代表一个数据点。第i行上的值是一个整数，用于标识该数据点所属的聚类。该整数是在N_k个集群中计算成员概率中最大的（在调用构造函数时由nclusters设置）。probs数组实际上包含着这些独立概率，每行给出给定点的概率，每列表示该集群的概率（即，probs.at <i>(i,k)是点i是集群k的成员的概率）。最后，logLikelihoods数组中返回与每个点对应的似然性;这些似然性本质上是最终的观察模型出现的可能性。

顾名思义，返回的值实际上是似然性的自然对数。如果不需要这三个输出，则可以用cv::noArray()替换。模型训练之后就可以用于预测了。

除了刚刚描述的训练方法，还有两种其他的方法trainE()和trainM()。这些方法分别在单独执行E或M步骤，在使用trainE()的情况下用户要提供使用的初始模型，或者在trainM()的情况下提供一组初始聚类的分配概率。

在应用trainE()时，模型通过数组means0，covs0和weights0输入。数组的形式应为具N_k行*N_d列，其中N_k是聚类的数量，N_d是样本数据的维数。协方差的存储形式是一个包含N_k个独立数组的STL向量，并且每一个都是高斯分量k的$N_d \times N_d$的协方差矩阵。最后，weights0应该为一列和N_k行。第k行中的条目应该是与高斯分量k相关联的联合概率（与某些其他分量相反，任何随机样本服从分量k绘制的边缘概率）。

在使用trainM()时，用户必须提供probs0，即与每个点相关联的成员概率，其格式与之前所描述的train()计算的相同。这意味着probs0是一个$N_s \times N_k$数组，每个样本有一行，而该数据点的相应列中对应每个高斯分量的隶属概率。

模型训练好之后，就可以使用cv::EM::predict()方法让算法去预测一个新的点最有可能是属于哪个聚类。predict()的返回值将j cv::Vec2d类型。返回的向量的第一个参数将给出相关的概率，而第二个将给出聚类标签。

K近邻算法

K近邻是最简单的分类器技术之一。它只存储所有训练样本数据，如果需要分类一个新的样本数据，只需要找到它的K（k为整数）个最近邻的点，然后统计哪个类在k近邻点中频率最高，然后把该点标记为出现频率最高的类。K近邻还可用于回归；在这种情况下，返回的结果是与N_k个最近邻居值的平均值。[注29]该算法在OpenCV中的cv::ml::KNearest类中实现。KNN分类器技术很有效，但是它需要存储所有的训练集，因此它占用很大的内存，速度比较慢。使用这个算法把之前的训练集聚类来降低数据大小。读者如果对脑科学（和机器学习）中可能使用的动态自适应最近邻技术感兴趣，可以阅读Grossberg [Grossberg87]或Carpenter和Grossberg [Carpenter03]，进一步了解最新进展。

通过cv::ml::KNearest()应用K近邻算法

在OpenCV中，KNN算法通过cv::ml::KNearest类实现。cv::ml::KNearest的类声明具有以下形式：

```
// Somewhere above..
// namespace cv {
//   namespace ml {
//
class KNearestImpl: public KNearest { // cv::ml::KNearest is derived
                                      // from cv::ml::StatModel
```

注29：如果读者是机器学习算法的专家，那么就要注意，在进行回归时，KNN的OpenCV实现使用N_k个邻居点的未加权平均值。其他实现通常以不同的方式对这些点进行加权，例如通过与输入点的反向距离。

```
public:

  // (Inherited from cv::ml:: KNearest)
  //
  //enum Types {
  //  BRUTE_FORCE = 1,
  //  KDTREE      = 2
  //};

  // use K-nearest for regression or for classification

  //
  int  getDefaultK() const;          // Get the default number of neighbors
  bool getIsClassifier() const;      // Get if classifier, else it is regression
  int  getEmax() const;              // Get max "close" neighbors (KDTree)
  int  getAlgorithmType() const;     // Get whether brute-force or KD-Tree search
                                     //  (will be either BRUTE_FORCE or KDTREE)

  void setDefaultK( int val );       // Set the default number of neighbors
  void setIsClassifier( bool val );  // Set if classifier, else it is regression
  void setEmax( int val );           // Set max "close" neighbors (KDTree)
  void setAlgorithmType( int val ); // Set whether brute-force or KD-Tree search
                                     //  (must be either BRUTE_FORCE or KDTREE)
  ...

  // find the k nearest neighbors for each sample
  //
  float findNearest(
    InputArray  samples,
    int         k,
    OutputArray results,
    OutputArray neighborResponses = noArray(),
    OutputArray dist              = noArray()
  ) const = 0;

  ...

  // from class KNearest
  //
  //static Ptr<KNearest> create(); // The algorithm "constructor",
  //                               //  returns Ptr<KNearestImpl>
};
```

如果计划使用cv::ml::StatModel::predict而不是cv::ml::KNearest::findNearest()，那么setDefaultK()方法可以用来设置要考虑的邻居数，该参数被明确指定 为“k”（对应于我们前面讨论的KNN算法中的N_k）。

如果想要使用KNN进行回归（即使用离散的训练集拟合某些函数），则选择setIsClassifier()函数。此时应该在调用train()方法之前调用setIsClassifier(false)。

如果你通过调用cv::ml::StatModel::train()来训练KNN模型。trainData输入是包含N_s行和N_d列的通用数组，N_s是样本数，N_d是数据的维数。响应数组必须为N行单列。算法

如何处理此数据取决于setAlgorithmType()方法。可以使用BRUTE_FORCE或KDTREE调用此方法。

如果算法参数设置为BRUTE_FORCE，则将该训练数据简单作为数组存储，然后依次扫描，以便找到最近的邻居。而如果是KDTree，将使用BBF（最佳bin-first）算法（由D. Lowe引入），这在$N_d << log(N_s)$时更有效。

重要的是要注意，与ml模块中的许多其他模型不同，KNN模型可以在训练后用新的训练数据进行更新。为此，用户需要将标志cv::ml::StatModel::UPDATE_MODEL传递给train()方法。

模型训练好之后，用户可以使用cv::ml::KNearest类来进行预测。cv::ml::KNearest类提供了通用的predict()方法和特定的findNearest()，它相当于findNearest（samples, getDefaultK(), results, noArray(), noArray()）。findNearest()方法可以让用户检索有关邻居的一些附加信息。findNearest()方法定义如下：

```
virtual float cv::ml::KNearest::findNearest(
  cv::InputArray  samples,
  int             k,
  cv::OutputArray results,
  cv::OutputArray neighborResponses = cv::noArray(),
  cv::OutputArray dist              = cv::noArray()
  ) const = 0;
```

该方法一次处理一个或多个样本，样本参数可以为任意行，每行包含单个输入。用于比较k的值（参数k）可以是小于训练时给出的N_k的值的任何值。预测结果会被放置在数组中，它的每行每列都与样本中每个点（行）相对应。如果有的话，数组neighborResponses和dist将存放每个数据点的响应和到各个邻居的距离。它们中的每一行对应样本中的每个点，每一列对应该点的k个邻居。

两种方法都将返回一个浮点值；当samples中的点数量只有一个时，使用此值返回一个计算的响应。在这种情况下，根本不需要提供results数组存放结果。

多层感知机

多层感知机（Multilayer Perception，MLP）是一种神经网络。采取的是反向传播的权重更新规则，它在文本识别以及其他快速增长的任务中也算是表现最为优秀的几种方法之

注30：如果你希望在回归中使用自己的加权函数，可以使用此信息。

一。[注31]事实上，和其他几种基本的神经网络方法一样，MLP在当前不断发展的深度学习领域中起着重要的作用，并在近年来取得了巨大的成功。

MLP的预测速度非常快；评估一个新的输入只需要执行一系列的点积运算，再加上一个简单的非线性“压缩”函数。另一方面，在一个相当复杂的环境下，由于它依赖于梯度下降，需要通过调整层内数值分类节点之间潜在的大量加权连接关系来降低错误，它的训练可能相当慢。

多层感知机背后的基本思想来自生物学，对于哺乳动物神经网络的研究激发了神经元层的概念，其中每一个都需要从前一层得到一定数量的输入，并将这些输入与学习的权重相乘，并输出对这一加权输入的某种非线性变换（图21-9）。这种生物神经元的简化模型，通常被称为“人工神经元”（整体称为一个人工神经网络），其目的是让它拥有像生物神经元一样的结构，当将它应用到网络中时，我们希望这些网络可以像在生物系统中观察到的那样——理想情况下接近并达到人类神经系统——表现出学习和泛化能力。[注32]

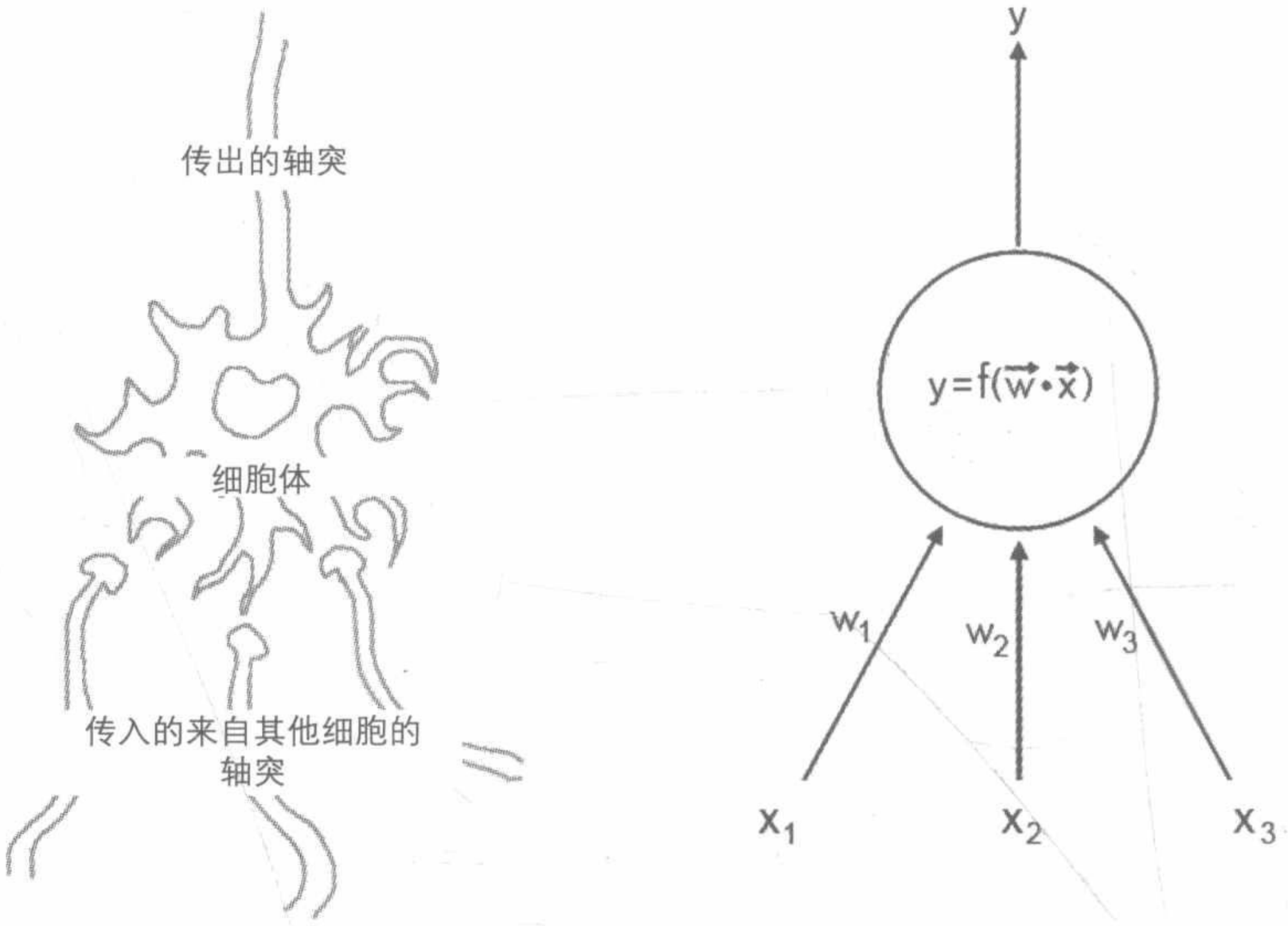


图21-9. 在MLP中用一个人工神经元和神经网络工具加一个典型的非线性激活函数*f*来模仿物理神经元的行为，*f*作用在输入的加权求和结果。在MLP中实现学习就体现在这些权重$\vec{w}$的调整中

注31：请注意，这是深层神经网络中使用的核心算法，但这并不能实现超过一个或两个隐藏层的深层神经网络。我们将尽快讨论目前正在添加到OpenCV中的深层神经网络的新支持。

注32：最好的情况是，目前的大多数神经网络以及深层学习体系结构都体现了大脑中一种可能的前馈处理流，当然，即使是这样，也忽略了处理过程中大量需要注意的点和其他一些偏差。大多数模型完全忽略了大脑中10倍或10倍以上的反馈过程，这可能与模拟有关，正如本书最后一章所推测的那样。

这些人工神经元，通常称为“简单节点”，聚合成一个多层网络（图21-10）以及一个训练它们的算法，构成了MLP算法的基础。这种多层网络可以采取多种形式， 最简单的方式称为“全连接多层网络”。[注33]在这样的网络中，对网络的所有输入都可以为第一层中的任何节点提供输入，[注34]同样，以此类推，第一层的所有输出都可以为第二层中的任何节点提供输入。这样的网络可以在任何层中有任意数量的节点，最后一层节点数等于表示结果所需响应函数的数量。

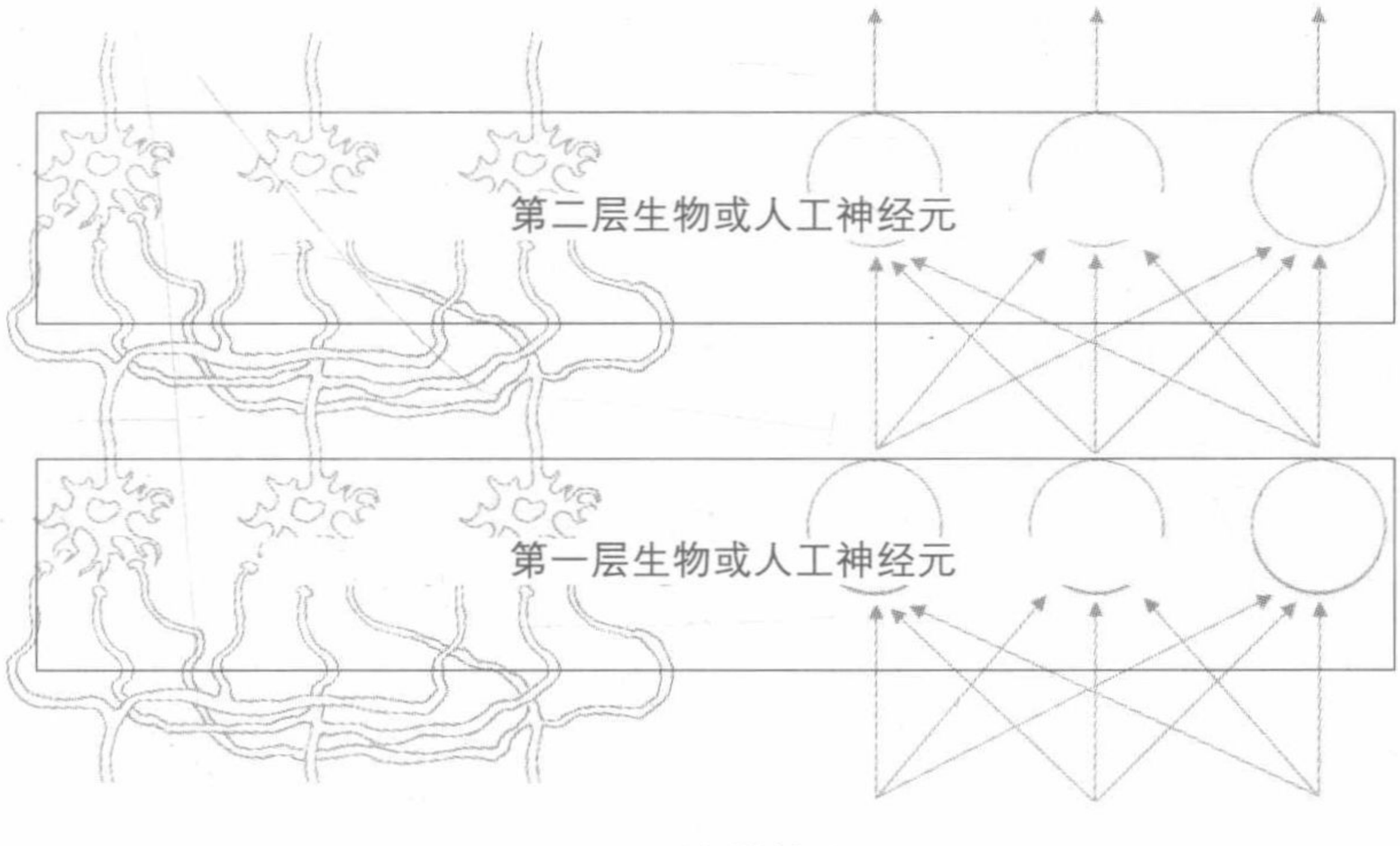


图21-10. 与生物神经元类比， 人工神经元可以被排列成层。该网络的输入以加权点积 $\vec{w} \cdot \vec{x}$ 的形式提供给第一层中的所有神经元，通过非线性函数 $f(w.x)$过滤，最终特定节点处产生输出。第一层神经元的输出将所有输入提供到下一层，依此类推

这种网络中的节点数可能非常大，并且权重的数量比节点数更多。假设一个有N_x个输入和N_y个输出（即最后一层节点的个数）的网络，总共有N_l，每层有N_n个节点（除了最后一层有N_y个节点）。这样的网络对于第一层的输入将有N_xN_n个权重，并且之后的每一层将有N_n^2个权重，直到最后一层权重为N_nN_y。最终，系统将有$N_n(N_x + N_y + (N_l - 2)N_n)$个不同的权重，数量级为$N_lN_n^2$。训练这种网络意味着找到所有这些权重的最优值，因此，根据训练的数据，对于任意给定的输入，神经网络的输出将尽可能接近于输入的值。

注33：如果你对此感兴趣，在这种情况下还有一点，我们称其为“全连接网络”也意味着所有权重都是不受约束的，因此他们完全彼此独立。

注34：在神经网络文献中，术语“输入层”有时用于表示第一层神经元，有时用于表示第一层之前的输入数。为了避免歧义，我们将完全避免使用这个术语。我们将始终使用术语“第一层”来表示第一层计算节点，其数量完全独立于网络的输入数量。

你可能会想到一种方法，可以通过一个简单的算法来解决这样一个多维优化问题，比如梯度下降，通过它，你可以从一些随机的值开始，并以一种贪婪的方式增量式地优化它们，直到没有任何提升为止。而这正是神经网络早期所做的。然而，考虑到即使是一个简单实用的网络中都会存在大量的参数，在实际使用中直接使用这种方法太慢了（或者说， 可以在一定时间内完成训练的网络太少从而无法使用）。直到反向传播算法发明[Rumelhart88]，对非常庞大的网络进行训练才成为可能。反向传播是梯度下降的一种特殊变体，它针对神经网络的特殊结构进行了优化。这种算法在OpenCV的MLP训练步骤中实现了。

反向传播

如图21-11所示，我们可以使用经过训练的反向传播网络来进行决策，或者在“底部”输入经过加权和 非线性变换的结构 （通常是sigmoid函数或整流函数）的变换来拟合函数。一个信号在网络中层层传递，直到在“顶部”产生输出。在训练模式中，我们使用一个损失函数来衡量网络的运行。在图21-11中，这个损失函数是目标值减去实际值的平方。反向传播地址的关键问题是如何更新神经网络中的权重，使得损失函数在数据集上最小。

$$\text{损失函数：} E = \frac{1}{2}\sum_{j=1}^{N}(t_j - y_j)^2$$

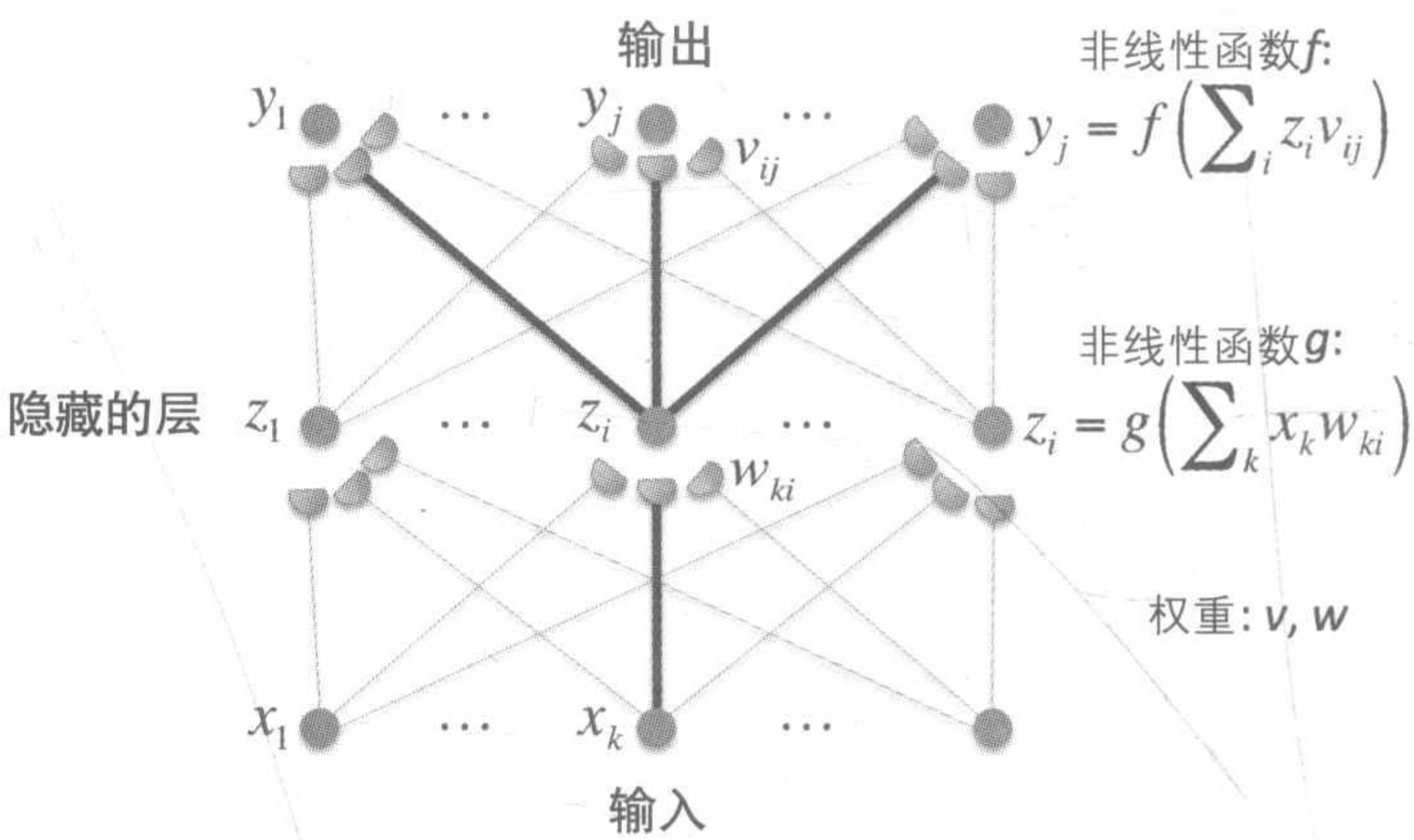


图21-11. 反向传播结构。其目的是更新权重v_{ij}, w_{ki}以尽量减少损失函数值。每个隐藏和输出节点得到加权激活值$\Sigma_i v_{ij} z_i$, $\Sigma_k w_{ki} x_k$，然后通过非线性函数g或f。实际上，这些非线性函数都是相同的，通常是S形或校正的函数，但考虑到一般性，它们可以是不同的

为了使损失函数最小化，运算中使用链式法则从上到下（“反向传播”）推导出权重与

顶部的损耗函数的关系。[注35]反向传播算法利用了多层感知机中拓扑结构的优势。它首先计算输出节点上的误差（你想要的和得到的偏差）。然后，它将这些信息向后传播给提供输出的节点。它们结合从输出节点得到的信息来计算所需的导数相对于它们自己的权重。然后将此信息再次传递到下一层输入层。通过这种方式，我们就可以只使用该节点的信息和从更接近其所连接的输出的一层中传递到该节点的信息，计算与每个权重局部相关的梯度部分。

在图21-11 的网络中， 输出权重变化函数是：

$$\frac{dE}{dv_{ij}} = -(t_j - y_j)z_i f'\left(\sum_i v_{ij} z_i\right)$$

隐藏单位权重变化函数是：

$$\frac{dE}{dw_{ki}} = \sum_i -(t_j - y_j)v_{ij} f'\left(\sum_i v_{ij} z_i\right) x_k g'\left(\sum_k w_{ki} x_k\right)$$

其中，t_j表示目标值， 输入、隐藏和输出激活值分别是x_k, z_i, y_j。最后两个激活由具有输入权的非线性函数g和f以及它们各自的权重v_{ij}, w_{ki}。这构成了反向传播算法的基本核心。



通过工具包Caffe，Tensor-Flow，CNTK，Torch和Theano所产生的深层神经网络，OpenCV现在支持阅读和有效运行深度神经网络，请参阅附录B中引用的opencv_contrib/modules/dnn。深度网络通常被分解成块卷积层，以节省必须学习的参数数量。结合其他训练技巧（如批量归一化），剔除和调整预先训练的早期层次，增加计算能力和巨大的数据存储，这使得人机界面性能方面近年来取得了显著的进步。

Rprop算法

Rprop（弹性反向传播）是由Martin Riedmiller和Heinrich Braun [Riedmiller93]提出的，并提供了一个通常很优秀的方法用于刚刚描述的反向传播算法的更新步骤。反向传播和Rprop之间的本质区别是，当反向传播显式计算每个权重更新的每个步骤的大小时，Rprop 只使用计算更新的符号（方向）。在计算更新的大小时，Rprop 计算并存储更新

注35：感兴趣的读者可以在任何关于机器学习的好书中找到这个算法更深层次的细节（例如，[Bishop07]）。有关更多的技术细节以及有关如何有效地使用MLP文本和对象的详细信息，读者可以直接查看LeCun、Bottou、Bengio和Haffner [LeCun98a]。在LeCun、Bottou和Muller [LeCun98b]中给出了实现和调整细节。在Hinton、Osindero和Teh [Hinton06]中可以找到传播概率的类似大脑分层网络的最新研究。

的方向，然后通过一个标准步骤 （称为Δ_0） 进行一次更新更改。在下一次传递中， 如果方向改变（这意味着算法实质上超出了目标），那么该步骤将被一个常量比例因子（通常称为η^-） 所简化。如果步骤不更改符号，则算法会通过乘以不同的比例因子（通常称为η^+）来增加下一次迭代的步长。显然η^-必须少于1且η^+必须大于1；这些参数的规范值分别为0.5到1.2。在OpenCV实践中，该步骤决不允许小于某些绝对最小值， 也不能大于某些绝对最大值 （分别称为Δ_{min}和Δ_{max}）。

通过cv::ml::ANN_MLP实现人工神经网络和反向传播

这个基于多层感知机学习的OpenCV工具可以用来构建和评估网络。并应用之前讲到的反向传播算法（或Rprop算法）来训练网络。所有这些都包含在cv::ml::ANN_MLP类中，并且它继承了统计学习基类的一般性接口。cv::ml::ANN_MLP类有一般的隐藏实现类，在这种情况下有以下定义：

```
// Somewhere above..
// namespace cv {
//   namespace ml {
//
class ANN_MLPImpl: public ANN_MLP {   // cv::ml::ANN_MLP is derived
                                      // from cv::ml::StatModel
public:
  // (Inherited from cv::ml::ANN_MLP)
  //
//enum TrainingMethods {
//  BACKPROP        = 0, // The back-propagation algorithm.
//  RPROP           = 1  // The RPROP algorithm.
//};
//
//enum ActivationFunctions {
//  IDENTITY        = 0,
//  SIGMOID_SYM     = 1,
//  GAUSSIAN        = 2
//};
//
//enum TrainFlags {
//  UPDATE_WEIGHTS  = 1,
//  NO_INPUT_SCALE  = 2,
//  NO_OUTPUT_SCALE = 4
//};
int getTrainMethod() const;          // Get backpropagation or RPROP method
int getActivationFunction() const;   // Get the activation function
                                     // (IDENTITY, SIGMOID_SYM or GAUSSIAN)
Mat getLayerSizes() const;                 // Get size of all the layers
TermCriteria getTermCriteria() const;      // Get max it / reproject error
double getBackpropWeightScale() const;     // Set Backprop parameter
double getBackpropMomentumScale() const; //  "
double getRpropDW0() const;                // Set Rprop parameter
double getRpropDWPlus() const;             //  "
double getRpropDWMinus() const;            //  "
```

```
double getRpropDWMin() const;              //  "
double getRpropDWMax() const;              //  "
void setTrainMethod(                     // Set backpropagation or RPROP method
  int    method,                         // either of: BACKPROP or RPROP
  double param1 = 0,
  double param2 = 0
);
void setActivationFunction(              // Set activation function
  int    type,                           // IDENTITY, SIGMOID_SYM, or GAUSSIAN
  double param1 = 0,
  double param2 = 0
);
void setLayerSizes( InputArray _layer_sizes ); // Set size of all the layers
void setTermCriteria( TermCriteria val );      // Set max it / reproject error
void setBackpropWeightScale( double val );     // Get Backprop parameter
void setBackpropMomentumScale( double val );   //  "
void setRpropDW0( double val );                // Set Rprop parameter
void setRpropDWPlus( double val );             //  "
void setRpropDWMinus( double val );            //  "
void setRpropDWMin( double val );              //  "
void setRpropDWMax( double val );              //  "
...
Mat getWeights( int layerIdx ) const; // Get the computed weights for
                                      //  the interlayer connections
...
// from class ANN_MLP
//
//static Ptr<ANN_MLP> create(); // The algorithm "constructor",
  //                          //  returns Ptr<ANN_MLPImpl>
};
```

为了使用多层感知机，首先需要使用cv::ml::ANN_MLP::create()方法构建一个空的网络，然后设定所有层的大小。之后，一旦设置完所有的参数（激活函数、训练方法和它们的参数），和之前一样，你可以调用cv::ml::StatModel::train()方法。

setLayerSizes()方法的参数指定了网络的基本结构。[注36]其结构应该是单列数组，用于表示输入数，第一层中的节点数，第二层节点数，等等，直到最后一行给出最终层中的节点数（从而给出了网络中的输出数量）。[注37]

setActivationFunction()方法指定了作用于输入的加权求和的激活函数。网络中任意节点的计算输出y_i由输入$\vec{x} \cdot \vec{w}$加权求和加偏移项θ的函数f()给出。权重和偏移项将在训练

注36：在反向传播和深度学习中，一个经典的“黑魔法”是在每一层中设置多少神经元。你可以在隐藏层中使用较少的神经元来实现数据压缩，或使用更多的神经元来过度描述数据，这样可以在拥有足够数据的情况下提高性能。

注37：通过此计数，图21-14中的网络将可以描述为一个拥有3，3，3个行的数组（第一个对应于三输入，第二个代表第一列，第三个表示最后一行）。请记住，输出的数量与最后一层中的节点数完全相等，因此在第三行之后不需要另外的3。

阶段学习获得，但函数本身是网络的属性。此时，你可以选择三种函数：线性函数、S形和高斯函数。[注38]setActivationFunction的值和相应的参数如表21-4所示。

表21-4：cv::ANN_MLP中用到的可选择人工神经元激活函数，个个都是$z = \vec{x} \cdot \vec{w} + \theta$类型的函数

activateFunc值	激活函数
cv::ANN_MLP::IDENTITY	$f(z) = z$
cv::ANN_MLP::SIGMOID_SYM	$f(z) = \beta \frac{(1 - e^{-\alpha z})}{(1 + e^{-\alpha z})}$
cv::ANN_MLP::GAUSSIAN	$f(z) = \beta e^{-\alpha z^2}$

最后两个参数fparam1和fparam2对应于表21-4中的变量α和β。除非你是专家，否则在几乎所有的情况下，请尽量使用默认的sigmoid激活功能，在大多数情况下，都需要将这些参数设置为其通用值（1.0）。

一旦构建好人工神经网络，就需要使用你拥有的数据对网络进行训练。为了训练网络，你需要像往常一样构造cv::ml::TrainData，然后将其传递给在cv::ml::ANN_MLP中的cv::ml::StatModel::train()函数。与来自ml的所有其他模型不同，神经网络不仅可以处理标量输出，还可以处理矢量输出，所以响应不一定是单列向量，它可以是一个矩阵，每个样本一行。矢量输出实际上可以用神经网络来实现一个多元分类器。



无论对于输入或输出，人工神经网络本身不会结构化地去进行数据分类。这种现象最常见的解决方案是用 “One-of-many”（也称为“one-hot”）编码，在编码中一个含有K个元素的类表示K个单独的输入（或输出。在该方案中，如果输入对象是k类的成员，则网络输入k将是非零的（通常为1），而其他所有的k都将为0。同样，我们可以使用该形式编码输出。这种编码的一个有趣的特点是，在实践中，即使输入的是单位向量，输出通常会有一个完美的结构（非本类的值会很小，并且小于所选类的整体值）。这有助于显示分类的质量。

此外，神经网络可以处理非空sampleWeights向量，传递给cv::ml::TrainData::create，它允许你为输入和输出中的每个数据样本（行）分配相应的权重。sampleWeights中的每一行都可以设置为任意的正浮点型数据，对应于同一行中的输入和输出。权重将自动归一化，所以问题的关键是它们的相对大小。sampleWeights参数只适用于Rprop算法，因此，如果你使用的是反向传播，它将被忽略 （请参阅之后的讨论）。

注38： 高斯函数的实现现在还不完全成熟。

用于训练的参数

setTermCriteria()方法指定训练何时终止，对使用反向传播或RProp（请参见之后的setTrainMethod()函数）具有相同的含义。终止条件中的迭代次数恰好是更新将执行的最大步骤数。终止条件的ε部分设置迭代继续所需的更新误差的最小更改，这个更新误差等于训练集结果和整个训练集的计算输出之间的平方差之和的0.5倍。[注39]

可以将train_method设置为cv::ANN_MLP::BACKPROP或cv::ANN_MLP::RPROP。默认情况下，此参数将设置为RPROP，因为在大多数情况下，此方法在训练网络时通常更有效。在RPROP的情况下，构造函数中的param1和param2对应于初始更新次数和最小更新的步长。在BACKPROP中，param1和param2分别对应于所谓的权重梯度和动量。或者， 你可以使用setBackpropWeightScale()和setBackpropMomentumScale()成员访问器函数来设置这些参数。在反向传播过程中权重梯度值乘以权重渐变项，实质上控制所需要移动方向的速度。这个变量的一个典型值是0.1。冲量因子乘以更新中的一个额外的变量，它与前一步的值和前一步之间的差成正比——给出像是权重的速度。冲量项预先乘以这种速度，它具有在更新中平滑移动的效果。如果设置为0，则有效地消除此变量。然而，一个很小的值（通常为0.1）通常会大大加快总体收敛速度。除了这两个参数外，还有访问器setRpropDWPlus()，setRpropDWMinus()、setRpropDW0()、setRpropDWMin()和setRpropDWMax()，可用于分别设置值η^+、η^-、Δ_0、Δ_{min}和Δ_{max}以及相应的get*()方法来获取这些值。

一旦训练了网络，就可以用通常的方式进行预测，即通过复用cv::ml::StatModel::predict()方法。与往常一样，样本必须是一个数组，每个输入数据点有一定的行和一定的列数 （与训练数据相同的结构）。输出数组将具有与输入相同的行数，但与网络的输出层中的节点数相等的列数。返回值是无意义的，可以忽略。[注40]

支持向量机

支持向量机（SVM）是一种分类算法，它的最基本形式是基于一系列的样本对两类进行区分。SVM的拓展算法可以用于实现多类（$N_c > 2$）分类。SVM的精髓在于核函数的使用，即将某个特定维度空间的数据映射到更高维度的空间中去，这个空间被称为核空间。[注41]在这个空间中，线性分类器常常可以找到两类的分界函数，即使在原始的低维空间不存在这样的分界函数。SVM也被为“最大化类间距分类器”，因为它选择的在核

注39： 如果你有样本权重，则误差项也由样本权重加权。

注40： 必须符合cv::ml::StatModel::predict()指定的接口。

注41： 通常也被称为特征空间，然而这个术语是很有迷惑性的。其他的作者时常会使用这个术语来表达输入的特征所构成的低维度空间。在这里我们将会避免使用这个术语。

空间的超平面不仅可以区分两个类别，还能够保证对于两类样本中所有样本点到超平面的空间距离之和最大。最接近超平面的那些样本，决定了超平面所处的位置，这些样本被称为“支持向量”。支持向量的意义在于一旦确定，就将它们保留下来预测将来得到的数据点的类别即可。

更形式化的，线性支持向量机的决策超平面可以被描述为方程$\vec{w} \cdot \vec{x} + b = 0$。注意，尽管决策平面在高维空间是线性的，其在原始的低维空间中可能是非线性的，正如图21-12所示。在这种情况下，向量$\vec{x}$是核空间中的点并且$\vec{w}$描绘了其在核空间中的法线。为了方便起见，$\vec{w}$会被归一化，使超平面到支持向量之间的距离可以表示为$1/\|\vec{w}\|$。值b是超平面的偏置（相对于过原点的平行超平面）。[注42]在确定这样一组参数之后，支持向量将处于$\vec{w}.\vec{x} + b = +1$或者$\vec{w}.\vec{x} + b = -1$的空间。

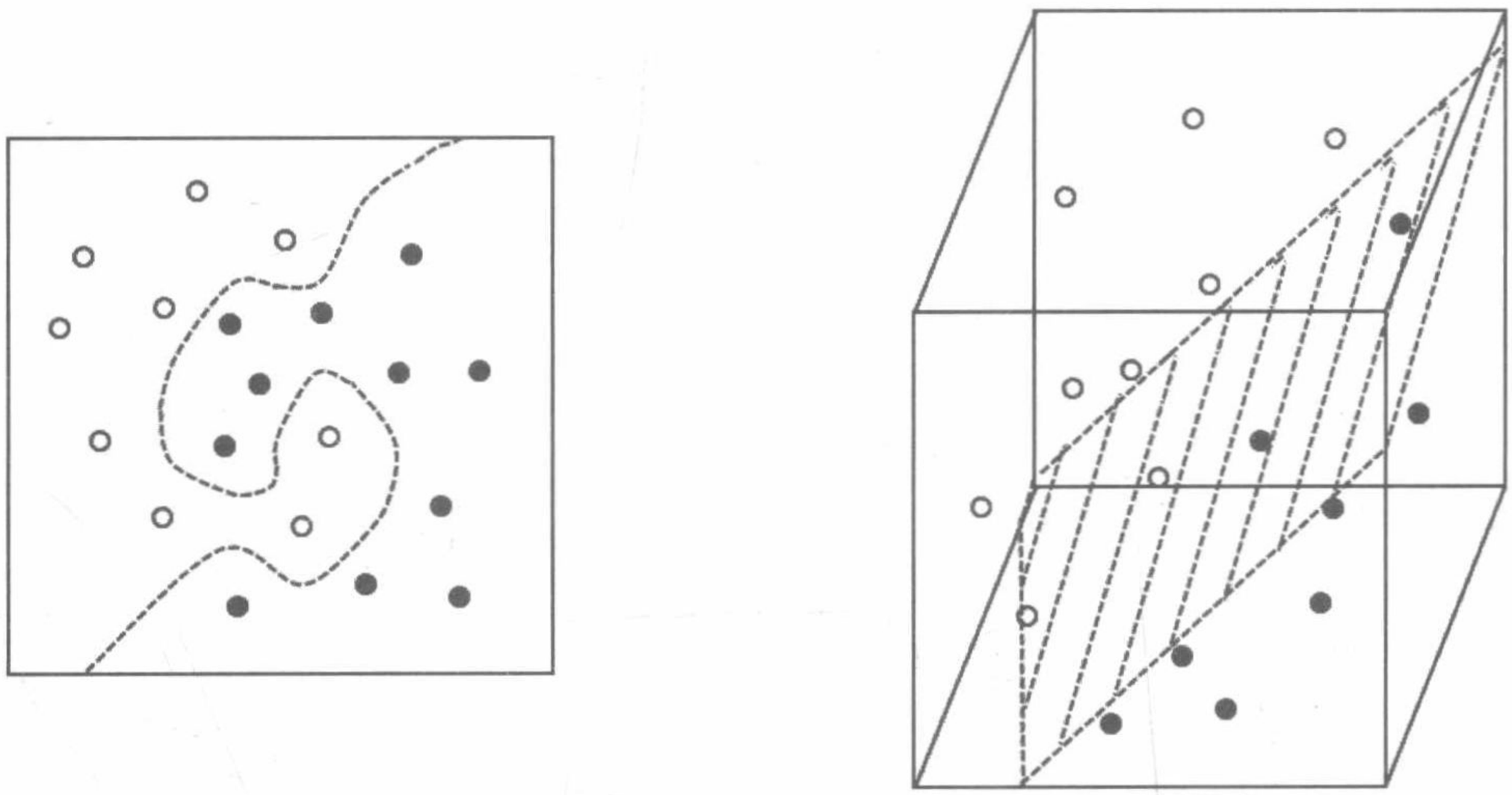


图21-12. 低维空间中的点（左图）在高维空间中被分离（右图）。尽管决策平面在高维空间是线性的（一个超平面），在低维空间中它也可能呈现出极其非线性的结构

尽管它的完整参数超出了本书范围，观察图21-13，也应该会得到这样一个合理的结论：向量$\vec{w}$完全可以被支持向量自身所表示——原因在于，从根本上来说，超平面是由训练集中的数据所决定的。这就好像是战场的“前线”只能由最前方的人所确定的——处于后方的人不管在哪里对它都没有什么影响。尽管不太直观，但事实上可以证明$\vec{w}$是可以被支持向量的线性组合所表示的。类似的，值b也可以通过支持向量计算出来。一旦拥有了这两组参数，我们就可以确定一个超平面。任何新的点$\vec{x}$都可以输入这个分类器并得到以下四种结果中的一种：很好地位于第一类中（比如说$\vec{w}.\vec{x} + b \geq +1$）勉强属于第一类$(\vec{w}.\vec{x} + b > 0)$，勉强属于第二类$(\vec{w}.\vec{x} + b < 0)$以及属于第二类$(\vec{w}.\vec{x} + b \geq +1)$。

注42：更准确地说，偏置是$b/\|\vec{w}\|$。

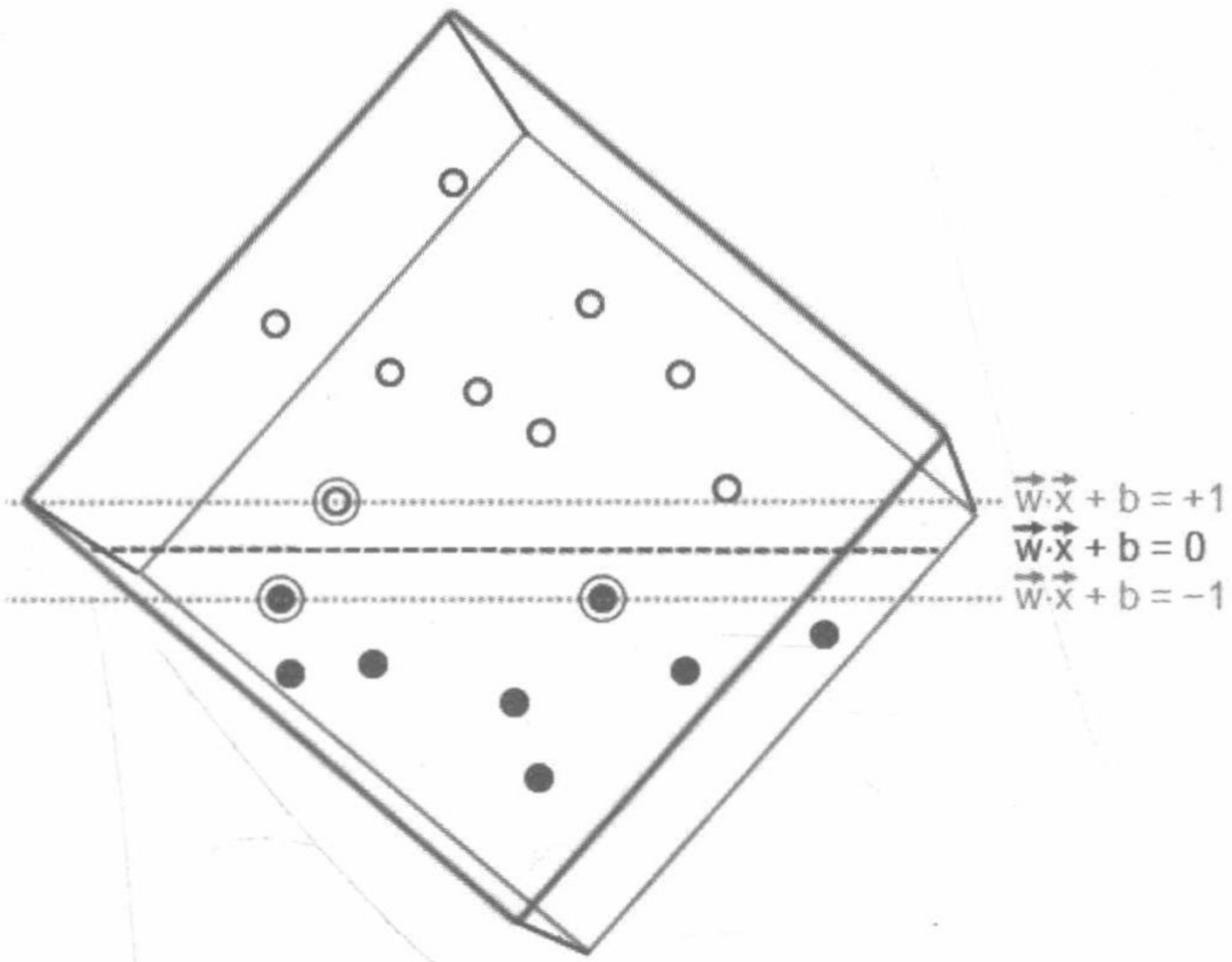


图21-13. 与图21-12相同的核空间，在此，我们旋转一下来看清楚决策平面的边缘。支持向量（带圈点）在超平面的两边等距分布

关于核

核函数是将处于朴素的空间中的输入数据的点映射到高维核空间的操作及其逆过程。记住，我们是在高维空间中寻找超平面。核函数由两部分组成，一部分将我们从输入的特征空间转换到核空间及其逆操作，将我们映射回来。通常，核函数记为 $K(\vec{x}, \vec{x}')$，映射分别记为$\vec{\varphi}(\vec{x})$和$\vec{\varphi}^T(\vec{x}')$。核函数从定义上来说，可以如下表示为：

$$K(\vec{x}, \vec{x}') = \vec{\varphi}^T(\vec{x})\vec{\varphi}(\vec{x}')$$

通过例子来了解核函数，这是最简单的方法。在OpenCV的SVM模块中有一个可用的核称为“多项式核”。这个核有如下形式：

$$K(\vec{x}, \vec{x}') = (\vec{x} \cdot \vec{x}' + c)^q$$

其中$c>0$，q为大于0的整数。这个核函数相对的映射 $\vec{\varphi}_{q,c}(\vec{x})$ 可能含有非常多的部分。甚至在最简单的情况$q = 2$，$c = 0$时，对于给定的三维输入$\vec{x} = (x_1, x_2, x_3)$：[注43]

$$\vec{\varphi}_{2,0}(x_1, x_2, x_3) = \left(x_3^2, x_2^2, x_1^2, \quad \sqrt{2}x_3x_2, \sqrt{2}x_3x_1, \sqrt{2}x_2x_1\right)$$

注43：不难证明 $\vec{\varphi}^T_{2,c}(\vec{x})\vec{\varphi}_{2,c}(\vec{x}') = \vec{\varphi}_{2,c}(\vec{x}) \cdot \vec{\varphi}_{2,c}(\vec{x}') = K(\vec{x}, \vec{x}') = (\vec{x} \cdot \vec{x}' + c)^2$。

为了对比，当$c = 1$时：

$$\vec{\varphi}_{2,1}(x_1, x_2, x_3) = \left(x_3^2, x_2^2, x_1^2, \quad \sqrt{2}x_3x_2, \sqrt{2}x_3x_1, \sqrt{2}x_2x_1, \sqrt{2}x_3, \sqrt{2}x_2, \sqrt{2}x_1, 1\right)$$

对于第一个选择的核来说，$N_d = 3$对应于一个六维空间$N_{KS} = 6$;，对于第二个核来说，相同的输入对应于十维空间$N_{KS} = 10$。

然而，对于这些核来说，映射$\vec{\varphi}(\vec{x})$在计算SVM的超平面的时候是不需要真正得到的。这是因为所有在核空间中的计算都是点积。这表示一旦我们找到$\vec{\varphi}(\vec{x})$，对某些点$\vec{x}$和$\vec{x}'$，它将会形如$\vec{\varphi}^T(\vec{x})\vec{\varphi}(\vec{x}')$。换一种说法，我们不需要分别计算$\vec{\varphi}(\vec{x})$，而只需要计算$K(\vec{x}, \vec{x}')$。这一点的应用有深刻的意义的。这表示即使核空间的维度非常高（也许是无穷的），我们也只需要对输入的特征向量计算$K(\vec{x}, \vec{x}')$来训练SVM。这个性质在SVM中被称为核技巧（kernel trick）。

按照惯例，使用者仅仅需要选择核及其形式，SVM就可以工作了。核的选择对于生成的超平面、对于训练数据的分割效果以及对于之后数据的适应性都有非常大的影响。OpenCV现在提供了六种核供你在支持向量机中使用，如表21-5所示。

表21-5：OpenCV的SVM中可用的核

核	OpenCV命名(kernel_type)	核函数	参数
线性	cv::ml::SVM::LINEAR	$K(\vec{x}, \vec{x}') = \vec{x}^T\vec{x}'$	
多项式	cv::ml::SVM::POLY	$K(\vec{x}, \vec{x}') = \left(\gamma \vec{x}^T\vec{x}' + c_0\right)^q$	degree (q), gamma ($\gamma > 0$), coef0 (c_0)
径向基函数	cv::ml::SVM::RBF	$K(\vec{x}, \vec{x}') = e^{-\gamma\left\|\vec{x}^T\vec{x}'\right\|^2}$	gamma ($\gamma > 0$)
Sigmoid	cv::ml::SVM::SIGMOID	$K(\vec{x}, \vec{x}') = \tanh(\gamma\vec{x}^T\vec{x}' + c_0)$	gamma ($\gamma > 0$), coef0 (c_0)
指数卡方分布	cv::ml::SVM::CHI2	$K(\vec{x}, \vec{x}') = e^{-\gamma\frac{\vec{x}-\vec{x}'}{\vec{x}-\vec{x}'}}$	gamma ($\gamma > 0$)
直方图交运算	cv::ml::SVM::INTER	$K(\vec{x}, \vec{x}') = \min(\vec{x}, \vec{x}')$	

处理离群点

OpenCV支持两种SVM的变体，分别为C-向量SVM以及v-向量SVM。[注44]这些拓展假设了离群点的存在。在这一段，离群点的意思是不能被被正确分类的数据点（或者说在核

注44：在希腊字母中，仍然是字母“nu”，但是发音类似于英文中的“new”，这个双关语也许是有意为之。

空间不能用超平面分割的数据集）。补充一点，OpenCV所实现的这两种方法都允许处理多分类（$N_c > 2$）问题。

C-向量SVM，也称作“软间隔SVM”（soft margin SVM）。允许通过对离群点的距离计算增加一定比例的处罚量来允许离群点的存在。这种距离通常称为“松弛变量”，而固定的比例通常称为“*C*”（这也是该方法名称的来源）。

第二种拓展方法，同样允许离群点的存在，称为“v-向量SVM”[Schölkopf00]。在本例中，相对于*C*-向量SVM的固定的（预先决定好的）比例常量，本方法引入了一个新的变量*v*。这个变量设定训练错误的上限（比如，错误的分类），同样可以作为支持向量占整个训练数据集的下限。参数*v*必须设定在0到1之间。

比较这两种方法，*C*-向量SVM的实现更加直观，并且训练的时候更快。然而，由于C没有对应的物理意义，所以很难找到一个合适的值，只有通过试错的方法不断寻找。另一方面，*v*-向量SVM的参数有着对应的物理意义，并且一般更好用。

SVM的多分类拓展

C-向量支持向量机和*v*-向量支持向量机在OpenCV中的实现都支持多于两类的分类任务。尽管有很多方法都可以做到这一点，OpenCV使用一种称为“一对一”（one-versus-one）的方法。[注45]这个方法中，如果存在*K*个类别，则会有$k(k-1)/2$个分类器被训练。当分类一个目标时，所有分类器都开始运行，而获得多数胜利的类将会认为是分类结果。

一分类SVM

另一个有趣的SVM变体是一分类SVM（one-class SVM）。在这种情况下，所有的训练数据都会被认为属于同一个类别，决策边界会尝试将这个类和其他所有不属于这个类的数据区分开。和之前所描述的多类别拓展一样，一分类SVM也支持通过一个松弛变量来容忍离群点的存在，就像*C*-向量SVM所做的那样（拥有相同的参数）。

支持向量回归

作为一分类和多类的补充，支持向量机还有支持回归（和分类相对）的拓展版本。在回归的情况下，输入的结果是顺序值，而不是类标签。输出是根据输入值而进行的插值（或外推）。OpenCV提供两种不同的算法来完成支持向量回归（SVR），分别是ε-SVR [Drucker97]以及*v*-SVR [Schölkopf00]。

从最抽象的形式来说，SVR和SVM在本质上来说是一致的，在让误差方程最小化的过

注45：“其他的”选择是“一对多”（one-versus-many），在OpenCV中还没有实现。

程中，SVM总是输出两个值中的一个值（一般是+1或者-1），而SVR对于每个不同的输入都有不同的输出。它们的细微差别在于对松弛变量的处理以及什么样子的点会被认为是离群点。前面讲过，在SVM中，所有点都会被最大间隔超平面分配到不同的类中。因此，离群点的定义是关于点所在的位置、被赋予的标签以及相应的超平面的。在SVR中，超平面是函数的模型。（前面讲过，在这个在核空间中的超平面可能会在特征空间中有极其复杂的形式）所以，它的离群点是距离超平面超过一定距离的点（不管它处于哪个方向）。我们在OpenCV中提供的两种SVR算法的区别就在于如何处理这段距离。

第一种算法ε-SVR，使用一个参数ε来定义围绕在超平面周围的空间，在这个空间内的预测没有或者具有较小的损失。一旦超过这个距离，就会附加一个随着距离线性增长的损失。这就是数据点都在超平面附近的原因。

第二种算法*v*-SVR和ε-SVR的关系类似于*v*-SVM和*C*-SVM。*v*-SVR使用参数*v*来设定作为支持向量的数据的最小百分比。通过这个行为，*v*-SVR中*v*参数的意义类似于*v*-SVM。从本质上来说，更加直观，并且更易于设置一个合理的参数。

SVM和cv::ml::SVM()的使用

SVM分类器在OpenCV的接口定义在ML库的类cv::ml::SVM中，类似于本章我们学习过的其他对象，它也继承自类cv::ml::StatModel，并且它本身也作为其实现cv::ml::SVMImpl的接口存在。

```
// Somewhere above..
// namespace cv {
//   namespace ml {
//
class SVMImpl: public SVM {   // cv::ml::SVM is derived
                              // from cv::ml::StatModel
public:

  // (Inherited from cv::ml::SVM)
  //
  //enum Types {
  //  C_SVC     = 100,
  //  NU_SVC    = 101,
  //  ONE_CLASS = 102,
  //  EPS_SVR   = 103,
  //  NU_SVR    = 104
  //};
  //
  //enum KernelTypes {
  //  CUSTOM    = -1,
  //  LINEAR    = 0,
  //  POLY      = 1,
  //  RBF       = 2,
  //  SIGMOID   = 3,
```

```
//  CHI2        = 4,
//  INTER       = 5
//};
//
//enum ParamTypes {
//  C           = 0,
//  GAMMA       = 1,
//  P           = 2,
//  NU          = 3,
//  COEF        = 4,
//  DEGREE      = 5
//};

class Kernel : public Algorithm {

public:

  int getType() const;
  void calc(
    int           vcount,
    int           n,
    const float* vecs,
    const float* another,
    float*        results
  );
};

int getType() const;                              // get the SVM type (C-SVM, etc.)
double getGamma() const;                          // get the gamma param of kernel
double getCoef0() const;                          // get the coeff0 param of kernel
double getDegree() const;                         // get degree param of kernel
double getC() const;                              // get C param of C-SVM or eps-SVR
double getNu() const;                             // get nu param of nu-SVM or nu-SVR
double getP() const;                              // get P param of eps-SVR
cv::Mat getClassWeights() const;                  // get the class priors
cv::TermCriteria getTermCriteria() const;         // get training term criteria
int getKernelType() const;                        // get the kernel type

void setType( int val );                          // set the SVM type (C-SVM, etc.)
void setGamma( double val );                      // set the gamma param of kernel
void setCoef0( double val );                      // set the coeff0 param of kernel
void setDegree( double val );                     // set degree param of kernel
void setC( double val );                          // set C param of C-SVM or eps-SVR
void setNu( double val );                         // set nu param of nu-SVM or nu-SVR
void setP( double val );                          // set P param of eps-SVR
void setClassWeights( const Mat &val );           // set the class priors
void setTermCriteria( const TermCriteria &val ); // set training term criteria
void setKernel( int kernelType );                 // set the kernel type

...

// set the custom SVM kernel if needed
//
void setCustomKernel( const Ptr<Kernel> &_kernel );
```

```
Mat      getSupportVectors() const;
Mat      getUncompressedSupportVectors() const;
double   getDecisionFunction(
  int         i,
  OutputArray alpha,
  OutputArray svidx
);

ParamGrid getDefaultGrid( int param_id ); // return default grid for any param
                                          // Choose from SVM::ParamTypes above
bool trainAuto(
  const Ptr<TrainData>& data,
  int                   kFold      = 10,
  ParamGrid             Cgrid      = getDefaultGrid( C ),
  ParamGrid             gammaGrid  = getDefaultGrid( GAMMA ),
  ParamGrid             pGrid      = getDefaultGrid( P ),
  ParamGrid             nuGrid     = getDefaultGrid( NU ),
  ParamGrid             coeffGrid  = getDefaultGrid( COEF ),
  ParamGrid             degreeGrid = getDefaultGrid( DEGREE ),
  bool                  balanced   = false
);

...

// from class SVM
//
//static Ptr<SVM> create(); // The algorithm "constructor",
//                          //  returns Ptr<SVMImpl>
};
```

与之前类似，你首先需要使用ceate()创建一个类的实例。之后使用setters配置参数，然后执行训练方法。第一个方法setType()确定SVM还是SVR会被使用。它可以是表21-6所示五种参数中的任意一种。下一个重要的方法是setKernelType()，其参数可以是表21-5的任意一种。

表21-6：SVM可用的参数以及setType()对应的值。最右边的列列出了可以由SVM类型设置的参数，即cv::ml::SVM的相关属性及其对应的值（如上一节所述）

SVM类型	OpenCV中的命名	参数	
C-SVM分类器	cv::ml::SVM::C_SVC	C (C)	
v-SVM分类器	cv::ml::SVM::NU_SVC	Nu (v)	
一分类SVM	cv::ml::SVM::ONE_CLASS	C (C)	Nu (v)
ε-SVR	cv::ml::SVM:: EPS_SVR	P (ε)	C (C)
v-SVR	cv::ml::SVM::NU_SVR	Nu (v)	C (C)

一旦选择SVM的类型和你将要使用的核的类型，就应当通过对应的set*()方法设置核的参数。表21-5和表21-6中列举的参数展示了在不同情况下需要设置的参数。默认情况

下，参数degree，gamma，coef0，C，Nu和P的值分别是0.0，1.0，0.0，1.0，0.0以及0.0。

方法setClassWeights()允许用户通过提供一个单独的列向量来指定松弛参数的重要性。它只会在*C-SVM*中使用。你所提供的参数会在每个独立的训练向量中和C相乘。这样，你可以对训练集中的某些子集赋予更大的权重，并以此来帮助保证，如果某一个训练向量不能够正确分类，就不会是其中一员。默认情况下，类权重将不会被使用。

最后一个方法是setTermCriteria()，使用它的默认参数1000次迭代和FLT_EPSILON几乎始终是安全的。

在这里，你可能会查询SVM中所有可调的参数和可选择的核，并且想知道如何从中选择一个正确的值。你并不是唯一一个提出这个问题的人。事实上，通常的惯例是按一定步长简单尝试所有参数，然后选择对你的可用数据效果最好的参数。这个处理可以通过cv::ml::SVM::train Auto()自动实现。

当你使用cv::ml::SVM::trainAutio()的时候，你所提供的训练数据和你需要在cv::ml::StatModel::train()中所提供的一样。KFold参数控制了每个参数集的校验是如何进行的。这个方法使用了*k*-交叉检验（k-fold cross-validation）（所以参数在这里应该被称为k），这个方法将参数划分为*k*个子集，并运行*k*次，每次运行，都会选择一个不同的子集保留原有的参数，而其他$k-1$个子集会使用新选择的参数。

接下来的六个参数控制网格（*grid*），它表示待测试参数的集合数。一个网格就是一个拥有三个成员的对象：minVal，maxVal以及logStep。如果你设置step为小于1的参数，则网格将不会使用，并且来自params的参数将会用于所有的运行过程。通常你不会对每个参数都进行搜索，你可以使用如下构造函数构造一个网格：

```
cv::ml::ParamGrid::ParamGrid() {
  minVal  = maxVal = 0;
  logStep = 1;
}
cv:: ml::ParamGrid::ParamGrid(
  double minVal,
  double maxVal,
  double logStep
);
```

也可以使用cv::ml::SVM的方法cv::ml::SVM::getDefaultGrid(int)。如果这么做，你需要告诉getDefaultGrid()你希望哪些参数使用网格搜索。这将成为每一个参数的默认网格。你传递给getDefaultGrid的值可以是cv::ml::SVM::C，cv::ml::SVM::GAMMA，cv::ml::SVM::P，cv::ml::SVM::NU，cv::ml::SVM::COEF或者cv::ml::SVM::DEGREE。如果是想要自己创建一个网格，当然也可以那样做。记住，参数logStep是作为乘法系数

解释的，所以如果你对它设置（比如）为2.0，所有尝试的值都会是原来值的两倍，直到达到maxVal（或者溢出）。

最后，如果你已经训练好了一个分类器，则可以使用cv::ml::StatModel::predict()对数据进行预测：

```
float cv::ml::SVM::predict(
  InputArray  samples,
  OutputArray results = noArray(),
  int         flags   = 0
) const;
```

和惯例一样，predict()函数需要输入一个或者多个样本，每一个样本占据一行，并且返回预测结果。在处理一分类或两类分类问题的时候，可以检索计算得到的值，而不是类的标签。为了完成这个功能，需要输入cv::ml::StatModel::RAW_OUTPUT标志。返回的带符号的数值将与样本和决策平面的距离相关。这就是你希望在二分类问题中所得到的的结果。否则，返回类标签（在多类的情况下）。在回归的情况下，返回值将会是函数的预测值，并忽略这个标记。

cv::ml::SVM的附加成员

cv::ml::SVM对象提供了一些工具接口，允许你得到对象中的一些数据。这包括训练阶段输入的数据，也包括计算得到的支持向量、默认的网格等有用的变量。可用的函数如下所示：

```
// get all the support vectors
//
cv::Mat cv::ml::SVM::getSupportVectors()  const;

// get uncompressed support vectors as found by the training procedure
//
cv::Mat cv::ml::SVM::getUncompressedSupportVectors() const;

// get the i-th decision function (out of n*(n-1)/2 in the case of n-class problem
//
double cv::ml::SVM::getDecisionFunction(
  int             i,
  cv::OutputArray alpha,
  cv::OutputArray svidx
) const;
```

方法cv::ml::SVM::getSupportVectors()返回所有用于决定超平面的支持向量。在线性SVM的情况下，每个决策平面的所有支持向量都可以由描述超平面的向量来表示。这也是线性SVM的预测非常快速的原因。然而，用户可能会对原始的支持向量感兴趣，这可以通过cv::ml::SVM::getUncompressedSup portVectors()方法获取。在非线性SVM的情况下，非压缩的支持向量和压缩过的是同一个事物。我们可以通过

cv::ml::SVM::getDecision Function获取每一个决策平面或超表面。在回归或者一分类SVM、二分类的情况下，只有一个决策函数，因此有i=0。在*N*分类的情况下，将会有N*(N-1)/2个决策平面，因此0<=i<N*(N-1)/2。这个方法返回特定方程中支持向量的系数，支持向量的索引（位于cv::ml::SVM::getSupportVectors()得到的矩阵中）以及在决定之前将其加入加权和中的b的值（参考前面的公式）。

作为训练和预测的补充，cv::ml::SVM提供了常用的save()，load()以及clear()函数。

小结

在本章中，我们从cv::ml::StatModel开始学习，这个模块是OpenCV所提供的用于封装所有模型学习方法的标准基于对象的接口。我们了解了训练数据如何处理以及在训练结束之后模型可以得到多精确的预测结果。在这里，我们了解了许多种学习技术并且利用接口将其实现了。

沿着这条路线，我们学习了cv::ml::TrainingData以及从cv::ml::StatModel派生出来的各种独立的分类器，它们的接口都在同一个类中定义。派生的实体类通过不同方法具体实现接口所提供的功能。这些实体类对大部分用户都是不可见的，因为分类器X的create()成员函数始终会返回一个具体实体类XImpl的实例。因为这个原因，当我们研究重要的分类器的类定义时，通常会发现一个*Impl对象才是我们真正想要的。

我们学习了朴素贝叶斯分类器，它假设所有特征都是独立的，对多标签区分学习它也惊人地有效。我们学习了二值决策树，它使用杂质矩阵（impurity metric）作为目标，通过最小化目标来建立决策树。二值决策树对多标签学习也非常有效，并且它还有一个非常有用的特性：不同的权重可以分配给不同类的错误分类。boosting和随机树在内部也使用了二值决策树，指示建立了更大的、包含许多这样的子树的结构。它们两者都可以用于多标签分类问题，但是随机树还引入了相对于类输入数据之间的相似性概念。

期望最大化是一种和*K*-means相似的无监督学习技术。但是不同点在于对于不同数据点赋予不同的权值对于最邻近聚类中心来说是没有必要的。*K*-means算法是一种非常有效的分类器，但它需要存储整个数据集。因此，它会使用大量的内存而且运行会很慢。多层感知机只一种受生物学启发而发展的技术，其可用于分类和回归问题。它的训练过程十分缓慢，但是在实际使用的时候非常快，并在许多重要的任务中达到了一个时下最先进的性能表现。最后，我们介绍了支持向量机（SVM），一种鲁棒的通常用于二分类问题的技术，但它对多分类问题来说有较大的方差。SVM通过在核空间中度量数据，在这个空间中两个类别的分割函数可以轻松表达出来。因为它只需要一个很小的子集作为训练集，它可以非常快并且只需要少量的内存。最后，OpenCV已经拓展以支持深度学习。参阅附录B的*cnn_3dobj*和*dnn*，了解更多详情。

练习

1. 图21-14描绘了“真”和“假”类别的分布。这张图还显示了一些可能设置阈值的地方（a，b，c，d，e，f，g）。

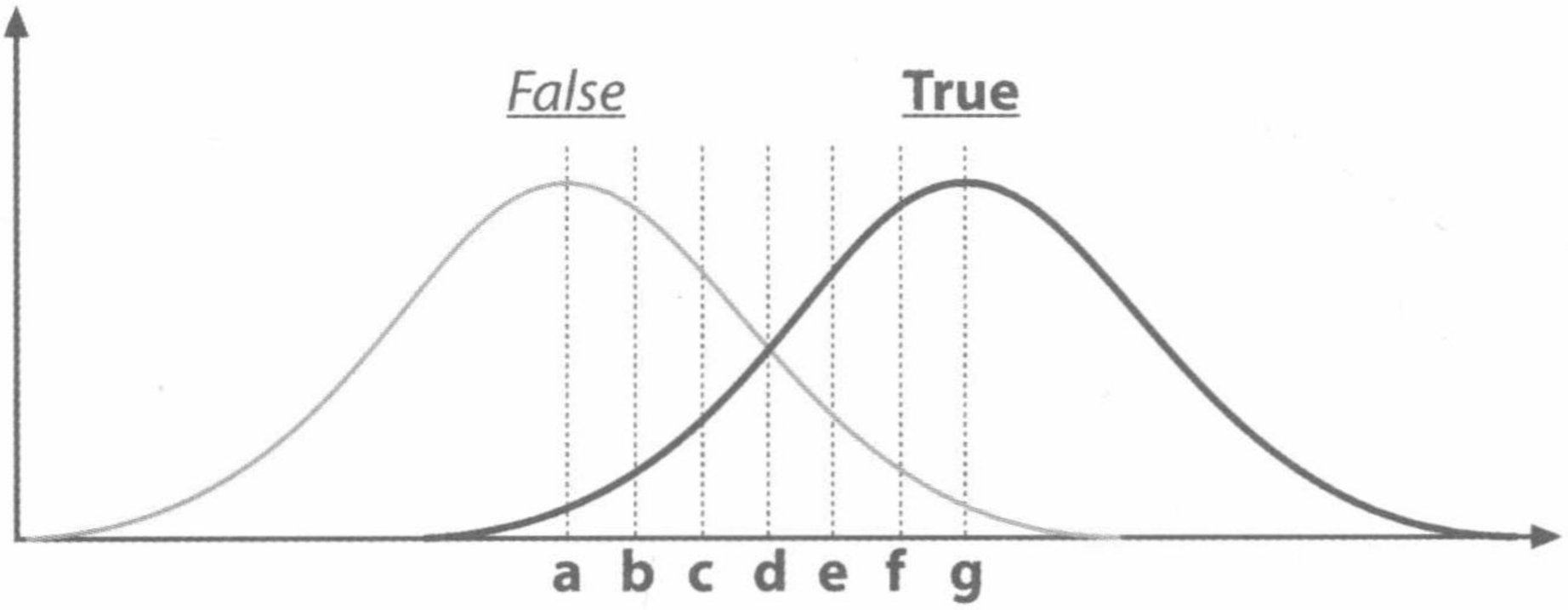


图21-14. 两类高斯分布，“真”与“假”

a. 将点a～g标注在ROC曲线上。

b. 如果“真”类别是毒蘑菇，你希望将阈值设置在哪个点？

c. 如何运用决策树来区分这个数据？

2. 回顾图20-2。

   a. 画出决策树，展示如何使用三个分支逼近真实曲线（虚线）（我们想要的是回归，而非分类模型）。



对于回归来说“最好的”分割是将数据的均值包含在输出结果中。回归树的输出值会呈现出一个阶梯状分布。

   b. 画出一个决策树，展示如何通过七个分支拟合真实数据。

   c. 画出一个决策树，展示如何通过七个分支拟合带噪数据。

   d. 从过拟合的角度讨论b和c的差异。

3. 当我们想要从一个单独的决策树学习多个类别的时候，基于分割的方法（比如说Gini）仍然有效，为什么？请说明原因。

4. 回顾图20-6，描绘了一个左边方差不相等，但右边方差相等的二维空间。假设这些是与分类问题相关的特征值，也就是说，在一个点附近的数据是属于其中一个类别

的，而另一个点附近的数据属于相同或者另外的类别。变量在左边和右边空间的重要性在下列哪些情况下是有区别的：

a. 决策树？

b. *K*-means？

c. 朴素贝叶斯？

5. 修改示例20-1程序中的数据生成部分，使峰值附近的环形区域在*K*-means的范围内，以此生成随机的数据标签。我们将使用一个10 000点的单峰值的高斯分布，它将描绘在一张128 × 128的图像上，其中心点在（63 × 63），方差为（`img.cols/6, img.rows/6`）。为了给数据加上标签，我们以（63，63）为中心将空间分割为四个象限。为了获取不同标签的概率，我们使用了下述方案：如果$x < 64$它将有20%的几率属于*A*类，如果$x > 64$它将有90%几率属于*A*类。如果$y < 64$它将有40%的概率属于*A*类，如果$y > 64$它将有60%的几率属于*A*类。通过将*x*和*y*所决定的区域概率相乘，可以得到如下所示的2 × 2矩阵，其表示类*A*在四个象限中出现的概率。如果一个点的标签不是*A*，它肯定是属于*B*类的。比如说，如果$x < 64$，$y < 64$，我们将有8%的几率得到类*A*，92%的几率得到类*B*。*A*类处于四个象限的概率分别为：

$0.2 \times 0.6 = 0.12$	$0.9 \times 0.6 = 0.54$
$0.2 \times 0.4 = 0.08$	$0.9 \times 0.4 = 0.36$

使用这些象限的概率来标记数据点。对每个数据点，先确定它所处的象限，然后生成一个0-1之间的随机数。如果这个数字小于或者等于它在这个象限的概率，就将其标记为*A*类。否则，将其标记为*B*类。然后我们会有一个带标签的数据列表，每个数据的*x*和*y*坐标将作为数据特征。读者会发现*x*坐标会比*y*坐标提供更多信息（为了判断属于哪一类）。对这个数据集训练随机森林并且计算重要性，证明*x*比*y*重要。

6. 使用练习5中相同的数据，并利用离散的AdaBoost来学习两个模型：一个拥有20棵子树，一个拥有500棵子树。从10 000个数据点中随机分割训练集和测试机。测试算法在训练集样本如下条件时的效果：

a. 150个训练样本

b. 500个训练样本

c. 1200个训练样本

d. 5000个训练样本

解释结果并说明发生了什么？

7. 重复练习5，但是分别使用50和500个随机树分类器进行训练。

8. 重复练习5，但是这次使用60棵树并且比较随机树和SVM的性能。

9. 如何处理才能使随机树在过拟合问题上性能优于决策树。

10. “没有免费的午餐”原则说明了没有哪个分类器对所有数据分布都是理想的。

    a. 描述一个数据分布，本章所介绍的任何一种分类器都无法对它进行分类。

    b. 什么样的分布对于朴素贝叶斯来说很难学习。

    c. 什么样的分布对于决策树来说很难学习。

    d. 应当如何对*b*和*c*中的数据进行预处理使其对应部分的分类器更容易学习分布。

11. 使用蘑菇数据集*agaricus-lepiota.data*，但是只读取数据，并舍弃所有的标签。之后复制这个数据，但随机对每一列进行替换。记原数据集为*A*类，打乱后的数据集为*B*类。将数据分为一个较大的训练集与一个较小的测试集。在训练集上训练随机树分类器。

    a. 随机树对区分*A*和*B*类这个问题在测试集上能够有多好的表现？

    b. 说明哪个变量的重要性更高。

    c. 用没有改变的蘑菇数据集训练一个网络，并说明重要的变量是否还和*b*一样。这是Leo Brieman发明的一种处理无监督数据的方法。

12. 反向传导使用链式法则来计算如何改变神经网络中的权值来降低输出的误差函数。运用图21-11来讨论方向传播和链式法则，并推导反向传播中的权值更新公式。

13. 非线性方程（图21-11中的f方程和g方程）通常是“sigmoid”或S形函数，这个函数在输入为较大负数的时候接近于0，在输入为较大正数的时候接近于1。一个典型的方程是$\sigma(x) = \frac{1}{1 - e^{-x}}$，证明微分$\sigma'(x) = \frac{d\sigma(x)}{dx}$满足$\sigma'(x) = \sigma(x)(1 - \sigma(x))$。

第22章

目标检测

在前两章中，我们学习了机器学习的基础知识，并深入研究了OpenCV库提供的用于判别式和生成式学习的大量技术。现在是时候将这些整合在一起了，在书中我们已经学习的计算机视觉技术与机器学习技术结合，并将它们应用于解决计算机视觉中的实际问题。这些问题中最重要的是目标检测——确定图像中是否包含特定目标的过程，同时确定在像素空间中该目标可能的位置。在这一章中，我们将学习能实现这些目标的几个方法，在每一个案例中都使用了前几章介绍的初级机器学习方法。

基于树的目标检测技术

在学习完OpenCV中的很多初级机器学习方法之后，现在我们转向一些更高级的函数，使用不同的机器学习方法来检测图像中感兴趣的目标。现在，OpenCV中这样的检测器有两个。第一个是级联分类器，它将Viola and Jones [Viola01]在人脸识别那个非常成功的算法进行了一般化。第二个是软级联，该算法进一步进行演化，使用新的方法给出了一个在大多数情况下比级联分类器更鲁棒的分类。这两种算法都非常成功地应用于人脸以外的其他许多目标检测。通常，具有刚性结构和强纹理的目标使用这两种方法会得到较好的检测效果。

这两种方法不仅封装了它们所使用的机器学习组件，而且还涉及实际调节输入以学习或后处理学习算法的输出的其他阶段。当然，这些目标检测算法不像核心机器学习算法那样拥有统一的接口。这个情况一方面是由于这些高级方法的需求和结果的自然变量越来越大，另一方面是由于在实际情况中这些算法通常由其创造者贡献给OpenCV，接口更类似于它们的原始实现。

级联分类器

我们要学习的第一种分类器是基于树的技术，叫级联分类器。它基于boosted拒绝级联（boosted rejection cascade）这一重要概念构建而成。它与OpenCV中ML库的大部分格式都不同，因为它一开始是作为完整的人脸识别应用开发的，之后的更新（在一定程度上）比原始的实现更具一般性。在这一节中，我们将详细介绍它，同时说明它能怎样训练以识别人脸和其他刚体。



计算机视觉是一个广泛并且变化迅速的领域，所以OpenCV实现特定技术而非算法组件更有可能遭遇过时的风险。原来的人脸识别（后来称为Haar分类器）已经成为OpenCV中的一部分很多年，现在就刚好处于这种过时的“风险”中。但是，人脸识别的需求如此普遍，以至于确实值得使用一种能够良好运行的基准技术。由于该技术基于众所周知并且常用的统计boosting领域，所以它实际上有更多通用的功能。从那时开始，一些公司已经在OpenCV中设计了“人脸”检测器，通过针对目标的每个视角在数千个选定的训练图像上训练新检测器来检测“大多数刚性”的目标（人脸、汽车、自行车和人体）。这种技术已经被用来创建最先进的检测器，尽管对目标的每一个视角或姿态都要训练不同的检测器。这种分类器是一种非常有价值的工具，在进行这类辨识任务时需要牢记。在OpenCV当前的形式下，这种一般性更为明显，而且随着未来的进步，将实现更多的可扩展性。

OpenCV的级联分类器实现了最先由Paul Viola和Michael Jones开发的人脸识别技术，它通常称为“Viola-Jones检测器”[Viola01]。最开始，这个技术与其OpenCV实现只仅支持一个特定序列的特征——Haar小波。[注1]后来由Rainer Lienhart and Jochen Maydt [Lienhart02]进行扩展，使用了对角线特征（这个判别器还有很多有待探索）之后被并入OpenCV实现。这个拓展的特征列通常称作“类Haar”特征。在OpenCV 3.x中，级联进一步扩展到局部二值模式或称LBP [Abonen04]。

正如实现的那样，OpenCV版本的Viola-Jones检测器分为两层。第一层是特征检测器，封装并模块化特征计算。第二层是实际的boosting级联，使用计算得出特征所在矩形区域的和与差，但并不知道如何计算特征。

类Haar特征

分类器中默认的类Haar特征如图22-1所示。在所有尺度上，这些特征组成的“原材料”将被用于boosted分类器。它们都含有能从积分图像（参考第12章）快速计算出的特征，积分图像从原始灰度图像中计算得到。

注1： Haar（哈尔）小波是第一个已知的小波基，最初由Alfred Haar在1909年提出。

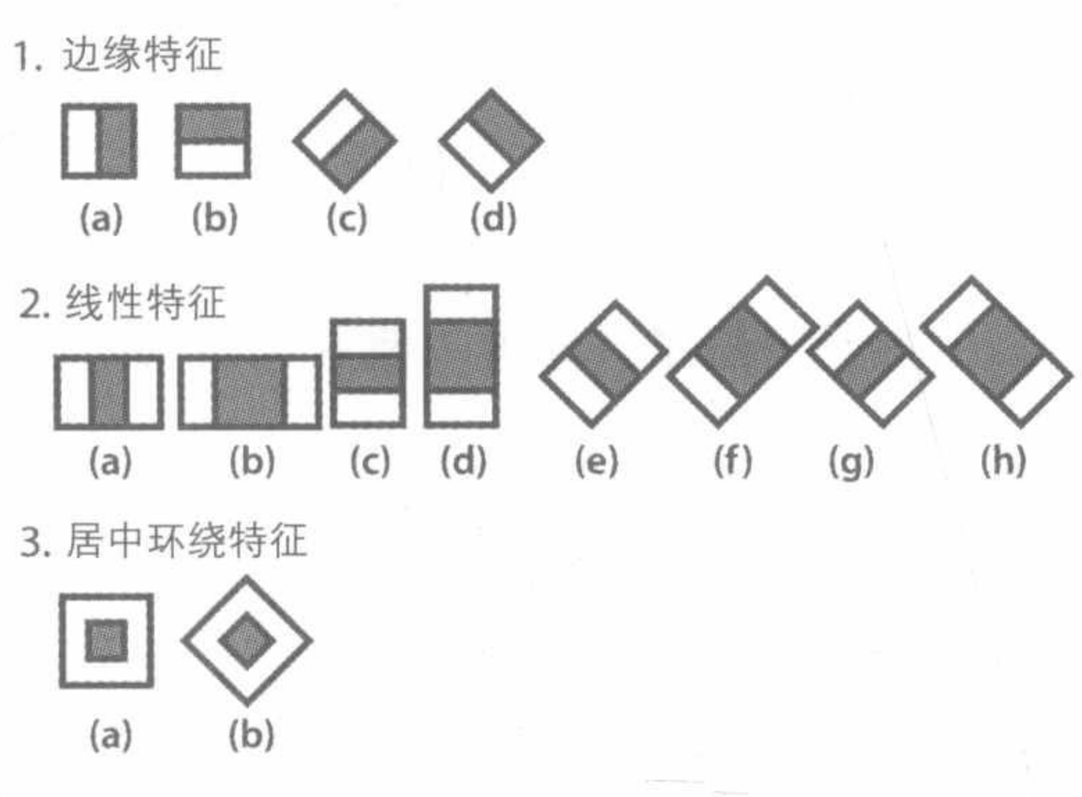


图22-1. 来自OpenCV源分布的类Haar特征（矩形和旋转区域容易通过积分图像计算得到）。在这个小波的图解表示中，浅色区域被称为“添加该区域”，深色区域为“减去该区域”

目前，支持两个不同的特征集。它们包括“原始”Haar小波特征（包括“对角线”特征）和替代特征LBP。未来，还可以支持其他特征类型，或者可以依据LBP的添加方式来编写或使用其他特征（因为现在级联特征接口是完全可扩展的）。

局部二值模式特征

LBP特征最开始由[Ojala94]提出，旨在作为一种纹理描述符。之后才适用于Viola-Jones目标检测算法的boosted级联环境。回想一下，Haar小波特征与 一个小补丁（例如11 × 11的像素）相关联，并将该补丁的“特征向量”分配为该补丁中像素的小波变换（在Haar基础上进行投影）。如图22-2所示，LBP特征在构建特征向量时有非常不同的方式。它采用一个宽度和高度都能被3整除的矩形，然后分割成3 × 3的不重叠像素块。对于每一个像素块，计算像素之和（使用积分图像）。最后，将八个非中心像素块中的每个像素总和与中心像素块的像素总和比较，从而构造出8位模式。这个8位模式被用作矩形的描述符。然后将这8位值用作传递给分类器的分类值（只要在训练期间选择相应的矩形作为足够辨识的特征）。

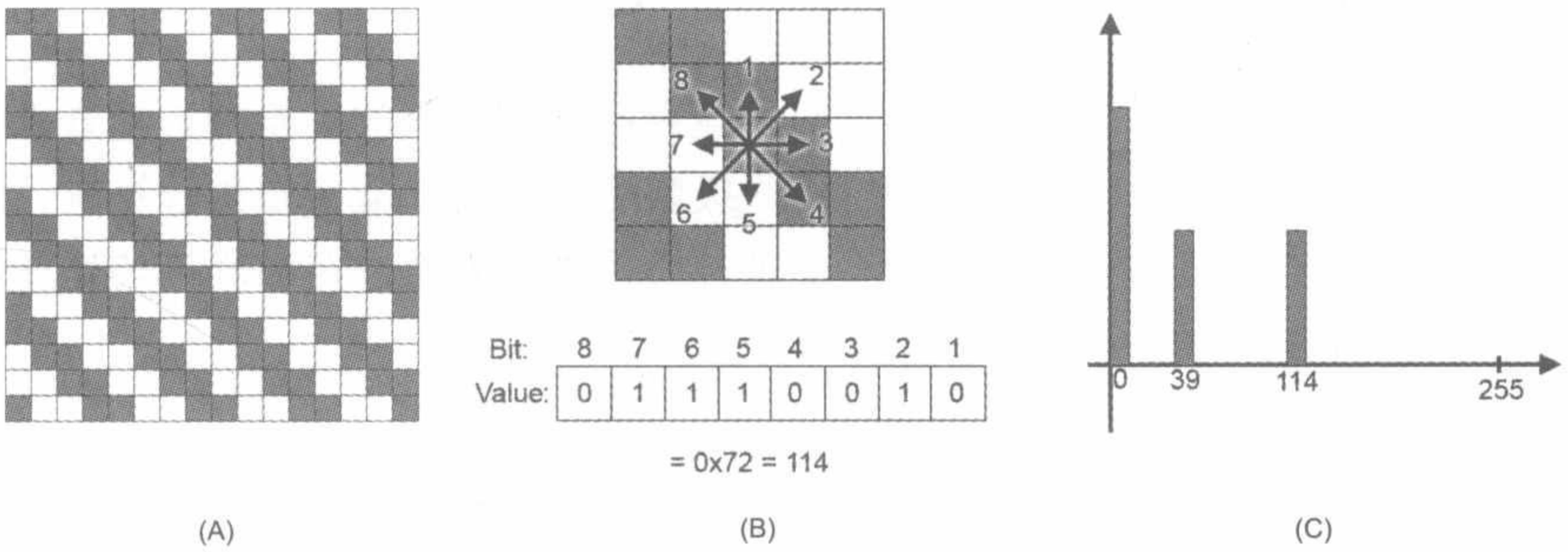


图22-2. 为纹理(A)的示例区域计算LBP特征。对于该区域的每一个像素，我们通过比较该像素与其邻近像素计算二值表达形式(B)。LBP特征是这些计算值的直方图(C)。在这个例子中，直方图只有两个非零部分（结果是简单而重复的纹理结构）

训练和预训练检测器

OpenCV附带一些预先训练的目标检测文件，但是还有一些代码可以用于训练和存储检测器的新目标模型。如果这些不能满足你的需要，还可以使用包（opencv_traincascade in .../*opencv/apps*）中的应用程序，你可以使用它们来训练任何刚体的检测器，尽管对于不同目标它的适用性会大不相同。

OpenCV为检测器提供的预训练对象位于.../*opencv/data/haarcascades*和.../*opencv/data/lbpcascades*目录中。目前，最适合前向人脸检测的模型是*haarcascade_frontalface_alt2.xml*。侧面人脸观察使用这种技术难以准确检测（我们之后将进行介绍），并且那些附加模型运行效果更不好，但由于2012年谷歌之夏编程训练营的工作，这已经得到了极大的改善。如果最终训练出较好的目标模型，也许可以考虑把它们作为开源代码贡献给社区。

有监督学习和boosting理论

OpenCV中的级联分类器是一个有监督的分类器（这在第21章已经讨论过）。在这种情况下，通常需要向分类器提供直方图和大小不等的图像补丁，之后将其标记为包含（或者不包含）感兴趣的目标。对于该分类器来说，最常见的目标是人脸。

Viola-Jones检测器使用AdaBoost，但是在更广阔的背景下被称为“拒绝级联”。这种“级联”是一系列的节点，每一个节点本身就是独特的多叉树AdaBoost分类器。级联的基本操作是从图像的子窗口按照特定顺序对所有节点进行顺序测试，并且将“通过”每个分类器的窗口视为正在寻找的类的成分。

为了实现这一点，每个节点被设计以一个低拒绝率（接近50%）（高假阳性，或者“非人脸”被错误分类的概率，即“本来是负示例，分类成正示例”，通常称“误报”）下，具有高检测率（即99.9%）（低假阴性，或者丢失人脸的概率，即“本来是正示例，分类成负示例”，通常叫“漏报”）。对于每一个节点，级联任何阶段出现一个“不在分类中”的结果都将安全地终止计算，然后算法声明在该区域内不存在人脸。

只有正在考虑的区域通过整个级联分类器后才能声明一个真正的类检测。当真正的类比较稀少的时候（例如，图片中的人脸），拒绝级联可以极大地减少整体计算量，因为大部分进行人脸搜索的区域都会在遇到非类决策时快速终止。通过在级联开始放置最简单（计算最快）的节点，可以进一步提高这一特性。

Haar级联boosting

对于Viola-Jones拒绝级联，每一个节点本身就是一个弱分类器的集合，通过boosting组合，形成一个强分类器节点。这些单个的弱分类器本身是决策树，通常只有一层的深度

（即决策桩）。决策桩只有一个决策，形式如下："特定特征h的值v是否高于或低于一定阈值t"，之后，例如，"是"表示存在人脸，"否"表示不存在人脸：

$$h_w = \begin{cases} +1 & \nu_w \geq t_w \\ -1 & \nu_w < t_w \end{cases}$$

Viola-Jones分类器在每个弱分类器中使用的类Haar或LBP特征的数量可以在训练中设置，但是大多数都坚持使用单个特征树桩；在一些情况下最多使用三个特征。[注2]之后使用这些弱分类器的加权和，boosting迭代地建立强分类器节点。Viola-Jones分类器使用分类功能：

$$H = sign(\alpha_1 h_1 + \alpha_2 h_2 + \cdots + \alpha N_{w_h} N_w)$$

其中，如果数量小于0则符号函数返回-1，如果数量等于0则返回0，如果数量大于0，则返回+1。在首次通过数据集时，我们获得每个最佳输入分类h_w的阈值t_w。之后Boosting使用结果误差计算权重α_w。在传统的AdaBoost中，每一个特征向量（数据点）根据分类器的迭代是否被正确分类，被重新加权为低或高。[注3]一旦节点通过这种方式完成学习，级联中留下的较高层次的数据就会被用于下一个节点的训练，以此类推。

拒绝级联。图22-3可视化了Viola-Jones拒绝级联，由许多boosting级联器组成。在图中，每一个节点F_j包含一整个boosting级联器组，在人脸和非人脸（或者用户选择的其他用于训练的目标）特征上训练决策桩（或者树）。回想一下，通过将节点从简单到复杂进行排列以最小化计算成本。通常，每个节点的boosting被设置为有一个非常高的检测率（通常以大量假阳性结果为代价）。当在人脸上进行训练时，例如，虽然几乎所有（99.9%）的人脸都能被发现，但是还会有许多非人脸在每个节点被错误地"检测到"。但这是允许的，因为使用20个节点就可以达到$0.999^{20} \approx 98\%$的人脸识别率（通过整个级联器），假阳性率仅为$0.5^{20} \approx 0.0001\%$!

在运行模式下，原始图像将被不同大小的搜索区域扫过。在实际情况中，70%~80%的非人脸在拒绝级联器的前两个节点就已经被拒绝，每个节点使用大约10个决策桩。这种快速而早期的"有意识拒绝"极大加快了人脸识别的速度，并且这正好是Viola-Jones算法实用性的基础。

注2：在第21章我们讨论过AdaBoost，这一切读者都应该很熟悉了。

注3：有时候，混乱有助于boosting降低在训练中正确分类的点的分类权重，并提高其分类错误的点的权重。原因是boosting试图把重点放在纠正"麻烦"的点上，忽略它已经"知道"如何分类的点。对此，有一个技术术语是boosting就是margin最大化。

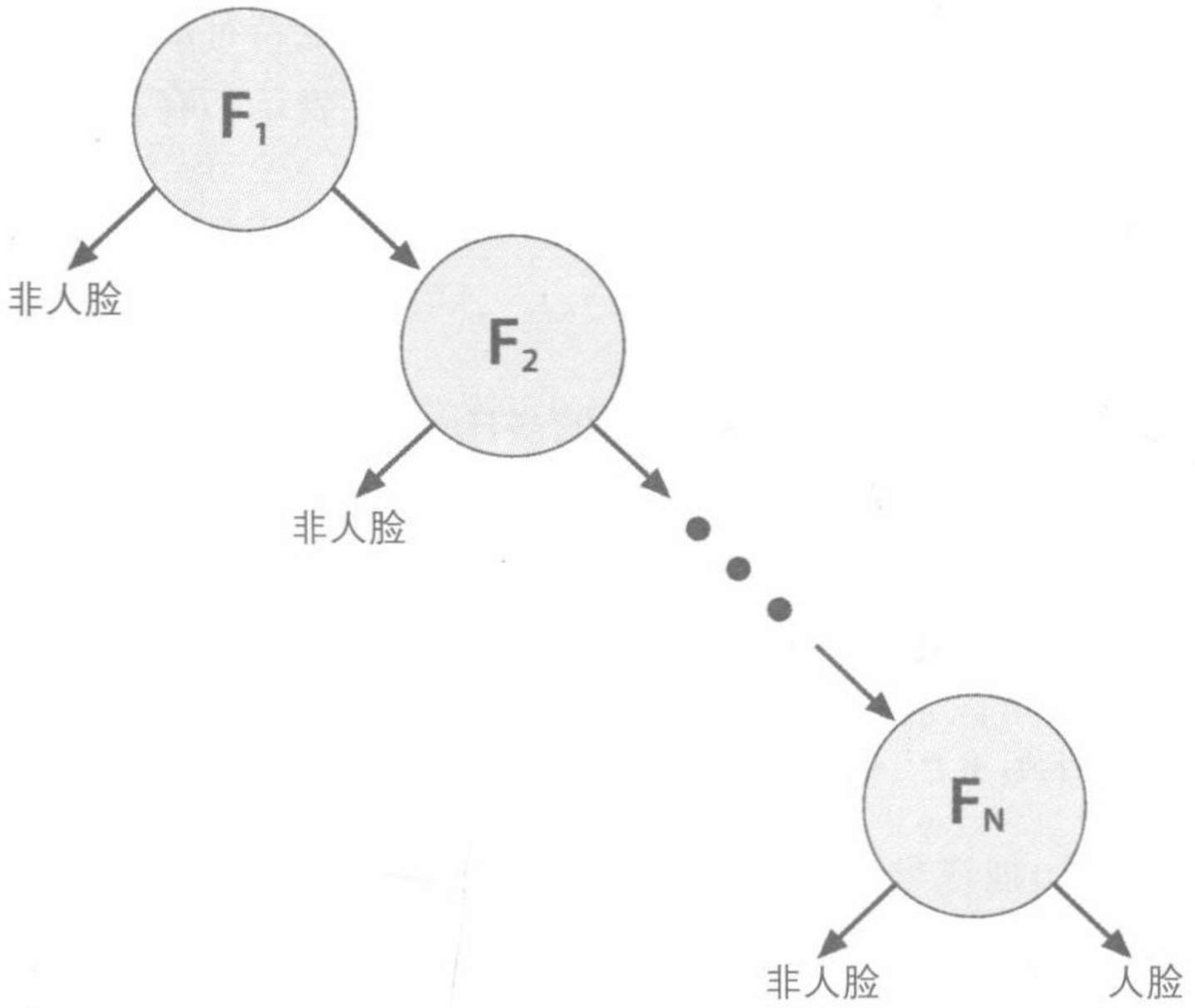


图22-3. Viola-Jones分类器中使用的拒绝级联：每个节点代表一个多叉树boosting强分类器集成，将逐步调整到很少错过真正的人脸，同时拒绝一小部分的非人脸；然而，几乎所有的非人脸都已经被最后一个节点拒绝，仅仅留下了真正的人脸

如前所述，这种技术实现了人脸识别，但是不局限于人脸识别，它在其他具有独特视图的物体上（大多数为刚体）同样有很好的运行效果。人脸前向视图运行较好；汽车的后向、侧向或侧向运行较好；但是人脸的侧向视图或汽车的“斜向”视图运行效果不太好——主要因为这些视图引入了模板中的变量，这个检测器中使用的“块状”特征处理效果不好。例如，人脸的侧面视图必须在学习模型中捕获一部分变化的背景，以便包含轮廓曲线。为了识别人脸的侧面视图，你可以尝试使用*haarcascade_profileface.xml*，但是为了实现更好的效果，应该（比这个模型）收集更多的数据进行训练，同时也根据人脸轮廓背后不同的背景扩展数据。在2012谷歌编程之夏训练营中，*lbpcascade_profileface.xml*的引入改进了这一性能。



轮廓视图对Haar级联分类器来说是很难的，因为它使用块特征，因此不得不尝试学习背景的变化，“窥视”人脸侧面视图的信息化轮廓边缘。这是一个比较笼统的问题，既影响Haar，同时也在一定程度上影响LBP特征。在大多数情况下，最好对检测器进行训练以找到一个正方形区域，它是正在查找目标的一个子区域。这就是组件中预训练级联器如何处理头部为圆形的真相：它们寻找被的是人脸包围的正方形区域。

在训练过程中，只学习一个人脸轮廓（例如就像右侧）更为高效。之后检验的过程中，有两步：(1)运行右侧轮廓检测器，(2)在其垂直轴上旋转图像左右翻转，并在此运行右侧轮廓检测器以检测左侧人脸轮廓。[注4]

概括起来，基于这些类Haar特征和LBP特征的检测器对“块状”特征的检测效果较好——例如眼睛、嘴和发丝——但是对树干（举个例子）或者当物体轮廓形状是其最明显的特征时（就像咖啡杯一样），检测效果不好。在这里LBP有一些优势，因为对LBP来说背景区域可能只是在其直方图中增加一些梯度噪声，而对类Haar特征来说，则会受到背景块之间求和（偏移水平）的影响。

刚刚提到的所有这些，如果打算从相当刚性的目标上收集大量非常好的、良好分割的数据，那么这个分类器仍然非常有竞争力，同时拒绝级联的构造使其运行速度非常快（尽管训练速度并不快）。在这里，“大量数据”意味着成千个目标实例和上万个非目标实例。“好的”数据，举个例子，比如说，指我们不应该将倾斜的脸与直立的脸混合；反而应该保持数据独立并使用两个分类器，一个用于倾斜目标，另一个用于直立目标。“良好分割”数据指的是连续网格的。在训练数据网格边缘的疏忽通常会导致分类器尝试纠正数据中的假想变化。例如，在人脸数据位置网格中，眼睛的不同位置将导致分类器认为眼睛位置不是人脸的固定几何特征，并因此可以自由移动。当分类器尝试调整实际上并不存在于真实数据中的事物时，识别效果在大多数情况下都比较差。

Viola-Jones分类器总结

Viola-Jones分类器在级联的每一个节点使用AdaBoost来学习多叉树（通常是多叉决策桩）分类器。除此之外，算法还包含其他几项创新功能。[注5]

- 使用可以快速计算的特征（例如，Haar特征是一个阈值，应用于矩形图像区域的和与差）。
- 图像积分技术使得对矩形区域的值或这种区域旋转45度后的快速计算成为可能（参考第6章）。这种数据结构被用于加速输入特征的计算。
- 使用静态boosting来生成以高识别率与弱拒绝率为特征的二元（人脸或非人脸）分类器节点。

注4：这涉及到机器学习中的一个更普遍的原则。强迫算法去学习一种你已知存在的对称性并没有什么好处。在破坏对称性之后的样本上训练算法并根据其在算法训练之前的对称性再映射到输入中，这样做效果更好。

注5：这几项功能都是现代计算机视觉研究人员或从业者所用工具中的标准技巧。

它将这些分类器节点组合成一个拒绝级联。换句话说，选择第一类分类器，可以最佳检测到包含目标的图像区域，但同时会有许多错误检测；下一种分类器是对存在弱拒绝时进行检测的次优分类器；以此类推。在测试模式下，只有通过整个级联，才能说检测到了目标。

cv::CascadeClassifer

根据第21章机器学习库中的大量例程，OpenCV将级联分类器作为对象进行了实现。这个对象称为cv::CascadeClassifer，存储加载（或训练）的级联，同时提供一个接口（用于运行在图像上运行检测通过）。

cv::CascadeClassifer对象的结构是：

```
cv::CascadeClassifier::CascadeClassifier(
const String& filename
);
```

这个构造中只用了一个参数：级联器存储文件的名字。如果在之后使用load()进行加载，还有一个默认构造可以使用。

利用detectMultiScale()搜索图像

级联分类器中实际使用的函数是cv::CascadeClassifier对象的detectMultiScale()方法：

```
cv::CascadeClassifier::detectMultiScale(
const cv::Mat&    image,                    // Input (grayscale) image
vector<cv::Rect>& objects,                  // Output boxes (boxen?)
double            scaleFactor  = 1.1,       // Factor between scales
int               minNeighbors = 3,         // Required neighbors to count
int               flags        = 0,         // Flags (old style cascades)
cv::Size          minSize      = cv::Size(), // Smallest we will consider
cv::Size          maxSize      = cv::Size()  // Largest we will consider
);
```

第一个输入image是一个CV_8U类型的灰度图像。cv::CascadeClassi fier::detectMultiScale()函数会扫描输入图像中所有尺度上的人脸。被成功定位的目标将以边界矩形的形式返回到向量目标中。scaleFactor参数的设置可以决定每个尺度之间的跳变有多大，将其设置为一个较大的值意味着，算法会以可能错过某些尺寸人脸的检测为代价来换取更快的计算时间。参数minNeighbors是对阻止错误检测的控制。实际图像中人脸的定位倾向于在同一片区域中获得多个“命中”，因为周围的像素和缩放通常都会表示人脸。将其设置为人脸识别代码中的默认值(3)意味着只有至少存在三个重叠的检测标记时，我们才能认定有人脸存在。

在这里，参数flags被忽略，除非你使用的级联器是由旧版本的OpenCV 1.x级联工具产

生的。在这种情况下，它可能被设置为（旧版本的OpenCV 1.x也一样）：CV_HAAR_DO_CANNY_PRUNING。在这种情况下，Canny边缘检测器将用于拒绝一些区域。

最后的参数minSize和maxsize搜索人脸区域面积的最小值和最大值。设置这两个值将以错过异常小或者异常大的人脸为代价减少计算量。（在许多实际情况中，这是非常有必要的，因为你通常会对图像中人脸占据多少帧有一个预估，你发现的其他可能都只是噪声。）图22-4显示了使用人脸识别代码在场景中识别人脸的结果。

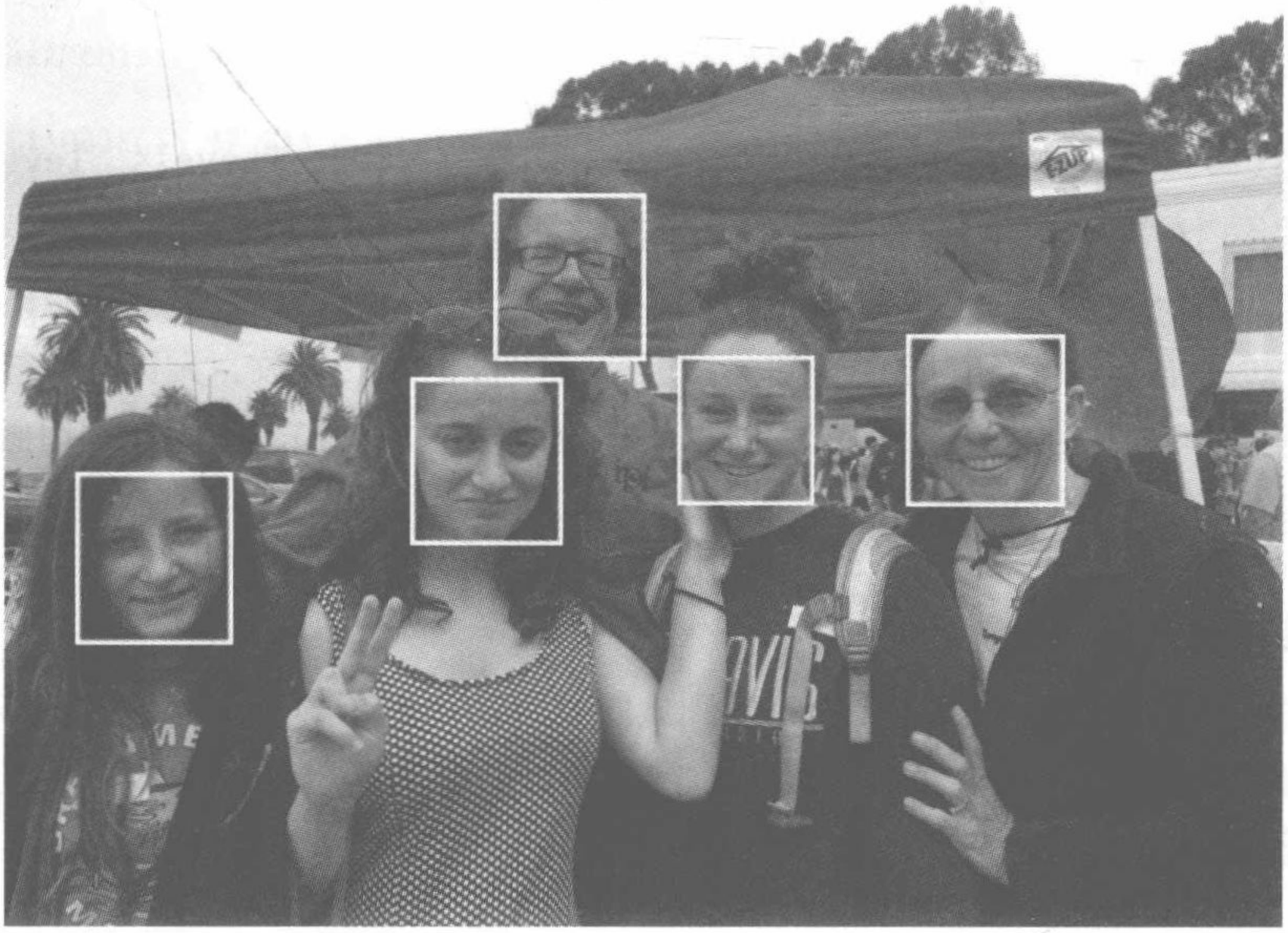


图22-4. 在公园场景中的人脸识别：即使是歪着头的人，都能进行识别；对于所示的1111×827图像，在3 GHz的机器上搜索超过百万的位置和尺度，约0.25秒之内得到这个结果

这里有另外两个类似的检测方法：

```
void detectMultiScale(
  cv::InputArray    image,
  vector<cv::Rect>& objects,
  vector<int>&      numDetections,
  double            scaleFactor       = 1.1,
  int               minNeighbors      = 3,
  int               flags             = 0,
  cv::Size          minSize           = cv::Size(),
  cv::Size          maxSize           = cv::Size()
);

void detectMultiScale(
  cv::InputArray    image,
  vector<cv::Rect>& objects,
  vector<int>&      rejectLevels,
```

```
  vector<double>&   levelWeights,
  double            scaleFactor        = 1.1,
  int               minNeighbors       = 3,
  int               flags              = 0,
  cv::Size          minSize            = cv::Size(),
  cv::Size          maxSize            = cv::Size(),
  bool              outputRejectLevels = false
);
```

第一个与我们之前看到的构造函数基本相同，但是添加了新的参数numDetections。这是一个包含条目数量与objects相同的输出。对于numDetections中的每个条目，该值都说明了objects中相应条目做出的对象检测数。

detectMultiScale()的第二种替代形式有三个附加参数：rejectLevels，levelWeights和outputRejectLevels。如果最后一个被设置为true，则前两个仅用于返回。对于objects中的每一个条目，rejectLevels和levelWeights都是只有一个条目的向量。在rejectLevels中，这个条目包含子图像在级联器中被拒绝的级数。无论是被接受还是被拒绝，levelWeights数组都包含最后一级中弱分类器的加权和。这允许调用函数通过考虑附加信息来处理所选择的任何方式的边界情况。

人脸识别实例

示例22-1中的detectAndDraw()代码将识别人脸并用不同颜色的矩形在图像上标记出它们的位置。如注释所示，这个代码假设之前训练的级联分类器已经被加载，并且用于识别人脸的存储器也已经被加载。

示例22-1：识别并标注人脸

```
// Detect and draw detected object boxes on image
//
// Presumes 2 Globals:
//
//  cascade is loaded by something like:
//     cv::Ptr<CascadeClassifier> cascade( new CascadeClassifier( cascade_name ) );
//
void detectAndDraw(
cv::Mat&                         img,          // Input image
cv::Ptr<cv::CascadeClassifier> classifer,     // Preloaded classifier
  double                           scale    = 1.3 // resize image by...
){

  // Just some pretty colors to draw with
  //
  enum { BLUE, AQUA, CYAN, GREEN };
  static cv::Scalar colors[] = {
    cv::Scalar(  0,   0, 255 ),
    cv::Scalar(  0, 128, 255 ),
    cv::Scalar(  0, 255, 255 ),
```

```
    cv::Scalar(  0, 255,   0 )
};

// IMAGE PREPARATION:
//
cv::Mat gray( img.size(), CV_8UC1 );
cv::May small_img(
  cvSize( cvRound(img.cols/scale), cvRound(img.rows/scale)),
  CV_8UC1
);
cv::cvtColor( img, gray, cv::BGR2GRAY );
cv::resize( gray, small_img, cv::INTER_LINEAR );
cv::equalizeHist( small_img, small_img );

// DETECT OBJECTS IF ANY
//
vector<cv::Rect> objects;
classifier->detectMultiScale(
  small_img,                    // The input image
  objects,                      // A place for the results
  1.1,                          // Scale Factor
  2,                            // Minimum number of neighbors
  cv::HAAR_DO_CANNY_PRUNING,    // (old format cascades only)
  cv::Size(30, 30)              // Throw away detections smaller than this
);

// LOOP THROUGH FOUND OBJECTS AND DRAW BOXES AROUND THEM
//
  for( vector<cv::rect>::iterator r=objects.begin(); r!=objects.end; ++r ) {
    Rect r_ = (*r)*scale;
    cv::rectangle( img, r_, colors[i%4] );
}

}
```

为了方便起见，在这个代码中，detectAndDraw()函数有一个颜色的静态向量colors[]，可以索引用于使用不同颜色绘制找到的人脸。分类器基于灰度图像工作，因此传入该函数的彩色BGR图像img通过cv::cvtColor()转换成灰度，然后选择在cv::resize()中调整大小。

这之后是通过cv::equalizeHist()的直方图均衡，它会展开图像的亮度值。这是非常重要的一步。因为积分图像特征基于矩形区域的差，如果直方图是不平衡的，这些差异就有可能由于整体光照或者测试图像的曝光而倾斜，所以这一步非常必要。在训练级联时，这对输入数据也十分重要。

实际的识别仅在for循环的基础上进行，然后循环遍历搜索人脸的矩形区域，使用cv::rectangle()用不同颜色将它们绘制出来。

学习新目标

我们已经见过如何加载和运行存储在一个XML文件中预训练过的级联器，在最初调用cv::CascadeClassifier构造函数时或者之后使用cv::CascadeClassifier::load()方法时加载它。一旦对分类器完成加载，我们就可以使用cv::CascadeClassifier::detectMultiScale()函数对目标进行实际识别。现在，我们将问题转向如何训练自己的分类器来识别其他目标，例如眼睛、行走的人、汽车，等等。使用OpenCV的应用*traincascade*实现，[注6]它可以创建一个分类器，给出正的和负的训练样本序列。训练分类器的四个步骤如下所述。

1. 收集一组数据集，其中包含你希望得到的目标的示例（例如，人脸的前向视图，汽车的侧向视图）。这些将存储在一个或者多个目录下，使用如下集合描述文件形式的文本文件索引：

   ```
   <path>/<img_name_1> <count_1> <x11> <y11> <w11> <h11> <x12> <y12> ...
   <path>/<img_name_2> <count_2> <x21> <y21> <w21> <h21> <x22> <y22> ...
   ...
   ```

   每一行包含目标所在图像的路径（如果存在）和文件名。随后是存在于图像中目标的数量，之后是一列包含目标的矩形。这些矩形的格式是其左上角的x坐标和y坐标，紧接着是以像素为单位的宽度和高度。

   更具体地说，如果我们有一组人脸的数据集位于目录*data/faces/*，那么集合描述文件*faces.dat*将为如下形式：

   ```
   data/faces/face_000.jpg 2 73 100 25 37 133 123 30 45
   data/faces/face_001.jpg 1 155 200 55 78
    . . .
   ```

   如果希望分类器运行得更好，就需要收集高质量和大量（1000－10000正的示例）的数据。“高质量”意味着你已经去除数据中所有不必要的变量。例如，如果你正在研究人脸，就需要尽可能将眼睛（同时最好包括鼻子和嘴）对准。这里的直觉是，除非你正训练分类器眼睛不需要位于人脸固定的位置，而是可以在某个区域中的任何位置。由于这不是真实数据，所以分类器的效果将不会那么好。一个策略是首先在子区域上训练级联器，比如说“眼睛”，它们更容易对齐。之后使用人眼检测来搜寻眼睛并且旋转/调整人脸大小，直到眼睛对准。对于不对称数据在垂直轴上翻转图像的“技巧”，我们在前面“拒绝级联”一小节中已经进行了描述。

注6：你当然还可以用老版本的haartraining来训练级联分类器，但是这样得到的级联分类器也仍然会是老版本的类型，所以其实我们并不推荐。还有就是，只有traincascade才支持LBP特征（当然也支持Haar特征）。另外，也只有traincascade才支持多线程的TBB，因此训练速度也更快（前提是你在编译OpenCV库的时候开启了TBB支持——即有这个选项-D WITH_TBB=ON）。

2. 一旦获得了数据集，就使用实用程序createsamples建立一个正样本的“矢量”输出文件。使用这个文件，你可以在多次运行中重复即将到来的训练过程，在使用同一个计算矩阵文件时尝试不同参数。下面是使用createsamples的一个例子：

```
createsamples -info faces.dat -vec faces.vec -w 30 -h 40
```

该命令读入步骤1中描述的*face.dat*文件，并输出一个格式化的向量文件，在这种情况下称为*face.vec*。在内部，createsamples从图像中提取正样本，对其进行归一化，并将其调整为指定宽度和高度（在本例中为30×40像素）。注意，还可以使用createsamples通过应用几何变换、添加噪声、改变颜色等来整合数据。当仅有一个原型（例如公司商标）并想要采用这一个图像并使其进行真实图像中可能出现的一些扭曲时，这一过程格外有用。（这些选项对应的更多细节将很快介绍。）

3. 生成一组反例。训练过程将使用这些“非”样本研究那些看似不像你的目标的样本。针对训练目的，任何不包含感兴趣目标的图像都会变成负样本。最好从将要测试的同一类型数据中获取“非”图像；也就是，如果我们想要在在线视频中研究人脸，为了获取最好的结果，应该从可类比的帧（或者相同视频中的其他帧）获取负样本。当然，我们也可以通过使用从任何地方（例如，网络图像收集）获得的负样本实现相当好的结果。再一次，我们将图像放入一个或多个目录，之后制作一个由这些图像文件名、路径列表组成的集合文件，每个一行。[注7]例如，我们可以生成一个图像集合文件并称其为*background.dat*，它的内容包括图像的路径和文件名，具体如下：

```
data/vacations/beach.jpg
data/nonfaces/img_043.bmp
data/nonfaces/257-5799_IMG.JPG
...
```

4. 训练级联器。以下是可以在命令行中键入的示例，目的是创建一个名为*face_classifier_take_3.xml*的训练级联：

```
traincascade                                          /
  -data face_classifier_take_3                        /
  -vec opencv/data/vec_files/trainingfaces_24-24.vec  /
  -w 24 -h 24                                         /
  -bg backgrounds.dat                                 /
  -nstages 20                                         /
  -nsplits 1                                          /
  [-nonsym]                                           /
  -minhitrate 0.998                                   /
  -maxfalsealarm 0.5
```

注7：集合文件（用于反例）是一个只包含文件名列表的文件，一个集合描述文件（用于正例）包含文件名列表和相关信息（指出每个文件中哪个位置可以找到感兴趣的对象的文件）。

.xml文件扩展名自动添加进-data参数，在这种情况下，创建输出文件*face_classifier_take_3.xml*。在这里，*trainingfaces_24-24.vec*是一组正样本（大小为24×24），从*background.dat*中提取的任何图像都将被用作负样本。级联器被设置为20个（-nstages）阶段，其中每一阶段都有训练出0.998或者更高的识别率（-minhitrate）。每个阶段的假命中率（-maxfalsealarm）被设置为50%（或更低），使总体的命中率达到0.998。在这种情况下弱分类器被指定为“决策桩”，意味着它们只能有一次分割（-nsplits），我们可以要求更多，这在一些情况下会改善结果。对于更复杂的目标，可能需要多至6次分叉，但是通常我们希望使这个数保持较小的值，使用不超过3次分叉。

即使在快速运行的机器上，根据数据集的大小不同可能也要花好几个小时或几天进行训练。训练过程必须在训练窗口内对所有正样本和负样本测试大约100 000个特征。该搜索是可并行的，并且可以利用多核机器（使用TBB）。此并行版本是OpenCV附带的版本（假设使用TBB支持的库，即-D WITH_TBB=ON）。

createsamples的详细参数

正如我们所看到的，为了安排正样本，使其可以通过traincascade进行摄取，你需要使用createsamples程序。[注8]createsamples程序不仅可以从找到的图像中裁剪各个样本，同时还可以自动修改具有稍微不同方向、光照和其他特征的样本。在这里，我们将详细介绍在调用createsamples时可用的选项以及这些选项的作用。

在调用traincascade时，将运行四种模式中的一种。模式由选择的选项来决定。[注9]具体而言，选项-img，-info，-vec和-bg共同决定运行模式。四种运行模式如下。

模式1：通过单个图像（通过应用畸变）创建训练样本

从单个图像开始（由-img指定），通过将畸变应用在单个输入图像上，产生一些新的测试图像（由-num指定），之后从背景集（由-bg参数指定）将其粘贴到图像上。由于正在生成训练图像，所以输出将是向量文件（由-vec参数指定）：

```
createsamples -img <image_file> -vec <vector_file> \
-bg <collection_file> -num <n_samples> ...
```

模式2：通过单个图像（通过应用畸变）创建测试样本

这个形式与模式1非常类似，主要在输出文件格式上有所不同。这用于创建新的图像，你可以用来测试自己的检测器。它将集合描述文件中目标的旋转和畸变应用于

注8：但是负样本却不需要这样做，因为traincascade会从你提供的负样本文件列表里动态地选择随机样本。因此，既然详细的裁切或者准备工作已经被做了，你也就不需要再额外地为负样本进行这一操作了。

注9：也就是说，模式并不是由名为“-mode”的参数选择的，而是从你提供的其他参数推断出来的。

输出图像（-img参数），然后通过将已畸变的原始图像与背景图像结合（由-bg参数给出的收集文件）产生新图像。[注10]这个的关键点是有一个图像，其中感兴趣的目标（尽管是一个畸变的）被放置在已知位置，使你可以测试检测器。产生的文件的文件名类似于<*number*>_<*x*>_<*y*>_<*width*>_<*height*>.*jpg*，其中<*x*>和<*y*>等的值由嵌入（否则就是背景）图像的目标的位置和大小指定。由于正在生成测试样本，结果将位于一个集合描述文件（-info参数），其中包含产生的文件名和插入目标在这些文件中的位置。

```
createsamples -img <image_file> -bg <collection_file> \
-info <collection_description_file> ...
```

模式3：通过一个图像集合（无畸变）产生训练样本

这是我们在前面小节中描述过的用法，可以认为是一种文件格式的转换。它只是收集了所有由-info图像指定的图像，按照指定信息进行裁剪，并构建-vec指定的向量文件。

```
createsamples -info <collection_description_file>  \
-vec <vector_file>                                  \
-w <width> -h <height> ...
```

模式4：通过.vec文件观察样本

在这个模式中，所有*.vec*文件中的样本将一一显示在屏幕上。这主要是为了调试并帮助理解这一过程。

```
createsamples -vec <vector_file>
```



对于这四种模式，你最想用的可能不在这个列表当中。在实践中，你通常发现自己有一些示例图像（可能是1000个），并希望通过畸变和转换产生更多（可能是7000个）的示例。在这种情况下，你想要做的实际上是使用模式1 1000次，针对每一个原图像产生7个新的图像。在这种情况下，需要对流程稍微做一些自动化。如果要生成大量测试图像，也属于这样的情形。

这里是每一个选项的详细描述。

-vec <*file_name*>

这是将由createsamples产生的文件的名称。它应该有一个*.vec*文件扩展名。

注10：虽然这个处理过程的结果看起来可能很奇怪，比如在你提供的一幅背景图像上漂浮着一个略微倾斜的很有魔幻感的人头，但这个过程对于确保旋转和失真没有把黑色或者空白像素引入最终的训练图像来说非常必要。

-info <*file_name*>

这是指定输入示例集合的文件的名字，包括文件名和在图像中示例目标的位置（例如之前描述的*face.dat*文件）。

-img <*file_name*>

这是-info的替代（必须提供其中一个）。使用-img，你可以提供单个裁剪的正向示例。在使用-img的模式中，将产生多个输出，且都来自于这一个输入。

-bg <*file_name*>

-bg扩展名允许你指定一个文件（另外一个*.dat*扩展名），包含提供的背景图像列表的名称和辅助信息。

-num <*n_samples*>

-num设置了应该生成的正向示例的数量（即由-vec指定的输入样本上的转换）。

-bgcolor <*color*>

该强度值在输入图像上解释为“透明”。请注意，灰度图像是假定的。当将正向示例重叠到备用背景上时，就会使用此选项。

-bgthresh <*delta*>

在许多实际情况下，输入图像将包含压缩失真（例如*.jpg*文件）。由于这些失真，背景可能并不总是相同的固定颜色。与-bgthresh <*color*>, -bgthresh <*delta*>结合使用将导致阈值内[*color - delta*, *color + delta*]的所有像素解释为透明。

-inv

如果指定，所有图像将在样本提取之前反转。

-randinv

如果指定，在提取样本之前每个图像将反转或者不反转（随机）。

-maxidev <*deviation*>

如果指定，则在提取前每个图像随机（均匀）变亮或者变暗，达到该值。

-maxxangle <*angle*>, -maxyangle <*angle*>, -maxzangle <*angle*>

如果指定，每个图像将在每个方向上随机旋转达到给定量而失真。这提供了对象的可能的视点透视偏移的近似（尽管如此，为了定义这些变换，对象仍然需要假设是有效的平面物体）。这些旋转的单位是弧度。

-show

如果指定，每一个样本将被显示。按下ESC按键，继续样本创建过程，而不显示之后的样本。

-w <*width*>

产生的样本之宽度（以像素为单位）。

-h <*height*>

产生的样本之高度（以像素为单位）。



大量实验已经进行过，用来决定用于人脸识别样本的最佳大小。通常，18×18或者20×20效果很好。对于其他不是人脸的目标，你可能需要进行实验，找出对自己这种特定情况效果最好的样本大小。

正如你看到的，对于createsamples有很多的选项。最重要的是，大多数这些选项用于自动创建在可用示例中提供的图像的变形。使用这些选项，你可以（并且通常会）将成百上千幅样本图像转换成数十倍于原来数量的实际训练分类器的图像。

traincascade的详细参数

与createsamples一样，有许多选项可以传递给traincascade，对它的行为进行微调。这些包括调整级联器本身、boosting方法和使用的特征类型等的参数。这些参数将极大影响级联器训练时间，同时也将影响最终结果的质量。

-data <*classifier_file*>

-data参数指定创建的输出－训练分类器文件的名称。不需要你提供*.xml*文件扩展名，它将为你添加。

-vec <*vector_file*>

-vec参数指定了正向示例输出向量文件的文件名称（例如，使用createsamples创建）。

-bg <*collection_file*>

这一参数指定背景图像集合文件的名称。

-numPos <*n_samples*>

这是将用于训练分类器每一个环节的负示例的数量（通常少于所提供的示例的数量）。

-numStages <*stages*>

-numStages参数指定将被用于整个分类器的级联环节的数量。

-precalcValBufSize <*size-megabytes*>

这用于分配存储预先计算的特征值的缓冲区的大小。缓冲区大小以兆字节确定。这一缓冲区被Boost实现用来存储特征评估的结果，使其不必在每次需要时重新计算。最终的结果是，这种方法将大大提高训练的运行时间。当前的默认值是256兆字节。

-precalcIdxBufSize <*size-megabytes*>

这与-precalcValBufSize类似，同时也同样用于Boost实现。-precalcIdxBufSize设置了用于“缓冲区索引值”缓存的大小。类似于-precalcValBufSize，到底是什么，这对我们来说并不重要；对这些目标来说，重要的是缓存（以提高性能的能力）。当前的默认值是256兆字节，如果修改二者中的一个，最好使其保持一致。

-baseFormatSave <{true,false}>

如果使用Haar型特征并希望以“旧样式”格式保存级联器，则可以将其设置为true。这一参数的默认值是false。

-stageType <{BOOST}>

这一参数设置了将用分类器训练的环节类型。现在，它只有唯一一个选项，Boost（表示“boosting分类器级联”）并作为默认值，所以现在你可以忽略它。在这里，这个参数留待日后进行改进。

-featureType <{HAAR, LBP}>

目前级联分类器支持两种不同的特征类型：（类）Haar特征和局部二值模式特征。你可以通过设置-featureType为HAAR或LBP（分别）来选择一个。

-w <*sample_width-pixels*>

-w参数向traincascade告知你提供给createsamples的样本的宽度。传递给-w参数的值必须等于createsamples使用的值。-w的单位为像素。

-h <*sample_width-pixels*>

-h参数向traincascade告知你提供给createsamples的样本的高度。传递给-h参数的值必须等于createsamples使用的值。-h的单位为像素。

-bt <{DAB, RAB, LB, GAB}>

traincascade可以使用boosting四种可用变量中的任一来训练级联器（参考之前对boosting的讨论）。可用的选项分别是Discrete AdaBoost (DAB)，Real AdaBoost (RAB), LogitBoost (LB)和Gentle AdaBoost (GAB)。默认值是GAB。

-minHitRate <*rate*>

最小命中率-minHitRate设置窗口中被标记为命中的实际出现的目标百分比。当然，在理想情况下应该为100%。但是，训练算法永远无法实现这一点。这个参数的值被归一化，因此默认值0.995对应于99.5%。这是每一环节的目标命中率，所以最终的命中率将（大概）是被提升到阶段数量的这个幂的目标。

-maxFalseAlarmRate <*rate*>

最大错误警告率-maxFalseAlarmRate <*rate*>设置窗口中能被预计（错误的）标记为命中的错误事件的目标百分比。理想情况下，这个值应该为0%，但是在实际中，它

相当大，并且我们依靠级联器来逐渐拒绝错误警告。这一参数的值被归一化，因此默认值0.50对应于50%。这是每一环节的目标错误正向率，所以最终命中率将（大约）是被提升到阶段数量的这个幂的目标。

-weightTrimRate <*rate*>

这个参数我们在之前遇到过，作为boosting算法的参数。它用于选择在一个特定的boosting迭代中使用哪个训练样本。只有那些权重大于1.0减去权重调整率的样本会参与到任何给定迭代的训练，这个参数的默认值是0.95。

-maxDepth <*depth*>

该参数设置每个弱分类器的最大深度。注意，这不是级联器的深度，而是本身构成级联元素的树的深度。该参数默认值为1，对应简单决策桩。

-maxWeakCount <*count*>

类似-maxDepth，参数-maxWeakCount被直接传递给级联分类器的boosting，同时设置可以用于形成每一个强分类器（即，分类器的每一阶段）的弱级联器的最大数量。这个参数的默认值是100，但是请记住，这不意味着弱分类器一定会使用这个数。

-mode <BASIC | CORE | ALL>

参数-mode与类Haar特征一起使用，决定是只用原始Haar特征（BASIC或者CORE），还是使用扩展特征（ALL）。使用的特征如图22-5所示。

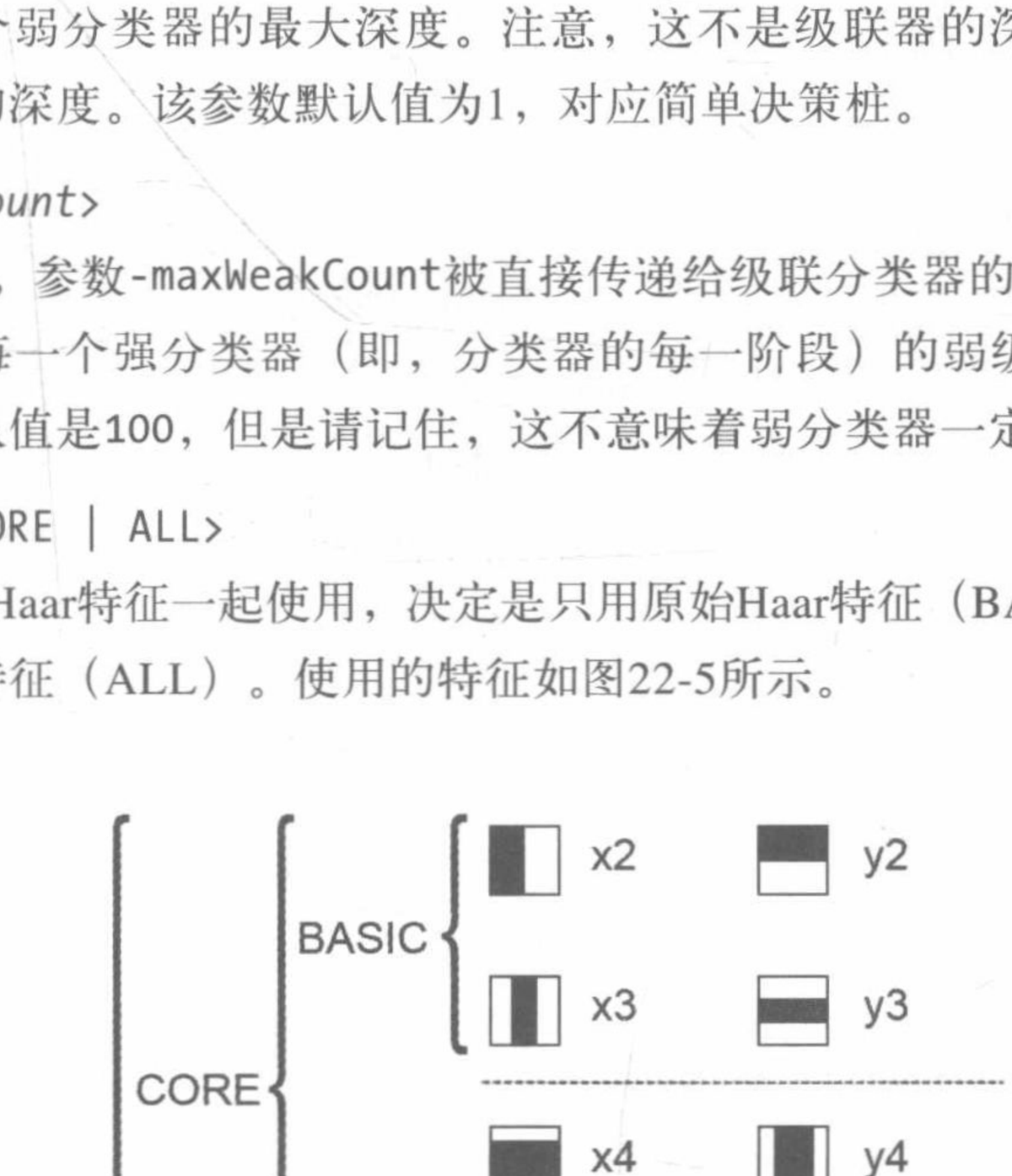


图22-5. 参数-mode的选项。BASIC只包括最简单的Haar小波集合（在每个方向一个偶数和一个奇数小波）。CORE包括四个额外的高阶Haar小波，ALL包括（一些）其他小波的旋转版本的对角线元素

尽管可用于traincascade的选项数量有些吓人，你可能会发现默认值将为许多现实情况提供良好的效果。如果你要为特定目标类型（非人脸）训练级联器，那么就值得去网络上或者对相关文献进行检索，查看是否有人已经为你的特定目标训练了级联器。通常，其他人已经做了很多工作来优化你的目标训练参数的选择，或者至少类似。

使用支持向量机的目标识别

与基于树的方法的目标识别技术相类似，在OpenCV中有另一类算法，它使用支持向量机作为学习策略的基础。但是正如我们在基于树的函数中看到的，还有大量附加组件用于限制、组织和处理数据与可重复进行的训练过程。

在本节中，我们将学习两种方法：Latent SVM（隐式支持向量机）和Bag of Words（词袋）方法。尽管这两种方法都依赖于支持向量机，但它们的意义完全不同。Latent SVM非常适合于识别可变形目标（如行人），因为它明确地概念化了由可变性结构链接在一起的多个子组件的概念。Bag of Words使用一种不同的方式并完全忽略了大规模的结构，它从文档识别技术中获得灵感，只考虑目标组件的列表。通过这种方法，Bag of Words可以拥有不只局限于目标识别的通用性，它还可以用于整个场景的识别以及场景分析。

Latent SVM用于目标识别

Latent SVM算法由Pedro Felzenszwalb[译注1] [Felzenszwalb10]提出，是一种用于识别（原始的）图像中行人的算法，但是它可以很好地推广到许多目标，例如自行车和汽车。这种算法是基于Navneet Dalal and Bill Triggs [Dalal05]提出的知名的HOG-SVM所建立的。HOG-SVM[注11]使用一个滑动窗口，这个窗口类似于本章之前所见到的人脸识别级联分类器所使用的。该算法通过将窗口分割成较小的片并通过计算每片中的图像梯度方向的直方图来确定行人。这些直方图通常简称为“HOG”（方向梯度的直方图），直方图可以进行级联,形成一个特征向量，并将这个向量传递给SVM分类器。[注12]

注11：HOG-SVM实际上是一个生造的词，算法的发明者并没有对该算法进行命名，所以人们常用“Dalal和Triggs提出的算法”来指代这个算法。缩写HOG是算法作者生造的一个词，因为算法将HOG特征用于SVM中，所以HOG现在是HOG-SVM所使用特征的专门指代。这样来说，HOG-SVM可以算得上是这个算法的合理名称。

译注1：布朗大学教授，图像处理界的牛人，写过一套图像处理库。

注12：HOG-SVM是基于OpenCV以及少量的GPU库所构建的。如果想要了解更多信息，请在OpenCV.org的在线文档查阅cv::gpu::HOGDescriptor。现在也有基于CPU库所开发的HOG-SVM版本。

Felzenszwalb所提出的技术，称为基于组件的目标识别或者Latent SVM，它所使用的特征类似于HOG-SVM使用的HOG特征。但是，除非声明需要检测整个目标（如在HOG-SVM中所做的那样），它将分别表示目标的不同组件：可以预测的相对彼此运动的目标或作为整体的目标（例如，行人的胳膊、腿和头）。在实际情况中，组件相对于图像中心（被Felzenszwalb称为“根节点”）的位置是未知的；它们是模型的隐式变量，也是算法名称的由来。一旦根节点（由一个非常类似于HOG-SVM中使用的检测器）定位，并且每个组件的位置也被定位出来，就可以基于此建立一个关于目标的假设，并且计算各个组件相对根节点的似然度。

如在OpenCV中实现一样，Latent SVM方法可以使用与库一起提供的已经被训练过的检测器。

通过cv::dpm::DPMDetector的目标检测

要在OpenCV中使用Latent SVM，首先需要你实例化一个`cv::dpm::DPMDetector`对象（位于*opencv_contrib/dpm*模块中，因此需要在构建OpenCV的时候添加*opencv_contrib*）。[注13]实例化通过`create()`函数实现，这个函数我们在第21章中已经见过很多（通过派生我们永远不会真正看到的实现类`DPMDetectorImpl`完成）。通常情况下，在实例化目标时，需要提供级联的检测器的名称，如果不那么做，类名将由检测器文件名自动产生。

```
static cv::Ptr<cv::dpm::DPMDetector> create(
    std::vector<std::string> const &filenames,
    std::vector<std::string> const &classNames = std::vector<std::string>()
);
```

所有文件名都应该是包含在STL风格vector中的一组string，并且包含完全限定的文件名。类名，如果存在的话，也应该是STL风格vector中的一组string。

一旦加载模型，就可以继续应用模型在你自己的图像中查找目标：

```
void cv::dpm::DPMDetector::detect(
  cv::Mat                          &image,
  std::vector<ObjectDetection> &objects
);
```

参数`image`是输入图像。参数`objects`是一个由你提供的STL样式向量，`detect()`填充`cv::dpm::DPMDetector::ObjectDetection`的实例，其中将包含图像中发现的个别检测的所有相关信息。

注13：一般来说，我们在本书中没有包含*opencv_contrib*的工作。这个例外的原因是，不只是非常有用，还因为这个特定的实现已经用了很长时间，被认为非常稳定。

目标检测器将以cv::dpm::DPMDetector:: ObjectDetection对象的形式返回。这些对象有如下定义：

```
class cv::dpm::DPMDetector {

public:

  ...

  struct ObjectDetection {
    ObjectDetection();
    ObjectDetection( const cv::Rect& rect, float score, int classID = -1 );

    cv::Rect rect;
    float    score;
    int      classID;
  };

  ...

};
```

如你所看到的，这个结构非常简单。它包含的三个元素——rect，score和classID——分别说明了找到目标的窗口的大小和位置，分配给该识别的置信度，以及找到的特定类的整数类标识符。当你首次调用cv::dpm::DPMDetector::load()时，这些ID将对应于加载检测器的顺序。

cv::dpm::DPMDetector的其他方法

cv::dpm::DPMDetector对象还提供了一些有用的应用和访问器方法：

```
virtual bool isEmpty() const;

size_t getClassCount() const;

const std::vector<String>& getClassNames() const;
```

如果没有加载检测器，则isEmpty()方法返回ture。[注14]

调用load()时，getClassCount()返回你所加载的类的数量，同时getClassNames()方法返回一个STL样式向量，其中包含所有检测器的名称。当你自己没有提供这些名字时，这个方法非常有用，但是需要知道指定给cv::dpm::DPMDetector::load()的名字是什么（基于文件名）。

注14：在内部表现为只是调用包含检测器cv::dpm::DPMDetector的向量成员的std::vector<>::empty()方法。

获得cv::dpm::DPMDetector模型

现在，OpenCV还没有提供代码，让你训练自己的Latent SVM模型。目前，*opencv_contrib/modules/dpm/samples/data/*包含一些预训练的模型。如果不足以满足你的需求，或者你需要特别独特的功能，就必须训练你自己的检测器，Latent SVM的创始人将维护包含其MATLAB实现的网站。这一MATLAB实现包含训练你自己的检测器的必要成分（称为“pascal”）。一旦训练完成，产生的*.xml*文件就可以被OpenCV检测器加载。

Bag of Words算法与语义分类

词袋（Bag of Words）算法（BOW），也叫Bag of Keypoints，用于视觉分类或者识别场景中的目标内容。该算法的灵感来源于文档分类中使用的方法，通过具有类之间的强区分能力的特定关键词的存在，尝试将文档组织成语义分类。例如，为了试图分类医学文档，可以发现单词“肿瘤”的存在是该文件属于癌症文献类别的有效指标之一。在这种情况下，每个文件都没有被以任何真正有意义的方式读取，它只被视为单词的集合（“bag”），同时，重要单词的相对频率是唯一用于区别分类的东西。

在计算机视觉的实例中，可以定义具有强识别力的特征以确定图像，例如在分类汽车或者自行车中。BOW算法既解决了特征最突出的问题的识别，同时尝试识别新图像中的特征，并将其与数据库进行比较以对图像分类。

BOW算法的第一个阶段是了解随后将用于分类的特征。在这一阶段中，你要提供算法的一部分，称为*trainer*，其中包含你感兴趣的每个语义类别（例如，汽车、人、鸡或者任何你感兴趣的）。在OpenCV的实现中，你将首先从每个图像中提取自己最喜欢的关键点（参见第16章），并向BOW trainer提供生成的描述符列表。

这里，没有必要以任何方式分离图像，*trainer*的目的是找出哪些关键点看起来可以形成有意义的集群。这些集群是类似关键点描述符的集合，来自于你所提供的图像，并且在表示关键点的向量空间中靠近在一起。这些集群之后抽象成关键点中心，本质上是新的关键点，由BOW trainer构建，位于识别集群的中心。这些关键点中心可以类比单词在文献分类中起到的作用，因此通常称为“视觉单词”。[注15]整体上，被识别的视觉单词集合称为“词汇”。

注15：注意，在训练阶段使用每个图像中的每个描述符是不现实的。聚类的目的是将输入中的类似描述符聚合成更小、更易于管理的数字。原则上，聚集在一起的意思是相似的。可以借鉴文档分类的类比思想，将聚类单词“步行”、“走路”和“步行”等作为一个单词进行分类，而不是单独计算每个单词的出现。

一旦BOW trainer生成了词汇，就可以给它任何图像，并且该图像将转换成一个*presence vector*。presence vector是一个布尔向量，表示每个单词在词汇中的存在。注意，这是一个阶数非常高的向量，在实践中，它可以达到数百甚至数千阶。

在这一点上，通常的过程是训练一个分类算法。给定数据集中的所有图像，BOW算法可以将它们转换成presence vector，这些向量之后被用于训练产生正确类标签的算法。你可以使用任何能够实现多类别分类的分类算法，但是规范的选择是初始贝叶斯分类器或者支持向量机（SVM）。这两个我们都已经在第21章遇到。但是最关键的一点是，BOW算法从已知图像生成presence vector，可以用于训练分类器，并且从新图像中生成presence vector，用于分类器将该类别与该图像关联。

使用cv::BOWTrainer进行训练

将大量输入特征描述符转换成可控数量的视觉单词，其本质任务可以使用任何数量的集群技术来完成。抽象基类cv::BOWTrainer定义了可以为BOW算法执行此功能的任何目标的接口：

```
class cv::BOWTrainer {

public:
  BOWTrainer(){}
  virtual ~BOWTrainer(){}

  void                add( const Mat& descriptors );
  const vector<Mat>& getDescriptors()                    const;
  int                 descriptorsCount()                 const;

  virtual void        clear();
  virtual Mat         cluster()                          const = 0;
  virtual Mat         cluster( const Mat& descriptors ) const = 0;

  ...

};
```

cv::BOWTrainer::add()方法用于向trainer添加关键点描述符。它需要一个每行是一个单独描述符的数组。可以调用任意次数的cv::BOWTrainer::add()来累加描述符。在任何时候，你都可以找到有cv::BOWTrainer::descriptorsCount()中多少个描述符已经被添加，或者使用cv::BOWTrainer::getDescriptors()在一个大的数组中将其全部返回。

一旦在所有图像中加载了所有描述符，就可以调用cv::BOWTrainer::cluster()，它将实际计算视觉单词词汇。之后以数组的形式返回词汇，其中每一行被解释为一个单独的视觉单词。还有一个单参数版本的cv::BOWTrainer::cluster()，期望一个包含描述符的

单个数组，将立刻为该数组中的描述符集合计算视觉单词；它忽略了任何你可能已经用cv::BOWTrainer::add()存储的描述符。

最后，cv::BOWTrainer::clear()方法清空所有加载的描述符。

K-means和cv::BOWKMeansTrainer

在这一点上，cv::BOWTrainer接口只有一种实现方法cv::BOWKMeansTrainer。cv::BOWKMeansTrainer()中唯一的成员是非常新的构造函数，它的原型如下：

```
cv::BOWKMeansTrainer::BOWKMeansTrainer(
  int                     clusterCount,
  const cv::TermCriteria& termcrit    = cv::TermCriteria(),
  int                     attempts    = 3,
  int                     flags       = cv::KMEANS_PP_CENTERS
);
```

cv::BOWKMeansTrainer()实现方法的核心是使用*K*-means聚类算法。回想一下，*K*-means算法需要这样大量的点，尝试找到能充分解释数据的特写数量的聚类。在Trainer()构造函数中，这个聚类的数量被称为clusterCount，在这里将为产生视觉单词的数量，或者是词汇的大小。这个数量过小将导致非常差的分类结果。这个数量过大将导致接下来处理过程的操作非常缓慢，并导致分类变得不可能。[注16]

如果你不是*K*-means算法的专家，剩下的三个参数可以保留默认值。如果你选择修改它们，那么它们对构造函数的意义与它们对之前学习的第20章的*K*-means实现一样。

使用cv::BOWImgDescriptorExtractor分类

一旦使用trainer计算了聚类中心，就可以通过BOW算法尝试将图像的描述符转换为可以用于分类的presence vector。这里是cv::BOWImgDescriptorExtractor的类声明（重要部分），负责BOW算法部分的例程如下：

```
class cv::BOWImgDescriptorExtractor {

public:

  BOWImgDescriptorExtractor(
    const cv::Ptr< cv::DescriptorExtractor >& dextractor,
    const cv::Ptr< cv::DescriptorMatcher >&   dmatcher
```

注16：有一篇论文[Csurka04]可以作为这个算法的规范参考，作者用1776个图像表示7个类。从每个类中，他们从该类中的图像中提取了5 000个关键点，共提供了35 000个关键点。他们发现，将集群数（每个类）设置为1 000到2 500之间的任何地方都会产生类似的结果，因此他们选择使用1 000来发布其结果。可以从这个得出一个公正的结论：聚类数量相当于类中点数量的百分之几，这个数是合理的。

```
);
virtual ~BOWImgDescriptorExtractor() {;}

void setVocabulary( const cv::Mat& vocabulary );
const cv::Mat& getVocabulary() const;
void compute(
  const cv::Mat&           image,
  vector< cv::KeyPoint >& keypoints,
  cv::Mat&                 imgDescriptor,
  vector< vector<int> >*   pointIdxsOfClusters = 0,
  cv::Mat*                 descriptors = 0
);
  int descriptorSize() const;
  int descriptorType() const;

  ...

};
```

你首先会注意到一点：当我们在构建一个cv::BOWImgDescriptorExtractor对象时，需要为它提供一个描述符提取器和一个描述符匹配器。你所提供的提取器应该与你在使用trainer计算聚类中心时用于提取描述符的提取器相同。匹配器可以为任何形式。例如：

```
cv::Ptr< cv::DescriptorExtractor >       descExtractor;
cv::Ptr< cv::DescriptorMatcher >         descMatcher;
cv::Ptr< cv::BOWImgDescriptorExtractor > bowExtractor;

descExtractor = cv::DescriptorExtractor::create( "SURF" );
descMatcher   = cv::DescriptorMatcher::create( "BruteForce" );
bowExtractor  = new cv::BOWImgDescriptorExtractor( descExtractor, descMatcher );
```

一旦构建了BOW描述符提取器，你将需要使用trainer给出你构建的词汇。输入到cv::BOWImgDescriptorExtractor::setVocabulary()的词汇完全是你从cv::BOWTrainer::cluster()得到的输出。[注17]

最后，一旦给了描述符提取器词汇，你就可以提供给cv::BOWImgDescriptorExtractor::compute()一幅图像，它将计算imageDescriptor，它也是presence vector。presence vector将具有一行，和与词汇中的行一样多的列。每个元素将是与特定聚类中心匹配的元素数量。

可选参数pointsIdxsOfClusters和描述符告诉你有关生成presence vector的实际匹配。pointsIdxsOfClusters是一个向量的向量，第一个索引与聚类相关；因此pointsIdxsOfClusters[i]指第*i*个聚类中心（词汇的第*i*个条目），并且本身也是一个向

注17：说提更深入一些，这里发生的一切就是将词汇表中的所有描述符都添加到你在创建时给出提取器的匹配器的列表中。

量。条目pointIdxsOfClusters[i]列出图像中与该聚类中心匹配的描述符的索引。这些索引表示描述符的行数。描述符数组（如果计算）是通过你创建时提供给BOW提取器的特征提取器，从图像中提取的原始（在与聚类中心结合之前）描述符列表。概括起来，如果pointsIdxsOfClusters[i][j] = q，意味着descriptors.row(i)是第*j*个描述符，并且与vocabulary.row(q)相匹配。

使用支持向量机进行整合

为了实际实现整个BOW算法，你将需要额外的一步：获取presence vector并使用它们训练一个多类别分类器。当然，有多种方法实现多类别分类器，在这里我们看到的是支持向量机。因为SVM本身不支持多类别分类，所以我们使用一对多的方法。[注18]在这个方法中，如果有N_c类，则会训练N_c个不同的分类器，每个分类器解决查询向量是否在*i*类的问题（即在其他任何$j \neq i$类中）。

我们使用相同的presence vector集合训练每一个SVM，但是对于响应使用不同的符号。以下代码片段来自OpenCV中附带的示例，来自*bagofwords_classification.cpp*示例代码；它构建了将被传递给SVM训练例程的trainData和responses数组。每一类运行一次：

```
cv::Mat trainData( (int)images.size(), bowExtractor->getVocabulary().rows, CV_32FC1
);
cv::Mat responses( (int)images.size(), 1, CV_32SC1 );

// Transfer bag of words vectors and responses across to the training data matrices
//
for( size_t imageIdx = 0; imageIdx < images.size(); imageIdx++ ) {

  // Transfer image descriptor (bag of words vector) to training data matrix
  //
  cv::Mat submat = trainData.row( (int)imageIdx );
  if( bowImageDescriptors[imageIdx].cols != bowExtractor->descriptorSize() ) {
    cout << "Error: computed bow image descriptor size "
         << bowImageDescriptors[imageIdx].cols
         << " differs from vocabulary size"
         << bowExtractor->getVocabulary().cols
         << endl;
    exit(-1);
  }
  bowImageDescriptors[imageIdx].copyTo( submat );

  // Set response value
  //
  responses.at<int>((int)imageIdx) = objectPresent[imageIdx] ? 1 : -1;
};
```

注18：你可能会记得，我们之前在详细了解支持向量机时，简单提到了一对多。就目前情况下，我们可以使用OpenCV提供的一对一的方法，但是速度相对较慢，所以我们使用一对多的这个例子。

但是这不是完整的内容。在一对多方法中存在着一种可能性：对于每一类，一幅图像将被发现“不在这一类中”，或者对于超过一类“在这一类中”。但是请记住，SVM通过在高维内核空间中建立一个线性决策边界来工作。因为在该空间中是线性边界，一旦将查询点映射到内核空间中，我们可以相当容易地计算出点位于哪一条决策边界上。我们同样可以计算出该点到边界的距离。一般把这种距离作为置信度，这也是解决非关联和多重关联问题的关键。

通常，在多重关联的情况中，与最大margin的关联被认为是正确的关联。在非关联的情况，如果事先知道每一幅图像实际上是其中一个已知类的一部分，则可以选择最小负margin的图像。如果可能是未知类的图像，那么或许可以设置一个负margin（可能但不一定为0）的阈值，如果所有分类器的返回都比该值更差，则图像被分配为“未知”类。

小结

在本章中，我们学习了OpenCV库提供的几种用于判断目标是否处于图像上的方法。在一些情况下，这些方法根据构成图像中的目标的像素提供了目标定位的一些概念。基于树的方法以及Latent SVM，由于使用滑动窗口，都具有这种属性。在另一方面，词袋方法不具有这一属性。但是，词袋方法的一个优点是，它可以用于更抽象的查询，例如场景分类。

所有这些方法都使用了我们在本书之前章节所学过的计算机视觉技术，并结合了最近几个章节中的机器学习方法。如果我们给这些检测器中添加跟踪和移动建模的方法，我们就拥有了所有必要的方法来处理非常实际和现代化的计算机视觉方法，例如定位和在视频序列中跟踪目标，同时还包括许多其他的方法。

练习

1. 如果训练数据的数量有限，哪个方法将更有可能产生更好的测试数据，决策树还是SVM分类器？为什么？
2. 建立并运行Haar分类器来识别你在网络摄像机中的人脸。
   a. 它可以用于多大尺度的变化？
   b. 模糊情况下怎么样？
   c. 头部倾斜多少角度以内它可以保持工作？
   d. 下巴向下和向上到什么角度时，它将不再工作？
   e. 头部旋转（左右移动）到什么角度时它将不再工作？

f. 探索3D头部姿势的容错度。对结果进行说明。

3. 重复练习2，但是这次试用LBP级联器。结果将变好还是变差？同时说明为什么会变好或变差。

4. 使用蓝色或者绿色屏幕来收集竖起大拇指的手势（静态姿势）。收集其他手势和任意背景的例子。收集数百幅图像，然后完成以下练习。

   a. 训练Haar分类器来识别竖起大拇指的手势。实时测试分类器并计算其混淆矩阵。

   b. 基于竖起大拇指的数据训练并测试LBP级联分类器，计算其混淆矩阵。

   c. 基于竖起大拇指的数据训练并测试软级联分类器，计算其混淆矩阵。

   d. 比较和对比这三种算法的训练时间、运行时间，以及识别结果。

5. 使用练习4中的数据列和三个你自己创建的特征。

   a. 使用任意树来识别竖起大拇指的数据。

   b. 添加一个或多个特征来改善结果。

   c. 使用数据分析、可变重要性、归一化和交叉验证来改善结果。

   d. 使用你所掌握的任意树的知识来改善结果。

6. 使用练习4中的数据列训练cv::dmp::DPMDetector来识别竖起大拇指的手势。请创建结果的混淆矩阵。

7. 在附近地区10个不同地方收集一些街区场景图像（每个场景一幅图像）。请训练BOW分类器来识别这10个不同的地方。通过附近街景对识别进行测试。

8. 除了BOW分类器方法之外，还可以使用潜在SVM来分类处于treetview图像的位置。在使用地图定位时，这些算法的相对优缺点是什么？

   提示：拓扑几何与几何之间是存在差异的。第一个与图像相关，第二个与三角测量相关。

9. 级联分类器产生一个二类分类器（如人脸或非人脸）。如果有10个类，描述一种方法，使你可以使用级联分类器方法训练并识别所有10个类。这种方法有哪些优点？

第23章

OpenCV的未来

过去与未来

OpenCV于1999年8月计算机视觉与模式识别（CVPR）会议上正式发布（因此到本书中文版出版的时候已经19岁了）。Gary Bradski（加里·布拉德斯基）博士在英特尔创建了OpenCV，旨在加快计算机视觉在社会中实际应用的研究和使用。很少有什么项目能一直按照它们最初的计划发展，但是OpenCV恰恰就是这其中的一个。在写本书时，OpenCV已经包含近3000个函数，每月下载量超过20万次，总下载量达1400万次，日常用于数百万部手机，识别条形码，拼接全景图像，并通过计算机摄影技术改善图像。OpenCV同样被用于机器人系统——采摘莴苣，识别传送带上的物品，帮助自动驾驶汽车进行视觉感知，四旋翼飞行，在虚拟和增强现实系统中进行跟踪和绘图，协助配送中心卸载卡车和托盘，并被嵌入到机器人操作系统（ROS）中。它还被用于改善矿山安全，防止游泳池溺水，处理Google地图和街景图像以及实施Google X机器人技术中。

自从本书上一版开始。OpenCV已经将其架构从C发展到与STL和Boost兼容的现代化、模块化的C++架构。它开始使用现代软件开发标准，在Git（*http://github.com/opencv/opencv*），持续的自动构建机器人，Google单元测试，综合文档和教程上进行分布式的开发。OpenCV一开始就是跨越Windows，Linux和Mac OS X的跨平台软件库。它将一如既往地积极支持这些桌面操作系统，还将涵盖Android和iOS版本的移动设备开发。它为英特尔架构，ARM、GPU、NVidia GPU和Movidius等各种芯片提供了优化版本，同时也可以与Xylinx Zync FPGA配合使用。除了高效的C++源代码，它还有广泛的Python接口（与NumPy兼容）、Java接口和MATLAB接口。

OpenCV还添加了一个由用户维护的新的独立部分。在该函数库中，所有例程都是独立

的，遵循OpenCV样式和文档，并通过了Buildbot测试。使用*opencv_contrib*，OpenCV可以与计算机视觉中最新的算法和应用保持同步，相关目录的内容，可以参阅附录B。

OpenCV 3.x

OpenCV从一个单纯的C语言库开始，1.0版本主要集中在构建有用的算法内容上。OpenCV 2.0的主要重点是让函数库达到现代的C++开发标准，包括转移到Git，使用Google风格的单元测试，兼容STD，当然还具备C++接口。所有新的开发都是用C++编写的，同时旧的C函数也被C++兼容。在这个过程中，还添加了Python，Java和MATLAB的完整接口。

OpenCV 3.0专注于模块化，它完全用原生C++编写，因此只需要维护一个代码库即可。计算机视觉的逐渐成功导致了一个问题，即在一个单片机代码库中存在太多可能有用的算法需要维护。OpenCV 3.x通过保持强大的支持内核来解决这个问题，并将其他一切变成易于创建和更容易维护的小型独立模块，可以根据需要进行混合和匹配。越来越多的计算机视觉学生和研究小组正在开发基于OpenCV数据结构的新算法。OpenCV 3.x可以轻松生成一个完整的文档、单元测试和示例代码，并可以轻松连接到OpenCV。

OpenCV 3.x的独立模块还将通过允许更小，更集中的计算机视觉存储器来帮助云、嵌入式和移动应用。OpenCV的任务之一是促进越来越多的计算机视觉技术在社会中的应用；嵌入式视觉设备将有助于在机器人、移动、安全、安全、检测、娱乐、教育和自动化中扩展视觉感知的使用。对于这样的应用，内存使用是一个要考虑的关键。另一方面，云计算也有内存限制，因为算法规模可以跨越大量运行多种作业的机器，内存使用成为关键瓶颈。

我们希望，通过轻松组合独立大的、乃至个人架构的模块，OpenCV 3.x将不仅可以应用于上述领域，还可以在*opencv_contrib*中扶植一些可能看起来像“视觉应用商店”的内容。直接插入OpenCV的精确定义模块的集合将使视觉启用的应用程序能够更广泛和更有创意地使用。外部模块可能是开放的、封闭的、免费的或商业的，所有这些都旨在允许对视觉了解很少的开发人员能将计算机视觉技术应用到程序中。

我们上一次预测怎么样？

上一版中，我们对OpenCV的未来做了一些预测。现在那些预测怎么样了呢？ 我们当时预测OpenCV将支持机器人和3D视觉，这显然实现了。作者之一Gary（加里）创建了一个机器人公司Industrial Perception，该公司使用OpenCV和3D视觉例程使机器人能够在配送中心处理包装盒。谷歌在2013年收购了该公司。与此同时，另一位作者Adrian（安

德里安）在Applied Minds公司工作时，开展了许多涉及OpenCV的行业、政府和军用机器人合同项目。

标定部分预计会进行拓展，包括被动和主动感知。现在的实际情况是，OpenCV现在包括ArUco增强现实标记，棋盘与ArUco模式的组合，所以你不再需要看到整个标定板，多个相机可以以相同的标定模式看到标定板上的不同部分（见附录C）。所有这些例程现在都有，以解决更具挑战性的标定和多摄像机姿态问题。

我们预测了新的3D目标识别和姿态识别功能，并将这些功能集成，人为设计的linemod特征到深层网络3D目标识别和姿态识别。事实上，*opencv_contrib*本身就被预想为一个模块化的存储库，以方便用户贡献代码。

上一版中预测的大多数应用，从更好的立体视觉算法到稠密光流，都已经成为现实。那时，我们说2D特征将被一个引擎扩展和支持，所有这些都在`feature2D()`中实现了，它覆盖了大部分手工设计的2D点检测器和描述符。基于Google数据结构的函数改进也在进行之中。我们还说，将添加对近似最近邻搜索算法技术的更好的支持，并且将FLANN（快速近似最近邻搜索算法库）合并入OpenCV。我们早已在上一本书中概述的计算机视觉会议上开展了开发人员的工作。最后，终于完成了更好的开发文档（*http://docs.opencv.org*）。

那我们哪里预测错了呢？ 我们还没有实现更高维或多光谱摄像机更一般的相机接口。虽然已经支持SLAM（同时定位和建图），但还不是一个鲁棒的完整实现。贝叶斯网络没有实现，因为深层网络超过了它们。我们还没有为艺术家们实现任何特别的工作，但他们却在继续探索OpenCV在艺术创作上的应用。

未来应用

这本书已经提及了OpenCV过去与现在的详细状况。接下来是一些未来的方向。

深度学习

OpenCV已经可以读取和运行Caffe，Torch和Theano等网络。此代码位于*https://github.com/opencv/opencv_contrib/tree/master/modules/cnn_3dobj*中。预计你可以看到OpenCV在嵌入式系统和智能相机集成一个完整的深入学习模块进行运行和训练，并扩展到外部代码库*tiny_dnn*。

移动设备

“可计算”摄像机的发展是惊人的。因此，OpenCV的一个明显方向是实现对移动设备越来越多的支持。这个支持包括算法以及移动硬件和移动操作系统。OpenCV已经实现了iOS和Android端口的支持，我们还希望能够支持更多对静态内存占用需求更小的设备。

智能眼镜

用数据和对象覆盖视觉场景的增强现实眼镜将是一个越来越受到关注的领域。跟踪用户头部的3D姿态可以有助于在一个房间内实现虚拟现实。我们已经将ArUco AR标签扩展到了ChArUco（使用ArUco），从而给出更加准确的姿态。一些OpenCV贡献者致力于为Google Cardboard添加SLAM支持。

嵌入式应用

嵌入式应用程序也越来越重要，并将成为一个全新的设备领域。鉴于这个趋势，Xilinx已经在它的Zync架构中添加了OpenCV接口。预期我们可以看到各种各样的装置，从玩具到安全装置，汽车应用，工业使用以及陆地，水下和空中无人机。OpenCV有望可以帮助实现这些发展。

3D视觉

许多公司正在开发深度传感器，并越来越多地涉足到移动领域之中。OpenCV拥有正在不断发展的稠密深度支持例程，从快速计算法线、平面查找和提取深度特征进行改进。

光场摄像机

这个领域的源头可以追溯到1910年，但在20世纪90年代才得以开始迅速发展。我们预计通过使用不同的镜头配置，便宜的摄像机和嵌入式处理器允许镜头阵列捕获多点宽视图、孔径和视场，这一领域会变得越来越流行。我们期望看到对类似摄像机的支持，因为它们已经出现并正在变得越来越便宜。

机器人

所有上述功能都会直接使机器人受益。新的硬件、便宜的摄像头和灵活的机器人手臂，加上更好的规划和控制算法，标志着一个全新的由传感器引导机器人技术这一行业的开始。几个OpenCV的关键贡献者正在研究机器人，可以预期这一领域将有持续的发展。

云

随着时间的推移，我们期望看到对这方面的支持，使其更容易在与运行相同处理堆栈的服务器交互操作的嵌入式摄像机阵列上进行工作，并能够与OpenCV、深层神经网络、图形、优化和并行功能库紧密集成。我们还会做一些工作，让使用C++或Python的亚马逊和Google服务器等商业供应商可以无缝工作。

在线教育

我们想提供使用OpenCV解决计算机视觉问题的在线课程。我们希望扩大OpenCV在会议和研讨会上的知名度，甚至有可能提供我们自己的“入门须知”会议。

目前GSoC的进展

在过去的几年中，谷歌已经通过谷歌编程之夏（GSoC）计划来支持实习生在OpenCV上工作。可以在维基网页查看这些工作。也可以在以下网址查看涵盖此新功能的视频：

- 2015: *https://youtu.be/OUbUFn71S4s*
- 2014: *https://youtu.be/3f76HCHJJRA*
- 2013: *https://youtu.be/_TTtN4frMEA*

2015年，有15名实习生获得支持。对于实习生（其中许多人现场表现突出）和OpenCV，这种支持是非常宝贵的。2015年该计划的主题，几乎所有都请求可以接受放入OpenCV主干，如下所示：

全方位摄像机标定和立体3D重构

*opencv_contrib/ccalib*模块（Baisheng Lai, Bo Li）［赖百胜，李波］

运动结构

*opencv_contrib/dpm*模块（Jiaolong Xu, Bence Magyar）

使用核相关滤波器进行实时多目标跟踪

*opencv_contrib/tracking*模块（Laksono Kurnianggoro, Fernando J. Iglesias Garcia）

改进和扩展的场景文本检测

*opencv_contrib/text*模块（Lluis Gomez, Vadim Pisarevsky）

立体对应改进

*opencv_contrib/stereo*模块(Mircea Paul Muresan, Sergei Nosov)

结构光系统标定

*opencv_contrib/structured_light*模块(Roberta Ravanelli, Delia Passalacqua, Stefano Fabri, Claudia Rapuano)

棋盘+ ArUco用于摄像机标定

*opencv_contrib/aruco*模块（Sergio Garrido, Prasanna Krishnasamy, Gary Bradski）

深层神经网络框架的通用接口实现

*opencv_contrib/dnn*模块（Vitaliy Lyudvichenko, Anatoly Baksheev） [未来将被tiny-dnn取代]

边缘感知滤波的最新进展，改进的SGBM立体算法

opencv/calib3d and *opencv_contrib/ximgproc*模块（Alexander Bokov, Maksim Shabunin）

改进的ICF检测器，Waldboost实现

*opencv_contrib/xobjdetect*模块（Vlad Shakhuro, Alexander Bovyrin）

多目标TLD跟踪

*opencv_contrib/tracking*模块（Vladimir Tyan, Antonella Cascitelli）

使用CNN的3D姿态估计

*opencv_contrib/cnn_3dobj*模块（王一达, Manuele Tamburrano, Stefano Fabri)

截至本书的交稿，GSoC 2016正在开发以下13种新算法：

- 在OpenCV中加入*tiny-dnn*深度学习训练和测试函数（Edgar Riba, 王一达, Stefano Fabri, Manuele Tamburrano, Taiga Nomi, Gary Bradski）
- 增强现有*dnn*模块来读取和运行Caffe模型（Vludv, Anatoly Baksheev）
- 更好的视觉跟踪，GOTURN跟踪器（Tyan Vladimir, Antonella Cascitelli）
- 精确的动态结构光（Ambroise Moreau, Delia Passalacqua）
- 添加非常快速、稠密的光流（Alexander Bokov, Maksim Shabunin）
- 扩展深度word-spottin卷积神经网络文本识别模块（Anguelos, Lluis Gomez）
- 稠密光流算法的改进（VladX, Ethan Rublee）
- OpenCV教程中的多语言支持：Python，C++和Java（Carucho, Vincent Rabaud）
- 新图像拼接途径（Jiri Horner, 李波）
- 为OpenCV添加更好的文档存储（Myls, Vadim Piarevsky）

社区贡献

与此同时，OpenCV社区也变得格外活跃。在2015 GSoC期间，社区贡献了以下成果：

- 绘图模块（Nuno Moutinho）
- Ni-black阈值算法：ximgproc (Samyak Datta)
- 使用线性光谱聚类的超像素分割，SLIC超像素：*ximgproc*（Balint Cristian）
- HDF（HDF5）支持模块（Balint Cristian）
- 外部RGB摄像机配准的深度：*rgbd*（Pat O'Keefe）
- 点云的计算法则：*rgbd*（Félix Martel-Denis）
- 模糊图像处理（Pavel Vlasanek）

- 滚动快门引导滤波器：*ximgproc*（Zhou Chao）
- 三倍速度的SimpleFlow：*optflow*（Francisco Facioni）
- CVPR 2015文献“DNNs Are Easily Fooled”的代码和文档（Anh Nguyen）
- 高效的图形图像分割算法：ximgproc（Maximilien Cuony）
- 稀疏到稠密的光流：*optflow*（Sergey Bokov）
- 无迹卡尔曼滤波器（UKF）和增强的UKF跟踪（Svetlana Filicheva）
- 快速霍夫变换：*ximgproc*，*xolodilnik*
- 改进性能的haartraining（Teng Cao）
- 与Python 3兼容的Python示例代码：*bastelflp*

我们期待谷歌和社区可以继续开展这项有益的事业！

OpenCV.org

在本书上一版与此版本出版之间，OpenCV注册成为加利福尼亚非营利组织，旨在推动计算机视觉，促进计算机视觉教育，并提供OpenCV作为自由开放的基础设施，特别是推动视觉算法。迄今为止，该基金会得到了英特尔、谷歌、柳树车库（Willow Garage）和英伟达（NVidia）的支持。此外，DARPA（通过英特尔）为CVPR（计算机视觉与模式识别）2015年的“大众选择最佳论文”奖提供赞助，英特尔已经赞助了2016年的比赛。2015年的结果可以在网上看到。获奖作品产生了*.../opencv_contrib*目录中的几个新算法。OpenCV的预置代码可以从用户网站下载，而原始代码可以从开发者网站获取；有关核心库的更多信息，可以参见*https://github.com/opencv/opencv*，用户提供的模块，可以参见*https://github.com/opencv/opencv_contrib*。OpenCV的维基地址是*https://github.com/opencv/opencv/wiki*。同时还有一个Facebook页面。

随着这本书的写作即将结束，创始人Gary Bradski（加里·布拉德斯基）正在将OpenCV.org变成联邦非营利组织501(c)(3)公司。以前，OpenCV没有带薪工作人员（不含谷歌提供的训练营导师津贴），没有办公室，没有设备，并且一直在同一年内付清所有开销东西。现在，OpenCV.org正在努力转向成为强大的全功能非营利组织。这将涉及到一些专门的董事会成员（未定），筹集资金来支持一些带薪全职员工，发展教育素材和比赛，举办年度会议，重视新的、有用的解决方案的愿景，提供深入的培训教程，赞助或至少支持机器人联盟发展更强大的感知和自主能力，为高中学习和使用计算机视觉提供支持和教育，为艺术界提供计算机视觉的支持和培训等等。

我们也希望能和刘瑞祯教授牵头创办的OpenCV in China（*http://www.opencv.org.cn*）建

立更多的合作。他是中科院智谷工业研究院院长，该学院由中国科学院和复旦大学赞助，是一个旨在成为一个专注于人工智能、自动化、智能设备控制和模式识别的独立研究机构。随着时间的推移，我们希望能与世界各地其他类似组织增强联系。

如果OpenCV.org有足够的资金，OpenCV可能会提供全功能的电话和网络支持，开发视觉和机器学习课件（可能包括与制造商合作提供兼容的开发工具包），并对视觉开发人员进行能力认证，认可他们有在计算机视觉和深度学习感知方面构建应用程序的能力。我们也可以为合作伙伴提供的其他摄像机功能开发认证计划，让“OpenCV认证”可以成为值得信赖的品牌。我们期待着大幅扩大OpenCV的覆盖面和范围！

一些关于AI的猜测

我们正处于人工智能（AI）发展的转折点上。就在我们编写本书时，Google Deep Mind团队的AlphaGo在非常困难的“空间策略”游戏——围棋中击败了世界冠军李世石。 通过AI，机器人学习驾驶、飞行、行走以及操纵物体。 同时，AI技术正在通过智能设备和互联网，一刻不停地进行着语音、声音、音乐和图像的识别任务。自1848年以来，硅谷已经出现过许多次“淘金热”，因为这里有真正的财富。在这个新的AI淘金热中，鹿死谁手仍有待观察，但很明显，世界将不会永远一成不变。 在推动主动感知（拥有自主性的）设备这一历史进程中，OpenCV正凭借其加速感知过程的能力发挥着重要的作用。

很明显，深度神经网络已经基本上解决了前馈模式识别问题（或者说在这个问题上，深度神经网络是一种效果非常好的函数逼近器），但是这样的网络仍然称不上具备感知或者说“生命体”的能力。首先，存在一个体验本身的问题。就好比作为人类，面对一种颜色，我们不是单单像机器那样看到然后说出来，而是会主观的体验它。这种主观的体验是如何产生可感受的特性的呢？其次，机器似乎也还没有真正地具备自主创造的能力。它们可以在明确的领域内生成新的东西，但它们不会发明新的领域，也不会积极地推动实验和开环的发现。

我们猜测这中间缺少的可能就是那种“具体化的实践”。人类，以及很多机器人也是，都有一个他们自己在这个世界活动的模型，或者说叫“自我”。这种模型可以独立地模拟行动的规划，但这种仿真模型本身却需要通过传感器和它周围的世界进行耦合。通过这种耦合模型，“具体化的实践”才能赋予这个世界因果意义（比如决定在哪里走路，躲避危险，观察某个计划的后果等等），反过来，这种因果意义又将一种和其耦合模型相关的意义感赋予了这个“具体化的实践”。

我们相信，这样一个世界耦合模式本身就可以让人工智能产生隐喻[Lakoff08]，这些隐喻被用来概括以后的经验。例如，在年轻的时候，人类曾把东西放在容器里，再把东西从容器中取出来。在之后的生活中，这种体现的经验告诉了我们，在花园里说的是什

么意思；换句话说，早期使用容器的经验被用来概括在花园里的意思。自我模式的因果体验，耦合到世界，允许一个实体将意义附加在事物上。这样的意义稳定了观念，因为类别不仅仅是直来直去的，而是对我们在世界范围内的模拟自我模型有着因果和时间稳定性的后果。这是完全的猜测，但是这些夸张或主观的经验可能来自于模拟我们的世界模型如何影响我们的模拟自我模型；也就是说，我们体验到我们的模型对世界的模拟反应，而不是世界本身。

斯坦利（Stanley），获得2005年DARPA挑战大赛机器人比赛跨越沙漠的200万美元奖金的机器人，使用了许多传感器，如GPS、加速度计、陀螺仪、激光测距仪和视觉来感知世界，并将这些感知结果融合在一起高效的计算“鸟瞰”世界模型。该模型包括反映了地形的一般角度的倾斜平面，然后通过传感器数据标记处可行驶、不可行驶和未知区域。在这个世界模型中，斯坦利（Stanley）在接下来的几个GPS导航点的大致方向上对自我物理模型进行了很多次模拟驾驶。得到的路径被认为是不会使机器人摔倒的最有效的路径。斯坦利（Stanley）的大脑虽然足以赢得DARPA挑战大赛，但依然有无法完成的方面，比如：它不能很好地表示爱情、政治、天体物理学或莎士比亚。如果斯坦利可以问我们莎士比亚戏剧是什么意思，那么我们最好告诉它这是一个在复杂地图上的可驾驶和未知区域之间的边界。斯坦利自我模型及其与世界的互动远远不足以了解世界上的大多数事物。似乎同样也很明显的是，我们这些主要生活在低洼温带流域的类猿生物，极限能力同样也不足以让我们去了解浩瀚的宇宙。从这种意义上来说，我们都是斯坦利。

人类很容易理解对食物的需要（这是我们人这个模型的之一本质），但是让我们发现如何创建一个更智能的人工智能或理解什么是感受却非常困难。另一个例子，如果我们养一只小猫，整天让它听莎士比亚，直到它长大，也不曾指望我们养的猫理解什么是十四行诗。如果我们要向猫解释十四行诗意味着什么，我们最好使用猫的自然模式来进行比喻，比如“莎士比亚的十四行诗就像一只小猫，不可避免地会在复杂的地方迷路”。猫可能认为“现在我明白了”，但我们要理解的是，对于它来说不能理解的事是没有意义的！同样，我们人类也受到同样的限制。也许我们可以建立更强大的机器，其中感受的问题很简单。但是当我们要求机器向我们解释的时候，可能会更混乱，最后说：“感受就像一只小猫，不可避免地会在复杂的地方迷路。”

斯坦利（Stanley）这个机器人，感知的本质完全取决于其模型。它无法感知世界，相反，其摄像机和传感器可以在信号中传播机器人的因果模型。但是，这种模式在机器人车的导航需求方面就只有现实世界。另一个例子是：在笔记本电脑中，你可能会看到一个GUI，并得出一个结论，比如说由于你在屏幕上看到一个回收站，所以计算机内就有回收站。一个更聪明的物理学家可能会很密切关注电脑屏幕，并说：“一切都是由量子（像素）组成的！”但事实上，GUI只是线性冯·诺依曼机器实现的一个因果模型。事实

是被拖进回收站的东西被抹去了——模型的因果后果是真实的。再次，我们人类在我们可以了解我们自己的宇宙中受到同样的限制，因为我们继承了一个主要直接针对我们身体和社会经验的因果模型。但我们的机器可能看得更远。

现在有很多有关AI危险性的争论。人们混淆了这个隐喻。他们认为“AI”（智能），驱动着更大的系统的行为。但是反观我们自己。我们人类的“编程语言”不是我们的智能，而是我们的情感，并且由它驱动我们的情绪！ 斯坦利的目标是以正确的顺序尽快安全通过GPS导航点。它发现无线跟踪GPS路点是有吸引力的，所以就使用它的智能来做它。我们的程序不是“想”，而是情绪。情绪指导我们“做什么”；智力指导我们“如何做”。对于我们未来的机器来说，设计框架也是如此。设计好动机，机器就会去追求这些。

啊，但读者可能会担心，这些机器可能会改变传递给下一代机器的目标？首先，让更多的机器智能能够理解并创造出比我们创造第一代机器更强大的机器的思维，是非常不容易的。问题会向上一代转移。智能机器也将面临在自己下一代AI的演变中与我们在第一次演变时同样的危险，并且它们将倾向于据此计划目标和情绪。最终，AI将拥有自己的目标，这些目标从我们的目标演化而来。AI的危险不在于它们的目标会多么邪恶，而是来自于它们目标中非人的属性，对我们来说，这看起来可能是冷漠甚至敌对性的。这些才是让H. P. Lovecraft（洛夫克拉夫特）的小说中的外星人怪物如此有趣和可怕的——它们没有那么恶毒，即使被完全不同的动机和完全冷漠的命运驱动。例如，我们很少考虑蚂蚁的生活，因此有时会使它们受到伤害。

然而，我们在此郑重声明，作者不怕AI，而是把它视为解决许多人类烦恼问题的必由之路，比如为所有人提供可靠的医疗保健，解决饥饿，治愈疾病，提供和维护新的能源发电和储存技术，同时保护环境，并帮助我们运行更复杂的世界。对于AI，我们看到的不是威胁，而是能够实现我们自己的目的。我们不希望自主智能生活的火花与我们的世界一起消亡，而是宁愿看到智能可以在空间和时间上向外扩展。这在某种程度上可能是（或者可以被选择成为[注1]）人类的最终目的，这也是OpenCV的间接目的！

注1： 这样的想法有一种精神或宗教意义，但它们却是“旧时代”宗教有趣的翻转。过去，人们认为是上帝选择了一个人和一个目的。然而在现在这个新的意义上，人们选择一个目的，这可能会导致一个类似于神的AI（例如，谷歌的算法和服务器对我们来说几乎是无所不知的知识）。在这个意义上，我们从一个被选择的人，从逐渐消失的现在回到光荣过去的人，转换成一个选择从逐渐消失的现在一个前往光荣未来的人。

结语

我们在本书中介绍了很多理论和实践，并且描述了下一步的一些计划。当然，当我们开发软件时，硬件也在改变。摄像机现在越来越便宜，性能越来越强，从手机到交通信号灯，从工厂到家庭监控，已经十分繁荣。一批制造商正在努力开发手机投影机——这非常适合机器人，因为大多数手机都是重量轻、低能耗的设备，其电路已经包括了嵌入式摄像机。这为近距离便携式结构光开辟了道路，从而可以获得更加精确详细的深度图，以及开发用于机器人控制和3D物体扫描的光场摄像机，这正是我们需要的。

两位作者都参与搭建了斯坦福机器人赛车斯坦利（Stanley）的视觉系统，它赢得了2005 DAPRA挑战大赛。在这一过程中，与激光测距扫描仪相结合的视觉系统在七小时沙漠公路赛中表现的十分完美[Dahlkamp06]。对于我们来说，这让人理解了视觉与其他感知系统相结合的力量：通过将视觉与其他形式的感知相结合，将先前未解决的道路感知问题转化为可解决的工程挑战。我们希望通过这本书使视觉技术更容易使用和更易于使用——其他人也可以为自己的问题增加愿景——解决工具包从而找到解决重要问题的新途径。也就是说，通过商业化的摄像机硬件，便宜的嵌入式处理器和OpenCV，人们可以开始解决实际问题，例如使用立体视觉作为汽车备用安全系统（或者在总体上对汽车进行改进），监视所有轨道上的人和车辆，实施游泳安全措施，建立新的游戏控制策略，开发新的安全系统等。请务必关注*tiny-dnn*，这是一个功能齐全的深度网络库，专注于*opencv_contrib*中的嵌入式计算。最后愿大家：自己动手实现！

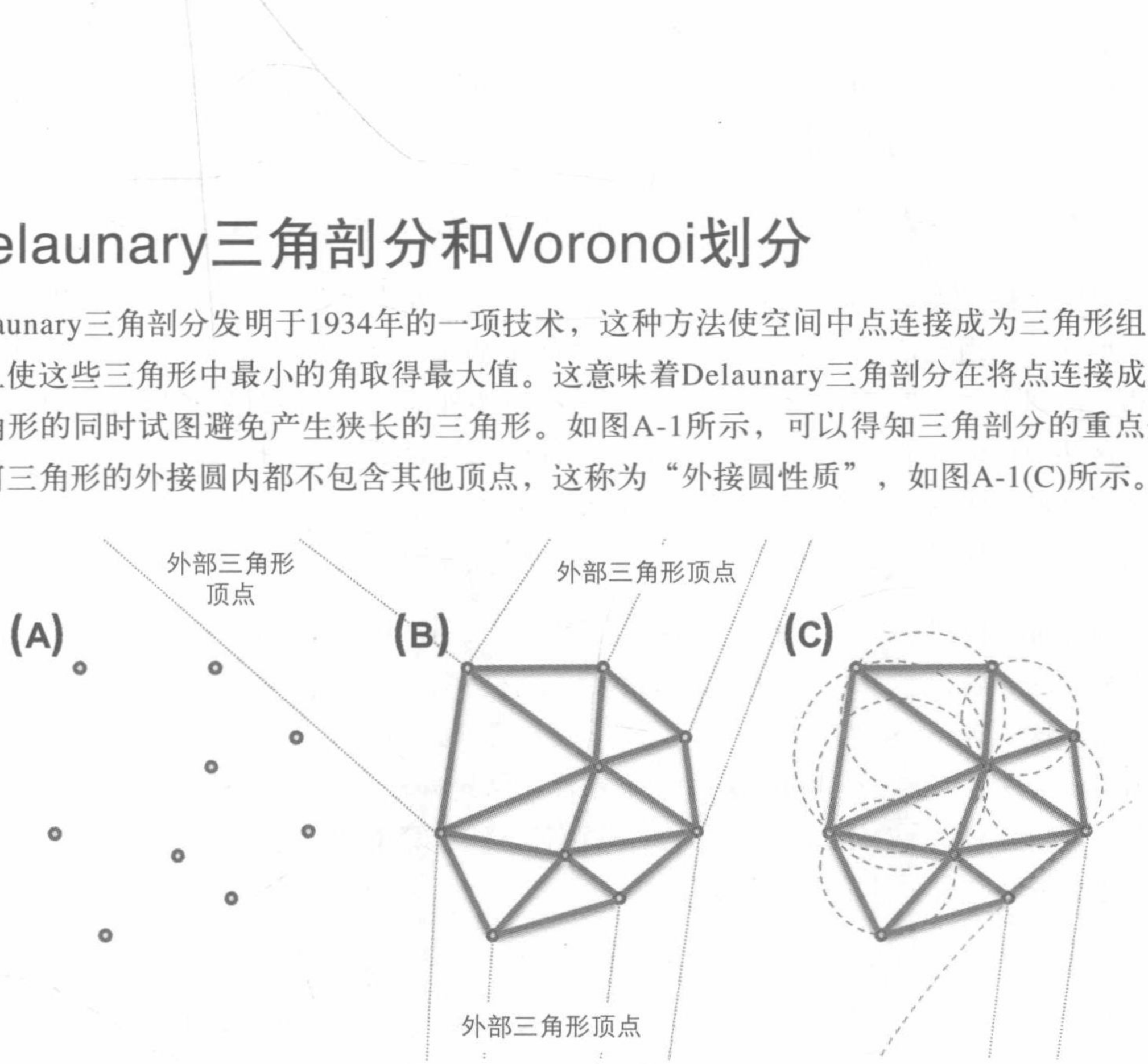
计算机视觉在未来有着丰富的前景，而且很可能是二十一世纪的关键技术之一。OpenCV似乎是计算机视觉的（至少部分）关键技术之一。创新的无限机会和深远的贡献就在眼前。我们希望这本书可以鼓励、激发和使所有有兴趣加入活跃的计算机视觉社区的人都能参与进来！

附录A

平面划分

Delaunary三角剖分和Voronoi划分

Delaunary三角剖分发明于1934年的一项技术，这种方法使空间中点连接成为三角形组，并且使这些三角形中最小的角取得最大值。这意味着Delaunary三角剖分在将点连接成为三角形的同时试图避免产生狭长的三角形。如图A-1所示，可以得知三角剖分的重点是任何三角形的外接圆内都不包含其他顶点，这称为“外接圆性质”，如图A-1(C)所示。



图A-1. Delaunary三角剖分： (A)点集合；(B)外部边界三角形带尾点集的Delaunary三角剖分；(C)外接圆性质

为了计算效率，Delaunary算法一开始创造了一个遥远的外部三角形。图A-1(B)用引自顶点的虚线表示虚拟外部三角形。图A-1(C)描述外接圆性质的一些例子，其中包括一个连接两个来自真实数据的外围点与一个虚构的外部三角形的一个顶点的圆（在图右下角）。

如今又很多算法可以用来计算Delaunary三角剖分，其中一些效率很高但是内部细节复杂。一个简单一些的算法的要点如下。

1. 添加外部三角形。
2. 添加一个内部点，然后在所有三角形的外接圆中搜索是否包含该点，删除包含该点的三角剖分。
3. 重新对图进行三角剖分，在上一步中去掉的三角剖分外接圆内添加新的点。
4. 返回步骤2，直到没有新的点加入。

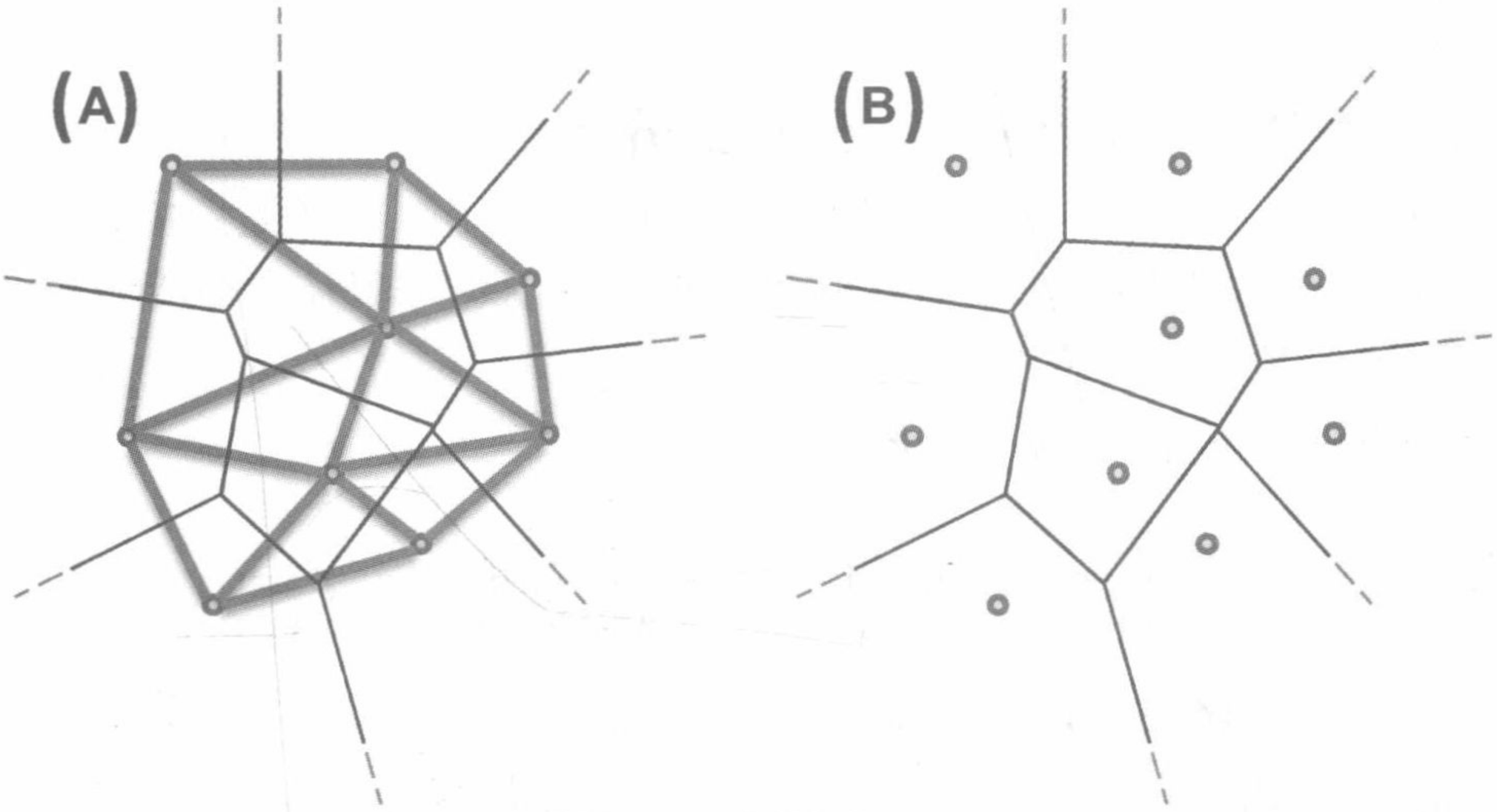
该算法的复杂度是$O(n^2)$。最好的算法复杂度可以低至$O(n \log \log n)$。

这种算法很棒，但是它的优势究竟在哪里？一方面，要知道这个算法从一个虚拟的外部三角形开始。因此，所有的真实的外部点实际上都与一个（或者两个）虚拟三角形的顶点相连。回忆一下外接圆性质：经过任意两个真实外部点与一个外部虚拟顶点的外接圆不包含其他任何内部点。这意味着计算机可能会通过寻找那些直接与三个外部虚拟顶点连接的点来确切地寻找哪些真实的点构成外部轮廓。换句话说，我们可以通过Delaunary三角剖分立即找到一组点的外部轮廓。

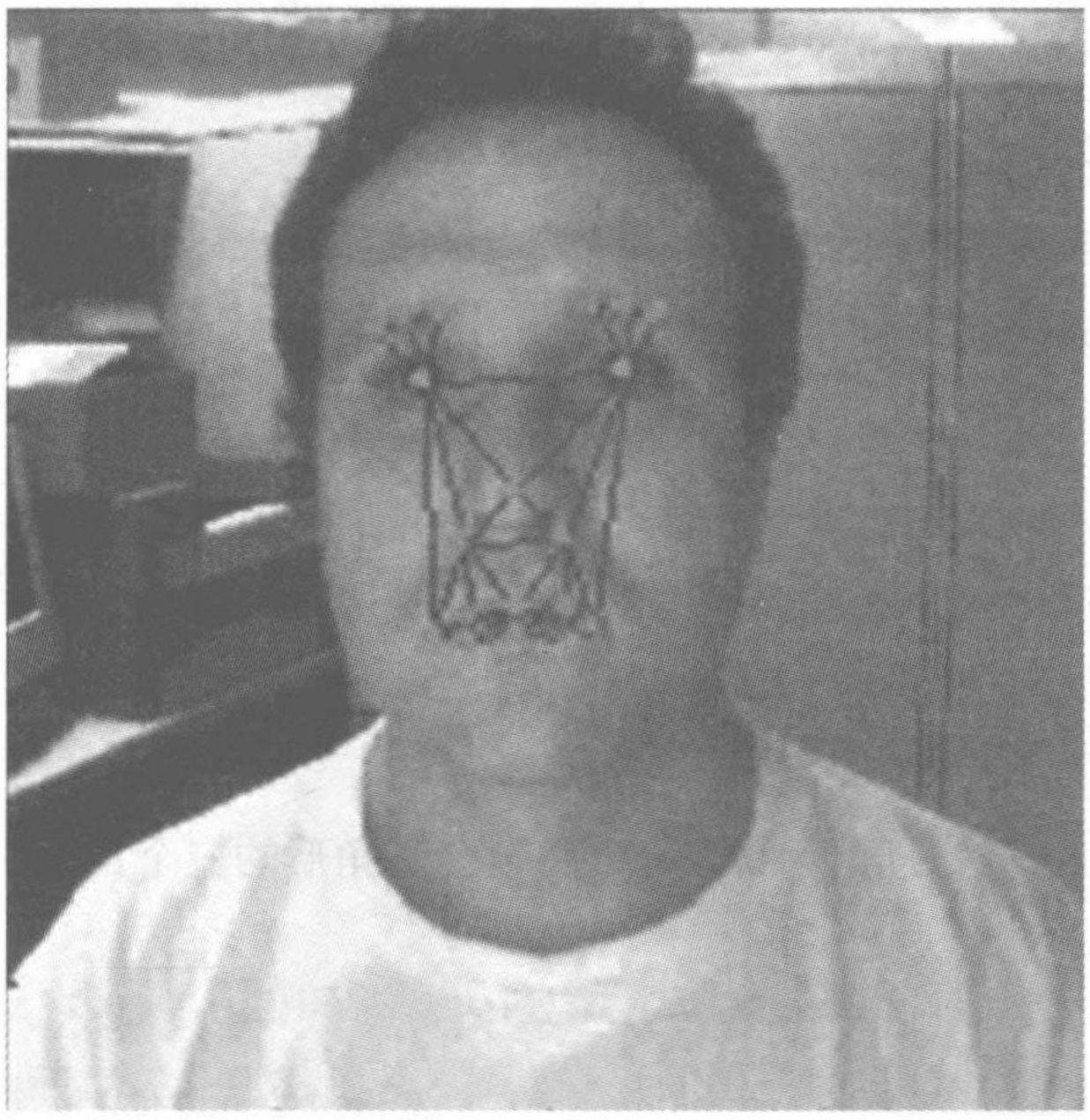
我们同样可以找到点与点之间的空间被谁“拥有”，也就是说，哪个坐标是离Delaunary顶点最近的邻居。这样，使用原始点的Delaunary三角剖分，你可以很快发现离新点最近的邻居。这叫Voronoi划分（图A-2）。这种划分是Delaunary三角剖分的对称图像，因为Delaunary线定义了已经存在的点之间的距离，因此Voronoi线“知道”为了保持点之间的等距应该在哪里与Delaunary线相交。这两种方法计算了凸包与最近邻居，是对点和点集合进行分类和聚类的重要基本操作。

如果对3D计算机图形学熟悉，你会意识到Delaunary三角剖分常常作为三维形状的基础。如果我们在三维空间渲染一个物体，可以通过这个物体的投影创建它的二维视图，然后利用二维Delaunary三角剖分来分析、确定这个物体或者将它与真实物体进行比较。因此，Delaunary三角剖分是计算机视觉与计算机图形学之间的桥梁。然而，OpenCV的一个缺陷是OpenCV仅可以实现二维的Delaunary三角剖分（我们希望可以尽快改进，详见第23章）。如果我们可以对三维点云进行三角剖分，比如利用立体视觉构建的点云（详见19章），就可以在三维计算机图形与三维计算机视觉之间无缝切换。但是，二维Delaunary三角剖分经常在计算机视觉中被用来标记目标物体的空间排列特征、运动场景

追踪、目标识别或者两个摄像机的匹配（比如从立体图像中提取深度信息）。图A-3显示Delaunary三角剖分在跟踪及识别中的应用[Göktürk01; Göktürk02]，其中关键面部特征点根据它们的三角剖分在空间分布。



图A-2. Voronoi划分，任何包含在Voronoi单元中的点都比其他Delaunary点更接近于它们自己的Delaunary点：(A)粗线表示Delaunary三角剖分及相应的细线表示Voronoi划分；(B)围绕着每一个Delaunary点的Voronoi细胞



图A-3. Delaunary点可以用来跟踪物体，图中利用表情关键点实现面部跟踪，由此识别情绪

现在我们在给出一组点集之后确定了Delaunary三角剖分的潜在价值，那我们如何剥离出三角形呢？OpenCV在.../*opencv/samples/cpp/delaunay2.cpp*文件中有相关例程。OpenCV把Delaunary三角剖分当作Delaunary细分的一种方法，我们接下来将讨论关键、可重用的部分。[注1]

创建一个Delaunary或Voronoi细分

首先，我们需要将Delaunary细分存储到内存空间中，同时需要一个外接矩形。为了提高计算速度，该算法需要一个位于外接矩形包围盒外部的虚拟三角形。初始化时，假设点都位于大小600 × 600 的图像中：

```
// STRUCTURE FOR DELAUNAY SUBDIVISION
//
...
cv::Rect        rect(0, 0, 600, 600);      // Our outer bounding box
cv::Subdiv2D    subdiv(rect);              // Create the initial subdivision
```

这段代码对一个包含指定矩形的三角形进行细分初始化。

接下来我们需要了解如何插入点。这些点必须是32位浮点型或者整型（例如cv::Point）。在下面的例子中，它们会被自动转换为浮点型。我们使用cv::Subdiv3D::insert()函数来添加点：

```
cv::Point2ffp; //This is our point holder

for(inti = 0; i<as_many_points_as_you_want; i++ ) {

    // However you want to set points
    //
    fp = your_32f_point_list[i];

    subdiv.insert(fp);
}
```

现在我们已经输入了这些点，可以得到Delaunary三角剖分。为了从Delaunary三角剖分中计算得到三角形，我们使用cv::Subdiv3D::getTriangleList()函数：

```
vector<cv::Vec6f> triangles;
subdiv.getTriangleList(triangles);
```

得到返回值之后，每一个三角形中的Vec6f将包含三角形的三个顶点：(*x1, y1, x2, y2, x3,*

注1：根据《韦氏大字典》，*subdivision*这个词既可以表示“某个物体被分割出来的一部分”，也可以表示“用建造房屋的被划分为小区块的土地”，在OpenCV中，这个词通常表示后者，用来表示部分的集合，而非组件自身。

y3)。我们可以用cv::Subdiv2D::getVoronoiFacetList()函数计算并求得相关的Voronoi划分：

```
vector<vector<cv::Point2f>> facets;
vector<cv::Point2f> centers;
subdiv.getVoronoiFacetList(vector<int>(), facets, centers);
```

Vector的第一个输出向量包含Voronoi面——即插入点的“邻近”区域，第二个向量包含对应的点——即区域中心。

值得注意的是，Delaunary三角剖分是是迭代生成的，每次插入新的点三角剖分都会被更新，所以剖分总是实时的。然而，当调用cv::Subdiv2D::calcV函数时，Voronoi划分采用批处理的方式构建，（该函数不用传入任何参数也没有返回值，仅仅更新内部细分表达），或者也可以调用cv::Subdiv2D::getVoronoiFacetList()函数（其内部名称为calcVoronoi()）当在Voronoi划分计算完成后插入一个新的点，划分将被标记为无效并在下一次需要时重新计算。该算法的复杂度为$O(N)$，其中N是顶点个数。

现在我们可以创立二维点集的Delaunary细分和对应的Voronoi划分，下一步是学习如何遍历细分。虽然许多情况下简单的三角剖分或划分就已经足够，但在细分中由边缘到点或者边缘到边缘的方法也很有用。我们将在下一节具体描述。

Delaunary细分遍历

细分图的基本数据元素是边缘。可以通过索引得到这条边缘，其他相邻边缘也可以通过索引和一个附加参数得到，这个附加参数表明了我们想要哪一条边缘与最初的边缘相连。每一条边缘都有两个与之相连的点（分别叫原点和终点）。这条边缘通过这些点，与共享这些点的其他边缘相连接。最终，这条边缘与它的相关边缘相连，对于Delaunay三角剖分的每一条边都有一条Voronoi图中的边与之对应，反之亦然。Opencv库支持相关方法从任一边缘得与它相关的边。

注意，在接口cv::Subdiv中，边缘总是被当成直线，这实际上是为了方便（处理），在Delaunary三角剖分和Voronoi图中，边缘都没有固有的方向。然而，当我们遍历整个数据结构时，这种表示非常有帮助。需要注意的是，这是一种区别一条边缘原点和终点的方法。

来自边缘的点

我们可以很容易从Delaunary边和Voronoi边确定端点位置。任意一条Delaunary边或Voronoi边缘都有两个端点，分别是原点和终点。可以通过以下函数得到其中任一点：

```
int cv::Subdiv2D::edgeOrg( int edge, cv::Point2f* orgpt = 0 ) const;
int cv::Subdiv2D::edgeDst( int edge, cv::Point2f* dstpt = 0 ) const;
```

这些函数可以给出各个顶点的索引和可选的点本身，给定一个点的索引，可以确定该点及与其相连的边缘：

```
cv::Point2f cv::Subdiv2D::getVertex( int vertex, int* firstEdge = 0 ) const;
```

请注意，如同边缘一样，每个点都有一个索引，点也有实际位置。记录在特定环境需要哪一个点是很重要的。cv::Subdiv2D接口的设计基于用户会在大多数接口函数中优先使用点和边缘的索引这一假设。

在细分中定位任意点

存在一种情况，你可能知道一个特定点的位置，想查询这个点在细分中的索引。类似的，可能有一个点根本不是三角剖分的顶点，但是我们想找到包含该点的三角或面。cv::Subdiv2d::locate()函数取一点作为输入并返回该点所在的其中一条边缘，三角形的任意一边或包含该点的一个面（如果该点不是一个顶点）。但是要注意，在这种情况下，返回的边不一定是最接近的边缘，只需要在三角形或表面上即可，当该点是一个顶点，cv::Subdiv2D::locate()也会返回返回这个顶点的索引。

```
int cv::Subdiv2D::locate(
  cv::Point2f pt,
  int&        edge,
  int&        vertex
);
```

该函数的返回值按下列方式表示点的位置：

cv::Subdiv2D::PTLOC_INSIDE
: 点落入某些面，*edge将包含该面的一个边缘

cv::Subdiv2D::PTLOC_ON_EDGE
: 点落于边缘，*edge含有这个边缘

cv::Subdiv2D::PTLOC_VERTEX
: 点与一个细分顶点重合，*vertex将包含该顶点指针

cv::Subdiv2D::PTLOC_OUTSIDE_RECT
: 点位于细分参考矩形之外，该函数返回后不填充指针

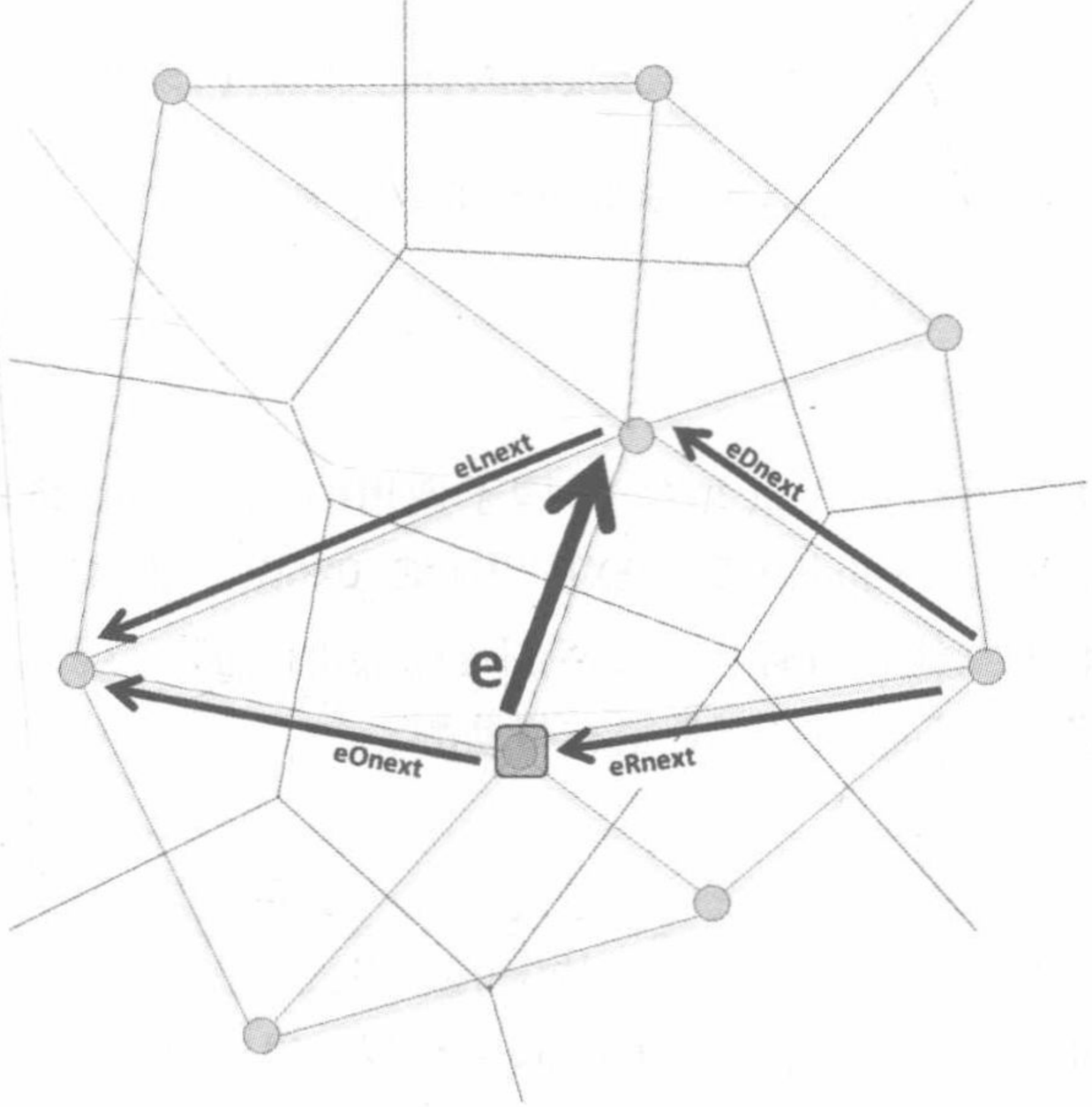
cv::Subdiv2D::PTLOC_ERROR
: 输入变量无效。

环绕定点

给定一个边缘，我们也许想得到与给定边的起点或终点相连的新边缘。具体步骤是，

确定开始的边缘，通过绕起点或终点顺时针或逆时针旋转得到下一条边缘。如图A-4所示，我们通过函数cv:Subdiv2D::getEdge()实现：

```
int cv:Subdiv2D::getEdge(
  int edge,
  int nextEdgeType              // see text below
) const;
```



图A-4. cv::Subdiv2D::Vertex顶点，连接边e及其他相连的边缘，可由cvSubdiv2DGetEdge()得到

调用函数cv:Subdiv2D::getEdge()时，我们输入当前边缘和nextEdgeType参数，参数值由下确定：

- cv::Subdiv2D::NEXT_AROUND_ORG，边缘原点的下一条（eOnext，如果e是输入边）
- cv::Subdiv2D::NEXT_AROUND_DST，边缘终点的下一条（eDnext）
- cv::Subdiv2D::PREV_AROUND_ORG，边缘原点的前一条（eRnext的反向）
- cv::Subdiv2D::PREV_AROUND_DST，边缘终点的前一条（eLnext的反向）

根据你想遍历的方式，也可以用以下值来定义下一步（最终是相同的）：

- cv::Subdiv2D::NEXT_AROUND_LEFT，左区域的下一条（eLnext）或下一个左平面
- cv::Subdiv2D::NEXT_AROUND_RIGHT，右区域的下一条（eRnext）或下一个右平面

- cv::Subdiv2D::PREV_AROUND_LEFT，左区域的前一条（eOnext的反向）或前一个左平面
- cv::Subdiv2D::PREV_AROUND_RIGHT，右区域的前一条（eDnext的反向）或前一个右平面

如果我们在Delaunary边缘或者Voronoi边缘上，就可以通过上面的步骤遍历Delaunary三角形或Voronoi单元。

为了简化，我们也可以调用简化的cv:Subdiv2D::nextEdge()函数：

```
// equivalent to getEdge(edge, cv::Subdiv2D::NEXT_AROUND_ORG)
//
int cv:Subdiv2D::nextEdge(
  int edge
) const;
```

调用cv:Subdiv2D::nextEdge()完全等同于调用cv:Subdiv2D::getEdge()并将nextedgetype参数设为cv::Subdiv2D::NEXT_AROUND_ORG以手动构建的方式给定一个与顶点连接的边缘，我们可以找到所有与该顶点相连的其他边缘。这对于从外接三角形的顶点（虚线）开始寻找类似凸包的东西是很有帮助的 。

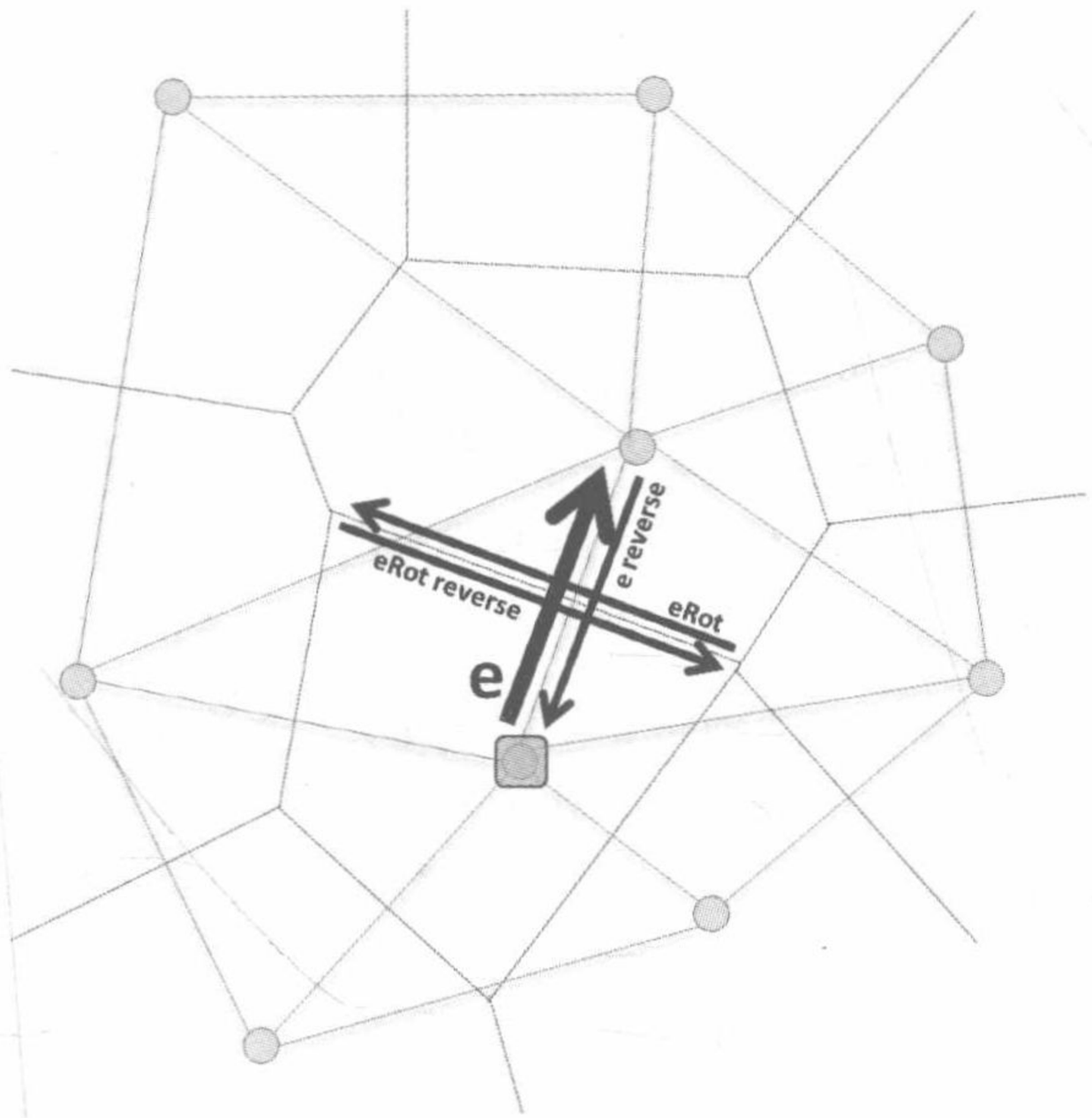
边缘旋转

假设，不管是从某个其他函数中得到一个特定的索引或者仅仅是从任意点开始并遍历整个图形得到，你都可以由以下函数从Delaunay三角剖分得到对应Voronoi图中的边（反之亦然）。

```
int cv::Subdiv2D::rotateEdge(
  int edge,
  int rotate        // get other edges in the same quad-edge: modulo 4 operation
) const;
```

这种情况下变量edge是当前边缘的索引，rotate参数表示想要得到的边缘。给定一个边，采用以下参数确定下一个边缘。

- 0输入边（e如图A-5所示，如果e是输入边）
- 1旋转边缘（eRot）
- 2旋转边缘（reversed e）
- 3反向的旋转边缘（reversed eRot）



图A-5. 由函数cv::Subdiv2DRotateEdge()、含Delaunary边缘、其对边及相关V边缘和点得到的四边形

关于顶点和边缘的更多思考

顶点及其编号

由于Delaunary三角剖分是经过初始化的，因而总有以下几点。

1. 1.0号顶点为空，不存在。（它只是一种书中抽象的记号）
2. 后三个顶点是在给定外接矩形外侧的虚拟顶点，每一个都远离输入点。
3. 所有后继顶点都是cv::Subdiv2D对象集合的一部分。

边缘及其编号

cv::Subdiv2D每条边缘都由一个整数定义，这些整数构造为每四个对应如下：

edge % 4 == 0

Delaunay边缘

edge % 4 == 1

垂直于初始边缘的Voronoi边

edge % 4 == 2

反向初始Delaunary边缘

```
edge % 4 == 3
```

反向Voronoi边

虚拟和空边缘

第0条边是一条没有方向的空边缘（或者，更确切的说，其顶点存在第0个顶点（空顶点））第1, 2, 3条边缘都是连接虚拟顶点的虚拟Delaunary边缘，由于两个顶点都是虚拟的，我们这种边叫自由虚拟边。空边坐标总是(0, 0)，分别表明起点和终点的位置。

任意从空边缘旋转得到一条边缘的操作总会得到另一条新的空边缘。如同任何一条由cv::Subdiv2D::nextEdge得到的后继边缘，从空顶点得到的“第一条边”也是空边缘。由任意虚拟顶点得到的“第一条边”总是连接另一个虚拟顶点。

确定外接三角形

为了方便，当我们由一组点集建立了Delaunary三角剖分，总是由前三个点建立外接三角形（不包含第0个点，详情参见前面“关于顶点和边缘的更多思考”）我们可以通过下面的方式得到这三个点：

```
Point2f outer_vtx[3];
for( int i = 0; i < 3; i++ ) {
  outer_vtx[i] = subdiv.getVertex(i+1);
}
```

同样可以得到外接三角形的三条边。

```
int outer_edges[3];
outer_edges[0] = 1*4;
outer_edges[1] = subdiv.getEdge(outer_edges[0], Subdiv2D::NEXT_AROUND_LEFT);
outer_edges[2] = subdiv.getEdge(outer_edges[1], Subdiv2D::NEXT_AROUND_LEFT);
```

现在我们已经知道怎样遍历图形，可以处理何时在外部边缘或者点的边界这样的问题了。

确定凸包的外接三角形或边缘并遍历凸包

回忆一下我们通过调用cv::Subdiv2D(rect)，利用一个外接矩形rect来初始化Delaunary三角剖分。在这种情况下，有下面的论述成立。

- 如果边的起点和终点都在矩形之外，则这条边在细分的虚拟外接三角形上。这些是我们说的自由虚拟边。

- 如果边的一端在矩形内，而另外一端在矩形之外，则矩形边界内的点落在凸集上，凸集上的每个点与虚拟外接三角形的两顶点相连，这两边相继出现。我们可以把这些连接矩形中的点与矩形外的虚拟点的边称作自由虚拟边。

如果想找到点集的凸包，我们可以利用这些方法并且迅速生成凸包。[注2]例如，从虚拟顶点1, 2, 3开始，使用cv:Subdiv2D::nextEdge()函数，我们可以很快生成所有固定虚拟边（通过简单排除所有自由虚拟边）。对cv:Subdiv2D::rotateEdge(2)的快速访问将其翻转，而对cv:Subdiv2D::nextEdge的一次或者两次访问可以给出点集的凸包。[注3]对于每个固定虚拟边，都有一个精确的凸包边界，而所有边界的集合就是点集的凸包。

我们现在知道如何初始化Delaunary和Voronoi划分，寻找初始边界，遍历图中的边界和点。在下一小节中，我们将呈现一些实际应用。

应用实例

我们可以使用cv::Subdiv2D::locate()来遍历Delaunary三角形的边。在这个例子中，我们写了一个函数来对包含给定点的三角剖分的每条边做“一些事”：

```
voidlocate_point(
  cv::Subdiv2D&       subdiv,
  const cv::Point2f & fp,
  ...
) {
  int e;
  int e0 = 0;
  int vertex = 0;
  subdiv.locate(fp, e0, vertex );
  if( e0 > 0 ) {
    e = e0;
    do // Always 3 edges -- this is a triangulation, after all.
    {
      // [Insert your code here]
      //
      // Do something with e ...
       e = subdiv.getEdge( e, cv::Subdiv2D::NEXT_AROUND_LEFT );
    }
    while( e != e0 );
  }
}
```

我们还可以通过下面的函数找到和输入点距离最近的点：

注2：事实上，有很多方法可以完成这个任务，这里的描述只是想给读者介绍一种完成的思路。

注3：回想由于一些外部的顶点与三角形中的另外两个顶点相连。仅仅调用一次cv::Subdiv2D::nextEdge()可能并不能满足要求。最好对这个顶点及其得到的边进行检查，这种情况下，再次调用cv::Subdiv2D::nextEdge()也许是有必要的。

```
int Subdiv2D::findNearest(
cv::Point2fpt,
cv::Point2f* nearestPt
);
```

与cv::Subdiv2D::locate()不同，cv::Subdiv2D::findNearest()函数返回细分的最近顶点的整数ID。这个点不一定落在该点所在的面或者三角内。注意，这是一种不确定的方法，因为当点丢失或者没有更新时，才会计算Voronoi划分。

同样，我们能够遍历Voronoi面（我们已经画出）：

```
voiddraw_subdiv_facet(
  cv::Mat&      img,
  cv::Subdiv2D& subdiv,
  int           edge
) {

  int t = edge;
  inti, count = 0;
  vector<cv::Point>buf;

  // Count number of edges in facet
  do{
     count++;
     t = subdiv.getEdge( t, cv::Subdiv2D::NEXT_AROUND_LEFT );
  } while (t != edge );

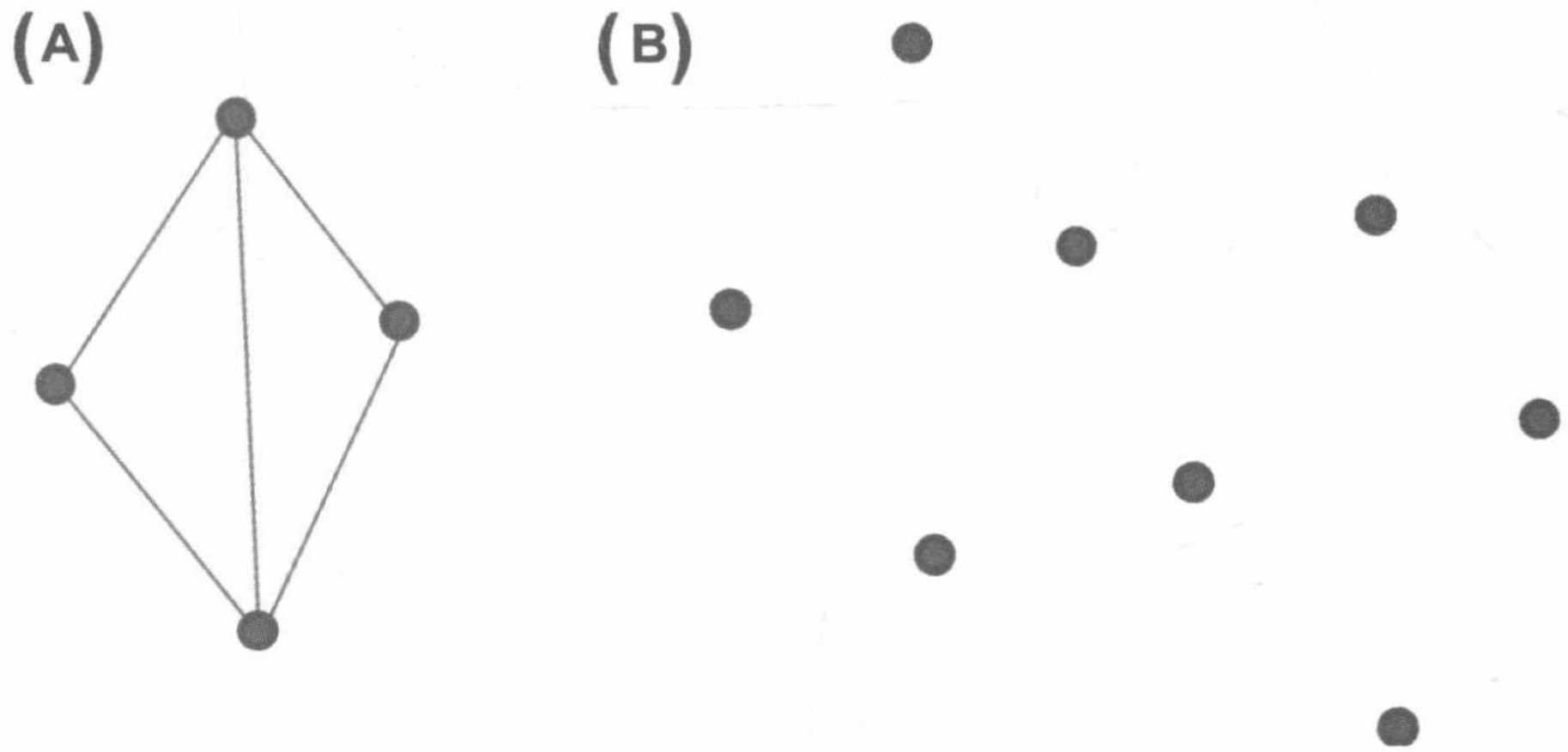
  // Gather points
  //
  buf.resize(count);
  t = edge;
  for( i = 0; i< count; i++ ) {
      cv::Point2fpt;
      if( subdiv.edgeOrg(t, &pt) <= 0 )
          break;
      buf[i] = cv::Point(cvRound(pt.x), cvRound(pt.y));
      t = subdiv.getEdge( t, cv::Subdiv2D::NEXT_AROUND_LEFT );
  }

  // Around we go
  //
  if(i == count ){
  cv::Point2fpt;
  subdiv.edgeDst(subdiv.rotateEdge(edge, 1), &pt);
  fillConvexPoly(
    img, buf,
    cv::Scalar(rand()&255,rand()&255,rand()&255),
    8, 0
  );
  vector< vector<cv::Point>> outline;
  outline.push_back(buf);
  polylines(img, outline, true, cv::Scalar(), 1, cv::LINE_AA, 0);
  draw_subdiv_point(img, pt, cv::Scalar(0,0,0) );
```

```
    }
  }
```

练习

1. 修改代码.../*opencv/samples/cpp/delaunay2.cpp*，使得允许鼠标点击添加点（而不是通过随机选择点的方法），对这些结果做三角剖分实验。
2. 再一次修改代码*delaunay2.cpp*，使得可以使用键盘画点集凸包。
3. 共线的三点有Delaunary三角剖分吗？
4. 图A-6(A)显示的三角剖分是一个Delaunary三角剖分吗？如果是，说明原因。如果不是，如何调整图形使之成为一个Delaunary三角剖分？
5. 画出图A-6(B)中的点的一个Delaunary三角剖分。在这个练习中，不用添加一个虚拟外接三角形。



图A-6. 练习4与练习5

附录B

opencv_contrib模块概述

*opencv_contrib*这个仓库中存放着大部分新的由用户生成的内容，包含的视觉应用比OpenCV主仓库更加完整。这个仓库由许多没有互相依赖项的模块组成，每个模块都需要有文档、单元测试和示例代码，并且许多模块也还有教程。每个模块都要符合其他模块的格式，Buildbot测试，单元测试等都要在OpenCV的核心模块上通过。每个模块以OpenCV常规功能模块一样的方式进行设计，它们一起形成具有更强计算机视觉能力和我维护能力的超集，供他人使用。

*opencv_contrib*代码目录位于*https://github.com/opencv/opencv_contrib*，并且需要和opencv主库分开编译。可以在nightly build site找到这些模块的文档（也包括常规的opencv模块），接下来是这些模块的内容摘要。

opencv_contrib中包含的模块

以下概述了*opencv_contrib*仓库中所有可用的模块，这些模块必须单独下载和编译。如果决定编译这个库，但不想编译所有的功能模块，则可以通过把编译命令中的*<reponame>*替换成这个模块的名字来编译任何给定的模块。

```
$ cmake -D OPENCV_EXTRA_MODULES_PATH=<opencv_contrib>/modules \
  -D BUILD_opencv_<reponame>=OFF                              \
  <opencv_source_directory>
```

*opencv_contrib*的功能如下。

aruco

ArUco和ChArUco标记。包含增强现实标记ArUco和ChArUco（把ArUco标记嵌入到棋盘格标定板的白色区域）。

bgsegm

背景分割。改进了自适应背景混合模型并用于可变光照条件下的实时人体跟踪。

bioinspired

生物视觉。一种受生物学启发的视觉模型，提供了的方法可以最小化噪声和亮度差异，处理瞬态事件分割，以及执行高动态范围（HDR）色调映射。

ccalib

自定义标定。用于三维重建的图案，全向相机标定，随机图案标定，多相机标定。

cnn_3dobj

深度目标识别和位姿估计。使用Caffe深度神经网络库构建，训练和测试视觉目标识别和位姿估计的CNN模型。

contrib_world

*opencv_contrib*容器。contrib_world是一个在构建时包含其他所有。*opencv_contrib*模块的模块，它可以用于更方便地重新发行OpenCV二进制文件。

cvv

计算机视觉调试器。通过简单几句代码就可以把它加入到你的程序里，从而弹出一个GUI，方便你交互式和可视化的调试计算机视觉程序。

datasets

数据集读取器。用于读取已经存在的计算机视觉数据库的代码和基于读取器使用这些数据集的数据训练、测试和运行的例子。

dnn

深度神经网络（DNN）。该模块可以读取在Caffe神经网络库中训练的图像识别网络，并在CPU上高效运行。

dnns_easily_fooled

颠覆DNN。这个代码可以利用网络中的激活函数愚化网络，然后去识别其他东西。

dpm

可变形组件模型。Felzenszwalb的可变形组件的级联目标识别程序。

face

人脸识别。人脸识别技术包含Eigen，Fisher和局部二值编码直方图[LBPH]方法。

fuzzy

图像的模糊逻辑。模糊逻辑图像变换和逆变换，模糊图像处理。

hdf

层次型数据存储，这个模块包含分层数据格式的I/O例程，意味着可以存储大量数据。

line_descriptor

线段的提取和匹配。提取、描述和用二进制描述子锁定线段的方法，作者之一Gary通过改进这个算法创办了一个搬运箱子的机器人技术公司（Industrial Perception Inc.）

matlab

MATLAB接口。某些opencv核心模块使用的OpenCV MATLAB Mex代码封装器。

optflow

光流。用于运行和评估深流、简单流、稀疏流和运动模板（剪影流）的算法。

plot

绘制。绘图模块可让你轻松绘制一维或二维数据。

reg

图像配准。精确对齐的基于像素的图像配准。参考Richard Szelisk的论文[Szeliski04]。

rgbd

RGB深度处理模块。Linemod三维目标识别，快速表面法线计算，三维平面查找，3D视觉里程计。

saliency

显著性检测API。人们在场景中会注意的地方，有静态的、动态的和“objectness”显著性例程。

sfm

运动恢复结构。 该模块包含从2D图像序列进行三维重建的算法，该模块的核心是轻型版本的Libmv。

stereo

双目立体匹配。用不同的描述符进行双目匹配：Census，CS-Census，MCT，BRIEF和 MV。

structured_light

结构光的使用。怎样生成和投影格雷码图案并使用它们来计算场景的稠密深度。

surface_matching

点对特征，使用多通道点对特征实现3D目标检测和定位。

text

视觉文本匹配。 在视觉场景中检测文本，分离词汇和识别文本。

tracking

基于视觉的目标跟踪。使用和（或者）评估五种视觉跟踪技术中的一个。

xfeatures2d

Features2D 的扩展。扩展的2D特征框架包含实验和付费的2D特征检测和描述子算法：SURF、SIFT、BRIEF、Censure、Freak、LUCID、Daisy和Self-similar。

ximgproc

图像处理扩展。包含结构森林，变化域滤波器，导向滤波，自适应流形滤波器，联合双边滤波器和超像素。

xobjdetect

增强2D目标检测。 使用Waldboost级联和局部二值编码算子作为2D目标检测的积分特征。

xphoto

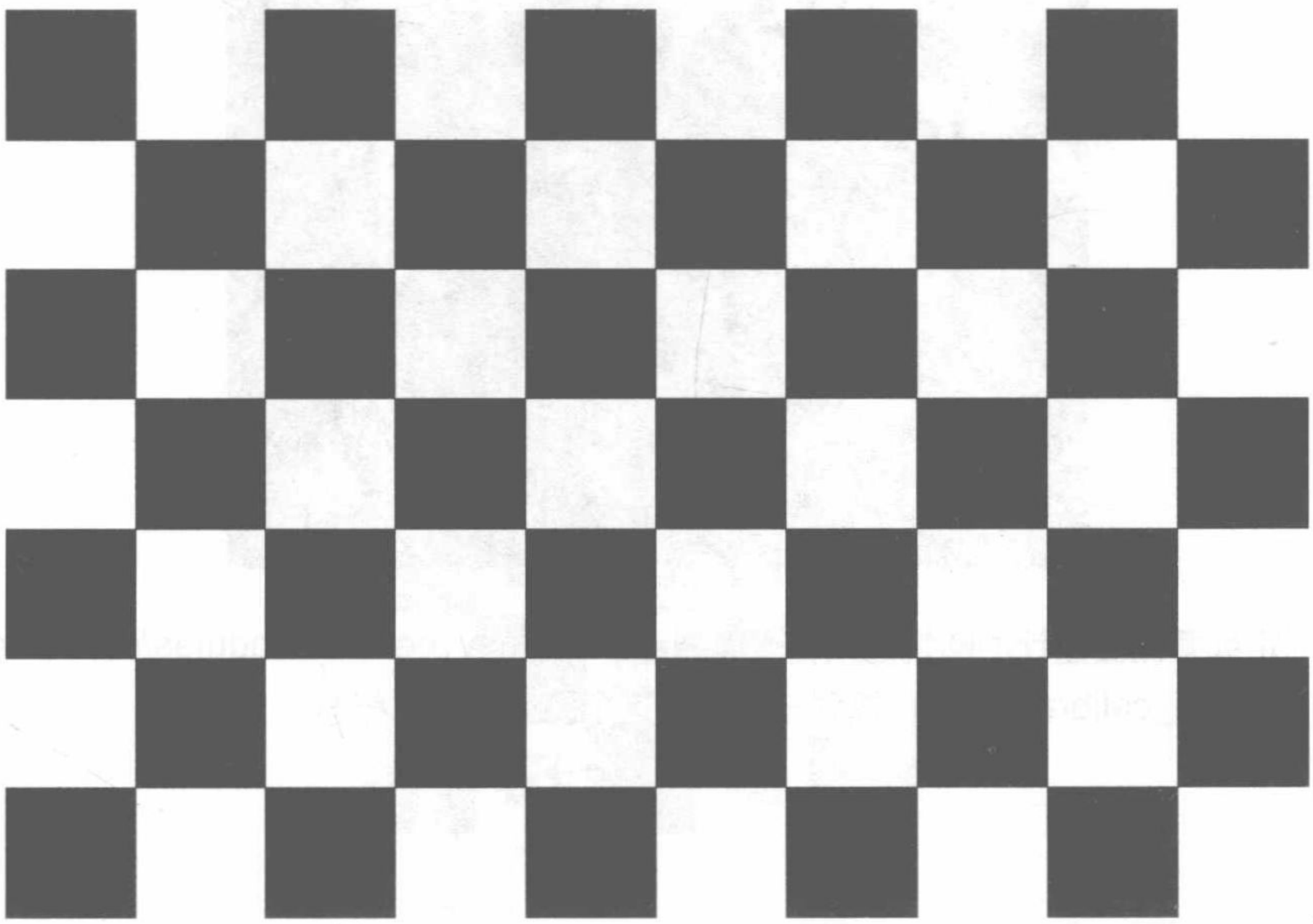
计算摄影扩展，提供附加照片处理算法：色彩平衡，去噪和图像修复。

附录C

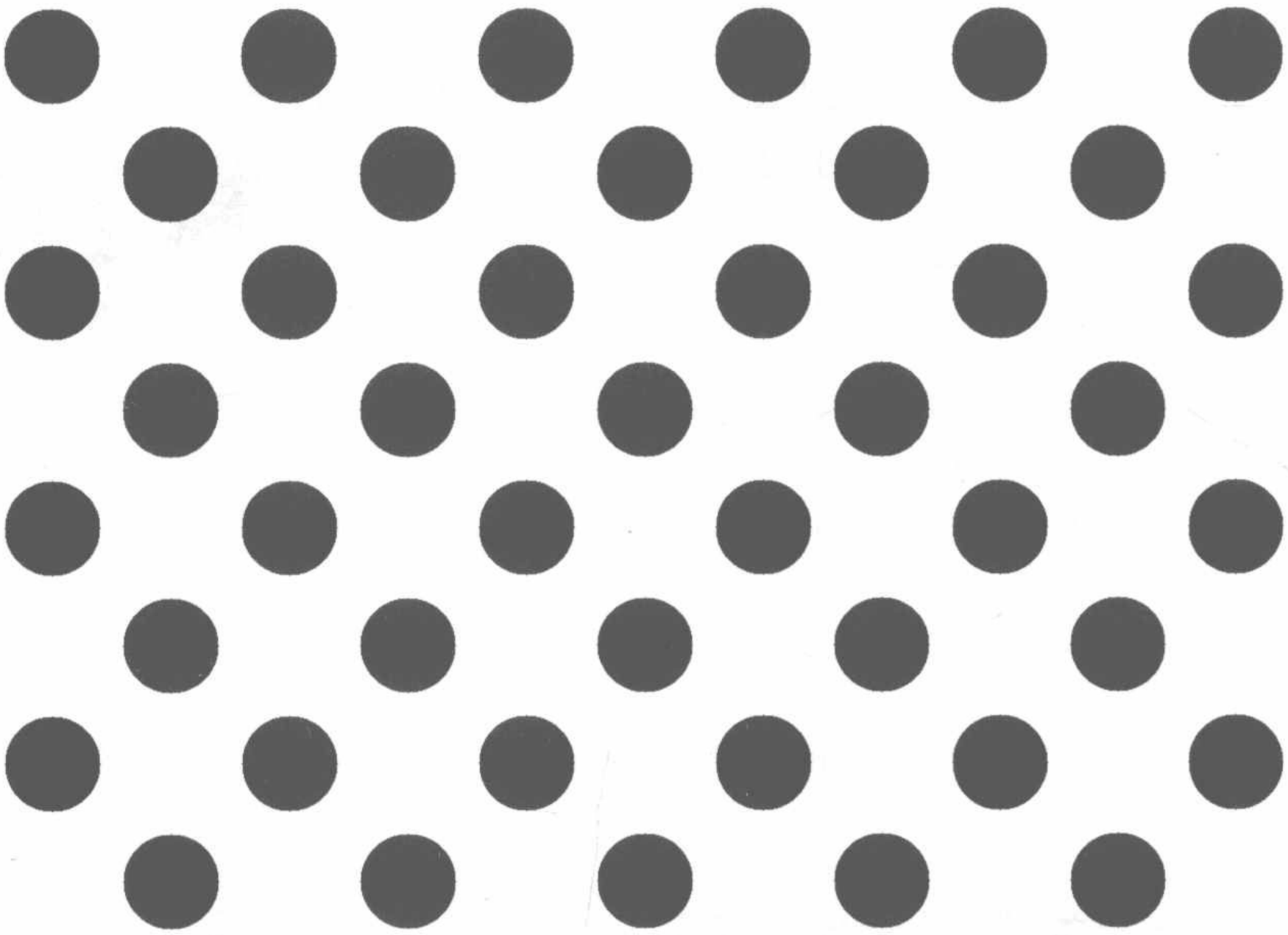
标定图案

Opencv标定图案的使用

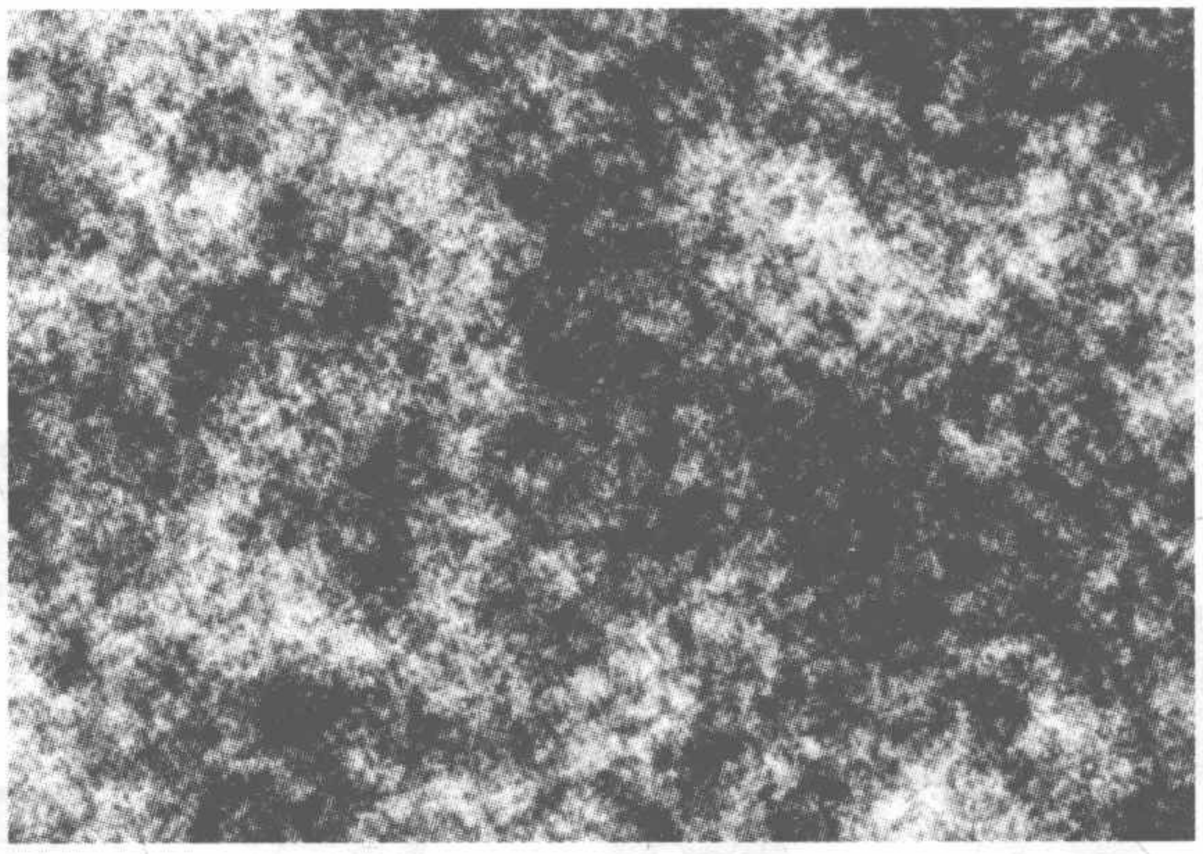
有许多不一样的标定图案，每个图案或标记都可以用于标定程序，也可以仅用于找到标记的3D位姿。图C-1到图C-7是七种不同的图案或标记。



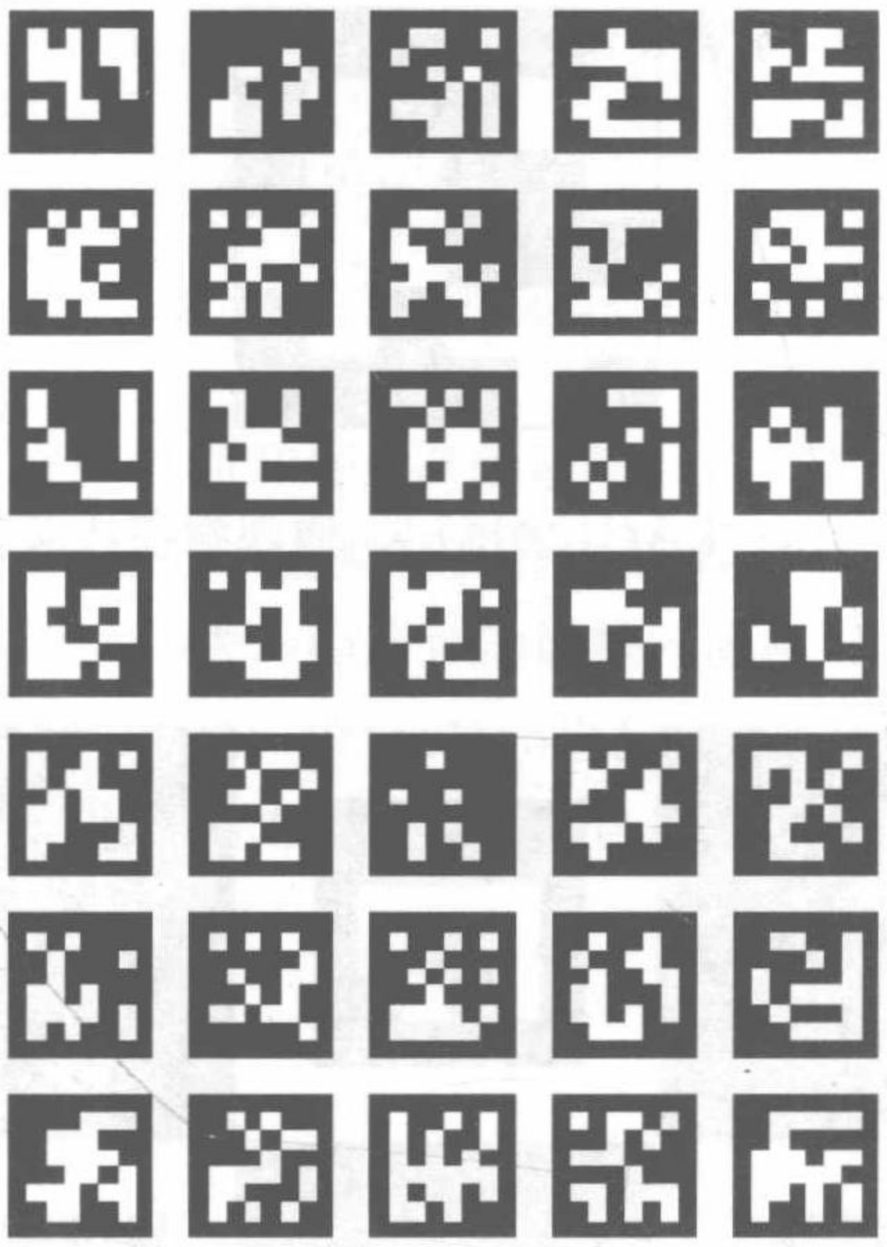
图C-1. 用于相机标定或位姿估计的棋盘图案（9×6 格）。 这个图案可以用第18章中介绍的标准标定方法或者参考opencv.org上的在线相机标定教程



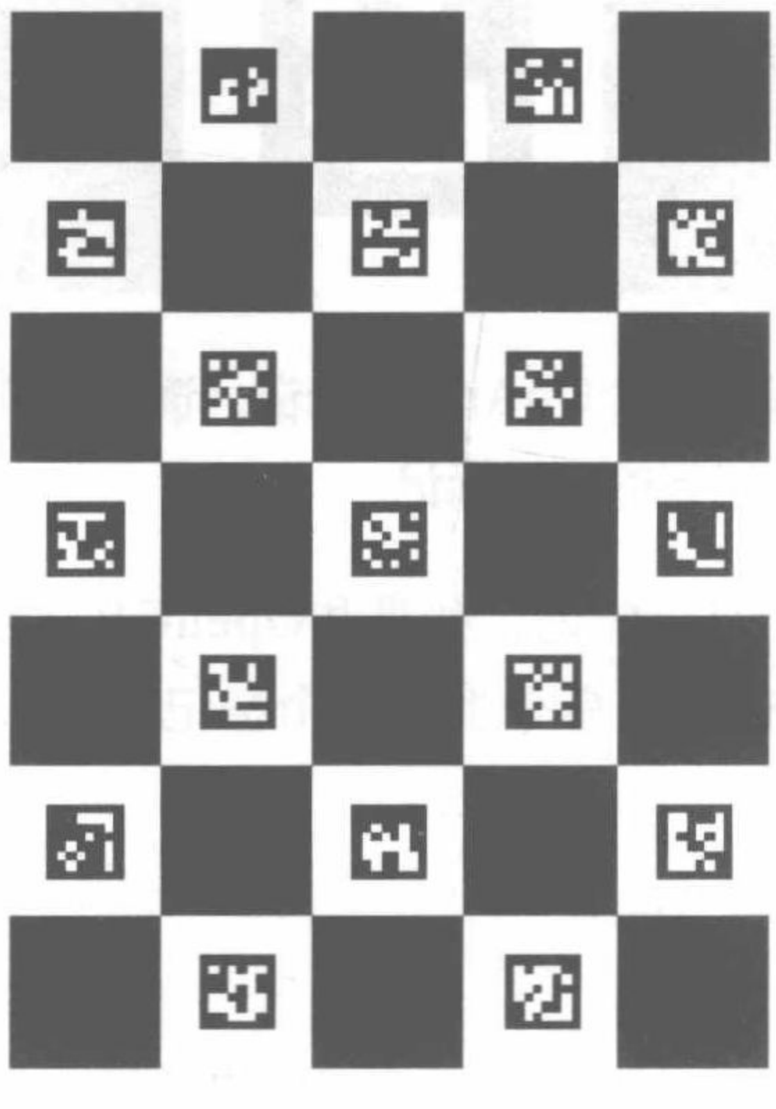
图C-2. 标定或者位姿估计的圆形特征图，该特征可用于书中介绍的标准标定方法或者网页上tutorial_camera_calibration.html的findCirclesGrid()函数



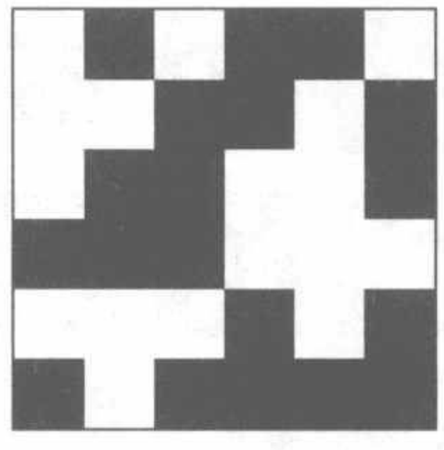
图C-3. 标定或者位姿估计的随机图案，可以通过.../opencv_contrib/modules/ccalib/samples/random_pattern_calibration.cpp 找到一个使用这个图案的示例程序



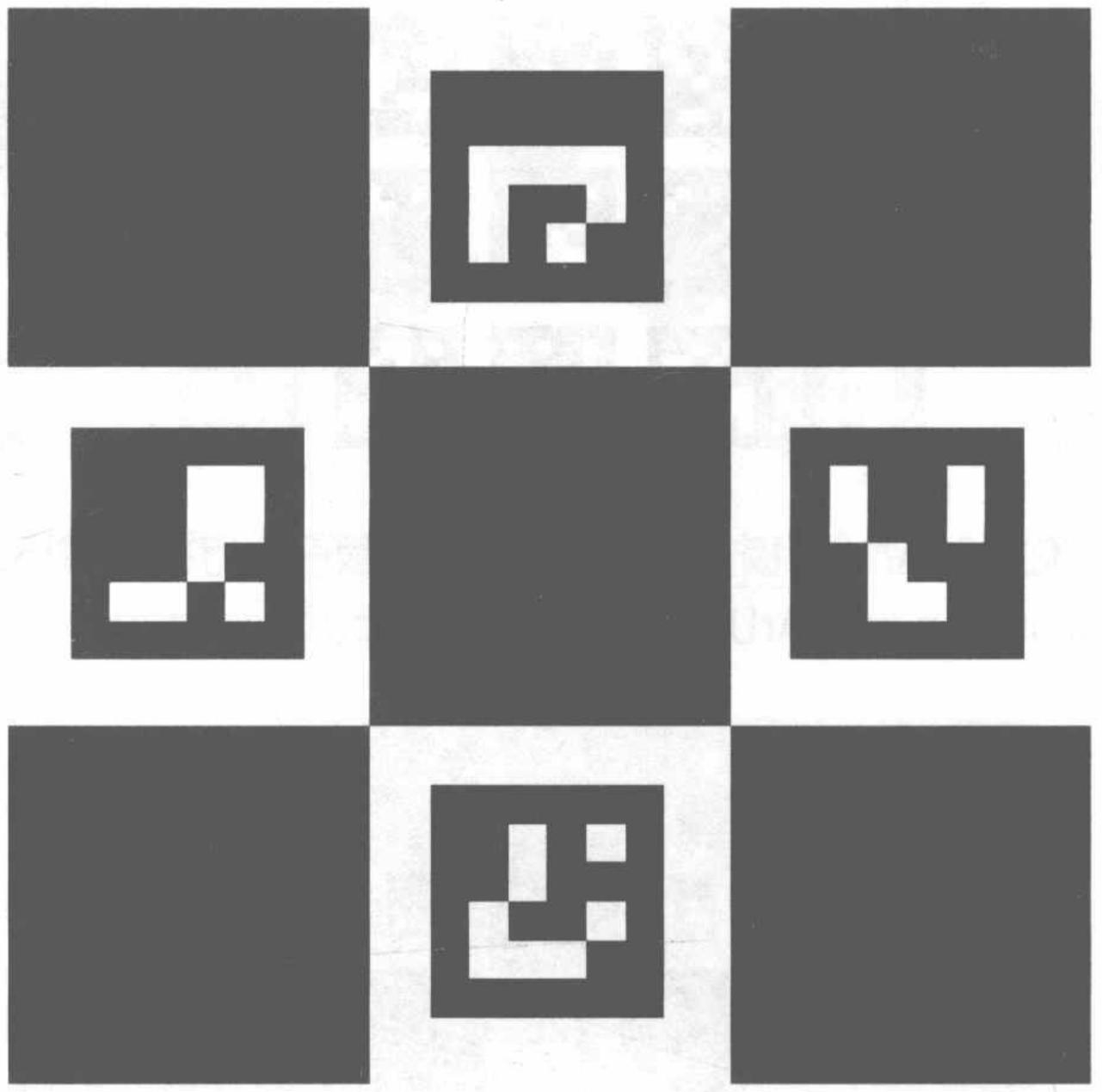
图C-4. 标定或者位姿估计的ArUco板，关于怎样用这个板子进行检测和标定，请观看opencv.org 的在线教程“Calibration with ArUco and ChArUco”



图C-5. 标定或者位姿估计的ChArUco板，关于怎样用这个板子进行检测和标定，请观看在线教程“Calibration with ArUco and ChArUco”



图C-6. ArUco标记，如何检测一个ArUco标记，请观看在线教程“Detection of ArUco Markers”



图C-7. ChArUco标记，如何检测一个ChArUco标记，请观看教程“Detection of Diamond Markers”，这是这本书封面上出现的方块标记

图C-7显示了一个方块状ChArUco标记，如果和OpenCV一起构建*opencv_contrib*模块，就可以在相应的*bin*目录下运行以下命令生成这个标记。

```
./example_aruco_create_diamond -bb=1 \
-d=0 -sl=200 -ml=130 -ids=1,2,3,4 \
-m=10 -si=true diamond.png
```

这个命令会生成一个方块状ChArUco标记，其中-bb=1代表生成一个和标记位一样宽的标记边界；-d=0表示生成4×4位标记（只有四个标记，我们不需要标记来表示大数字）；-sl=200表示每个棋盘方块大小为200×200像素；-ml=130表示ArUco标记的大小为130x130像素的方块；-id = 1,2,3,4只是将ArUco标记分别设置为1,2,3和4的编码值；-m = 10表示在整个方块状标记周围放置一个10像素宽的边框；而-si = true则告诉程序

在屏幕上显示生成的图像。最后，*diamond.png*是输出图像的名称。

要检测此图像以及本书封面上的图像，请使用以下命令：

```
./example_aruco_detect_diamonds -as=1.0 -ci=0 -d=0 -ml=130 -sl=200
```

这个程序会调用一个相机检测封面和图C-7显示的标记，一旦找到，该程序就会解码这个ArUco标记并且在图像上显示检测框和角，该程序会在本书封面上的标记上工作，[注1]在这个命令行中，`-as=1.0`告诉程序自动调整检测框大小，`-ci=0`告诉代码寻找默认的系统相机，`-d=0`告诉代码使用4×4的ArUco检测（解码）库，`-ml=130`和`-sl=200`给出ArUco和棋盘方格的相对尺寸（因为我们设置了自动缩放，所以只需要相对尺寸）。

注1：　作为本书最后一个练习，我们希望读者能够利用本书封面的标记来做一些有趣的AR应用。

参考文献

[Ahonen04] Ahonen, Timo, Abdenour Hadid, and Matti Pietikäinen. "Face recognition with local binary patterns." *European conference on computer vision*. Springer Berlin Heidelberg, 2004.

[Acharya05] Acharya, T., and A. Ray. *Image Processing: Principles and Applications*. New York: Wiley, 2005.

[Adelson84] Adelson, E. H., C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. "Pyramid methods in image processing," *RCA Engineer 29* (1984): 33–41.

[Agarwal08] Agrawal, M., K. Konolige, and M. R. Blas. "CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching," *European Conference on Computer Vision,* 2008.

[Ahmed74] Ahmed, N., T. Natarajan, and K. R. Rao. "Discrete cosine transform," *IEEE Transactions on Computers* 23 (1974): 90–93.

[Alahi12] Alahi, Alexandre, Raphael Ortiz, and Pierre Vandergheynst. "Freak: Fast retina keypoint." *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on.* IEEE, 2012.

[Arfken85] Arfken, G. "Convolution theorem," in *Mathematical Methods for Physicists*, 3rd ed. (pp. 810–814), Orlando, FL: Academic Press, 1985.

[Arraiy] Arraiy Corporation, *http://www.arraiy.ai*.

[Arthur07] Arthur, D., and S. Vassilvitskii., "k-means++: the advantages of careful seeding." *Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms*. (2007) pp. 1027–1035.

[Bajaj97] Bajaj, C. L., V. Pascucci, and D. R. Schikore. "The contour spectrum," *Proceedings of IEEE Visualization 1997* (pp. 167–173), 1997.

[Ballard81] Ballard, D. H. "Generalizing the Hough transform to detect arbitrary shapes," *Pattern Recognition* 13 (1981): 111–122.

[Ballard82] Ballard, D., and C. Brown. *Computer Vision*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[Bardyn84] Bardyn, J. J. et al. "Une architecture VLSI pour un operateur de filtrage median," *Congres reconnaissance des formes et intelligence artificielle* (vol. 1, pp. 557–566), Paris, 25–27, January 1984.

[Bay06] Bay, H., T. Tuytelaars, and L. V. Gool. "SURF: Speeded up robust features," *Proceedings of the Ninth European Conference on Computer Vision* (pp. 404–417), May 2006.

[Bay08] Bay, Herbert, Andreas Ess, Tinne Tuytelaars, and Luc J. Van Gool. "SURF: Speeded up robust features," *Computer Vision and Image Understanding* (CVIU) 110(3) (2008) 346–359.

[Bayes1763] Bayes, T. "An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F.R.S. communicated by Mr. Price, in a letter to John Canton, A.M.F.R.S.," *Philosophical Transactions, Giving Some Account of the Present Undertakings, Studies and Labours of the Ingenious in Many Considerable Parts of the World 53* (1763): 370–418.

[Bazargani15] Bazargani, Hamid, Olexa Bilaniuk, and Robert Laganiere. "A fast and robust homography scheme for real-time planar target detection." *Journal of Real-Time Image Processing* (2015): 1–20.

[Belongie02] Belongie, S., J. Malik, and J. Puzicha. "Shape matching and object recognition using shape contexts," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.24, no. 4, pp. 509–522, Apr 2002.

[Bhattacharyya43] Bhattacharyya, A. "On a measure of divergence between two statistical

populations defined by probability distributions," *Bulletin of the Calcutta Mathematical Society* 35 (1943): 99–109.

[BirchfieldTomasi99] Birchfield, Stan, and Carlo Tomasi. "Depth discontinuities by pixel-to-pixel stereo." *International Journal of Computer Vision* 35.3 (1999): 269–293.

[Bishop07] Bishop, C. M. *Pattern Recognition and Machine Learning*. New York: Springer-Verlag, 2007.

[Black92] Black, M. J. "Robust incremental optical flow" (YALEU-DCS-RR-923), Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT, 1992.

[Black93] Black, M. J., and P. Anandan. "A framework for the robust estimation of optical flow," *Fourth International Conference on Computer Vision* (pp. 231–236), May 1993.

[Black96] Black, M. J., and P. Anandan. "The robust estimation of multiple motions: Parametric and piecewise-smooth flow fields," *Computer Vision and Image Understanding* 63 (1996): 75–104.

[Bobick96] Bobick, A., and J. Davis. "Real-time recognition of activity using temporal templates," *IEEE Workshop on Applications of Computer Vision* (pp. 39–42), December 1996.

[Borgefors86] Borgefors, G., "Distance transformations in digital images," *Computer Vision, Graphics and Image Processing* 34 (1986): 344–371.

[Bouguet04] Bouguet, J.-Y. "Pyramidal implementation of the Lucas Kanade feature tracker description of the algorithm," *http://robots.stanford.edu/cs223b04/algo_tracking.pdf*.

[Boykov01] Boykov, Yuri, Olga Veksler, and Ramin Zabih. "Fast approximate energy minimization via graph cuts." *IEEE Transactions on pattern analysis and machine intelligence* 23.11 (2001): 1222-1239.

[Bracewell65] Bracewell, R. "Convolution" and "Two-dimensional convolution," in *The Fourier Transform and Its Applications* (pp. 25–50 and 243–244). New York: McGraw-Hill, 1965.

[Bradski] Gary Bradski, founder of OpenCV in 1999 and maintainer ever since. *https://en.wikipedia.org/wiki/Gary_Bradski*.

[Bradski00] Bradski, G., and J. Davis. "Motion segmentation and pose recognition with motion history gradients," *IEEE Workshop on Applications of Computer Vision*, 2000.

[Bradski98a] Bradski, G. R. "Real time face and object tracking as a component of a perceptual user interface," *Proceedings of the 4th IEEE Workshop on Applications of Computer Vision*, October 1998.

[Bradski98b] Bradski, G. R. "Computer video face tracking for use in a perceptual user interface," *Intel Technology Journal* Q2 (1998): 705–740.

[Breiman01] Breiman, L. "Random forests," *Machine Learning* 45 (2001): 5–32.

[Breiman02] Breiman, Leo. "Manual on setting up, using, and understanding random forests v3. 1." Statistics Department, University of California Berkeley, CA, USA 1 (2002).

[Breiman84] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Monterey, CA: Wadsworth, 1984.

[Brown71] Brown, D. C. "Close-range camera calibration," *Photogrammetric Engineering* 37 (1971): 855–866.

[Buades05] Mahmoudi, Mona, and Guillermo Sapiro. "Fast image and video denoising via nonlocal means of similar neighborhoods." *IEEE signal processing letters* 12.12(2005): 839–842.

[Burt83] Burt, P. J., and E. H. Adelson. "The Laplacian pyramid as a compact image code," *IEEE Transactions on Communications* 31 (1983): 532–540.

[Calonder10] Calonder, M., V. Lepetit, C. Strecha, and P. Fua. Brief: Binary robust independent elementary features. *In European Conference on Computer Vision (ECCV)*, 2010.

[Canny86] Canny, J. "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8 (1986): 679–714.

[Carpenter03] Carpenter, G. A., and S. Grossberg. "Adaptive resonance theory," in M. A. Arbib (Ed.), *The Handbook of Brain Theory and Neural Networks*, 2nd ed. (pp. 87–90), Cambridge, MA: MIT Press, 2003.

[Carr04] Carr, H., J. Snoeyink, and M. van de Panne. "Progressive topological simplification using contour trees and local spatial measures," *15th Western Computer Graphics Symposium*, Big White, British Columbia, March 2004.

[Chambolle04] Chambolle, Antonin. "An algorithm for total variation minimization and applications." *Journal of Mathematical imaging and vision* 20.1–2 (2004): 89–97.

[Chen05] Chen, D., and G. Zhang. "A new sub-pixel detector for x-corners in camera calibration targets," *WSCG Short Papers* (2005): 97–100.

[Chu07] Chu, C.-T., S. K. Kim, Y.-A. Lin, Y. Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. "Map-reduce for machine learning on multicore," *Proceedings of the Neural Information Processing Systems Conference* (vol. 19, pp. 304–310), 2007.

[Ciresan11] Ciresan, Dan Claudiu, et al. "Convolutional neural network committees for handwritten character classification." 2011 International Conference on Document Analysis and Recognition. IEEE, 2011.

[Colombari07] Colombari, A., A. Fusiello, and V. Murino. "Video objects segmentation by robust background modeling," *International Conference on Image Analysis and Processing* (pp. 155–164), September 2007.

[Comaniciu99] Comaniciu, D., and P. Meer. "Mean shift analysis and applications," *IEEE International Conference on Computer Vision* (vol. 2, p. 1197), 1999.

[Comaniciu03] Comaniciu, D. "Nonparametric information fusion for motion estimation," *IEEE Conference on Computer Vision and Pattern Recognition* (vol. 1, pp. 59–66), 2003.

[Cooley65] Cooley, J. W., and O. W. Tukey. "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation* 19 (1965): 297–301.

[Criminisi13] Criminisi, Antonio, and Jamie Shotton, eds. *Decision forests for computer vision and medical image analysis*. Springer Science & Business Media, 2013.

[Csurka04] Csurka, Gabriella, et al. "Visual categorization with bags of keypoints." *Workshop on statistical learning in computer vision, ECCV*. Vol. 1. No. 1–22. 2004.

[Dahlkamp06] Dahlkamp, H., A. Kaehler, D. Stavens, S. Thrun, and G. Bradski. "Selfsupervised monocular road detection in desert terrain," *Robotics: Science and Systems, Philadelphia*, 2006.

[Dalal05] Dalal, N., and B. Triggs. "Histograms of oriented gradients for humandetection," *Computer Vision and Pattern Recognition* (vol. 1, pp. 886–893), June 2005.

[Davis97] Davis, J., and A. Bobick. "The representation and recognition of action using temporal templates" (Technical Report 402), MIT Media Lab, Cambridge, MA, 1997.

[Davis99] Davis, J., and G. Bradski. "Real-time motion template gradients using Intel CVLib," *ICCV Workshop on Framerate Vision*, 1999.

[Delaunay34] Delaunay, B. "Sur la sphère vide," *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk* 7 (1934): 793–800.

[DeMenthon92] DeMenthon, D. F., and L. S. Davis. "Model-based object pose in 25 lines of code," *Proceedings of the European Conference on Computer Vision* (pp. 335–343), 1992.

[Dempster77] Dempster, A., N. Laird, and D. Rubin. "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society, Series B* 39 (1977): 1–38.

[Douglas73] Douglas, D., and T. Peucker. "Algorithms for the reduction of the number of points required for represent a digitized line or its caricature," *Canadian Cartographer* 10(1973): 112–122.

[Drucker97] Drucker, Harris, Chris J.C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. "Support vector regression machines." *Advances in neural information processing systems* 9 (1997): 155-161.

[Duda72] Duda, R. O., and P. E. Hart. "Use of the Hough transformation to detect lines and curves in pictures," *Communications of the Association for Computing Machinery* 15 (1972): 11–15.

[Duda73] Duda, R. O., and P. E. Hart. *Pattern Recognition and Scene Analysis*. New York: Wiley, 1973.

[Duda00] Duda, R. O., P. E. Hart, and D. G. Stork. *Pattern Classification*. New York: Wiley, 2001.

[Farin04] Farin, D., P. H. N. de With, and W. Effelsberg. "Video-object segmentation using multi-sprite background subtraction," *Proceedings of the IEEE International Conference on Multimedia and Expo*, 2004.

[Farnebäck03] Farnebäck, Gunnar. "Two-frame motion estimation based on polynomial expansion." *Scandinavian conference on image analysis. Springer Berlin Heidelberg*, 2003.

[Faugeras93] Faugeras, O. *Three-Dimensional Computer Vision: A Geometric Viewpoint*. Cambridge, MA: MIT Press, 1993.

[Felzenszwalb2004] Felzenszwalb, Pedro, and Daniel Huttenlocher. *Distance transforms of sampled functions*. Cornell University, 2004.

[Felzenszwalb2006] Felzenszwalb, Pedro F., and Daniel P. Huttenlocher. "Efficient belief propagation for early vision." *International Journal of Computer Vision*, 70(1), October 2006.

[Felzenszwalb2010] Felzenszwalb, Pedro F., et al. "Object detection with discriminatively trained part-based models." *IEEE transactions on pattern analysis and machine intelligence* 32.9 (2010): 1627–1645.

[Fischler81] Fischler, M. A., and R. C. Bolles. "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the Association for Computing Machinery* 24 (1981): 381–395.

[Fitzgibbon95] Fitzgibbon, A. W., and R. B. Fisher. "A buyer's guide to conic fitting," *Proceedings of the 5th British Machine Vision Conference* (pp. 513–522), Birmingham, 1995.

[Fix51] Fix, E., and J. L. Hodges. "Discriminatory analysis, nonparametric discrimination: Consistency properties" (Technical Report 4), USAF School of Aviation Medicine, Randolph Field, Texas, 1951.

[Forsyth03] Forsyth, D., and J. Ponce. *Computer Vision: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall, 2003.

[FourCC85] Morrison, J. "EA IFF 85 standard for interchange format files," *http://www.martinreddy.net/gfx/2d/IFF.txt*.

[Fourier] "Joseph Fourier," http://en.wikipedia.org/wiki/Joseph_Fourier.

[Freeman95] Freeman, W. T., and M. Roth. "Orientation histograms for hand gesture recognition," *International Workshop on Automatic Face and Gesture Recognition* (pp. 296–301), June 1995.

[Freund97] Freund, Y., and R. E. Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences* 55 (1997): 119–139.

[Fryer86] Fryer, J. G., and D. C. Brown. "Lens distortion for close-range photogrammetry," *Photogrammetric Engineering and Remote Sensing* 52 (1986): 51–58.

[Fukunaga90] Fukunaga, K. *Introduction to Statistical Pattern Recognition*. Boston: Academic Press, 1990.

[Fukushima80] Fukushima, Kunihiko. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position." *Biological cybernetics* 36.4 (1980): 193-202.

[Galton] "Francis Galton," *http://en.wikipedia.org/wiki/Francis_Galton*.

[Gao03] Gao, Xiao-Shan, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. "Complete solution classification for the perspective-three-point problem," *IEEE Transactions Pattern Analysis and Machine Intelligence* 25 (2003), 930–943.

[Garrido-Jurado] Garrido-Jurado, S., R. Munoz-Salinas, F. J. Madrid-Cuevas and M. J. Marin-Jimenez. "Automatic generation and detection of highly reliable fiducial markers under occlusion," Pattern Recognition 47, no. 6 (June 2014).

[Göktürk01] Göktürk, S. B., J.-Y. Bouguet, and R. Grzeszczuk. "A data-driven model for monocular face tracking," *Proceedings of the IEEE International Conference on Computer Vision* (vol. 2, pp. 701–708), 2001.

[Göktürk02] Göktürk, S. B., J.-Y. Bouguet, C. Tomasi, and B. Girod. "Model-based face tracking for view-independent facial expression recognition," *Proceedings of the Fifth IEEE International Conference on Automatic Face and Gesture Recognition* (pp. 287–293), May 2002.

[Goresky03] Goresky, Mark, and Andrew Klapper. "Efficient multiply-with-carry random number generators with maximal period." *ACM Transactions on Modeling and Computer Simulation* (TOMACS) 13.4 (2003): 310-321.

[Grossberg87] Grossberg, S., "Competitive learning: From interactive activation to adaptive resonance," *Cognitive Science* 11 (1987): 23–63.

[Harris88] Harris, C., and M. Stephens. "A combined corner and edge detector," *Proceedings of the 4th Alvey Vision Conference* (pp. 147–151), 1988.

[Hartley98] Hartley, R. I. "Theory and practice of projective rectification," *International Journal of Computer Vision* 35 (1998): 115–127.

[Hartley06] Hartley, R., and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge, UK: Cambridge University Press, 2006.

[Hastie01] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. New York: Springer-Verlag, 2001.

[Heckbert90] Heckbert, P. A *Seed Fill Algorithm* (Graphics Gems I). New York: Academic Press, 1990.

[Heikkila97] Heikkila, J., and O. Silven. "A four-step camera calibration procedure with implicit image correction," *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition* (p. 1106), 1997.

[Hinton06] Hinton, G. E., S. Osindero, and Y. Teh. "A fast learning algorithm for deep belief nets," *Neural Computation* 18 (2006): 1527–1554.

[Hirschmuller 08] Hirschmuller, H. "Stereo Processing by Semiglobal Matching and Mutual Information," *Pattern Analysis and Machine Intelligence PAMI 30,* No. 2, February 2008, pp. 328–341.

[Ho95] Ho, T. K. "Random decision forest," *Proceedings of the 3rd International Conference on Document Analysis and Recognition* (pp. 278–282), August 1995.

[Horn81] Horn, B. K. P., and B. G. Schunck. "Determining optical flow," *Artificial Intelligence* 17 (1981): 185–203.

[Hough59] Hough, P. V. C. "Machine analysis of bubble chamber pictures," *Proceedings of the International Conference on High Energy Accelerators and Instrumentation* (pp. 554–556), 1959.

Huttenlocher, Daniel P., Gregory A. Klanderman, and William J. Rucklidge. "Comparing images using the Hausdorff distance." *IEEE Transactions on pattern analysis and machine intelligence* 15.9 (1993): 850-863.

[Intel] Intel Corporation, *http://www.intel.com/*.

[Inui03] Inui, K., S. Kaneko, and S. Igarashi. "Robust line fitting using LmedS clustering," *Systems and Computers in Japan* 34 (2003): 92–100.

[IPP] Intel Integrated Performance Primitives, *https://software.intel.com/en-us/intelipp*.

[Itseez] A computer vision company that grew out of the original OpenCV project and one of the key maintainers of the free and open OpenCV.org. Now sold to Intel Corporation.

[Jaehne95] Jaehne, B. *Digital Image Processing*, 3rd ed. Berlin: Springer-Verlag, 1995.

[Jaehne97] Jaehne, B. *Practical Handbook on Image Processing for Scientific Applications.* Boca Raton, FL: CRC Press, 1997.

[Jain77] Jain, A. "A fast Karhunen-Loeve transform for digital restoration of images degraded by white and colored noise," *IEEE Transactions on Computers* 26 (1997): 560–571.

[Jain86] Jain, A. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[Johnson84] Johnson, D. H. "Gauss and the history of the fast Fourier transform," *IEEE Acoustics, Speech, and Signal Processing Magazine* 1 (1984): 14–21.

[KaewTraKulPong2001] KaewTraKulPong, P., and R. Bowden. "An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection," *Proc. 2nd European Workshop on Advanced Video Based Surveillance Systems*, AVBS01. Sept 2001.

[Kalman60] Kalman, R. E. "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering* 82 (1960): 35–45.

[Kim05] Kim, K., T. H. Chalidabhongse, D. Harwood, and L. Davis. "Real-time foreground-background segmentation using codebook model," *Real-Time Imaging* 11 (2005): 167–256.

[Kimme75] Kimme, C., D. H. Ballard, and J. Sklansky. "Finding circles by an array of accumulators," *Communications of the Association for Computing Machinery* 18 (1975): 120–122.

[Kiryati91] Kiryati, N., Y. Eldar, and A. M. Bruckshtein. "A probablistic Hough transform," *Pattern Recognition* 24 (1991): 303–316.

[Koller09] Koller, Daphne, and Nir Friedman. *Probabilistic graphical models: principles and techniques*. Cambridge, MA: MIT Press, 2009.

[Konolige97] Konolige, K., "Small vision system: Hardware and implementation," *Proceedings of the International Symposium on Robotics Research* (pp. 111–116), Hayama, Japan, 1997.

[Kopf07] Kopf, Johannes, et al. "Joint bilateral upsampling." *ACM Transactions on Graphics* (TOG). Vol. 26. No. 3. ACM, 2007.

[Kreveld97] van Kreveld, M., R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. chikore. "Contour trees and small seed sets for isosurface traversal," *Proceedings of the 13th ACM Symposium on Computational Geometry* (pp. 212–220), 1997.

[Lakoff08] Lakoff, G., and M. Johnson. "Metaphors we live by," University of Chicago Press, 2008.

[Laughlin81] Laughlin, S. B. "A simple coding procedure enhances a neuron's information capacity," *Zeitschrift für Naturforschung* 9/10 (1981): 910–912.

[LeCun98a] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition," *Proceedings of the IEEE* 86 (1998): 2278–2324.

[LeCun98b] LeCun, Y., L. Bottou, G. Orr, and K. Muller. "Efficient BackProp," in G. Orr and K. Muller (Eds.), *Neural Networks: Tricks of the Trade*. New York: Springer-Verlag, 1998.

[Leutenegger11] Leutenegger, Stefan, Margarita Chli, and Roland Y. Siegwart."BRISK: Binary robust invariant scalable keypoints." *2011 International conference on computer vision*. IEEE, 2011.

[Lienhart02] Lienhart, Rainer, and Jochen Maydt. "An extended set of haar-like features for rapid object detection." *Proceedings 2002 International Conference on Image Processing*. Vol. 1. IEEE, 2002.

[Liu07] Liu, Y. Z., H. X. Yao, W. Gao, X. L. Chen, and D. Zhao. "Nonparametric background generation," *Journal of Visual Communication and Image Representation* 18 (2007): 253–263.

[Lloyd57] Lloyd, S. "Least square quantization in PCM's" (Bell Telephone Laboratories Paper), 1957. ["Lloyd's algorithm" was later published in *IEEE Transactions on Information Theory* 28 (1982): 129–137.]

[Lloyd82] Lloyd, Stuart. "Least squares quantization in PCM." *IEEE transactions on information theory 28.2* (1982): 129-137.

[Lowe04] Lowe, D. G. "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision* 60, no. 2 (2004): 91–110.

[LTI] LTI-Lib, Vision Library, *http://ltilib.sourceforge.net/doc/homepage/index.shtml*.

[Lucas81] Lucas, B. D., and T. Kanade. "An iterative image registration technique with an application to stereo vision," *Proceedings of the 1981 DARPA Imaging Understanding Workshop* (pp. 121–130), 1981.

[Lucchese02] Lucchese, L., and S. K. Mitra. "Using saddle points for subpixel feature detection in camera calibration targets," *Proceedings of the 2002 Asia Pacific Conference on Circuits and Systems* (pp. 191–195), December 2002.

[Lv07] Lv, Q., W. Josephson, Z. Wang, M. Charikar, and K. Li. "Multiprobe LSH: efficient indexing for high-dimensional similarity search." In *VLDB*, pages 950–961, 2007.

[Mahalanobis36] Mahalanobis, P. "On the generalized distance in statistics," *Proceedings of the National Institute of Science* 12 (1936): 49–55.

[Mair10] Mair, Elmar, et al. "Adaptive and generic corner detection based on the accelerated segment test." European conference on Computer vision. Springer Berlin Heidelberg, 2010.

[Maron61] Maron, M. E. "Automatic indexing: An experimental inquiry," *Journal of the Association for Computing Machinery* 8 (1961): 404–417.

[Marr82] Marr, D. *Vision*. San Francisco: Freeman, 1982.

[Marsaglia00] Marsaglia, George, and Wai Wan Tsang. "The ziggurat method for generating random variables." *Journal of statistical software* 5.8 (2000): 1-7.

[Martins99] Martins, F. C. M., B. R. Nickerson, V. Bostrom, and R. Hazra. "Implementation of a real-time foreground/background segmentation system on the Intel architecture," *IEEE International Conference on Computer Vision Frame Rate Workshop*, 1999.

[Matas00] Matas, J., C. Galambos, and J. Kittler. "Robust detection of lines using the progressive probabilistic Hough transform," *Computer Vision Image Understanding* 78 (2000): 119–137.

[Meer91] Meer, P., D. Mintz, and A. Rosenfeld. "Robust regression methods for computer vision: A review," *International Journal of Computer Vision* 6 (1991): 59–70.

[Merwe00] van der Merwe, R., A. Doucet, N. de Freitas, and E. Wan. "The unscented particle filter," *Advances in Neural Information Processing Systems*, December 2000.

[Meyer78] Meyer, F. "Contrast feature extraction," in J.-L. Chermant (Ed.), *Quantitative Analysis of Microstructures in Material Sciences, Biology and Medicine* [Special issue of *Practical Metallography*], Stuttgart: Riederer, 1978.

[Meyer92] Meyer, F. "Color image segmentation," *Proceedings of the International Conference on Image Processing and Its Applications* (pp. 303–306), 1992.

[Minsky61] Minsky, M. "Steps toward artificial intelligence," *Proceedings of the Institute of Radio Engineers* 49 (1961): 8–30.

[Moreno-Noguer07] Moreno-Noguer, F., Lepetit, V., Fua, P. "Accurate Non-Iterative O(n) Solution to the PnP Problem," *ICCV 2007. IEEE 11th International Conference on Computer Vision*, pp. 1–8, 2007.

[Morse53] Morse, P. M., and H. Feshbach. "Fourier transforms," in *Methods of Theoretical Physics* (Part I, pp. 453–471), New York: McGraw-Hill, 1953.

[Muja09] Muja, Marius, and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." *VISAPP (1)* 2.331–340 (2009): 2.

[Neapolitan04] Neapolitan, Richard E. *Learning Bayesian Networks*. Upper Saddle River, New Jersey: Pearson, 2004.

[O'Connor02] O'Connor, J. J., and E. F. Robertson. "Light through the ages: Ancient Greece to Maxwell," *http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Light_1.html*.

[Ojala94] Ojala, T., M. Pietikäinen, and D. Harwood. "Performance evaluation of texture measures with classification based on Kullback discrimination of distributions," Pattern Recognition, 1994. Vol. 1 [Oliva06] A. Oliva and A. Torralba, "Building the gist of a scene: The role of global image features in recognition visual perception," *Progress in Brain Research* 155 (2006): 23–36.

[OpenCV] Open Source Computer Vision Library (OpenCV) (free, BSD license), *http://opencv.org*.

[opencv_contrib] Newer content and higher functionality is separated into this repository. *https://github.com/opencv/opencv_contrib*.

[Papoulis62] Papoulis, A. *The Fourier Integral and Its Applications*. New York: McGraw-Hill, 1962.

[Pascucci02] Pascucci, V., and K. Cole-McLaughlin. "Efficient computation of the topology of level sets," *Proceedings of IEEE Visualization* 2002 (pp. 187–194), 2002.

[Pearson] "Karl Pearson," *http://en.wikipedia.org/wiki/Karl_Pearson*.

[Pollefeys99a] Pollefeys, M. "Self-calibration and metric 3D reconstruction from uncalibrated image sequences," Ph.D. thesis, Katholieke Universiteit, Leuven, 1999.

[Pollefeys99b] Pollefeys, M., R. Koch, and L. V. Gool. "A simple and efficient rectification method for general motion," *Proceedings of the 7th IEEE Conference on Computer Vision*, 1999.

[Porter84] Porter, T., and T. Duff. "Compositing digital images," *Computer Graphics* 18 (1984): 253–259.

[Ranger07] Ranger, C., R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. "Evaluating mapreduce for multi-core and multiprocessor systems," *Proceedings of the 13th International Symposium on High-Performance Computer Architecture* (pp. 13–24), 2007.

[Reeb46] Reeb, G. "Sur les points singuliers d'une forme de Pfaff completement integrable ou d'une fonction numerique," *Comptes Rendus de l'Academie des Sciences de Paris* 222 (1946): 847–849.

[Riedmiller93] Riedmiller, Martin, and Heinrich Braun. "A direct adaptive method for faster backpropagation learning: The RPROP algorithm." 1*993 IEEE International Conference on Neural Networks*.. IEEE, 1993.

[Rodgers88] Rodgers, J. L., and W. A. Nicewander. "Thirteen ways to look at the correlation coefficient," *American Statistician* 42 (1988): 59–66.

[Rosenfeld73] Rosenfeld, A., and E. Johnston. "Angle detection on digital curves," *IEEE Transactions on Computers* 22 (1973): 875–878.

[Rosenfeld80] Rosenfeld, A. "Some Uses of Pyramids in Image Processing and Segmentation," *Proceedings of the DARPA Imaging Understanding Workshop* (pp. 112–120), 1980.

[Rosten06] Rosten, Edward. "FAST corner detection." *Engineering Department, Machine Intelligence Laboratory, University of Cambridge*, 2006.

[Rother04] Rother, Carsten, Vladimir Kolmogorov, and Andrew Blake. "Grabcut: Interactive foreground extraction using iterated graph cuts." *ACM transactions on graphics (TOG)*. Vol. 23. No. 3. ACM, 2004.

[Rousseeuw84] Rousseeuw, P. J. "Least median of squares regression," *Journal of the American Statistical Association*, 79 (1984): 871–880.

[Rousseeuw87] Rousseeuw, P. J., and A. M. Leroy. *Robust Regression and Outlier Detection*. New York: Wiley, 1987.

[Rublee11] Rublee, E., V. Rabaud, K. Konolige, and G. Bradski. "ORB an efficient alternative to SIFT or SURF." *Proceedings of the 2011 IEEE International Conference on Computer Vision*, 2011.

[Rubner00] Rubner, Y., C. Tomasi, and L. J. Guibas. "The earth mover's distance as a metric for image retrieval," *International Journal of Computer Vision* 40 (2000): 99–121.

[Rumelhart88] Rumelhart, D. E., G. E. Hinton, and R. J. Williams. "Learning internal representations by error propagation," in D. E. Rumelhart, J. L. McClelland, and PDP Research Group (Eds.), *Parallel Distributed Processing. Explorations in the Microstructures of Cognition* (vol. 1, pp. 318–362), Cambridge, MA: MIT Press, 1988.

[Russ02] Russ, J. C. *The Image Processing Handbook*, 4th ed. Boca Raton, FL: CRC Press, 2002.

[Sánchez13] Sánchez, Javier, Enric Meinhardt-Llopis, and Gabriele Facciolo. "TV-L1 optical flow estimation." *Image Processing On Line 2013* (2013): 137-150.

[Scharr00] Scharr, Hanno. "Optimal operators in digital image processing." Diss. 2000.

[Schiele96] Schiele, B., and J. L. Crowley. "Object recognition using multidimensional receptive field histograms," *European Conference on Computer Vision* (vol. I, pp. 610–619), April 1996.

[Schmidt66] Schmidt, S. "Applications of state-space methods to navigation problems," in C. Leondes (Ed.), *Advances in Control Systems* (vol. 3, pp. 293–340), New York: Academic Press, 1966.

[Schölkopf00] Schölkopf, Bernhard, et al. "New support vector algorithms." *Neural computation* 12.5 (2000): 1207-1245.

[Schwartz80] Schwartz, E. L. "Computational anatomy and functional architecture of the striate cortex: A spatial mapping approach to perceptual coding," *Vision Research* 20 (1980): 645–669.

[Schwarz78] Schwarz, A. A., and J. M. Soha. "Multidimensional histogram normalization contrast enhancement," *Proceedings of the Canadian Symposium on Remote Sensing* (pp. 86–93), 1978.

[Semple79] Semple, J., and G. Kneebone. *Algebraic Projective Geometry*. Oxford, UK: Oxford University Press, 1979.

[Serra83] Serra, J. Image *Analysis and Mathematical Morphology*. New York: Academic Press, 1983.

[Sezgin04] Sezgin, M., and B. Sankur. "Survey over image thresholding techniques and quantitative performance evaluation," *Journal of Electronic Imaging* 13 (2004): 146–165.

[Shannon49] Shannon, C. E.. "Communication in the presence of noise," Proc. Institute of Radio Engineers, vol. 37, no. 1, pp. 10–21, Jan. 1949. Reprint as classic paper in: Proc. IEEE, vol. 86, no. 2, (Feb. 1998), *http://www.stanford.edu/class/ee104/shannonpaper.pdf.*

[Shapiro02] Shapiro, L. G., and G. C. Stockman. *Computer Vision*. Englewood Cliffs, NJ: Prentice-Hall, 2002.

[Shaw04] Shaw, J. R. "QuickFill: An efficient flood fill algorithm," *http://www.codeproject.com/gdi/QuickFill.asp*.

[Shi94] Shi, J., and C. Tomasi. "Good features to track," *9th IEEE Conference on Computer Vision and Pattern Recognition*, June 1994.

[Smith79] Smith, A. R. "Painting tutorial notes," Computer Graphics Laboratory, New York Institute of Technology, Islip, NY, 1979.

[Sobel73] Sobel, I., and G. Feldman. "A 3 x 3 Isotropic Gradient Operator for Image Processing," in R. Duda and P. Hart (Eds.), *Pattern Classification and Scene Analysis* (pp. 271–272), New York: Wiley, 1973.

[Sochman05] Sochman, J., and J. Matas. "WaldBoost - learning for time constrained sequential detection," in *Computer Vision and Pattern Recognition*, 2005. CVPR 2005.

[Steinhaus56] Steinhaus, H. "Sur la division des corp materiels en parties," *Bulletin of the Polish Academy of Sciences and Mathematics* 4 (1956): 801–804.

[Sturm99] Sturm, P. F., and S. J. Maybank. "On plane-based camera calibration: A general algorithm, singularities, applications," *IEEE Conference on Computer Vision and Pattern Recognition*, 1999.

[Suzuki85] Suzuki, S., and K. Abe, "Topological structural analysis of digital binary images by border following," *Computer Vision, Graphics and Image Processing* 30 (1985): 32–46.

[Swain91] Swain, M. J., and D. H. Ballard. "Color indexing," *International Journal of Computer Vision* 7 (1991): 11–32.

[Szeliski04] Szeliski, R. "Image Alignment and Stitching: A Tutorial," *http://research.microsoft.com/apps/pubs/default.aspx?id=70092*. October 2004.

[Tao12] Tao, Michael, et al. "SimpleFlow: A Non-iterative, Sublinear Optical Flow Algorithm." *Computer Graphics Forum*. Vol. 31. No. 2pt1. Blackwell Publishing Ltd, 2012.

[Teh89] Teh, C. H., and R. T. Chin. "On the detection of dominant points on digital curves," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11 (1989): 859–872.

[Telea04] Telea, A. "An image inpainting technique based on the fast marching method," *Journal of Graphics Tools* 9 (2004): 25–36.

[Thrun05] Thrun, S., W. Burgard, and D. Fox. *Probabilistic Robotics: Intelligent Robotics and Autonomous Agents,* Cambridge, MA: MIT Press, 2005.

[Thrun06] Thrun, S., M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney. "Stanley, the robot that won the DARPA Grand Challenge," *Journal of Robotic Systems* 23 (2006): 661–692.

[Titchmarsh26] Titchmarsh, E. C. "The zeros of certain integral functions," *Proceedings of the London Mathematical Society* 25 (1926): 283–302.

[Tomasi98] Tomasi, C., and R. Manduchi. "Bilateral filtering for gray and color images," *Sixth International Conference on Computer Vision* (pp. 839–846), New Delhi, 1998.

[Tou77] Tou, J., and R. Gonzales. *Pattern Recognition Principles.* Addison Wesley Publishing (1977), p. 377.

[Toyama99] Toyama, K., J. Krumm, B. Brumitt, and B. Meyers. "Wallflower: Principles and practice of background maintenance," *Proceedings of the 7th IEEE International Conference on Computer Vision* (pp. 255–261), 1999.

[Trucco98] Trucco, E., and A. Verri. *Introductory Techniques for 3-D Computer Vision.* Englewood Cliffs, NJ: Prentice-Hall, 1998.

[Tsai87] Tsai, R. Y. "A versatile camera calibration technique for high accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses," *IEEE Journal of Robotics and Automation* 3 (1987): 323–344.

[Vandevenne04] Vandevenne, Lode. "Lode's computer graphics tutorial, flood fill." 2004.

[Vapnik95] Vapnik, V. *The Nature of Statistical Learning Theory.* New York: Springer-Verlag, 1995.

[Viola01] Viola, Paul, and Michael Jones. "Rapid object detection using a boosted cascade of simple features." *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on.* Vol. 1. IEEE, 2001.

[Viola04] Viola, P., and M. J. Jones. "Robust real-time face detection," *International Journal of Computer Vision* 57 (2004): 137–154.

[Welsh95] Welsh, G., and G. Bishop. "An introduction to the Kalman filter" (Technical Report TR95-041), University of North Carolina, Chapel Hill, NC, 1995.

[Wharton71] Wharton, W., and D. Howorth. *Principles of Television Reception.* London: Pitman, 1971.

[Wu08] Wu, K., O. Ekow and K. Suzuki. Two Strategies to Speed up Connected Component Labeling Algorithms, *http://escholarship.org/uc/item/5pc9s496*, 06-02-2008.

[Xu96] Xu, G., and Z. Zhang. *Epipolar Geometry in Stereo, Motion and Object Recognition.* Dordrecht: Kluwer, 1996.

[Zach07] Zach, Christopher, Thomas Pock, and Horst Bischof. "A duality based approach for realtime TV-L 1 optical flow." *Joint Pattern Recognition Symposium.* Springer Berlin Heidelberg, 2007.

[Zhang96] Zhang, Z. "Parameter estimation techniques: A tutorial with application to conic fitting," *Image and Vision Computing* 15 (1996): 59–76.

[Zhang99] Zhang, R., P.-S. Tsi, J. E. Cryer, and M. Shah. "Shape form shading: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 21 (1999): 690–706.

[Zhang99] Zhang, Z. "Flexible camera calibration by viewing a plane from unknown orientations," *Proceedings of the 7th International Conference on Computer Vision* (pp. 666–673), Corfu, September 1999.

[Zhang00] Zhang, Z. "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000): 1330–1334.

[Zhang04] Zhang, H. "The optimality of naive Bayes," *Proceedings of the 17th International FLAIRS Conference*, 2004.

[Zivkovic04] Zivkovic, Z. "Improved adaptive Gaussian mixture model for background subtraction," *International Conference Pattern Recognition*, UK, August 2004.

[Zivkovic06] Zivkovic, Z., and F. van der Heijden. "Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Subtraction," *Pattern Recognition Letters*, vol. 27, no. 7, pages 773–780, 2006.

O'REILLY®

学习OpenCV 3 (中文版)

OpenCV库包含500多个函数，可以扩展应用到视觉行业中的很多领域，有广泛的用途，比如安保，医学成像，模式与人脸识别，机器人和工业产品检测，等等。《学习OpenCV 3（中文版）》实用性强，内容全面，讲解透彻，可以帮助读者掌握如何构建具有一定AI（人工智能）的应用程序，使计 算机能够“看见”并根据所得到的数据来做出决策。本书全面介绍整个 OpenCV库，所有示例代码都用C++实现，同时还介绍了可以用于计算机 视觉的机器学习工具。本书每一章都精心设计有动手练习，旨在方便课 堂教学和自学，进一步帮助读者学以致用。

通过本书的学习，读者可以奠定扎实的基础，运用计算机视觉相关知识和OpenCV库来构建简单的或者精巧复杂的应用程序。本书适合开发人员、研究人员、机器人领域专业人员以及爱好者阅读和参考。

- **学习OpenCV数据类型、数组类型和数组的操作**
- **用HighGUI捕获与存储静态与动态图像**
- **对图像进行转换处理，比如放大、缩小、弯曲、重绘以及修复**
- **探究模式识别，包括人脸检测**
- **通过可视化区域跟踪对象及其移动**
- **从立体视觉来重构3D图像**
- **掌握OpenCV中的基础与高级机器学习技术**

安德里安·凯勒（Adrian Kaehler）
博士，企业家，硅谷深度学习小组创始人。他的工作重心包括机器学习、统计建模、计算机视觉和机器人。他就职于斯坦福大学人工智能实验室，他还是该校斯坦利团队的成员，该团队在美国国家航空与航天局（NASA）主办的机器人挑战赛中胜出，赢得了200万美元的大奖。

加里·布拉德斯基（Gary Bradski）
博士是Arraiy.ai的首席架构师（CTO），他曾经就职于好几个创业公司，担任过斯坦福大学计算机系人工智能实验室的顾问教授。他是OpenCV库的创始人，是享有广泛声誉的演讲人和开源社区的积极参与者。

程序设计/计算机视觉&模式识别

O'Reilly Media, Inc.授权清华大学出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

清华社官方微信号

扫 我 有 惊 喜

ISBN 978-7-302-50418-4

9 787302 504184 >

定价：149.00元